

# JAVA

Programming  
For  
Software Engineers



داهنمای  
برنامه نویسی

Java

برای  
مهندسیین  
نرم افزار

[www.txt.ir](http://www.txt.ir)

هوالحق

# JAVA

بهمن ماه ۱۳۸۵

دانشگاه قم

استاد راهنما : سرکار خانم مهندس داودآبادی

گردآورندگان :

لیلا حاجی اسماعیلی

سحر دلشادفر

مهدیه رافع مطلق

فاطمه ملازاده

راضیه مومن

Java به وسیله Ed ،Chris World ،Patrick Haughton ،Mike Gosling James Shridon Frank در سال ۱۹۹۱ در لابراتوار شرکت sun پایه ریزی شد و پس از ۱۸ ماه اولین نسخه کاری آن ایجاد شد. این زبان در ابتدا Oak (به معنای بلوط) نامیده می شد. در بین پاییز ۱۹۹۱ تا بهار ۱۹۹۶ این زبان از نسخه اولیه تغییر نام داده و به صورت عمومی ارائه شد.

برای اکثر کسانی که تنها نامی از جاوا شنیده اند ( البته در ایران ) و یا در حد خیلی ابتدایی با این زبان کار کرده اند.

جاوا مترادف است با برنامه نویسی شبکه ، اجرای خیلی کند ، خیلی سخت ، غیرقابل فهم ، بدون رابط کاربر و ... ولی آیا واقعا این چنین است ؟

برنامه نویسی شبکه : مطمئنا یکی از بازوهای قدرتمند Java است ولی فقط یکی از صدها نقاط قدرت آن.

اجرای خیلی کند : در مقایسه با برنامه های native compile تقریبا درست است، البته به دلیل ساختار این زبان و مدلهاي امنیتی قوی آن ولی تفاوت آن چندان محسوس نیست.

**غیر قابل فهم :** شاید برای برنامه نویسان روال گرا و ناآشنا با Object Oriented Programming اینگونه باشد.

**خیلی سخت :** اصلا اینگونه نیست. شاید برای افراد راحت طلب درست باشد.

**بدون رابط کاربر :** اگر منظور IDE باشد، بیشتر از انگشتان دست برای این زبان IDE وجود دارد. کافی است کمی در مورد آن تحقیق کرد تا با IDE های آن مانند Eclipse ، NetBeans ، IntelliJ ، JBuilder ، Web sphere ، Text pad و... آشنا شد و اگر منظور از رابط کاربر ساخت GUI است، که حرف کاملاً غلطی است. ساختار رابط کاربری در Java از قوی ترین ساختارهای گرافیکی است.

"Java فردا بهتر از امروز خواهد بود." شاید این یک شعار دهان پرکن به نظر برسد. ولی نگاهی به روند تکامل آن، گویای این مطلب خواهد بود که این جمله چندان هم یک شعار نیست. ولی این بهبودها از کجا آمده است؟ از تغییر در زیربنا و ساختار Java نبوده بلکه قسمت اعظم آن، تغییر در کتابخانه های Java بوده، به مرور زمان، Sun خیلی از توابع کتابخانه ای Java را برای

سازگاری بیشتر تغییر داد. مانند تغییر در مدل گرافیکی یا تغییر مدل رویدادها و بازنویسی قسمتهایی از آن و همچنین اضافه کردن ویژگیهای مهمی مانند چاپ کردن که در ویرایشهای اولیه نبود. نتیجه این تغییرات، Platform برنامه نویسی بسیار مفید با قابلیت های زیاد در نسخه های بعدی بود.

به علت عدم وجود مراجع کامل فارسی، عدم وجود انسان های متخصص و استادان خبره در ایران به اندازه کافی، همچنین به علت گران تمام شدن server های آن و یا عدم وجود پروژه های enterprise در ایران ( که اگر هم چنین پروژه هایی باشند، معمولا به شرکت های خارجی سپرده می شوند ) بر آن شدیم تا منبع جامع فارسی را در مورد پیاده سازی این زبان و ساختارهای آن برای آشنایی دانشجویان مهندسی کامپیوتر ارائه دهیم.

بهمن ماه ۱۳۸۵

دانشگاه قم

عنوان .....	صفحه
تاریخچه جاوا .....	۲
نصب برنامه .....	۱۲
انواع داده ها ، متغیرها .....	۲۳
عملگرها .....	۵۹
عبارات کنترلی .....	۸۴
آشنایی با کلاس ها .....	۱۳۶
وراثت ، Inheritance .....	۱۷۱
بسته ها و رابط ها .....	۲۰۷
انواع داده مرکب ، پیاده سازی انواع ساختمان داده ها .....	۲۳۶
مدیریت حافظه .....	۲۸۶
امکانات ویژه .....	۲۹۰
پیوست .....	۳۱۸
منابع .....	۳۳۱

# تاریخچه Java

عناوین این بخش :

مدل زبان و کاربردهای آن

ایجاد java

Applet ها و Application ها

امنیت

قابلیت حمل (portability)

بایت کد (Byte Code ) معجزه Java

امنیت در Java

لایه های امنیتی Java

زبان و کامپایلر Java

کنترل و بررسی بایت کدها

Verifier

بارکننده کلاس (class loader)

بعضی از محدودیت های اپلت ها

در این فصل با مدل زبان و کاربرد های آن آشنا میشوید.

## تاریخچه Java

در اواخر دهه ۸۰ و اوایل دهه ۹۰ میلادی، زبان ++C که یک زبان شیء گرا بود، جلودار زبان های برنامه نویسی شد. در واقع به نظر می رسید که برنامه نویسان بالاخره زبان موردعلاقه و کارای خود را پیدا کرده اند.

++C زبانی بود که بعلت استفاده از قدرت زبان C و کارایی بالای خود، می توانست برای ایجاد سطح وسیعی از برنامه ها به کار رود.

ولی مانند گذشته، افزایش تقاضاها باعث پیشرفت برنامه ها شد. طی مدت چند سال world wide web و اینترنت به طور گسترده ای پیشرفت کردند. این واقعه باعث انقلابی دیگر در برنامه نویسی شد.

## ایجاد java

هدف اولیه از ایجاد java، یک زبان مستقل از Platform بود که توانایی ایجاد نرم افزارهایی برای استفاده در وسایل مختلف الکترونیکی مانند مایکروپیوها و کنترل کننده های بی سیم را داشته باشد. در واقع وسایلی که از cpu های متفاوتی به عنوان کنترلر استفاده می کنند. مشکل عمده این بود که زبان هایی مانند ++C و C برای ایجاد برنامه هایی در یک وسیله مشخص ایجاد شده اند. می توان برنامه های ++C را بر روی هر نوع cpu کامپایل کرد، ولی این کار مستلزم طراحی یک کامپایلر ++C برای آن CPU است و ایجاد کامپایلر نیز عملی هزینه بر و زمان بر می باشد. بنابراین یک راه حل ساده تر و کم هزینه تر مورد نیاز است. برای رفع این مشکل، Gosling و تیم کاری او، کار بر روی یک زبان قابل حمل و مستقل از Platform را شروع کردند که توانایی ایجاد کدی را داشته باشد که بر روی هر نوع cpu و در محیط های کاری مختلف قابل اجرا باشد. نتیجه کار این تیم، نهایتاً به ایجاد java انجامید. پس می توان گفت هدف اولیه این زبان، حل مشکل اتصال وسایل مختلف خانگی به یکدیگر بود. ولی این هدف به شکست انجامید به این دلیل که هیچ کارخانه سازنده وسایل خانگی مایل به استفاده از این طرح نبود. سپس این زبان دوباره و با این هدف طراحی شد که بر روی تلویزیون های کابلی کار کند. این هدف نیز به علت عدم احساس نیاز کارخانه های سازنده به شکست انجامید.



موفقیت های java را می توان از سال ۱۹۹۴ و با فراگیر شدن world wide web عنوان کرد. هنگامیکه Sun متوجه شد Java یک زبان ایده آل برای وب است. به این دلیل که Java توانایی ایجاد برنامه هایی را داشت که بر روی هر سیستمی و با هر سیستم عاملی قابلیت اجرا داشتند و در Web نیز کاربران مختلف با سیستم عامل های مختلف حضور دارند. در واقع این نکته برای تیم طراحی Java آشکار شد که مشکلی که این تیم در ایجاد یک کد در کنترلرهای وسایل مختلف با آن روبرو است، همان مشکلی است که در ایجاد کد برای اینترنت نیز وجود دارد و از همین راه حل می توان آن را نیز حل کرد. از این پس طراحان Java توجه خود را از وسایل الکترونیکی به برنامه نویسی اینترنت معطوف کردند. Java به طور گسترده از سال ۱۹۹۶ ارائه شد و از همان ابتدا به عنوان یک زبان مفید شناخته شد. نه به دلیل تبلیغ های پیرامون آن (!) بلکه به دلیل نیاز به زبان که دارای ساختارهای اینچنین بود. ولی آیا باید موفقیت های Java را تنها به دلیل توسعه وب دانست؟ مطمئناً چنین نیست. هر چند وب نقشی عمده و کلیدی در توسعه Java داشت ولی Java را باید زبانی موفق برای تولید برنامه های کاربردی و در وسایل الکترونیکی نیز به شمار آورد. در واقع می توان گفت اگر وب نبود، Java یک زبان برنامه نویسی مفید ولی ناشناخته و برای وسایل الکترونیکی باقی می ماند.

Java بیشتر خصوصیات خود را از C و ++C گرفته است. طراحان این زبان می دانستند که استفاده از شکل دستوری c و خاصیت شیئی گرایی ++c باعث توجه برنامه نویسان پیشرفته و با تجربه C++/C می شود. علاوه بر این تشابه ظاهری، Java بعضی از خواص این دو زبان را که باعث قدرتمند شدن آن دو شده نیز استفاده کرده است.

به علت شباهت های بین Java و ++C ممکن است Java را نسخه اینترنتی ++C تصور کنیم ولی این تصور اشتباه است. Java تفاوت هایی با ++C دارد. درست است که Java متأثر از ++C است. ولی نباید آن را یک نسخه دیگر از ++c دانست. برای مثال Java هیچ گونه سازگاری با ++C ندارد. ولی اگر شما یک برنامه نویس ++c باشید، خود را بیگانه با Java احساس نمی کنید (به علت شباهت های ظاهری این دو) همچنین java به عنوان یک جایگزین برای ++C طراحی نشده است. Java برای حل یکسری مشکلات خاص و ++C نیز برای حل یکسری مشکلات دیگر طراحی شده است. هر دو این زبان ها، سال هاست که در کنار هم استفاده می شوند. به طور کلی زبان های برنامه نویسی به دو دلیل توسعه می یابند :

۱- برای سازگاری با تغییرات بوجود آمده است.

۲- برای پیشرفته کردن هنر برنامه نویسی (Art of Programming). نیاز به برنامه های مستقل از Platform در اینترنت (سازگاری با تغییرات بوجود آمده) یک دلیل عمده در پیشرفت Java محسوب می شود. همچنین Java شامل تغییراتی می شود که در روش برنامه نویسی ایجاد شده است: به عنوان مثال شئی گرایی در Java به غایت خود رسیده است.

## Applet ها و Application ها

برنامه های ایجاد شده با Java بر دو نوع هستند : Applet ها و برنامه های کاربردی. برنامه های کاربردی، برنامه هایی هستند که بر روی یک کامپیوتر و تحت نظر سیستم عامل آن اجرا می شوند. برنامه های کاربردی ایجاد شده توسط Java، مانند برنامه های ایجاد شده توسط سایر زبان ها مانند C و C++ هستند. تفاوت عمده Java با دیگر زبان ها در ایجاد اپلت ها است. اپلت برنامه ای است که برای ارسال از طریق اینترنت و اجرا بر روی مرورگر وب میزبان ایجاد می شود. یک اپلت مانند سایر فایل ها در اینترنت (فایل های صوتی، تصویری، عکس ها و...) download شده و اجرا می شود. ولی تفاوت اپلت با این فایل ها را می توان اینگونه برشمرد که اپلت یک برنامه هوشمند است و نه فقط یک عکس یا تصویر ساده. با همه جالب بودن اپلت ها، اگر Sun نمی توانست دو مشکل عمده را در آنها حل کند هیچ کاربرد دیگری نمی توانستند داشته باشند. امنیت و قابل حمل بودن مشکلات عمده در طراحی اپلت ها بودند.

### امنیت :

هر زمان که یک برنامه ساده را از اینترنت نصب می کنیم، در واقع با یک ریسک روبرو هستیم. آیا این برنامه خطری برای سیستم ایجاد می کند ؟ قبل از Java کاربران تمایل زیادی به نصب برنامه ها از اینترنت به علت خطرات ناشی از آن نشان نمی دادند. خطراتی مانند ویروسهای مختلف کامپیوتری که می توانند موجب خطراتی در سیستم شوند. و یا برنامه هایی که اطلاعات با ارزش و خصوصی را از روی سیستم ما به سرقت می برند. ولی اگر از یک مرورگر همسو با Java استفاده می کنید. می توانید با اطمینان اپلت های Java را از اینترنت download کرده و اجرا کنید. Java این امنیت را از طریق محدود کردن اپلت ها به محیط اجرایی Java و سلب

اجازه از یک اپلت برای دسترسی به قسمتهای دیگر سیستم، ایجاد می کند. برای مثال یک اپلت اجازه استفاده از فایل های روی سیستم میزبان را ندارد. این خاصیت Java را می توان یکی از مهمترین نقاط مثبت آن دانست.

### قابلیت حمل (portability)

همان طور که می دانید کامپیوترهای مختلف با سیستم عامل های مختلف ( WinX ، Mac Os ، UNIX ، Solaris و...) و یا حتی سیستم هایی با سیستم عامل های خاص آن ها وجود دارند که می توانند به اینترنت نیز متصل شوند. یک برنامه که بخواهد بر روی تمامی این سیستم ها اجرا شود، نیازمند یک کد قابل حمل و قابل اجرا است. راه حلی که Java برای ایجاد امنیت در برنامه های خود ایجاد کرده است، این مشکل را نیز حل میکند.

### بایت کد ( Byte Code ) معجزه Java

کلید حل دو مشکل عنوان شده بایت کد Java است. در واقع خروجی ایجاد شده توسط کامپایلر Java ، یک کد قابل اجرا (executable) نیست بلکه بایت کد است. بایت کد یکسری دستورات بهینه شده است که توسط سیستم در حال اجرای Java (ماشین مجازی Java یا Java Virtual Machine) قابل اجرا است. JVM یک مفسر برای بایت کد به حساب می آید. پس ما در واقع ابتدا برنامه را توسط کامپایلر Java کامپایل کرده و سپس خروجی این کامپایلر را که یک فایل بایت کد (class) است، توسط JVM اجرا می کنیم. همان طور که می دانید، خروجی کامپایلر یک کد قابل اجرا است. در واقع اکثر زبان های امروزی، از کامپایلر به تنهایی استفاده می کنند و نه از مفسر و دلیل آن را می توان کارایی بهتر کامپایلرها برشمرد. ولی استفاده از JVM باعث می شود که بتوان به دو هدف عمده Java دست یافت. به این دلیل که با ایجاد بایت کد ، می توان برنامه کد شده را بر روی هر سیستمی و در هر نقطه از دنیا به وسیله JVM اجرا کرد. تنها نیاز به نصب JVM بر روی سیستم عامل خود داریم. لطفاً به این نکته توجه کنید که با وجود آنکه جزئیات JVM از یک سیستم عامل به یک سیستم عامل دیگر فرق می کند، ولی همه آن ها یک بایت کد یکسان را ایجاد می کنند. اگر لازم بود که برنامه Java ما به کد محلی سیستم کامپایلر

شود، یک برنامه می بایست برای cpu های مختلف و سیستم عامل های متفاوتی که به اینترنت متصل هستند، جداگانه کامپایل می شد که این راه حل خوبی نیست. در عوض می توان با اجرای بایت کد یکسان در سیستم های متفاوت، به یک راه حل خوب برای قابل حمل بودن دست یافت. همچنین می توان ادعا کرد که امنیت نیز در حد بسیار خوبی است. زیرا اجرای هر برنامه Java، زیر نظر و کنترل JVM است و می تواند جلوی دسترسی غیر مجاز برنامه به اطلاعات و منابع سیستم را بگیرد.

کد ایجاد شده توسط کامپایلر Java (بایت کد) را می توان یک کد ماشین برای ماشین مجازی Java دانست. بایت کد باعث تحقق شعار "run any where ,Write Once" شد. به لطف JVM و کامپایلر Java می توانیم برنامه Java را بر روی هر سیستمی نوشته و توسط کامپایلر Java آن را کامپایل کنیم. و خروجی آن (بایت کد) را بر روی هر سیستمی و توسط JVM اجرا کنیم.

## امنیت در Java

یکی از مهمترین دلایلی که زبان Java در اینترنت گسترش پیدا کرد، امنیت بالای آن است. به مقوله امنیت می توانیم از چند دیدگاه نگاه کنیم. اولین مورد حفظ اطلاعات و اسرار خصوصی ما می باشد. مانند کلمه عبور پست الکترونیکی ما یا اسناد خصوصی مانند دفترچه یادداشت های ما اگر یک برنامه بدون اجازه ما این اطلاعات را حتی به دلایل غیر مغرضانه جمع آوری کند، ما نمی توانیم به امنیت سیستم خود اعتماد کنیم. از دیدگاه دیگر می توانیم به برنامه های معروف خراب کاری (مانند ویروس ها و کرم ها) اشاره کنیم که با به هم ریختن اطلاعات پایه ای سیستم، ما را دچار مشکلاتی می کنند. مطمئناً تمامی ما با این ویروس ها سرو کار داشته و خواهیم داشت و اگر به نحوی یک ویروس وارد سیستم شود سعی در پیدا کردن مسیر ورود آن داشته و یا برنامه ای را که به آن اجازه ورود داده مسدود می کنیم. همچنین ممکن است یک برنامه به طور نا خواسته باعث خرابی در فایل های سیستم ما شود. مطمئناً از این لحظه به بعد ما به آن برنامه و شرکت سازنده آن اعتماد نخواهیم کرد. ذکر این نکته نیز لازم است که هر چقدر ما از یک سیستم مطمئن استفاده کنیم چنانچه خود ما اصول امنیتی را رعایت نکنیم سیستم ما امن نخواهد بود. کاربرانی که به هر فایل اجرایی که در اینترنت رسیده آن را اجرا می کنند و یا نامه های

الکترونیکی دارای پیوست اجرایی را باز می کنند راهی برای نفوذ به سیستم خود می گشایند که هیچ سیستم امنیتی نمی تواند جلوی آن را بگیرد.

## لایه های امنیتی Java

سیستم امنیتی Java از چهار قسمت تشکیل شده است :

- ۱- زبان Java به گونه ای طراحی شده است که مطمئن باشد و کامپایلر آن تضمین می کند که کدی که ایجاد می کند، قوانینی امنیتی آن را زیر پا نمی گذارد.
- ۲- بایت کد هایی که اجرا می شوند تحت نظارت و کنترل هستند که قوانین امنیتی را زیر پا نگذارند هدف این لایه این است که بایت کد هایی را که توسط کامپایلری خراب یا قلابی ایجاد شده اند را نیز مد نظر قرار دهد.
- ۳- بار کننده کلاس ها (class loader) بررسی می کند که کلاس ها به محدودیت ها دسترسی نداشته باشند.
- ۴- JAPI (Java Application Programming Interface) مخصوص امنیت که می تواند جلوی خراب کاری applet ها در سیستم را بگیرد. این لایه آخر بستگی به امنیتی دارد که از سه لایه دیگر تضمین می شود.

## زبان و کامپایلر Java

زبان C و زبان هایی مانند آن که دارای امکاناتی برای کنترل دسترسی به اشیاء هستند، همچنین دارای روشهایی برای جعل کردن دسترسی به اشیاء (یا قسمتی از اشیاء) هستند که این روش ها معمولاً با استفاده از اشاره گر ها انجام می شود. در این زبان ها دو مشکل امنیتی مهم داریم:

- ۱- هیچ شیئی نمی تواند خود را از دسترسی از خارج محفوظ کند.

- ۲- یک زبان دارای اشاره گرهای قوی، می تواند دارای Bug هایی باشد که ممکن است امنیت را به خطر اندازد. Java این مشکل را با برداشتن کنترل اشاره گر ها از برنامه نویس و محول کردن آن به JVM برطرف می کند. هنوز هم این اشاره گر ها برای دسترسی به اشیاء وجود دارند، ولی کاملاً تحت نظارت JVM بوده و کنترل می شوند. همچنین امکاناتی که برای آرایه ها در Java

داریم، نه تنها کار با آرایه ها را آسان تر می کند، بلکه آنها را کاملاً کنترل کرده و محدوده کار آنها را بررسی و مراقبت می کنند.

### کنترل و بررسی بایت کدها

اگر یک فرد خرابکار کامپایلر Java را تغییر داده و آن را برای مقاصد خود تنظیم کند تکلیف چیست؟ قبل از اجرای هر بایت کد، سیستم در حال اجرای Java (Java Runtime Environment) کاملاً آنها را بررسی کرده و مطمئن می شود که بر اساس قوانین امنیتی هستند. این بررسی می تواند شامل مواردی مثل تغییر اشاره گرها، دسترسی به محدودیت ها، استفاده غیر مجاز از اشیاء (مثل اشیائی که با فایل ها کار می کنند) ، فراخوانی متد ها با آرگومان های غیر واقعی یا مقادیر غیر مجاز یا سربار شدن پشته باشد. عمل بررسی بایت کدها در JRE، به وسیله Verifier انجام می شود.

### Verifier

بایت کدها برای اینکه موارد امنیتی را نقض نکنند، توسط یک قسمت از JRE به نام Verifier بررسی می شوند. بایت کدها توسط این قسمت بررسی شده و تمامی مراحل اجرایی آن مورد بررسی قرار می گیرند تا انواع پارامترها و آرگومان ها و نتایج آنها درست باشد. بنابراین این قسمت را باید نگیان ورودی بایت کدها دانست که تنها اجازه ورود به بایت کدهای سالم را می دهد. Verifier قسمت اصلی برقرار کننده امنیت در Java است و اگر در پیاده سازی آن اشتباه شود، امنیت برنامه را به خطر می اندازد. مادامی که ما مطمئن هستیم JVM خود را از شرکت Sun دریافت کرده ایم، می توانیم به امنیت آن اطمینان کنیم. بنابراین بهتر است هنگام نصب Java، کاملاً بررسی کرده و مطمئن شویم بسته نرم افزاری که به عنوان Virtual Machine نصب می کنیم، محصول شرکت Sun باشد. هنگامی که بایت کدها از این قسمت عبور می کنند، می توانیم مطمئن باشیم که این بایت کد، با تغییر عملوندهای پشته آن را سربار نمی کند، از پارامترها، آرگومان ها و نتایج برگشتی به درستی استفاده می کند، به طور غیر صحیح داده ها را تبدیل نمی کند ( مثلاً از int به اشاره گر) و به فیلدهای اشیاء ما، به صورت غیر مجاز دسترسی

ندارد. بنابراین مفسر ما می تواند از این لحظه به بعد، بدون نگرانی از بایت کد آن را اجرا کند.

### بارکننده کلاس (class loader)

بارکننده کلاس، نوعی دیگر از نگهدارنده امنیتی ما است. وقتی یک فایل بایت کد در سیستم load می شود، از سه محدوده ممکن است وارد شود :

۱- کامپیوتر محلی

۲- شبکه محلی که از firewall می گذرد

۳- اینترنت

توجه کنید که ما می توانیم مسیرهای متفاوت دیگری نیز داشته باشیم. زیرا به عنوان یک برنامه نویس، این اجازه را داریم که بارکننده خود را طراحی کرده و با جلب اعتماد مشتری به آن ارائه کنیم. یا به عنوان کاربر، مسیرهای دیگری را برای JRE یا مرورگر خود تعریف کنیم. بارکننده هیچگاه به یک کلاس که از مسیر نامطمئن تر وارد شده باشد، اجازه نمی دهد که خود را به عنوان کلاسی از مسیر مطمئن جایگزین کند. به عنوان مثال داده های فایل های ورودی / خروجی سیستم، تنها به عنوان کلاس های محلی تعریف شده اند. بنابراین تنها از مسیر کامپیوتر محلی می توانند وارد شوند. بنابراین هیچ کلاسی که از مسیری غیر از کامپیوتر محلی وارد شود، اجازه کار با فایل های سیستم را ندارد. علاوه بر آن کلاس های یک محدوده خاص، تنها به متدهای public کلاس های محدوده دیگر دسترسی دارند. پس کلاس هایی که از محدوده ای غیر از کامپیوتر محلی بار می شوند، حتی نمی توانند متدهای فایل های ورودی/خروجی سیستم را ببینند. همچنین هر Applet که از روی شبکه بارگذاری می شود، در یک بسته (Package) جداگانه قرار می گیرد. بدین معنی که اپلت ها حتی از یکدیگر نیز محافظت می شوند.

### بعضی از محدودیت های اپلت ها

۱- اپلت به هر آدرس دلخواهی در حافظه دسترسی ندارد. برخلاف دیگر محدودیت های اپلت که توسط مرورگر اعمال می شوند، این محدودیت یکی از خواص زبان Java است و توسط verifier بررسی می شود.

۲- اپلت ها در هیچ حالتی به فایل های سیستم محلی که بر روی آن اجرا می شوند دسترسی ندارند. آنها نمی توانند هیچ فایلی از سیستم را بخوانند یا بر روی آن بنویسند. یا حتی هیچ اطلاعاتی در مورد فایل بگیرند. بنابراین نمی توانند بفهمند که آیا فایلی وجود دارد یا خیر یا تاریخ آخرین تغییرات آن را بفهمند.

۳- اپلت ها نمی توانند کتابخانه های محلی را بارگذاری کنند یا فراخوانی متدهای محلی را تعریف کنند.

۴- اپلت ها اجازه ندارند که از `system.getProperty` به گونه ای استفاده کنند که اطلاعات در مورد کاربر یا سیستم او دریافت کنند مانند `username` یا `directory home`. ممکن است از این متد استفاده کنند تنها برای فهمیدن نسخه Java در حال استفاده.

۵- اپلت ها اجازه تعریف مشخصات برای سیستم را ندارند.

۶- از Java ۱,۱ به بعد، اپلت اجازه ایجاد یا تغییر هیچ `Thread` یا `Thread group` را که جزء `Thread Group` خود آن اپلت نباشد، ندارد.

۷- اپلت ها اجازه تعریف هیچ کدام از انواع `Class Loader`، `Security Manager`، `ContentHandlerFactory` یا `SocketImplFactory`

`URLStreamHandlerFactory` را ندارند و باید از آنهایی که موجود است استفاده کنند.

۸- اپلت تنها می تواند یک ارتباط تحت شبکه با سیستمی که از روی آن `download` شده ایجاد کند.

۹- اپلت نمی تواند به پورتهایی پایین تر از ۱۰۲۴ گوش کند.

۱۰- اگر اپلت بخواهد به یک پورت گوش کند، تنها می تواند ارتباطی از سیستمی که از آن `download` شده باشد.



# نصب برنامه

## عناوین این بخش :

نصب برنامه

نوشتن يك برنامه ساده و آماده کردن آن برای اجرا

نحوه کامپایل برنامه اصلی

اجرای فایل اصلی ( Virtual Java Machine )

نحوه اجرای فایل اصلی

خطای گرامر ( Grammatical Error )

اشکال ( Bug )

توضیحات ( Command )

پیغام خطا

سایر کامپایلرها

\* به همراه CD ، نرم افزار مربوطه و چند نمونه کد ضمیمه شده است.

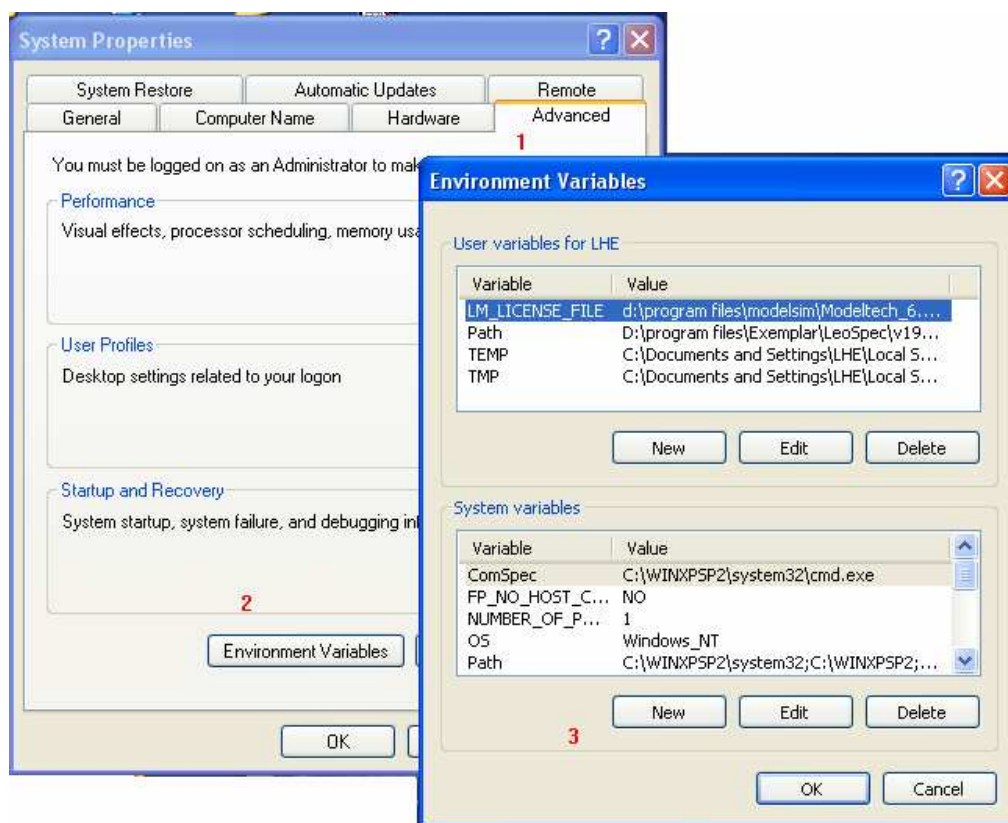
## نصب برنامه

جهت اجرای برنامه های جاوا بسته نرم افزاری j2sdk-1\_3\_1\_01-win که به منظور استفاده بر روی سیستم عامل ویندوز می باشد را مانند سایر نرم افزار ها در مسیر دلخواه نصب می کنیم.

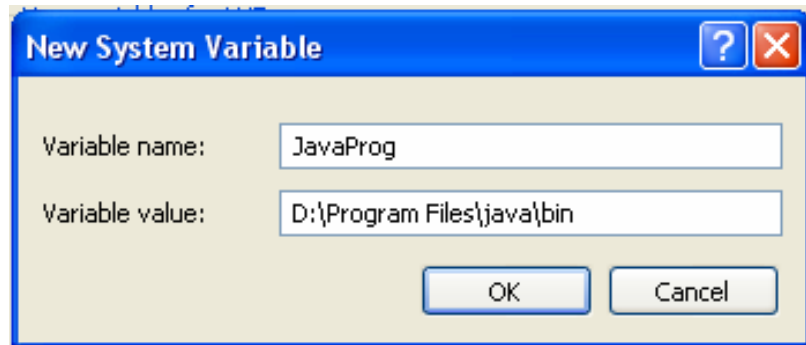


بعد از اتمام نصب برنامه می بایست مسیر شاخه ای که فایل های java و javac در آنها می باشد طبق روال زیر تنظیم شود.

- ۱ – بر روی My Computer کلیک راست کرده و Properties را انتخاب می کنید.
- ۲- به تب Advanced رفته و بر روی Environment Variables کلیک کنید.
- ۳ – در پنجره فوق در بخش System Variables دکمه New را انتخاب کنید.



۴ – در این مرحله می بایست مسیری که فایل های java و javac در آن می باشد تعریف کنیم. این فایل ها در شاخه bin می باشند. مسیر باید به طور کامل طبق شکل زیر بیان شود. برای نام نیز می توانید هر نامی را انتخاب کنید.



بعد از ok کردن ، Variable فوق به لیست اضافه می شود. پنجره ها را ok کنید. در این مرحله نصب برنامه به اتمام رسید.

حال با چگونگی اجرای يك برنامه آشنا می شویم.

قبل از آن می بایست به این نکته توجه داشت که توضیحات زیر مربوط به نسخه ای است که ما آن را نصب کرده ایم. چنانچه شما از محیط جاوای دیگری استفاده می کنید ، قطعاً روش نصب و اجرای برنامه و امکانات محیط برنامه متفاوت خواهد بود. لذا ، اطلاعات بیشتر را در مستندات کامپایلر خود مشاهده نمایید.

**نوشتن يك برنامه ساده و آماده کردن آن برای اجرا**

برای آماده سازی يك برنامه مسیر زیر را طی کنید.

Start > Programs > Accessories > Notepad

برنامه notepad را باز کنید. طبق آنچه آموخته اید و بر اساس برنامه های نمونه ای که توضیح دادیم ، برنامه ای را بنویسید.

در این مرحله ما برنامه ای ساده را انتخاب کرده ایم :

```
class Example {  
  
public static void main ( String args [] ){
```

```

System.out.println("This is First Program with JAVA Language");
}
}

```

بعد از نوشتن برنامه از گزینه File > Save as با پسوند java ذخیره نمایید.

نکته :

۱ – اسم برنامه شما باید هم نام class شما باشد. ( مثلا در اینجا نام برنامه Example خواهد بود. )

۲ – برنامه را بهتر است در همان مسیر فایل های java و javac یعنی فولدر bin ذخیره نمایید.

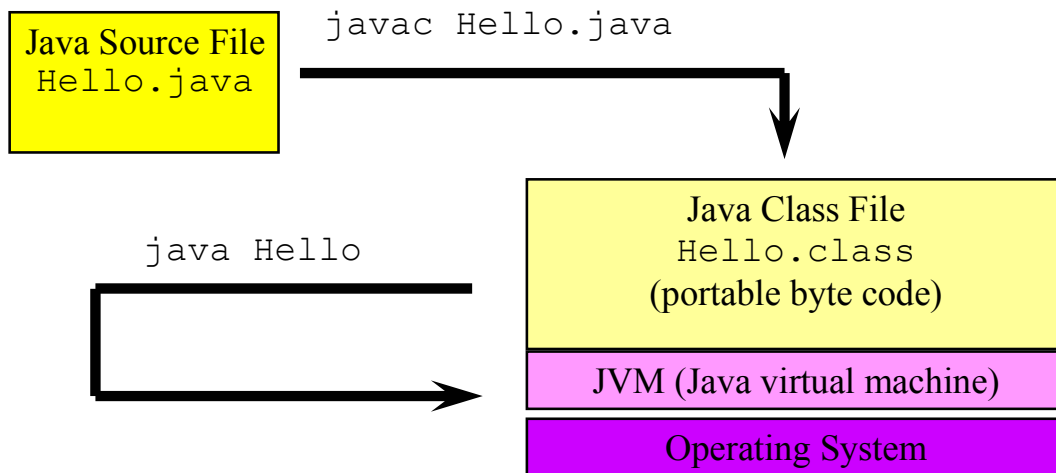
۳ – می توانید نام فایل به همراه پسوند آن را در بین دو علامت " " بنویسید ، مانند "Example.java" ، زین پس این فایل source file ما شناخته میشود.

تا این مرحله شما با استفاده از ویرایشگر Notepad يك فایل text ساخته اید که کدهای جاوا را به صورت حروف یا character در آن ذخیره کرده اید. محتویات این فایل را می توانید بر روی مانیتور مشاهده یا از آن پرینت بگیرید و یا تغییر دهید. کامپیوتر چنین فایلی را نمی تواند مستقیما اجرا کند ، چراکه به صورت بایت ذخیره شده است ، لذا باید آن را به byte code تبدیل کرد. عمل تبدیل برنامه اصلی به بایت کد توسط کامپایلر جاوا می باشد.



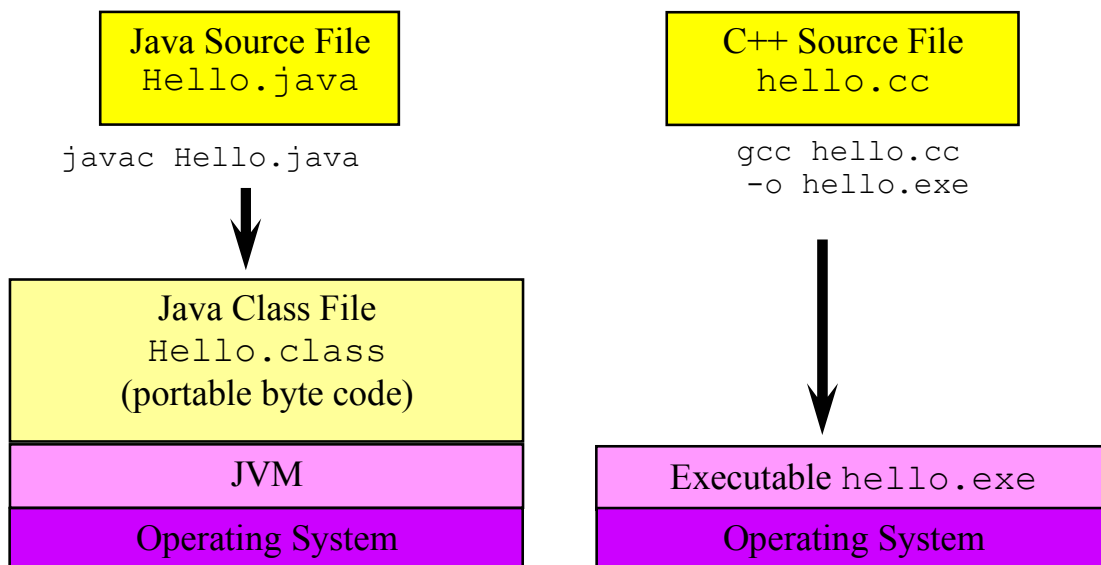
همان طور که ملاحظه کردید حاصل تبدیل برنامه به بایت کد ، فایل با پسوند class. می باشد. مطالب بالا به صورت زیر گویایی بیشتری خواهد داشت :

## Java Computation Model



### Byte Code

vs. Machine Code



## نحوه کامپایل برنامه اصلی

از مسیر زیر Command Prompt را اجرا کنید :

```
Start > run > cmd
```

با استفاده از دستور `DIR *.java` تمامی فایل های جاوای موجود در دایرکتوری فعلی نمایش داده میشود.

نکته :

۱ – برای تغییر درایو به صورت زیر عمل کنید :

نام درایو:

۲ – برای رفتن به دایرکتوری های یک درایو از دستور زیر استفاده کنید :

... \ نام دایرکتوری \ نام دایرکتوری Cd

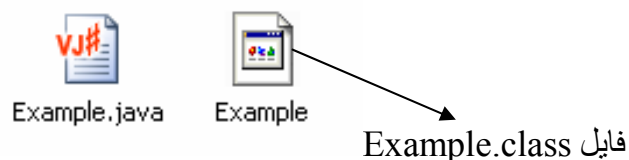
۳ – برای خروج از یک دایرکتوری دستور زیر کاربرد دارد :

```
cd..
```

با دستوراتی که عنوان شد به شاخه `bin` بروید. با استفاده از دستور

```
Javac Exmample.java
```

برنامه را کامپایل کنید. در صورتیکه هیچ گونه خطایی در برنامه نداشته باشید ، بعد از فشردن `enter` ، فایل بایت کد ( `.class` ) ساخته میشود.



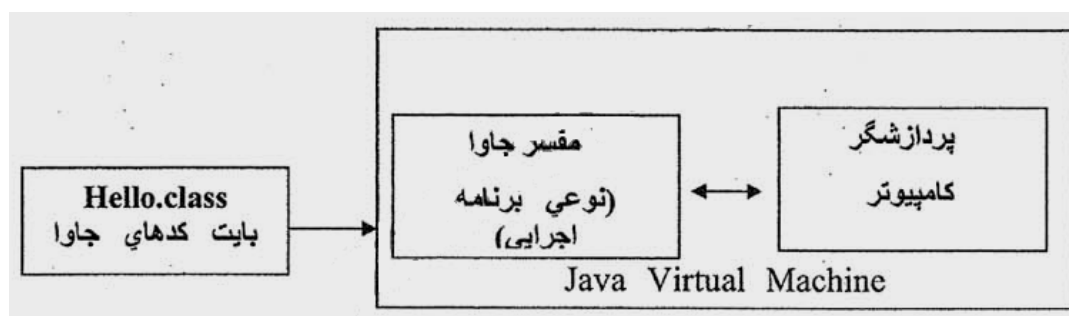
نکته : چنانچه مسیر `path` را درست وارد نکرده و یا برنامه را درست نصب نکرده باشید بعد از وارد کردن دستور `javac ...` یک پیغام خطا ظاهر میشود. بهتر است یک بار دیگر برنامه را نصب کنید.

```
'javac' is not recognized as an internal or external command,  
operable program or batch file.
```

اجرای فایل اصلی ( Virtual Java Machine )

جهت اجراي برنامه هاي جاوا مي توان از سخت افزار مخصوص استفاده كرد. راه ديگر اين است كه از نرم افزارهاي ويژه براي اين كار استفاده شود. در اين صورت نرم افزار ، بابت كدهاي جاوا ( فايل class ) را خوانده ، فرامين آن ها را اجرا مي كند. اين برنامه را اصطلاحاً مفسر interpreted و عمليات آن را interpretation مي نامند. مفسر بابت كدهاي جاوا نوعي برنامه اجرايي است كه در هر سيستم كامپيوتري قابل اجراست. هر يك از شرکتهاي سخت افزاري نوعي مفسر جاوا كه با قطعات و سيستم هاي توليدي آن شركت هم خواني داشته باشد بر روي محصولات سخت افزاري خود نصب مي كنند تا توانايي اجراي برنامه هاي جاوا را داشته باشد. در اين صورت مفسر جاوا در هر سيستم كامپيوتري به همراه پردازشگر ( processor ) كامپيوتر به ماشين مجازي جاوا تبديل مي گردد كه وظيفه اجراي برنامه هاي جاوا را به عهده دارند.

به بيان ديگر پردازشگر كامپيوتر بدون مفسر جاوا قادر به اجراي برنامه هاي جاوا نمي باشد اما به كمك آن ، اين توانايي را پيدا خواهند كرد.



همانطور كه ذكر شد هر سيستم سخت افزاري بدون نرم افزار مفسر جاوا قادر به اجراي برنامه هاي جاوا نيست. سيستم هاي سخت افزاري اگر چه از لحاظ نوع قطعات با يكدیگر فرق دارند اما هنگامي كه يك برنامه جاوا به بابت كد تبديل مي شود ، اين بابت كدها در تمام سخت افزارها يكسان هستند. در واقع حاصل ترجمه يك برنامه جاوا در تمام سخت افزارها بابت كد يكساني است كه در تمام سيستمهاي سخت افزاري بدون هيچ اشكالي قابل اجراست ، حتي اين بابت كدها را از طريق اينترنت به تمام سيستمهاي سخت افزاري مي توان منتقل و اجرا كرد و اين يكي از دلايل محبوبيت جاواست. به بيان ديگر برنامه نويسان جاوا مجبور نيستند برنامه هاي خود را در تمام سيستمهاي سخت افزاري و سيستم هاي عامل مختلف چك كنند و اين يك مزيت بسيار مهم براي هر برنامه نويس محسوب مي شود.

## نحوه اجرای فایل اصلی

برای اجرای برنامه اصلی از دستور java استفاده میشود.

java Example

بعد از اجرای این دستور شما متن خود را مشاهده می کنید و خط فرمان در اختیار شما قرار میگیرد :

```
D:\Program Files\java\bin>javac Example.java
D:\Program Files\java\bin>java Example
This is First Program with JAVA Language
D:\Program Files\java\bin>
```

چند نکته :

نکته ۱ : توجه داشته باشید که جاوا به حروف بزرگ و کوچک حساس می باشد ، حتی در نام فایل ها. بنابراین اگر می نوشتیم java example برنامه اجرا نمی شد. اما آنچه بین " " نوشته میشود چاپ میشود و بزرگ و کوچکی آن مهم نیست.

نکته ۲ : JVM جهت اجرای برنامه به سراغ متد main میرود ، در واقع محل آغاز برنامه متد main می باشد.

نکته ۳ : کلیه دستوراتی که اجرایی هستند با ; در انتهای خط آنها مشخص میشود.

نکته ۴ : سعی کنید برنامه ها را به صورت دندانه دار بنویسید. این امر سبب خوانایی برنامه شما میشود. اما اگر برنامه را به طور خطوط پشت سر هم بنویسید نیز اجرا میشود.

نکته ۵ : قرار دادن خطوط خالی بین سطور و همچنین بین کلمات خالی به کامپایلر و اجرا وارد نمی کند.

## خطای گرامر ( Grammatical Error )

اگر حین تایپ کدهای جاوا اشتباهی صورت گیرد آن را اصطلاحاً خطای گرامری می نامند ، در این موارد کامپایلر با ذکر شماره سطر که خطای دستوری در آن قرار دارد با قرار دادن علامت ^ در زیر کلمه واجد خطا ، شما را به خطای دستوری راهنمایی می کند.



در اینجا لازم به تذکر است که علاوه بر دقیق تایپ کردن کدهای جاوا ، هنگام تایپ دستورات کامپایل و اجرا در محیط Command Prompt دقت کنید چون اشتباه تایپ کردن دستور نیز منجر به ایجاد پیغام خطا می گردد. بنابراین اولین قدم در رفع خطاهای کامپایل صحت دستور کامپایل را بررسی کرده و سپس به سراغ فایل اصلی خود بروید.

### اشکال ( Bug )

اگر برنامه ای بدون خطای دستوری کامپایل و بدون خطا اجرا گردد ولی نتیجه اجرا چیز جز خواست برنامه نویس باشد به آن اشکال می گویند. نتیجتاً عملیات تصحیح اشکال را رفع اشکال ( Debug ) گویند.

### توضیحات ( Comment )

توضیحات جمله یا جملاتی هستند که درباره برنامه یا قسمتی از آن توضیح می دهد. نوشتن توضیحات برای کامپایل و اجرا کردن برنامه الزامی نیست و صرفاً جهت دادن اطلاعات به خوانندگان کد است. در هر جای برنامه که به توضیحات نیاز باشد اگر جمله کوتاه و یک سطر بود توضیحات را پس از // و اگر چند سطر بود پس از /\* و \*/ قرار دهید. توجه : توضیحات اصلاً به بایت کد تبدیل نمی شوند.

### پیغام خطا

در حین کار با Command Prompt ممکن است با خطای مختلفی رو به رو شوید که مهمترین آن به شرح زیر است :

۱ – اگر به جای `javac Example.java` ، `java Example.java` بنویسید خطای زیر صادر میشود.

```
Exception in thread "main" java.lang. NoClassDefenitionFoundError :  
Hello / java
```

- ۲ – اگر به جای javac Example ، javac Example.java بنویسید ، پیغامی ظاهر می شود که نحوه درست نوشتن دستور را توضیح می دهد.
- ۳ – اگر هنگام نوشتن اسم برنامه حروف کوچک و بزرگ را رعایت نکنید با درج ^ در زیر حرف مربوطه شما را متوجه خطا می سازد.
- ( در نکات زیر نام برنامه Hello.java فرض شده است. )

**Hello.java:1:'class' or 'interface' expected**

**Class Hello**

**^**

**1 error**

نکته : در هنگام خطا ، خط دستور را نشان می دهد و با رسیدن به اولین خطا برنامه متوقف می شود.

- ۴ – اگر در هنگام تعریف کلاس اولین { را فراموش کنید پیغامی مشابه زیر صادر میشود.

**Hello.java:1'{' expected**

**class Hello**

**^**

**1 error**

- ۵ – اگر ; را فراموش کنید چنین پیغامی نوشته میشود.

**Hello. Java:5:"," expected**

**System.out.println (" Hello world!")**

**^**

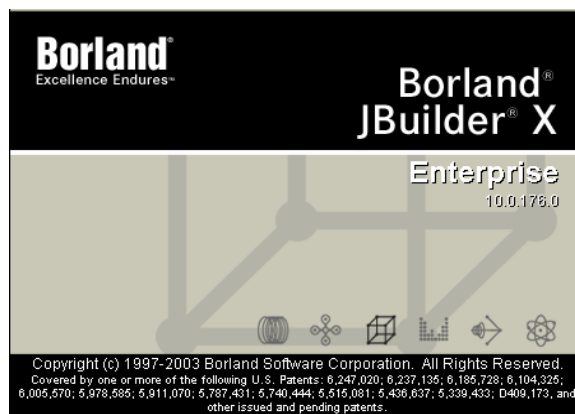
**1 error**

## سایر کامپایلرها

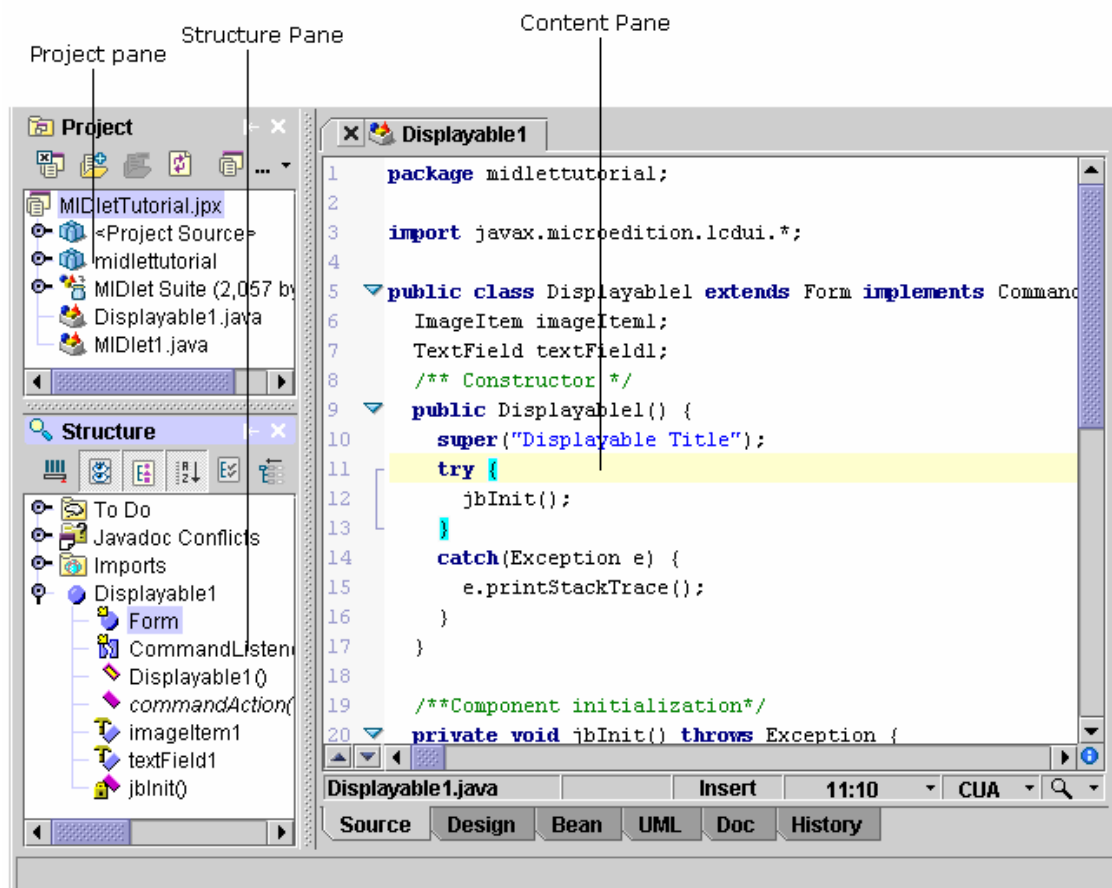
کامپایلرهای جاوا انواع مختلفی دارند ، که نرم افزارهای زیر دارای محیطهای ویژوالی هستند و بالاترین ورژن آن قیمتی حدود ۵-۶ هزار دلار می باشد. ولی ورژنهای پایین تر را می توان از طریق اینترنت دانلود کرد.

## Jbuilder X – ۱

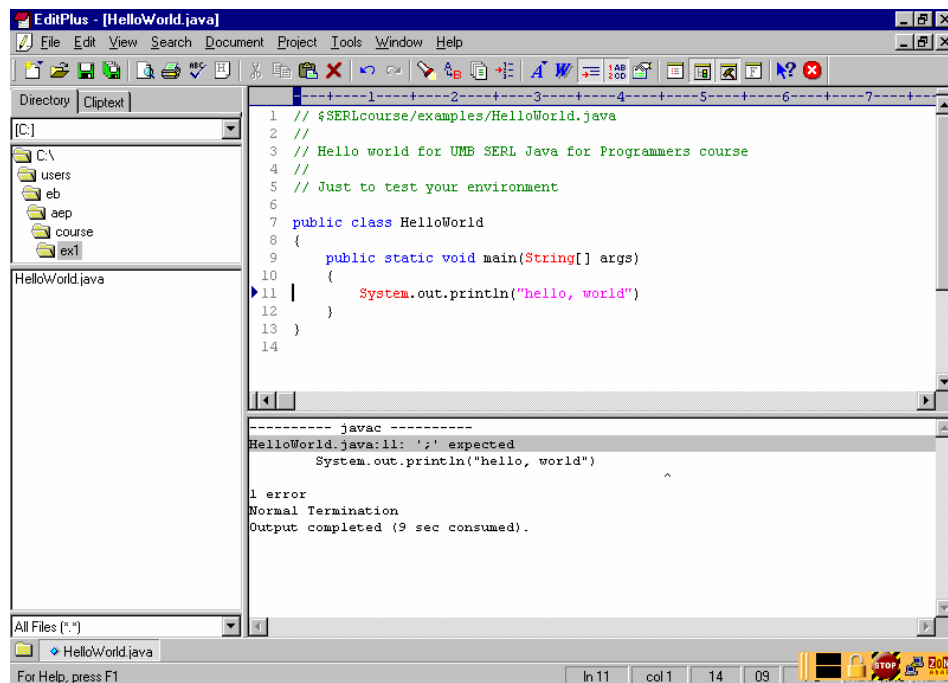
این برنامه ساخت شرکت Borland می باشد.



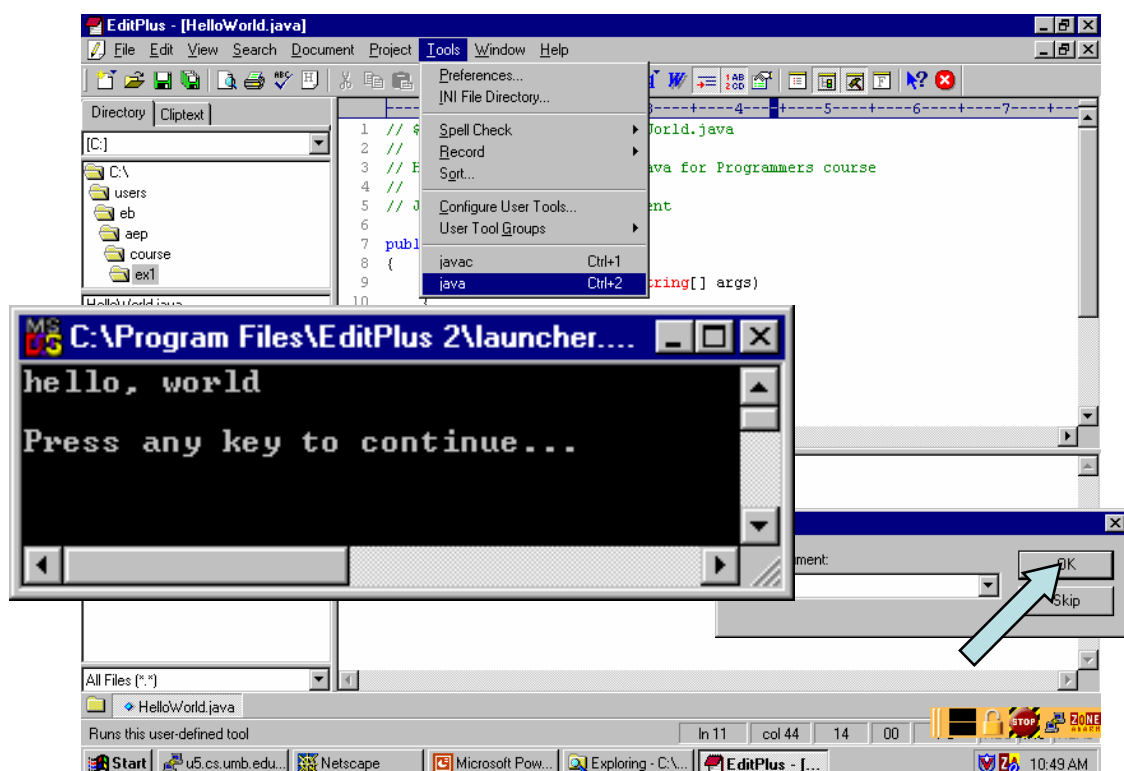
نمایشی از محیط برنامه



## EditPlus – ۲



نمونه ای از اجرای برنامه به صورت تصویر ملاحظه می کنید :



# انواع داده ها ، متغیرها

## عناوین این بخش :

کنترل شدید نوع داده های جاوا

انواع داده های پایه

اعداد صحیح

انواع داده های اعشاری با ممیز شناور

کاراکترها

داده های بولی

نگاهی دقیق تر به لیترال ها

متغیرها

تبدیل و casting

آرایه ها

چند نکته در مورد رشته ها

در متن این فصل به انقیاد ها ، کنترل نوع ، تبدیل نوع ، پیاده سازی سخت افزاری و انواع عملیات اولیه اشاره می شود. همچنین اشاراتی به آرایه ها شده است که در بخش ساختمان داده ها نیز توضیحاتی داده شده است.

## انواع داده ها، متغیرها و آرایه ها در جاوا

این بخش به بررسی سه مورد از عناصر پایه جاوا اختصاص یافته است: انواع داده ها، متغیرها و آرایه ها. جاوا نیز همچون تمام زبان های برنامه سازی مدرن، از انواع داده ها پشتیبانی می کند. از این نوع داده ها می توانید برای تعریف متغیرها و ایجاد آرایه ها استفاده کنید. همان گونه که خواهید دید، نگرش جاوا نسبت به این مورد، شفاف، کارآمد و پیوسته است.

### کنترل شدید نوع داده های جاوا

لازم است در همین ابتدای کار گفته شود که جاوا زبانی است که نوع داده ها به شدت در آن کنترل می شود. در واقع، بخشی از امنیت و استحکام جاوا از این امر ناشی می شود. اینک ببینیم که این کاریعنی چه. نخست اینکه، هر متغیر نوعی دارد، هر عبارت نوعی دارد، و هر یک از انواع داده ها به دقت تعریف شده اند. دوم اینکه، هنگام تخصیص تمام مقادیر، چه به طور صریح و چه از طریق ارسال پارامترها در حین فراخوانی متدها، سازگاری نوع داده ها بررسی می شود. برخلاف برخی از زبان ها، تبدیل خودکار انواع داده های سازگار در جاوا انجام نمی گیرد. کامپایلر جاوا تمام عبارت ها و پارامترها را جهت حصول اطمینان از سازگاری انواع داده ها بررسی می کند. عدم تطبیق نوع داده ها، خطاهایی هستند که می بایست پیش از پایان کامپایل شدن هر کلاس تصحیح شوند.

### انواع داده های پایه

هشت نوع داده پایه در جاوا تعریف شده است: `float`، `char`، `long`، `int`، `short`، `byte`، `Boolean` و `double`، به این نوع داده های پایه، داده های ساده نیز گفته می شود و از هر دو واژه در این کتاب استفاده شده است. این نوع داده ها را می توان به چهار گروه تقسیم نمود:

- اعداد صحیح – این گروه شامل `int` , `short` , `byte` و `long` است که اعداد کامل علامت دار می باشند.
- اعداد اعشاری باممیز شناور- این گروه شامل `float` و `double` است که نمایانگر اعداد اعشاری می باشند.
- کاراکترها- این گروه شامل `char` است که نمایانگر انواع نمادها در مجموعه کاراکترهاست ؛ از قبیل حروف و ارقام.
- بولی- این گروه شامل `boolean` است که نوع ویژه ای برای نمایش مقادیر `true/ false` است.

موارد پیش گفته را می توانید به گونه ای که هستند بکار برید و یا اینکه آرایه ها یا انواع کلاس های خاص خودتان را بسازید. از این رو ، انواع داده های پیش گفته ، پایه و اساس انواع داده های دیگری را که ایجاد می کنید ، تشکیل می دهند.

انواع داده های پایه ، نمایانگر مقادیر واحد هستند و نه شی های مرکب. اگرچه جاوا کاملاً شی گراست ، اما انواع داده های پایه این گونه نیستند. آنها مشابه انواع داده های ساده ای هستند که در بیشتر زبانهای غیر شی گرا یافت می شوند. دلیل این امر، کارایی و بازدهی است. تبدیل انواع داده های پایه به شی ها ، کارایی را بیش اندازه کاهش می داد.

انواع داده های پایه به گونه ای تعریف می شوند تا یک محدوده و رفتار ریاضی معین داشته باشند. زبان هایی چون `C` و `C++` امکان تغییر اندازه اعداد صحیح را بر اساس شرایط محیط اجرا فراهم می آورند.

اما ، جاوا این گونه نیست. به دلیل نیاز به داشتن قابلیت انتقال برنامه های جاوا ، محدوده انواع داده ها ثابت است. به عنوان مثال ، `int` همیشه و بدون توجه به محیط اجرا ، ۳۲ بیتی است. این امر امکان نوشتن برنامه هایی را فراهم می سازد که یقیناً بدون هرگونه انتقال در معماریهای ماشین ها اجرا می شوند. اگرچه تعیین اندازه اعداد صحیح ممکن است موجب کاهش کارایی در برخی از محیطها شود، اما انجام این کار برای رسیدن به قابلیت انتقال ضروری است.

اینک به بررسی هریک از انواع داده ها می پردازیم .

## اعداد صحیح

در زبان جاوا چهار نوع عدد صحیح تعریف شده است : byte ، short ، int و long. تمام این نوع داده ها علامت دار هستند؛ مقادیر مثبت و منفی. جاوا از اعداد صحیح غیر علامت دار «فقط-مثبت» پشتیبانی نمی کند. بسیاری از زبان های کامپیوتری دیگر، هم از اعداد صحیح مثبت و هم منفی پشتیبانی می کنند. اما طراحان جاوا احساس کردند که اعداد صحیح فاقد علامت غیر ضروری هستند. بخصوص اینکه ، آنها تصور می کردند که مفهوم اعداد فاقد علامت عمدتاً برای مشخص شدن رفتار بیت منتهی الیه سمت چپ ، که علامت داده های نوع int را تعیین می کند ، به کار برده می شود.

طول اعداد صحیح ، مقدار فضایی مصرفی آنها در حافظه نیست و در عوض رفتار متغیرها و عبارتهای آن نوع را مشخص می کند. محیط زمان اجرای جاوا می تواند از هر اندازه ای استفاده کند ، مشروط بر این که متغیرها و عبارتها متناسب با نوع داده تعریف شده رفتار نمایند. درحقیقت، حداقل یکی از نگارشها ، byte و short را به صورت مقادیر ۳۲ بیتی (۸ و ۱۶ بیتی) ذخیره می کند تا کارایی افزایش یابد، چرا که اندازه "word" در بیشتر کامپیوترهای مورد استفاده امروزی ، ۳۲ بیت است.

همان گونه که در جدول ذیل نشان داده شده است ، طول و محدوده مقادیر انواع داده های صحیح متغیر است :

نام	طول	محدوده مقادیر
long	۶۴	-9,223,372,036,854,775,808 تا 9,223,372,036,854,775,807
int	۳۲	-2,147,483,648 تا 2,147,483,647
short	۱۶	-32,768 تا 32,767
byte	۸	-128 تا 127

اینک به بررسی هریک از آنها می پردازیم.

## Byte



کوچکترین نوع اعداد صحیح ، byte است. byte ، نوعی عدد صحیح ۸ بیتی علامت دار است که محدوده آن از ۱۲۸- تا ۱۲۷ است. متغیرهای نوع byte بخصوص هنگام کار با جریانی از داده های یک شبکه یا فایل مفید واقع می شوند. آنها همچنین هنگام کار با داده های باینری خاص که ممکن است مستقیماً با سایر انواع داده های توکار جاوا سازگار نباشند ، مفید واقع می شوند. این نوع متغیرها با استفاده از کلمه کلیدی byte تعریف می شوند. به عنوان مثال ، در سطر ذیل متغیر به نام b و c از نوع byte تعریف می شوند :

```
byte b, c ;
```

## Short

short ، نوعی عدد صحیح ۱۶ بیتی علامت دار است. محدوده آن از ۳۲۷۶۸- تا ۳۲۷۶۷ می باشد. این نوع داده ها احتمالاً کمترین استفاده را در جاوا دارند ، چرا که تعریف آنها به گونه است که نخستین بایت سمت راست آنها ، بیشترین ارزش را دارد ( فرمتی به نام big-endian ) این نوع داده ها عمدتاً در کامپیوترهای ۱۶ بیتی که به طور فزاینده ای کمیاب می شوند ، کاربرد دارند. به مثالهایی از تعریف متغیرهای short توجه کنید:

```
short s;  
short t;
```

توجه : "Endianness" واژه ای است که چگونگی ذخیره سازی داده های چند بایتی ، از قبیل short ، int و long ، در حافظه را توصیف می کند. اگر ۲ بایت برای داده های نوع short لازم باشد ، کدامیک در ابتدا قرار می گیرد؛ بایت با ارزش بیشتر یا بایت با ارزش کمتر؟ اینکه گفته می شود کامپیوتری big-endian است ، بدین معناست که ابتدا بایت با ارزش بیشتر ذخیره می شود و سپس بایت با ارزش کمتر قرار می گیرد. کامپیوترهایی چون SPARS و PowerPC از نوع "big-endian" هستند و کامپیوترهای مبتنی بر سری x86 اینتل ، از نوع "little-endian".

## int

متداولترین نوع داده های صحیح ، int است. این نوع داده ها ، ۳۲ بیتی علامت دار بوده و محدوده آنها از ۶۴۸ , ۴۸۳ , ۱۴۷ , ۲- تا ۶۴۷ , ۴۸۳ , ۱۴۷ , ۲ می باشد. متغیرهای نوع int

علاوه بر کاربردهای دیگر، برای کنترل حلقه ها و به عنوان شاخص آرایه ها نیز عموماً به کار برده می شوند. هرگاه با عبارتی متشکل از داده های نوع byte ، short و int و اعداد صحیح لیترال سر و کار داشته باشید ، کل عبارت پیش از انجام محاسبات به int ارتقاء می یابد. int ، متنوعترین و کارآمدترین نوع داده هاست و در بیشتر مواقعی که اعدادی برای شمارش یا شاخص آرایه ها نیاز دارید و یا نیاز به محاسبات اعداد صحیح دارید ، می بایست از آن استفاده نمایید. ممکن است این گونه به نظر رسد که استفاده از short یا byte موجب صرفه جویی در فضای حافظه می شود ، اما هیچ تضمینی وجود ندارد که جاوا آنها را به طور خودکار به int ارتقاء ندهد. به خاطر داشته باشید که «نوع داده» ، تعیین کننده رفتار است و نه اندازه (تنها استثنای موجود ، آرایه ها هستند که چنانچه از نوع byte باشند ، برای هر عنصرشان از یک بایت استفاده می شود و برای آرایه های نوع short و int به ترتیب از ۲ و ۴ بایت به ازای هر عنصر استفاده می شود ).

## Long

Long ، نمایانگر داده های ۶۴ بیتی علامت دار است و برای شرایطی مفید واقع می شود که int برای نگهداری مقدار مورد نظر، به اندازه کافی بزرگ نباشد. محدوده داده های نوع long کاملاً بزرگ است. این موضوع ، long را برای مواقعی که اعداد صحیح بزرگ لازم باشند ، مفید ساخته است. به عنوان مثال ، برنامه ذیل مسافتی که نور در مدت روزهای مشخص شده طی خواهد کرد را محاسبه می کند.

```
// compute distance light travels using long variables.
class light {
    public static void main(String args[]) {
        int lightspeed;
        long days;
        long seconds;
        long distance;
        // approximate speed of light in miles per second
        lightspeed = 186000;
        days = 1000; // specify number of days here
```

```

seconds = days * 24 * 60 * 60; // convert to seconds
distance = lightspeed * seconds; // compute distance
system.out.print("In " + days);
system.out.print("days light will travel about");
system.out.println(distance + " miles.");
}
}

```

خروجي برنامه به شکل ذیل خواهد بود:

In 1000 days light will travel about 16070400000000 miles.

واضح است که نتیجه برنامه فوق را نمی توان در متغیري از نوع int نگهداري کرد.

### انواع داده هاي اعشاري با ممیز شناور

اعداد اعشاري با ممیز شناور، اعداد real نیز نامیده مي شوند و براي ارزیابي عباراتي مفید هستند که نگهداري قسمت اعشاري نیز ضروري باشد. به عنوان مثال ، حاصل محاسباتي چون جذر و محاسبات مثلثاتي چون سینوس و کسینوس ، اعدادي هستند که نگهداريشان مستلزم استفاده از متغیرهاي اعشاري با ممیز شناور است. مجموعه استاندارد (IEEE-754) عملگرها و داده هاي اعشاري با ممیز شناور در جاوا پیاده سازي شده است. float و double دو نوع متداولي هستند که نمایانگر اعداد اعشاري با دقت ساده و مضاعف مي باشند. طول و محدوده آنها در ذیل نشان داده شده است :

نام	طول بر حسب بیت	محدوده تقریبي مقادیر
double	۶۴	$4/9 \times e^{-324}$ تا $1/8 \times e^{308}$
float	۳۲	$1/4 \times e^{-45}$ تا $3/4 \times e^{38}$

این دو نوع در ذیل بررسی شده اند.

### Float

float ، نمایانگر مقادیر اعشاری با دقت ساده است که از ۳۲ بیت برای ذخیره سازی استفاده می کند. «دقت ساده» در برخی از پردازنده ها سریعتر است و به اندازه نصف «دقت مضاعف» به فضای ذخیره سازی نیاز دارد ، اما وقتی که مقادیر بسیار بزرگ یا بسیار کوچک باشند ، نتایج نادرست خواهند شد.

متغیرهای نوع float زمانی مفید واقع می شوند که جز اعشاری لازم باشد و دقت بالا ضرورت نداشته باشد. مثلاً ، float برای نمایش مقادیر ارزی برحسب دلار و سنت مفید خواهد بود. مثالی از تعریف دو متغیر float در ذیل نشان داده شده است:

```
float hightemp, lowtemp;
```

## double

برای «دقت مضاعف» ، که توسط کلمه کلیدی double مشخص می شود ، از ۶۴ بیت برای ذخیره سازی مقادیر استفاده می شود. واقعیت امر آن است که «دقت مضاعف» در برخی از پردازنده های مدرنی که برای محاسبات ریاضی با سرعت بالا بهینه شده اند ، سریعتر است. حاصل توابع ریاضی چون sin()، cos() و sqrt() ، مقدار double است. وقتی نیاز به حفظ دقت محاسبات در محاسبات تکراری داشته باشید و یا هنگام پردازش و مدیریت اعداد بسیار بزرگ ، double بهترین گزینه خواهد بود.

برنامه کوتاه ذیل از متغیرهای double برای محاسبه مساحت يك دایره استفاده می کند :

```
// compute the area of a circle.
class Area {
    public static void main (string args[]) {
        double pi, r, a;
        r = 10.8; // radius of circle
        pi = 3.1416; // pi, approximately
        a = pi * r * r; // compute area
        system.out.println("Area of circle is" + a);
    }
}
```

## کاراکترها

در جاوا براي ذخيره سازي کاراکترها از char استفاده مي شود. اما ، برنامه سازان C/C++ بايد هوشيار باشند:

char در C ، C++ ، عدد صحيحی است که طول آن ۸ بیت است. اما در جاوا این گونه نیست. در عوض جاوا از یونی کد براي نمایش کاراکترها استفاده مي کند. یونی کد ، مجموعه کاراکترهاي کاملاً بين المللي است که مي توانند نمایانگر تمام کاراکترهاي موجود در همه زبانهاي طبیعی باشند. یونی کد در واقع اجتماع مجموعه کاراکترهاي بیشماري از قبیل لاتین، یونانی، عربي، اسلاو، عبري، ژاپني، مجاري وغيره، به شمار مي آید. به همین دلیل به ۱۶ بیت نیاز دارد. از این رو char در جاوا ، ۱۶بیتی است. محدوده از صفر تا ۶۵۵۳۶ است. char منفي وجود ندارد. مجموعه کاراکترهاي استاندارد آسکي هنوز در محدوده صفر تا ۱۲۷ قرار دارد و مجموعه کاراکترهاي ۸ بیتی گسترش یافته (ISO-Latin-1) در محدوده صفر تا ۲۵۵ قرار دارد. از آنجايي که جاوا براي فراهم ساختن امکان نوشتن آپلت جهت استفاده در سرتاسر جهان طراحی شده است ، طبیعی است که از یونی کد براي کاراکترها استفاده نماید. البته ، استفاده از یونی کد براي زبانهاي چون انگلیسي ، آلماني ، اسپانیايي يا فرانسوي که کاراکترهايشان به آساني با ۸ بیت قابل نمایش هستند، قدری ناکارآمد است. اما این گونه موارد، بهاي قابلیت انتقال عمومي وجهاني است. توجه : براي کسب اطلاعات بیشتر درباره یونی کد به <http://www.unicode.org> رجوع کنید. برنامه زیر کاربرد متغیرهاي char را نشان مي دهد :

```
// demonstrate char data type.
class chardemo {
    public static void main (string args[]) {
        char ch1, ch2;
        ch1 = 88; // code for x
        ch2 = ' y';
        system. out.print("ch1 and ch2: ");
        system. out.println(ch1 + " " + ch2);
    }
}
```

خروجي این برنامه به شکل زیر است :

```
ch1 and ch2: x y
```

توجه داشته باشید که مقدار ۸۸ که نمایانگر مقدار آسکی (ویونی کد) متناظر با حرف X است ، به ch1 تخصیص می یابد. همان گونه که گفته شد ، مجموعه کاراکترهای آسکی ، ۱۲۷ مقدار نخست را در مجموعه کاراکترهای یونی کد اشغال می کنند. به همین دلیل تمام «حقه های قدیمی» که در گذشته در خصوص کاراکترها به کار برده اید ، در جاوا نیز قابل استفاده خواهند بود. اگرچه داده های نوع char ، عدد صحیح نیستند ، اما در بسیاری از موارد می توانید به گونه ای با آنها کار کنید که گویی عدد صحیح هستند. این امر به شما امکان می دهد تا دو کاراکتر را با هم «جمع» (ادغام) کنید و یا مقدار یک متغیر کاراکتری را افزایش دهید. به عنوان مثال ، برنامه ذیل را در نظر بگیرید:

```
// char variables behave like integers.
class charDemo2 {
    public static void main (String args[]) {
        char ch1;
        ch1 = ' x';
        System.out.println ("ch1 contains " + ch1);
        ch1++; // increment ch1
        System.out.println("ch1 is now" + ch1);
    }
}
```

خروجی برنامه به شکل زیر است:

```
ch1 contains x
ch1 is now y
```

در برنامه بالا ، ابتدا مقدار X به ch1 تخصیص می یابد. سپس ، ch1 افزایش داده می شود. این کار موجب ذخیره شدن Y در ch1 می شود؛ یعنی کاراکتر پس از X در مجموعه آسکی (یونی کد).

**داده های بولی**

يکي ديگر از انواع داده هاي پايه در جاوا ،boolean نام دارد براي مقادير منطقي است. تنها يکي از دو مقدار false يا true را مي توان به اين نوع متغيرها نسبت داد. حاصل تمام عملگرهاي رابطه اي همين است؛ از جمله  $a < b$ .  
جملات شرطي که بر عبارات کنترلي چون if و for نظارت دارند ، بايد boolean باشند.  
برنامه ذيل کاربرد boolean را نشان مي دهد :

```
// demonstrate Boolean values.
class BoolTest {
    public static void main (string args[]) {
        boolean b;
        b = false;
        system. out.println("b is" + b);
        b = true;
        system. out.println("b is" + b);
        // a Boolean value can control the if statement
        if (b) system. out.println("This is not executed.");
        // outcome of a relational operator is a boolean value
        system. out.println("10>9 is" + (10>9));
    }
}
```

خروجي برنامه به شکل زير است:

```
b is false
b is true
This is executed.
10> 9 is true
```

سه نکته جالب توجه درباره اين برنامه وجود دارد. نخست اينکه ، همان گونه که ملاحظه مي کنيد ، وقتي يك مقدار boolean توسط println() نمايش داده مي شود ، true يا false در خروجي ظاهر مي شود.

دوم اينکه ، مقدار يك متغير بولي به تنهائي براي کنترل عبارت if کفايت مي کند. نيازي به نوشتن عبارت به صورت زير نيست :

```
if (b == true) ...
```

سوم اینکه ، خروجي هر عملگر رابطه اي ، از جمله  $<$  ، يك مقدار بولي است. به همين دليل است كه حاصل عبارت  $10 > 9$ ، true است. به علاوه ، پرانتزهاي اضافي پيرامون  $9 > 10$  ضروري هستند ، چرا كه عملگر "+" نسبت به ">" از اولويت بالاتري برخوردار است.

## ليترال هاي صحيح

احتمالا اعداد صحيح ، متداولترين نوع داده ها در برنامه ها هستند. هر مقدار عددي كامل ، يك ليترال صحيح به شمار مي آيد. ۱، ۲، ۳ و ۴۲ چند مثال از ليترال هاي صحيح هستند. تمام اعداد پيش گفته ، مقادير دسيمال هستند ، يعني هر يك از آنها بيانگر يك عدد در مبناي ۱۰ مي باشد. دو مبناي ديگر نيز در ليترال هاي صحيح قابل استفاده هستند ، اکتال (مبناي ۸) و هگزادسيمال (مبناي ۱۶). در جاوا رقم صفر در ابتداي اعداد اکتال قرار مي گيرد. وجود صفر در سمت چپ اعداد دسيمال معمولي بي معناست. از اين رو ، مقدار به ظاهر درست ۰۹ سبب توليد خطا توسط کامپايلر خواهد شد ، چرا كه ۹ در خارج از محدوده صفر تا ۷ اکتال قرار دارد. يك مبناي متداولتر براي اعداد مورد استفاده برنامه سازان ، هگزادسيمال است كه به خوبي با "word"هاي مضرب ۸ ، مثلاً ۸، ۱۶، ۳۲ و ۶۴ بيت ، مطابقت دارد.

مشخصه اعداد هگزادسيمال ، وجود 0x در ابتداي آنهاست. محدوده ارقام هگزادسيمال از صفر تا ۱۵ است ، بنابر اين A تا F (يا a تا f) جايجزين ۱۰ تا ۱۵ مي شوند.

ليترال هاي موجب ايجاد مقدار int مي شوند كه در جاوا ، مقدار صحيح ۳۲ بيتي هستند. چون نوع مقدار متغيرها به شدت در جاوا كنترل مي شود ، ممكن است از خود بپرسيد كه چگونه ممكن است يك ليترال صحيح را بدون بروز خطاي عدم تطابق ، به ساير داده هاي نوع صحيح ، از قبيل byte يا long ، تخصيص داد. وقتي يك مقدار ليترال در محدوده نوع مقصد باشد ، هيچ خطايي پيش نمي آيد. همچنين ، يك ليترال صحيح را هميشه مي توان به يك متغير long ، تخصيص داد. اما براي استفاده از يك ليترال long ، بايد صراحتا براي كامپايلر مشخص كنيد كه مقدار آن ليترال از نوع long است. اين كار با اضافه كردن حرف L (يا l) به انتهاي (سمت راست) مقدار ليترال انجام مي شود. به عنوان مثال ، 0X7fffffffffffffff يا 9223372036854775807 ، بزرگترين مقدار ليترال نوع long است.



## لیترال های اعشاری

اعداد اعشاری با ممیز شناور، نمایانگر اعداد دسیمال با بخش اعشاری می باشند. این اعداد را می توان به صورت استاندارد یا با نماد علمی نمایش داد. در حالت استاندارد، یک عدد کامل و سپس علامت اعشار و در آخر نیز بخش اعشاری نمایش داده می شود.

به عنوان مثال ، ۰٫۶۶۶۷ ، ۳٫۱۴۱۵۹ ، ۲٫۰ ،

نمایانگر اعداد اعشاری با ممیز شناور به صورت استاندارد هستند. در نماد علمی ، از مقدار استاندارد ، عدد اعشاری با ممیز شناور و پسوندی که نمایانگر توانی از ۱۰ است ، استفاده می شود. توان به صورت E (یا e) و یک عدد دسیمال نمایش داده می شود که یا مثبت است یا منفی. 23E23 ، ۰۶۰۲۲E23 ، 314159-05 و 2e+100 چند نمونه از این اعداد هستند.

در جاوا به طور پیش فرض از «دقت مضاعف» برای لیترال اعشاری با ممیز شناور استفاده می شود. برای مشخص کردن لیترال های float ، باید F یا f را به انتهای آنها بیفزایید. با افزودن D یا d نیز می توانید لیترال های double را به طور صریح مشخص کنید. البته انجام این کار اضافی است. در پیش فرض double از ۶۴ بیت استفاده می شود ، در صورتی که در مقادیر float از ۳۲ بیت استفاده می شود.

## لیترال های بولی

لیترال های بولی بسیار ساده هستند. تنها دو مقدار منطقی بولی وجود دارد: true و false . مقادیر true و false به هیچ گونه نمایش عددی تبدیل نمی شوند. لیترال true در جاوا برابر با "۱" نیست و لیترال false نیز با "۰" برابر نیست. این مقادیر در جاوا تنها به متغیرهای بولی قابل تخصیص می باشند و یا در عبارتهای بولی همراه با عملگرهای بولی قابل استفاده هستند.

## لیترال های کاراکتری

کاراکترها در جاوا ، ایندکس های مجموعه کاراکترهای یونی کد به شمار می آیند. کاراکترها ، مقادیر ۱۶ بیتی هستند که به اعداد صحیح قابل تبدیل بوده و با عملگرهایی چون "+" و "-" قابل

پردازش و مدیریت می باشند. هر لیترال کاراکتری در بین علائم نقل قول نمایش داده می شود. تمام کاراکترهای اسکی قابل رویت را می توان مستقیماً در بین علائم نقل قول نوشت ، مثلاً 'a'، 'z' و '@'. برای کاراکترهایی هم که مستقیماً قابل نوشتن نیستند ، می توان از " escape sequence" استفاده نمود که امکان وارد کردن

کاراکترهای مورد نیاز را فراهم سازد؛ مثلاً \" به جای کاراکتر علامت نقل قول تکی. \n\" به جای کاراکتر نمایانگر سطر جدید. مکانیزمی هم برای وارد کردن مستقیم مقدار یک کاراکتر به صورت اکتال یا هگزادسیمال وجود دارد. برای کاراکترهای اکتال، از '\ و سپس عدد سه رقمی وارد کنید. به عنوان مثال، '141' نمایانگر 'a' است. برای هگزادسیمال نیز 'u'، و سپس چهار رقم هگزادسیمال وارد می شود. به عنوان مثال ، 'u0061'، حرف 'a' در مجموعه کاراکترهای ISO-Latin-1 است ، چرا که بایت نخست • است. 'ua432' هم یک کاراکتر ژاپنی است. فهرست این کاراکترها در جدول 1 نشان داده شده است.

## لیترال های رشته ای

لیترال های رشته ای در جاوا همچون بیشتر زبانهای دیگر مشخص می شوند - با نوشتن یک سری کاراکتر در بین علائم نقل قول جفتی ، چند مثال از لیترال های رشته ای در ذیل نشان داده شده است.

```
"Hello Word"
```

```
" two\nlines"
```

```
"\"This is in quotes\""
```

قراردادهای اکتال\ هگزادسیمال و " escape sequence" هایی که برای لیترال های کاراکتری تعریف شده اند ، به همان صورت برای لیترال های رشته ای نیز قابل استفاده اند. نکته مهمی که باید درباره رشته های جاوا توجه کنید ، آن است که آن است که این رشته باید در یک سطر آغاز و خاتمه یابند. بر خلاف زبانهای دیگر، هیچ گونه کاراکتر ادامه خط در اینجا وجود ندارد.

توجه : همان گونه که می دانید، رشته ها در برخی زبانهای دیگر، از جمله C/C++ ، به صورت آرایه ای از کاراکترها پیاده سازی می شوند. اما در جاوا این گونه نیست. رشته ها در واقع شی

هستند. همان گونه که خواهید دید ، چون جاوا رشته ها را به صورت شی پیاده سازی می کند ، قابلیت های زیادی برای مدیریت رشته ها دارد که هم قدرتمند هستند و هم استفاده از آنها آسان است.

#### کاراکترهای Escape Sequence

Escape Sequence	شرح
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicod character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

#### متغیرها

متغیرها ، واحد پایه ذخیره سازی در برنامه های جاوا هستند. هر متغیر به وسیله ترکیبی از یک شناسه ، نوع و مقدار اولیه (دلخواه) تعریف می شوند. به علاوه ، همه متغیرها دارای محدوده و دوره حیات هستند. محدوده هر متغیر مشخص می کند که آن متغیر در چه قسمتهایی قابل استفاده است. عناصر پیش گفته در ذیل بررسی شده اند.

#### شیوه تعریف کردن متغیرها

همه متغیرها را باید در جاوا تعریف و سپس به کار برد. شکل کلی تعریف کردن متغیرها در ذیل آورده شده است:

```
type identifier [ = value][, identifier [ = value]... ];
```

type ، یکی از انواع داده های تجزیه ناپذیر جاوا ، نام یک کلاس یا رابط است. identifier ، نام متغیر است. متغیرها را می توان از طریق مشخص کردن یک علامت تسلاوی و یک مقدار، مقداردهی نمود. به خاطر داشته باشید که حاصل عبارت مقداردهی باید مقداری از همان نوع مشخص شده برای متغیر باشد.

برای آنکه بیش از یک متغیر از نوع مورد نظر تعریف کنید ، نام آنها را به وسیله کاما از یکدیگر جدا کنید. به چند مثال ذیل از تعریف متغیرهای گوناگون توجه کنید. دقت کنید که برخی از آنها دارای مقدار اولیه هستند.

```
int a, b, c;                // declares three ints, a, b, and c.
int d = 3, e, f = 5;        // declares three more ints, initializing
                             // d and f.
byte z = 22;                // initializes z.
double pi = 3.14159         // declares an approximation of pi.
char x = 'x';               // the variable x has the value 'x'.
```

شناسه هایی که انتخاب می کنید ، چیزی در نامشان ندارند که بیانگر نوعشان باشد. جاوا امکان تخصیص هر یک از انواع داده ها را به گونه شناسه می دهد.

## مقداردهی اولیه پویا

اگرچه در مثالهای قسمت پیش ، از ثابتها به عنوان مقدار اولیه استفاده شده است ، اما جاوا امکان مقداردهی پویای متغیرها را به وسیله عبارتهای معتبر در زمان تعریف آنها نیز فراهم کرده است. به عنوان مثال ، به برنامه کوتاه ذیل توجه کنید که طول وتر یک مثلث قائم الزاویه را با در دست داشتن دو ضلع دیگر محاسبه می کند :

```
// Demonstrate dynamic initialization.
class DynInit {
    public static void main(String args[]) {
```

```

double a = 3.0, b = 4.0;
// c is dynamically initialized
double c = Math.sqrt(a * a + b * b);
system.out.println("Hypotenuse is" + c);
}
}

```

در اینجا سه متغیر محلی (a، b و c) تعریف شده است. دو متغیر نخست، a و b، به وسیله دو ثابت مقداردهی شده اند. اما، c به طور پویا با طول وتر مقداردهی می شود (با استفاده از قضیه فیثاغورث).

در برنامه از یکی دیگر از متدهای توکار جاوا `sqrt()`، استفاده شده است که عضوی از کلاس `Math` می باشد و جذر آرگومان خودش را محاسبه می کند. نکته کلیدی این کار آن است که در عبارت مقداردهی می توان از هر عنصری که در زمان مقداردهی معتبر است استفاده نمود، از جمله متدها، متغیرها یا لیترال ها.

### محدوده دستیابی و دوره حیات متغیرها

تا به حال تمام متغیرهای قابل استفاده، در ابتدای متد `main()` تعریف شده اند. اما، جاوا امکان تعریف متغیرها را در هر بلوکی فراهم می سازد. همان گونه که می دانیم هر بلوک با '{' آغاز و به '}' ختم می شود. هر بلوک، محدوده ای را تعیین می کند. از این رو، هر بار که بلوک جدیدی را آغاز می کنید، محدوده جدیدی ایجاد می شود. هر محدوده تعیین می کند که کدام شی ها برای سایر بخشهای برنامه قابل رؤیت هستند. دوره حیات آن شی ها نیز تعیین می شود.

بسیاری از زبانهای کامپیوتری دیگر، دو دسته عمومی از محدوده ها را تعریف می کنند: عمومی و محلی. اما، این محدوده های قدیمی به خوبی با مدل شی گرای دقیق جاوا مطابقت ندارند. اگرچه امکان ایجاد متغیرهایی وجود دارد که در نهایت به داشتن یک محدوده عمومی منجر می شود، اما انجام این کار عموماً یک استثنا به شمار می آید و نه یک قاعده کلی. در زبان جاوا، دو محدوده اصلی، محدوده هایی هستند که توسط یک کلاس تعریف می شوند و نیز محدوده هایی که توسط یک متد تعریف می شوند. حتی این تمایز نیز قدری مصنوعی است، اما، چون محدوده کلاس، چندین

خصوصیات و ویژگی منحصر به فرد دارد که به محدوده تعریف شده توسط يك متد اعمال نمی شوند ، این تمایز معقول به نظر می رسد.

محدوده ای که توسط هر متد تعریف می شود ، با آکولاد باز آن آغاز می شود. اما اگر آن متد پارامترهایی داشته باشد ، آنها نیز در محدوده آن متد قرار می گیرند.

به طور کلی ، متغیر هایی که در يك محدوده تعریف می شوند ، برای قسمتهایی که در خارج از آن محدوده قرار دارند ، قابل رؤیت (دستیابی) نیستند. از این رو ، وقتی متغیری را درون يك محدوده تعریف می کنید ، در واقع آن را محلی می سازید و آن را در مقابل دستیابی و یا تغییرات غیر مجاز محافظت می کنید. در حقیقت ، قوانین محدوده ، پایه و اساس لازم برای نهان سازی را فراهم می سازند.

محدوده ها ممکن است تودرتو باشند. به عنوان مثال ، هر بار که بلوکی را ایجاد می کنید ، در واقع محدوده جدید تودرتویی را ایجاد می کنید. وقتی چنین اتفاقی می افتد ، محدوده بیرونی ، محدوده درونی را در بر می گیرد. این بدین معناست که شی های تعریف شده در محدوده بیرونی ، برای دستورالعمل های محدوده درونی قابل رؤیت خواهند بود. اما ، عکس این مطلب صادق نیست. شی هایی که در محدوده درونی تعریف می شوند ، برای محدوده بیرونی قابل رؤیت نخواهند بود.

برای درک تأثیر محدوده های تودرتو ، برنامه ذیل را در نظر بگیرید :

```
// Demonstrate block scope.
class scope {
    public static void main(string args[ ]) {
        int x; // known to all code within main
        x = 10;
        if (x == 10) { // start new scope
            int y = 20; // known only to this block
            // x and y both known here.
            system.out.println("x and y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100; // Error! y not known here
        // x is still known here.
        system.out.println("x is" + x);
    }
}
```

}

همان گونه که در توضیحات مشخص است ، متغیر  $x$  در ابتدای محدوده `main()` تعریف شده است و برای کل دستورالعمل های درون `main()` قابل دستیابی است. درون بلوک `if`، متغیر  $y$  تعریف شده است. چون در این بلوک متغیری تعریف شده است ، آن متغیر ( $y$ ) تنها در همان بلوک خودش قابل رؤیت است. به همین دلیل است که سطر  $y=100$  در خارج آن بلوک به « توضیح » مبدل شده است. اگر نماد « توضیح » را از ابتدای آن سطر بردارید، با خطای زمان کامپایل مواجه خواهید شد ، چرا که  $y$  در بیرون محدوده اش قابل رؤیت (دستیابی) نیست. در بلوک `if` می توان از  $x$  استفاده کرد ، چرا که دستورالعمل های درون یک بلوک ( یعنی ، محدوده تودرتو ) ، به متغیرهای تعریف شده در محدوده محیطی خود ، دستیابی دارند.

در هر بلوک می توان متغیرها را در هر نقطه ای تعریف کرد، اما متغیرها تنها پس از تعریف شدن قابل استفاده خواهند بود. از این رو، اگر متغیری را در ابتدای یک متد تعریف کنید، در آن صورت در کل متد قابل رؤیت خواهد بود. اما اگر متغیری را در انتهای بلوکی تعریف کنید ، در آن صورت عملاً بی فایده خواهد بود ، چرا که هیچ دستورالعملی به آن دستیابی نخواهد داشت. به عنوان مثال ، عبارت زیر نادرست است ، چرا که متغیر `count` پیش از تعریف شدن قابل استفاده نیست:

```
// This fragment is wrong!  
count = 100; // oops! cannot use count before it is declared!  
int count;
```

به این نکته نیز باید توجه کنید : متغیرها هنگام ورود به محدوده شان ایجاد می شوند و هنگام خروج از محدوده شان نیز از بین برده می شوند. این بدین معناست که هیچ متغیری مقدارش را پس از خروج از محدوده اش نخواهد داشت. بنابراین متغیرهایی که درون یک متد تعریف می شوند ، مقدارشان را در فواصل زمانی بین فراخوانی متد نخواهند داشت. همچنین ، متغیری که در یک بلوک تعریف می شود ، مقدار

را پس از خروج از بلوک از دست خواهد داد. از این رو، دوره حیات هر متغیر به محدوده اش وابسته است. اگر متغیری در حین تعریف مقدار دهی شود، در آن صورت هر بار به هنگام ورود به آن بلوک ، از نو مقدار دهی می شود. به عنوان مثال ، برنامه بعدی را در نظر بگیرید :

```
// Demonstrate lifetime of a variable.
class LifeTime {
    public static void main(string args[]) {
        int x;
        for (x = 0; x<3; x++) {
            int y = -1; // y is initialized each time block is entered
            system.out.println("y is :" + y); // this always prints -1
            y = 100;
            system.out.println("y is now : " + y);
        }
    }
}
```

خروجي این برنامه در ذیل نشان داده شده است :

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

همان گونه که می بینید ، هر بار به هنگام ورود به حلقه for ، متغیر y با ۱- مقدار دهی می شود. با وجود آنکه سپس مقدار ۱۰۰ به آن تخصیص می یابد ، اما مقدار جدید را از دست می دهد. و آخرین نکته : اگرچه بلوک ها می توانند تودرتو باشند، اما نمی توان متغیری همانم با یکی از متغیرهای محدوده بیرونی تعریف کرد. به عنوان مثال ، برنامه ذیل معتبر نیست :

```
// This program will not compile
class scopeErr {
    public static void main(string args[]) {
        int bar = 1;
        {
            // creates a new scope
            int bar = 2 // compile time error - bar already defined!
        }
    }
}
```

**تبدیل و casting**



اگر تجربه برنامه سازی داشته باشید ، در آن صورت می دانید که تخصیص یک نوع مقدار به متغیری از یک نوع دیگر نسبتاً متداول است. اگر آن دو نوع سازگار باشند ، در آن صورت جاوا عمل تبدیل را به صورت خودکار انجام خواهد داد. به عنوان مثال ، همیشه امکان تخصیص یک مقدار int به یک متغیر long وجود دارد. اما تمام داده های مختلف سازگار نیستند و از این رو ، همه تبدیلات به طور ضمنی ممکن نیست. به عنوان نمونه ، هیچ تبدیلی برای double به byte تعریف نشده است. خوشبختانه ، باز هم امکان کسب رویه های تبدیل برای انواع داده های ناسازگار وجود دارد. برای انجام این کار باید از " casting " استفاده کنید؛ تبدیل صریح بین انواع داده های ناسازگار. اینک به بررسی تبدیل خودکار و casting می پردازیم.

### تبدیلات خودکار جاوا

وقتی یک نوع داده به متغیری از نوع دیگر تخصیص می یابد ، چنانچه شرایط ذیل مهیا باشند ، عمل تبدیل خودکار انجام می شود :

- آن دو نوع سازگار باشند.
  - نوع مقصد بزرگتر از نوع مبدأ باشد.
- هرگاه این دو شرط برقرار باشد ، نوعی «تبدیل همراه با بزرگ سازی» انجام می شود. به عنوان مثال ، نوع int همیشه برای نگهداری مقادیر byte به اندازه کافی بزرگ است ، بنابراین استفاده از عبارت casting به صورت صریح ضرورت ندارد.
- برای «تبدیلات همراه با بزرگ سازی» ، انواع داده های عددی ، از جمله اعداد صحیح و اعشاری با ممیز شناور ، با یکدیگر سازگار هستند. اما ، انواع داده های عددی با char یا boolean سازگار نیستند. همچنین ، char و boolean با یکدیگر سازگار نیستند.
- همان گونه که پیش از این گفته شد ، وقتی یک مقدار ثابت لیترال صحیح در متغیرهای نوع byte ، short یا long ذخیره می شوند ، جاوا باز هم عمل تبدیل خودکار را انجام می دهد.

### انجام casting برای انواع داده های ناسازگار

اگرچه تبدیلات خودکار مفید واقع می شوند ، اما این تبدیلات تمام نیازها را برطرف نمی سازند. به عنوان مثال ، اگر بخواهید یک مقدار `int` را به یک متغیر `byte` تخصیص دهید ، چه می کنید؟ این تبدیل به طور خودکار انجام نمی شود ، چرا که هر `byte` کوچکتر از `int` است. این نوع تبدیلات گاهی اوقات ، « تبدیل همراه با کوچک سازی » نامیده می شود ، چرا که مقدار `int` به طور صریح کوچکتر می شود تا در مقصد قابل ذخیره باشد.

برای آنکه تبدیل بین دو نوع ناسازگار را انجام دهید ، باید از `casting` استفاده کنید، یعنی تبدیل صریح انواع داده ها. شکل کلی انجام کار به صورت زیر است:

```
(target-type) value
```

`target-type` ، مشخص کننده نوع داده ای است که مقدار مورد نظر باید به آن تبدیل شود. به عنوان مثال ، در عبارت زیر، عمل `casting` از `int` به `byte` انجام می شود. چنانچه مقدار صحیح بزرگتر از محدوده `byte` باشد ، مقدارش از طریق تقسیم بر محدوده `byte` و به دست آوردن باقیمانده تقسیم کوچک می شود.

```
int a;  
byte b;  
// ...  
b = (byte) a;
```

هنگام تخصیص یک مقدار اعشاری با ممیز شناور به یک نوع صحیح ، تبدیل به گونه ای دیگر انجام می شود : برش.

همان گونه که می دانید ، اعداد صحیح فاقد قسمت اعشاری هستند. از این رو، وقتی یک مقدار اعشاری با ممیز شناور به یک نوع صحیح تخصیص می یابد، قسمت اعشاری از دست می رود. به عنوان مثال ، اگر مقدار  $1/23$  به یک متغیر صحیح تخصیص یابد ، مقدار حاصل ، ۱ خواهد بود.  $0/23$  برش داده خواهد شد. البته ، اگر اندازه عدد صحیح حاصل بزرگتر از آن باشد که در متغیر مقصد ذخیره شود ، در آن صورت مقدارش از طریق تقسیم به محدوده نوع مقصد کاهش می یابد. برنامه ذیل چند نوع تبدیل را نشان می دهد که نیاز به `casting` دارند :

```
// Demonstrate casts.  
class conversion {  
    public static void main(string args[ ]) {  
        byte b;  
        int i = 257;  
        double d = 323.142;
```

```

        system.out.println("\ nconversion of int to byte.");
        b = (byte) i;
        system.out.println("i and b" + i + " " + b);
        system.out.println("\ nconversion of double to int.");
        i = (int) d;
        system.out.println("d and i" + d + " " + i);
        system.out.println("\ nconversion of double to byte.");
        b = (byte) d;
        system.out.println("d and b"+ d + " " + b);
    }
}

```

خروجي برنامه در زیر نشان داده شده است :

```
conversion of int to byte.
```

```
i and b 257 1
```

```
conversion of double to int.
```

```
d and i 323.142 323
```

```
conversion of double to byte.
```

```
d and b 323.142 67
```

اینک هر يك از تبدیلات را بررسی می کنیم. وقتی عمل "casting" جهت ذخیره مقدار ۲۵۷ در متغیر byte انجام می شود ، نتیجه ، باقیمانده تقسیم ۲۵۷ بر ۲۵۶ (محدوده byte) می شود که در این مثال ، يك است. وقتی d به يك int تبدیل می شود ، قسمت اعشاري آن از دست می رود و مقدارش به باقیمانده تقسیم به ۲۵۶ کاهش می یابد که در این مثال، ۶۷ است.

### ارتقاء خودکار انواع داده ها در عبارتها

علاوه بر عملیات تخصیص ، تبدیل نوع داده ها در شرایط دیگر هم محتمل است : در عبارتها. برای درك دلیل این امر، مثال ذیل را در نظر بگیرید. در برخی از عبارات ، دقت مورد نیاز برای يك مقدار میانجی گاهی اوقات از محدوده يکی از عملوندها تجاوز می کند. به عنوان مثال، عبارت زیر را بررسی کنید :

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

نتیجه جمله میانی  $a*b$  از محدوده عملوندهای نوع `byte` تجاوز می کند. برای مدیریت این گونه مسائل ، جاوا در حین ارزیابی عبارتها ، عملوندهای نوع `byte` یا `short` را به طور خودکار به `int` ارتقاء می دهد. این بدین معناست که جمله میانی  $a*b$  با استفاده از `int` انجام می شود و نه `byte`. از این رو، نتیجه جمله میانی  $(50 * 40 = 2000)$  حتی با وجود اینکه `a` و `b` هر دو از نوع `byte` هستند ، معتبر خواهد بود.

گرچه عمل ارتقاء خودکار مفید است ، اما ممکن است موجب بروز خطای زمان کامپایل نیز بشود. به عنوان نمونه ، مثال به ظاهر درست زیر موجب بروز خطا می شود :

```
byte b = 50;
b = b * 2; // Error! cannot assign an int to a byte!
```

در این مثال نتیجه کاملاً معتبر  $50*2$  قرار است در متغیر `byte` ذخیره شود. اما چون عملوندها هنگام ارزیابی عبارت به طور خودکار به `int` ارتقاء می یابند ، نتیجه نیز به `int` ارتقاء می یابد. از این رو نتیجه عبارت که از نوع `int` است ، بدون استفاده از `casting` در `byte` قابل ذخیره سازی نیست. این امر حتی اگر مقدار در دست تخصیص ، همچون همین مثال ، در متغیر مقصد قابل ذخیره باشد، باز هم صادق است. در مواقعی که از عواقب مسئله سرریز آگاهی دارید ، باید از عمل `casting` صریح استفاده کنید؛ همچون مثال زیر:

```
byte b = 50;
b = (byte) (b * 2);
```

که نتیجه آن مقدار معتبر و درست ۱۰۰ است.

**قوانین ارتقاء انواع داده ها**

جاوا علاوه بر ارتقاء byte ها و short ها به int ، چندین قانون برای ارتقاء دارد که در عبارات رعایت می شوند. این قوانین به شرح ذیل هستند: نخست اینکه ، تمام مقادیر byte و short به گونه ای که گفته شد ، به int ارتقاء می یابند. سپس ، اگر یکی از عملوندها از نوع long باشد ، کل عبارت به long ارتقاء می یابد. اگر یکی از عملوندها از نوع float باشد، در آن صورت کل عبارت به float ارتقاء می یابد و اگر یکی از عملوندها از نوع double باشد، نتیجه هم double خواهد بود.

برنامه زیر نشان می دهد که چگونه هر یک از مقادیر موجود در عبارت ارتقاء داده می شود تا با آرگومان دیگر عملوند مطابقت داشته باشد:

```
class promote {
    public static void main(String args[] ) {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.64f;
        double d = .1234;
        double result = (f * b) + (i / c) - (d * s);
        System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
        System.out.println("result = " + result);
    }
}
```

اینک به بررسی دقیق ارتقا انواع داده ها در این سطر از برنامه می پردازیم :

```
double result = (f * b) + (i / c) - (d * s);
```

در جمله نخست  $b \cdot f$  به float ارتقاء می یابد نتیجه جمله نیز float می شود. سپس در جمله  $i / c$  ، به int ارتقاء می یابد و نتیجه جمله نیز به نوع int تبدیل می شود. سپس در جمله  $d \cdot s$  ، مقدار s به double ارتقاء می یابد و نوع جمله نیز double می شود و بالاخره اینکه این سه مقدار میانجی float ، int و double بررسی می شود. نتیجه float به علاوه یک int و float خواهد بود. سپس نتیجه float منهای آخرین double به double ارتقاء می یابد که نوع نتیجه نهایی عبارت است.

## آرایه ها

هر آرایه، گروهی از متغیرهای هم‌نوع است که ارجاع به آنها از طریق یک نام مشترک صورت می‌گیرد. در جاوا می‌توان آرایه‌هایی از انواع مختلف را به صورت یک بعدی یا چند بعدی ایجاد نمود. هر یک از عناصر هر آرایه از طریق ایندکس خودشان قابل دستیابی هستند. آرایه‌ها روش مناسبی برای گروه بندی اطلاعات مرتبط به هم می‌باشند. توجه: اگر با C/C++ آشنایی دارید، مراقب باشید. عملکرد آرایه‌ها در جاوا با عملکردشان در آن زبانها تفاوت دارد.

### آرایه های تک بعدی

هر آرایه «تک بعدی»، اساساً فهرستی از متغیرهای هم‌نوع است. برای آنکه آرایه‌ای ایجاد کنید، ابتدا باید یک متغیر آرایه‌ای از نوع مورد نظر ایجاد نمایید. فرم کلی تعریف آرایه‌های تک بعدی به صورت زیر است:

```
type var-name[ ];
```

type در اینجا نوع آرایه را مشخص می‌کند. نوع آرایه، نوع داده یکایک عناصر تشکیل دهنده آرایه را مشخص می‌کند. از این رو، نوع آرایه مشخص می‌کند که چه نوع داده‌هایی در آرایه ذخیره خواهند شد. به عنوان مثال، آرایه‌ای به نام month-days از نوع int در زیر تعریف می‌شود:

```
int month-days[ ];
```

اگرچه تعریف بالا موجب تثبیت متغیر آرایه‌ای month-days می‌شود، اما در واقع هنوز هیچ آرایه‌ای وجود ندارد. در حقیقت، آرایه month-days که با null مقداردهی می‌شود، نمایانگر آرایه بدون مقدار است. برای آنکه month-days را با آرایه‌ای فیزیکی از مقادیر صحیح مرتبط سازید، بایست آرایه را با استفاده از new ایجاد و آن را به month-days تخصیص دهید. new، عملگر ویژه‌ای برای تخصیص حافظه است.

فرم کلی new برای آرایه های تک بعدی به صورت زیر است :

```
array-var = new type [size];
```

type مشخص کننده نوع داده ها ، size مشخص کننده تعداد عناصر آرایه و array-var ، متغیر آرایه است که با آرایه مرتبط می شود. یعنی ، برای آنکه از new برای تخصیص آرایه استفاده کنید، باید نوع و تعداد عناصری که باید تخصیص داده شوند را مشخص نمایید. عناصری که به وسیله new به آرایه تخصیص می یابند ، به طور خودکار با صفر مقداردهی می شوند. در این مثال یک آرایه ۱۲ عنصری از اعداد صحیح و با month-days مرتبط می شود.

```
month-days = new int[12];
```

پس از اجرای عبارت بالا، month-days به آرایه ای از ۱۲ عدد صحیح اشاره خواهد داشت. به علاوه، تمام عناصر آرایه با صفر مقداردهی خواهند شد.

اینک مطالب پیش گفته را مرور می کنیم : ایجاد هر آرایه نوعی فرآیند دو مرحله ای است. نخست اینکه ، باید متغیری از آرایه مورد نظر تعریف کنید. دوم اینکه، باید حافظه محل آرایه را با استفاده از new تخصیص دهید و آن را به متغیر آرایه نسبت دهید.

از این رو، تمام آرایه ها در جاوا به صورت پویا تخصیص می یابند. اگر مفهوم تخصیص پویا برایتان نا آشناست، نگران نباشید. این موضوع به تفصیل بررسی خواهد شد.

پس از تخصیص آرایه، با مشخص کردن ایندکس عناصر در بین گروه، به راحتی می توانید به هر یک از آنها دستیابی داشته باشید. ایندکس تمام آرایه ها از صفر آغاز می شود. به عنوان مثال، عبارت زیر مقدار ۲۸ را به دومین عناصر month-days تخصیص می دهد.

```
month-days[1] = 28;
```

سطر زیر هم سبب نمایش مقدار ذخیره شده در ایندکس شماره ۳ (عناصر چهارم) می شود.

```
system.out.println(month-days[3]);
```

تمام مطالب پیش گفته در برنامه زیر به کار برده شده اند. این برنامه، آرایه ای متشکل از تعداد روزهای هر ماه ایجاد می کند.

```
// Demonstrate a one-dimensional array.
```

```
class array {  
    public static void main(string args[ ]) {  
        int month-days[ ];  
        month-days = new int[12];  
        month-days[0] = 31;  
        month-days[1] = 28;
```

```

        month-days[2] = 31;
        month-days[3] = 30;
        month-days[4] = 31;
        month-days[5] = 30;
        month-days[6] = 31;
        month-days[7] = 31;
        month-days[8] = 30;
        month-days[9] = 31;
        month-days[10] = 30;
        month-days[11] = 31;
        system.out.println("April has" + month-days[3] + " days.");
    }
}

```

وقتي اين برنامه را اجرا مي كنيد ، تعداد روزهاي ماه آوريل را نمايش مي دهد. همان گونه كه گفته شد ، ايندكس آرايه هاي جاوا از صفر آغاز مي شود، بنابر اين تعداد روزهاي ماه آوريل ، month-days[3] يا ۳۰ است.

امكان تركيب تعريف متغير آرايه با تخصيص خود آرايه نيز به صورت زير وجود دارد :

```
int month-days[ ] = new int[12];
```

روش بالا ، روشي است كه عموماً در برنامه هاي حرفه اي جاوا خواهيدديد. آرايه ها را مي توان هنگام تعريف كردن، مقداردهي نمود. فرآيند انجام اين كار شباهت زيادي به روند انجام آن براي انواع داده هاي پايه و ساده دارد. براي اين كار، مقادير بين دو آكولاد نوشته شده و كاما از يكدیگر جدا مي شوند. كاماها، مقادير عناصر آرايه را از يكدیگر جدا مي كنند. آرايه به طور خودكار به گونه اي ايجاد مي شود تا فضاي كافي براي عناصري كه براي مقداردهي اوليه مشخص مي كنيد ، وجود داشته باشد. نيازي به استفاده از new نيست. به عنوان مثال ، براي آنكه تعداد روزها را براي هر ماه مشخص كنيد ، مثال زير آرايه اي از نوع اعداد صحيح ايجاد و با تعداد روزها مقداردهي مي كند :

```

// An improved version of the previous program.
class AutoArray {
    public static void main(String args[ ]) {
        int month-days[ ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
        30, 31 };
        system.out.println("April has" + month-days[3] + " days.");
    }
}

```



```

    }
}

```

وقتي برنامه بالا را اجرا مي كنيد ، همان خروجي مثال پيش از آن را خواهيدديد.

جاوا به شدت كنترل مي كند كه تصادفا اقدام به ذخيره مقادير در خارج از محدوده آرايه نكنيد و يا به آنها ارجاع نداشته باشيد. سيستم زمان اجراي جاوا كنترل مي كند تا اطمينان حاصل شود كه تمام ايندكس هاي آرايه در محدوده آن قرار دارند. به عنوان مثال ، سيستم زمان اجراي جاوا مقدار هر يك از ايندكس هاي month-days را كنترل مي كند تا اطمينان حاصل شود كه تمام آنها در بازه بسته صفر و ۱۱ قرار دارند. اگر بخواهيد به عناصر خارج از محدوده آرايه دستيابي داشته باشيد (اعداد منفي يا اعداد بزرگتر از طول آرايه) ، با خطاي زمان مواجه خواهيد شد.

يك مثال ديگر كه از يك آرايه تك بعدي استفاده مي كند. اين برنامه ميانه بين مجموعه اي از اعداد را محاسبه مي كند :

```

// Average an array of values.
class Average {
    public static void main(String args[] ) {
        double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5 };
        double result = 0;
        int i;

        for(i =0; i < 5; i++)
            result = result + nums[i];

        System.out.println("Average is" + result / 5);
    }
}

```

## آرايه هاي چند بعدي

آرايه هاي چند بعدي در جاوا، آرايه اي از آرايه ها هستند. اين آرايه ها همچون آرايه هاي چند بعدي معمولي به نظر مي رسند. اما، همان گونه كه خواهيدديد، چند تفاوت جزئي وجود دارد. براي آنكه متغير آرايه چند بعدي را تعريف كنيد، هر يك از ايندكس ها را بين دو كروشه بنويسيد. به عنوان مثال ، در عبارت زير يك آرايه دو بعدي به نام twoD تعريف شده است.

```
int twoD[ ] [ ] = new int[4] [5]
```

عبارت بالا يك آرايه  $4 \times 5$  را تخصيص داده و آن را به twoD نسبت مي دهد. ماتريس مزبور به صورت آرايه اي از آرايه هاي نوع int پياده سازي مي شود. اين آرايه همچون شكل ۱-۳ خواهد بود.

برنامه زير، هر يك از عناصر اين آرايه را از چپ به راست و از بالا به پايين شماره گذاري نموده و سپس آن مقادير را نمايش مي دهد :

```
// Demonstrate a two-dimensional array.
class TwoDArray {
    public static void main(string args[ ]) {
        int twoD[ ] [ ]= new int[4] [5];
        int i, j, k = 0;
        for(i=0; i < 4; i++)
            for(j=0; j < 5; j++) {
                twoD[i] [j] = k;
                k++;
            }
        for(i=0; i < 4; i++) {
            for(j=0; j < 5; j++)
                system.out.print(twoD[i] [j] + " ");
            system.out.println( );
        }
    }
}
```

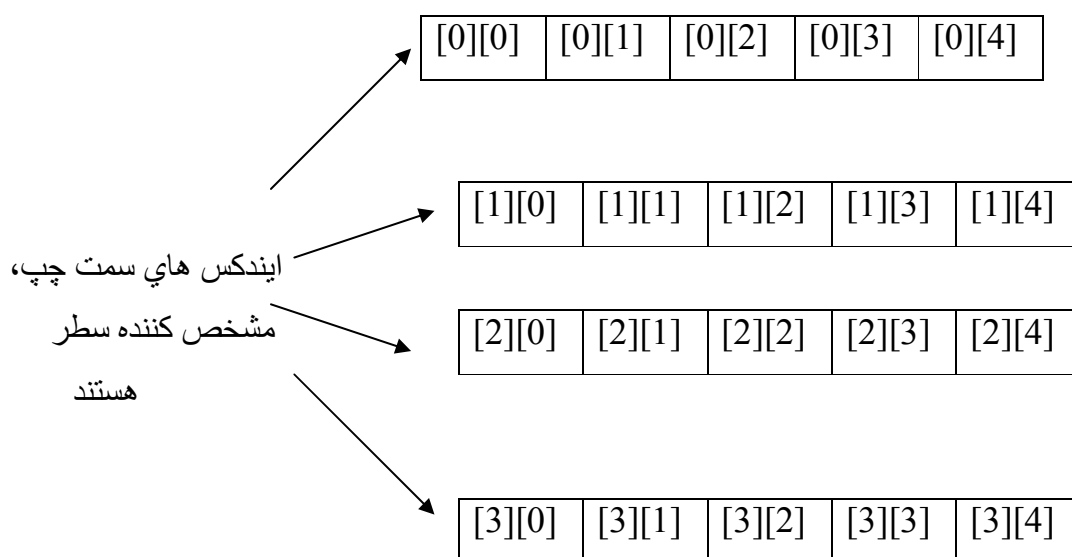
خروجي برنامه به شكل زير است:

```
0  1  2  3  4
5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
```

وقتي حافظه اي را به يك آرايه چند بعدي تخصيص مي دهيد ، كافي است تنها حافظه بعد اول (سمت چپ ترين) را مشخص كنيد. بعدي ديگر را مي توانيد به طور جداگانه تخصيص دهيد. به عنوان مثال ، در عبارت زير ، حافظه بعد اول twoD به هنگام تعريف آرايه تخصيص مي يابد. حافظه بعد دوم به طور دستي تخصيص مي يابد.

```
int twoD[ ] [ ]= new int[4] [ ];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

ایندکس های سمت راست ، مشخص کننده ستون هستند



با فرض اینکه : `int twoD[ ] [ ]= new int[4] [5];`

اگرچه در این مثال تخصیص جداگانه بعد دوم آرایه ها هیچ مزیتی ندارد ، اما انجام این کار در شرایط دیگر ممکن است سودمند باشد. به عنوان مثال ، وقتی ابعاد آرایه ها را به طور دستی تخصیص می دهید ، نیاز به تخصیص همان تعداد عنصر برای هر یک از ابعاد نخواهید داشت. همان گونه که پیش از این گفته شد ، از آنجا که آرایه های چند بعدی در واقع آرایه هایی از آرایه ها هستند ، طول هر آرایه تحت کنترل شماست. به عنوان مثال ، برنامه زیر یک آرایه دو بعدی ایجاد می کند که در آن اندازه های بعد دوم مساوی نیستند.

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
    public static void main(string args[ ]) {
        int twoD[ ] [ ]= new int[4] [ ];
        twoD[0] = new int[1];
        twoD[1] = new int[2];
```

```

twoD[2] = new int[3];
twoD[3] = new int[4];
int i, j, k = 0;
for(i=0; i < 4; i++)
    for(j=0; j < i+1; j++) {
        twoD[i][j] = k;
        k++;
    }
for(i=0; i < 4; i++) {
    for(j=0; j < i+1; j++)
        system.out.println( );
    }
}

```

خروجي برنامه در ذیل نشان داده شده است :

```

0
1 2
3 4 5
6 7 8 9

```

آرایه ای که به وسیله برنامه ایجاد می شود به صورت شکل زیر است.

[0][0]			
[1][0]	[1][1]		
[2][0]	[2][1]	[2][2]	
[3][0]	[3][1]	[3][2]	[3][3]

استفاده از آرایه های چند بعدی غیر معمول (یا نامنظم) ممکن است برای بسیاری از برنامه های کاربردی مناسب نباشد ، چرا که وجود آرایه های چند بعدی در برنامه ها خلاف انتظار کاربران خواهد بود. با این وجود ، آرایه های نامنظم ممکن است در برخی از شرایط به طور کارآمدی مورد استفاده قرار گیرند. به عنوان مثال ، اگر نیاز به آرایه دو بعدی بسیار بزرگی داشته باشید که تنها برخی از عناصر آنها مقدار داشته باشند ( یعنی ، آرایه ای که تمام عناصرش مورد استفاده نباشند ) ، در آن صورت آرایه های نامنظم راه حل کاملی به شمار می آیند.

مقداردهي آرايه هاي چندبعدي هم ميسر است. براي انجام اين كار، كافي است مقادير اوليه هر يك از ابعاد را بين دو آكولاد بنويسيد. در برنامه زير ماتريسي ايجاد مي شود كه مقدار هر عنصر، حاصل ضرب ايندكس هاي سطر و ستون است. همچنين توجه داشته باشيد كه علاوه بر مقادير ليترال، از جملات جبري نيز مي توانيد براي مقداردهي اوليه استفاده كنيد.

```
// Initialize a two-dimensional array.
class Matrix {
    public static void main(string args[ ]) {
        double m[ ] [ ] = {
            { 0*0, 1*0, 2*0, 3*0 },
            { 0*1, 1*1, 2*1, 3*1 },
            { 0*2, 1*2, 2*2, 3*2 },
            { 0*3, 1*3, 2*3, 3*3 }
        };
        int i, j;
        for(i=0; i < 4; i++) {
            for(j=0; j < i+1; j++)
                system.out.print(m[i] [j] + " ");
            system.out.println( );
        }
    }
}
```

وقتي اين برنامه را اجرا مي كنيد، خروجي ذيل را خواهيدديد:

```
0.0  0.0  0.0  0.0
0.0  1.0  2.0  3.0
0.0  2.0  4.0  6.0
0.0  3.0  6.0  9.0
```

همان طور كه مي بينيد، هر سطر از آرايه به گونه اي كه در فهرست مقادير اوليه مشخص شده است، مقدار مي گيرد.

اينك به بررسي چند مثال ديگر از کاربرد آرايه هاي چند بعدي مي پردازيم. در برنامه زير يك آرايه سه بعدي  $3 \times 4 \times 5$  ايجاد مي شود. سپس حاصل ضرب ايندكس هاي هر عنصر در همان عنصر ذخيره مي شود و در نهايت، اين حاصل ضربها نمايش داده مي شود.

```
// Demonstrate a two-dimensional array.
```

```

class ThreeDMatrix {
    public static void main(String args[]) {
        int threeD[ ][ ][ ] = new int[3][4][5];
        int i, j, k;
        for(i=0; i <3; i++)
            for(j=0; j <4; j++)
                for(k=0; k < 5; k++)
                    threeD[i][j][k] = i * j * k;
        for(i=0; i <3; i++) {
            for(j=0; j <4; j++) {
                for(k=0; k < 5; k++)
                    System.out.print(threeD[i][j][k] + " ");
                System.out.println( );
            }
        }
    }
}

```

خروجي برنامه به شكل زير خواهد بود :

```

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

```

```

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

```

```

0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24

```

روش ديگري براي تعريف كردن آرايه ها

از روش ديگري نيز مي توان براي تعريف كردن يك آرايه استفاده كرد :

```
type[ ] var-name;
```

در این روش ، کروش ها پس از مشخصه نوع آرایه قرار می گیرند و نه نام متغیر آرایه.

```
int a1 = new int[3];
```

```
int a2 = new int[3];
```

تعاریف ذیل نیز معادل هستند:

```
char twod1[ ] [ ] = new char[3] [4];
```

```
char[ ] [ ] twod2 = new char[3] [4];
```

این روش برای مواقعی مفید است که همزمان چندین آرایه همونوع تعریف می شوند. مثلاً :

```
int[ ] nums, nums2, nums3; // create three arrays
```

عبارت بالا سه متغیر آرایه ای از نوع int تعریف می کند. نتیجه مثال بالا همچون عبارت زیر است :

```
int name[ ], name2[ ], name3[ ]; // create three arrays
```

این روش همچنین برای مشخص کردن يك آرایه به عنوان مقدار حاصل از يك متد مفید است.

### چند نکته درباره رشته ها

همان گونه که ممکن است متوجه شده باشید ، در مباحث انواع داده ها و آرایه ها هیچ مطلبی درباره رشته ها یا داده های نوع string مطرح نشد. دلیل این مطلب آن است که جاوا از چنین نوعی پشتیبانی نمی کند- البته نه به شکل معمول. واقعیت امر آن است که داده های رشته ای جاوا که string نامیده می شوند ، یکی از انواع داده های پایه و ساده به شمار نمی آیند و همین طور، آرایه ای از کاراکترها نیز به شمار نمی آیند. بلکه در عوض شی می باشند و ارائه شرح کاملی از این نوع داده ها ، مستلزم آشنایی با برخی از ویژگی های مرتبط با شیء هاست. بدین ترتیب ، این موضوع در آینده و پس از بررسی شی ها مورد بررسی قرار خواهد گرفت. با این وجود ، برای آنکه بتوانید در برنامه های نمونه از رشته های ساده استفاده کنید، شرح خلاصه ذیل در اینجا ارائه شده است.

از string براي تعريف كردن متغيرهاي رشته اي استفاده مي شود. همچنين مي توانيد آرايه هاي رشته اي را تعريف كنيد. ثابتهاي رشته اي كه بين علائم نقل قول نوشته مي شوند را مي توان به متغيرهاي نوع string اختصاص داد. متغيرهاي نوع string را مي توان به ساير متغيرهاي نوع string اختصاص داد. شيءهاي نوع string را نيز مي توان به عنوان آرگومان در `println()` به كار برد. به عنوان مثال ، به دو عبارت زير توجه كنيد:

```
string str =" this is a test";  
system.out.println(str);
```

str در اینجا شیئی از نوع string است که رشته "this is a test" به آن اختصاص می یابد. این رشته به وسیله عبارت `println()` نمایش داده می شود.

### نکته اي درباره نشانه روها براي برنامه سازان C/C++

اگر از برنامه سازان با تجربه C/C++ باشيد، در آن صورت مي دانيد كه اين زبانها از نشانه روها پشتیباني مي كنند. اما، در اینجا هیچ اشاره اي به نشانه روها نشده است. دليل اين امر ساده است : جاوا از نشانه روها پشتیباني نمي كند و استفاده از آنها مجاز نيست ( يا به عبارت درست تر، جاوا از نشانه روهایی كه توسط برنامه سازان قابل دستيابي يا تغيير باشند ، پشتیباني نمي كند). جاوا نمي تواند از نشانه روها پشتیباني كند، چرا كه انجام اين كار به برنامه هاي جاوا امكان مي دهد تا شكافي بين محيط اجراي جاوا وكمپيوتر ميزبان اجرا نمايند ( به خاطر داشته باشيد كه نشانه روها مي توانند به هر آدرسي از حافظه ارجاع داشته باشند حتي آدرس هايي كه ممكن است خارج از سيستم زمان اجراي جاوا باشند ).

چون C/C++ به طور گسترده اي از نشانه روها استفاده مي كنند، ممكن است چنين تصور كنيد كه از دست دادن آنها، ايراد قابل توجهي براي جاوا به شمار مي آيد. اما اين گونه نيست. جاوا به گونه اي طراحي شده است كه تا وقتي در محدوده محيط اجرا بمانيد، هيچگاه نياز به يك نشانه رو نخواهيد داشت و از كاربرد آنها سودي نخواهيد برد.



# عملگرها

## عناوین این بخش :

عملگرهای حسابی

عملگرهای بیتی

عملگرهای منطقی بیتی

عملگرهای رابطه ای

عملگرهای منطقی بولی

عملگر تخصیص

عملگر ؟

تقدم عملگر

کاربرد پرانتز

## عملگر

جاوا عملگرهای غنی زیادی دارد. بیشتر عملگرهای آن را می توان به چهار گروه زیر تقسیم کرد : حسابی ، بیتی ، رابطه ای و منطقی. جاوا همچنین چند عملگر اضافی دارد که شرایط ویژه را مدیریت می کنند.

توجه : بیشتر عملگرها در جاوا همچون زبانهای C/C++/C# کار میکنند. اما تفاوتی جزئی هم وجود دارد.

## عملگرهای حسابی

عملگرهای حسابی به همان صورتی که در جبر به کار می رود ، در جملات حسابی به کار برده می شوند.

عملگر	نتیجه
+	جمع
-	تفریق
*	ضرب
/	تقسیم
%	باقیمانده تقسیم
++	افزایش
+=	جمع و تخصیص
-=	تفریق و تخصیص
*=	ضرب و تخصیص
/=	تقسیم و تخصیص
%=	باقیمانده و تخصیص
--	کاهش

عملوندهای عملگرهای حسابی باید از نوع عددی باشند. آنها را نمی توان همراه با داده های نوع Boolean به کار برد ، اما می توان بر روی داده های نوع char به کار برد ، چون char در جاوا اساساً زیرمجموعه int می باشد.

## عملگرهای حسابی پایه

عملگرهای حسابی پایه – جمع، تفریق، ضرب، تقسیم – همان گونه ای عمل می کنند که برای داده های عددی انتظار دارید. عملگر "-" یک فرم یگانگی دارد که تنها عملوندش را منفی می کند. به خاطر داشته باشید که وقتی عملگر تقسیم (/) با داده های صحیح به کار برده می شود، حاصل تقسیم فاقد بخش اعشاری خواهد بود.

برنامه ساده زیر، عملگرهای حسابی را نشان می دهد. این برنامه همچنین تفاوت بین تقسیم اعداد صحیح و تقسیم اعداد اعشاری را نشان می دهد.

```
// Demonstrate the basic arithmetic operators.
class BasicMath {
public static void main(String args[] ){
// arithmetic using integers
System.out.println("Integer Arithmetic");
int a = 1 + 1;
int a = a * 3;
int a = b / 4;
int a = c - a;
int a = - d;
System.out.println("a = " + a);
System.out.println("a = " + b);
System.out.println("a = " + c);
System.out.println("a = " + d);
System.out.println("a = " + e);
// arithmetic using doubles
System.out.println("\nFloating Point Arithmetic");
double da = 1 + 1;
double db = da * 3;
double dc = db / 4;
double dd = dc - a;
double de = - dd;
System.out.println("da = " + da);
System.out.println("db = " + db);
System.out.println("dc = " + dc);
```

```

System.out.println("dd = " + dd);
System.out.println("de = " + de);
}
}

```

بعد از اجراي برنامه خروجي به صورت زیر است :

```

integer Arithmetic
a=2
b=6
c=1
d=-1
e=1
floating point arithmetic
da=2
db=6
dc=1.5
dd=-0.5
de=0.5

```

## عملگر باقیمانده تقسیم

عملگر باقیمانده تقسیم ، % ، باقیمانده عمل تقسیم را برمیگرداند. این عملگر را می توان هم برای اعداد اعشاری با ممیز شناور ، و هم برای انواع داده های صحیح به کار برد.

```

// Demonstrate the % operator.
class Modulus {
public static void main(String args[] ){
int x = 42;
double y = 42.3;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}
}

```

خروجي برنامه به شكل زير مي باشد :

$X \bmod 10 = 2$

$Y \bmod 10 = 2.25$

## عملگرهاي تخصيص

جاوا عملگرهاي ويژه اي دارد كه با استفاده از آنها مي توان يك عمل حسابي را با عمل تخصيص تركيب نمود. همان گونه كه احتمالا مي دانيد ، عبارتي چون موارد زير در برنامه سازي متداولند :

`a = a + 4;`

عبارت بالا را در جاوا مي توان به شكل زير نيز نوشت :

`a += 4;`

در عبارت بالا از عملگر "+" استفاده شده است. عملکرد هر دو عبارت يكسان است : مقدار `a` را به اندازه 4 واحد افزايش مي دهند.

مثال هايي ديگر :

`a = a % 2;`

`a %= 2;`

عملگرهاي تخصيص براي تمام عملگرهاي حسابي باينري موجودند. از اين رو ، هر عبارت به شكل زير

`var = var op expression;`

را مي توان به صورت زير بازنويسي كرد :

`var op= expression;`

عملگرهاي تخصيص دو فايده دارند.

۱ – در حجم كار تايب قدرتي صرفه جويي مي شود ، چرا كه "فرم کوتاه" معادل خود هستند.

۲ – پياده سازي آنها در سيستم زمان اجراي جاوا ، كارآمدتر از معادلشان است. به خاطر اين دلايل اغلب اين عملگرها را در برنامه هاي حرفه اي جاوا خواهيدديد.

## افزايش و كاهش

"++" و "--" ، عملگرهاي افزايش و کاهش هستند. همانگونه که خواهید دید، این عملگرها چند خصوصیت ویژه دارند که آنها را کاملا جذاب ساخته اند. بحث خود را با مرور دقیق عملکرد این دو عملگر آغاز می کنیم.

عملگر افزایش ، يك واحد به عملوند خود مي افزايد. عملگر کاهش نیز يك واحد از عملوند خود مي كاهد. به عنوان مثال ، عبارت زیر :

```
x = x + 1;
```

را می توان با استفاده از عملگر افزایش به صورت ذیل بازنویسی کرد :

```
x++;
```

همچنین :

```
x = x - 1;
```

```
x--;
```

این عملگرها از این جهت که می توانند به صورت postfix ( عملگر پس از عملوند ) و همینطور prefix ( عملوند پس از عملگر ) باشند ، منحصر به فرد هستند. در مثالهاي بالا هیچ تفاوتی بین فرم هاي postfix و prefix وجود ندارد. اما ، وقتی عملگرهاي افزایش و یا کاهش بخشی از يك جمله جبري بزرگتر باشند، در آن صورت تفاوتی جزئی ، ولیکن تاثیرگذار ، بین این دو فرم وجود خواهد داشت. در فرم prefix ، پیش از آنکه از مقدار متغیر در جمله جبري استفاده شود ، عملوند افزایش یا کاهش می یابد. در فرم postfix ، مقدار متغیر در جمله جبري به کار برده می شود ، و سپس عملوند تغییر داده می شود. به عنوان مثال :

```
x = 42;
```

```
y = ++x;
```

در این حالت مقدار ۴۳ به y تخصیص می یابد، چرا که عمل افزایش پیش از تخصیص مقدار x به y انجام می شود. از این رو سطر  $y = ++x$  معادل دو عبارت زیر می باشد :

```
x = x + 1;
```

```
y = x;
```

اما وقتی مثال بالا به صورت زیر باشد :

```
x = 42;
```

```
y = x++;
```

مقدار x پیش از اجرای عملگر افزایش تخصیص می یابد، بنابراین مقدار ۴۲ به y تخصیص می یابد. البته ، در هر دو حالت مقدار ۴۳ در x ذخیره می شود. در اینجا ، سطر  $y = x++$  معادل دو عبارت ذیل است :

```
y = x;
x = x + 1;
```

## عملگرهای بیتی

جاوا چندین عملگر بیتی دارد که می توان با انواع داده های صحیح char ، short ، int ، long و byte به کار برد. این عملگر بر روی یکایک بیت های عملوند خود عمل می کنند. فهرست این عملگرها در جدول صفحه بعد آورده شده است.

عملگر	نتیجه
~	not یکانی بیتی
&	and بیتی
	or بیتی
^	xor بیتی
>>	شیفت به راست
>>>	شیفت به راست با صفر اضافی
<<	شیفت به چپ
&=	and بیتی و تخصیص
!=	or بیتی و تخصیص
^=	xor بیتی و تخصیص
>>=	شیفت به راست و تخصیص
>>>=	شیفت به راست و تخصیص با صفر اضافی
<<=	شیفت به چپ و تخصیص

از آنجایی که عملگرهای بیتی ، بیت های اعداد صحیح را پردازش و مدیریت می کنند ، مهم است که با تاثیر این چنین پردازشها بر روی مقادیر آشنا باشید. بخصوص اینکه ، دانستن اینکه جاوا چگونه مقادیر صحیح را ذخیره می کند و چگونه اعداد منفی را نمایش می دهد، مفید واقع می شود.

تمام اعداد صحیح به وسیله اعداد باینری با طول متغییر نمایش داده می شوند. به عنوان مثال ، مقدار ۴۲ نوع byte ، 00101010 است که ارزش مکانی هر رقم ، توانی از ۲ می باشد.

انواع داده هاي صحيح ( به غير از char ) ، اعداد صحيح علامت دار هستند. اين بدین معناست که مي توانند نمايانگر مقادير مثبت و منفي باشند. جاوا از نوعي روش رمزگذاري به نام " مکمل ۲ " استفاده مي کند که در آن اعداد منفي از طريق معکوس کردن تمام بيت ها ( تبديل يك ها به صفر و بلعکس ) و سپس افزودن يك واحد به حاصل ، نمايش داده مي شوند. به عنوان مثال ، 42- با معکوس کردن تمام بيت ها عدد ۴۲ ( يا 00101010 ) ، که نتیجه اش 11010101 است ، و سپس افزودن يك واحد به آن نمايش داده مي شود ( يعني ، 11010110 ). براي رمزگشايي هر عدد منفي ، نخست تمام بيتها را معکوس کنید، و سپس يك واحد به نتیجه بيافزاييد.

### عملگرهاي منطقي بيتي

عملگرهاي منطقي بيتي عبارتند از & ، | ، ^ و ~ . نتیجه اين عمليات در جدول زير نشان داده شده است. در بحث ذيل به خاطر داشته باشيد که عملگرهاي بيتي به يکايک بيتهاي هر عملوند اعمال مي شوند.

A	B	A   B	A & B	A ^ B	~ A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

### NOT بيتي

عملگر يگاني NOT (~) که مکمل بيتي نيز ناميده مي شود، تمام بيت هاي عملوند خود را معکوس مي کند. به عنوان مثال ، عدد ۴۲ که معادل باينري آن به شکل زير است :

00101010

پس از اعمال عملگر NOT به شکل زير تبديل مي شود :

11010101



## AND بیتی

چنانچه هر دو عملوند AND (&) يك باشد ، حاصل نیز يك مي شود ، در غير اين صورت در كليه حالات حاصل صفر مي شود.

	00101010	42
&	00001111	15
	<hr/>	
	00001010	10

## OR بیتی

عملگر OR (|) ، بیت ها را به گونه اي تركيب مي كند كه چنانچه هر يك از بیت هاي عملوندها يك باشد، بیت حاصل نیز يك مي شود :

	00101010	42
	00001111	15
	<hr/>	
	00101111	47

## XOR بیتی

عملگر XOR (^) ، بیت ها را به گونه اي تركيب مي كند كه چنانچه دقیقا يك عملوند يك باشد ، نتیجه يك شود. در غير اين صورت ، نتیجه صفر مي شود.

	00101010	42
^	00001111	15
	<hr/>	
	00100101	37

به چگونگي تغيير بیت ها دقت كنید. هرگاه عملوند دوم صفر باشد ، عملوند نخست تغيير نمي كند. اين ویژگی براي برخي از عملیات پردازش بیت ها مفید واقع مي شود.

## کاربرد عملگرهای منطقی بیتی

برنامه زیر عملگرهای منطقی بیتی را نشان می دهد :

```
// Demonstrate the bitwise logical operators.
class BitLogic {
public static void main(String args[] ){
String binary[] = {
"0000"/ "0001"/ "0010"/ "0011"/ "0100"/ "0101"/ "0110"/ "0111"/
"1000"/ "1001"/ "1010"/ "1011"/ "1100"/ "1101"/ "1110"/ "1111"
};
int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
int c = a | b;
int d = a & b;
int e = a ^ b;
int f = ( ~a & b ) | ( a & ~b );
int g = ~a & 0x0f;
System.out.println(" a = " + binary[a]);
System.out.println(" b = " + binary[b]);
System.out.println(" a|b = " + binary[c]);
System.out.println(" a&b = " + binary[d]);
System.out.println(" a^b = " + binary[e]);
System.out.println(" ~a&b|a&~b = " + binary[f]);
System.out.println(" ~a = " + binary[g]);
}
}
```

a ، b در این مثال بیت هایی دارند که نمایانگر هر چهار حالت ممکن برای دو رقم باینری هستند :  
۱-۱ ، ۰-۱ ، ۱-۰ ، ۰-۰ . با نگاه کردن به نتایج c و d می توانید به چگونگی عملکرد | و & بر روی هر یک از بیت ها دریابید. رشته آرایه ای که binary نام دارد ، حاوی نمایش باینری اعداد صفر تا ۱۵ است. ایندکس های آرایه در این مثال به گونه ای هستند تا نمایش باینری هر یک از نتایج را نشان دهند. آرایه به گونه ای ساخته شده است تا نمایش رشته ای مقدار باینری n ، در binary[n] ذخیره شود. مقدار a- با 0x0f (00001111) AND می شود تا مقدارش به کمتر از ۱۶ کاهش یابد تا بتوان با استفاده از آرایه binary نمایش داد. خروجی برنامه در ذیل نشان داده شده است :

a = 0011

```

b = 0110
a | b = 0111
a & b = 0010
a ^ b = 0101
~ a & b | a & ~ b = 0101
~ a = 1100

```

## شیفت به چپ

عملکرد شیفت به چپ (<<) ، تمام بیت های عملوندش را به اندازه تعداد دفعات مشخص شده به سمت چپ انتقال می دهد. شکل کلی آن در زیر نشان داده شده است :

```
Value << num
```

Num مشخص کننده تعداد دفعاتی است که مقدار value باید به چپ انتقال یابد. یعنی ، "<<" تمام بیت های مقدار مورد نظر را به اندازه num مرتبه به سمت چپ انتقال می دهد. به ازای هر بار انتقال (شیفت) ، بیت منتهی الیه سمت چپ از دست می رود و یک بین صفر از سمت راست اضافه می شود. این بدین معناست که وقتی عملگر "شیفت به چپ" به یک عملوند int اعمال می شود، بیت ها پس از گذشت از موقعیت سی و یکم از دست می روند. چنانچه عملوند از نوع long باشد، در آن صورت بیت ها پس از گذشت از موقعیت شصت و سوم از دست می روند.

وقتی مقادیر نوع byte و short را شیفت می دهید ، عمل ارتقا خودکار جاوا منجر به تولید نتایج غیرمنتظره می شود. همان گونه که می دانید ، وقتی یک جمله جبری ارزیابی می شود، مقادیر نوع byte و short به int ارتقا می یابند. به علاوه نتیجه یک چنین جمله ای نیز int می شود. این بدین معناست که نتیجه شیفت به چپ در مقادیر byte ، short و int می شود و بیت های شیفت شده هم تا زمانی از موقعیت سی و یکم نگذرند ، از دست نمی روند. به علاوه ، مقادیر byte و short منفی نیز هنگام ارتقا به int از سمت بیت علامتشان بزرگ می شوند. از این رو، بیت های سمت چپ با یک پر می شوند. به همین دلیل است که انجام عمل شیفت به چپ در مقادیر نوع byte و short بدین معناست که باید بایت های سمت چپ نتیجه نوع int را نادیده بگیرد. به عنوان مثال ، اگر یک مقدار نوع byte را به سمت چپ شیفت بدهید ، آن مقدار ابتدا به int ارتقا می یابد و سپس شیفت داده می شود. این بدین معناست که اگر منظورتان نتیجه "شیفت به چپ" مقدار byte باشد،

باید سه بایت سمت چپ نتیجه را نادیده بگیرید. آسانترین روش برای انجام این کار ، آن است که نتیجه را مجدداً با استفاده از "casting" به byte تبدیل کنید. این مفهوم در مثال زیر نشان داده شده است :

```
// Left shifting a byte value.
class ByteShift {
public static void main(String args[] ){
byte a = 64/ b;
int i;
i = a << 2;
b =( byte( )a << 2);
System.out.println("Original value of a : " + a);
System.out.println("i and b : " + i + " " + b);
}
}
```

خروجی حاصل از این برنامه در ذیل نشان داده شده است :

```
Original value of a: 64
I and b: 256      0
```

چون a برای ارزیابی به int ارتقا می یابد ، دو مرتبه "شیفت به چپ" مقدار ۶۴ ( 0000 0000 ) منجر به مقدار ۲۵۶ ( 1 0000 0000 ) برای i می شود. اما ، مقدار b پس از شیفت صفر می شود، چون بایت سمت چپ صفر می شود. تنها بیت یک آن، پس از شیفت از دست رفته است. چون نتیجه هر شیفت چپ، دو برابر شدن مقدار اولیه است، برنامه سازان از این امر به عنوان یک جایگزین کارآمدتر برای ضرب در ۲ استفاده می کنند. اما باید مواظب هم باشید. اگر یک بیت "یک" را به موقعیت آخرین بیت سمت چپ (بیت ۳۱ یا ۶۳ ) انتقال دهید، مقدارتان منفی خواهد شد. این نکته در برنامه زیر نشان داده شده است :

```
// Left shifting as a quick way to multiply by 2.
class MultByTwo {
public static void main(String args[] ){
int i;
int num = 0xFFFFFFFF;
for(i=0; i<4; i++){
num = num << 1;
System.out.println(num);
}
}
```

```
}  
}
```

خروجي برنامه به صورت زیر است :

```
536870908  
1073741816  
2147483632  
-32
```

مقدار اولیه به دقت انتخاب شده است تا پس از ۴ مرتبه "شیفت به چپ" به ۳۲- تبدیل شود. همان گونه که می بینید وقتی یک بیت "یک" به موقعیت ۳۱ انتقال می یابد ، عدد حاصل منفي می شود.

### شیفت به راست

عملگر شیفت به راست (>>) ، تمام بیتهاي عملونش را به اندازه تعداد دفعات مشخص شده به سمت راست انتقال می دهد. شکل کلی آن در زیر نشان داده شده است :

```
value >> num
```

num مشخص کننده تعداد دفعاتي است که مقدار value باید به سمت راست انتقال یابد. يعني ، ">>" تمام بیت هاي مقدار مورد نظر را به اندازه num مرتبه به سمت راست انتقال می دهد. مثال زیر مقدار ۳۲ را ۲ مرتبه به سمت راست شیفت می دهد تا مقدار ۸ در a ذخیره شود :

```
int a = 32;  
a = a >> 2; // a now contains 8
```

وقتي بیت هاي يك مقدار به خارج از محدوده اش شیفت داده شوند، تمام آن بیت ها از دست می روند. مثلا ، مثال زیر مقدار ۳۵ را ۲ مرتبه به راست شیفت می دهد و در نتیجه با از دست رفتن بیت هاي سمت راست ، مجددا مقدار ۸ در a ذخیره می شود.

```
int a = 35;  
a = a >> 2; // a still contains 8
```

بررسي همان عمل با اعداد باینري ، اتفاقات را به وضوح نشان می دهد :

```
00100011      35  
>> 2  
00001000      8
```

هر بار که مقداري را به راست شیفت مي دهيد ، آن مقدار بر ۲ تقسيم مي شود — و باقيمانده نادیده انگاشته مي شود. از اين ویژگی مي توانيد براي انجام عمل تقسيم بر ۲ به شکل کارآمدتر استفاده كنيد. البته ، بايد اطمینان حاصل كنيد که بیت ها از سمت راست خارج نمي شوند. وقتي مقداري را به سمت راست شیفت مي دهيد، سمت چپ ترين بیت ها که توسط عمل شیفت راست نمايان شده اند، به وسيله محتوای پيشين سمت چپ ترين بیت جایگزین مي شوند. اين عمل "sign extension" نامیده شده و سبب حفظ علامت اعداد منفي به هنگام شیفت دادن آنها به سمت راست مي شود. به عنوان مثال ،  $1 >> 8$  - ، برابر 4- است که باینري اين عمل در ذیل نشان داده شده است :

```
11111000      -8
```

```
>>1
```

```
11111100      -4
```

جالب است بدانيد که اگر ۱- را به سمت راست شیفت دهيد ، نتیجه همیشه ۱- باقي مي ماند، چرا که فرآیند "sign extension" سبب مي شود تا همواره بیت هاي يك در سمت چپ جایگزین شوند.

گاهی اوقات انجام فرآیند، پيش گفته به هنگام شیفت دادن مقادير به سمت راست ضرورت ندارد. به عنوان مثال ، برنامه زیر يك مقدار نوع byte را به رشته هگزادسیمال معادلش تبدیل مي کند. دقت كنيد که مقدار حاصل از شیفت ، از طریق AND کردن با 0x0f ماسك مي شود تا نتیجه "sign extension" نادیده انگاشته شود و در نتیجه ، بتوان از مقدار حاصل به عنوان ایندکس آرایه کاراکترهاي هگزادسیمال استفاده نمود.

```
// Masking sign extension.
class HexByte {
static public void main(String args[] ){
char hex[] = {
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
};
byte b =( byte )0xf1
System.out.println("b = 0x" + hex[(b >> 4 )& 0x0f] + hex[b & 0x0f]);
}
}
```

خروجي برنامه در زیر نشان داده شده است :

```
b = 0xf1
```

### شیفت به راست بدون علامت

همان گونه که در قسمت پیش دیدید، عملگر  $\gg$  ، بیت منتهی الیه سمت چپ را به طور خودکار هر بار پس از شیفت با مقدار پیشین آن پر می کند. این کار موجب حفظ علامت مقدار مورد نظر می شود. اما، گاهی این امر لازم نمی شود. به عنوان مثال ، اگر چیزی را شیفت می دهید که نمایانگر يك مقدار عددي نیست ، ممکن است نیاز به "sign extension" نداشته باشید. این شرایط هنگام کار با مقادیر مبتنی بر پیکسل و گرافیک متداول است. در این گونه موارد ، عموماً باید يك صفر را به بیت منتهی الیه سمت چپ انتقال می دهد.

عملکرد  $\ggg$  در مثال زیر نشان داده شده است. در این مثال ،  $a$  با ۱- مقداردهی شده و بنابراین ، کل ۳۲ بیت با يك پر می شود. این مقدار سپس ۲۴ مرتبه به سمت راست انتقال یافته و ۲۴ بیت سمت چپ با صفر پر شده و عمل sign extension نادیده انگاشته می شود. بدین ترتیب ۲۵۵ در  $a$  ذخیره میشود.

```
int a = -1;
a = a >>> 24;
```

انجام عمل بالا در ذیل با ارقام باینری تکرار شده است تا بیشتر با اتفاقاتی که رخ می دهند آشنا شوید :

```
11111111 11111111 11111111 11111111 -1 in binary as an int
>>>24
00000000 00000000 00000000 11111111 255 in binary as an int
```

عملکرد  $\ggg$  اغلب به اندازه ای که دوست دارید مفید واقع نمی شود ، چراکه تنها برای مقادیر ۳۲ و ۶۴ بیتی معنا خواهد داشت. به خاطر داشته باشید که مقادیر کوچکتر در جملات جبری به طور خودکار به  $\text{int}$  ارتقا می یابند. این بدین معناست که sign-extension رخ می دهد و عمل شیفت به جای مقادیر ۸ یا ۱۶ بیتی ، در مقدار ۳۲ بیتی انجام می شود. یعنی ، ممکن است برخی

افراد انتظار داشته باشند که عمل شیفت به راست بدون علامت در مقادیر نوع byte به گونه ای انجام شود تا مقداردهی بیت ها با صفر ، از بیت موقعیت ۷ آغاز شود. اما این گونه نیست ، چراکه در واقع يك مقدار ۳۲ بیتی شیفت داده می شود. برنامه زیر نشانگر این مطلب است :

```
// Unsigned shifting a byte value.
class ByteUShift {
static public void main(String args[] ){
char hex[] = {
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
};
byte b =( byte )0xf1
byte c =( byte( )b >> 4);
byte d =( byte( )b >>> 4);
byte e =( byte(( )b & 0xff )>> 4);
System.out.println(" b = 0x"
+ hex[(b >> 4 )& 0x0f] + hex[b & 0x0f]);
System.out.println(" b >> 4 = 0x"
+ hex[(c >> 4 )& 0x0f] + hex[c & 0x0f]);
System.out.println(" b >>> 4 = 0x"
+ hex[(d >> 4 )& 0x0f] + hex[d & 0x0f]);
System.out.println("(b & 0x0f )>> 4 = 0x"
+ hex[(e >> 4 )& 0x0f] + hex[e & 0x0f]);
}
}
```

خروجی این برنامه نشان می دهد که چگونه عملگر >>> در حین کار با مقادیر نوع byte ، به ظاهر کاری انجام نمی دهد. در اینجا ، متغیر b با يك مقدار دلخواه نوع byte مقداردهی شده است. حاصل شیفت به راست b ( به اندازه ۳ بیت )، به خاطر انجام عمل مورد انتظار sign extension در c ذخیره می شود ( یعنی 0xff ). سپس حاصل همان عمل شیفت، البته بدون علامت، که انتظار می رود 0xf باشد، به d تخصیص می یابد. اما در حقیقت چون عمل sign extension هنگام ارتقا b به int ( پیش از عمل شیفت ) رخ داده است، 0xff در d ذخیره می شود. در جمله آخر نیز مقدار نوع byte متغیر b از طریق AND شدن با ۸ بیت ماسک شده، چهار مرتبه به راست شیفت داده میشود، و حاصل آنکه باید 0xf باشد، در c ذخیره می شود.



توجه داشته باشید که برای متغیر d از عملگر >>> استفاده نشده است، چرا که وضعیت بیت علامت پس از AND کردن مشخص می شود.

```
b = 0xf1
b >> 4 - 0xff
b >>> 4 = 0xff
( b & 0xff ) >> 4 = 0x0f
```

## عملگرهای بیتی و تخصیص

تمام عملگرهای بیتی باینری فرم کوتاهی دارند که مشابه عملگرهای جبری است که عمل تخصیص را با عمل بیتی ترکیب می کنند. به عنوان مثال، دو عبارت زیر که مقدار a را به اندازه ۴ بیت به سمت راست شیفت می دهند، معادل یکدیگرند :

```
a = a >> 4;
a >>= 4;
```

همین طور دو عبارت زیر هم که جمله بیتی a OR b به a تخصیص می یابد، معادل یکدیگرند :

```
a = a | b;
a |= b;
```

در برنامه زیر ۴ فیلد صحیح ایجاد شده و سپس فرم کوتاه عملگرهای بیتی و تخصیص برای پردازش و مدیریت آنها استفاده می شود :

```
class OpBitEquals {
public static void main(String args[] ){
int a = 1;
int b = 2;
int c = 3;
a |= 4;
b >>= 1;
c <<= 1;
a ^= c;
System.out.println("a = " + a );
System.out.println("b = " + b );
System.out.println("c = " + c );
}
}
```

خروجي برنامه در ذیل نشان داده شده است :

a = 3

b = 1

c = 6

### عملگرهاي رابطه اي

عملگرهاي رابطه اي ، رابطه يك عملوند را نسبت به يك عملوند ديگر مشخص مي كنند. اين عملگرها تساوي و ترتيب عملوندها را تعيين مي كنند. فهرست عملگرهاي رابطه اي در ذیل نشان داده شده است :

عملگر	نتيجه
==	مساوي است با
!=	خالف است با
>	بزرگتر از
<	کوچکتر از
>=	بزرگتر يا مساوي است با
<=	کوچکتر يا مساوي است با

نتيجه اين عمليات، يك مقدار بولي است. عملگرهاي رابطه اي بيشتر در جملات جبري كه عبارت if را كنترل مي كنند ، و همين طور در حلقه هاي گوناگون به كار برده ميشوند. انواع مختلف داده ها در جاوا ، از جمله اعداد صحيح ، اعداد اعشاري با ممیز شناور ، کاراکترها و مقادير بولي را مي توان با استفاده از عملگر تساوي (==) يا "!=" مقايسه نمود. توجه داشته باشيد كه عملگر تساوي در جاوا از دو "=" تشكيل شده است ( به خاطر داشته باشيد : از علامت "=" براي تخصيص استفاده مي شود). تنها مقادير عددي را مي توان با عملگرهاي نشانگر ترتيب مقايسه نمود. يعني ، تنها عملوندهاي صحيح ،اعشاري يا ممیز شناور و کاراکتری را مي توان مقايسه نمود تا مشخص شود كه کداميك بزرگتر و کداميك کوچکترند. همان گونه كه گفته شد، نتيجه حاصل از عملگر رابطه اي ، يك مقدار بولي است. مثلا عبارت ذیل كاملا معتبر هستند :

```
int a = 4;
```

```
int b = 1;
```

```
boolean c = a < b;
```

در این حالت نتیجه  $a < b$  ( که false است ) در c ذخیره می شود.

اگر تجربه کار با C/C++ را دارید ، لطفا به نکته زیر دقت کنید. انواع عبارت زیر در C/C++ بسیار متداولند :

```
int done;  
// ...  
if ( ! done ) ... // valid in C/C++  
if ( done ) ... // but not valid in Java
```

عبارت بالا را باید در جاوا به صورت زیر نوشت :

```
If ( done == 0 ) ... // This is Java-style  
If ( done != 0 ) ...
```

دلیل این امر آن است که تعریف true و false در جاوا همچون C/C++ نیست. True در C/C++ ، هر مقدار مخالف صفر است و false نشانگر مقدار صفر. True و false در جاوا مقادیر غیر عددی هستند که ارتباطی با صفر و غیر صفر ندارند. بنابراین ، برای مقایسه با صفر و غیر صفر ، باید از يك یا چند عملگر رابطه ای استفاده کنید.

## عملگرهای منطقی بولی

عملگرهای منطقی بولی که در اینجا نشان داده شده اند، تنها برای عملوندهای نوع Boolean قابل استفاده هستند. تمام عملگرهای منطقی بولی دو مقدار نوع Boolean را با هم ترکیب می کنند تا نتیجه يك مقدار نوع Boolean شود.

عملگر	نتیجه
&	AND منطقی
	OR منطقی
^	XOR منطقی
	OR اتصال کوتاه
&&	AND اتصال کوتاه
!	NOT یگانی منطقی
&=	AND و تخصیص
=	OR و تخصیص

XOR و تخصیص	^=
مساوی است با	==
مخالف است با	!=
if-then-else سه تایی	?:

عملکرد عملگرهای بولی منطقی ( & ، | و ^ ) بر روی مقادیر بولی همچون عملکردشان بر روی بیت های یک عدد صحیح است. عملکرد منطقی "!" وضعیت بولی را معکوس می کند : `!true == false` و `false == true` !. نتیجه هر یک از عملیات منطقی در جدول زیر نشان داده شده است :

A	B	A   B	A & B	A ^ B	~ A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

برنامه زیر تقریباً معادل مثال BitLogic است که پیش از این مطرح شده است. با این تفاوت که به جای بیت های باینری بر روی مقادیر بولی عمل می کند :

```
// Demonstrate the boolean logical operators.
class BoolLogic {
public static void main(String args[] ){
boolean a = true;
boolean b = false;
boolean c = a | b;
boolean d = a & b;
boolean e = a ^ b;
boolean f = ( !a & b ) | ( a & !b );
boolean g = !a;
System.out.println(" a = " + a);
System.out.println(" b = " + b);
System.out.println(" a|b = " + c);
System.out.println(" a&b = " + d);
System.out.println(" a^b = " + e);
System.out.println("!a&b|a&!b = " + f);
System.out.println(" !a = " + g);
}
```

}

پس از اجرای این برنامه ، خواهید دید که همان قوانین منطقی بیت ها ، به مقادیر نوع Boolean نیز اعمال می شوند. همان گونه که از خروجی ذیل مشخص است ، نمایش رشته ای مقادیر بولی جاوا ، یکی از دو مقدار لیترال true یا false است :

```
a = true
b = false
a | b = true
a & b = false
a ^ b = true
a & b | a & ! b = true
! a = false
```

### عملگرهای منطقی اتصال کوتاه

جاوا دو عملگر بولی جالب دارد که در بسیاری از زبانهای کامپیوتری دیگر وجود ندارد. این عملگرها ، نگارشهای ثانویه دو عملگر AND و OR بولی هستند و تحت عنوان عملگرهای منطقی "اتصال کوتاه" شناخته شده اند. همان گونه که از جدول قسمت پیش مشخص است، نتیجه عملگر OR زمانی true است که A مقدارش true باشد ، صرفنظر از مقدار B. همین طور ، حاصل عملگر AND زمانی false می شود که A مقدارش false باشد ، صرفنظر از مقدار B. اگر به جای فرم های | و & این عملگرها از دو فرم || و && استفاده کنید ، در آن صورت چنانچه جاوا بتواند نتیجه جمله شرطی را تنها به وسیله عملوند سمت چپ تعیین کند ، در آن صورت دیگر عملوند سمت راست را ارزیابی نخواهد کرد. این امر برای مواقعی که عملوند سمت راست به true یا false بودن عملوند سمت چپ وابسته باشد ، بسیار مفید است. به عنوان مثال ، عبارت زیر نشان می دهد که چگونه با استفاده از ارزیابی منطقی اتصال کوتاه می توانید پیش از يك عملوند تقسیم اطمینان حاصل کنید که حاصل آن درست خواهد بود یا خیر :

```
if ( denom != 0 && num / denom > 10 )
```

چون ار فرم اتصال کوتاه AND ( && ) استفاده شده است ، احتمال بروز استثنای "زمان-اجرا" به دلیل صفر بودن denom از بین می رود. اگر این سطر با استفاده از نگارش & عملگر AND

نوشته می‌شد ، می‌بایست هر دو طرف عملگر منطقی ارزیابی می‌شد و در نتیجه ، استثنای "زمان-اجرا" در صورت صفر بودن denom پیش می‌آید.

خوب است در مواقعی که با منطق بولی سر و کار دارید ، از فرم های اتصال کوتاه AND و OR استفاده کنید ، و نگارشهای تکی نمادی را صرفاً برای عملیات بیتی باقی بگذارید. اما ، این قانون استثنا هم دارد. به عنوان مثال ، عبارت زیر را در نظر بگیرید :

```
if ( c==1 & e++ < 100 ) d = 100;
```

استفاده از & در اینجا تضمین می‌کند که افزایش مقدار e صرفنظر از یک بودن مقدار c انجام می‌شود.

## عملگر تخصیص

عملگر تخصیص با علامت "=" نشان داده می‌شود. عملکرد عملگر تخصیص در جاوا همچون سایر زبانهای کامپیوتری است. شکل کلی آن به صورت زیر است :

```
Var = expression ;
```

نوع داده var باید با نوع expression سازگار باشد.

عملکرد تخصیص ویژگی جالبی دارد که ممکن است با آن آشنا نباشید : امکان ایجاد زنجیره ای از عملیات تخصیص را فراهم می‌سازد. به عنوان مثال ، به عبارت زیر توجه کنید :

```
int x, y, z;
```

```
x = y = z = 100; // set x, y, and z to 100;
```

در عبارت بالا تنها با استفاده از یک عبارت مقدار هر سه متغیر ۱۰۰ میشود. دلیل میسر بودن این ویژگی آن است که "=" عملگری است که نتیجه آن ، سمت راست عبارت است. از این رو ، مقدار  $z=100$  ، ۱۰۰ است که به y تخصیص می‌یابد ، و آن نیز به نوبه خود به x تخصیص می‌یابد. استفاده از "زنجیر تخصیص" روش آسانی برای نسبت دادن یک مقدار مشترک به گروهی از متغیرهاست.

## عملگر ؟

جاوا نوعي عملگر سه تايي ويژه دارد كه مي تواند جانشين برخي از انواع خاص عبارتهاي if-then-else شود. اين عملگر ، "?" است. عملگر "?" ممكن است در نگاه اول گيج كننده به نظر برسد ، اما پس از تسلط يافتن بر آن مي توانيد به شكل موثري از آن استفاده كنيد. شكل كلي آن به صورت زير است :

```
expression1 ? expression2 : expression3
```

expression1 مي تواند هر جمله اي با حاصل نوع Boolean باشد. اگر حاصل آن true باشد ، در آن صورت expression2 ارزيابي مي شود ، در غير اين صورت expression3 ارزيابي خواهد شد. نتيجه عملگر ؟ ، همان نتيجه ارزيابي جملات آن است. به مثالي از کاربرد ؟ توجه كنيد :

```
ratio = denom == 0 ? 0 : num / denom
```

وقتي جاوا اين عبارت را ارزيابي مي كند ، نخست به جمله سمت چپ علامت سوال توجه مي كند. اگر denom برابر صفر باشد ، در آن صورت جمله بين علامت سوال و ":" ارزيابي شده و به عنوان مقدار كل "?" مورد استفاده قرار ميگيرد. اگر denom مخالف صفر باشد ، در آن صورت جمله پس از ":" ارزيابي شده و به عنوان مقدار كل "?" به كار برده مي شود سپس نتيجه حاصل از "?" به ratio تخصيص مي يابد.

در زير برنامه اي مشاهده مي كنيد كه عملگر ؟ را نشان مي دهد. اين برنامه از عملگر فوق براي نگهداري مقدار مطلق يك متغير استفاده مي كند.

```
// Demonstrate ?.  
class Ternary {  
public static void main(String args[] ){  
int i, k;  
i = 10;  
k = i < 0 ? -i : i; // get absolute value of i  
System.out.print("Absolute value of ");  
System.out.println(i + " is " + k);  
i = -10;  
k = i < 0 ? -i : i; // get absolute value of i  
System.out.print("Absolute value of ");  
System.out.println(i + " is " + k);  
}  
}
```

خروجي این برنامه بصورت زیر می باشد :

Absolute value of 10 is 10

Absolute value of- 10 is 10

## تقدم عملگرها

جدول زیر ترتیب حق تقدم عملگرهاي جاوا را از بالاترین اولویت تا پایین ترین نشان می دهد. دقت کنید که در سطر اول اقلامي وجود دارد که معمولاً بعنوان عملگر درباره آنها فکر نمیکنید : پرانتزها ، گروه ها و عملگر نقطه (.) ، این موارد جداساز نامیده می شوند ، اما عملکرد آنها در جملات جبري همچون عملگرهاست. پرانتزها برای تغییر تقدم يك عمل به کار برده می شوند. گروه ها امکان مشخص کردن ایندکس آرایه ها را فراهم می سازد. از عملگر نقطه برای ارجاع به شی ها استفاده میشود.

بالاترین			
()	[]		
++	--	~	~
*	/	%	
+	-		
>>	>>>	<<	
>	=>	<	<=
==	!=		
&			
^			
&&			
?:			
=	op=		
پایین ترین			

## کاربرد پرانتزها



پرانتزها حق تقدم عملياتي را كه دربر گرفته اند ، افزايش مي دهند. اينكار اغلب براي نگهداري  
نتيجه دلخواهتان ضروري است. بعنوان مثال ، عبارت زير را در نظر بگيريد :

$a \gg b + 3$

اين عبارت ابتدا ۳ را به  $b$  اضافه نموده و سپس  $a$  را مطابق آن نتيجه بطرف راست حركت مي  
دهد. اين عبارت را مي توان با استفاده از پرانتزهاي اضافي بصورت زير دوباره نويسي نمود :

$a \gg (b + 3)$

اما ، اگر بخواهيد ابتدا  $a$  را با مكانهاي  $b$  بطرف راست حركت داده و سپس ۳ را به نتيجه آن  
اضافه كنيد ، بايد عبارت را بصورت زير در پرانتز قرار دهيد :

$(a \ll b) + 3$

علاوه بر تغيير حق تقدم عادي يك عملگر ، پرانتزها را مي توان گاهي براي روشن نمودن مفهوم  
يك عبارت نيز بكار برد. براي هر كسي كه كد شما را مي خواند، درك يك عبارت پيچيده بسيار  
مشكل است. اضافه نمودن پرانتزهاي اضافي و روشنگر به عبارات پيچيده مي تواند از ابهامات  
بعدي جلوگیری نماید. بعنوان مثال ، كداميك از عبارات زير راحت تر خوانده و درك مي شوند ؟

$a | 4 + c \gg b \& 7$

$(a | (((4 + c) \gg b) \& 7))$

يك نکته ديگر : پرانتزها ( بطور كلي خواه اضافي باشند يا نه ) سطح عملکرد برنامه شما را  
كاهش نمي دهند. بنابراين ، اضافه كردن پرانتزها براي كاهش ابهام تأثيري روي برنامه شما  
نخواهد داشت.

# عبارات کنترلی

## عناوین این بخش :

عبارات انتخاب جاوا

عبارات تکرار

عبارات پرش

مدیریت استثناها

در این فصل به کنترل ترتیب اجرای برنامه ، دستورات شرطی و تکرار پرداخته میشود.

## عبارات کنترلي

هر زبان برنامه سازي از عبارات کنترلي براي هدايت جريان اجراي برنامه بر اساس تغييرات وضعيت آن استفاده مي کند. عبارات کنترلي برنامه هاي جاوا را مي توان به دسته هاي ذيل طبقه بندي نمود : انتخاب ، اکرار و پرش. عبارات انتخاب ، به برنامه هاي تان امکان مي دهند تا مسيرهاي اجرايي مختلفي را بر اساس نتيجه يك جمله جبري يا وضعيت يك متغير انتخاب کنند. عبارات تکرار ، به برنامه ها امکان مي دهند تا يك يا چند عبارت را تکرار کنند ( يعني ، عبارات تکرار ، حلقه ها را تشکيل مي دهند. ) عبارات پرش ، به برنامه هاي تان امکان مي دهند تا به صورت غير خطي اجرا شوند.

توجه : عبارات کنترلي جاوا مشابه زبانهاي C#/C++/C مي باشد. اما چند تفاوت وجود دارد به ويژه عبارات break و continue.

## عبارات انتخاب جاوا

جاوا از دو دستور انتخاب پشنيابي مي کند : if و switch. با اين دستورات شما اجراي برنامه را براساس شرايطي که فقط حين اجراي برنامه اتفاق مي افتند کنترول مي کنيد. اگر سابقه برنامه نويسي با C++/C را نداريد، از قدرت و انعطاف پذيري موجود در اين دو دستور متعجب و شگفت زده خواهيد شد.

### if

دستور if دستور انشعاب شرطي در جاوا است. از اين دستور مي توان استفاده نمود و اجراي برنامه را طي دو مسير متفاوت به جريان انداخت. شکل کلي اين دستور بصورت زير است :

```
if( condition )statement1;  
else statement2;
```

در اینجا هر statement ممکن است يك دستور منفرد یا يك دستور مرکب قرار گرفته در آکولاد ( یعنی يك بلوك ) باشد. condition ( شرط ) هر عبارتي است كه يك مقدار boolean را برمي گرداند . جمله else اختياري است.

if بصورت زیر کار می کند : اگر شرایط محقق باشد ( حاصل condition ، مقدار true باشد . ) آنگاه statement1 اجرا می شود. در غیر اینصورت statement2 ( در صورت وجود ) اجرا خواهد شد. تحت هیچ شرایطی هر دو دستور با هم اجرا نخواهند شد. بعنوان مثال ، در نظر بگیرید :

```
int a, b;  
// ...  
if(a > b ) a = 0;  
else b = 0;
```

اگر a کوچکتر از b باشد ، آنگاه a برابر صفر می شود. در غیر اینصورت b برابر صفر خواهد شد. در هیچ شرایطی این دو متغیر در آن واحد برابر صفر نمی شوند.

غالب اوقات ، عبارتي كه براي كنترل if استفاده می شود شامل عملگرهاي رابطه اي است ، اما از نظر فني ضرورتي وجود ندارد. عبارت if را می توان همچون مثال زیر با يك متغیر بولي هم كنترل نمود :

```
boolean dataAvailable;  
// ...  
if( dataAvailable )  
    ProcessData();  
else  
    waitForMoreData();
```

به خاطر داشته باشید كه فقط يك دستور می تواند مستقيماً بعد از if یا else قرار گیرد. اگر بخواهید دستورات بیشتری داخل نمایید ، نيازي به ايجاد يك بلوك ندارید ، نظیر این قطعه كه در زیر آمده است :

```
int bytesAvailable;  
// ...  
if( bytesAvailable > 0 ) {  
    ProcessData();  
    bytesAvailable -= n;
```

```

} else
    waitForMoreData();

```

در این مثال ، هر دو دستور داخل بلوك if اجرا خواهند شد اگر bytesAvailable بزرگتر از صفر باشد.

برخي از برنامه نویسان استفاده از آكولاد را هنگام استفاده از if ، حتي زمانیکه فقط يك دستور در هر جمله وجود داشته باشد را مناسب میدانند. این امر سبب مي شود تا بعدا بتوان براحتي دستور دیگری را اضافه نمود و نگرانی از فراموش کردن آكولادها نخواهید داشت. در حقیقت ، فراموش کردن تعریف يك بلوك هنگامی که نیاز است ، یکی از دلایل رایج بروز خطاها میباشد. بعنوان مثال قطعه زیر از يك كد را در نظر بگیرید :

```

int bytesAvailable;
// ...
if( bytesAvailable > 0 ){
    ProcessData();
    bytesAvailable -= n;
} else
    waitForMoreData();
    bytesAvailable = n;

```

بنظر خیلی روشن است که دستور bytesAvailable = n; طوري طراحی شده تا داخل جمله else اجرا گردد ، و این بخاطر سطح طراحی آن است. اما حتما بیاد دارید که فضاي خالي براي جاوا اهميتي ندارد و راهي وجود ندارد که کامپایلر بفهمد چه مقصودي وجود دارد. این كد بدون مشکل کامپایل خواهد شد ، اما هنگام اجرا بطور ناصحیح اجرا خواهد شد. مثال بالا به صورت زیر اصلاح میشود :

```

int bytesAvailable;
// ...
if( bytesAvailable > 0 ) {
    ProcessData();
    bytesAvailable -= n;
} else
    waitForMoreData();

```

```
bytesAvailable = n;
}
```

## if های تو در تو

منظور از if تو در تو ، آن است که عبارت if دیگری ، در يك if یا else دیگر قرار می گیرد. if های تودرتو در برنامه سازی بسیار متداولند. وقتی if ها را به صورت تو در تو می نویسد ، مطلب اصلی که باید به خاطر بسپارید آن است که هر عبارت else همیشه به نزدیکترین عبارت if موجود در همان بلوک else که با else دیگری هم مرتبط نیست ، ارجاع دارد. به مثال زیر توجه کنید :

```
if(i == 10){
    if( j < 20 ) a = b;
    if( k > 100 ) a = d; // this if is
    else a = c;          // associated with this else
}
else a = d;              // this else refers to if(i == 10)
```

همانگونه که توضیحات نشان می دهند ، else نهایی با if(j<20) مرتبط نیست ، چراکه در همان بلوک قرار ندارد ( با وجود آنکه نزدیکترین if بدون else است ). بلکه در عوض ، آخرین else با if(i==10) مرتبط است. else داخلی به if(k>100) ارجاع دارد ، چرا که نزدیکترین if در همان بلوک است.

## نردبان if-else-if

يك ساختار برنامه نویسی رایج براساس يك ترتیب از if های تودرتو بنا شده است. این ساختار نردبان if-else-if است و بصورت زیر می باشد :

```
if(condition)
    statement;
else if(condition)
    statement;
else if(condition)
    statement;
```

```
.
.
.
else
```

```
    statement;
```

دستورات if از بالا به پایین اجرا می شوند. مادامیکه یکی از شرایط کنترل کننده if صحیح باشد (true)، دستور همراه با آن if اجرا می شود، و بقیه نردبان رد خواهد شد. اگر هیچکدام از شرایط صحیح نباشند، آنگاه دستور else نهایی اجرا خواهد شد. else نهایی بعنوان شرط پیش فرض عمل می کند، یعنی اگر کلیه شرایط دیگر صحیح نباشند، آنگاه آخرین دستور else انجام خواهد شد. اگر else نهایی وجود نداشته باشد و سایر شرایط ناصحیح باشند، آنگاه هیچ عملی انجام نخواهد گرفت.

در زیر، برنامه ای را مشاهده می کنید که از نردبان if-else-if استفاده کرده تا تعیین کند که یک ماه مشخص در کدام فصل واقع شده است.

```
// Demonstrate if-else-if statement.
class IfElse{
    public static void main(String args){
        int month = 4; // April
        String season;

        if(month == 12 || month == 1 || month == 2)
            season = "Winter";
        else if(month == 3 || month == 4 || month == 5)
            season = "Spring";
        else if(month == 6 || month == 7 || month == 8)
            season = "Summer";
        else if(month == 9 || month == 10 || month == 11)
            season = "Autumn";
        else
            season = "Bogus Month";

        System.out.println("April is in the" + season + ".");
    }
}
```

خروجی این برنامه بقرار زیر می باشد :

April is in the Spring.

ممکن است بخواهید این برنامه را آزمایش کنید. خواهید دید که هیچ فرقی ندارد که چه مقداری به month بدهید ، يك و فقط يك دستور انتساب داخل نردبان اجرا خواهد شد.

## Switch

دستور switch ، دستور انشعاب چند راهه در جاوا است. این دستور راه ساده ای است برای تغییر مسیر اجرای بخشهای مختلف يك کد براساس مقدار يك عبارت می باشد. این روش يك جایگزین مناسب تر برای مجموعه های بزرگتر از دستورات if-else-if است. شکل کلی دستور switch به قرار زیر می باشد :

```
switch(expression) {  
  case value1:  
    //statement sequence  
    break;  
  case value2:  
    //statement sequence  
    break;  
  .  
  .  
  .  
  case valueN:  
    //statement sequence  
    break;  
  default:  
    //default statement sequence  
}
```

expression باید از نوع byte ، short ، int یا char باشد ، هر يك از مقادیر values در دستورات case باید از نوع سازگار با expression باشند. هر يك از مقادیر case باید يك لیترال منحصر بفرد باشد ( یعنی باید يك ثابت ، نه متغیر ، باشد ). مقادیر case نباید تکراری باشند. دستور switch بشرح فوق عمل می کند : مقدار expression با هر يك از مقادیر لیترال عبارت case مقایسه می شوند. اگر مقدار متناظری پیدا شود ، کد سلسله ای تعقیب کننده



آن دستور case اجرا خواهد شد. اگر هیچیک از ثابت ها با مقدار expression برابر نباشند ، آنگاه دستور پیش فرض (default) اجرا خواهد شد ، اما دستور default اختیاری است. اگر هیچیک از case ها تطابق نیابد و default وجود نداشته باشد آنگاه عمل اضافی دیگری انجام نخواهد شد.

از عبارت break داخل دستور switch استفاده شده تا سلسله يك دستور را پایان دهد. هنگامیکه با يك دستور break مواجه می شویم ، اجرا به خط اول برنامه که بعد از کل دستور switch قرار گرفته ، منشعب خواهد شد. بدین ترتیب کنترل از عبارت switch خارج میشود.

در زیر مثال ساده ای را مشاهده می کنید که از دستور switch استفاده نموده است :

```
//A simple example of the switch.
class SampleSwitch{
    public static void main ( String args[] ) {
        for(int i=0; i<6; i++ )
            switch(i){
                case 0:
                    System.out.println("i is zero.");
                    break;
                case 1:
                    System.out.println("i is one.");
                    break;
                case 2:
                    System.out.println("i is two.");
                    break;
                case 3:
                    System.out.println("i is three.");
                    break;
                default:
                    System.out.println("i is greater than 3.");
            }
    }
}
```

خروجی این برنامه بقرار زیر می باشد :

```
i is zero.
i is one.
i is two.
```

```
i is three.  
i is greater than 3.  
i is greater than 3.
```

همانطوریکه مشاهده می کنید ، داخل حلقه ، دستوراتی که همراه ثابت case بوده و با i مطابقت داشته باشند ، اجرا خواهند شد. سایر دستورات پشت سر گذاشته می شوند. بعد از اینکه i بزرگتر از ۳ بشود ، هیچ يك از مقادیر case برابر مقدار expression نخواهد بود ، و بنابراین عبارت default اجرا می شود.

دستور default اختیاری است. اگر break را حذف کنید ، اجرای برنامه با case بعدی ادامه خواهد یافت. گاهی بهتر است چندین case بدون دستورات break در بین آنها داشته باشیم. بعنوان مثال ، برنامه بعدی را در نظر بگیرید :

```
//In a switch/ break statements are optional.  
class MissingBreak{  
    public static void main(String args[] ){  
        for(int i=0; i<6; i++ )  
            switch(i){  
                case 0:  
                case 1:  
                case 2:  
                case 3:  
                case 4:  
                    System.out.println("i is less than 5");  
                    break;  
                case 5:  
                case 6:  
                case 7:  
                case: ^  
                case 9:  
                    System.out.println("i is less than 10");  
                    break;  
                default:  
                    System.out.println("i is 10 or more");  
            }  
    }  
}
```

خروجي اين برنامه بقرار زير خواهد بود :

```
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

همانطوریکه مشاهده می کنید، اجرا طی هر case ، به محض رسیدن به یک دستور break (یا انتهای switch) متوقف می شود.

در حالیکه مثال قبلی برای توصیف نظر خاصی طراحی شده بود ، اما بهر حال حذف دستور break کاربردهای عملی زیادی در برنامه های واقعی دارد. برای نشان دادن کاربردهای واقعی تر این موضوع ، دوباره نویسی برنامه نمونه مربوط به فصول سال را مشاهده نمایید. این روایت جدید همان برنامه قبلی از switch استفاده می کند تا پیاده سازی موثرتری را ارائه دهد.

```
//An improved version of the season program.
class Switch{
    public static void main(String args[]){
        int month = 4;
        String season;
        switch(month){
            case 12:
            case 1:
            case 2:
                season = "Winter";
                break;
            case 3:
            case 4:
            case 5:
                season = "Spring";
```

```

        break;
    case 6:
    case 7:
    case 8:
        season = "Summer";
        break;
    case 9:
    case 10:
    case 11:
        season = "Autumn";
        break;
    default:
        season = "Bogus Month";
    }
    System.out.println("April is in the" + season + ".");
}
}

```

## عبارات Switch تو در تو

می توانید از يك switch بعنوان بخشی از ترتیب يك دستور switch خارجی تر استفاده نمایید. این حالت را switch تو در تو می نامند. از آنجاییکه دستور switch تعریف کننده بلوک مربوط به خودش می باشد، هیچ تلاقی بین ثابتهای case در switch داخلی و آنهایی که در switch خارجی قرار گرفته اند، بوجود نخواهد آمد. بعنوان مثال، قطعه بعدی کاملاً معتبر است.

```

switch(count){
    case 1:
        switch(target){ // nested switch
            case 0:
                System.out.println("target is zero");
                break;
            case 1:// no conflicts with outer switch
                System.out.println("target is one");
                break;
        }
    }
}

```

```
break;  
case 2: // ...
```

در اینجا دستور case 1 در switch داخلی با دستور case 1 در switch خارجی تلاقی نخواهد داشت. متغیر count فقط با فهرست case ها در سطح خارجی مقایسه می شود. اگر count برابر ۱ باشد، آنگاه target با فهرست case های داخلی مقایسه خواهد شد. به طور خلاصه، سه جنبه مهم از دستور switch قابل توجه هستند:

۱ - switch با if متفاوت است چون switch فقط آزمایش کیفیت انجام می دهد، در حالیکه if هر نوع عبارت بولی را ارزیابی می کند. یعنی که switch فقط بدنبال یک تطابق بین مقدار عبارت و یکی از ثابت های case خودش می گردد.

۲ - دو ثابت case در switch نمی توانند مقادیر یکسان داشته باشند. البته، یک دستور switch قرار گرفته داخل یک switch خارجی تر می تواند ثابت های case مشترک داشته باشد.

۳ - یک دستور switch معمولاً بسیار کارآمدتر از یک مجموعه از if های تودرتو شده است.

آخرین نکته بخصوص جالب توجه است زیرا روشنگر نحوه کار کامپایلر جاوا می باشد. کامپایلر جاوا هنگامیکه یک دستور switch را کامپایل می کند، به هر یک از ثابت های case سرکشی نموده و یک جدول jump table می سازد که برای انتخاب مسیر اجرا براساس مقدار موجود در عبارت استفاده می شود. بنابراین، اگر باید از میان گروه بزرگی از مقادیر انتخاب نمایید، یک دستور switch نسبت به یک ترتیب از if-else ها که بطور معادل و منطقی کد بندی شده باشد، بسیار سریعتر اجرا خواهد شد. کامپایلر قادر است اینکار را انجام دهد چون می داند که ثابت های case همه از یک نوع بوده و باید خیلی ساده با عبارت switch برای کیفیت مقایسه شوند. کامپایلر چنین شناسایی را نسبت به یک فهرست طولانی از عبارات if ندارد.

## دستورات تکرار ( Iteration Statements )

دستورات تکرار در جاوا عبارتند از for، while و do-while. این دستورات آن چه را ما حلقه می نامیم، ایجاد می کنند. احتمالاً می دانید که حلقه یک مجموعه از دستورالعملها را بطور تکراری

اجرا مي كند. تا اينكه يك شرط پاياني را ملاقات نمايد. همانطوريكه بعدا خواهيدديد، جاوا حلقه اي دارد كه براي كليۀ نيازهاي برنامه نويسي مناسب است.

## While

حلقه while اساسي ترين دستور حلقه سازي (looping) در جاوا است. اين دستور ماداميكه عبارت كنترل كننده ، صحيح (true) باشد، يك دستور يا يك بلوك را تكرر مي كند. شكل كلي اين دستور بقرار زير است :

```
while( condition ){  
    //body of loop  
}
```

شرط يا condition ممكن است هر عبارت بولي باشد. ماداميكه عبارت شرطي صحت داشته باشد ، بدنه حلقه اجرا خواهد شد . هنگاميكه شرط صحت نداشته باشد ، كنترل بلافاصله به خط بعدي كدي كه بلافاصله پس از حلقه جاري قرار دارد ، منتقل خواهد شد. اگر فقط يك دستور منفرد در حال تكرر باشد ، استفاده از ابروها غير ضروري است .

در اينجا يك حلقه while وجود دارد كه تا ۱۰ را محاسبه كرده و دقيقاً ده خط "tick" را چاپ مي كند.

```
//Demonstrate the while loop.  
class While {  
    public static void main(String args[]){  
        int n = 10;  
        while(n > 0 ){  
            System.out.println("tick" + n );  
            n--;  
        }  
    }  
}
```

هنگاميكه اين برنامه را اجرا مي كنيد، ده مرتبه "tick" را انجام خواهد داد :

```
tick 10  
tick 9  
tick 8  
tick 7
```

```
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

از آنجاییکه حلقه `while` عبارت شرطی خود را در بالای حلقه ارزیابی میکند ، اگر شرط ابتدایی ناصحیح باشد ، بدنه حلقه اجرا نخواهد شد. بعنوان مثال ، در قطعه زیر ، فراخوانی `println()` هرگز اجرا نخواهد شد.

```
int a = 10, b = 20;
while(a > b)
System.out.println("This will not be displayed");
```

بدنه `while` یا هر حلقه دیگر در جاوا ممکن است تهی باشد. زیرا دستور تهی دستوری که فقط شامل ؛ باشد ، از نظر قواعد ترکیبی در جاوا معتبر است. بعنوان مثال ، برنامه زیر را در نظر بگیرید :

```
//The target of a loop can be empty.
class NoBody {
public static void main(String args[]){
int i/ j;
i = 100;
j = 200;
//find midpoint between i and j
while(++i < --j); // no body in this loop
System.out.println("Midpoint is" + I );
}
}
```

این برنامه نقطه میانی (midpoint) بین `i` و `j` را پیدا می کند و خروجی زیر را تولید خواهد کرد :

```
Midpoint is 150
```

در اینجا چگونگی کار حلقه `while` را می بینید. مقدار `i` افزایش و مقدار `j` کاهش می یابد. سپس این دو مقدار با یکدیگر مقایسه می شوند. اگر مقدار جدید `i` همچنان کمتر از مقدار جدید `j` باشد ، آنگاه حلقه تکرار خواهد شد . اگر `i` مساوی یا بزرگتر از `j` بشود ، حلقه متوقف خواهد شد. تا

هنگام خروج از حلقه ، i مقداری را می گیرد که بین مقادیر اولیه i و j می باشد. ( بدیهی است که این رویه هنگامی کار می کند که i کوچکتر از مقدار اولیه j باشد. ) همانطوریکه می بینید ، نیازی به بدنه حلقه نیست ، کلیه عملیات داخل خود عبارت شرطی اتفاق می افتد. در کدهای حرفه ای نوشته شده دیگر جاوا ، وقتی که عبارت کنترل کننده توانایی مدیریت کلیه جزئیات خود را داشته باشد ، حلقه های کوتاه غالباً بدون بدنه کد بندی می شوند.

## Do-While

گفتیم اگر عبارت شرطی کنترل کننده يك حلقه while در ابتدا ناصحیح باشد آنگاه بدنه حلقه اصلاً اجرا نمی شود. اما گاهی مایلیم در چنین شرایطی ، بدنه حلقه حداقل یکبار اجرا شود. بعبارت دیگر ، در حالات خاصی مایلید تا عبارت پایان دهنده در انتهای حلقه را آزمایش کنید. خوشبختانه ، جاوا حلقه ای را عرضه می کند که دقیقاً همین کار را انجام می دهد. حلقه do-while همواره حداقل یکبار بدنه خود را اجرا می کند ، زیرا عبارت شرطی آن در انتهای حلقه قرار گرفته است. شکل کلی آن بصورت زیر است :

```
Do {  
//body of loop  
}while(condition);
```

هر تکرار از حلقه do-while ابتدا بدنه حلقه را اجرا نموده ، سپس به ارزیابی عبارت شرطی خود می پردازد. اگر این عبارت صحیح (true) باشد ، حلقه اجرا خواهد شد. درغیراینصورت حلقه پایان می گیرد. نظیر کلیه حلقه های جاوا ، شرط باید يك عبارت بولی باشد. اینجا يك روایت دیگر از برنامه (tick) وجود دارد که حلقه do-while را نشان می دهد. خروجی این برنامه مشابه برنامه قبلی خواهد بود :

```
//Demonstrate the do-while loop.  
class DoWhile {  
public static void main(String args[]){  
int n = 10;  
do{  
System.out.println("tick" + n);  
n--;  
}while ( n > 0 );
```



```
}  
}
```

حلقه موجود در برنامه قبلي ، اگر چه از نظر تکنیکی صحیح است ، اما می توان آن را به شکل کار اتری بصورت زیر دوباره نویسی نمود :

```
do {  
System.out.println("tick " + n);  
} while (--n > 0);
```

در این مثال ، عبارت  $(--n > 0)$  ، عمل کاهش  $n$  و آزمایش برای صفر را در یک عبارت گنجانده است. عملکرد آن بقرار بعدی است. ابتدا دستور  $n$  اجرا می شود و  $n$  را کاهش داده و مقدار جدید را به  $n$  برمی گرداند. این مقدار سپس با صفر مقایسه می شود. اگر بزرگتر از صفر باشد ، حلقه ادامه می یابد. در غیر اینصورت حلقه پایان می گیرد.

حلقه `do-while` بویژه هنگام پردازش انتخاب منو بسیار سودمند است ، زیرا معمولاً مایلید تا بدنه یک حلقه منو حداقل یکبار اجرا شود. برنامه بعدی را که یک سیستم `Help` ساده را برای دستورات تکرار و انتخاب در جاوا پیاده سازی می کند در نظر بگیرید :

```
//Using a do-while to process a menu selection  
class Menu{  
public static void main(String args[])  
throws java.io.IOException{  
char choice;  
do {  
System.out.println("Help on:");  
System.out.println(" 1 .if");  
System.out.println(" 2 .switch");  
System.out.println(" 3 .while");  
System.out.println(" 4 .do-while");  
System.out.println(" 5 .for\n");  
System.out.println("Choose one:");  
choice =( char )System.in.read();  
} while(choice < '1' || choice > '5' );  
System.out.println("\n");  
switch(choice){  
case '1':  
System.out.println("The if:\n");  
System.out.println("if(condition( statement;");
```

```

System.out.println("else statement;");
break;
case '2':
System.out.println("The switch:\n");
System.out.println("switch(expression){");
System.out.println(" case constant:");
System.out.println(" statement sequence;("
System.out.println(" break;");
System.out.println(" //...");
System.out.println("}");
break;
case '3':
System.out.println("The switch:\n");
System.out.println(while(condition )statement; ");
break;
case '4':
System.out.println("The do-while:\n");
System.out.println("do {");
System.out.println(" statement; ");
System.out.println("} while( condition );");
break;
case '5':
System.out.println("The for:\n");
System.out.print("for(init; condition ;iteration)");
System.out.println(" statement;");
break;
}
}
}

```

اکنون يك اجراي نمونه توليد شده توسط اين برنامه را مشاهده مي كنيد :

Help on:

1. if
2. switch
3. while
4. do-while
5. for

Choos one:

4

The do-while:

```
do {  
statement;  
} while (condition);
```

در برنامه ، از حلقه do-while برای تصدیق اینکه کاربر يك گزینه معتبر را وارد کرده باشد ، استفاده می شود. در غیر اینصورت ، به کاربر مجددا اعلان خواهد شد. از آنجاییکه منو باید حداقل یکبار بنمایش درآید ، do-while لقه کاملی برای انجام این مقصود است. چند نکته دیگر درباره این مثال : دقت کنید که کاراکترها از صفحه کلید بوسیله فراخوانی system.in.read() خوانده می شوند. این یکی از توابع ورودی کنسول در جاوا است.

اگر چه بررسی تفصیلی روشهای I/O جاوا به بحثهای بعدی موكول شده ، اما از system.in.read() در اینجا برای بدست آوردن گزینه کاربر استفاده شده است. این تابع کاراکترها را از ورودی استاندارد می خواند. ( کاراکترها بعنوان عدد صحیح برگردانده می شوند ، و به همین دلیل از طریق casting به char تبدیل می شوند. ) وسیله ورودی طبق پیش فرض ، همچون یک بافر خطی است ، بنابراین برای ارسال کاراکترهای تاپی شده به برنامه ، می بایست کلید ENTER را فشار دهید.

کار کردن با کنسول ورودی جاوا می تواند قدری ناراحت کننده باشد. به علاوه بیشتر برنامه های واقعی و اپلت های جاوا ، گرافیکی و مبتنی بر پنجره خواهند بود. نکته دیگر اینکه چون از system.in.read() استفاده شده است ، برنامه باید از throws.java.io.IOException استفاده کند.

## for

خواهید دید که حلقه for يك ساختار قدرتمند و بسیار روان است. شکل کلی دستور for بصورت زیر است :

```
for(initialization; condition ;iteration; ){  
//body  
{
```

اگر فقط يك دستور باید تکرار شود ، نیازی به آکولادها نیست. عملکرد حلقه for بشرح بعدی است. وقتی که حلقه برای اولین بار شروع می شود مقدار دهی اولیه در حلقه اجرا می شود.

معمولا ، این بخش يك عبارت است که مقدار متغیر کنترل حلقه را تعیین می کند ، که بعنوان يك شمارشگر ، کنترل حلقه را انجام خواهد داد. مهم است بدانیم که عبارت مقدار دهی اولیه فقط یکبار اجرا می شود. سپس شرط مورد ارزیابی قرار می گیرد . این شرط باید يك عبارت بولی باشد. این بخش معمولا مقدار متغیر کنترل حلقه را با مقدار هدف مقایسه می کند. اگر عبارت صحیح (true) باشد، آنگاه بدنه حلقه اجرا خواهد شد. اگر ناصحیح باشد حلقه پایان می گیرد. بعد، بخش تکرار (iteration) حلقه اجرا می شود . این بخش معمولا عبارتی است که مقدار متغیر کنترل را افزایش یا کاهش می دهد. آنگاه حلقه تکرار خواهد شد ، ابتدا عبارت شرطی را ارزیابی می کند ، سپس بدنه حلقه را اجرا می کند و سرانجام عبارت تکرار را در هر گذر (pass) اجرا میکند. این روال آنقدر ادامه می یابد تا عبارت شرطی ناصحیح (false) گردد .

در زیر روایت جدیدی از برنامه "tick" را می بینید که از يك حلقه for استفاده کرده است :

```
//Demonstrate the for loop.
class ForTick {
public static void main(String args[]){
int n;
for(n=10; n>0; n--){
System.out.println("tick" + n);
}
}
```

### تعریف متغیرهای کنترل حلقه در حلقه for

غالبا متغیری که يك حلقه for را کنترل می کند ، فقط برای همان حلقه مورد نیاز بوده و کاربرد دیگری ندارد. در چنین حالتی ، می توان آن متغیر را داخل بخش مقدار دهی اولیه حلقه for اعلان نمود. بعنوان مثال در اینجا همان برنامه قبلی را مشاهده می کنید که متغیر کنترل حلقه یعنی n بعنوان يك int در داخل حلقه for اعلان شده است.

```
//Declare a loop control variable inside the for.
class ForTick {
public static void main(String args[]){
//here/ n is declared inside of the for loop
for(int n=10; n>0; n--)
```

```
System.out.println("tick" + n);
}
}
```

هنگامیکه يك متغیر را داخل يك حلقه for اعلان مي كنيد ، يك نکته مهم را بايد به ياد داشته باشید : قلمرو آن متغیر هنگامیکه دستور for انجام مي شود ، پايان مي يابد. ( يعني قلمرو متغیر محدود به حلقه for است. ) خارج از حلقه for حیات آن متغیر متوقف مي شود. اگر بخواهید از این متغیر کنترل حلقه در جاي دیگری از برنامه اتان استفاده کنید ، نباید آن متغیر را داخل حلقه for اعلان نمایید.

درشرایطی که متغیر کنترل حلقه جاي دیگری موردنیاز نباشد، اکثر برنامه نویسان جاوا آن متغیر را داخل for اعلان مي کنند. بعنوان مثال ، در اینجا يك برنامه ساده را مشاهده مي كنید که بدنبال اعداد اول مي گردد. دقت كنید که متغیر کنترل حلقه ، چون جاي دیگری مورد نیاز نیست ، داخل for اعلان شده است.

```
//Test for primes.
class FindPrime {
public static void main(String args[]){
int num;
boolean isPrime = true;
num = 14;
for(int i=2; i <= num/i; i++){
if((num % i )== 0 ){
isPrime = false;
break;
}
}
if(isPrime )System.out.println("Prime");
else System.out.println("Not Prime");
}
}
```

## استفاده از کاما Comma

شرایطی پیش می آید که مایلید بیش از يك دستور در بخش مقدار دهی اولیه initialization و تکرار (iteration) بگنجانید. بعنوان مثال ، حلقه موجود در برنامه بعدی را در نظر بگیرید :

```

class Sample {
public static void main(String args[]){
int a, b;
b = 4;
for(a=1; a<b; a++){
System.out.println("a = " + a);
System.out.println("b = " + b);
b--;
}
}
}

```

همانطوریکه می بینید ، حلقه توسط ارتباط متقابل دو متغیر کنترل می شود. از آنجاییکه حلقه توسط دو متغیر اداره می شود ، بجای اینکه b را بصورت دستی اداره کنیم ، بهتر است تا هر دو را در دستور for بگنجانیم. خوشبختانه جاوا راهی برای اینکار دارد برای اینکه دو یا چند متغیر بتوانند یک حلقه for را کنترل کنند ، جاوا به شما امکان می دهد تا چندین دستور را در بخشهای مقدار دهی اولیه و تکرار حلقه for قرار دهید. هر دستور را بوسیله یک کاما از دستور بعدی جدا می کنیم.

حلقه for قبلی را با استفاده از کاما ، خیلی کاراتر از قبل می توان بصورت زیر کد بندی نمود :

```

//Using the comma.
class Comma {
public static void main(String args[]){
int a, b;
for(a=1, b=4; a<b; a++, b--){
System.out.println("a = " + a);
System.out.println("b + " = b);
}
}
}

```

در این مثال ، بخش مقدار دهی اولیه ، مقادیر a و b و را تعیین می کند. هربار که حلقه تکرار می شود ، دو دستور جدا شده توسط کاما در بخش تکرار (iteration) اجرا خواهند شد. خروجی این برنامه بقرار زیر می باشد :

```

a=1
b=4

```

a=2

b=3

نکته : اگر با C++/C آشنایی دارید ، حتما می دانید که در این زبانها ، علامت کاما يك عملگر است که در هر عبارت معتبري قابل استفاده است. اما در جاوا اینطور نیست. در جاوا ، علامت کاما يك جدا کننده است که فقط در حلقه for قابل اعمال می باشد.

## برخی از اشکال حلقه for

حلقه for از تعدادي گوناگونيها پشتیبانی می کند که قدرت و کاربري آن را افزایش می دهند. دلیل انعطاف پذيري آن است که لزومي ندارد که سه بخش مقداردهي اولیه ، آزمون شرط و تکرار ، فقط براي همان اهداف مورد استفاده قرار گیرند. در حقیقت ، سه بخش حلقه for براي هر هدف مورد نظر شما قابل استفاده هستند. به چند مثال توجه فرمائید.

يکي از رایجترین گوناگونيها مربوط به عبارت شرطي است. بطور مشخص ، لزومي ندارد این عبارت ، متغیر کنترل حلقه را با برخی مقادیر هدف آزمایش نماید. در حقیقت ، شرط کنترل کننده حلقه for ممکن است هر نوع عبارت بولي باشد. بعنوان مثال ، قطعه زیر را در نظر بگیرید :

```
boolean done = false;
for(int i=1; !done; i++){
//...
if(interrupted ()) done=true;
}
```

در این مثال ، حلقه for تا زمانی که متغیر بولي done معادل true بشود ، اجرا را ادامه خواهد داد. این مثال مقدار i را بررسی نمی کند . اکنون يکي دیگر از گوناگونيهاي جالب حلقه for را مشاهده می کنید. ممکن است يکي یا هر دو عبارت مقدار دهی اولیه و تکرار قابل حذف باشند ، نظیر برنامه بعدي :

```
//Parts of the for loop can be empty.
class ForVar {
public static void main(String args[]){
int i;
boolean done = false;
i = 0;
for (;!done;){
```

```

System.out.println("i is" + i);
if(i == 10 )done = true;
i++;
}
}
}

```

در اینجا عبارتهای مقدار دهی اولیه و تکرار به خارج از for انتقال یافته اند. برخی از بخشهای حلقه for تهی هستند. اگر چه در این مثال ساده چنین حالتی هیچ ارزشی ندارد ، اما در حقیقت شرایطی وجود دارد که این روش بسیار کارا و سودمند خواهد بود. بعنوان مثال ، اگر شرط اولیه بصورت يك عبارت پیچیده و در جای دیگری از برنامه قرار گرفته باشد و یا تغییرات متغیر کنترل حلقه بصورت غیر ترتیبی و توسط اعمال اتفاق افتاده در داخل بدنه حلقه تعیین شود ، پس بهتر است که این بخشها را در حلقه for تهی بگذاریم . اکنون یکی دیگر از گوناگونیهای حلقه for را مشاهده می کنید. اگر هر سه بخش حلقه for را تهی بگذارید ، آنگاه بعد يك حلقه نامحدود ( حلقه ای که هرگز پایان نمی گیرد ) ایجاد کرده اید. بعنوان مثال :

```

for (;;) {
//...
}

```

این حلقه تا ابد ادامه خواهد یافت ، زیرا هیچ شرطی برای پایان گرفتن آن تعبیه نشده است. اگر چه برخی برنامه ها نظیر پردازشهای فرمان سیستم عامل مستلزم يك حلقه نامحدود هستند ، اما اکثر حلقه های نامحدود در واقع حلقه هایی هستند که ملزومات پایان گیری ویژه ای دارند. بزودی خواهید دید ، راهی برای پایان دادن به يك حلقه حتی يك حلقه نامحدود نظیر مثال قبلی وجود دارد که از عبارت شرطی معمولی حلقه استفاده نمی کند .

## شکل for-each از حلقه for

شکل دیگری از حلقه for به سبک زیر می باشد. مزیت این رویه آن است که کلمه کلیدی جدیدی لازم نیست و روتین های موجود نیز تغییر نمی یابند. به این سبک ، حلقه پیشرفته می گویند.

```

For ( type itr-var :collection) statement-block

```



Type مشخص کننده نوع ، int-var نام متغیر تکرار است که عناصر یک کلکسیون را یک به یک از ابتدا تا انتها دریافت می کند. کلکسیونی که چرخه ای برای آن ایجاد می شود collection نامیده می شود.  
به مثال زیر توجه کنید :

```
//use a for-each style for loop.
Class ForEach{
Public static void main (String args[]){
Int nums[] = {1,2,3,4,5,6,7,8,9,10};
Int su, = 0;
//use for-each style for to display and sum the values
for ( int s, nums ){
system.out.println("value is: " + x );
sum +=x;
}
system.out.println("summation : " + sum );
}
}
```

خروجی به صورت زیر است :

```
Valu is : 1
Valu is : 2
Valu is : 3
Valu is : 4
Valu is : 5
Valu is : 6
Valu is : 7
Valu is : 8
Valu is : 9
Valu is : 10
Summation : 55
```

**تکرار در آرایه های چند بعدی**

این کاربرد با یک مثال مشخص می شود.

```
//use for-each style for on a two-dimensional array.
Class foreacha {
Public static void main ( String args[]){
Int sum = 0;
Int nums[][] = new int[3][5];
//give nums some values
for ( int i = 0 ; i<3 ; i++ )
for ( int j = 0 ; j<5 ; j++ )
nums[i][j] = (i+1)*(j+1);
//use for-each for to display and sum the values
for ( int x[] : nums ){
for ( int y : x )
system.out.println("value is: " + y );
sum += y;
}
}
system.out.println("Summation: ") + sum);
}
}
```

خروجي به صورت زیر است :

```
Valu is : 1
Valu is : 2
Valu is : 3
Valu is : 4
Valu is : 5
Valu is : 2
Valu is : 4
Valu is : 6
Valu is : 8
Valu is : 10
Valu is : 3
Valu is : 6
Valu is : 9
Valu is : 12
Valu is : 15
Summation : 90
```

## حلقه هاي تودرتو

نظير كليۀ زبانهاي برنامه نويسي ، جاوا نيز امکان تودرتو کردن حلقه ها را دارد. يعني يك حلقه داخل حلقه ديگري قرار خواهد گرفت. بعنوان مثال ، در برنامه بعدي حلقه هاي for تودرتو نشده اند :

```
//Loops may be nested.
class Nested {
public static void main(String args[]){
int i, j;
for(i=0; i < 10;i++){ ++
for(j=i; j < 10;j(++
System.out.print;("."))
System.out.println;)(
}
}
}
```

## عبارات پرش

جاوا از سه عبارت پرش break ، continue ، return پشتیباني میکند. اين عبارات ، کنترل را به بخش ديگري از برنامه تان انتقال مي دهند.

## Break

اين دستور چهار کاربرد دارد :

۱ – همانطور که دیدید به اجراي متوالي عبارات در يکي از حالات عبارت switch پايان مي بخشد.

۲ – استفاده از break براي خروج از حلقه ، با استفاده از break مي توانيد يك حلقه را فوراً پايان بخشيد و از جمله شرطي و مابقي عبارات موجود در قسمت body صرف نظر کنيد. وقتي نوبت به اجراي عبارت Break در يك حلقه مي شود ، اجراي حلقه به پايان رسيده و کنترل اجراي برنامه به نخستين عبارت پس از حلقه هدايت مي شود.

این دستور مختص قطع جریان حلقه نیست ، بلکه دستورات شرطی وظیفه این کار را دارند و در شرایط خاص قابل استفاده است.

۳ – استفاده از break به عنوان نوعی goto : جاوا فاقد عبارت goto است ، با وجود آنکه درک و نگهداشت روتین های goto دشوار است و از برخی از بهینه سازی های کامپایلر جلوگیری می کند ، جاوا سعی کرده است با این نوع از دستور break به گونه ای دیگر همان کار goto را انجام دهد. با اجرا این دستور از اجرای یک یا چند بلوک صرفنظر می شود. این کار برای هر بلوکی امکانپذیر است. ضمناً می توانید مشخص کنید اجرا از کجا متوقف شود. break مزایای goto را بدون مشکلات آن فراهم میکند. این دستور به فرم زیر استفاده می شود :

```
Break label;
```

Lable نشان دهنده بلوک مورد نظر می باشد.

```
//using break as a civilized form of goto.
Class Break{
    Public static void main ( String args[] ){
        Boolean t = true;
        first: {
            second: {
                third: {
                    system.out.println("Before the break.");
                    if(t) break second; // break out of second block
                    system.out.println("This won't execute");
                }
                system.out.println("This won't execute");
            }
            system.out.println("This is after second block.");
        }
    }
}
```

خروجی به فرم زیر است :

```
Before the break.
This is after second block.
```

۴ - یکی دیگر از کاربردها خروج از حلقه های تو در توست.

```
// using Break to exit from nested loops
class BreakLoop4{
```

```

public static void main( String args[]){
    outer: for( int i=0; i<3; i++ ){
        System.out.print("Pass " + I + ":" );
        for ( int j=0; j<100; j++ ){
            if ( j==10 ) break outer;    // exit both loops
            System.out.print(j + " ");
        }
        System.out.println("This will not print");
    }
    System.out.println("Loops complete");
}
}

```

خروجي از قرار زیر است :

```

Pass 0 : 0 1 2 3 4 5 6 7 8 9 loops complete.

```

## Continue

گاهی لازم می شود اجرای حلقه پیش از رسیدن به انتها تکرار شود. یعنی ممکن است بخواهید اجرای حلقه ادامه باید ، اما پردازش مابقی عبارات قسمت body آن برای تکرار جاری متوقف شود. این عبارت در حلقه های while و do-while سبب انتقال مستقیم کنترل به جمله شرطی کنترل کننده حلقه میشود. در حلقه for کنترل نخست به قسمت iteration و سپس به جمله شرطی عبارت for هدایت می شود. در هر سه حلقه ، عبارات پس از continue نادیده انگاشته میشود.

```

// Demonstrate continue.
Class Continue {
    Public static void main ( String args[] ){
        for ( int i=0; i<10; i++ ){
            System.out.print(i + " ");
            if ( i%2 == 0 ) continue;
            System.out.println(" ");
        }
    }
}

```

با استفاده از عملگر % بررسی می شود که مقدار I زوج است یا خیر. اگر چنین باشد ، حلقه بدون چاپ کاراکتر سطر جدید ادامه می یابد. خروجی مانند زیر است :

0 1  
2 3  
4 5  
6 7  
8 9

مانند دستور break می توان از برجسب نیز استفاده کرد.

کاربردهای درست برای عبارت break نادر است چون جاوا مجموعه غنی از عبارات ایجاد حلقه دارد که برای بیشتر کاربردها مناسب می باشند ، اما برای شرایط ویژه ای که تکرار زودتر از حد تعیین شده حلقه نیاز باشد ، عبارت continue روش ساختاریافته ای برای انجام این کار فراهم نموده است.

## Return

برای بازگشت صریح از يك متد به کار می رود. یعنی سبب بازگرداندن کنترل اجرا به برنامه فراخوان متد می شود. بدین ترتیب ، این عبارت به عنوان پرش طبقه بندی شده است. با استفاده از عبارت return در متدها می توان کنترل اجرا را در هر لحظه به روتین فراخوان متد بازگرداند. از این رو اجرای متدی که در آن اجرا می شود را فوراً پایان می دهد.

## استفاده از استثنائات

اداره کردن استثنائات يك مکانیسم قدرتمند برای کنترل برنامه های پیچیده فراهم آورده که دارای چندین ویژگی پویای حین اجرا است . مهم این است که throw ، ، throws ، finally و catch را بعنوان شیوه های واضح برای اداره خطاها و شرایط محدوده غیر طبیعی در منطق برنامه هایتان در نظر بگیرید. اگر مثل اکثر برنامه نویسان باشید، آنگاه وقتی يك روش عقیم می شود، احتمالاً سعی می کنید تا يك کد خطا را برگردانید . وقتی در جاوا مشغول برنامه نویسی هستید ، باید این عادت را کنار

بگذارید. وقتی يك روش مي تواند عقيم باشد، بهتر است يك استثنائ پرتاب نماييد . اين شيوه  
بمراتب بهتري براي اداره حالات عقيمي است.  
نکته : دستورات اداره كردن استثنائ در جاوا نبايد بعنوان يك مكانيسم عمومي براي انشعاب سازي  
غير محلي تلقي شود. اگر اينكار را انجام دهيد ، فقط برنامه را مغشوش کرده و نگهداري آنها را  
مشكل مي سازيد.

## اداره استثنائات

استثنا (exception) يك شرايط غير طبيعي است كه در زمان اجرا در بين مراحل مختلف يك كد  
حادث مي شود . بعبارت ديگر استثنائ يك خطاي حين اجرا است . در زبانهاي كامپيوتري كه  
اداره استثنا را پشتيباني نمي كنند ، خطاها بايد بصورت دستي كنترل و اداره شوند معمولا از  
طريق كدهاي خطا (error codes) و غيره . اين شيوه بسيار طاقت فرسا و مشكل آفرين است .  
اداره استثنا در جاوا از بروز اين مشكلات در پردازش جلوگیری کرده و مدیریت خطاي حين اجرا  
را در دنياي شي گرايي نمايد.

## اصول اداره استثنا

يك استثنا در جاوا، شيئي است كه يك شرط استثنائي ( يعني يك خطا ) را كه در قطعه اي از كد  
حادث شده ، توصيف مي كند . وقتي يك شرط استثنائ ايجاد مي شود يك شي كه آن استثنائ را  
معرفي مي كند ايجاد شده و در روشي كه آن خطا را ايجاد نموده ، پرتاب مي شود . (Thrown)  
آن روش ممكن است استثنائ را خودش اداره نمايد و يا از آن گذر كند . در هر صورت ، در نقطه  
اي استثنائ گرفته شده (caught) و پردازش مي شود . استثنائات ممكن است توسط سيستم حين  
اجراي جاوا توليد شوند ، و يا امكان دارد بصورت دستي توسط كدهاي شما بوجود آيند . استثنائات  
پرتاب شده توسط جاوا با خطاهاي اصلي كه از قوانين زبان جاوا تخطي ميكنند و يا محدوديتهاي  
محيط اجرايي جاوا را زير پا مي گذارند ، مرتبط هستند . استثنائات توليد شده دستي نوعا براي  
گزارش نمودن برخي شرايط خطا به فراخواننده يك روش استفاده مي شوند.

اداره استثنا در جاوا توسط پنج واژه کلیدی اعمال می شود: try ، catch ، throw ، throws و finally. بطور خلاصه عملکرد آنها را توضیح می دهیم . دستورات برنامه ای که مایلید برای استثنائات نشان دهید داخل یک بلوک try گنجانده میشوند. اگر داخل این بلوک یک استثنای حادث شود ، پرتاب خواهد شد . کد شما می تواند این استثنا را توسط catch گرفته و آن را بروشی منطقی اداره نماید. استثنائات تولید شده توسط سیستم بطور خودکار توسط سیستم حین اجرای جاوا پرتاب می شوند .

برای اینکه یک استثنای را بصورت دستی پرتاب کنیم ، از واژه کلیدی throw استفاده می کنیم . هر استثنایی که بیرون از یک روش پرتاب می شود باید توسط یک جمله throws مشخص شود . هر کدی که باید کاملاً " قبل از برگردانهای یک روش اجرا شود در یک بلوک finally قرار داده می شود. شکل عمومی یک بلوک اداره استثنای بصورت زیر می باشد:

```
try {  
    // block of code to monitor for errors  
}  
catch( Exception Type1 exOb ){  
    // exception handler for Exception Type1  
}  
catch( Exception Type2 exOb ){  
    // exception handler for Exception Type2  
}  
//...  
finally {  
    // block of code to be executed before try block ends  
}
```

استثنایی است که حادث شده است.

## انواع استثنا

کلیه انواع استثنا زیر کلاسهای از کلاس توکار throwable می باشند. بنابراین throwable در بالای سلسله مراتب کلاس استثنای (exception) قرار دارد . بلافاصله پس از throwable دو



زیر کلاس وجود دارند که استثنائات را به دو شاخه مجزا تقسیم می کنند ، سر عنوان يك شاخه Exception است . این کلاس برای شرایط استثنایی که برنامه های کاربر باید بگیرد ، استفاده می شود . همچنین از این کلاس ، زیر کلاسی می سازید تا انواع استثنای سفارشی خودتان را ایجاد نمایید . يك زیر کلاس با اهمیت از Exception تحت عنوان RuntimeException وجود دارد . استثنائات این نوع برای برنامه هایی که شما می نویسید و مواردی نظیر " تقسیم بر صفر " و " نمایه سازی غیر معتبر آرایه " را در آن می گنجانید ، بطور خودکار تعریف می شود . شاخه دیگر تحت عنوان Error است که استثنائاتی را تعریف می کند که انتظار نداریم . تحت شرایط عادی توسط برنامه شما گرفته شوند . استثنائات از نوع Error توسط سیستم حین اجرا یا برای نشان دادن خطاهایی که با خود محیط حین اجرای جاوا سر و کار دارند ، استفاده می شود . سرریزی پشته نمونه ای از این خطاهاست . استثنائات نوع Error نوعا در پاسخ به شکستهای مصیبت باری که معمولا توسط برنامه اره آنها نیست ، بوجود می آیند .

### استثنائات گرفته نشده Uncaught Exceptions

قبل از اینکه بیاموزید که چگونه استثنائات را در برنامه تان اداره نمایید بهتر است که بفهمید در صورت عدم اداره استثنائات چه اتفاقی می افتد . برنامه ساده بعدی دربرگیرنده يك عبارت است که بعمد باعث يك خطای تقسیم بر صفر می شود .

```
class Exc0 {
public static void main(String args[] ){
int d = 0;
int a = 42 / d;
}
}
```

وقتی سیستم حین اجرای جاوا تلاش خود برای انجام تقسیم بر صفر را آشکار میسازد يك شی ، استثنا جدیدی ساخته و سپس آن را پرتاب می کند . (throws) این کار باعث توقف اجرای EXC0 می گردد ، زیرا یکبار که يك استثنا پرتاب شود ، باید توسط يك اداره کننده استثنا گرفته شده (caught by an exception handler) و بلافاصله برای آن کاری انجام گیرد . در این مثال

، ما اداره كننده استثنا خودمان را عرضه نكرده ايم ، بنابر اين استثنائ توسط اداره كننده پيش فرض فراهم آمده بوسيله سيستم حين اجراي جاوا گرفته شده است . هر استثنائي كه توسط برنامه شما گرفته نشود ، در نهايت توسط اداره كننده پيش فرض پردازش خواهد شد . اداره كننده پيش فرض ، يك رشته كه استثنائ را توصيف نموده نمايش مي دهد ، يك ردياب پشته (stack tracer) از نقطه اي كه در آن استثنائ اتفاق افتاده چاپ مي نمايد و برنامه را ختم مي كند .

اين برنامه هنگاميكه مثال فوق توسط مفسر حين اجراي جاوا JDK اجرا مي شود حاصل زير را توليد مي كند :

```
java.lang.ArithmeticException :/ by zero
at Exc0.main(Exc0.java:4)
```

توجه فرماييد كه چگونه نام كلاس ، EXC0، نام روش ، main، نام فايل ، EXC0.java و شماره خط ، 4 ، همگي در رديابي ساده اين پشته گنجانده شده اند . همچنين دقت نماييد كه نوع استثنا پرتاب شده يك زير كلاس از Exception تحت عنوان Arithmetic Exception مي باشد كه خيلي دقيق توضيح مي دهد كه چه نوع خطايي اتفاق افتاده است . همانطوريكه قبلا توضيح داديم ، جاوا چندين نوع استثنائ تو كار را عرضه مي كند كه با انواع گوناگون خطاهاي حين اجرا كه احتمالا توليد ميشوند، مطابقت مي يابند .

ردياب پشته هميشه سلسله فراخوانيهاي روش كه منجر به بروز خطا شده اند را نمايش مي دهد . بعنوان مثال ، يك روايت ديگر از مثال قبلي را مشاهده مي كنيد كه همان خطا را معرفي مي كند اما در روشي جدا از main() :

```
class Exc1 {
static void subroutine () ({
int d = 0;
int a = 10 / d;
public static void main(String args[] ){
Exc1.subroutine) (;
}
}
```

ردیاب رشته بدست آمده از اداره کننده پیش فرض استثنای نشان می دهد که چگونه پشته فراخوانی کل بنمایش درآمده است:

```
java.lang.ArithmeticException :/ by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

همانطوریکه می بینید ، پایین پشته خط ۷ روش main وجود دارد که تحت عنوان Subroutine است و سبب بروز خطا در خط ۴ شده است. پشته فراخوان برای اشکال زدایی (debugging) کاملاً مفید است ، زیرا سلسله دقیق مراحل را که منجر به خطا شده اند.

### استفاده از try و catch

اگرچه اداره کننده پیش فرض استثنا فراهم شده توسط سیستم حین اجرای جاوا برای اشکال زدایی مفید است ، اما معمولاً بدنال آن هستید تا یک استثنا خودتان اداره نمایید. انجام این کار دو مزیت دارد. اول اینکه امکان تثبیت خطا را دارید. دوم اینکه ، اینکار مانع ختم خودکار برنامه خواهد شد . اگر هر بار که در برنامه شما خطایی بروز می کند ، برنامه اتان متوقف شده و یک ردیاب پشته چاپ نماید ، آنگاه اکثر کاربران برنامه اتان گیج و سردرگم میشوند. خوشبختانه جلوگیری از چنین حالتی بسیار ساده است.

برای جلوگیری از این وضعیت و اداره یک خطای حین اجرا ، خیلی ساده کدی را که می خواهید نمایش دهید (monitor) داخل یک بلوک try قرار دهید. بلافاصله بعد از این بلوک ، یک جمله catch قرار دهید که نوع استثنائی را که مایلید بگیرد مشخص می کند . برای اینکه سهولت اینکار را نشان دهیم ، برنامه بعدی را نگاه کنید که دربرگیرنده یک بلوک try و یک جمله catch میباشد که ArithmeticException تولید شده توسط خطای تقسیم بر صفر را پردازش می کند .

```
class Exc2 {
public static void main(String args[] ){
int d/ a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
```

```

System.out.println("This will not be printed.");
} catch( ArithmeticException e ){ // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}

```

خروجی این برنامه بقرار زیر می باشد :

```

Division by zero.
After catch statement.

```

دقت نمایید که فراخوانی `println()` داخل بلوک `try` هرگز اجرا نمی شود. هر بار که یک استثنای پرتاب می شود، برنامه ، انتقالات خارج از بلوک `try` به بلوک `catch` را کنترل می کند . بعبارت بهتر ، `catch` فراخوانی نمی شود بنابراین اجرا هرگز از یک `catch` به بلوک `try` بر نمی گردد . بنابراین ، خط "This will not be printed" بنمایش در نمی آید . هر بار که دستور `catch` اجرا شود ، کنترل برنامه با خط بعدی برنامه تعقیب کننده مکانیسم کل `try/catch` ادامه خواهد یافت .

یک دستور `try` و `catch` و یک واحد (unit) تشکیل می دهند . قلمرو جمله `catch` محدود شده به آن دستوراتی است که بلافاصله قبل از دستور `try` مشخص شده اند . یک دستور `catch` نمی تواند استثنای پرتاب شده توسط یک دستور دیگر `try` را بگیرد ، مگر در حالت دستورات تودرتو شده `try` که باختصار توضیح داده ایم . دستوراتی که بوسیله `try` محافظت می شوند . باید توسط ابروها احاطه شوند . ( یعنی آنها باید داخل یک بلوک قرار گیرند . ) نمی توانید از `try` روی یک دستور منفرد استفاده نمایید . این تغییر در روایت 1.0.2 از JDK ز معرفی شده است .

هدف اکثر جملات خوش ساخت `catch` باید این باشد که شرط استثنایی را از سر گرفته و سپس طوری برنامه را ادامه دهد که گویا خطا هرگز اتفاق نیفتاده است . بعنوان مثال ، در برنامه بعدی هر تکرار حلقه `for` دو عدد صحیح تصادفی را کسب می کند . آن دو عدد صحیح بر یکدیگر تقسیم شده و جواب آن برای تقسیم عدد ۱۲۳۴۵ استفاده می شود . جواب نهایی در `a` قرار می گیرد . اگر هر یک از عملیات تقسیم منجر به خطای " تقسیم بر صفر " شود ، آن خطا گرفته شده و مقدار `a` برابر صفر قرار گرفته و برنامه ادامه می یابد .

```
// Handle an exception and move on.
import java.util.Random;
class HandleError {
public static void main(String args[] ){
int a=0/ b=0/ c=0;
Random r = new Random() ;
for(int i=0; i<23000; i++ ){
try {
b = r.nextInt() ;
c = r.nextInt() ;
a = 12345 /( b/c);
} catch( ArithmeticException e ){
System.out.println("Division by zero.");
a = 0; // set a to zero and continue
}
System.out.println("a :" + a);
}
}
}
```

### نمایش توصیفی از یک استثنا

Throwable روش `toString()` ، (بوسیله `object` تعریف شده ) را لغو میکند ، بطوریکه یک رشته دربرگیرنده توصیفی از استثنای را برمی گرداند. می توانید خیلی ساده با گذر دادن استثنا بعنوان یک آرگومان ، این توصیف را در یک دستور `println()` به نمایش بگذارید. بعنوان مثال ، بلوک `catch` در برنامه قبلی را می توان بصورت زیر دوباره نویسی نمود .

```
catch( ArithmeticException e ){
System.out.println("Exception :" + e);
a = 0; / set a to zero and continue
}
```

وقتی این روایت را در برنامه جایگزین نمایید ، و برنامه تحت مفسر JDK جاوا اجرا شود ، هر خطای تقسیم بر صفر ، پیام بعدی را نمایش خواهد داد :

Exception : java.lang.ArithmeticException : / by zero

توانایی نمایش توصیفی از یک استثنا اگرچه در این متن ارزش خاصی ندارد، اما در سایر شرایط بسیار ارزشمند است بخصوص هنگام کار با استثنائات یا هنگام اشکال زدایی.

## جملات catch چند گانه

در بعضی مواقع از یک قطعه کوچک کد بیش از یک استثنا بوجود می آید . برای اداره چنین شرایطی ، می توانید دو یا چند جمله catch مشخص نمایید که هر کدام یک نوع متفاوت از استثنای بگیرند . وقتی یک استثنای پرتاب می شود ، هر دستور catch بترتیب امتحان می شود، و اولین دستوری که نوع آن با نوع استثنا مطابقت داشته باشد ، اجرا خواهد شد . بعد از اجرای یک دستور catch ، سایر دستورات پشت سر گذاشته می شوند و اجرای بعد از بلوک try/catch ادامه خواهد یافت . برنامه بعدی دو نوع استثنای مختلف را بدام می اندازد :

```
// Demonstrate multiple catch statements.
class MultiCatch {
public static void main(String args[] ){
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} catch(ArithmeticException e ){
System.out.println("Divide by 0 :" + e);
} catch(ArrayIndexOutOfBoundsException e ){
System.out.println("Array index oob :" + e);
}
System.out.println("After try/catch block.");
}
}
```

چون  $a$  مساوي صفر مي شود، اين برنامه اگر بدون پارامترهاي خط فرمان آغاز شود منجر به يك استثنا "تقسيم بر صفر" خواهد شد. اگر يك آرگومان خط فرمان بوجود آوريد و  $a$  را معادل مقداري بزرگتر از صفر قرار دهيد، برنامه شما تقسيم را نجات مي دهد. اما اين برنامه سبب `ArrayIndexOutOfBoundsException` مي گردد، چون آرايه  $C$  از نوع `int` داراي طول 1 است، همچنان برنامه تلاش مي کند تا مقداري را به  $C[42]$  منسوب نمايد. در اینجا خروجي توليد شده بوسيله اجراي هر دو راه را مشاهده مي کنيد:

```
C:\>java MultiCatch
a = 0
Divide by 0 :java.lang.ArithmeticException :/ by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1
Array index oob java.lang.ArrayIndexOutOfBoundsException :42
After try/catch blocks.
```

وقتي از دستورات `catch` چند گانه استفاده مي کنيد، مهم است بدانيد که زير کلاسهاي استثنائي بايد قبل از هر يك از کلاس بالاهاي مربوطه قرار گيرند. اين بدان دليل است که يك دستور `catch` که از يك کلاس بالا استفاده مي کند، استثنائات آن نوع و زير کلاسهاي آن نوع را خواهد گرفت. بنابراين، اگر يك زير کلاس بعد از کلاس بالايش بيابيد، هرگز به آن زير کلاس نمي رسد. بعلاوه، در جاوا، کد غير قابل دسترس نوعي خطا است. بعنوان مثال، برنامه بعدي را در نظر بگيريد:

```
/* This program contains an error.
A subclass must come before its superclass in
a series of catch statements. If not/
unreachable code will be created and a
compile-time error will result.
*/
class SuperSubCatch {
public static void main(String args[] ){
try {
int a = 0;
int b = 42 / a;
```

```

} catch(Eeception e ){
System.out.println("Generic Exception catch.");
}
/* This catch is never reached because
ArithmeticException is a subclass of Exception .*/
catch(ArithmeticException e ){ // ERROR - unreachable
System.out.println("This is never reached.");
}
}
}
}

```

اگر بخواهید این برنامه را کامپایل کنید ، يك پیغام خطا دریافت می کنید که میگوید دومین دستور catch غیرقابل رسیدن است . از آنجاییکه ArithmeticException يك زیرکلاس از Exception است ، اولین دستور catch کلیه خطاهای بر مبنای Exception از جمله ArithmeticException را اداره میکند. بدین ترتیب دومین دستور catch هرگز اجرا نخواهد شد ، برای برطرف کردن این مشکل ترتیب دو عبارت catch را تغییر دهید.

### دستورات تودرتو شده try

دستورات try را می توان تودرتو نمود . یعنی يك دستور try را می توان داخل بلوك يك try دیگر قرار داد. هر بار که يك دستور try وارد می شود ، متن آن عبارت روی پشته نشانده می شود. اگر يك دستور try داخلی تر فاقد يك اداره کننده catch برای يك استثنا خاص باشد، پشته دور زده نشده و اداره کننده catch مربوط به دستور بعدی try برای يك تطبیق مورد جستجو قرار می گیرد . این حالت تداوم می یابد تا اینکه یکی از دستورات catch موفق شود و یا تا زمانی که کلیه دستورات تودرتو شده try تمام شوند ، اگر هیچیک از دستورات catch با استثنا مطابقت نداشته باشند ، آنگاه سیستم حین اجرای جاوا خودش استثنای را اداره می کند . در زیر مثالی را مشاهده میکنید که از دستورات تودرتو شده try استفاده نموده است:

```

// An example of nested try statements.
class NestTry {
public static void main(String args[] ){
try {

```



```

int a = args.length;
/* If no command-line args are present/
the following statement will generate
a divide-by-zero exception .*/
int b = 42 / a;
System.out.println("a = " + a);
try { // nested try block
/* If one command-line arg is used/
then a divide-by-zero exception
will be generated by the following code .*/
if(a==1 ) a = a/(a-a); // division by zero
/* If two command-line args are used/
then generate an out-of-bounds exception .*/
if(a==2 ){
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e ){
System.out.println("Array index out-of-bounds :" + e)
}
} catch(ArithmeticException e ){
System.out.println("Divide by 0 :" + e);
}
}
}

```

همانطوریکه مشاهده می کنید، یک بلوک try را داخل دیگری جای می دهد . برنامه بصورت زیر کار می کند . وقتی برنامه را بدون آرگومانهای خط فرمان اجرا می کنید یک استثنا " تقسیم بر صفر " توسط بلوک خارجی تر try ایجاد می شود . اجرای برنامه توسط یک آرگومان خط فرمان یک استثنا " تقسیم بر صفر " را از داخل بلوک تودرتو شده try تولید می کند. چون بلوک داخلی تر این استثنای را نمی گیرد استثنا به بلوک خارجی تر try گذر داده می شود و در آنجا اداره خواهد شد . اگر برنامه را بدون آرگومانهای خط فرمان اجرا نمایید، یک استثنا " محدوده آرایه " از داخل بلوک داخلی تر try تولید خواهد شد. در زیر اجراهای نمونه ای وجود دارند که هر یک از حالات فوق را نشان می دهند:

```
C:\>java NestTry
Divide by 0 :java.lang.ArihmeticException :/ by zero
```

```
C:\>java NestTry One
a = 1
Divide by 0 :java.lang.ArihmeticException :/ by zero
```

```
C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException :42
```

اگر فراخوانی روش هم وجود داشته باشد، آنگاه تودرتو نمودن دستورات try ممکن است به روشهایی پنهان تر اتفاق بیفتد. بعنوان مثال، می توانید فراخوانی به یک روش را درون یک بلوک try انجام دهید. درون آن روش یک دستور دیگر try وجود دارد. در این حالت، try درون روش همچنان داخل یک بلوک خارجی تر try که روش را فراخوانی می کند تودرتو می شود. در زیر برنامه قبلی را مشاهده می کنید که در آن بلوک تودرتو شده try بداخل روش nesttry() نقل مکان کرده است.

```
/* Try statements can be implicitly nested via
calls to methods .*/
class MethNestTry {
static void nesttry(int a ){
try { // mested try block
/* If one command-line arg is used/
then a divide-by-zero exception
will be generated by the following code .*/
if(a==1 )a = a/(a-a); // division by zero
/* If two command-line args are used/
then generate an out-of-bounds exception .*/
if(a==2 ){
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e ){
System.out.println("Array index out-of-bounds : " + e)
```

```

}
}
public static void main(String args[] ){
try {
int a = args.length;
/* If no command-line args are present/
the following statement will generate
a divide-by-zero exception .*/
int b = 42 / a;
System.out.println("a = " + a);
nesttry(a);
} catch(ArithmeticException e ){
System.out.println("Divide by 0 :" + e);
}
}
}

```

خروجي ماند همان مثال قبلي است.

## Throw

تا بحال شما فقط گرفتن استثنائاتي كه توسط سيستم حين اجراي جاوا پرتاب شده را ديده ايد . اما برنامه شما مي تواند يك استثنائي را بطور صريح با استفاده از دستور throw پرتاب نمايد . شكل عمومي throw بقرار زير مي باشد:

```
ThrowThrowableInStance;
```

در اينجا Throwable Instance بايد يك شي از نوع throwable يا يك زير كلاس از throwable باشد . انواع ساده ، نظير int يا char ، همچون كلاسهاي غير از throwable نظير string و object و نميتوانند بعنوان استثنائات استفاده شوند . دو شيوه براي بدست آوردن يك شي throwable وجود دارد : استفاده از يك پارامتر داخل جمله catch و يا ايجاد يك شي ئ جديد با عملگر new . جريان اجرا بلافاصله بعد از دستور throw متوقف مي شود . دستورات بعدي اجرا نخواهند شد. نزديكترين بلوك بسته شده try مورد جستجو قرار مي گيرد تا يك دستور catch

پیدا شود که با نوع استثنا مطابقت داشته باشد. اگر مطابقت حاصل شود کنترل به آن دستور منتقل می شود. اگر مطابقت پیدا نشود، آنگاه دستور بعدی بسته شده try مورد جستجو قرار می گیرد، و همینطور الی آخر. اگر هیچ catch مطابقت کننده ای پیدا نشود، آنگاه اداره کننده پیش فرض استثنای مکتبی به برنامه داده و ردیاب پشته را چاپ می کند. در اینجا يك برنامه نمونه مشاهده می کنید که يك استثنا را ایجاد و پرتاب می نماید. اداره کننده ای که این استثنای را می گیرد، مجدداً آن را به اداره کننده بیرونی پرتاب خواهد نمود.

```
// Demonstrate throw.
class ThrowDemo {
    static void demoproc () {
        try {
            throw new NullPointerException("demo");
        } catch (NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch (NullPointerException e) {
            System.out.println("Recaught : " + e);
        }
    }
}
```

این برنامه دو شانس برای کار با يك خطای مشترک دارد. اول، main() يك متن استثنا تنظیم می کند و سپس demoproc() را فراخوانی می کند. روش demoproc() آنگاه يك متن دیگر اداره کننده استثنا را تنظیم نموده و بلافاصله يك نمونه جدید از NullPointerException را پرتاب می کند که روی خط بعدی گرفته خواهد شد. مجدداً استثنا پرتاب خواهد شد. حاصل بقرار زیر خواهد شد.

```
Caught inside demoproc
Recaught :java.lang.NullPointerException :demo
```

برنامه همچنین نشان می دهد چگونه یکی از اشیا استثنا استاندارد جاوا ایجاد می شود توجه بیشتری به این خط داشته باشید :

```
+ throw new NullPointerException("demo");
```

در اینجا ، new استفاده شده تا یک نمونه از NullPointerException ساخته شود . کلیه استثنائات حین اجرای توکار جاوا دو سازنده دارند: یکی بدون پارامتر و یکی با یک پارامتر رشته ای . وقتی از شکل دوم استفاده می شود ، آرگومان یک رشته را مشخص می کند که استثنا را توصیف می نماید . وقتی که شیء بعنوان یک آرگومان به print() یا println() استفاده می شود ، این رشته بنمایش درمی آید. این رشته را همچنین می توان بوسیله یک فراخوانی getMessage() که توسط Throwable توصیف شده بدست آورد .

## Throws

اگر یک روش ظرفیت ایجاد یک استثنای را دارد و آن را اداره نمی کند، آن روش باید این رفتار خود را مشخص نماید بگونه ای که فراخوانان آن روش خودشان را در مقابل استثنائات محافظت نمایند . اینکار با گنجانیدن یک جمله Throws در اعلان روش انجام میگیرد. یک جمله throws انواع استثنائاتی که یک روش ممکن است پرتاب نماید و فهرست بندی می کند . اینکار برای کلیه استثنائات ضروری است ، بغیر از انواع Error یا RuntimeException ، یا استثنائات مربوط به زیر کلاسهایی آنها . کلیه سایر استثنائاتی که یک روش می تواند پرتاب نماید ، باید در جمله throws اعلان شوند . اگر اینکار انجام نشود ، یک خطای (compile-time) حاصل خواهد شد . شکل عمومی یک اعلان روش که دربرگیرنده جمله throws باشد ، بصورت زیر است :

```
type method-name(parameter-list )throws exception-list
{
// body of method
}
```

در اینجا exception-list يك فهرست جدا شده با کاما از استثنائاتي است كه يك روش مي تواند پرتاب نمايد . مثال بعدي يك برنامه ناصحيح است كه تلاش مي كند يك استثنائ را كه نمي تواند بگيرد ، پرتاب نمايد. چون برنامه يك جمله throws را مشخص نمي كند تا اين حقيقت را اعلان نمايد ، برنامه كامپايل نخواهد شد :

```
// This program contains an error and will not compile.
class ThrowsDemo {
    static void throwOne ()({
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[] ){
        throwOne) (;
    }
}
```

براي كامپايل نمودن اين مثال ، بايد دو متغير بوجود آوريد . ابتدا ، لازم است اعلان كنيد throwOne() روشي است كه IllegalAccessException را پرتاب مي نمايد . دوم ، روش main() بايد يك دستور try/catch تعريف نمايد كه اين استثنا را بگيرد .

```
// This is now correct.
class ThrowsDemo {
    static void throwOne )(throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[] ){
        try {
            throwOne) (;
        } catch( IllegalAccessException e ){
            System.out.println("Caught " + e);
        }
    }
}
```

اجراي اين برنامه ، خروجي زير را توليد مي كند :

```
inside throwOne
```

```
caught java.lang.IllegalAccessException: demo
```

## Finally

وقتي استثنائات پرتاب مي شوند ، اجرا در روش تا حدي انقصال و قطعي پيدا مي كند مسير غير خطي كه جريان طبيعي روش را تغيير مي دهد . بسته به اينكه روش چگونه كد بندي شده باشد ، حتي امكان دارد كه يك استثنائ باعث شود تا روش نابهنگام برگردد . در برخي روشها چنين مشكلي ممكن است پيش آيد . بعنوان مثال اگر يك روش فايلي را هنگام ورود باز نموده و هنگام خروج آن را ببندد ، آنگاه شما ماييل نيستيد كدي كه فايل را مي بندد بوسيله مكانيسم اداره استثنائ پشت سر گذاشته شود. واژه كليدي finally طراحي شده تا اين احتمال وقوع را شناسايي كند . Finally يك بلوك كد ايجاد مي كند كه بعد از اينكه يك بلوك try/catch تكميل شده و قبل از كد تعقيب كننده بلوك try/catch اجرا خواهد شد . بلوك finally چه يك استثنائ پرتاب شود چه نشود ، اجرا خواهد شد . اگر يك استثنائي پرتاب شود بلوك finally اجرا خواهد شد حتي اگر هيچيك از دستورات catch با استثنائ مطابقت نداشته باشند .

هرگاه يك روش نزديك است كه از داخل يك بلوك try/catch به فراخواننده برگردد توسط يك استثنائ گرفته نشده يا كي دستور برگشت صريح ، جمله finally همچنين قبل از برگشتهاي روش اجرا خواهد شد . اينكار ممكن است براي بستن دستگيره هاي فايل و آزاد كردن ساير منابعي كه ممكن است در ابتدايي يك روش با هدف معين كردن آنها قبل از برگشت دادن بسيار مفيد باشد . جمله finally اختياري است . اما هر دستور try مستلزم حداقل يك جمله catch يا يك جمله finally مي باشد . در اينجا برنامه اي وجود دارد كه سه روش موجود در سه راه را نشان مي دهد كه هيچيك از آنها بدون اجراي جملات finally خود هستند .

```
// Demonstrate finally.
class FinallyDemo {
// Through an exception out of the method.
static void procA () {
try {
```

```

System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {
System.out.println("procA's finally");
}
}
// Return from within a try block.
static void procB (){
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
}
}
// Execute a try block normally.
static void procC (){
try {
System.out.println("inside procC");
} finally {
System.out.println("procC's finally");
}
}
public static void main(String args[] ){
try {
procA();
} catch( Eeception e ){
System.out.println("Exception caught");
}
procB();
procC();
}
}

```

در این مثال ، procA() با پرتاب يك استثنای، بطور نابهنگام try را مي شكند . جمله finally روي خارج راه اجرا مي شود . دستور try مربوط به procB() توسط يك دستور return اجرا



مي شود . جمله finally قبل از اينكه procB() برگشت نمايد اجرا مي شود . در procC() دستور try بطور طبيعي و بدون خطا اجرا مي شود . اما بلوك finally همچنان اجرا شده است . يادآوري : اگر يك بلوك finally با يك try همراه باشد ، بلوك finally براساس نتيجه try اجرا خواهد شد . در زير خروجي حاصل از برنامه قبلي را مشاهده مي كنيد :

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
```

## استثنائات توکار جاوا

داخل بسته نرم افزاري استاندارد java.lang ، جاوا چندين کلاس استثنائ را تعريف مي کند . چند تا از اين کلاسها در مثالهاي قبلي استفاده شدند . عمومي ترين اين استثنائات زير کلاسهاي نوع استاندارد RuntimeException مي باشند . بعلاوه لزومي ندارد آنها در هر فهرست throws مربوط به يك روش گنجانده شوند . در زبان جاوا ، اينها را استثنائات کنترل شده (unchecked exception) مي نامند . زيرا کامپايلر کنترل نمي کند که آيا يك روش اين استثنائات را اداره مي کند يا آنها را پرتاب مي نمايد . آنها را در جدول 1 فهرست نموده ايم . جدول 2 آن استثنائاتي را فهرست مي کند که توسط java.lang تعريف شده اند و بايد در يك فهرست throws مربوط به روش گنجانده شوند ، اگر آن روش بتواند يکي از اين استثنائات را توليد نموده ، اما خودش آن را اداره نکند . اين استثنائات را استثنائات کنترل شده (checked exceptions) مي نامند . جاوا چندين نوع ديگر از استثنائات را تعريف مي کند که با کتابخانه هاي گوناگون کلاس جاوا مرتبط هستند .

زير کلاسهاي کنترل نشده RuntimeException در جاوا

توضيح	استثنا
خطاي جبري ، مثل تقسيم بر صفر	ArithmeticException

<b>ArrayIndexOutOfBoundsException</b>	نمایه آرایه خارج از محدوده است
<b>ArrayStoreException</b>	انتساب به يك عضو آرایه از يك نوع سازگار
<b>ClassCastException</b>	تبدیل cast غیر معتبر
<b>IllegalArgumentException</b>	آرگومان غیر مجاز استفاده شده برای فراخوانی مجدد يك روش
<b>IllegalMonitorStateException</b>	عملیات ناشی غیر مجاز ، نظیر منتظر ماندن روی يك بند قفل نشده . (unlocked thread)
<b>IllegalThreadStateException</b>	عملیات درخواست شده ناسازگار با وضعیت بند جاری
<b>IndexOutOfBoundsException</b>	برخی انواع نمایه خارج از محدوده است
<b>NegativeArrayException</b>	آرایه ایجاد شده با يك اندازه منفی
<b>NullPointerException</b>	کاربرد غیر معتبر از يك مرجع تهی
<b>NumberFormatException</b>	تبدیل غیر معتبر يك رشته به يك فرمت رقمی
<b>SecurityException</b>	تلاش برای نقض امنیت

استثنائات کنترل شده تکرار در جاوا

استثنا	توضیح
<b>ClassNotFoundException</b>	کلاس پیدا نشده است
<b>CloneNotSupportedException.</b>	تلاش برای تولید مثل يك شي که رابط Cloneable را پیاده سازی نمیکند.
<b>IllegalAccessException</b>	دسترسی به يك کلاس انکار شده است
<b>InstantiationException</b>	تلاش برای ایجاد يك شي از يك کلاس یا يك رابط
<b>InterruptedException</b>	توقف يك thread توسط يك thread دیگر
<b>NoSuchFieldException</b>	عدم وجود فیلد درخواستی
<b>NoSuchMethodException</b>	عدم وجود متد درخواستی

ایجاد نمودن زیر کلاسهای استثنای مربوط به خودتان

اگرچه استثنائات توکار جاوا اکثر خطاهای رایج را اداره می کنند ، احتمال دارد بخواهید انواع استثنای مربوط به خودتان را ایجاد کنید تا شرایط مشخص پیش آمده در برنامه های شما را اداره کنند. انجام اینکار بسیار ساده است : فقط يك زیرکلاس از Exception تعریف نمایید ( که البته يك زیرکلاس از Throwable می باشد ). ( زیر کلاسهای شما لزومی ندارند تا واقعا " چیزی را پیاده سازی کنند.

کلاس Exception هیچگونه روشی برای خود تعریف نمی کند . البته این کلاس از روشهای فراهم شده توسط Throwable ارث می برد . بدین ترتیب کلیه استثنائات شامل آنهاییکه شما ایجاد کرده اید ، دارای روشهایی هستند که توسط Throwable تعریف شده و در دسترس آنها می باشند آنها را در جدول زیر نشان داده ایم . همچنین ممکن است بخواهید يك یا چند تا از این روشها را در کلاسهای استثنایی که ایجاد کرده اید ، لغو نمایید.

### روشهای تعریف شده توسط Throwable

شرح	متد
يك شي Throwable را که شامل يك ردیاب تکمیل شده پشته است برمی گرداند . این شي ممکن است مجددا پرتاب شود .	Throwable fillInStackTrace()
توصیفی از استثنای را برمی گرداند	String getMessage()
ردیاب پشته را نمایش می دهد	Void printStackTrace()
شي از نوع string را برمی گرداند که متضمن شرحی از استثناست. این متد هنگام نمایش محتوای شي throwable به وسیله println() فراخوانده میشود.	String toString()

مثال بعدی يك زیر کلاس جدید از Exception اعلان می کند و سپس از آن زیر کلاس استفاده می کند تا علامت يك شرط را به يك روش ارسال نماید . این زیر کلاس ، روش toString() را لغو می کند ، و با استفاده از println() اجازه می دهد تا توصیف استثنای بنمایش درآید .

```
// This program creates a custom exception type.  
class MyException extends Exception {
```

```

private int detail;
MyException(int a ){
detail = a;
}
public String toString (){
return "MyException[" + detail + "]";
}
}

class ExceptionDemo {
static void compute(int a )throws MyException {
System.out.println("Called compute(" + a + ")");
if(a > 10)
throw new MyException(a);
System.out.println("Normal exit");
}
public static void main(String args[] ){
try {
compute(1);
compute(20);
} catch( MyException e ){
System.out.println("Caught " + e);
}
}
}

```

این مثال یک زیر کلاس از Exception تحت عنوان MyException تعریف می کند. این زیر کلاس خیلی ساده است : این زیر کلاس فقط یک سازنده به علاوه یک روش toString() انباشته شده دارد که مقدار استثنای را نمایش می دهد . کلاس ExceptionDemo یک روش تحت نام compute() معرفی می کند که یک شیء MyException را پرتاب می کند . وقتی که پارامتر عدد صحیح مربوط به compute() بزرگتر از ۱۰ باشد ، استثنای پرتاب خواهد شد . روش main() یک اداره کننده استثنای برای MyException قرار می دهد ، سپس compute() را با یک مقدار مجاز ( کمتر از ۱۰ ) و نیز یک مقدار غیر مجاز برای نشان دادن دو مسیر موجود در کد فراخوانی می کند . حاصل بصورت زیر است :

Called compute(1)  
Normal exit  
Called Compute(20)

# آشنایی با کلاس ها

## عناوین این بخش :

اصول کلاس ها  
شیوه تعریف کردن شی  
تخصیص متغیرهای ارجاع به شی  
معرفی متدها ( جایگزین توابع )  
Constructor ها  
کلمه کلیدی this  
باز پس گیری حافظه بلا استفاده  
متد finalize()  
کلاس Stack

در این فصل علاوه بر نوع داده انتزاعی با نحوه تعریف و فراخوانی زیربرنامه (متد) ، کنترل زیربرنامه ، نحوه اختصاص حافظه به آنها ، ارسال پارامتر و زیربرنامه بازگشتی آشنا میشوید

## اصول کلاس‌ها

کلاس‌ها در هسته مرکزی جاوا جای دارند. کلاس‌ها، ساختار منطقی هستند که کل زبان جاوا بر روی آن ساخته شده است، چرا که شکل و ماهیت شی‌ها را تعریف می‌کنند. بدین ترتیب، کلاس‌ها پایه و اساس برنامه‌سازی شی‌گرا را در جاوا تشکیل می‌دهند. هر موضوعی که بخواهید در برنامه‌های جاوا پیاده‌سازی کنید، می‌بایست در یک کلاس نهان<sup>۱</sup> شود.

شاید مهمترین نکته‌ای که باید درباره کلاس‌ها یاد گرفت آن است که نوع جدیدی از داده‌ها را تعریف می‌کنند. داده‌های نوع جدید را پس از تعریف شدن می‌توان برای ایجاد شی‌های نوع مورد نظر به کار برد. از این رو، هر کلاس، الگویی<sup>۲</sup> برای یک شیء است. و هر شیء هم نمونه-ای<sup>۳</sup> از یک کلاس به شمار می‌آید. چون هر شیء، نمونه‌ای از یک کلاس است، اغلب خواهید دید که دو واژه شی و نمونه به جای یکدیگر به کار برده می‌شوند.

وقتی کلاسی را تعریف می‌کنید، ماهیت و فرم دقیق آن معرفی می‌شود. این کار با مشخص کردن داده‌های درون آن و روتین‌هایی که بر روی آن داده‌ها عمل می‌کنند، انجام می‌شود. اگر چه بسیاری از کلاس‌های ساده ممکن است تنها دربرگیرنده روتین‌ها یا داده‌ها باشند، اما بیشتر کلاس‌های مطرح در کارهای واقعی، هر دو را شامل می‌شوند. همان گونه که خواهید دید، روتین‌های هر کلاس، رابط منتهی به داده‌های آن را تعریف می‌کنند.

هر کلاس با استفاده از کلمه کلیدی Class تعریف می‌شود. کلاس‌ها می‌توانند بسیار پیچیده‌تر باشند (و معمولاً هستند) شکل عمومی تعریف هر کلاس در ذیل نشان داده شده است.

```
Class classname {  
    Type instance-variable1;  
    Type instance-variable2;  
    II...  
    Type instance-variableN;  
  
    Type methodname1(parameter-list){  
        //body of method  
    }  
  
    Type methodname2(parameter-list){
```

---

<sup>۱</sup> - encapsulated.

<sup>۲</sup> - template.

<sup>۳</sup> - instance.

```
//body of method
}
// ...
Type methodNameN(parameter-list) {
//body of method
}
}
```

داده‌هایی، یا متغیرهایی، که در هر کلاس تعریف می‌شوند، «نمونه متغیر»<sup>۴</sup> نامیده می‌شوند. روتین‌ها نیز در متدها جای می‌گیرند. به طور کلی، به متدها و متغیرهایی که در هر کلاس تعریف می‌شوند، اعضای<sup>۵</sup> کلاس گفته می‌شود. در بیشتر کلاس‌ها، متدهای تعریف شده برای هر کلاس هستند که بر روی نمونه متغیرها کار می‌کنند و به آنها دستیابی دارند. از این رو، این متدها هستند که چگونگی استفاده از داده‌های هر کلاس را تعیین می‌کنند.

دلیل اینکه متغیرهای هر کلاس، نمونه متغیر خوانده می‌شوند، آن است که هر نمونه از یک کلاس (یعنی، هر شیء از یک کلاس)، کپی خاص خود را متغیرها دارد. از این رو، داده‌های هر شیء، جداگانه و خاص خود آن بوده و با داده‌های یک شیء دیگر یکسان نیستند.

کلاس‌های جاوا نیاز به متد main() ندارند. تنها زمانی چنین متدی مشخص می‌شود که کلاس مورد نظر، نقطه آغازین برنامه‌تان باشد. به علاوه، آلت‌ها اصلاً نیاز به متدی به نام main() ندارند.

معرفی کلاس و پیاده سازی متدها در یکجا ذخیره می‌شوند و به طور جداگانه تعریف نمی‌شوند. این امر گاهی اوقات سبب ایجاد فایل‌های java بسیار بزرگ می‌شود، چه آنکه هر کلاس باید کلاً در یک فایل واحد تعریف شود.

## یک کلاس ساده

مطالعه کلاس‌ها را با یک مثال ساده آغاز می‌کنیم، برای این کار کلاسی به نام Box تعریف می‌کنیم که دارای سه نمونه متغیر به نام width, height, و depth است. در حال حاضر، Box فاقد هر گونه متد است (اما در آینده متدهایی به آن افزوده خواهد شد).

<sup>4</sup> - instance variable.

<sup>5</sup> - member.



```

Class Box {
double width;
double height;
double depth;
}

```

همان گونه که گفته شد، هر کلاس، نوع جدیدی از داده‌ها را تعریف می‌کند. در این مثال خاص، نوع جدیدی که ایجاد می‌شود، Box نامیده شده است. از این نام برای تعریف شیء‌ها نوع Box استفاده خواهد شد. مهم است به خاطر بسپارید که تعریف هر کلاس جدید تنها سبب ایجاد یک الگو<sup>۶</sup> می‌شود؛ یک شیء واقعی ایجاد نمی‌شود.

برای آنکه یک شیء Box ایجاد شود، می‌بایست از عبارتی همچون سطر زیر استفاده کنید.

```
Box mybox new BOX ( ); // create a Box object called mybox
```

پس از آنکه عبارت بالا اجرا شد، mybox به عنوان نمونه‌ای از Box ایجاد خواهد شد. از این رو، یک واقعیت «فیزیکی» از کلاس Box ایجاد خواهد شد.

باز هم لازم به ذکر است که هر گاه نمونه‌ای از یک کلاس را ایجاد می‌کنید شیئی ایجاد می‌شود که نسخه خاص خودش را از هر یک از نمونه متغیرهای تعریف شده در آن کلاس خواهد داشت. از این رو، هر شیء Box، نسخه‌های خاص خودش را از نمونه متغیرهای height, width و depth خواهد داشت. برای دستیابی به این متغیرها باید از عملگر نقطه<sup>۷</sup> (.) استفاده کنید. این عملگر، نام شیء را به نام «نمونه متغیر» مرتبط می‌کند. به عنوان مثال، برای آنکه مقدار ۱۰۰ را به متغیر width از mybox تخصیص دهید، از عبارت زیر استفاده کنید:

```
mybox.width=100;
```

عبارت بالا برای کامپایلر مشخص می‌کند که مقدار ۱۰۰ را به نسخه‌ای از width که در شیء mybox است تخصیص دهد. به طور کلی، از عملگر نقطه (.) برای دستیابی به نمونه متغیرها و متدهای موجود در یک شیء استفاده می‌شود.

هر شیء نسخه‌های خاص خودش را از نمونه متغیرها خواهد داشت. این بدین معناست که اگر دو شیء نوع Box داشته باشید، هر یک از آنها، نسخه‌های خاص خودشان را از height, width و depth خواهند داشت. مهم است بدانید که تغییراتی که در نمونه متغیرهای یک شیء ایجاد می‌شوند. هیچ تأثیری بر نمونه متغیرهای شیء دیگر نخواهند داشت.

<sup>۶</sup> . Template.

<sup>۷</sup> . dot.

## شیوه تعریف کردن شیءها

همان گونه که در بالا شرح داده شد، وقتی کلاسی را ایجاد می‌کنید، در واقع یک نوع جدید برای داده‌ها ایجاد می‌شود. از این نوع جدید می‌توانید برای تعریف کردن شیءهایی از آن نوع استفاده کنید. اما، رسیدن به شیءهای یک کلاس، نوعی فرآیند دو مرحله‌ای است. نخست اینکه، باید متغیری از نوع کلاس تعریف کنید. این متغیر سب تعریف یک شیء نمی‌شود. بلکه در عوض، متغیری است که می‌تواند به یک شیء ارجاع داشته باشد. دوم اینکه، می‌بایست یک نسخه فیزیکی واقعی از شیء به دست آورید و آن را به آن متغیر تخصیص دهید. این کار را می‌توانید با استفاده از عملگر new انجام دهید. عملگر new، حافظه‌ای را به طور پویا (یعنی در زمان اجرا) به شیء تخصیص می‌دهد و نشانی آن را برمی‌گرداند. این نشانی سپس در متغیر ذخیره می‌شود. از این رو، تمام شیءهای نوع کلاس در جاوا باید به طور پویا تخصیص یابند.

در نمونه برنامه‌های زیر از سطری مشابه عبارت زیر برای تعریف شیئی از نوع Box استفاده خواهد شد:

```
Box mybox = new Box ( );
```

دو مرحله پیش گفته در عبارت بالا ترکیب شده‌اند. عبارت بالا را می‌توان برای نشان دادن هر یک از مراحل به صورت زیر بازنویسی کرد:

```
Box mybox; // declare reference to object  
Mybox = new Box ( ) ; // allocate a Box object
```

در سطر نخست، mybox به عنوان نشانی شیئی از نوع Box تعریف می‌شود. پس از اجرای این خط، مقدار null در mybox ذخیره خواهد شد که نشانگر آن است که متغیر هنوز به هیچ شیء واقعی ارجاع ندارد. هر گونه اقدام برای استفاده از mybox در این مرحله منجر به بروز خطای زمان کامپایل خواهد شد. سطر دوم هم موجب تخصیص شیء واقعی و تخصیص نشانی آن به mybox می‌شود. پس از اجرای سطر دوم، می‌توانید از mybox به گونه‌ای استفاده کنید که گویی یک شیء Box است. اما mybox صرفاً نشانی حافظه شیء Box واقعی را نگهداری می‌کند.

## تخصیص متغیرهای ارجاع به شیء

وقتي عمل تخصیص انجام می‌گیرد، عملکرد متغیرهای ارجاع به شیء با آنچه انتظار دارید تفاوت دارد. فکر می‌کنید دو عبارت زیر چه عملی انجام می‌دهند؟

```
Box b1 = new Box ( );  
Box b2 = b1;
```

ممکن است چنین تصور کنید که نشانی نسخه‌ای از شیئی که b1 به آن ارجاع دارد. به b2 تخصیص می‌یابد. یعنی، ممکن است چنین فکر کنید که b1 و b2 به شیء‌های جداگانه و متمایزی ارجاع دارند. اما، این تصور درست نیست. بلکه در عوض، پس از اجرای عبارات بالا، b1 و b2 هر دو به یک شیء ارجاع خواهند داشت. تخصیص b1 و b2 موجب تخصیص حافظه یا کپی کردن بخشی از شیء اولیه نمی‌شود. بلکه صرفاً سبب می‌شود که b2 نیز به همان شیئی که b1 به آن ارجاع دارد، ارجاع داشته باشد. از این رو، هر گونه تغییر در شیء از طریق b2، بر شیئی که b1 به آن ارجاع دارد، تأثیر خواهد گذاشت، چرا که هر دو آنها یک شیء هستند.

## معرفی متدها

کلاس‌ها معمولاً از دو چیز تشکیل می‌شوند: نمونه متغیرها<sup>۸</sup> و متدها<sup>۹</sup>. موضوع متدها بسیار گسترده‌است، چرا که جاوا قدرت و انعطاف‌پذیری زیادی را در آنجا جای داده است. شکل کلی هر متد به صورت زیر است:

```
Type name(parameter-list) {  
    // body of method  
}
```

Type، نوع داده‌هایی را مشخص می‌کند که متد باز می‌گرداند. Type می‌تواند هر یک از انواع مورد بررسی قبلی باشد، از جمله انواع کلاس‌هایی که خودتان ایجاد می‌کنید. چنانچه متد چیزی را برنگرداند، type باید void باشد. نام متد نیز به وسیله name مشخص می‌شود. از هر شناسه معتبری می‌توانید به عنوان نام استفاده کنید؛ البته به غیر از مواردی که برای اقلام موجود در همان محدوده جاری استفاده شده اند. Parameter-list، فهرست زوج‌هایی (نوع و شناسه) است که با کاما از یکدیگر جدا می‌شوند. پارامترها اساساً متغیرهایی هستند که مقدار آرگومان‌های

---

<sup>۸</sup> - instance variable.

<sup>۹</sup> - method.

ارسالي به متد را هنگام فراخواني آن دريافت مي‌کنند. چنانچه متد پارامتري نداشته باشد، اين فهرست خالي خواهد بود.

متدهايي که نوع مقدار حاصل از فراخواني آنها چي به غير از void باشد، مقداري را با استفاده از عبارت return به روتين فراخوان بازمي‌گردانند:

```
Return value;
```

Value ، مقداري است که برگردانده مي‌شود.

### افزودن متد به کلاس Box

اگر چه ايجاد کلاس‌هايي که تنها حاوي داده باشند کاملاً درست است، اما اين امر به ندرت رخ مي‌دهد. در بيشتر مواقع از متدها براي دستيابي به نمونه متغيرهاي تعريف شده در کلاس‌ها استفاده خواهد شد. در حقيقت، متدها، رابط دستيابي به بيشتر کلاس‌ها را تعريف مي‌کنند. اين امر به ايجاد کننده کلاس‌ها امکان مي‌دهد تا شمائي ساختارهاي داده‌اي مرتبط با کلاس را پشت سر متدهاي شفاف‌تر پنهان نمايد. علاوه بر تعريف متدهايي که دستيابي به داده‌ها را فراهم مي‌سازند، امکان تعريف متدهايي که توسط خود کلاس‌ها به طور داخلي مورد استفاده قرار مي‌گيرند نيز فراهم شده است.

اينک کار خود را با افزودن متدي به کلاس Box آغاز مي‌کنيم.

```
Class box {
    Double width;
    Double height;
    Double depth;

    // display volume of a box
    Void volume ( ) {
        System.out.print ("volume is");
        System.out.print(width *height*depth);
    }
}

Class BoxDemo3 {
    Public staticvoid main (string args [] ) {
```

```

Box mybox1 = new Box ( ) ;
Box mybox2 = new Box ( ) ;

// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;

/*assign different values to mybox2's
Instance variables */
Mybox2.width = 3;
Mybox2.height = 6;
Mybox2.depth = 9;

//display volume of first box
Mybox1.volume ( ) ;

//display volume of second box
Mybox2.volume ( ) ;
    }
}

```

خروجی برنامه در ذیل آورده شده است که البته با خروجی مثال پیش یکسان است.

```

Volume is 3000.0
Volume is 162.0

```

وقتی دستیابی به نمونه متغیرها به وسیله روتینی انجام می‌گیرد که در همان کلاس تعریف متغیرها تعریف نشده است، در آن صورت باید این کار از طریق نام شیء و عملکرد نقطه (.) انجام شود. اما، وقتی این کار به وسیله روتینی انجام می‌شود که بخشی از همان کلاس مربوط به متغیرهاست، در آن صورت متغیرها به طور مستقیم قابل ارجاع می‌باشند. این مطلب درباره متدها نیز صادق است.

### بازگرداندن مقادیر

دو نکته مهم درباره مقادیر حاصل از فراخوانی متدها وجود دارد که باید به خوبی با آنها آشنا باشید:

نوع داده‌های حاصل از فراخوانی متد باید با نوعی که در تعریف متد مشخص شده است، سازگار باشد. به عنوان مثال، اگر نوع مقداری که یک متد بازمی‌گرداند، Boolean باشد، نمی‌توانید مقدار صحیحی را بازگردانید.

متغیر دریافت‌کننده مقدار حاصل از فراخوانی متد (مثلاً `vol` در این مثال)، باید با نوعی که در تعریف متد مشخص شده است، سازگار باشد.

### افزودن متدهای پارامتریک

اگر چه برخی از متدها نیاز به پارامتر ندارند، اما بیشتر متدها این گونه نیستند. پارامترها، امکان عمومیت بخشیدن به متدها را فراهم می‌سازند. یعنی، متدهای پارامتریک می‌توانند بر روی انواع داده‌ها عمل کنند، و یا در شرایط نسبتاً مختلف مورد استفاده قرار گیرند. برای درک این نکته به مثال بسیار ساده زیر توجه کنید. متد زیر، مجذور عدد ۱۰ را بازمی‌گرداند:

```
Return I + 1 * i; int square( )
{
    Return 10 * 10;
}
```

اگر چه این متد واقعاً مجذور عدد ۱۰ را بازمی‌گرداند، اما کاربرد آن بسیار محدود است، اما اگر متد را به گونه‌ای تغییر دهیم تا پارامتری را همچون مثال زیر دریافت کنید، در آن صورت `square()` بسیار مفیدتر می‌شود.

```
Int square (int i)
{
    }
}
```

بدین ترتیب `square()` اینک مجذور هر مقداری که با آن فراخوانده می‌شود را بازمی‌گرداند. یعنی، `square()` به متد همه منظوره‌ای مبدل شده است که به جای عدد ۱۰۰، مجذور هر عدد صحیح را محاسبه می‌کند.

حفظ تمایز بین دو واژه پارامتر و آرگومان از اهمیت خاصی برخوردار است. منظور از پارامتر، متغیری است که توسط متد تعریف می‌شود و وقتی متد فراخوانده می‌شود، مقداری را دریافت می‌کند. به عنوان مثال، در متد بالا، `i` پارامتر به شمار می‌آید. منظور از آرگومان، مقداری است که

هنگام فعال سازي متد به آن ارسال مي‌شود. به عنوان مثال، در `square(100)` ، عدد ۱۰۰ به عنوان آرگومان استفاده مي‌شود. در متد `square()` ، پارامتر `i` ، آن مقدار را دريافت مي‌کند.

## Constructor ها

جاوا اين امکان را فراهم ساخته تا شيءها خودشان را به هنگام ايجاد، مقدار دهی کنند اين مقداردهی خودکار، از طريق استفاده از يك constructor انجام مي‌شود. constructor ، يك شيء را به محض ايجاد مقدار دهی مي‌کند. نام آن با نام کلاسي که در آن قرار دارد يکسان بوده و از نظر ساختار گرامري نيز مشابه متدهاست. هر constructor پس از تعريف، به طور خودکار به محض ايجاد شيء فراخوانده مي‌شود. اين وظيفه constructorهاست که وضعيت داخلي يك شيء را در همان ابتداي کار تعيين کنند (مقدار دهی اوليه)، تا روتيني که نمونه‌اي از کلاس را ايجاد مي‌کند، فوراً شيء قابل استفاده و مقدار دهی شده‌اي داشته باشد.

```
/*Here, Box uses a constructor to initialize the dimensions of a box.
*/
Class Box {
Double width;
Double height;
Double depth;

//This is the constructor for Box.
Box ( ) {
System.out.println("Constructing Box");
Width = 10;
Height = 10;
depth = 10;
}

/// compute and return volume
Double volume ( ) {
Return width * height * depth;
}
```

```
}
```

```
Class BoxDemo6 {  
Public static void main (String args [] ) {  
//declare,allocate, and initialize Box objects  
Box mybox1 = new Box ( ) ;  
Box mybox2 = new Box ( ) ;  
  
Double vol;  
  
//get volume of first box  
Vol=mybox1.volume ( ) ;  
System.out.println("Volume is " + vol);  
  
//get volume of second box  
Vol=mybox2.volume ( ) ;  
System.out.println("volume is " + vol);  
}  
}
```

وقتي برنامه اجرا مي‌شود، نتايج آن به شكل زير خواهد بود:

```
Constructing Box  
Constructing Box  
Volume is 1000.0  
Volume is 1000.0
```

همان گونه که ملاحظه مي‌کنید ، mybox1 و mybox2 هر دو هنگام ايجاد به وسيله constructor که Box() نام دارد، مقدار دهی شده‌اند. از آنجايي که Constructor ابعاد 10×10×10 را به تمام مکعب‌ها نسبت مي‌دهد، حجم mybox1 و mybox2 برابر خواهد شد. عبارت println() در Box() صرفاً به خاطر نمايش صحت عملکرد آن است. بيشتر constructor ها چيزي را نمايش نخواهند داد. آنها صرفاً عمل مقداردهي اوليه را براي شيء انجام مي‌دهند.

اينک مي‌توانيد حدس بزنيد که چرا وجود پرانتزها پس از نام کلاس ضروري است. آنچه که واقعاً رخ مي‌دهد، آن است که constructor کلاس فراخوانده مي‌شود. از اين رو، در سطر زير،

```
Box mybox1 = new Box ( ) ;
```



`newBox()` موجب فراخوانی `constructor` کلاس می‌شود که همانام با خود کلاس است (یعنی `Box()` ) وقتی `constructor` ای را صریحاً برای کلاسی تعریف نمی‌کنید. جاوا این کار را به طور پیش فرض انجام می‌دهد. به همین دلیل است که سطر بالا در نگارشهای پیشین مثال `Box` ، که فاقد تعریف `constructor` بودند، به خوبی کار می‌کرد. `constructor` پیش فرض، تمام نمونه متغیرها را به طور خودکار با صفر مقداردهی می‌کند. `Constructor` پیش فرض اغلب برای کلاس‌های ساده کفایت می‌کند، اما معمولاً برای کلاس‌های پیچیده‌تر کفایت نمی‌کند. وقتی `constructor` خاص خودتان را تعریف می‌کنید، `constructor` پیش فرض دیگر به کاربرده نمی‌شود.

### Construdtor های پارامتریک

اگر چه `constructor` مثال پیش `Box()` عمل مقدار دهی اولیه را برای شیء `Box` انجام می‌دهد، اما چندان مفید نیست - ابعاد تمام مکعبها یکسان خواهد بود. باید به دنبال روشی برای ساخت شیء‌های `Box` با ابعاد گوناگون باشیم. راه حل آسان برای انجام این کار، افزودن پارامترهایی به `constructor` است. همان گونه که احتمالاً حدس زده‌اید، انجام این کار موجب مفیدتر شدن آن می‌شود. به عنوان مثال، در نگارش جدید `Box` ، یک `constructor` پارامتریک تعریف شده است که ابعاد هر مکعب را براساس تعداد پارامترها تعیین می‌کند.

```
//This is the constructor for box.  
Box (double w, double h, double d)    {  
    Width = w;  
    Height = h;  
    Depth = d;  
}
```

### کلمه کلیدی this

گاهی اوقات متدها نیاز به ارجاع به شیئی دارند که آنها را فعال کرده است. جاوا برای فراهم ساختن این امکان، کلمه کلیدی `this` را تعریف کرده است. با استفاده از `this` در هر متد می‌توان

به شيء جاري ارجاع نمود. يعني ، `this` همیشه ارجاع به شيءي دارد که متد براي آن فعال شده است. هر جا که ارجاع به شيءي از کلاس جاري مجاز باشد، مي توان از `this` استفاده نمود. براي درك بهتر اينك `this` به چه چيزي ارجاع دارد، به نگارش زير از `Box()` توجه كنيد.

```
// A redundant use of this.  
Box(double w, double h, double d) {  
    This.width = w;  
    This.height = h;  
    This.depth = d;  
}
```

اين نگارش از `Box()` دقيقاً همچون نگارش قبلي کار مي کند. استفاده از `this` بي مورد است، اما کاملاً صحيح است. `this` در اين نگارش، همیشه به شيءي که متد را فرا مي خواند، ارجاع خواهد داشت. اگر چه کاربرد آن در اين مثال بي مورد است، اما در ساير موارد مفيد واقع مي شود.

### بازپس گيري حافظه بلا استفاده

از آنجايي که شيء ها با استفاده از عملگر `new` به طور پويا تخصيص مي يابند، ممکن است از خود بپرسيد که چگونه از بين برده مي شوند و حافظه آنها چگونه براي استفاده هاي آتي آزاد مي شود. در برخي از زبانها، از قبيل `C++` ، شيء هايي که به طور پويا تخصيص داده مي شوند را بايد به صورت دستي با استفاده از عملگر `delete` ، آزاد نمود. جاوا از رويه ديگري استفاده مي کند؛ آزاد سازي را به طور خودکار براي تان انجام مي دهد. تکنیکی که از آن براي انجام اين کار استفاده مي شود، `garbage collection` نام دارد. عملکرد آن به اين شرح است؛ وقتي هيچ گونه ارجاعي به يك شيء وجود نداشته باشد، فرض مي شود که شيء ديگر مورد نياز نبوده و حافظه آن نيز بازپس گرفته مي شود. در زبان جاوا، برخلاف `C++` ، ديگر نيازي به از بين بردن شيء ها نيست. اين کار تنها به صورت نامنظم و گاه و بيگاه در طي اجراي برنامه انجام مي شود.

### متد `Finalize()`

گاهی اوقات برخی از شیء‌ها نیاز به انجام عملیات خاص پیش از بین بردن دارند. به عنوان مثال، اگر شیئی از منابع غیر جاوا، از قبیل `handle` یک فایل یا فونت خاص، استفاده می‌کند، در آن صورت بهتر است پیش از آزادسازی آن شیء، از آزاد شدن آن منابع اطمینان حاصل نمایید. جاوا برای مدیریت این گونه شرایط، مکانیزمی به نام `finalization` دارد. با استفاده از این مکانیزم می‌توانید عملیات خاصی را مشخص کنید تا درست پیش از آزاد سازی یک شیء، تماماً انجام شوند.

برای پیاده سازی این مکانیزم در هر کلاس، کافی است متد `finalize()` را تعریف کنید. محیط زمان اجرای جاوا این متد را هنگام بازیافت شیئی از آن کلاس فرا می‌خواند. در متد `finalize()` باید آن عملیاتی را مشخص کنید که باید پیش از بین بردن یک شیء انجام شوند. قسمتی که از مسئولیت بازپس‌گیری حافظه بلا استفاده را دارد، به طور متناوب اجرا شده و شیء‌هایی را جستجو می‌کند که دیگر ارجاعی به آنها صورت نمی‌گیرد و به‌طور غیر مستقیم نیز از طریق سایر شیء‌ها به آنها ارجاع نمی‌شود. درست پیش از آزاد کردن هر شیء، سیستم «زمان اجرای» جاوا متد `finalize()` را برای آن شیء فرا می‌خواند.

شکل کلی این متد در زیر نشان داده شده است:

```
Protected void finalize()
{
    //finalization code here
}
```

کلمه کلیدی `protected`، مشخصه‌ای<sup>10</sup> است که از دستیابی به `finalize()` توسط روتین‌های خارج از همان کلاس جلوگیری می‌کند.

مهم است بدانید که `finalized()` تنها پیش از بازپس‌گیری حافظه شیء‌ها فراخوانده می‌شود. مثلاً زمانی که یک شیء در خارج از محدوده‌اش قرار می‌گیرد، این متد فراخوانده نمی‌شود.

## Overload کردن متدها

در زبان جاوا این امکان فراهم شده تا دو و یا بیش از دو متد هم‌نام در یک کلاس تعریف نمود، مشروط بر اینکه تعریف پارامترهای آنها متفاوت می‌باشد در این گونه موارد گفته می‌شود که

---

<sup>10</sup> - specifier.

متدها، Overload شده اند و به این فرآیند، method overload گفته می‌شود. این فرآیند یکی از روشهایی است که جاوا از طریق آن پلی مورفیزم پشتیبانی می‌کند. وقتی متد overload شده ای فعال می‌شود، جاوا از انواع و یا تعداد آرگومان ها برای تعیین اینکه کدام نگارش از متدها overload شده فراخوانده شده است، از این رو، متدها overload شده از جهت نوع و تعداد پارامترها با یکدیگر تفاوت دارند. اگرچه نوع مقادیری که این متدها بر می‌گرانند ممکن است متفاوت باشد، اما نوع مقادیر به تنهایی برای تمایز بین آنها کفایت نمی‌کند. وقتی که جاوا با عبارت فراخوانی این گونه متدها مواجه می‌شود، متدی را اجرا می‌کند که پارامترهای آن با آرگومان های مورد استفاده در عبارت فراخوانی مطابق داشته باشد. مثال ساده زیر فرآیند overload کردن متدها را نشان می‌دهد:

```
// Demonstrate method overloading.
class overload Demo {
    void test ( ) {
        system.out.println ("No parameters");
    }
// overload test for one integer parameter.
void test (int a) {
    system.out.println ("a: " + a );
// overload test for two integer parameters.
void test (int a, in b)
    system.out.println ("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test (double a) {
    system.out.printl ("double a: " + a);
    return a *a;
}
}
class overload {
    public static void main (strain gags [ 1) {
        overload Demo ob = new overload Demo ( ) :
        double result ;

// call all versions of test ( )
ob.test ( ) ;
```

```

ob.test (10) ;
ob.test (10,20);
result = ob.test (123.25);
system.out.println ("Result of ob.test (123.25) : " + result) ;
    }
}

```

```

No parameters
a:    20
a and b: 10  20
double a: 123.25
Result of ob.test (123.25): 15190.5625

```

همان گونه که ملاحظه می‌کنید، ( ) test چهار مرتبه overload شده است. نخستین نگارش آن پارامتری ندارد، نگارش دوم تنها یک پارامتر از نوع اعداد صحیح دارد، نگارش سوم دو پارامتر از نوع اعداد صحیح دریافت می‌کند، و نگارش چهارم نیز تنها یک پارامتر از نوع double دارد. وقتی متد overload شده ای فراخوانده می‌شود، جاوا تطابق بین آرگومان های مورد استفاده برای فراخوانی، و پارامترهای متد را بررسی می‌کند. اما، نیازی نیست که این تطابق همیشه دقیق باشد. تبدیل خودکار انواع داده ها در برخی از شرایط، نقش مهمی را تعیین متدی که باید فعال شود، ایفا می‌کند.

دلیل اینکه فرآیند overload کردن متدها از پلی مورفیسم پشتیبانی می‌کند، آن است که این فرآیند یکی از راههایی است که جاوا از طریق آن، مدل «یک رابط، چند متد» را پیاده سازی می‌کند. در زبانهایی که از overload کردن پشتیبانی نمی‌کنند، معمولاً دو یا سه نگارش از این تابع وجود دارد که نامشان قدری با یکدیگر تفاوت دارد. به عنوان مثال، تابع ( ) abs در زبان C قدر مطلق یک عدد صحیح را بر می‌گرداند، و ( ) labs قدر مطلق یک عدد صحیح نوع long را بر می‌گرداند، و ( ) fabs نیز قدر مطلق یک مقدار اعشاری با ممیز شناور را بر می‌گرداند. از آنجایی که زبان C از overload کردن متدها پشتیبانی نمی‌کند، با وجود آنکه هر سه تابع اساساً یک کار انجام می‌دهند، اما هر یک از توابع پایه خاص خودشان را داشته باشند. وضعیت کار را از نظر مفهومی پیچیده تر از آنچه که واقعاً هست، می‌کند. اگرچه مفهوم سه تابع یکسان است، اما باید سه نام مختلف را به خاطر بسپارید. این وضعیت در جاوا پیش نمی‌آید، چرا که متدهای قدر مطلق می‌توانند از نام مشترکی استفاده کنند. متدی به نام ( ) abs را در کتابخانه استاندارد کلاس های

جاوا وجود دارد. این متد در کلاس match برای مدیریت انواع مختلف داده ها overload شده است. جاوا بر اساس نوع آرگومان تصمیم می‌گیرد که کدام نگارش تابع فراخوانده شود. ارزش overload به خاطر آن است که امکان دستیابی به متدهای مرتبط به هم را از طریق کاربرد نام مشترک فراهم ساخته است. از این رو، نام abs نمایانگر عمل عمومی است که انجام می‌شود. این وظیفه کامپایلر است که نگارش خاص مورد نظر را برای هر یک از شرایط انتخاب کند، و شما به عنوان برنامه ساز تنها باید عمل عمومی که انجام می‌شود را به خاطر بسپارید. وقتی متدی را overload می‌کنید، هر یک از نگارشهای آن می‌توانند یکی از کارهای مورد نظران را انجام دهند. هیچ قانونی مبنی بر اینکه متدهای overload شده باید با یکدیگر مرتبط باشند وجود ندارد. اما از منظر سبک کار، فرآیند overload کردن متدها به خوبی خود القاء کننده نوعی رابطه است. از این رو، اگرچه با استفاده از نام مشترک می‌توانید متدهای غیر مرتبط را overload کنید، اما توصیه می‌شود این کار را انجام ندهید.

## Overload کردن Constructorها

متدهای Constructor را نیز می‌توانید همچون متدهای معمولی overload کنید. در حقیقت، در بیشتر کلاس های مربوط به کارهای واقعی constructor های overload شده، نه تنها استثنا به شمار نمی‌آیند، بلکه کاملاً معمول خواهند بود.

## استفاده از شیءها به عنوان پارامتر

تا به حال تنها از انواع داده های پایه و ساده به عنوان پارامتر متدها استفاده کرده ایم. اما، ارسال شیءها به متدها هم درست و هم متداول است. به عنوان مثال، برنامه کوتاه زیر را در نظر بگیرید:

```
// objects may be passed to methods.
Class test {
    Int a . b ;

    Test (in i, int ) {
```

```

        a = i;
        b = j ;
    }
    // return true if o is equal to the invoking object Boolean equals
    (test o) {
        in (o.a == a & & o.b == b) return true;
        else return false;
    }
}

class passob {
    public static void main (string agrs [ ]) {
        test ob1= new test (100 , 22) ;
        test ob2 = new test (100 , 22) ;
        test ob3 = new test (-1 , -1);

        system.out.println ("ob1 == ob2: " + ob1.equals (ob2) ) ;

        system.out.println (: ob1 == ob3: " + ob1.equals (ob3) ) ;
    }
}

ob1== ob2: true
ob1 == ob3: false

```

همان گونه که ملاحظه می‌کنید، متد ( equals ) در برنامه Test، برابر بودن دو شیء را مقایسه و نتیجه را بر می‌گرداند. یعنی، شیء فعال کننده متد را با شیء ارسالی مقایسه می‌کند. اگر مقادیر آنها یکسان باشد، در آن صورت مقدار حاصل متد، true خواهد بود. در غیر این صورت، حاصل آن false خواهد بود.

یکی از متداولترین کاربردهای پارامترهای نوع شیء، به استفاده از constructorها مربوط می‌شود. غالباً نیاز به ایجاد شیء‌های جدیدی خواهد داشت که مقدار دهی اولیه آنها می‌بایست همچون یکی از شیء‌های موجود باشد. برای انجام این کار باید constructorای تعریف کنید تا شیئی از نوع کلاس خودش را به عنوان پارامتر دریافت کند. به عنوان مثال، نگارش زیر از کلاس Box امکان مقدار دهی اولیه یک شیء با استفاده از یک شیء دیگر را فراهم می‌سازد:

```

Class Box {

```

```

    double width;
    double height;
    double depth;

// construct clone of an object
Box (Box ob)    { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
}
// constructor used when all dimensions specified
Box (double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
}
// constructor used when no dimensions specified
Box ( ) {
    width = -1; // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1; // box
}

// constructor used when cube is created
Box (double len) {
    width = height = depth = len;
}

// compute and return volume
double volume ( ) {
    return width * height * depth;
}
}

class overloadcons 2P
public static void main (string args [ ]) {
// create boxes using the various constructors
Box mybox1= new box (10 , 20 , 15);

```



```

Box mybox2= new box ( );
Box mycube = new box (7) ;

Box myclone = new box (mybox1) ;

Doble vol;

/ / get volume of first box
vol = mybox1.volume ( ) ;
system.out.println ("volume of mybox1 is " + vol) ;
/ / get volume of second box
vol= mebox2.volume ( ) ;
system.out.println ("volume of mybox2 is " + vol) ;

/ / get volume of cube
vol = mycube.volume ( ) ;
system.out.println ("volume of clone is " +vol) ;
    }
}

```

### نگاهی دقیقتر به روند ارسال پارامترها

بطور کلی، در هر زبان برنامه سازی دو روش برای ارسال آرگومان ها به يك سابروتین وجود دارد. روش نخست، "call-by-value" نام دارد. در این روش، مقدار آرگومان به پارامتر سابروتین کپی می شود. بنابراین، تغییراتی که در پارامتر اعمال می شوند، هیچ تأثیری بر آرگومان نخواهند داشت. روش دوم، "call-by-reference" نام دارد. در این روش، نشانی آرگومان (و نه مقدار آرگومان) به پارامتر ارسال می شود. از این نشانی در سابروتین برای دستیابی به خود آرگومان مشخ شده در عبارت فراخوانی استفاده می شود. این بدین معناست که تغییراتی که در پارامتر اعمال می شوند، بر آرگومان مورد استفاده برای فراخوانی سابروتین تأثیر خواهند گذاشت. همان گونه که خواهید دید، جاوا بسته به چیزی که ارسال می شود، از هر دو روش استفاده می کند. وقتی در جاوا یکی از انواع داده های پایه را به متدی ارسال می کنید، از روش نخست استفاده می شود. از این رو، آنچه برای پارامتر دریافت کننده آرگومان رخ می دهد، بازتابی در خارج از متد نخواهد داشت.

وقتي شيئي را به متدي ارسال مي كنيد، وضعيت به طور چشمگيري تغيير مي كند، چرا كه شيءها با روش "call-by-reference" ارسال مي شوند. به خاطر داشته باشيد كه وقتي متغيري از نوع كلاس ايجاد مي كنيد، تنها نوعي نشاني به يك شيء ايجاد مي شود. از اين رو، وقتي اين نشاني را به متدي ارسال مي كنيد، پارامتر دريافت كننده آن، به همان شيئي ارجاع خواهد شد كه آرگومان متناظرش به آن ارجاع دارد. اين بدین معناست كه شيءها با روش "call-by-reference" به متدها ارسال مي شوند. تغييراتي كه در متد به شيء اعمال مي شوند، بر شيء مورد استفاده به عنوان آرگومان تأثير خواهند داشت. به عنوان مثال، برنامه زير را در نظر بگيريد:

```
// objects are passed by reference.
Class test {
    in a, b ;

    test (int i, int j) {
        a = i ;
        b = j ;
    }
    / / pass an object
    void meth (Test o) {
        o.a * = 2;
        o.b = 2 ;
    }
}

Class callbref {
    public static void main (String args [ ]) {
        Test ob = new Test (15 , 20);

        System.out.println ("ob.a and ob.b before call: " +
                               ob.a + " " + ob.b) ;

        ob.meth (ob) ;

        System.out.println ("ob.a and ob.b after call: " +
                               ob.a + " " + ob.b) ;
    }
}
```

ob.a and ob.b before call: 15 20

ob.a and ob.b after call: 30 10

به عنوان نکته اي جالب بد نيست بدانيد كه وقتي نشاني شيئي به يك متد ارسال مي شود، خود نشاني به صورت "call-by-value" ارسال مي شود. اما، از آنجايي كه مقدار در دست ارسال، به شيء ارجاع دارد، کپی مقدار آن نیز به همان شيء متناظر با آرگومان ارجاع خواهد شد.

شيء هايي كه به عنوان خروجي متدها باز گردانده مي شوند

هر متد انواع مختلفي از داده ها را بازگرداند؛ از جمله انواع كلاس هايي را كه ايجاد مي كنيد. به عنوان مثال، متد ( ) incrByTen در برنامه صفحه بعد، شيئي را به عنوان خروجي باز مي گرداند كه مقدار a در آن، ۱۰ واحد بيشتر از مقدار a در شيء فعال كننده متد است.

```
/ / Returning an object.
```

```
class Test
```

```
    int a ;
```

```
    Test (in i)    {
```

```
        a = I ;
```

```
    }
```

```
    Test incrByTen ( )    {
```

```
        Test temp = new Test (a+10) ;
```

```
        Return temp;
```

```
    }
```

```
}
```

```
Class Retub    {
```

```
    Public static void main (string args [ ] )    {
```

```
        Test ob1 = new Test (2) ;
```

```
        Test ob2 ;
```

```
        ob2 = ob1.incrByTen ( ) ;
```

```
        system.out.println ("ob1.a: + ob1.a) ;
```

```
        system.out.println ("ob2.a: + ob2.a) ;
```

```
        ob2 = ob2.incrByTen ( ) ;
```

```
        system.out.println ("ob.a after second increase: "
                               + ob.a) ;
```

```
    }
```

```
}
```

```
ob1.a : 2
```

```
ob2.a: 12
```

```
ob2.a after second increase: 22
```

از آنجایی که تمام شیء‌ها با استفاده از `new` به طور پویا تخصیص داده می‌شوند، نیازی نیست که به دلیل پایان یافتن اجرای متدی که شیء در آن ایجاد شده، نگران خارج شدن شیء از محدوده دستیابی باشید. تا زمانی که نشانی شیء را در جایی از برنامه خود داشته باشید، دوره حیات آن ادامه خواهد یافت. هرگاه دیگر ارجاعی به شیء نباشد. حافظه آن در مرتبه آتی اجرای «سیستم باز پس گیری حافظه بلا استفاده»، آزاد می‌شود.

## قابلیت بازگشت

جاوا از مسئله قابلیت بازگشت<sup>۱۱</sup> پشتیبانی می‌کند. منظور از قابلیت بازگشت، فرآیند تعریف کردن یک چیز بر حسب خودش می‌باشد. «قابلیت بازگشت» خصوصیتی است که متدها امکان می‌دهد تا خودشان را فرا بخوانند. متدی که خودش را فرا بخواند، بازگشتی<sup>۱۲</sup> نامیده می‌شود. مثال کلاسیک «قابلیت بازگشت»، محاسبه فاکتوریل اعداد است. روش محاسبه فاکتوریل هر عدد با استفاده از روش «قابلیت بازگشت» در ذیل نشان داده شده است:

```
// A simple example of recursion.
Class Factorial {
    // this is a recursive function
    int fact (int n) {
        int result;
        if (n == 1) return 1;
        result = fact (n-1) * n;
        return result ;
    }
}

class Recursion {
    public static void main (String args [ ]) {
        Factorial f = new Factorial ( );
    }
}
```

---

<sup>۱۱</sup>- Recursion.

<sup>۱۲</sup>- Recursive.

```

        system.out.println ("Factorial of 3 is " + f.fact (3) ) ;
        system.out.println ("Factorial of 4 is " + f.fact (4) ) ;
        system.out.println ("Factorial of 5 is " + f.fact (5) ) ;
    }
}

```

وقتي متدي خود را فرا مي خواند، فضاي مورد نياز متغيرهاي محلي جديد و پارامترها در پشته تخصیص مي يابد، و قسمت اجرايي متد با همين متغيرهاي جديد از ابتدا اجرا مي شود. با پايان رسيدن نتيجه هر عبارت فراخواني، متغيرهاي محلي قديمي و پارامترها از روي پشته برداشته مي شوند، و اجرا از نقطه آغاز فرا خواني در خود متد ادامه مي يابد. اصطلاحاً گفته مي شود که متدهاي بازگشتي همچون تلسکوپ ها، حرکت رو به جلو و عقب دارند.

نگارشهاي بازگشتي بسياري از روتين ها ممکن است به دليل «سربار»<sup>۱۳</sup> اضافي حاصل از فراخوانيهاي اضافي، قدرتي کندتر از معادلهاي تکراري آنها اجرا شود. بسياري از اين گونه فراخوانيها (براي متدهاي بازگشتي) ممکن است به سر ريز پشته منجر شود. چون فضاي لازم براي پارامترها و متغيرهاي محلي از پشت تأمین مي شود و هر عبارت فراخواني جديد موجب ايجاد نسخه جديدي از اين متغيرها مي شود، امکان دارد پشته با کمبود فضا مواجه شود. اگر چنين اتفاقي رخ دهد، سيستم زمان اجراي جاوا بروز استثنا خواهد شد. اما، احتمالاً نبايد نگران اين مسئله باشيد، مگر آنکه روتين بازگشتي مورد استفاده تان، درست عمل نکند.

مزيت اصلي متدهاي بازگشتي آن است که با استفاده از آنها مي توان نگارشهاي شفافتر و ساده تري از الگوريتم ها را نسبت به معادلهاي تکراري خودشان ايجاد نمود. به عنوان مثال، پياده سازي الگوريتم مرتب سازي QuickSort با روش تکراري کاملاً دشوار است. به نظر مي رسد که حل برخي از مسائل، به ويژه مسائل مرتبط با AI، با راه حلهاي بازگشتي آسانتر باشد. و بالاخره اينکه، روش تفکر بازگشتي براي بخري از افراد آسانتر از روش تفکري تکراري است.

وقتي متدهاي بازگشتي مي نويسيد، بايد در جايي از آن عبارت If استفاده کنيد تا متد مقداري را بدون فراخواني مجدد خودش بازگرداند. اگر اين کار انجام ندهيد، پس از فراخواني متد، هرگز چيزي باز نخواهد گشت. اين خطا هنگام کار با اين روش بسيار متداول است. در حين نوشتن متد از عبارت Println ( ) استفاده کنيد تا قادر به رديابي اتفاقات درون آن باشيد، و به راحتی بتوانيد اجراي آن را در صورت بروز هرگونه خطا قطع کنيد.

---

<sup>13</sup> - Overhead.

## مقدمه اي بر كنترل دستيابي

همان گونه كه مي دانيد، «نهان سازي»<sup>14</sup>، داده ها را با روتين هايي كه آنها را پردازش و مديريت مي كنند، مرتبط مي سازد. اما، «نهان سازي» خصوصيت مهم ديگري هم دارد: كنترل دستيابي. از طريق نهان سازي مي توانيد كنترل كنيد كه کدام قسمتهاي برنامه مي توانند به اعضاي كلاس مورد نظر دستيابي داشته باشند. با تحت كنترل در آوردن دستيابي مي توانيد از كار برد نادرست جلوگیری كنيد.

چگونگي دستيابي به هر كي از اعضاي يك كلاس به وسيله «مشخصه دستيابي»<sup>15</sup> مورد استفاده براي تعريف كردن آن تعيين مي شود. جاوا مجموعه غني از مشخصه هاي دستيابي را فراهم ساخته است.

مشخصه دستيابي جاوا عبارتند از `protected`, `private`, `public`. جاوا همچنين سطح دستيابي پيش فرضي را تعريف کرده است. مشخصه دستيابي `protected`، تنها زماني اعمال مي شود كه وراثت مطرح باشد.

وقتي عضوي از يك كلاس به وسيله مشخصه `public` تعريف مي شود، در آن صورت آن عضو توسط هر روتين ديگري قابل دستيابي خواهد بود. وقتي عضوي از يك كلاس به وسيله مشخصه `private` تعريف مي شود، در آن صورت تنها توسط ساير اعضاي كلاس خودش قابل دستيابي خواهد بود.

```
public int I ;
private double j ;
private int myMethod (ina a, char b) { / / . . . }
```

براي آشنائي با چگونگي کاربرد مسئله كنترل دستيابي در يك مثال عملي تر، نگارش بهبود يافته كلاس `stack` را در نظر بگيريد.

```
class stack {
    / * Now, broth stack and tos are private. This means
       that they cannot be accidentally or maliciously
       altered in a way that would be harmful to the stack.
```

---

<sup>14</sup>- encapsulation.

<sup>15</sup>- access specifier.

```

* /
private int stack [ ] = new int [10] ;
private int tos ;
/ / Initialize top-of-stack
Stack ( ) {
    tos = -1
}
/ / push an item onto the stack
void push (int item) {
    if (tos == 9)
        system.out.println (stack is full.);
    else.
        Stack [++tos] = item ;
}
/ / pop an item from the stack
int pop ( ) {
    if (tos < 0) {
        system.out.println ("stack underflow.") ;
        return 0;
    }
    else
        return stack [tos - -] ;
}
}

```

همان گونه که ملاحظه می‌کنید، اینک هم stack، که برای نگهداری پشته است، و هم tos، که به عنوان شاخص عنصر روی پشته به کار برده می‌شود، به صورت private تعریف شده اند. این بدین معناست که بدون استفاده از ( ) push و ( ) pop قابل دستیابی یا تغییر نمی باشند. Private ساختن tos، از دستیابی آن از دیگر بخشهای برنامه و تغییر عمدی مقدار آن به مقداری که در خارج از محدوده آرایه stack باشد، جلوگیری می‌کند.

عموماً، هر عضو از یک کلاس باید تنها در ارتباط با شیئی از همان کلاس خودش مورد دستیابی قرار گیرد. اما، این امکان وجود دارد که بتوان اعضای را ایجاد نمود که به تنهایی و بدون ارجاع به نمونه خاصی از کلاس، قابل استفاده باشند. برای ایجاد این گونه اعضا از کلمه کلیدی static در ابتدای سطر تعریف آنها استفاده کنید. وقتی عضوی از یک کلاس به صورت static تعریف می‌شود، دستیابی به آن پیش از ایجاد شیئی از همان کلاس، و بدون ارجاع هر گونه شی مقدور

خواهد بود. هم متدها و هم متغیرها را می‌توانید، به صورت static تعریف می‌شود، آن است که باید پیش از ایجاد هر شیئی قابل فراخوانی باشد.

نمونه متغیرهایی که به صورت static تعریف می‌شود عملاً متغیرهای عمومی می‌شوند. وقتی شی‌هایی از کلاس يك متغیر static تعریف می‌شوند هیچ نسخه‌ای از آن متغیر ایجاد نمی‌شود. در عوض، تمام نمونه‌های آن کلاس، متغیرهای static یکسانی را به اشتراک می‌گذارند.

متدهایی که به صورت static تعریف می‌شوند، چندین محدودیت دارند:

تنها سایر متدهای static را می‌توانند فراخوانند.

باید تنها با داده‌های static کار کنند.

به هیچ عنوان نمی‌توانند از this یا super استفاده کنند (کلمه کلیدی supper به وراثت مربوط می‌شود و در فصل آتی شرح داده شده است).

اگر برای تعیین مقدار اولیه متغیرهای static نیاز به انجام محاسبات دارید، می‌توانید بلوکی را به صورت static تعریف کنید تا دقیقاً تنها يك مرتبه هنگام بارگذاری اولیه کلاس اجرا شود. مثال زیر کلاسی را نشان می‌دهد که يك متد static چند متغیر static و يك بلوک static (برای مقدار دهی اولیه) دارد:

```
// Demonstrate static variable, methods, and blocks.
class useStatic {
    static int a = 3 ;
    static int b;
    static void meth (int x)  {
        system.out.println ("x = " +x)
        system.out.println ("a = " + a)
        system.out.println ("b = " +b)
    }
    static  {
        system.out.println ("static block initialized.");
        b = a* 4;
    }
    Public static void man (string args [ ] ) {
        meth (42) ;
    }
}
```



به محض اینکه کلاس UseStatic بارگذاری<sup>۱۶</sup> می‌شود. تمام عبارات static اجرا می‌شوند. نخست، مقدار ۳ به a تخصیص می‌یابد، سپس بلوک static اجرا می‌شود (پیامی را نمایش می‌دهد)، و آخر مقدار حاصل  $a*4$  به b تخصیص می‌یابد. سپس ( ) main فراخوانده می‌شود، و ۴۲ را هنگام فراخوانی ( ) meth برای x ارسال می‌کند. سه عبارت ( ) println، دو متغیر static (a و b) و همین‌طور متغیر محلی x را نمایش می‌دهند.

خروجی برنامه در ذیل نشان داده شده است:

```
Static block initialized
```

```
x = 42
```

```
a = 3
```

```
b = 12
```

متدها و متغیرهای ایستا را می‌توان در خارج از کلاسی که تعریف شده‌اند، مستقل از هر شیئی به کار برد. برای انجام این کار کافی است نام کلاس و سپس عملگر نقطه (.) را پیش از نامشان بنویسید. به عنوان مثال، چنانچه بخواهید متد ایستایی را از خارج کلاس خودش فرا بخوانید، می‌توانید این کار را با استفاده از فرم کلی زیر انجام دهید:

```
Classname.method ( )
```

### مروری بر آرایه‌ها

آرایه‌ها به صورت شیء پیاده‌سازی می‌شوند. به همین دلیل است که خصوصیت ویژه‌ای در رابطه با آرایه‌ها وجود دارد که بد نیست از آن بهره‌مند شوید. اندازه هر آرایه - یعنی، تعداد عناصری که در آرایه قابل ذخیره‌اند، در نمونه متغیر length آن است. تمام آرایه‌ها این متغیر را دارند، و اندازه آرایه همیشه در آن خواهد بود. برنامه زیر این خصوصیت را نشان می‌دهد:

```
/ / This program demonstrates the length array member.
```

```
class Length {
    public static void main (string args [ ] ) {
        int a1 [ ] = new int [10] ;
        int a2 [ ] = {3, 5, 7, 1, 8, 9, 44, -10}
        int a3 [ ] = {4, 3, 2, 1} ;
        system.out.println ("length of 1 is " + a1.length) ;
        system.out.println ("length of 2 is " + a1.length) ;
        system.out.println ("length 1 of 3 is " + a1.length) ;
    }
}
```

---

<sup>۱۶</sup>- Load.

```

    }
}

length of a1 is 10
length of a2 is 8
length of a3 is 4

```

همان گونه که ملاحظه می‌کنید، اندازه هر يك از آرایه ها نمایش داده شده است. به خاطر داشته باشید که مقدار length هیچ ارتباطی با تعداد عناصری که در آرایه وجود دارند، ندارد. مقدار آن تنها نمایانگر تعداد عناصری است که می‌توان در آرایه ذخیره کرد.

مقدمه ای بر کلاس‌های داخلی و تودرتو

امکان ایجاد يك کلاس در هر کلاس دیگر فراهم شده است؛ به این گونه کلاس ها، کلاس های تودرتو گفته می‌شود. محدود این کلاس ها به محدوده کلاسی که در آن قرار دارند، محدود می‌شود. از این رو، اگر کلاس B و در کلاس A تعریف شود، در آن صورت B برای A شناخته شده خواهد بود، اما در خارج از آن خیر. این کلاس‌ها به اعضای کلاسی که در آن تعریف شده اند، دستیابی دارند، از جمله اعضای private. اما در کلاس بیرونی به اعضای کلاسی بیرونی به اعضای کلاسی که در خودش تعریف شده است، دستیابی ندارد.

دو نوع کلاس تودرتو وجود دارد: ایستا و غیر ایستا. کلاس های تودرتوی ایستا، کلاس هایی هستند که از static برای تعریف آنها استفاده می‌شود. چون این کلاس ها ایستا هستند، باید از طریق يك شيء به اعضای کلاسی که در آن تعریف می‌شوند، دستیابی داشته باشند. یعنی، نمی‌توانند مستقیماً به اعضای آن کلاس دستیابی داشته باشند. کلاس های تودرتوی ایستا به دلیل محدودیت به ندرت مورد استفاده قرار می‌گیرند.

مهمترین نوع از کلاس های تودرتو، کلاس های داخلی<sup>۱۷</sup> هستند این کلاس ها، غیر ایستا می‌باشند و به تمام متغیرها و متدهای کلاس خارجی خود دستیابی دارند، و می‌توانند با همان روش خاص اعضای غیر ایستا، مستقیماً به آنها ارجاع داشته باشند. از این رو، هر کلاس داخلی کاملاً در محدوده کلاس در برگیرنده خود است.

---

<sup>17</sup> - Inner.

برنامه زیر چگونگی تعریف و استفاده از یک کلاس داخلی را نشان می‌دهد. کلاس Outer یک «نمونه متغیر» به نام outer-x، یک «نمونه متد» به نام test () دارد و کلاسی به نام Inner در آن تعریف می‌شود.

```
// Demonstrate an inner class.
class outer {
    int outer-x = 100 ;

    void test ( ) {
        Inner inner= new Inner ( );
        inner.display ( ) ;
    }
} // this is an inner class
class Inner {
    void display ( ) {
        system.out.println ("display: outer-x = " + outer - x) ;
    }
}

class InnerClassDemo {
    public static void main (string args [ ] ) {
        outer outer = new outer ( ) ;
        outer.test ( ) ;
    }
}

display: outer-x = 100
```

در این برنامه، یک کلاس داخلی به نام Inner در محدوده کلاس outer تعریف می‌شود. بنابراین، روتین های کلاس Inner می‌توانند مستقیماً به متغیر outer-x دسترسی داشته باشند. مهم است بدانید که کلاس Inner تنها در محدوده کلاس outer شناخته شده است. چنانچه روتینی در خارج از کلاس outer اقدام به ایجاد نمونه ای از کلاس Inner نماید، کامپایلر جاوا خطایی را نمایش خواهد داد. به طور کلی، کلاس های تودرتو، تفاوتی با سایر عناصر برنامه ندارند: تنها در محدوده ای که در آن تعریف شده اند، شناخته شده می‌باشند.

هر کلاس داخلی به تمام اعضای کلاسی که در آن تعریف شده است، دستیابی دارد، اما عکس این مطلب صادق نیست. اعضای کلاس داخلی تنها در محدوده همان کلاس شناخته شده اند و در کلاس خارجی قابل استفاده نیستند.

اگرچه کلاس های تودرتو در بیشتر برنامه های روزمره به کار برده نمی شود، اما هنگام مدیریت رویدادها در اپلت ها واقعاً مفید واقع می شوند.

استفاده از کلاس های تودرتو در مجموعه مشخصات 1.0 جاوا مجاز نبود. استفاده از آنها از جاوا ۱/۱ آغاز شده است.

## بررسی کلاس String

String احتمالاً متداولترین کلاس در کتابخانه کلاس های جاوا به شمار می آید. دلیل بارز این مطلب آن است که رشته ها بخش بسیار مهمی از برنامه سازی به شمار می آیند.

نخستین چیزی که باید درباره رشته ها بدانید، آن است که هر رشته ای که ایجاد می کنید، در واقع شیئی از کلاس String است. حتی ثابتهای رشته های هم شیء به شمار می آیند. به عنوان مثال در عبارت زیر،

```
System.out.println ("This is a string, too");
```

رشته "This is a string, too" نوعی ثابت رشته ای به شمار می آید. خوشبختانه، روش مدیریت ثابتهای رشته ای در جاوا همچون مدیریت رشته های «معمولی» در زبانهای کامپیوتری دیگر است، بنابراین از این جهت مشکلی نخواهیم داشت.

دومین مطلبی که باید درباره رشته ها بدانید آن است که شیءهای نوع String، تغییر ناپذیر هستند. یعنی پس از ایجاد شیءهای String، محتوای آنها قابل تغییر نخواهد بود. اگرچه این موضوع ممکن است محدودیت جدی به نظر آید، اما به دو دلیل این گونه نیست:

- اگر نیاز به تغییر رشته ای داشته باشید، همیشه می توانید نمونه جدیدی ایجاد کنید که متضمن تغییرات مورد نظر باشد.

- کلاسی نظیر String به نام StringBuffer در جاوا تعریف شده است که امکان تغییر رشته ها را فراهم می سازد، بنابراین تمام کارهای پردازش مربوط به رشته ها هنوز در جاوا قابل انجام هستند (StringBuffer در بخش دوم کتاب بررسی شده است).

روش هاي گوناگوني براي ايجاد رشته ها وجود دارد. آسانترين روش، استفاده از عبارتي چون مثال زير است:

```
String mystring = "this is a test" ;
```

پس از ايجاد يك شيء String، آن را مي توانيم در هر شرايطي كه کاربرد رشته ها مجاز است، به كار بريند.

عملگر "+" در جاوا براي شيء هاي نوع String تعريف شده است. از آن براي ادغام دو رشته استفاده مي شود. به عنوان مثال، نتيجه عبارت زير،

```
String my string = "I" + "like" + "Java" ;
```

نخيره شدن "I like Java" در MyString مي شود.

با استفاده از () equals مي توانيد تسلاوي دو رشته را بررسي كنيد. با فراخواني متد () length مي توانيد طول يك رشته به دست آوريد. با استفاده از () charAt هم مي توانيد كاراكثر موجود در موقعيت مورد نظر در رشته را به دست آوريد. شكل كلي اين سه متد در ذيل نشان داده شده است:

```
boolean equals (String object)
```

```
in length ( )
```

```
char charAt (in index)
```

### استفاده از آرگومان هاي خط فرمان

گاهي اوقات لازم مي شود كه اطلاعاتي را هنگام اجراي يك برنامه براي آن ارسال كنيم. اين كار با استفاده از ارسال آرگومان هاي خط فرمان<sup>18</sup> به () main انجام مي شود. منظور از آرگومان خط فرمان، اطلاعاتي است كه هنگام اجراي برنامه، مستقيماً پس از نام برنامه در خط فرمان نوشته مي شوند. دستيابي به آرگومان هاي خط فرمان در برنامه هاي جاوا كاملاً آسان است - اين اطلاعات به صورت رشته اي در آرايه String ارسالي به () main نگهداري مي شوند.

### Varargs: آرگومان هاي با طول متغير

---

<sup>18</sup> - Command-line.

J2SE 5، ویژگی جدیدی را به جاوا افزوده است که ایجاد متدهایی را ساده می‌کند که نیاز به تعداد متغیری آرگومان دارند. این ویژگی، `varargs` نامیده شده است و از کلمات `variable-length` یا `arguments` گرفته شده است. متدی که تعداد آرگومان هایش متغیر است، متد `variable-arity` یا صرفاً متد `varargs` نامیده می‌شود.

شرایطی که تعداد آرگومان های یک متد متغیر باشد، غیر معمول به شمار می‌آیند. به عنوان مثال، متدی که باری اتصال به اینترنت به کار می‌رود، نام کاربری، کلمه عبور، نام فایل، پروتکل و غیره را نیاز خواهد داشت، اما در صورت عدم تأمین برخی از آنها می‌توانید از مقادیر پیش فرض استفاده کنید. در چنین شرایطی می‌توانید تنها آرگومان هایی که مقادیر پیش فرض در خصوص آنها قابل استفاده نیستند را ارسال نمود.

تا پیش از عرضه J2SE 5، دو روش برای مدیریت آرگومان های با طول متغیر وجود داشت که هیچکدام از آنها خوشایند بود. نخست اینکه، اگر حداکثر تعداد آرگومان ها کوچک و مشخص بود، در آن صورت نگارشهای مختلفی از متد به صورت `overload` شده باری هر یک از حالات فراخوانی متد ایجاد می‌شد. اگرچه این روش برای برخی از شرایط عملیاتی بود، اما تنها برای شرایط بسیار محدودی کاربرد داشت.

در شرایطی که حداکثر تعداد آرگومان ها، بزرگتر یا نامعین بود، از روش دیگری استفاده می‌شد. در آن روش، آرگومان ها در آرایه ای نگهداری می‌شدند و سپس آرایه به متد ارسال می‌شد. آرگومان ها با طول متغیر به سه نقطه (...) مشخص می‌شوند.

هر متد می‌تواند پارامترهای «متعارفی» همراه با یک پارامتر با طول متغیر داشته باشد. اما پارامتر با طول متغیر باید هنگام تعریف متد، آخرین پارامتر باشد. به عنوان مثال، تعریف زیر کاملاً قابل قبول است:

```
int doIt (int a, int b, double c, int ... vals) {
```

به خاطر داشته باشید که `varargs` باید آخرین پارامتر باشد.

محدودیت دیگری نیز وجود دارد که باید نسبت به آن آگاه باشید: تنها یک پارامتر `varargs` مجاز است. به عنوان مثال، عبارت زیر نادرست است:

```
int doIt (int a, int b, double c, int ... vals, double ... morevals)
{ // Error!
```

تعریف پارامتر `Varargs` دوم غیر مجاز است.

## Overload کردن متدهای Varargs

متدهایی که آرگومان با طول متغیر دارند را نیز می‌تواند Overload نمود. به عنوان مثال، در برنامه زیر، متد vaTest() سه مرتبه Overload شده است:

```
// varargs and overloading
class varArgs3 {
    Static void vaTest (int... v) {
        System.out.print ("vaTest (int ...): " +
            " Number of args: " + v.length +
            " Contents: " );
        for (int x : v)
            system.out.print (x + " ");
        system.out.println ( ) ;
    }
    static void vaTest (boolean ...v) {
        system.out.print ("vaTest (boolean ...) " +
            " Number of args: " + length +
            " Contents: " ) ;
        for (boolean x : v)
            system.out.print (x + " ");
        system.out.print ( ) ;
    }
    Static void vaTest (String msg, int ... v) {
        system.out.print ("vaTest (String, in ...) : " +
            " msg + v.length +
            " Contents: " ) ;
        for (int x : v)
            system.out.print (x + " ");
        system.out.print ( ) ;
    }
}
public static void main (staring args [ ] )
{
    vaTest (1, 2, 3) ;
    vaTest (" Testing: ", 10, 20) ;
}
```

```
    vaTest (true, false, false);  
}  
}
```

```
vaTest (int ...): Number of arags: 3 contents: 1 2 3  
vaTest (string, int ...): Testing:: 2 contents: 10 20  
vaTest (int ...): Number of arags: 3 contents: true false false
```

این برنامه هر دو روش overload کردن متدهای varargs را نشان می‌دهد. نخست اینکه، نوع داده‌های پارامتر varargs ممکن است مختلف باشد. در خصوص متدهای vaTest (int ...) و vaTest (Boolean ...) همین گونه است. به خاطر داشته باشید که وجود "..." سبب می‌شود تا با پارامتر به صورت آرایه‌ای از نوع مشخص شده برخورد شود. روش دوم برای overload کردن متدهای varargs، افزودن یک پارامتر معمولی است.



وراثت

## Inheritance

عناوین این بخش :

مبانی وراثت

کاربرد کلمه کلیدی super

Multilevel

زمان فراخوانی Constructor ها

Override کردن متدها

توزیع (dispatch) پویای متدها

چرا متدهای لغو شده ؟

استفاده از کلاسهای مجرد (abstract)

استفاده از Final با وراثت

کلاس Object

وراثت را یکی از سنگ بناهای برنامه نویسی شی گراست ، زیرا امکان ایجاد طبقه بندیهای سلسله مراتبی را بوجود می آورد . با استفاده از وراثت ، می توانید یک کلاس عمومی بسازید که ویژگیهای مشترک یک مجموعه اقلام بهم مرتبط را تعریف نماید . این کلاس بعدا ممکن است توسط سایر کلاسها بارث برده شده و هر کلاس ارث برنده چیزهایی را که منحصر بفرد خودش باشد به آن اضافه نماید . در روش شناسی جاوا ، کلاسی که بارث برده می شود را کلاس بالا (superclass) می نامند . کلاسی که عمل ارث بری را انجام داده و ارث برده است را زیر کلاس (subclass) می نامند . بنابراین ، یک " زیر کلاس " روایت تخصصی تر و مشخص تر از یک " کلاس بالا " است . زیر کلاس ، کلیه متغیرهای نمونه و روشهای توصیف شده توسط کلاس بالا را بارث برده و منحصر بفرد خود را نیز اضافه می کند.

## مبانی وراثت

برای ارث بردن از یک کلاس ، خیلی ساده کافیت تعریف یک کلاس را با استفاده از واژه کلیدی extends در کلاس دیگری قرار دهید . برای فهم کامل این مطلب ، مثال ساده ای را نشان می دهیم . برنامه بعدی یک کلاس بالا تحت نام A و یک زیر کلاس موسوم به B ایجاد می کند . دقت کنید که چگونه از واژه کلیدی extends استفاده شده تا یک زیر کلاس از A ایجاد شود.

```
// A simple example of inheritance.
// Create a superclass.
class A {
    int i, j;
    void showij (){
        System.out.println("i and j : " + i + " " + j);
    }
}

// Create a subclass by extending class A.
class B extends A {
    int k;
    void showk (){
        System.out.println("k : " + k);
    }
}
```

```

void sum )({
System.out.println("j+j+k : " +( i+j+k));
}
}

class SimpleInheritance {
public static void main(String args[] ){
A superOb = new A) (;
B subOb = new B) (;
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb :");
superOb.showij) (;
System.out.println) (;
/* The subclass has access to all public members of
its superclass .*/
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb :");
subOb.showj) (;
subOb.showk) (;
System.out.println) (;
System.out.println("Sum of i/ j and k in subOb:");
subOb.sum) (;
}
}

```

خروجي این برنامه ، بقرار زیر می باشد :

Contents of superOb:

i and j :10 20

Contents of subOb:

i and j :7 8

k :9

```
Sum of i/ j and k in subOb:  
i+j+k :24
```

همانطوریکه می بینید ، زیر کلاس B دربرگیرنده کلیه اعضا کلاس بالایی مربوطه یعنی A است . بهمین دلیل است که subob می تواند به i و j دسترسی داشته و showij() را فراخوانی نماید . همچنین داخل sum() می توان بطور مستقیم i و j و همانگونه که قبلا بخشی از B بودند ، ارجاع نمود . اگرچه A کلاس بالایی B می باشد ، اما همچنان یک کلاس کاملاً مستقل و متکی بخود است . کلاس بالا بودن برای یک زیر کلاس بدان معنی نیست که نمی توان خود آن کلاس بالا را بتنهایی مورد استفاده قرار داد . بعلاوه ، یک زیر کلاس می تواند کلاس بالایی یک زیر کلاس دیگر باشد . شکل عمومی اعلان یک class که از یک کلاس بالا ارث می برد ، بصورت زیر است :

```
class subclass-name extends superclass-name {  
    // body of class  
}
```

برای هر زیر کلاسی که ایجاد می کنید ، فقط یک کلاس بالا می توانید تعریف کنید . جاوا از انتقال وراثت چندین کلاس بالا به یک کلاس منفرد پشتیبانی نمی کند . ( از این نظر جاوا با++C متفاوت است که در آن وراثت چند کلاسه امکان پذیر است . ) قبلاً گفتیم که می توانید یک سلسله مراتب از وراثت ایجاد کنید که در آن یک زیر کلاس ، کلاس بالایی یک زیر کلاس دیگر باشد . اما ، هیچ کلاسی نمی تواند کلاس بالایی خودش باشد .

### دسترسی به اعضا و وراثت

اگرچه یک زیر کلاس دربرگیرنده کلیه اعضا کلاس بالایی خود می باشد، اما نمیتواند به اعضای از کلاس بالا که بعنوان private اعلان شده اند ، دسترسی داشته باشد . بعنوان مثال ، سلسله مراتب ساده کلاس زیر را در نظر بگیرید :

```
/* In a class hierarchy/ private members remain  
private to their class.  
This program contains an error and will not  
compile.  
*/
```

```
// Create a superclass.
class A {
int i; // public by default
private int j; // private to A
void setij(int x/ int y ){
i = x;
j = y;
}
}

// A's j is not accessible here.
class B extends A {
int total;
void sum ){
total = i + j; // ERROR/ j is not accessible here
}
}

class Access {
public static void main(String args[] ){
B subOb = new B ();
subOb.setij(10/ 12);
subOb.sum) ();
System.out.println("Total is " + subOb.total);
}
}
```

این برنامه کامپایل نخواهد شد زیرا ارجاع به `j` داخل روش `sum()` در `B` ر سبب نقض دسترسی خواهد شد. از آنجاییکه `j` بعنوان `private` اعلان شده ، فقط توسط سایر اعضای کلاس خودش قابل دسترسی است و زیر کلاسها هیچگونه دسترسی به آن ندارند.

یادآوری : یک عضو کلاس که بعنوان `private` اعلان شده برای کلاس خودش اختصاصی خواهد بود. این عضو برای کدهای خارج از کلاسش از جمله زیر کلاسها ، قابل دسترسی نخواهد بود .

**یک مثال عملی تر**

اجازه دهید به يك مثال عملي تر پردازيم كه قدرت واقعي وراثت را نشان خواهد داد. در اینجا، روایت نهایی کلاس Box بنحوی گسترش یافته تا یک عنصر چهارم تحت نام weight را دربرگیرد. بدین ترتیب، کلاس جدید شامل depth, height, width و weight ویک box خواهد بود.

```
// This program uses inheritance to extend Box.
class Box {
double width;
double height;
double depth;
// construct clone of an object
Box(Box ob ){ // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w/ double h/ double d ){
width = w;
height = h;
depth = d;
}
// constructor used when all dimensions specified
Box ){
width =- 1; // use- 1 to indicate
height =- 1; // an uninitialized
depth =- 1; // box
}
// compute and return volume
double volume ){
return width * height * depth;
}
}
// Here/ Box is extended to include weight.
class BoxWeight extends Box {
double weight; // weight of box
```

```
// constructor for BoxWeight
BoxWeight(double w/ double h/ double d/ double m ){
width = w;
height = h;
depth = d;
weight = m;
}
}

class DemoBoxWeight {
public static void main(String args[] ){
Boxweight mybox1 = new BoxWeight(10/ 20/ 15/ 34.3);
Boxweight mybox2 = new BoxWeight(2/ 3/ 4/ 0.076);
double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
}
}
```

خروجي اين برنامه بصورت زیر می باشد :

```
Volume of mybox1 is 3000
Weight of mybox1 is 34.3
```

```
Volume of mybox2 is 24
Weight of mybox2 is 0.076
Boxweight
```

کلیه مشخصات Box را بارث برده و به آنها عنصر weight را اضافه می کند . برای Boxweight ضرورتی ندارد که کلیه جوانب موجود در Box را مجددا ایجاد نماید . بلکه می تواند بسادگی Box را طوری گسترش دهد تا اهداف خاص خودش را تأمین نماید .

یک مزیت عمده وراثت این است که کافیسف فقط یکبار یک کلاس بالا ایجاد کنید که خصلتهای مشترک یک مجموعه از اشیاء را تعریف نماید ، آنگاه می توان از آن برای

ایجاد هر تعداد از زیر کلاسهایی مشخص تر استفاده نمود. هر زیر کلاس می تواند دقیقاً با طبقه بندی خودش تطبیق یابد. بعنوان مثال ، کلاس بعدی ، از Box بارث برده و یک خصلت رنگ (color) نیز در آن اضافه شده است.

```
// Here/ Box is extended to include color.
class ColorBox extends Box {
int color; // color of box
ColorBox(double w/ double h/ double d/ double c ){
width = w;
height = h;
depth = d;
color = c;
}
}
```

بیاد آورید که هرگاه یک کلاس بالا ایجاد نمایید که وجوه عمومی یک شی را تعریف کند ، می توان از آن کلاس بالا برای تشکیل کلاسهایی تخصصی تر ارث برد . هر زیر کلاس خیلی ساده فقط خصلتهایی منحصر بفرد خودش را اضافه می کند . این مفهوم کلی وراثت است . یک متغیر کلاس بالا می تواند به یک شی زیر کلاس ارجاع نماید یک متغیر ارجاع مربوط به یک کلاس بالا را می توان به ارجاعی ، به هر یک از زیر کلاسهایی مشتق شده از آن کلاس بالا ، منتسب نمود . در بسیاری از شرایط ، این جنبه از وراثت کاملاً مفید و سودمند است . بعنوان مثال ، مورد زیر را در نظر بگیرید :

```
class RefDemo {
public static void main(String args[] ){
Boxweight weightbox = new BoxWeight(3/ 5/ 7/ 8.37);
Box plainbox = new Box();
double vol;
vol = weightbox.volume();
System.out.println("Volume of weightbox is " + vol);
System.out.println("Weight of weightbox is " +
weightbox.weight);
System.out.println();
// assign BoxWeight reference to Box reference
```



```

plainbox = weightbox;
vol = plainbox.volume(); // OK/ volume )(defined in Box
System.out.println("Volume of plainbox is " + vol);
/* The following statement is invalid because plainbox
dose not define a weight member .*/
// System.out.println("Weight of plainbox is " + plainbox.weight
}
}

```

در اینجا weightbox يك ارجاع به اشیاء Boxweight است و plainbox يك ارجاع به اشیاء Box است . از آنجاییکه Boxweight يك زیر کلاس از Box است ، می توان plainbox را بعنوان يك ارجاع به شیء weightbox منتسب نمود . نکته مهم این است که نوع متغیر ارجاع و نه نوع شیئی که به آن ارجاع شده است که تعیین می کند کدام اعضای قابل دسترسی هستند . یعنی هنگامیکه يك ارجاع مربوط به يك شیء زیر کلاس ، به يك متغیر ارجاع کلاس بالا منتسب می شود ، شما فقط به آن بخشهایی از شیء دسترسی دارید که توسط کلاس بالا تعریف شده باشند . بهمین دلیل است که plainbox نمی تواند به weight دسترسی داشته باشد حتی وقتی که به يك شیء Boxweight ارجاع می کند . اگر به آن فکر کنید ، آن را احساس می کنید زیرا يك کلاس بالا آگاهی و احاطه ای نسبت به موارد اضافه شده به زیرکلاس مربوطه اش نخواهد داشت . بهمین دلیل است که آخرین خط از کد موجود در قطعه قبلی از توضیح رج شده است . برای يك ارجاع Box امکان ندارد تا به فیلد weight دسترسی داشته باشد ، چراکه فیلدی با این نام توسط Box تعریف نشده است.

## کاربرد کلمه کلیدی super

در مثالهای قبلی کلاسهای مشتق شده از Box به کارایی و قدرتمندی که امکان داشت ، پیاده سازی نشدند . بعنوان مثال ، سازنده Boxweight بطور صریحی فیلدهای width ، height و depth در Box() را مقدار دهی اولیه می کند . این امر نه تنها کدهای پیدا شده در کلاس بالایی آنها را دو برابر می کند که غیر کاراست ، بلکه دلالت دارد بر اینکه يك زیر کلاس باید دسترسی به این اعضای داشته باشد . اما شرایطی وجود دارند که می خواهید يك کلاس بالا ایجاد کنید که جزئیات پیاده سازی خودش را خودش نگهداری کند . در این شرایط ، راهی برای يك زیر کلاس

وجود ندارد تا مستقیماً به این متغیرهای مربوط به خودش دسترسی داشته و یا آنها را مقداردهی اولیه نماید. از آنجاییکه کپسول سازی یک خصلت اولیه oop است، پس باعث تعجب نیست که جاوایه حلی برای این مشکل فراهم کرده باشد. هرگاه لازم باشد تایک زیر کلاس به کلاس بالایی قبلی خودش ارجاع نماید، اینکار را با استفاده از واژه کلیدی super انجام می دهیم. super دو شکل عمومی دارد. اولین آن سازنده کلاس بالا را فراخوانی می کند. دومین آن بمنظور دسترسی به یک عضو کلاس بالا که توسط یک عضو زیر کلاس مخفی مانده است، استفاده می شود.

### استفاده از super

یک زیرکلاس میتواند روش سازنده تعریف شده توسط کلاس بالایی مربوطه را بااستفاده از این شکل super فراخوانی نماید:

```
super( parameter-list);
```

در اینجا parameter-list مشخص کننده هر پارامتری است که توسط سازنده در کلاس بالا مورد نیاز باشد. super() باید همواره اولین دستور اجرا شده داخل یک سازنده زیر کلاس باشد.

بنگرید که چگونه از super() استفاده شده، و همچنین این روایت توسعه یافته از کلاس Boxweight() را در نظر بگیرید:

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
double weight; // weight of box
// initialize width, height, and depth using super()
BoxWeight(double w, double h, double d, double m ){
super(w, h, d); // call superclass constructor
weight = m;
}
}
```

در اینجا Boxweight() فراخوانی super() را با پارامترهای w، h و d انجام می دهد. این کار سبب فراخوانده شدن سازنده Box() شده با استفاده از این مقادیر width، height و depth و را مقدار دهی اولیه می کند. دیگر Boxweight خودش این مقادیر اولیه را مقدار دهی نمی کند. فقط کافی است تا مقدار منحصر بفرد خود weight را مقدار دهی اولیه نماید. این عمل Box را آزاد می گذارد تا در صورت تمایل این مقادیر را private بسازد. در مثال قبلی، super() با سه آرگومان فراخوانی شده بود. اما چون سازندگان ممکن است انباشته شوند، می توان super() را با استفاده از هر شکل تعریف شده توسط کلاس بالا فراخوانی نمود. سازنده ای که اجرا می شود، همانی است که با آرگومانها مطابقت داشته باشد. بعنوان مثال، در اینجا یک پیاده سازی کامل از Boxweight وجود دارد که سازندگان را برای طرق گوناگون و ممکن ساخته شدن یک box فراهم می نماید. در هر حالت super() با استفاده از آرگومانهای تقریبی فراخوانی میشود. دقت کنید که height، width و depth داخل Box بصورت اختصاصی درآمده اند.

```
// A complete implementation of BoxWeight.
class Box {
private double width;
private double height;
private double depth;
// construct clone of an object
Box(Box ob ){ // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d ){
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box () {
width = -1; // use -1 to indicate
```

```

height =- 1; // an uninitialized
depth =- 1; // box
}
// constructor used when cube is created
Box(double len ){
width = height = depth = len;
}
// compute and return volume
double volume (){
return width * height * depth;
}
}
// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob ){ // pass object to constructor
super(ob);
weight = ob.weight;
}
// constructor used when all parameters are specified
Box(double w, double h, double d, double m ){
super(w, h, d); // call superclass constructor
weight = m;
}
// default constructor
BoxWeight (){
super();
weight =- 1;
}
// constructor used when cube is created
BoxWeight(double len, double m ){
super(len);
weight = m;
}
}
class DemoSuper {
public static void main(String args[] ){

```

```

BoxWeight mybox1 = new BoxWeight(10 .20 .15 .34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
BoxWeight mybox3 = new BoxWeight(); // default
BoxWeight mycube = new BoxWeight(3, 2);
BoxWeight myclone = new BoxWeight(mybox1);
double vol;
vol = mybox1.vilume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();
vol = mybox2.vilume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
System.out.println();
vol = mybox3.vilume();
System.out.println("Volume of mybox3 is " + vol);
System.out.println("Weight of mybox3 is " + mybox3.weight);
System.out.println();
vol = myclone.vilume();
System.out.println("Volume of myclone is " + vol);
System.out.println("Weight of myclone is " + myclone.weight);
System.out.println();
vol = mycube.vilume();
System.out.println("Volume of mycube is " + vol);
System.out.println("Weight of mycube is " + mycube.weight);
System.out.println();
}
}

```

این برنامه خروجی زیر را تولید می کند :

```

Volume of mybox1 is 3000
Weight of mybox1 is 34.3

```

```

Volume of mybox2 is 24
Weight of mybox2 is 0.076

```

```

Volume of mybox3 is- 1

```

```
Weight of mybox3 is- 1
```

```
Volume of myclone is 3000
```

```
Weight of myclone is 34.3
```

```
Volume of mycube is 27
```

```
Weight of mycube is 2
```

توجه بیشتری نسبت به این سازنده در `Boxweight()` داشته باشید:

```
// construct clone of an object
BoxWeight(BoxWeight ob ){ // pass object to constructor
super(ob);
weight = ob.weight;
}
```

توجه کنید که `super()` با یک شیء سازنده `Boxweight` نه از نوع `Box` فراخوانی شده است و نیز سازنده `Box (ob)` را فراخوانی می کند. همانطوریکه قبلاً ذکر شد، یک متغیر کلاس بالا را می توان برای ارجاع به هر شیء مشتق شده از آن کلاس مورد استفاده قرار داد. بنابراین، ما قادر بودیم یک شیء `Boxweight` را به سازنده `Box` گذر دهیم. البته `Box` فقط نسبت به اعضای خودش آگاهی دارد.

اجازه دهید مفاهیم کلیدی مربوط به `super()` را مرور نماییم. وقتی یک زیرکلاس `super()` را فراخوانی می کند، در اصل سازنده کلاس بالایی بلافاصله خود را فراخوانی می کند. بنابراین `super()` همواره به کلاس بالایی بلافاصله قرار گرفته در بالایی کلاس فراخوانده شده، ارجاع می کند. این امر حتی در یک سلسله مراتب چند سطحی هم صادق است. همچنین `super` باید همواره اولین دستوری باشد که داخل یک سازنده زیر کلاس اجرا می شود.

## دومین کاربرد `super`

دومین شکل `super` تا حدودی شبیه `this` کار می کند، بجز اینکه `super` همواره به کلاس بالایی زیر کلاسی که در آن استفاده می شود، ارجاع می کند. شکل عمومی این کاربرد بصورت زیر است:

Super .member

در اینجا ، member ممکن است يك روش يا يك متغير نمونه باشد . این دومین شکل super برای شرایطی کاربرد دارد که در آن اسمی اعضای يك زیر کلاس ، اعضا با همان اسمی را در کلاس بالا مخفی می سازند . این سلسله مراتب ساده کلاس را در نظر بگیرید :

```
// Using super to overcome name hiding.
class A {
    int i;
}
// Create a subclass by extending class A.
class B extends A {
    int i; // this i hides the in A
    B(int a, int b ){
        super.i = a; // i in A
        i = b; // i in B
    }
    void show (){
        System.out.println("i in superclass :" + super.i);
        System.out.println("i in subclass :" + i);
    }
}
class UseSuper {
    public static void main(String args[] ){
        B subOb = new B(1, 2);
        subOb.show();
    }
}
```

این برنامه خروجی زیر را نمایش می دهد :

```
i in superclass :1
i in subclass :2
```

اگرچه متغیر نمونه i در B ر متغیر i در A ر را پنهان می سازد ، اما super امکان دسترسی به i تعریف شده در کلاس بالا بوجود می آورد . همانطوریکه خواهید دید همچنین میتوان از super برای فراخوانی روشهایی که توسط يك زیر کلاس مخفی شده اند.

## ایجاد يك سلسله مراتب چند سطحي (Multilevel)

مي توانيد سلسله مراتبي بسازيد كه شامل چندين لايه وراثت بدخواه شما باشند. كاملا موجه است كه از يك زير كلاس بعنوان كلاس بالاي يك كلاس ديگر استفاده كنيم. بعنوان مثال اگر سه كلاس A، B و C داشته باشيم آنگاه C مي تواند يك زير كلاس از B و يك زير كلاس از A باشد. وقتي چنين شرايطي اتفاق مي افتد، هر زير كلاس كليۀ خصلتهاي موجود در كليۀ كلاس بالاهاي خود را بارث مي برد. در اين شرايط، C كليۀ جنبه هاي B و A و را بارث مي برد. در برنامه بعدي، زير كلاس Boxweight بعنوان يك كلاس بالا استفاده شده تا زير كلاس تحت عنوان shipment را ايجاد نمايد. shipment كليۀ خصلتهاي Boxweight و Box را به ارث برده و يك فيلد بنام cost به آن اضافه شده كه هزينه كشتيراني يك محموله را نگهداري مي كند.

```
// Extend BoxWeight to include shipping costs.
// Start with Box.
class Box {
private double width;
private double height;
private double depth;
// construct clone of an object
Box(Box ob ){ // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d ){
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box () ({
width = 1; // use- 1 to indicate
```



```

height =- 1; // an uninitialized
depth =- 1; // box
}
// constructor used when cube is created
Box(double len ){
width = height = depth = len;
}
// compute and return volume
double volume(){
return width * height * depth;
}
}
// Add weight.
class BoxWeight extends Box {
double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob ){ // pass object to constructor
super(ob);
weight = ob.weight;
}
// constructor used when all parameters are specified
BoxWeight(double w, double h, double d, double m ){
super(w, h, d); // call superclass constructor
weight = m;
}
// default constructor
BoxWeight (){
super();
weight =- 1;
}
// constructor used when cube is created
BoxWeight(double len, double m ){
super(len);
weight = m;
}
}
// Add shipping costs
class Shipment extends BoxWeight {

```

```

double cost;
// construct clone of an object
Shipment(Shipment ob ){ // pass object to constructor
super(ob);
cost = ob.cost;
}
// constructor used when all parameters are specified
BoxWeight(double w, double h, double d, double m, double c ){
super(w, h, d); // call superclass constructor
cost = c;
}
// default constructor
Shipment (){
super();
cost =- 1;
}
// constructor used when cube is created
BoxWeight(double len, double m, double c ){
super(len, m);
cost = c;
}
}

class DemoShipment {
public static void main(String args[] ){
Shipment shipment1 = new Shipment(10, 20, 15, 10, 3.41);
Shipment shipment2 = new Shipment(2, 3, 4, 0.76, 1.28);
double vol;
vol = shipment1.volume();
System.out.println("Volume of shipment1 is " + vol);
System.out.println("Weight of shipment1 is " + shipment1.weight);
System.out.println("Shipping cost :$" + shipment1.cost);
System.out.println();
vol = shipment2.volume();
System.out.println("Volume of shipment2 is " + vol);
System.out.println("Weight of shipment2 is " + shipment2.weight);
System.out.println("Shipping cost :$" + shipment2.cost);
}
}

```

خروجي اين برنامه بصورت زير مي باشد :

```
Volume of shipment1 is 3000
```

```
Weight of shipment1 is 10
```

```
Shipping cost :$3.41
```

```
Volume of shipment2 is 24
```

```
Weight of shipment2 is 0.76
```

```
Shipping cost :$1.28
```

بدليل وراثت ، shipment مي تواند از كلاسهاي تعريف شده قبلي Box و Boxweight استفاده نمايد و فقط اطلاعات اضافي كه براي کاربرد خاص خودش نياز دارد ، اضافه نمايد . اين بخشي از ارزش وراثت است . وراثت امكان استفاده مجدد از كدهاي قبلي را بخوبي بوجود آورده است . اين مثال يك نکته مهم ديگر را نشان مي دهد ، super() همواره به سازنده موجود در نزديكترين كلاس بالا ارجاع مي كند . super() در shipment سازنده Boxweight را فراخواني ميكند . super() در Boxweight سازنده موجود در Box را فراخواني ميكند . در يك سلسله مراتب كلاس ، اگر يك سازنده كلاس بالا نيازمند پارامترها باشد ، آنگاه كلييه زير كلاسها بايد آن پارامترها را بالاي خط (up the line) بگذرانند . اين امر چه يك زير كلاس پارامترهاي خودش را نياز داشته باشد چه نياز نداشته باشد ، صحت خواهد داشت .

نکته : در مثال قبلي ، كل سلسله مراتب كلاس ، شامل Box ، Boxweight و shipment همگي در يك فايل نشان داده مي شوند . اين حالت فقط براي راحتي شما است . اما در جاوا ، هر يك از سه كلاس بايد در فايلهاي خاص خودشان قرار گرفته و جداگانه كامپايل شوند . در حقيقت ، استفاده از فايلهاي جداگانه يك مي و نه يك استثنا در ايجاد سلسله مراتب كلاسهاست .

## زمان فراخواني Constructor ها

وقتي يك سلسله مراتب كلاس ايجاد مي شود ، سازندگان كلاسها كه سلسله مراتب را تشكيل مي دهند به چه ترتيب فراخواني مي شوند ؟ بعنوان مثال ، با يك زير كلاس تحت نام B و يك كلاس بالا تحت نام A ، آيا سازنده A قبل از سازنده B فراخواني ميشود ، يا بالعكس ؟ پاسخ اين است كه در يك سلسله مراتب كلاس ، سازندگان بترتيب مشتق شدنشان از كلاس بالا به زير كلاس

فراخواني مي شوند . بعلاوه چون super() بايد اولين دستوري باشد كه در يك سازنده زير كلاس اجرا مي شود ، اين ترتيب همانطور حفظ مي شود ، خواه super() استفاده شود يا نشود . اگر super() استفاده نشود آنگاه سازنده پيش فرض يا سازنده بدون پارامتر هر يك از زير كلاسها اجرا خواهند شد . برنامه بعدي نشان مي دهد كه چه زماني سازندگان اجرا مي شوند :

```
// Demonstrate when constructors are called.
// Create a super class.
class A {
A () {
System.out.println("Inside A's constructor.")
}
}
// Create a subclass by extending class A.
class B extends A {
B () {
System.out.println("Inside B's constructor.")
}
}
// Create another subclass by extending B.
class C extends B {
C () {
System.out.println("Inside C's constructor.")
}
}
class CallingCons {
public static void main(String args[] ){
C c = new C();
}
}
```

خروجي اين برنامه بشرح زير مي باشد :

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

همانطوریکه مشاهده می کنید، سازندگان بترتیب مشتق شدنشان فراخوانی می شوند. اگر درباره آن تفکر کنید ، می فهمید که توابع سازنده بترتیب مشتق شدنشان اجرا می شوند . چون يك کلاس بالا نسبت به زیر کلاسهایی خود آگاهی ندارد ، هر گونه مقدار دهی اولیه که برای اجرا شدن نیاز داشته باشد، جدا از و احتمالا پیش نیاز هر گونه مقدار دهی اولیه انجام شده توسط زیر کلاس بوده و بنابراین ، باید اول این کار انجام شود.

## Override کردن متدها

در يك سلسله مراتب کلاس ، وقتی يك روش در يك زیر کلاس همان نام و نوع يك روش موجود در کلاس بالایی خود را داشته باشد، آنگاه میگویند آن روش در زیر کلاس ، روش موجود در کلاس بالا را لغو نموده یا از پیشروی آن جلوگیری می نماید . وقتی يك روش لغو شده از داخل يك زیر کلاس فراخوانی می شود ، همواره به روایتی از آن روش که توسط زیر کلاس تعریف شده ، ارجاع خواهد نمود و روایتی که کلاس بالا از همان روش تعریف نموده ، پنهان خواهد شد . مورد زیر را در نظر بگیرید :

```
// Method overriding.
class A {
    int i, j;
    A(int a, int b ){
        i = a;
        j = b;
    }
    // display i and j
    void show () {
        System.out.println("i and j : " + i + " " + j);
    }
}

class B extends A {
    int k;
    B(int a, int b, int c ){
        super(a, b);
        k = c;
    }
}
```

```

}
// display k -- this overrides show )(in A
void show (){
System.out.println("k :" + k);
}
}
class Override {
public static void main(String args[] ){
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show )(in B
}
}

```

حاصل تولید شده توسط این برنامه بقرار زیر می باشد :

K : 3

وقتی show() روی یک شیء از نوع B فراخوانی می شود ، روایتی از show که داخل B تعریف شده مورد استفاده قرار میگیرد. یعنی که ، روایت show() داخل B ، روایت اعلان شده در A را لغو می کند .

اگر می خواهید به روایت کلاس بالایی یک تابع لغو شده دسترسی داشته باشید ، این کار را با استفاده از super انجام دهید . بعنوان مثال ، در این روایت از B روایت کلاس بالایی show() داخل روایت مربوط به زیر کلاس فراخوانی خواهد شد . این امر به کلیه متغیرهای نمونه اجازه می دهد تا بنمایش درآیند .

```

class B extends A {
int k;
B(int a, int b, int c ){
super(a, b);
k = c;
}
void show (){
super.show(); // this calls A's show)(
System.out.println("k :" + k);
}
}

```

اگر این روایت از A را در برنامه قبلی جایگزین نمایید، خروجی زیر را مشاهده می کنید :

```
i and j :1 2
k:3
```

در اینجا، `super.show()` روایت کلاس بالایی `show()` را فراخوانی می کند . لغو روش فقط زمانی اتفاق می افتد که اسمی و نوع دو روش یکسان باشند. اگر چنین نباشد ، آنگاه دو روش خیلی ساده انباشته (overloaded) خواهند شد . بعنوان مثال ، این روایت اصلاح شده مثال قبلی را در نظر بگیرید:

```
// Methods with differing type signatures are overloaded -- not
// overridden.
class A {
    int i, j;
    A(int a, int b ){
        i = a;
        j = b;
    }
    // display i and j
    void show () {
        System.out.println("i and j : " + i + " " + j);
    }
}
// Create a subclass by extending class A.
class B extends A {
    int k;
    B(int a, int b, int c ){
        super(a, b);
        k = c;
    }
    // overload show()
    void show(String msg ){
        System.out.println(msg + k);
    }
}
class Override {
    public static void main(String args[] ){
```

```

B subOb = new B(1, 2, 3);
subOb.show("This is k :"); // this calls show (in B
subOb.show(); // this calls show (in A
}
}

```

حاصل تولید شده توسط این برنامه بقرار زیر می باشد :

```

This is k:3
i and j :1 2

```

روایت show() در B ر يك پارامتر رشته ای می گیرد. این عمل سبب متفاوت شدن تاییدیه نوع آن از نوع موجود در A شده ، که هیچ پارامتری را نمی گیرد. بنابراین نباشستگی ( یا مخفی شدن اسم ) اتفاق نمی افتد.

### توزیع (dispatch) پویای متدها

اگر در لغو روشها چیزی فراتر از يك قرارداد فضایی نام وجود نداشت ، آنگاه این عمل در بهترین حالت ، ارضا نوعی حس کنجکاو و فاقد ارزش عملی بود. اما این چنین نیست . لغو روش تشکیل دهنده اساس یکی از مفاهیم پر قدرت در جاوا یعنی " توزیع پویای روش " است . این يك مکانیسم است که توسط آن يك فراخوانی به تابع لغو شده در حین اجرا ، در عوض زمان کامپایل ، از سر گرفته می شود. توزیع پویای روش مهم است چون طریقی است که جاوا با آن چند شکلی را درست حین اجرا پیاده سازی می نماید.

توضیح را با تکرار يك اصل مهم شروع میکنیم : يك متغیر ارجاع کلاس بالا میتواند به يك شیء زیر کلاس ارجاع نماید . جاوا از این واقعیت استفاده کرده و فراخوانی به روشهای لغو شده را حین اجرا از سر می گیرد . وقتی يك روش لغو شده از طریق يك ارجاع کلاس بالا فراخوانی می شود، جاوا براساس نوع شیء ارجاع شده در زمانی که فراخوانی اتفاق می افتد ، تعیین می کند که کدام روایت از روش باید اجرا شود .

بنابراین ، عمل تعیین روایت خاص از يك روش ، حین اجرا انجام می گیرد . وقتی به انواع مختلف اشیاء ارجاع شده باشد، روایتهای مختلفی از يك روش لغو شده فراخوانی خواهند شد . بعبارت دیگر ، این نوع شیء ارجاع شده است ) نه نوع متغیر ارجاع ( که تعیین می کند کدام



روایت از روش لغو شده باید اجرا شود . بنابراین اگر يك كلاس بالا دربرگیرنده يك روش لغو شده توسط يك زیر كلاس باشد ، آنگاه زمانی که انواع مختلف اشیاء از طریق يك متغیر ارجاع كلاس بالا مورد ارجاع قرار می گیرند روایتهای مختلف آن روش اجرا خواهند شد .  
در اینجا مثالی را مشاهده میکنید که توزیع پویای روش را به شما نشان میدهد :

```
// Dynamic Method Dispatch
class A {
void callme () {
System.out.println("Inside A's callme method");
}
}
class B extends A {
// override callme()
void callme () {
System.out.println("Inside B's callme method");
}
}
class C extends A {
// override callme()
void callme () {
System.out.println("Inside C's callme method");
}
}
class Dispatch {
public static void main(String args[] ){
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
```

```
}  
}
```

خروجي اين برنامه بقرار زير مي باشد :

```
Inside A's callme method  
Inside B's callme method  
Inside C's callme method
```

اين برنامه يك كلاس بالاي تحت نام A و دو زير كلاس آن تحت نام B و C و را ايجاد مي كند. زير كلاسهاي B و C و سبب لغو callme() اعلان شده در A مي گردند. درون روش main() اشيائي از نوع A، B و C اعلان شده اند. همچنين يك ارجاع از نوع A بنام r اعلان شده است. سپس برنامه يك ارجاع به هر يك از انواع اشيا به r را نسبت داده و از آن ارجاع براي فراخواني callme() استفاده مي كند. همانطوريكه حاصل اين برنامه نشان مي دهد، روايتي از callme() كه بايد اجرا شود توسط نوع شئي كه در زمان فراخواني مورد ارجاع قرار گرفته، تعيين مي شود. اگر اين تعيين توسط نوع متغير ارجاع يعني r انجام ميگرفت شما با سه فراخواني به روش callme() مربوط به A مواجه مي شديد.

نکته : کسانی که با ++C آشنا هستند تشخیص می دهند که روشهای لغو شده در جاوا مشابه توابع مجازی (virtual functions) در ++C هستند.

### چرا متدهاي لغو شده ؟

قبلا هم گفتيم كه روشهاي لغو شده به جاوا اجازه پشتيباني از چند شكلي حين اجرا را مي دهند. چند شكلي به يك دليل براي برنامه نويس شي ئ گرا لازم است : اين حالت به يك كلاس عمومي اجازه مي دهد تا روشهايي را مشخص نمايد كه براي كلييه مشتقات آن كلاس مشترك باشند، و به زير كلاس ها اجازه مي دهد تا پياده سازيهاي مشخص برخي يا كلييه روشها را تعريف نمايند. روشهاي لغو شده راه ديگري براي جاوا است تا " يك رابط و چندين روش " را بعنوان يكي از وجوه چند شكلي پياده سازي نمايد.

بخشي از كليد كاربرد موفقيت آميز چند شكلي، درك اين نكته است كه كلاس بالاها و زير كلاسها يك سلسله مراتب تشكيل ميدهند كه از مشخصات كوچكتر به بزرگتر حركت مي كنند. اگر كلاس

بالا بدرستي استفاده شود، کلیه اجزائی که يك زیر کلاس مي تواند بطور مستقیم استفاده نماید ، تعریف مي کند. این امر به زیر کلاس قابلیت انعطاف تعریف روشهای خودش را مي دهد ، که همچنان يك رابط منسجم را بوجود مي آورد . بنابراین ، بوسیله ترکیب وراثت با روشهای لغو شده ، يك کلاس بالا مي تواند شکل عمومی روشهایی را که توسط کلیه زیر کلاسهای مربوطه استفاده خواهند شد را تعریف نماید.

چند شکلي پويا و حين اجرا يکي از قدرتمندترین مکانیسمهایی است که طراحی شي گرایي را مجهز به استفاده مجدد و تنومندي کدها نموده است . این ابزار افزایش دهنده قدرت کتابخانه های کدهای موجود برای فراخواني روشهای روي نمونه های کلاسهای جديد بدون نیاز به کامپایل مجدد مي باشد در حالیکه يك رابط مجرد و زیبا را نیز حفظ مي کنیم .

### بکار بردن لغو روش

اجازه دهید تا به يك مثال عملي تر که از لغو روش استفاده مي کند ، نگاه کنیم . برنامه بعدي يك کلاس بالا تحت نام Figure را ايجاد مي کند که ابعاد اشيا مختلف دو بعدي را ذخيره مي کند . این برنامه همچنین يك روش با نام area() را تعریف مي کند که مساحت يك شي را محاسبه مي کند. برنامه ، دو زیر کلاس از Figure مشتق مي کند . اولين آن Rectangle و دومين آن Triangle است . هر يك از این زیر کلاسها area() را طوري لغو میکنند که بترتيب مساحت يك مستطیل و مثلث را برگردان کنند.

```
// Using run-time polymorphism.
class Figure {
double dim1;
double dim2;
Figure(double a, double b ){
dim1 = a;
dim2 = b;
}
double area () {
System.out.println("Area for Figure is undefined.");
return 0;
}
```

```

}
class Rectangle extends Figure {
Rectangle(double a, double b ){
super(a, b);
}
// override area for rectangle
double area (){
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure {
Triangle(double a, double b ){
super(a, b);
}
// override area for right triangle
double area (){
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}
class FindAreas {
public static void main(String args[] ){
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " + figref.area)();
figref = t;
System.out.println("Area is " + figref.area)();
figref = f;
System.out.println("Area is " + figref.area)();
}
}

```

حاصل این برنامه بقرار زیر است :

```
Inside Area for Rectangle.  
Area is 45  
Inside Area for Triangle.  
Area is 40  
Area for Figure is undefined.  
Area is 0
```

از طریق مکانیسم دوگانه وراثت و چند شکلی حین اجرا ، امکان تعریف يك رابط منسجم که برای چندین نوع اشیای مختلف ، اما بهم مرتبط ، استفاده می شود ، وجود دارد . در این حالت ، اگر از Figure يك شي ئ مشتق شود ، پس با فراخواني area() می توان مساحت آن شي ئ را بدست آورد . رابط مربوط به این عملیات صرفنظر از نوع شکل هندسی مورد استفاده ، همیشه یکسان است.

### استفاده از کلاسهای مجرد (abstract)

شرایطی وجود دارد که می خواهید يك کلاس بالا تعریف نمایید که ساختار يك انتزاع معین را بدون يك پیاده سازی کامل از هر روشی ، اعلان نماید. یعنی گاهی می خواهید يك کلاس بالا ایجاد کنید که فقط يك شکل عمومی شده را تعریف کند که توسط کلیه زیر کلاسهایش باشتراک گذاشته خواهد شد و پر کردن جزئیات این شکل عمومی بعهد هر يك از زیر کلاس ها واگذار می شود . يك چنین کلاسی طبیعت روشهایی که زیر کلاسها باید پیاده سازی نمایند را تعریف می کند . يك شیوه برای وقوع این شرایط زمانی است که يك کلاس بالا توانایی ایجاد يك پیاده سازی با معنی برای يك روش را نداشته باشد . تعریف area() خیلی ساده يك نگهدارنده مکان (place holder) است . این روش مساحت انواع شي ئ را محاسبه نکرده و نمایش نمی دهد . هنگام ایجاد کتابخانه های خاص کلاس خود ، خواهید دید که غیر معمول نیست اگر يك روش هیچ تعریف بامعنی در متن (context) کلاس بالای خود نداشته باشد . این شرایط را بدو طریق می توانید اداره نمایید . يك طریق این است که يك پیام هشدار (warning) گزارش نمایید . اگرچه این روش در برخی شرایط خاص مثل اشکال زدایی (debugging) مفید است ، اما روش دائمی نیست . ممکن است روشهایی داشته باشید که باید توسط زیر کلاس لغو شوند تا اینکه آن زیرکلاس معنادار بشود.

کلاس Triangle را در نظر بگیرید . اگر area() تعریف نشود، این کلاس هیچ معنایی ندارد . در این حالت ، شما بدنبال راهی هستید تا مطمئن شوید که يك زیر کلاس در حقیقت کلیه روشهای ضروری را لغو می کند . راه حل جاوا برای این مشکل روش مجرد abstract method است . می توانید توسط زیر کلاسها و با مشخص نمودن اصلاح کننده نوع abstract ، روشهای معینی را لغو نمایید . به این روشها گاهی subclasser responsibility اطلاق میشود ، زیرا آنها هیچ پیاده سازی مشخص شده ای در کلاس بالا ندارند . بنابراین يك زیرکلاس باید آنها را لغو نماید چون نمی تواند بسادگی روایت تعریف شده در کلاس بالا را استفاده نماید . برای اعلان يك روش مجرد ، از شکل عمومی زیر استفاده نمایید .

```
abstract type name( parameter-list);
```

همانطوریکه مشاهده می کنید در اینجا بدنه روش معرفی نشده است . هر کلاسی که دربرگیرنده يك یا چند روش مجرد باشد ، باید بعنوان مجرد اعلان گردد . برای اعلان يك کلاس بعنوان مجرد ، بسادگی از واژه کلیدی abstract در جلوی واژه کلیدی class در ابتدای اعلان کلاس استفاده می نمایید . برای يك کلاس مجرد هیچ شیئی نمی توان ایجاد نمود . یعنی يك کلاس مجرد نباید بطور مستقیم با عملگر new نمونه سازی شود . چنان اشیائی بدون استفاده هستند ، زیرا يك کلاس مجرد بطور کامل تعریف نشده است . همچنین نمی توانید سازندگان مجرد یا روشهای ایستای مجرد اعلان نمایید . هر زیر کلاس از يك کلاس مجرد باید یا کلیه روشهای مجرد موجود در کلاس بالا را پیاده سازی نماید ، و یا خودش بعنوان يك abstract اعلان شود . در اینجا مثال ساده ای از يك کلاس با يك روش مجرد مشاهده می کنید که بعد از آن يك کلاس قرار گرفته که آن روش را پیاده سازی می کند :

```
// A Simple demonstration of abstract.
abstract class A {
    abstract void callme();
    // concrete methods are still allowed in abstract classes
    void callmetoo () {
        System.out.println("This is a concrete method.");
    }
}
class B extends A {
```

```

void callme () {
    System.out.println("B's implementetion of callme.");
}
}

class AbstractDemo {
    public static void main(String args[] ) {
        B b = new B();
        b.callme();
        b.callmetoo();
    }
}

```

توجه کنید که هیچ شیئی از کلاس A در برنامه اعلان نشده است . همانطوریکه ذکر شد ، امکان نمونه سازی یک کلاس مجرد وجود ندارد . یک نکته دیگر : کلاس A یک روش واقعی با نام callmetoo() را پیاده سازی می کند . این امر کاملاً مقبول است. کلاسهای مجرد می توانند مادامیکه تناسب را حفظ نمایند ، دربرگیرنده پیاده سازیها باشند.

اگرچه نمی توان از کلاسهای مجرد برای نمونه سازی اشیای استفاده نمود، اما از آنها برای ایجاد ارجاعات شی می توان استفاده نمود زیرا روش جاوا برای چند شکلی حین اجرا از طریق استفاده از ارجاعات کلاس بالا پیاده سازی خواهد شد . بنابراین ، باید امکان ایجاد یک ارجاع به یک کلاس مجرد وجود داشته باشد بطوریکه با استفاده از آن ارجاع به یک شی زیر کلاس اشاره نمود . شما استفاده از این جنبه را در مثال بعدی خواهید دید.

با استفاده از یک کلاس مجرد، می توانید کلاس Figure را توسعه دهید. چون مفهوم با معنایی برای مساحت یک شکل دو بعدی تعریف نشده وجود ندارد ، روایت بعدی این برنامه area() را بعنوان یک مجرد داخل Figure اعلان می کند . این البته بدان معنی است که کلیه کلاسهای مشتق شده از Figure باید area() را لغو نمایند .

```

// Using abstract methods and classes.
abstract class Figure {
    double dim1;
    double dim2;
    Figure(double a, double b ) {
        dim1 = a;
        dim2 = b;
    }
}

```

```

}
// area is now an abstract method
abstract double area();
}
class Rectangle extends Figure {
Rectangle(double a, double b ){
super(a, b);
}
// override area for rectangle
double area () {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure {
Triangle(double a, double b ){
super(a, B);
}
// override area for right triangle
double area () {
System.out.println("Inside Area for Teriangle.");
return dim1 * dim2 / 2;
}
}
class AbstractAreas {
public static void main(String args[] ){
// Figure f = new Figure(10, 10); // illegal now
Rectangle r = new Rectanlge(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref; // this is OK/ no object is created
figref = r;
System.out.println("Area is " + figref.area)();
figref = t;
System.out.println("Area is " + figref.area)();
}
}

```



همانطوریکه توضیح درون `main()` نشان می دهد ، دیگر امکان اعلان اشیای از نوع `Figure` وجود ندارد ، چون اکنون بصورت مجرد است . کلیه زیر کلاسهای `Figure` باید `area()` را لغو نمایند . برای اثبات این امر ، سعی کنید یک زیر کلاس ایجاد نمایید که `area()` را لغو نمی کند . حتماً یک خطای `comple-time` در زمان کامپایل دریافت می کنید .

اگرچه امکان ایجاد یک شی از نوع `Figure` وجود ندارد ، اما می توانید یک متغیر ارجاع از نوع `Figure` ایجاد نمایید . متغیر `fighref` بعنوان ارجاعی به `Figure` اعلان شده و بدان معنی است که با استفاده از آن می توان به یک شی از هر کلاس مشتق شده از `Figure` ، ارجاع نمود . همانطوریکه توضیح دادیم ، تعیین اینکه کدام نگارش از متدهای `override` شده باید در زمان اجرا فعال شوند ، از طریق متغیرهای ارجاع به فوق کلاس ها صورت میگیرد .

### استفاده از **Final** با وراثت

واژه کلیدی `final` سه کاربرد دارد . اول برای ایجاد مشابه یک ثابت اسم دار . دو کاربرد دیگر مربوط به وراثت هستند و بررسی خواهند شد . استفاده از `final` برای ممانعت از لغو کردن در حالیکه لغو کردن روش یکی از جنبه های قدرتمند جاوا است ، اما زمانهایی وجود دارند که می خواهید مانع وقوع لغو روش گردید . برای غیر مجاز کردن لغو یک روش ، `final` را بعنوان یک اصلاحگر در شروع اعلان آن مشخص نمایید . روشهای اعلان شده بعنوان `final` نمی توانند لغو شوند . قطعه بعدی نشان دهنده `final` است :

```
class A {
    final void meth () {
        System.out.println("This is a final method.");
    }
}

class B extends A {
    void meth () { // ERROR! Can't override.
        System.out.println("Illegal!");
    }
}
```

چون `meth()` بعنوان `final` اعلان شده ، نمی توان آن را در `B` لغو نمود . اگر چنین تلاشی بکنید ، نتیجه یک خطای `complle-time` خواهد بود . روشهای اعلان شده بعنوان `final` گاهی می توانند باعث افزایش عملکرد شوند:

چون کامپایلر می داند که آنها توسط یک زیر کلاس لغو نخواهند شد، کامپایلر از فراخوانی `inline` به آنها آزاد می شود. وقتی یک تابع کوچک `final` فراخوانی میشود اغلب کامپایلر جاوا می تواند کد بایستی را برای زیر روال مستقیم درون خطی (`inline`) کد کامپایل شده در روش فراخواننده کپی نماید، و بدین ترتیب بالاسریهای پر هزینه همراه یک فراخوان روش را حذف می نماید `Inlining` . فقط یک گزینه با روشهای `final` است . بطور طبیعی ، جاوا فراخوانی به روشها را بصورت پویا و در زمان حین اجرا حل و فصل می کند. این عمل را `late binding` می نامند . اما از آنجاییکه روشهای `final` را نمی توان لغو نمود ، یک فراخوانی را می توان در زمان کامپایل حل و فصل نمود . این را `early binding` می نامند.

### استفاده از `final` برای جلوگیری از وراثت

گاهی ممکن است بخواهید مانع ارث بردن از یک کلاس بشوید . برای انجام اینکار قبل از اعلان کلاس از واژه کلیدی `final` استفاده نمایید. اعلان نمودن یک کلاس بعنوان `final` بطور ضمنی کلیه روشهای آن را نیز بعنوان `final` اعلان میکند. حتما می دانید که اعلان یک کلاس بعنوان هم `abstract` و هم `final` غیر مجاز است چون یک کلاس `abstract` بتنهایی کامل نبوده و برای تکمیل پیاده سازیها متکی به زیرکلاسهایی خود می باشد . در اینجا یک مثال از کلاس `final` را مشاهده می کنید :

```
final class A {  
    //...  
}  
  
// The following class is illegal.  
class B extends A { // ERROR! Can't subclass A  
    //...  
}
```

همانطوریکه از " توضیحات " مشخص شده ، ارث بردن B از A ز غیر مجاز شده چون A بعنوان final اعلان شده است.

## کلاس Object

کلاس شي The object class يك كلاس مخصوص يعني object توسط جاوا تعريف شده است. كليه كلاسهاي ديگر زير كلاسهاي object هستند. يعني object كلاس بالايي كليه كلاسهاي ديگر است. بدین ترتیب يك متغیر ارجاع نوع object می تواند به يك شي از هر كلاس ديگري ارجاع نماید. همچنین ، نظریه اینکه آرایه ها بعنوان كلاس پیاده ساري می شوند ، يك متغیر از نوع object می تواند به هر آرایه اي ارجاع نماید. Object روشهاي بعدي را تعريف مي كند. كه بدین ترتیب همه آنها در هر شیئي قابل دسترسی هستند .

### متد

### عملکرد

**object clone()**

شي جديدي ايجاد مي كند كه تكثير شده همان شي است .

**boolean equals ( Object object )**

تعيين مي كند كه آیا شي با شي ديگر مساوي است يا نه.

**void finalize()**

قبل از اینکه يك شي بدون استفاده در چرخه مجدد قرار گیرد ، فراخواني ميشود .

**class get class()**

كلاس يك شي را حين اجرا نگهداري ميكند.

**int hash code()**

كد hash همراه با اشیاء فراخوان را برگردان مي كند .

**void notify()**

اجراي يك رشته در حال انتظار روي شي فراخوان كننده را از سر مي گيرد .

**void notify All()**

اجراي كليه رشته هاي در حال انتظار روي شي و فراخوان كننده را از سر مي گيرد .

**string to string()**

رشته اي كه توصیف كننده شي و است را برمي گرداند .

روي يك نخ (thread) ديگر از اجرا منتظر

**void wait()**

مي ماند.

**void wait ( long milliseconds )**

**Void wait ( long milliseconds, int nanoseconds)**

روشهاي `getClass()` ، `notify()` ، `notify All()` و `wait` و به عنوان `final` اعلان مي شوند. ممکن است ساير موارد را لغو نماييد . فعلا به دو روش دقت نماييد :

`equals()` و `toString()` . متد `equals()` محتوي دو شي را با يکديگر مقايسه نموده و اگر دو شي معادل هم باشند ، اين روش `true` را برمي گرداند و در غير اينصورت `false` را برمي گرداند . روش `toString()` رشته اي که دربرگيرنده توصيفي از شي که روي آن فراخواني شده ، را برمي گرداند . اين روش همچنين هرگاه يك شي ئ حاصل استفاده از `println()` باشد بطور خودکار فراخواني خواهد شد . بسياري از کلاسها اين روش را لغو مي کنند. انجام اين کار به آن کلاسها امکان مي دهد تا شرحي را مشخصا براي نوع شي هايي که ايجاد مي کنند ، آماده سازند.

# بسته ها و رابطها

## عناوین این بخش :

مدل زبان و کاربردهای آن

بسته ها و رابطها

بسته ها packages

تعریف يك بسته

درك مفهوم CLASS PATH

محافظت دسترسی Access protection

وارد کردن بسته ها

رابطها interfaces

تعریف نمودن يك رابط

پیاده سازی رابطها

دسترسى به پیاده سازیها از طریق ارجاعات رابط

پیاده سازی نسبي (partial)

بکار بردن رابطها

متغیرها در رابطها

## بسته ها و رابطها

بسته ها (packages) ، ظروفی برای کلاسها هستند که برای نگهداری فضاهای تقسیم بندی شده اسامی کلاسها استفاده می شوند . بعنوان مثال ، يك بسته به شما اجازه می دهد تا يك کلاس با نام list ایجاد نموده که آن را در بسته خود ذخیره نمایید بدون نگرانی از اینکه این کلاس با يك کلاس دیگر با نام list و ذخیره شده در جاي دیگری تلاقی داشته باشد. بسته ها بصورت سلسله مراتبی ذخیره شده و بطور صریح به تعریف کلاس جدید وارد می شوند .

از طریق استفاده از واژه کلیدی interface ، جاوا به شما اجازه می دهد تا رابط را از پیاده سازی مربوط به آن کاملاً مجرد نمایید. با استفاده از interface می توانید يك مجموعه از روشهایی که قادرند برای يك یا چندین کلاس پیاده سازی شوند را مشخص نمایید. خود interface بواقع هیچ نوع پیاده سازی را تعریف نمیکند . اگر رابطها مشابه با کلاسهای مجرد abstract هستند ، اما يك ظرفیت بیشتر دارند : يك کلاس می تواند بیش از يك رابط را پیاده سازی نماید. اما، يك کلاس فقط میتواند از يك کلاس بالایی تکی ( مجرد یا غیره ) ارث ببرد.

بسته ها (packages) و رابطها (interfaces) دو عنصر اصلی در يك برنامه جاوا می باشند . بطور کلی ، يك فایل منبع جاوا می تواند دربرگیرنده یکی از ( یا کلیه ) بخشهای داخلی چهارگانه زیر باشد:

يك دستور بسته ای تکی ( اختیاری ) A single package statement

به هر تعداد دستورات وارده ( اختیاری ) Any number of import statements

يك اعلان کلاس تکی بعنوان public ( اجباری ) A single public class declaration

به هر تعداد کلاسهای اختصاصی برای بسته ( اختیاری )

Any number of classes private to the package

تا بحال در مثالهای مشاهده شده فقط یکی از این موارد یعنی اعلان کلاس تکیه است.

## بسته ها packages

قبلاً نام هر مثالی از کلاس را از يك فضای نام مخصوص (name space) می گرفتیم . یعنی برای هر کلاس باید يك نام منحصر بفرد استفاده می شد تا از اختلاط نامها جلوگیری شود. بدون

يك راه مناسب براي مديريت فضاي اسامي ، پس از مدتي دچار مشكلاتي خواهيد شد و مجبوريد براي هر كلاس منفرد ، اسامي توصيفي و مشكلي اختيار كنيد. همچنين نيازمند راهي هستيد تا مطمئن شويد كه ناميكه براي يك كلاس انتخاب مي كنيد بطور منطقي منحصر بفرد بوده و با نام كلاسهاي انتخاب شده توسط ساير برنامه نويسان تصادم پيدا نخواهد كرد . خوشبختانه جاوا مكانيسمي براي بخش بندي فضاي نام كلاس به قطعات قابل توجه (chunks) و قابل مديريت فراهم آورده است . اين مكانيسم همان بسته يا package است . بسته هم يك روش نامگذاري و هم يك مكانيسم كنترل رويت پذيري است. مي توانيد كلاسهايي را داخل يك بسته تعريف كنيد كه بوسيله كدهاي خارج از بسته قابل دسترسي نباشند . همچنين مي توانيد اعضاي كلاسي را تعريف كنيد كه فقط در معرض رويت ساير اعضا همان بسته قرار داشته باشند . اين امر به كلاسهاي شما امكان مي دهد تا يك آگاهي دروني از يكدیگر داشته باشند اما آن اطلاعات را براي ديگران افشاگري نکنند .

### تعريف يك بسته

ايجاد يك بسته بسيار آسان است :

خيلي ساده يك فرمان package بعنوان اولين دستور در فايل منبع جاوا بگنجانيد. هر كلاس اعلان شده داخل آن فايل به بسته مشخص شده تعلق خواهد داشت . دستور package يك فضاي نام را تعريف مي كند كه كلاسها داخل آن ذخيره مي شوند . اگر دستور فوق را حذف نماييد ، اسامي كلاس در بسته پيش فرض قرار مي گيرند، كه هيچ اسمي ندارد. اگرچه بسته پيش فرض براي برنامه هاي کوتاه و نمونه مناسب است ، اما براي برنامه هاي کاربردي واقعي كفايت نمي كند . اكثر اوقات ، يك بسته براي كدهاي شما تعريف مي كنيم. شكل عمومي دستور package بصورت زير است :

```
package pkg;
```

در اينجا pkg نام بسته است. بعنوان مثال ، دستور بعدي يك بسته تحت نام Mypackage ايجاد مي كند :

```
package MyPackage;
```

جاوا از دایرکتوریهای فایل سیستم برای ذخیره سازی بسته ها استفاده می کند . بعنوان مثال فایلهای class برای کلاسهای که بعنوان بخشی از Mypackage اعلان می کنید ، باید در يك دایرکتوری تحت نام Mypackage ذخیره شوند. بیاد آورید که جاوا بسیار حساس است ، بنابراین اسم دایرکتوری باید دقیقا با اسم بسته مطابقت داشته باشد .

این امکان وجود دارد که بیش از يك فایل شامل يك دستور یکسان package باشند .دستور package خیلی ساده مشخص می کند که کلاسهای تعریف شده در يك فایل به کدام بسته تعلق دارند. این دستور سایر کلاسهای موجود در فایلها دیگر را از اینکه بخشی از همان بسته اول باشند ، منع نمی کند . بسیاری از بسته های دنیای واقعی روی تعداد زیادی از فایلها گسترده می شوند .

می توانید يك سلسله مراتب از بسته ها بسازید . برای انجام اینکار ، خیلی ساده اسم هر بسته را از اسم بالایی اش و با استفاده از يك نقطه جدا سازید . شکل عمومی يك دستور package چند سطحی بقرار زیر است :

```
package pkg1[.pkg2[.pkg3]];
```

سلسله مراتب بسته باید در فایل سیستم توسعه جاوا منعکس شود . بعنوان مثال يك بسته اعلان شده بعنوان

```
package java.awt.image;
```

باید در java/awt/image ، java\awt\image یا java:awt:image بترتیب روی فایل سیستم windows ، unix ، یا Macintosh ذخیره شود. اسم بسته خود را خیلی با دقت انتخاب نمایید. نمی توانید اسم يك بسته را تغییر دهید مگر اینکه اسم دایرکتوری که کلاسها در آن ذخیره شده اند را تغییر دهید .

## درك مفهوم CLASS PATH

قبل از اینکه مثالی برای استفاده از يك بسته معرفی کنیم ، لازم است بحث مختصری درباره متغیر محیطی CLASS PATH داشته باشیم . اگرچه بسته ها بسیاری از مشکلات از نقطه نظر کنترل



دسترسي و اختلاط فضاي نام را حل مي کنند ، اما هنگام کامپايل و اجرا نمودن آنها با مشکلات عجيب و غريبي مواجه مي شويد . زيرا مكان مشخص در نظر گرفته شده بعنوان ريشه سلسله مراتب بسته توسط کامپايلر جاوا و بوسيله CLASS PATH كنترل مي شود . تاكنون شما همه كلاسها را در يك بسته پيش فرض يكسان و بدون اسم ذخيره مي كرديد . انجام اينكار به شما اجازه مي داد تا بسادگي كد منبع را كامپايل نموده و مفسر جاوا را روي نتيجه ، با اسم بردن از كلاس روي خط فرمان ، اجرا نماييد . اين مراحل بخوبي كار مي كرد زيرا مسير جاري در حال كار و پيش فرض (۰) معمولا در متغير محلي CLASS PATH قرار دارد كه البته بصورت پيش فرض براي سيستم حين اجراي جاوا تعريف شده است . اما وقتي بسته ها مخلوط شوند كار بهمين راحتی نخواهد بود .

فرض كنيد كه يك كلاس با نام packTest در يك بسته با نام test ايجاد کرده ايد . چون ساختار دايركتوري شما بايد با بسته هاي شما مطابقت داشته باشد يك دايركتوري تحت عنوان test ايجاد نموده و PackTest.java را در آن قرار مي دهيد . بعدا ميتوانيد test را دايركتوري جاري قرار دهيد و PackTest.java را كامپايل نماييد . اين امر سبب مي شود كه PackTest.class در دايركتوري test ذخيره شود . وقتي تلاش مي كنيد تا packTest را اجرا كنيد ، مفسر جاوا يك پيام خطا مشابه "can't find class packTest" گزارش مي كند . دليل آن است كه اكنون كلاس فوق را يك بسته با نام test ذخيره شده است . ديگر نمي توانيد بسادگي بعنوان packTest به آن ارجاع نماييد . بايد به آن كلاس با احتساب سلسله مراتب بسته آن ارجاع نماييد و هر بسته را با يك نقطه از ديگري جدا كنيد . اين كلاس اكنون بايد با نام test.packTest خوانده مي شود . اما اگر تلاش كنيد تا از test.packTest استفاده كنيد ، همچنان يك پيام خطا مشابه "can't find class test/ppackTest" دريافت خواهد نمود . دليل اينكه همچنان يك پيام خطا دريافت مي كنيد در متغير CLASS PATH شما نهفته است . بياد آوريد كه CLASS PATH در بالاي سلسله مراتب كلاس قرار مي گيرد . چيزي شبیه زیر را دربرمي گیرد :

```
. ;C:\java classes
```

كه به سيستم حين اجراي جاوا مي گويد تا دايركتوري در حال كار جاري را كنترل نموده و همچنين دايركتوري نصب standard java developers kit را كنترل نمايد . مشكل اين است

که دایرکتوری test در دایرکتوری در حال کار جاری وجود ندارد زیرا در خود دایرکتوری test قرار دارید. در این نقطه دو انتخاب خواهید داشت :

دایرکتوریهای یک سطح بالاتر را تغییر داده و javatest.packTest را آزمایش کنید و یا اینکه بالایی سلسله مراتب توسعه کلاس را به متغیر محیطی CLASS PATH اضافه نمایید . آنگاه می توانید javatest.packTest را از هر دایرکتوری مورد استفاده قرار دهید و جاوا فایل درست class را پیدا خواهد نمود. بعنوان مثال ، اگر روی کد منبع خودتان در یک دایرکتوری تحت نام C:\mmyjava در حال کار باشید ، آنگاه CLASS PATH خود را روی C:\myjava; . C:\java\classes قرار دهید.

### یک مثال کوتاه از بسته

اگر بحث قبلی را در نظر داشته باشید ، می توانید این بسته ساده را آزمایش کنید :

```
// A simple package
package MyPack;

class Balance {
    String name;
    double bal;
    Balance(String n/ double b ){
        name = n;
        bal = b;
    }
    void show (){
        if(bal<0)
            System.out.print("-->> ");
        System.out.println(name + " :$" + bal);
    }
}

class AccountBalance {
    public static void main(String args[] ){
        Balance current[] = new Balance[3];
        current[0] = new Balance("K .J .Fielding"/ 123.23);
        current[1] = new Balance("Will Tell"/ 157.02);
```

```
current[2] = new Balance("Tom Jackson"/- 12.33);
for(int i=0; i<3; i++ )current[i].show ();
}
}
```

این فایل را Mypack/java نامیده و آن را در يك دایرکتوري تحت نام Mypack قرار دهید . سپس فایل را کامپایل کنید . مطمئن شوید که فایل حاصل class. نیز در دایرکتوري Mypack قرار گرفته باشد . سپس اجراي کلاس Account Balance را آزمایش کنید، البته با استفاده از خط فرمان بعدي

```
java MyPack.AccountBalance
```

بیاد داشته باشید که وقتی مي خواهید این فرمان را اجرا کنید، لازم است تا در دایرکتوري بالاي Mypack باشید و یا اینکه متغیر محیطي CLASS PATH خود را بطور مناسبی قرار داده باشید .

همانطوریکه توضیح دادیم ، Account Balance اکنون بخشی از بسته Mypack است . این بدان معنی است که نمی تواند بوسیله خودش اجرا شود . یعنی نمی توانید از این خط فرمان استفاده نمایید .

```
Java AccountBalance
```

## محافظت دسترسی Access protection

قبلا مي دانستید که دسترسی به يك عضو private در يك کلاس فقط به سایر اعضای همان کلاس واگذار شده است . بسته ها بعد دیگری به کنترل دسترسی مي افزایند . همانطوریکه خواهید دید ، جاوا سطوح چندي از محافظت برای اجازه کنترل خوب طبقه بندی شده روی رویت پذیری متغیرها و روشهای داخل کلاسها ، زیر کلاسها و بسته ها فراهم مي نماید .

کلاسها و بسته ها هر دو وسایلي برای کپسول سازی بوده و دربرگیرنده فضاي نام و قلمرو متغیرها و روشها مي باشند . بسته ها بعنوان ظروفی برای کلاسها و سایر بسته های تابعه هستند . کلاسها بعنوان ظروفی برای داده ها و کدها مي باشند . کلاس کوچکترین واحد مجرد در جاوا

است. بلحاظ نقش متقابل بین کلاسها و بسته ها ، جاوا چهار طبقه بندی برای رویت پذیری اعضای کلاس مشخص کرده است:

زیر کلاسها در همان بسته

غیر زیر کلاسها در همان بسته

زیر کلاسها در بسته های مختلف

کلاسهایی که نه در همان بسته و نه در زیر کلاسها هستند.

سه مشخصگر دسترسی یعنی private ، public و protected و فراهم کننده طیف گوناگونی از شیوه های تولید سطوح چند گانه دسترسی مورد نیاز این طبقه بندیها هستند. جدول زیر این ارتباطات را یکجا نشان داده است.

public	protected	Nomodifier	private	
Yes	Yes	Yes	Yes	همان کلاس
Yes	Yes	Yes	No	زیر کلاس همان بسته
Yes	Yes	Yes	No	غیر زیر کلاس همان بسته
Yes	Yes	No	No	زیر کلاس بسته های مختلف
Yes	No	No	No	غیر زیر کلاس بسته های مختلف

اگرچه مکانیسم کنترل دسترسی در جاوا ممکن است بنظر پیچیده باشد، اما میتوان آن را بصورت بعدی ساده گویی نمود . هر چیزی که بعنوان public اعلان شود از هر جایی قابل دسترسی است. هر چیزی که بعنوان private اعلان شود خارج از کلاس خودش قابل رویت نیست. وقتی يك عضو فاقد مشخصات دسترسی صریح و روشن باشد ، آن عضو برای زیر کلاسها و سایر کلاسهای موجود در همان بسته قابل رویت است . این دسترسی پیش فرض است . اگر می خواهید يك عضو ، خارج از بسته جاری و فقط به کلاسهایی که مستقیماً از کلاس شما بصورت زیر کلاس در آمده اند قابل رویت باشد ، پس آن عضو را بعنوان protected اعلان نمایید.

يك کلاس فقط دو سطح دسترسی ممکن دارد : پیش فرض و عمومی . (public) وقتی يك کلاس بعنوان public اعلان می شود ، توسط هر کد دیگری قابل دسترسی است. اگر يك کلاس دسترسی پیش فرض داشته باشد ، فقط توسط سایر کدهای داخل همان بسته قابل دسترسی خواهد بود.

## يك مثال از دسترسي :

مثال بعدي كليہ تركيبات مربوط به اصلاحگرهاي كنترل دسترسي را نشان مي دهد . اين مثال داراي دو بسته و پنج كلاس است. بياد داشته باشيد كه كلاسهاي مربوط به دو بسته متفاوت ، لازم است در دايركتوريهايي كه بعداز بسته مربوطه اشان نام برده شده در اين مثال p1 و p2 و ذخيره مي شوند.

منبع اوليه بسته سه كلاس تعريف مي كند : protection ، Derived و samepackage .  
اولين كلاس چهار متغير int را در هر يك از حالات مختلف مجاز تعريف مي كند. متغير n با حفاظت پيش فرض اعلان شده است. m-pri بعنوان private ، n-pro ، بعنوان protected و n-pub بعنوان public مي باشند.

هر كلاس بعدي در اين مثال سعي مي كند به متغيرهايي در يك نمونه از يك كلاس دسترسي پيدا كند. خطوطي كه به لحاظ محدوديتهاي دسترسي ، كامپايل نمي شوند با استفاده از توضيح يك خطي // از توضيح خارج شده اند . قبل از هر يك از اين خطوط توضيحي قرار دارد كه مكانهايي را كه از آنجا اين سطح از حفاظت اجازه دسترسي مي يابد را فهرست مي نمايد. دومين كلاس Derived يك زير كلاس از protection در همان بسته p1 است. اين مثال دسترسي Derived را به متغيري در protection برقرار مي سازد بجز n-pri كه يك private است.  
سومين كلاس Samepackage يك زير كلاس از protection نيست ، اما در همان بسته قرار دارد و بنابر اين به كليہ متغيرها بجز n-pri دسترسي خواهد داشت .

```
package p1;
public class Protection {
    int n = 1;
    private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;
    public Protection () {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

```
}  
}
```

فایل Derived.java :

```
package p1;  
class Derived extends Protection {  
    Derived () {  
        System.out.println("derived constructor");  
        System.out.println("n = " + n);  
        // class only  
        // System.out.println("n_pri = " + n_pri);  
        System.out.println("n_pro = " + n_pro);  
        System.out.println("n_pub = " + n_pub);  
    }  
}
```

فایل Samepackage.java :

```
package p1;  
class SamePackage {  
    SamePackage () {  
        Protection p = new Protection();  
        System.out.println("same package constructor");  
        System.out.println("n = " + p.n);  
        // class only  
        // System.out.println("n_pri = " + p.n_pri);  
        System.out.println("n_pro = " + p.n_pro);  
        System.out.println("n_pub = " + p.n_pub);  
    }  
}
```

اکنون کد منبع یک بسته دیگر یعنی p2 را مشاهده می کنید. دو کلاس تعریف شده در p2 دو شرایطی را که توسط کنترل دسترسی تحت تاثیر قرار گرفته اند را پوشش داده است. اولین کلاس یعنی 2 protection یک زیر کلاس p1.protection است. این کلاس دسترسی به کلیه متغیرهای مربوط به p1.protection را بدست می آورد غیر از n-pri ( چون private است ) و n\_متغیری که با محافظت پیش فرض اعلان شده است. بیاد داشته باشید که پیش فرض فقط اجازه دسترسی از داخل کلاس یا بسته را می دهد نه از زیر کلاس های بسته های اضافی. در نهایت،

کلاس otherpackage فقط به يك متغير n-pub که بعنوان public اعلان شده بود دسترسی خواهد داشت .

فایل Protection2.java :

```
package p2;
class Protection2 extends p1.Protection {
    Protection2 () {
        System.out.println("derived other package constructor");
        // class or package only
        System.out.println("n = " + n);
        // class only
        // System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

فایل OtherPackage.java :

```
package p2;
class OtherPackage {
    OtherPackage () {
        p1.Protection p = new p1.protection();
        System.out.println("other package contryctor");
        // class or package only
        System.out.println("n = " + p.n);
        // class only
        // System.out.println("n_pri = " + p.n_pri);
        // class/ subclass or package only
        // System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

اگر مایلید تا این دو بسته را آزمایش کنید ، در اینجا دو فایل آزمایشی وجود دارد که می توانید از آنها استفاده نمایید. یکی از این فایلها برای بسته p1 را در زیر نشان داده ایم :

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo {
public static void main(String args[] ){
Protection ob1 = new Protection();
Derived ob2 = new Derived();
SamePackage ob3 = new SamePackage();
}
}
```

فایل آزمایشی برای p2 بقرار زیر می باشد :

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
public class Demo {
public static void main(String args[] ){
Protection2 ob1 = new
```

## وارد کردن بسته ها

با قبول این امر که بسته ها وجود داشته و مکانیسم خوبی برای جدا کردن کلاسهای متمایز از یکدیگر هستند، آنگاه براحتی می فهمید که چرا کلیه کلاسهای توکار جاوا در بسته ها ذخیره می شوند. در بسته پیش فرض بدون نام ، کلاسهای بسته ای جاوا وجود ندارند . کلیه کلاسهای استاندارد در برخی بسته های دارای نام ذخیره می شوند . چون کلاسهای داخل بسته ها باید کاملاً با نام یا اسمی بسته های خود کیفیت دهی شوند ، ممکن است تایپ نام مسیر بسته که با نقطه از هم جدا می شود برای هر یک کلاسی که مایلید استفاده نمایید ، کاری کسل کننده باشد . بهمین دلیل ، جاوا دربرگیرنده دستور import ( وارد کردن ) برای آوردن کلاسهای مشخص یا کل بسته ها در معرض رویت می باشد. هر کلاسی که یکبار وارد شود ، فقط با استفاده از اسم آن بطور مستقیم قابل ارجاع است. دستور import یک وسیله راحت برای برنامه نویسان است و از نظر تکنیکی دیگر نیازی نیست تا یک برنامه کامل جاوا نوشته شود . اگر قصد دارید تا به چند ده کلاس در برنامه اتان ارجاع نمایید ، دستور import حجم زیادی از کار تایپی را کاهش می دهد.



در يك فايل منبع جاوا ، دستورات import بلافاصله بعد از دستور package ( اگر وجود داشته باشد ) و قبل از تعريف مربوط به هر كلاس قرار مي گيرند. شكل عمومي دستور import بقرار زير است :

```
import pkg1[pkg2](.classname |*);
```

در اينجا pkg1 نام يك بسته سطح بالاست و pkg2 نام يك بسته تابعه داخل بسته خارجي است كه با يك نقطه جدا شده است . هيچ محدوديت عملي در خصوص عمق سلسله مراتب بسته وجود ندارد ، بجز مواردی كه توسط فايل سيستم اعمال مي شود . در نهايت ، يا يك classname صريح را مشخص مي كنيد و يا از يك ستاره (\*) استفاده مي كنيد كه نشان مي دهد كه كامپايلر جاوا بايد كل بسته را وارد نمايد . اين قطعه كد هر دو شكل مذكور را نشان مي دهد :

```
import java.util.Date;
import java.io.*;
```

احتياط : استفاده از ستاره ممكن است زمان كامپايل را افزايش دهد بخصوص كه اگر چندين بسته بزرگ را وارد نماييد . بهمين دليل بهتر است بطور صريح نام كلاسهايي كه مي خواهيد استفاده نماييد بنويسيد ، اما استفاده از ستاره هيچ تاثيري روي عملکرد حين اجرا يا اندازه كلاسهاي شما نخواهد داشت . كلييه كلاسهاي استاندارد جاوا داخل در جاوا در يك بسته تحت نام java ذخيره مي شوند. توابع اصلي زبان در يك بسته java تحت نام java.lang ذخيره مي شوند . بطور معمول ، شما هر بسته يا كلاسي را كه نياز داشته باشيد وارد خواهيد كرد اما چون جاوا بدون توابع موجود در java.lang بلا استفاده خواهد بود ، لذا اين بسته بطور ضمني توسط خود كامپايلر براي كلييه برنامه ها وارد خواهد شد. اين حالت معادل آن است كه خط بعدي در بالاي كلييه برنامه هاي شما قرار بگيرد :

```
import java.lang.*;
```

اگر يك كلاس با همان اسم در دو بسته متفاوتي كه با استفاده از ستاره وارد کرده ايد ، قرار داشته باشد ، كامپايلر ساكت (silent) مي ماند ، مگر اينكه سعي كنيد يكي از آن كلاسها را استفاده نماييد. در آن حالت ، با يك خطاي compile-time مواجه شده و مجبور مي شويد تا بطور صريح نام كلاس همراه با مشخصات بسته آن را ذكر نماييد. هر جايي از يك نام كلاس استفاده مي كنيد ،

مي توانيد از نام با كيفيت كامل استفاده نماييد. كه دربرگيرنده سلسله مراتب كامل بسته آن مي باشد.  
بعنوان مثال اين قطعه ، از يك دستور import استفاده مي كند:

```
import java.util.*;
class MyDate extends Date {
}
```

همان مثال بدون دستور import بقرار زير خواهد بود :

```
class MyDate extends java.util.Date {
}
```

وقتي يك بسته وارد ميشود، فقط آن اقليمي در داخل بسته كه بعنوان public اعلان شده اند در دسترس غير زير كلاسهاي موجود در كد وارد شده مي باشند . بعنوان مثال اگر مي خواهيد كلاس Balance از بسته Mypack كه قبلا" نشان داده ايم ، بعنوان يك كلاس مستقل براي کاربردهاي عمومي خارج از Mypack در دسترس باشد ، پس بايد آن را بعنوان public اعلان نموده و آن را در فايل خودش ، بصورت زير قرار دهيد :

```
package MyPack;
/* Now/ the Balance class/ its constructor/ and its
show ) (method are public .This means that they can
be used by non-subclass code outside their package.
*/
public class Balance {
String name;
double bal;
public Balance(String n/ double b ){
name = n;
bal = b;
}
public void show (){
if(bal<0)
System.out.print("-->> ");
System.out.println(name + " :$" + bal);
}
}
```

همانطوریکه می بینید ، کلاس Balance اکنون بعنوان public می باشد. همچنین سازنده این کلاس و روش show() در آن نیز بعنوان public می باشند . این بدان معنی است که همه آنها توسط هر نوع کدی خارج از بسته Mypack قابل دسترسی هستند . بعنوان مثال ، در اینجا Test Balance وارد کننده Mypack شده و سپس قادر است از کلاس Balance استفاده نماید :

```
import MyPack.*;
class TestBalance {
public static void main(String args[] ){
/* Because Balance is public/ you may use Balance
class and call its constructor .*/
Balance test = new Balance("J .J .Jaspers"/ 99.88);
test.show(); // you may also call show()
}
}
```

بعنوان يك تجربه ، مشخصگر public را از کلاس Balance برداشته و سپس سعی کنید ا بروز می کنند.

## رابطها Interfaces

بااستفاده از واژه کلیدی interface ، می توانید رابط يك کلاس را از پیاده سازی آن کلاس بطور کامل مجرد نمایید. یعنی با استفاده از interface می توانید مشخص نمایید يك کلاس چکاري باید انجام دهد ، اما چگونگی آنرا مشخص نخواهید کرد. رابطها از نظر قواعد صرف و نحو مشابه کلاسها هستند ، اما فاقد متغیرهای نمونه هستند و روشهای آنها بدون بدنه اعلان می شود . در عمل ، این بدان معنی است که می توانید رابطهایی تعریف کنید که درباره چگونگی پیاده سازی خود فرضیه ای نمی سازند. هر بار که رابط تعریف شود ، هر تعدادی از کلاسها می توانند آن رابط interface را پیاده سازی نمایند. همچنین ، يك کلاس می تواند هر تعداد رابطها را پیاده سازی نماید.

برای پیاده سازی يك رابط، کلاس باید مجموعه کامل روشهای تعریف شده توسط رابط را ایجاد نماید. اما ، هر کلاسی آزاد است تا جزئیات پیاده سازی خودش را تعیین نماید. با استفاده از واژه

کلیدی interface ، جاوا به شما امکان می دهد تا حداکثر بهره از جنبه " یک رابط و چندین روش " در چند شکلی را بدست آورید . رابطها بمنظور حمایت از سرگیری پویای روش در حین اجرا طراحی می شوند. بطور معمول ، برای اینکه یک روش از یک کلاس به کلاس دیگر فراخوانی شود ، هر دو کلاس باید در زمان کامپایل حضور داشته باشند، بطوریکه کامپایلر جاوا بتواند آنها را کنترل نموده و سازگاری روشها را تایید نماید. این امر بتنهایی منجر به یک محط کلاس دهی ایستا و غیر قابل توسعه خواهد شد. در یک چنین سیستمی بناچار عملگرایی (functionality) در سلسله مراتب کلاس بالاتر و بالاتر می رود بطوریکه مکانیسم فوق ، در دسترس زیر کلاسهای بیشتری قرار خواهد گرفت. رابطها طراحی میشوند تا از بروز چنین مشکلی جلوگیری بعمل آورند. آنها قطع ارتباط بین تعریف یک روش یا مجموعه ای از روشها و سلسله مراتب وراثت را بوجود می آورند. از آنجاییکه رابطها در سلسله مراتب متفاوتی از کلاسها قرار گرفته اند، برای کلاسهایی که بر حسب سلسله مراتب کلاس نامرتب هستند ، امکان پیاده سازی همان رابط وجود دارد. این نقطه ای است که قدرت واقعی رابطها را نمایان می سازد .

نکته :

رابطها ، آن میزان عملگرایی لازم برای بسیاری از کاربردها را بوجود می آورند که در غیر اینصورت ناچار از متوسل شدن به وراثت چند گانه در زبانی نظیر ++C خواهیم شد.

### تعریف نمودن یک رابط

یک رابط مشابه یک کلاس تعریف می شود . شکل عمومی یک رابط بصورت زیر می باشد :

```
access interface name {  
    return-type method-name1(parameter-list);  
    return-type method-name2(parameter-list);  
    type final-varname1 = value;  
    type final-varname2 = value;  
    //...  
    return-type method-nameN(parameter-list);  
    type final-varnameN = value;  
}
```

در اینجا ، دسترسی (access) یا public است یا اصلا بکار نمی رود. وقتی که مشخصگر دسترسی را نگنجانیم ، آنگاه دسترسی پیش فرض اعمال شده و رابط فقط برای سایر اعضا بسته ای که در آن اعلان انجام گرفته ، در دسترس خواهد بود. وقتی که بعنوان public اعلان می شود ، رابط ، توسط هر کد دیگری قابل استفاده خواهد بود. همان نام رابط است و بجای آن هر شناسه معتبری را می توان بکار برد . توجه کنید که روشهای اعلان شده دارای بدنه نیستند. انتهای آنها بعد از parameter-list يك (;) قرار میگیرد و آنها ضرورتاً روشهای مجرد هستند: ممکن است هیچ پیاده سازی پیش فرضی برای روشهای مشخص شده داخل يك رابط وجود نداشته باشد . هر کلاسی که دربرگیرنده يك رابط باشد ، باید کلیه روشها را پیاده سازی نماید . متغیرها را می توان داخل اعلانات رابط ، اعلان نمود. آنها بطور ضمنی final و static هستند. بدین معنی که آنها توسط کلاس پیاده سازی قابل تغییر نیستند . آنها همچنین باید با يك مقدار ثابت ، مقدار دهی اولیه شوند . اگر رابط خودش بعنوان يك public اعلان شده باشد ، کلیه روشها و متغیرها بطور ضمنی public خواهند بود.

اکنون مثالی از يك تعریف رابط را مشاهده می کنید. این مثال يك رابط ساده را تعریف می کند که دربرگیرنده روشی تحت نام callback() است که يك پارامتر تکی عدد صحیح را می گیرد .

```
interface Callback {  
void callback(int param);  
}
```

### پیاده سازی رابطها

هر بار رابطی را تعریف نمایید ، يك یا چندین کلاس می توانند آن رابط را پیاده سازی نمایند . برای پیاده سازی يك رابط ، جمله implements را در تعریف کلاس بگنجانید و سپس روشهای تعریف شده توسط رابط را ایجاد نمایید. شکل عمومی يك کلاس که دربرگیرنده جمله implements است ، مشابه مورد زیر می باشد :

```
access class classname [extends superclass]  
[implements interface [/interface...]] {
```

```
// class-body  
}
```

در اینجا (access) یا public است یا اصلاً استفاده نمی شود. اگر یک کلاس بیش از یک رابط را پیاده سازی نماید، رابطها را با یک علامت کاما از یکدیگر جدا می کنیم. اگر یک کلاس دو رابط را که روش یکسانی را اعلان می کنند، پیاده سازی نماید، آنگاه آن روش یکسان توسط سرویس گیرندگان هر یک از رابطها استفاده خواهد شد. روشهایی که یک رابط را پیاده سازی می کنند باید بعنوان public اعلان شوند. همچنین تایید نوع روش پیاده سازی باید دقیقاً با تایید نوع مشخص شده در تعریف interface مطابقت و سازگاری داشته باشد. در اینجا یک مثال از کلاس ساده ای مشاهده می کنید که رابط callback را پیاده سازی می نماید:

```
class Client implements Callback {  
    // Implement Callback's interface  
    public void callback(int p ){  
        System.out.println("callback called with " + p);  
    }  
}
```

دقت داشته باشید که callback() با استفاده از مشخصگر دسترسی public اعلان شده است. یادآوری: وقتی که یک روش رابط را پیاده سازی می کنید، بعنوان public اعلان خواهد شد. برای کلاسهایی که رابطها را پیاده سازی می کنند، هم مجاز و هم رایج است که اعضا اضافی برای خودشان تعریف نمایند. بعنوان مثال، روایت بعدی از client پیاده سازی callback را انجام داده و روش nonIfaceMeth() را اضافه می نماید:

```
class Client implements Callback {  
    // Implement Callback's interface  
    public void callback(int p ){  
        System.out.println("callback called with " + p);  
    }  
    void nonIfaceMeth () {  
        System.out.println("Classes that implement interfaces " +  
            "may also define other members/ too.");  
    }  
}
```

}

## دسترسي به پياده سازيها از طريق ارجاعات رابط

مي توانيد متغيرهايي را بعنوان ارجاعات شي اعلان كنيد كه بجاي يك نوع كلاس از يك رابط استفاده نمايند. هر نمونه اي از كلاسي كه رابط اعلان شده را پياده سازي مي كند را مي توان در چنان متغيري ذخيره نمود. وقتي يك روش را از طريق يكي از اين ارجاعات فراخواني مي كنيد ، روايت صحيح براساس نمونه واقعي رابطي كه به آن ارجاع شده ، فراخواني خواهد شد. اين يكي از جنبه هاي كليدي رابطهاست. روشي كه بايد اجرا شود در حين اجرا بصورت پويا مورد جستجو قرار مي گيرد و به كلاسها اجازه مي دهد تا ديرتر از كدي كه روشها را روي آن فراخواني ميكنند، ايجاد شوند. كد فراخواننده مي تواند از طريق يك رابط بدون اينكه نيازي به دانستن درباره "callee" داشته باشد، توزيع نمايد. اين پردازش مشابه استفاده از يك ارجاع كلاس بالا براي دسترسي به يك شي زير كلاس است.

احتياط : چون جستجوي پويا بدنبال يك روش در حين اجرا، در مقايسه با احضار روش معمول در جاوا ، متحمل بالاسري (over head) بزرگي مي شود ، بايد مراقب باشيد تا از رابطها بطور اتفاقي در كدهاي performance-critical استفاده نكنيد . مثال بعدي روش callback () را از طريق يك متغير ارجاع رابط، فراخواني مي كند :

```
class TestIface {
public static void main(String args[] ){
    Callback c = new Client();
    c.callback(42);
}
}
```

حاصل اين برنامه بقرار زير مي باشد :

```
callback called with 42
```

دقت كنيد كه متغير C طوري تعريف شده كه از نوع رابط callback باشد. با اين وجود يك نمونه از client به آن منتسب شد. اگرچه مي توان از C براي دسترسي به روش callback() استفاده

نمود ، اما نمي تواند به هيچيك از اعضاي كلاس client دسترسي داشته باشد . يك متغير ارجاع رابط فقط از روشهاي اعلان شده توسط اعلان interface خود ، آگاهي دارد . بدین ترتيب ، نمي توان از C براي دسترسي به nonIfaceMeth() استفاده نمود چون توسط client و نه توسط callback تعريف شده است.

درحاليكه مثال قبلي نشان ميدهد كه يك متغير ارجاع رابط چگونه ، بطور مكانيكي مي تواند به يك پياده سازي شي ء دسترسي داشته باشد ، اما قدرت چند شكلي يك چنين ارجاعي را نمايش نمي دهد . براي مشاهده اين کاربرد ، ابتدا دومين پياده سازي Callback را بصورت زير ، ايجاد نماييد :

```
// Another implementation of Callback.
class AnotherClient implements Callback {
// Implement Callback's interface
public void callback(int p ){
System.out.println("Another version of callback");
System.out.println("p squared is " + ( p*p));
}
}
```

اکنون کلاس بعدي را امتحان کنید :

```
class TestIface2 {
public static void main(String args[] ){
Callback c = new Client();
AnotherClient ob = new AnotherClient();
c.callback(42);
c = ob; // c now refers to AnotherClient object
c.callback(42);
}
}
```

حاصل این برنامه بقرار زیر می باشد :

```
callback called with 42
Another version of callback
p squared is 1764
```



همانطوریکه می بینید ، روایتی از `callback()` که فراخوانی شده توسط نوع شیئی که `C` به آن در حین اجرا ارجاع می کند ، تعیین می شود . اگرچه این مثال بسیار ساده است ، یک برنامه عملی تر و کوتاهتر را مشاهده خواهید نمود.

## پیاده سازی نسبی (partial)

اگر یک کلاس دربرگیرنده یک رابط باشد ، اما روش تعریف شده توسط آن رابط را کاملاً پیاده سازی نکند ، آنگاه آن کلاس باید بعنوان `abstract` اعلان شود . بعنوان مثال :

```
abstract class Incomplete implements Callback {
    int a, b;
    void show () {
        System.out.println(a + " " + b);
    }
    //...
}
```

در اینجا کلاس `Incomplete` روش `callback()` را پیاده سازی نمیکند و باید بعنوان `abstract` اعلان شود. هر کلاسی که از `Incomplete` ارث میبرد باید یا روش `callback()` را پیاده سازی نماید و یا خودش بعنوان `abstract` اعلان شود .

## بکار بردن رابطها

برای درک قدرت رابطها ، اجازه دهید تا به یک مثال عملی تر بپردازیم . قبلاً یک کلاس موسوم به `stack` را توسعه دادیم که یک پشته ساده و با اندازه ثابت را پیاده سازی می کرد . اما ، راههای متعددی برای پیاده سازی یک پشته وجود دارد . بعنوان مثال ، یک پشته ممکن است دارای اندازه ثابت و یا قابل گسترش باشد .

همچنین می توان پشته را در یک آرایه ، یک فهرست پیوندی (`kinded list`) ، یک درخت دودویی و امثالهم نگهداری نمود . مهم نیست که پشته چگونه پیاده سازی می شود رابط به پشته یکسان می ماند. یعنی روشهای `push()` و `pop()` معرف رابط به پشته هستند و این رابطها

مستقل از جزئیات پیاده سازی می باشند. چون رابط به یک پشته از پیاده سازی آن جدا می باشد ، تعریف رابط یک پشته خیلی ساده است ، و هر پیاده سازی مشخصات خاص خود را تعریف خواهد کرد . به دو مثال نگاه کنید . اول اینکه در اینجا رابطی وجود دارد که یک پشته عدد صحیح را تعریف می کند . آن را در یک فایل تحت نام IntStack.java قرار دهید . این رابط توسط هر دو نوع پیاده سازی پشته استفاده خواهد شد .

```
// Define an integer stack interface.
interface IntStack {
void push(int item); // store an item
int pop(); // retrieve an item
```

برنامه بعدی یک کلاس تحت نام FixedStack ایجاد می کند:

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
private int stck[];
private int tos;
// allocate and initialize stack
FixedStack(int size ){
stck = new int[size];
tos = - 1;
}
// Push an item onto the stack
public void push(int item ){
if(tos==stck.length-1 )// use lenggth member
System.out.println("Stack is full.");
else
stck[++tos] = item;
}
// Pop an item from the stack
public int pop (){
if(tos < 0 ){
System.out.println("Stack underflow.");
return 0;
}
else
```

```

return stck[tos--];
}
}
class IFTest {
public static void main(String args[] ){
FixedStack mystack1 = new FixedStack(5);
FixedStack mystack2 = new FixedStack(8);
// push some numbers onto the stack
for(int i=0; i<5; i++ )mystack1.push(i);
for(int i=0; i<8; i++ )mystack2.push(i);
// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<5; i++)
System.out.println(mystack1.pop() );
System.out.println("Stack in mystack2:");
for(int i=0; i<8; i++)
System.out.println(mystack2.pop() );
}
}

```

در زیر يك پياده سازي ديگر از IntStack وجود دارد كه پشته پويا با استفاده از همان تعريف Interface ايجاد مي كند. در اين پياده سازي ، هر پشته با يك طول اوليه ساخته ۲ مي شود. اگر اين طول اوليه تجاوز شود، آنگاه پشته از نظر اندازه افزايش مي يابد . هر بار اطاق بعدي نياز باشد ، اندازه پشته دو برابر خواهد شد .

```

// Implement a "growable" stack.
class DynStack implements IntStack {
private int stck[];
private int tos;
// allocate and initialize stack
DynStack(int size ){
stck = new int[size];
tos =- 1;
}
// Push an item onto the stack
public void push(int item ){
// if stack is full/ allocate a larger stack

```

```

if(tos==stck.length-1 ){
int temp[] = new int(stck.length * 2); // duble size
for(int i=0; i + stck = temp;
stck[++tos] = item;
}
else
stck[++tos] = item;
}
// Pop an item from the stack
public int pop (){
if(tos < 0 ){
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}
class IFTest2 {
public static void main(String args[] ){
DynStack mystack1 = new FixedStack(5);
DynStack mystack2 = new FixedStack(8);
// these loops cause each stack to grow
for(int i=0; i<12; i++ )mystack1.push(i);
for(int i=0; i<20; i++ )mystack2.push(i);
System.out.println("Stack in mystack1:");
for(int i=0; i<12; i++)
System.out.println(mystack1.pop() );
System.out.println("Stack in mystack2:");
for(int i=0; i<20; i++)
System.out.println(mystack2.pop() );
}
}

```

کلاس بعدی هم از پیاده سازی FixedStack و هم از DynStack استفاده می کند . این کلاس اینکار را طریق تعریف یک ارجاع رابط انجام می دهد. این بدان معنی است که فراخوانی های push() و pop() در حین اجرا ( بجای زمان کامپایل ) از سرگرفته خواهد شد.

```

/* Create an interface variable and
access stacks through it.
*/
class IFTest3 {
public static void main(String args[] ){
    IntStack mystack; // create an interface reference variable
    DynStack ds = new DynStack(5);
    FixedStack fs = new FixedStack(8);
    mystack = ds; // load dynamic stack
    // push some numbers onto the stack
    for(int i=0; i<12; i++ )mystack.push(i);
    mystack = fs; // load fixed stack
    for(int i=0; i<8; i++ )mystack.push(i);
    mystack = ds;
    System.out.println("Values in dynamic stack:");
    for(int i=0; i<12; i++)
    System.out.println(mystack.pop() );
    mystack = fs;
    System.out.println("Values in fixed stack:");
    for(int i=0; i<8; i++)
    System.out.println(mystack.pop() );
}
}

```

در این برنامه ، `mystack` یک ارجاع است به رابط `IntStack`. بدین ترتیب ، هرگاه آن به `ds` ارجاع میکند، از روایتهای `push()` و `pop()` تعریف شده بوسیله پیاده سازی `DynStack` استفاده می کند . وقتی که به `fs` ارجاع می کند ، از روایتهای `push()` و `pop()` که بوسیله `FixedStack` تعریف شده ، استفاده می کند. همانطوریکه توضیح دادیم ، تمام اسن تعیین کنندگیها در حین اجرا انجام می گیرند. دسترسی به پیاده سازیهای چندگانه از یک رابط از طریق یک متغیر ارجاع رابط یکی از شیوه های کاملاً قدرتمند جاوا برای رسیدن به چند شکلی در حین اجرا می باشد .

### متغیرها در رابطها

می توانید از رابطها برای وارد کردن ثابتهای اشتراک گذاشته شده به کلاسهای چند گانه بسادگی

از اعلان يك رابط كه دربرگیرنده متغیرهایی باشد كه با مقادیر دلخواه مقداردهی اولیه شده باشند، استفاده نمایید . وقتی كه آن رابط را در يك كلاس می گنجانید ( یعنی وقتی كه رابط را پیاده سازی می کنید ) کلیه اسمی آن متغیرها بعنوان ثابت ها در قلمرو خواهند بود . این کار مشابه استفاده از فایل feader در C++/C برای ایجاد يك رقم بزرگ از ثابتهای #defined و با اعلانات const می باشد. اگر يك رابط دربرگیرنده هیچ روشی نباشد، آنگاه هر کلاسی كه دربرگیرنده آن رابط باشد در واقع چیزی را پیاده سازی نمی کند . مثل این است كه آن كلاس متغیرهای ثابت را به فضایی اسم كلاس بعنوان متغیرهای final وارد می کرده است .

مثال بعدی از این تکنیک برای پیاده سازی يك تصمیم گیرنده خودکار (automated decision maker) استفاده نموده است .

```
import java.Random;
interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}
class Questionimplements SharedConstants {
    Random rand = new Random();
    int ask () {
        int prob = (int)(100 * rand.nextDouble());
        if( prob < 30)
            return NO; // 30%
        else if( prob < 60)
            return YES; // 30%
        else if( prob < 75)
            return LATER; // 15%
        else if( prob < 98)
            return SOON; // 13%
        else
            return NEVER; // 2%
    }
}
```

```

}
class AskMe implements SharedConstante {
static void answer(int result ){
switch(result ){
case NO:
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break;
case MAYBE:
System.out.println("Maybe");
break;
case LATER:
System.out.println("Later");
break;
case SOON:
System.out.println("Soon");
break;
case NEVER:
System.out.println("Never");
break;
}
}
public static void main(String args[] ){
Question q = new Question();
answer(q.ask) ();
answer(q.ask) ();
answer(q.ask) ();
answer(q.ask) ();
}
}

```

دقت داشته باشید که این برنامه از یکی از کلاسهای استاندارد جاوا یعنی Random استفاده میکند. این کلاس فراهم کننده اعداد شبه تصادفی است. این کلاس در برگیرنده چندین روش است که به شما امکان نگهداری ارقام تصادفی در شکل مورد نیاز برنامه آن را می دهد. در این مثال، از

روش `nextDouble()` استفاده شده است. این روش اعداد تصادفی در محدوده ۰,۰ تا ۱۱,۰ را برمی گرداند.

در این برنامه نمونه دو کلاس `Question` و `AskMe` و هر دو رابط `SharedConstants` را پیاده سازی می کنند، جایی که `No`، `Yes`، `MAYBE`، `SOON`، `LATER` و `NEVER` تعریف شده اند. داخل هر یک کلاس، کد به این ثابت ها مراجعه می کند بطوریکه گویا هر کلاس آنها را بطور مستقیم تعریف نموده یا مستقیماً از آنها ارث برده است. در اینجا حاصل یک اجرایی نمونه از این برنامه را مشاهده می کنید. دقت کنید که نتایج در هر بار اجرا متفاوت خواهد بود.

```
Later
Soon
No
Yes
```

### رابطها را می توان گسترش داد

یک رابط با استفاده از واژه کلیدی `extends` می تواند از یک رابط دیگر ارث ببرد. دستور زبان مشابه کلاسهای ارث برنده است. وقتی یک کلاس رابطی را پیاده سازی می کند که از رابط دیگری ارث برده است، باید پیاده سازیهای کلیه روشهای تعریف شده داخل زنجیره وراثت را فراهم نماید. مثالی را مشاهده می کنید:

```
// One interface can extend another.
interface A {
void meth1();
void meth2();
}
// B now includes meth1 (and meth2 -- ) (it adds meth3.) (
interface B extends A {
void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
public void meth1 () {
```



```

System.out.println("Implement meth1.")("");
}
public void meth2 () {
System.out.println("Implement meth2.")("");
}
public void meth3 () {
System.out.println("Implement meth3.")("");
}
}
}
class IFExtend {
public static void main(String args[] ) {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}

```

بعنوان يك تجربه ممكن است بخواهيد سعي كنيد تا پياده سازي meth1() را در كلاس Myclass جابجا نماييد. اين عمل سبب بروز خطاي comile-time خواهد شد. همانطوريكه قبل گفتيم، هر كلاسي كه يك رابط را پياده سازي مي كند بايد كليۀ روشهاي تعريف شده توسط آن رابط شامل هر كدام كه از ساير رابطها ارث برده اند را پياده سازي نمايد.

بسته ها و رابطها هر دو بخش مهمي از محيط برنامه نويسي جاوا هستند.

# انواع داده مرکب

## پیاده سازی انواع ساختمان داده ها

### عناوین این بخش :

رشته ها

آرایه ها

فایل ها

در این فصل با نحوه پیاده سازی انواع ساختمان داده آشنا خواهید شد.

## رشته ها

جاوا از رشته ها به عنوان یکی از انواع داده ها پشتیبانی نمیکند. داده های رشته ای جاوا که String نامیده می شوند ، یکی از انواع داده های پایه و ساده به شمار نمی آیند و همین طور آرایه ای از کاراکترها نیز به شمار نمی آیند. بلکه در عوض شیء میباشند .

از String برای تعریف کردن متغیر های رشته ای استفاده می شود. همچنین می توان آرایه ها رشته ای تعریف کرد. ثابت های رشته ای که بین علائم نقل قول نوشته میشوند را میتوان به متغیرهای نوع String تخصیص داد. متغیرهای نوع String را میتوان به سایر متغیر های نوع String تخصیص داد. به عنوان مثال

```
String str="this is a test"
```

```
System.out.println(str);
```

str در اینجا شیئی از نوع string است که رشته "this is a test" به آن تخصیص می یابد. این رشته به وسیله عبارت println() نمایش داده می شود. شیء های نوع string ویژگی ها و خصوصیات ویژه زیادی دارند که آنها را بسیار قدرتمند و آسان می سازند.

string متداول ترین کلاس در کتابخانه کلاس های جاوا به شمار می آید. دلیل بارز این مطلب آن است که رشته ها بخش بسیار مهمی از برنامه سازی به شمار می آید.

نخستین چیزی که باید درباره رشته ها بدانید , آن است که هر رشته ای که ایجاد می کنید در واقع شیئی از کلاس string است. حتی ثابت های رشته ای هم شیئی به شمار می آیند. به عنوان مثال در عبارت زیر

```
System.out.println("this is a string,too");
```

رشته "this is a string,too" نوعی ثابت رشته ای به شمار می آید. روش مدیریت ثابت های رشته ای در جاوا همچون مدیریت رشته های معمولی در زبان های کامپیوتری دیگر است.

دومین مطلبی که باید درباره رشته ها بدانید آن است که شیء های نوع string تغییرناپذیر هستند. یعنی پس از ایجاد شیء های string محتوای آن ها قابل تغییر نخواهد بود. اگرچه این موضوع ممکن است محدودیت جدی به نظر آید, اما به دو دلیل اینگونه نیست:

۱\_ اگر نیاز به تغییر رشته ای داشته باشید, همیشه می توانید نمونه جدیدی ایجاد کنید که متضمن تغییرات مورد نظر باشد.

۲\_ کلاسی نظیر string به نام StringBuffer در جاوا تعریف شده است که امکان تغییر رشته ها را فراهم می سازد، بنابر این تمام کارهای پردازش مربوط به رشته ها هنوز در جاوا قابل انجام هستند.

روش های گوناگونی برای ایجاد رشته ها وجود دارد. آسانترین روش استفاده از عبارتی چون مثال زیر است:

```
String mystring = "this is a test";
```

پس از ایجاد یک شیء فقهدهل آن را می توانید در هر شرایطی که کاربرد رشته ها مجاز است به کار برید .

به عنوان مثال عبارت زیر mystring را نمایش می دهد:

```
System.out.println(my string);
```

عملگر "+" در جاوا برای شیء های نوع String تعریف شده است. از آن برای ادغام دو رشته استفاده میشود. به عنوان مثال نتیجه عبارت زیر

```
String mystring = "I" + "like" + "java. ";
```

ذخیره شدن "I like java." در mystring می شود.

کلاس string چندین متد دارد. برخی از آنها در اینجا بررسی شده اند. با استفاده از equals() می توانید تساوی دو رشته را بررسی کنید. با فراخوانی متد length() می توانید طول یک رشته را به دست آورید. با استفاده از charAt() هم می توانید کاراکتر موجود در موقعیت مورد نظر در رشته را بدست آورید. شکل کلی این سه متد در ذیل نشان داده شده است:

```
Boolean equals(String object)
```

```
Int length()
```

```
Char charAt(int index)
```

البته از آرایه های رشته ای نیز می توانید همچون سایر انواع آرایه ها استفاده کنید. به عنوان مثال:

```
Class stringdemo3{
Public static void main(string args[]){
String str[] = {"one","two","three"};
For (int i=0;i<str.length;i++)
System.out.println("str["+i+"]:"+str[i]);
}
}
```

خروجی :

```
Str[0]:one  
Str[1]:two  
Str[2]:three
```

پیاده سازی رشته ها به صورت شیئی های توکار این امکان را برای جاوا فراهم ساخته است تا ویژگی های زیادی در اختیاران بگذارد که مدیریت رشته ها را آسان می سازند. به عنوان مثال جاوا متد هایی برای مقایسه دو رشته, جستجوی یک زیر رشته, ادغام دو رشته و تغییر بزرگی و کوچکی حروف هر رشته دارد. همچنین شیئی های String را با چند روش می توان ایجاد نمود. بنابر این ایجاد رشته ها به هنگام نیاز آسان می شود. وقتی یک شیئی نوع String ایجاد میکنیم, رشته ای ایجاد می شود که قابل تغییر نخواهد بود. یعنی پس از ایجاد شیء های string کاراکترهای تشکیل دهنده آنها رانمی توانید تغییر دهید. این امر در نگاه نخست ممکن است نوعی محدودیت به شمار آید اما واقعا این گونه نیست. باز هم میتوانید تمام انواع عملیات مربوط به رشته ها را انجام دهید. تفاوت کار در آن است که هر بار نیاز به نگارش تغییر یافته ای از هر رشته مطرح باشد شیئی String جدیدی ایجاد می شود که متضمن تغییرات خواهد بود. رشته اولیه تغییر نیافته باقی می ماند. دلیل استفاده از این رویه آن است که پیاده سازی رشته های ثابت و تغییر ناپذیر نسبت به رشته های قابل تغییر کارآمدتر خواهد بود. جاوا برای مواقعی که نیاز به رشته های تغییر ناپذیر باشد دو گزینه فراهم کرده است: `StringBuffer`, `StringBuilder` هر دو مورد می توانند برای نگهداری رشته هایی بکار روند که پس از ایجاد قابل تغییرند.

کلاس های `String`, `StringBuffer`, `StringBuilder` در `java.lang` تعریف شده اند. از این رو هر سه آن ها به طور خودکار در اختیار تمام برنامه ها قرار دارند هر سه به صورت `final` تعریف شده اند و این بدان معناست که هیچ کلاس دیگری را نمیتوان از آن ها مشتق نمود. این امر سبب پاره ای بهینه سازی شده است که موجب افزایش کارایی عملیات متداول مربوط شده اند تمام کلاس های مزبور رابط `CharSequence` را پیاده سازی میکنند.

گفتن اینکه رشته های موجود در شیئی های نوع `String` غیر قابل تغییر هستند, بدین معناست که محتوای نمونه های `String` را نمیتوان پس از ایجاد تغییر داد. اما تغییری که به صورت نشانی `String` ها تعریف می شود در هر لحظه می تواند به شیئی `String` دیگری ارجاع داشته باشد.

## String کلاس Constructor

کلاس String از چند constructor پشتیبانی می کند. برای آن که یک string خالی ایجاد کنید, constructor پیش فرض را فرا بخوانید. به عنوان مثال عبارت زیر سبب ایجاد نمونه ای از String بدون هر گونه کاراکتر در آن می شود.

```
String s=new String();
```

اغلب اوقات نیاز به ایجاد رشته هایی با مقدار اولیه خواهد داشت. کلاس String, سازنده های مختلفی برای انجام این کار فراهم کرده است. برای آن که یک String ایجاد و با آرایه ای از کاراکترها مقداردهی کنید از سازنده زیر استفاده نمایید:

```
String(char chars[])
```

به مثال زیر توجه کنید:

```
Char chars[]={ 'a','b','c'};  
String s = new string(chars);
```

سازنده بالا مقدار اولیه "abc" را به s تخصیص می دهد. با استفاده از constructor ذیل می توانید بخشی از یک آرایه را به عنوان مقدار اولیه مشخص کنید:

```
String(char chars[],int startindex,int numchars)
```

Startindex مشخص کننده ایندکس محل آغاز کارکترهای مورد نظر و numchars هم نشان دهنده تعداد کاراکترهایی است که باید به کار برده شوند. به مثال زیر توجه کنید:

```
Char chars[] = { 'a','b','c','d','e','f'};  
String s = new string (char,2,3);
```

عبارت بالا مقدار اولیه "cde" را به s تخصیص میدهد. با استفاده از سازنده زیر می توانید شیء String ای با مقدار یک شیء String دیگر ایجاد کنید.

```
String(String strObj)
```

strObj شیء نوع String است. مثال زیر را در نظر بگیرید:

```
//construct one string from another.  
Class Makestring{  
Public static void main(String args[]){  
Char c[] ={'j','a','v','a'};  
String s1 = new String(c);
```

```
String s2 = new String(s1);
System.out.println(s1);
System.out.println(s2);
}
}
```

خروجی برنامه در ذیل نشان داده شده است:

```
Java
Java
```

همانگونه که ملاحظه می کنیم رشته یکسانی در s1, s2 وجود دارد.

اگر چه داده های نوع char در جاوا از ۱۶ بیت برای نشان دادن مجموعه کاراکترهای یونی کد استفاده می کنند اما در فرمت رشته های مورد استفاده در اینترنت از آرایه های بایتی ۸ بیتی متشکل از مجموعه کاراکتر اسکی استفاده میشود. چون رشته های اسکی ۸ بیتی متداول می باشند کلاس String هم سازنده هایی را برای مقدار دهی رشته های در مواقع استفاده از یک آرایه نوع byte فراهم ساخته است. فرم آنها در زیر نشان داده شده است:

```
String(byte asciiChars[])
String(byte asciiChars[],int startIndex,int numChars)
```

AsciiChars مشخص کننده آرایه بایت ها است. فرم دوم نیز امکان مشخص کردن بخشی از کاراکتر های آرایه را فراهم می سازد. تبدیل بایت به کاراکتر در هر دو فرم بالا با استفاده از روش رمز گذاری پیش فرض محیط جاری انجام می شود. برنامه زیر از این سازنده ها استفاده میکند:

```
//construct string from subset of char array.
Class substringcons{
Public static void main (String args[]){
Byte ascii[]={65,66,67,68,69,70};
String s1=new String(ascii);
System.out.println(s1);
String s2=new String(ascii,2,3);
System.out.println(s2);
}
}
```

خروجی حاصل از اجرای برنامه در ذیل نشان داده شده است :

```
Abcdef
```

Cde

نگارشهای دیگری از سازنده های بایت به رشته نیز تعریف شده اند که در آنها می توانید روش رمز گزاری تبدیل بایت ها به کاراکتر ها را تعیین کنید اما بیشتر اوقات بهتر از روش پیش فرض محیط جاری استفاده نما یید.

توجه: وقتی یک شیء String از روی آرایه ای ایجاد می کنیم محتوای آرایه به آن کپی میشود. اگر محتوای آرایه را پس از ایجاد رشته تغییر دهید String تغییر نیافته باقی می ماند. با استفاده از سازنده زیر می توانید یک String از روی StringBuffer ایجاد نمایید.

```
String(StringBuffer strBufObj)
```

Constructor هایی که به وسیله ی J2SE 5 افزوده شده اند. دو سازنده به وسیله J2SE 5 به String افزوده شده اند. نخستین مورد که در ذیل نشان داده شده است از مجموعه کاراکتر های یونی کد گسترش یافته پشتیبانی می کند.

```
String(int codePoints[],int startindex,int numchars)
```

codePoints آرایه ای است که شامل کد های یونی کد است. رشته حاصل از محدوده ای که از startindex آغاز شده و numChars کاراکتر است تشکیل می شود. دومین سازنده جدید از کلاس StringBuilder پشتیبانی می کند. فرم کلی آن در زیر نشان داده شده است:

```
String(StringBuilder strBuilderObj)
```

متد بالا String ای از کلاس StringBuilder ارسالی در strBuilderObj ایجاد می کند.

## طول رشته ها

طول هر رشته تعداد کاراکتر های موجود در آن است. برای به دست آوردن این مقدار می توان متد length() را به صورت زیر فرا بخوانید: حاصل عبارات زیر ۳ است چرا که سه کاراکتر در رشته s وجود دارد.

```
Char chars[] = {'a','b','c'};
```



```
String s = new String(chars);  
System.out.println(s.length());
```

## عملیات ویژه رشته ها

از آنجایی که رشته ها بخش متداول و مهمی از برنامه سازی با جاوا به شمار می آیند جاوا امکانات ویژه ای برای عملیات مربوط به رشته ها در ساختار گرامری زبان گنجانده است. این عملیات شامل ایجاد خودکار نمونه های جدید String از روی لیترال های رشته ای، ادغام چندین شیء String با استفاده از عملگر "+" و تبدیل انواع داده های دیگر به رشته ها می باشد. متد های مجزایی برای انجام تمام این عملیات وجود دارد اما جاوا برای افزایش شفافیت برنامه ها و راحتی برنامه سازی این کارها را به طور خودکار انجام می دهد.

## لیترال های رشته ای

در مثال های پیشین نشان داده شد که چگونه می توان نمونه های String را از آرایه ای از کاراکترها با استفاده از عملگر new ایجاد نمود. اما روش آسانتری برای انجام این کار وجود دارد، استفاده از لیترال ها. جاوا برای هر لیترال رشته ای در برنامه تان یک شیء String را به طور خودکار می سازد. به عنوان مثال عبارات زیر دو رشته معادل را ایجاد می کنند:

```
Char chars[] = {'a','b','c'};  
String s1 = new String(chars);  
String s2 = "abc";//use string literal
```

چون یک شیء String برای هر یک از لیترال های رشته ای ایجاد می شود هر جا که شیء های String قابل استفاده باشند می توانید از لیترال ها استفاده نمایید. به عنوان مثال همان گونه که در مثال زیر نشان داده شده است متدها را میتوانید مستقیماً برای رشته هایی که بین علائم نقل قول نوشته می شوند فرا بخوانید درست همچون فرا خوانی آن ها با متغیرهای ارجاع به شیء های String در مثال زیر length() برای رشته "abc" فرا خوانده میشود که نتیجه آن نمایش ۳ است.

```
System.out.println("abc".length());
```

## ادغام رشته ها

جاوا به طور کلی امکان اعمال عملگرها به شیء های String را فراهم نمی سازد. تنها استثنا این موضوع عملگر "+" است که دو رشته را ادغام می کند و نتیجه آن یک شیء string است. این امر امکان استفاده چند عمل ادغام با "+" را فراهم می سازد. به عنوان مثال سه رشته در عبارات زیر با یکدیگر ادغام می شوند.

```
String age="9";
String s = "he is "+age+"years old";
System.out.println(s);
```

نتیجه عبارت بالا نمایش "he is 9 years old" است. یکی از کاربرد های ادغام رشته ها هنگام ایجاد رشته های بسیار طولانی نمایان میشود. به جای آن که رشته های طولانی در متن برنامه هایتان شکسته و از سطر بعد ادامه یابند می توانید آن ها را به چند قسمت تقسیم و با "+" ادغام کنید. به مثال های زیر توجه کنید:

```
//using concatenation to prevent long lines.
Class concat{
Public static void main(String args[]){
String longstr="this could have been"+
"a very long line that would have "+
"wrapped around.but string concatenation"+
"prevent this.";
System.out.println(longstr);
}
}
```

## ادغام رشته ها با انواع داده های دیگر

رشته ها را می توانید با انواع داده های دیگر ادغام کنید. به عنوان مثال نگارش نسبتاً متفاوتی از مثال پیش توجه کنید:

```
Int age = 9;
String s = "he is"+age+"years old.";
System.out.println(s);
```

Age در این مثال متغیر از نوع int است اما خروجی حاصل همچون مثال پیش است.  
زیرا مقدار int موجود در age به طور خودکار به رشته معادلش تبدیل میشود.

## استخراج کاراکترها

### charAt()

برای آن که کاراکتر واحدی را از یک String استخراج کنید با استفاده از متد charAt میتوانید مستقیماً کاراکتر مورد نظر را مشخص کنید. فرم کلی آن به صورت زیر است:

```
Char charAt(int where)
```

Where ایندکس کاراکتر مورد نظر است. مقدار where باید غیر منفی و مشخص کننده محلی از رشته باشد. charAt() کاراکتر محل مورد نظر را بر میگرداند به عنوان مثال

```
Char ch;
```

```
Ch="abc".charAt(1);
```

مقدار "b" را به ch تخصیص می دهد.

### GetChars()

اگر نیاز به استخراج بیش از یک کاراکتر داشته باشید، میتوانید از متد GetChars() استفاده نمایید. فرم کلی آن به صورت زیر است:

```
Void GetChars(int sourceStart,int sourceEnd,char target[],int  
targetStart)
```

targetStart مشخص کننده ایندکس ابتدای زیر رشته، و sourceEnd هم مشخص کننده ایندکس پس از آخرین کاراکتر زیر رشته مورد نظر است. از این رو، زیر رشته مورد نظر متضمن کاراکترهای targetStart تا sourceEnd-1 خواهد بود.

آرایه ای که کاراکتر ها در آن قرار می گیرند به وسیله target مشخص می شود. ایندکس مربوط به target جهت کپی کردن زیر رشته نیز در targetStart ارسال خواهد شد. باید دقت نمود که آرایه target باید به اندازه کافی برای ذخیره سازی کاراکتر های موجود در زیر رشته مورد نظر بزرگ باشد.

برنامه زیر عملکرد GetChars() را نشان می دهد:

```
Class getcharsDemo{
Public static void main(String args[]){
String s="this is a demo of the getchars method.";
Int start=10;
Int end=14;
Char buf[]=new char[end-start];
s. getChars(start,end,buf,0);
System.out.println(buf);
}
}
```

خروجی برنامه در ذیل نشان داده شده است:

Demo

## مقایسه رشته ها

کلاس String چندین متد دارد که رشته ها یا زیر رشته ها را مقایسه میکنند.

### **equalsIgnoreCase(),equals()**

برای آن که تساوی دو رشته را بررسی کنید از ( equals() استفاده کنید. فرم کلی آن به صورت زیر است:

### **Boolean equals(Object str)**

String شیء string ای است که با شیء String فعال کننده متد مقایسه میشود. چنانچه کاراکترهای موجود در رشته ها برابر باشند و ترتیب شان نیز یکسان باشد حاصل فراخوانی true خواهد بود. در غیر این صورت حاصل متد false خواهد بود. حروف بزرگ و کوچک در حین مقایسه یکسان تلقی نمی شوند.

برای آن که حروف بزرگ و کوچک در حین مقایسه یکسان تلقی شوند متد equalsIgnoreCase() را فرا بخوانید. حروف A-Z هنگام مقایسه با a-z یکسان تلقی می شوند. فرم کلی متد به صورت زیر است:

### **Boolean equalsIgnoreCase(String str)**

Str شیء String ای است که با شیء string فعال کننده متد مقایسه می شود. حاصل این متد نیز در صورت یکسان بودن کاراکتر ها و ترتیبشان true و در غیر این صورت false خواهد بود.

### **Equals() در مقابل ==**

لازم است بدانید که متد equals() و عملگر "==" دو عمل متفاوت انجام می دهند. همان گونه که پیش از این شرح داده شد متد equals() کاراکترهای موجود در شیء String مورد نظر را مقایسه می کند. عملگر "==" دو نشانی را مقایسه میکند تا مشخص شود که هر دو به نمونه یکسانی از یک شیء ارجاع دارند یا خیر.

### **compareTo()**

اغلب اوقات دانستن اینکه دورشته مشابه هستند یا خیر. کفایت نمی کند. برای برنامه های مرتب سازی باید بتوان تشخیص داد که کدام کاراکتر کوچکتر مساوی یا بزرگتر از مورد بعدی است. رشته ای کوچکتر از یک رشته دیگر تلقی میشود که از نظر ترتیب پیش از آن جایی داشته باشد. ورشته ای بزرگتر از یک رشته دیگر تلقی میشود که از نظر ترتیب پس از آن جای بگیرد. متد compareTo() کلاس string این کار را انجام میدهد. فرم کلی آن در ذیل نشان داده شده است:

### **Int compareTo(string str)**

Str همان string ای است که با string فعال کننده ی متد مقایسه می شود. نتیجه مقایسه برگردانده شده و به صورت زیر تفسیر می شود:

کوچکتر از صفر: رشته فعال کننده متد کوچکتر از str است.

بزرگتر از صفر: رشته فعال کننده متد بزرگتر از str است.

صفر: دورشته برابر هستند.

Compareto() بزرگی و کوچکی حروف را مقایسه در نظر می گیرد. کلمه ای که به آن اضافه شده است با یک حرف بزرگ شروع شده است.

اگر می خواهید بزرگی و کوچکی حروف هنگام مقایسه دورشته نادیده انگاشته شوند. از compareToIgnoreCase() استفاده کنید.

## Int compareToIgnoreCase(String str)

حاصل این متد همچون compareTo() است با این تفاوت که حروف بزرگ و کوچک یکسان تلقی میشوند.

## جستجوی رشته ها

کلاس String دو متد در اختیارتان می گذارد که با استفاده از آن ها می توانید هر رشته را برای کاراکتر یا کاراکتر های مورد نظر جستجو کنید:

indexOf() : نخستین نمونه از یک کاراکتر یا زیر رشته را جستجو می کند.

lastIndexOf() : آخرین نمونه از یک کاراکتر یا زیر رشته را جستجو می کند.

این دو متد به چند روش مختلف overload شده اند. در تمام موارد ایندکس محل آغاز کاراکتر یا زیر رشته در صورت موفقیت و -1 در صورت عدم موفقیت برگردانده می شود.

برای اینکه نخستین نمونه از یک کاراکتر را جستجو کنید , از فرم زیر استفاده نمایید:

```
Int indexOf(int ch)
```

برای آن که آخرین نمونه از یک کاراکتر را جستجو کنید, از فرم زیر استفاده نمایید:

```
Int lastIndexOf(int ch)
```

Ch کاراکتری است که جستجو می شود.

برای آن که نخستین یا آخرین نمونه از یک زیر رشته را جستجو کنید, از فرم های زیر استفاده نمایید:

```
Int indexOf(String str)
```

```
Int lastIndexOf(String str)
```

Str مشخص کننده زیر رشته مورد نظر است.

با استفاده از فرم های زیر می توانید نقطه آغاز جستجو را مشخص کنید:

```
Int indexOf(int ch,int startIndex)
```

```
Int lastIndexOf(int ch,int startIndex)
```

```
Int indexOf(String str,int startIndex)
```

```
Int lastIndexOf(String str,int startIndex)
```

startIndex مشخص کننده ایندکس محل آغاز جستجو است. جستجو در متد indexOf() از این ایندکس تا آخر رشته ادامه می یابد. در متد lastIndexOf() نیز جستجو از startIndex آغاز می شود.

```
//Demonstrate indexOf() and lastIndexOf()
Class indexofdemo{
Public static void main(String args[]){
String s="now is the time for all good men "+
"to come to the aid of their country.";
System.out.println(s);
System.out.println("indexOf(t)="+s.indexOf('t'));
System.out.println("indexOf(the)="+s.indexOf("the"));

System.out.println("lastIndexOf(the)="+s.lastIndexOf("the"));
System.out.println("indexOf(t,10)="+s.indexOf('t',10));
System.out.println("lastIndexOf(t,60)="+s.lastIndexOf('t',60));
System.out.println("indexOf(the,10)="+s.indexOf("the",10));
System.out.println("lastIndexOf(the,60)="+s.lastIndexOf("the",60));
}
}
```

خروجی حاصل از اجرای برنامه در زیر نشان داده شده است :

```
Now is the time for all good men to come to the aid of their country.
indexOf(t)=7
lastIndexOf(t)=65
indexOf(the)=7
last indexOf(the)=55
indexOf(t,10)=11
lastIndexOf(t,60)=55
indexOf(the,10)=44
lastIndexOf(the,60)=55
```

## تغییر رشته ها

چون شیء های string تغییر ناپذیر هستند هرگاه بخواهید یک string را تغییر دهید میبایست آنرا به StringBuffer یا StringBuilder کپی کنید و یا از یکی از متدهای زیر استفاده کنید

## Substring

با استفاده از این متد می توانید یک زیر رشته را استخراج نمایید. این متد دو فرم دارد. فرم نخست:

```
String substring(int startIndex)
```

startIndex ایندکس محل شروع زیر رشته را مشخص میکند. حاصل این فرم از متد زیر رشته ای است که از محل startIndex آغاز و تا انتهای رشته ادامه مییابد.

فرم دوم نیز امکان مشخص کردن ایندکس ابتدا و انتهای زیر رشته را فراهم میسازد.

```
String substring(int startIndex, int endIndex)
```

startIndex مشخص کننده ایندکس ابتدا و endIndex مشخص کننده نقطه انتهایی است. در برنامه زیر از substring برای جایگزین کردن تمام نمونه های یک زیر رشته با یک زیر رشته دیگر استفاده می شود.

```
//substring replacement.
Class stringreplace{
Public static void main(string args[]){
    String org = "This is atest.This is,too.";
    string search = "is";
    string sub = "was";
    string result = " ";
    int I;
    do {
        System.out.println(org);
        I = org.indexOf(search);
        if(I != -1)
        {
            result = org.substring(0,I) ;
            result = result + sub;
            result = result + org.substring(I + search.length());
            Org = result;
        }
    }
    While (I != -1);
}
```



```
}
```

خروجی:

```
This is a test.This is,too.  
Thwas is a test.This is,too.  
Thwas is a test.This is,too.  
Thwas is a test.Thwas is,too.  
Thwas is a test.Thwas is,too.
```

## Concat()

می توانید دو رشته را به کمک این متد ادغام کنید.

```
String concat(string str)
```

این متد سبب ایجاد شیء جدیدی می شود که حاوی رشته فعال کننده متد است که محتوای str به آن اضافه می شود. concat() عمل " + " را انجام می دهد. به عنوان مثال عبارت زیر سبب ذخیره شدن "onetwo" در s2 می شود.

```
String s1 = "one"  
String s2 = s1.concat("two");
```

## Replace()

این متد دو فرم دارد

فرم نخست : تمام نمونه های یک کاراکتر در رشته فعال کننده متد با کاراکتری دیگر جایگزین می شوند.

```
String replace(char original,char replacement)
```

Original مشخص کننده کاراکتری است که باید بوسیله کاراکتری جایگزین شود که با replacement مشخص می شود. رشته حاصل برگردانده می شود. به عنوان مثال عبارت زیر سبب ذخیره "Heww" در s میشود.

```
String a = "Hello".replace('l','w');
```

در فرم دیگر این متد یک سری کاراکتر با یک سری دیگر جایگزین می شود.

```
String replace(CharSequence original,Charsequence replacement)
```

## Trim()

این متد نسخه ای از رشته فعال کننده خود را برمی گرداند که تمام فاصله های ابتدایی و انتهایی از آن حذف شده اند.

```
String trim()
```

مثال:

```
String s = "Hello world".trim();
```

بدین ترتیب رشته "Hello world" در s قرار می گیرد.

### تبدیل داده ها با استفاده از `valueOf()`

این متد داده هارا از فرمت داخلی خودشان به فرم خوانا تبدیل می کند. این متد یک متد ایستاست که در کلاس `string` برای انواع داده های مختلف جاوا `overload` شده است تا تمامشان بدرستی به یک رشته قابل تبدیل باشند. `valueOf` برای نوع `object` نیز `overload` شده است تا شی های هر کلاسی به عنوان آرگومان این متد قابل استفاده باشند

چند فرم این متد در زیر نشان داده شده است

```
static String valueOf(double num)
static String valueOf(long num)
static String valueOf(Object ob)
static String valueOf(char chars[])
```

`valueOf()` هنگام نیاز به نمایش رشته ای انواع داده های دیگر فراخوانده می شود- به عنوان مثال در طی عملیات ادغام داده هاست. این متد مستقیما با هر نوع داده ای فرا خوانده می شود و حاصل آن نیز نمایش `string` گونه آرگومانش است. تمام انواع داده های پایه به نمایش رشته ای معادل خودشان تبدیل می شوند. هر شی که به `valueOf` ارسال میشود نتیجه فراخوانی متد `toString` شی را برمی گرداند

`Valueof()` برای بیشتر آرایه ها رشته نسبتا مرموزی را برمی گرداند که نشان می دهد نوعی آرایه است. اما در مورد آرایه های نوع `char` یک شی `string` ایجاد میکند که شامل کاراکترهای موجود در آرایه `char` است.

فرم کلی این متد:

```
static String valueOf (char chars[],int startIndex,int numchars)
```

chars آرایه ایست که کاراکترها در آن قرار دارند. startIndex مشخص کننده ایندکس آرایه کاراکترهایی است که زیررشته مورد نظر از آنجا آغاز می شود و numchars نیز طول رشته را مشخص می کند.

### تغییر کوچکی و بزرگی کاراکترها در یک رشته

متد toLowerCase() تمام حروف بزرگ یک رشته را به حروف کوچک تبدیل می کند.  
متد toUpperCase() تمام حروف کوچک یک رشته را به حروف بزرگ تبدیل می کند. تغییری در کاراکترهای غیر حرفی داده نمی شود.  
فرم های کلی این متدها:

```
String toLowerCase()  
String toUpperCase()
```

در مثال زیر از هر دو متد استفاده شده است :

```
//demonstrate toUpperCase() and toLowerCase().  
Class changeCase{  
Public static void main(String args[]){  
String s = "This is a test";  
System.out.println( "original: " + s);  
String upper = s.toUpperCase();  
String lower = s.toLowerCase();  
  
System.out.println("Uppercase: " + upper);  
System.out.println("Lowercase: " + lower);  
}  
}
```

خروجی :

```
Original: This is a test.  
Uppercase: THIS IS A TEST.  
Lowercase: this is a test.
```

## چند متد دیگر کلاس string

Int codepointAt (int i) ، کد پونیت یونی کد موقعیت i را برمی گرداند. بوسیله 5 J2SE اضافه شده است.

intcodePointBefore (int i) ، کد پونیت یونی کد موقعیت پیش از i را برمی گرداند. بوسیله 5 J2SE اضافه شده است.

Int codepointCount(int start,int end) ، تعداد کدپونیت های موجود در بین start و end-1 رادر String فعال کننده متد برمی گرداند بوسیله 5 J2SE اضافه شده است.

Boolean contains(CharSequence ,str) ، چنانچه رشته str درشئ فعال کننده متد وجود داشته باشد حاصل آن true و در غیر این صورت false خواهد بود. بوسیله 5 J2SE اضافه شده است.

Boolean contentEquals(Charsequence str) ، چنانچه رشته فعال کننده متد با رشته str یکسان باشد حاصل آن true و در غیر این صورت false خواهد بود. بوسیله 5 J2SE اضافه شده است.

## StringBuffer

StringBuffer, کلاسی نظیر string است که بسیاری از قابلیت رشته ها را دارد و دنباله تغییرناپذیری از کاراکترها با طول ثابت است. در مقابل StringBuffer نمایانگر دنباله ای قابل رشد و قابل نوشتن از کاراکترهاست کاراکترها و زیر رشته هایی را میتوان به میانه یا انتهای StringBuffer اضافه نمود. StringBuffer به طور خودکار رشد می کند تا فضای کافی برای این گونه اضافات فراهم شود و اغلب کاراکترهایی بیش از آنچه لازم باشد پیشاپیش به آن اختصاص یابد تا فضا برای رشد وجود داشته باشد. جاوا از هر دو کلاس زیاد استفاده می کند اما بسیاری از برنامه نویسان تنها با string سروکار دارند و با استفاده ها از عملگر "+" کار با StringBuffer در پشت پرده را به جاوا واگذار میکنند.

## سازنده های StringBuffer

```
StringBuffer()  
StringBuffer(int size)  
StringBuffer(String str)  
StringBuffer(CharSequence char)
```

سازنده پیش فرض که فاقد پارامتر است فضا برای ۱۶ کاراکتر را بدون تخصیص مجدد رزرو میکند. دومی یک آرگومان از نوع صحیح دارد که اندازه بافر را مشخص میکند. سومی آرگومانی از نوع string دارد که مقدار اولیه شیء stringBuffer را مشخص و فضا برای ۱۶ کاراکتر را بدون تخصیص مجدد رزرو میکند. وقتی طول مشخصی برای بافر درخواست نشود stringBuffer فضا برای ۱۶ کاراکتر اضافی را پیش بینی می کند چرا که تخصیص مجدد از نقطه نظر زمانی فرایندی پرهزینه به شمار می آید. همچنین تخصیص مجدد به طور مکرر سبب چند تکه شدن حافظه می شود stringBuffer با پیش بینی فضا برای چند کاراکتر اضافی تعداد تخصیص های مجدد احتمالی را کاهش می دهد. سازنده ی چهارم شیء ایجاد می کند که حاوی رشته ی مشخص شده در char خواهد بود.

## Length(),capacity()

اندازه جاری هر stringBuffer را میتوان بامتد length() و ظرفیت کامل آنرا از طریق متد capacity() بدست آورد فرم کلی به صورت زیر است.

```
int length()  
int capacity()
```

مثال:

```
//stringbuffer length vs.capacity  
Class stringBufferDemo{  
Public static void main(String args[]){  
StringBuffer sb=new StringBuffer("Hello");  
  
System.out.println("buffer = "+sb);  
System.out.println("length = "+sb.length());  
System.out.println("capacity =" + sb.capacity());  
}
```

```
}
```

خروجی برنامه نشان میدهد که `stringbuffer` چگونه فضای بیشتری برای پردازشهای آتی رزرو میکند :

```
Buffer = Hello  
Length = 5  
Capacity = 21
```

چون `sb` هنگام ایجاد با "Hello" مقداردهی می شود طول آن ۵ است. اما ظرفیت آن ۲۱ است چرا که فضا برای ۱۶ کاراکتر اضافی بطور خودکار به آن اضافه میشود.

## **ensureCapacity()**

اگر میخواهید فضا برای چند کاراکتر را پس از ایجاد `stringbuffer` پیشاپیش تخصیص دهید میتوانید از `ensureCapacity()` استفاده کنید. اگر از قبل بدانید که تعداد زیادی از رشته های کوچک را به `StringBuffer` اضافه خواهید کرد این امر مفید واقع خواهد شد. فرم کلی این متد به صورت زیر است:

```
Void ensureCapacity(int capacity)
```

`Capacity`, اندازه بافر را مشخص میکند.

## **setLength()**

برای آنکه اندازه بافر شی های `StringBuffer` را مشخص کند از `setLength` استفاده نمایید. فرم کلی آن به صورت زیر است.

```
Void setLength(int len)
```

`Len` اندازه بافر را مشخص می کند. مقدار آن باید مثبت باشد.

وقتی اندازه بافر را افزایش دهید کاراکترهای تهی به انتهای بافر موجود اضافه میشود. اگر `setLength()` را با مقداری کوچکتر از مقدار جاری حاصل از `length()` فراخوانی کنید در آن صورت کاراکترهای پس از طول جدید از بین خواهند رفت. برنامه ساده `setCharAtDemo` در قسمت زیر از `setLength()` برای کوتاه کردن `stringBuffer` استفاده میکند.

## charAt(),setCharAt()

مقدار هریک اراکاراکترهای یک stringBuffer را می توان از طریق متد charAt() بدست آورد. با استفاده از بدست آورد. با استفاده از setCharAt() نیز میتواند مقدار هرکاراکتر را در هر stringBuffer تعیین کنید. فرم کلی آن به صورت زیر است:

```
Char charAt(int where)
```

```
Void setCharAt(int where,char ch)
```

Where برای charAt() ایندکس کاراکتر مورد نظر را مشخص می کند و ch نیز مقدار جدید کاراکتر را مشخص میکند where در هر دو متد باید مثبت باشد و نباید محلی پس از انتهای بافر را مشخص کند.

مثال:

```
//Demonstrate charAt() and setCharAt().
Class setCharAtDemo{
Public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
System.out.println("buffer before = " + sb);
System.out.println("charAt(1) before= " + sb.charAt(1));
Sb.setCharAt(1,'i');
Sb.setLength(2);
System.out.println("buffer after = " + sb);
System.out.println("charAt(1) after= " + sb.charAt(1));
}
}
```

خروجی :

```
Buffer before = Hello
charAt (1) before = e
buffer after = Hi
charAt (1)after = i
```

## getchars()

برای آنکه زیررشته ای از یک stringBuffer را به آرایه ای کپی کنید از متد getChars() استفاده کنید فرم کلی آن به صورت زیر است:

```
Void getChars(int sourceStart,int sourceEnd,char target[],int  
targetStart)
```

sourceStart ایندکس ابتدای زیر رشته را مشخص می کند و sourceEnd نیز ایندکسی را مشخص میکند که به اندازه یک واحد بیش از انتهای زیررشته موردنظر است. یعنی زیر رشته از کاراکترهای sourceStart تا sourceEnd-1 را در بر خواهد گرفت. آرایه ای که کاراکترها به آن کپی می شوند بوسیله target مشخص می شود. ایندکس محل کپی شدن زیر رشته در target نیز از طریق targetStart ارسال می شود. باید دقت نمود که آرایه target باید برای ذخیره تعداد کاراکترهای مورد نظر فضای کافی داشته باشد.

## Append()

متد append معادل رشته هر نوع داده دیگر را به انتهای شیء StringBuffer فعال کننده متد اضافه می کند. این متد چندین نگارش overload شده دارد.

```
StringBuffer append(String str)  
Stringbuffer append(int num)  
StringBuffer append(Object obj)
```

String.valueOf() برای یکایک پارامترها فراخوانده می شود تا معادل رشته ای آنها را بدست آورد. نتیجه این کار به شیء StringBuffer جاری اضافه میشود. تمام نگارشهای append() خود بافر را برمی گردانند. این امر سبب می شود تا فراخوانیهای متد مزبور به طور زنجیره ای همچون مثال زیر انجام شود:

```
//Demonstrate append().  
Class appendDemo{  
Public static void main(string args[]){  
string s;  
int a=42;  
stringBuffer sb = new stringBuffer(40);  
s = sb.append("a = ").append(a).append("i").toString();  
  
System.out.println(s);  
}
```



خروجی:

```
a = 42!
```

متد `append()` بیشتر اوقات هنگام استفاده از عملگر "+" برای شیء های `string` فراخوانده می شود. جاوا تغییرات مربوط به نمونه های هر `string` را بطور خودکار به عملیات مشابه در نمونه های `stringBuffer` تغییر میدهد از این رو عمل ادغام سبب فعال شدن `append()` برای شیء `StringBuffer` میشود. کامپایلر پس از انجام عمل ادغام متد `toString()` را یک مرتبه دیگر فرا می خواند تا `StringBuffer` را به یک ثابت `String` تبدیل کند. تمام این کارها ممکن است به شکل غیر معقولی پیچیده به نظر رسد. چرا نباید یک کلاس `string` با همان رفتار `StringBuffer` داشت؟ پاسخ این پرسش کارایی است. محیط زمان اجرای جاوا با علم بر تغییر ناپذیر بودن شیء های `string` بهینه سازی های زیادی می تواند انجام دهد.

## Insert()

متد `insert()` یک رشته را در رشته دیگر درج می کند. این متد `overload` شده است تا علاوه بر `string` ها، `object` ها و `CharSequence` ها انواع داده های پایه را نیز بپذیرد. این متد نیز همچون `append()`، `String.valueOf()` را برای بدست آوردن معادل رشته ای که با آن فراخوانده شده است فرا می خواند.

این رشته سپس در شیء `StringBuffer` درج می شود. این متد چندین فرم مختلف دارد:

```
StringBuffer insert(int index,String str)
StringBuffer insert(int index,char ch)
StringBuffer insert(int index,Object obj)
```

ایندکس مشخص کننده محلی از شیء `StringBuffer` است که رشته در آن درج می شود.

برنامه زیر "like" را بین "I" و "Java" درج می کند :

```
//Demonstrate insert().
Class insertDemo {
Public static void main(String args[]) {
StringBuffer sb = new StringBuffer("I Java!");
Sb.insert(2,"like");
System.out.println(sb);
}
```

```
}  
}
```

خروجی :

I like Java!

## Reverse()

با استفاده از reverse() میتوانید کاراکترهای هرشیء StringBuffer را معکوس کنید:

```
StringBuffer reverse()
```

این متد شیء معکوس شده ای را برمیگرداند که هنگام فراخوانی ارسال شده است. برنامه زیر کاربرد این متد را نشان میدهد:

```
//Using reverse() to reverse astringBuffer.  
Class ReverseDemo{  
Public static void main(String args[]){  
StringBuffer s = new StringBuffer("abcdef");  
System.out.println(s);  
s.reverse();  
System.out.println(s);  
}  
}
```

خروجی :

abcdef  
fedcba

## delete(),deleteCharAt

با استفاده از این دو متد میتوانید کاراکترهای StringBuffer را حذف کنید.

```
StringBuffer delete(int startIndex,int endIndex)  
StringBuffer deleteCharAt(int loc)
```

متد delete() یکسری کاراکتر را ازشیء فراخوان حذف می کند. startIndex مشخص کننده ایندکس نخستین کاراکتری است که باید حذف شود و endIndex نیز مشخص کننده ایندکس نقطه

پس از آخرین کاراکتری است که باید حذف شود. از این رو کاراکترهای `startIndex` تا `startIndex-1` حذف میشوند. شیء `StringBuffer` حاصل برگردانده می شود. متد `deleteCharAt()` کاراکتر موجود در موقعیت `loc` را حذف می کند. این متد نیز شیء `StringBuffer` حاصل را برمی گرداند.

مثال:

```
//Demonstrate delete() and deleteCharAt()
Class deleteDemo {
Public static void main(String args[]){
StringBuffer sb = new stringBuffer("This is a test");
Sb.delete(4,7);
System.out.println("After delete: " + sb);
}
}
```

خروجی:

```
After delete: This is a test
After deleteCharAt: his a test
```

## Replace()

با فراخوانی متد `replace()` می توانید مجموعه ای از کاراکترها را در یک شیء `StringBuffer` جایگزین مجموعه دیگری بکنید.

```
StringBuffer replace(int startIndex,int endIndex,String str)
```

زیررشته ای که جایگزین می شود به وسیله ایندکس های `startIndex`,`endIndex` مشخص می شود از این رو زیررشته ای که از `startIndex` آغاز شده و تا `endIndex-1` ادامه دارد جایگزین می شود. رشته جایگزین هم در `str` ارسال می شود. شیء `stringbuffer` حاصل نیز برگردانده می شود.

```
//Demonstrate replace()
Class replaceDemo{
Public static void main(String args[]){
stringbuffer sb = new stringBuffer("This is a test");
sb.replace(5,7,"was");
System.out.println("After replace: " + sb);
}
```

```
}  
}
```

خروجی :

After replace: This was a test

## Substring()

با فراخوانی `substring()` می توانید قسمتی از `stringBuffer` را بدست آورید. این متد در دو فرم قابل استفاده است:

```
String substring (int startIndex)  
String substring ( int startIndex,int endIndex)
```

فرم نخست زیررشته ای را برمی گرداند که از این `startIndex` آغاز شده و تا انتهای شیء `StringBuffer` ادامه می یابد. فرم دوم نیز زیر رشته ای را بر می گرداند که از `startIndex` آغاز شده و تا `endIndex-1` ادامه می یابد. این متدها دقیقاً همچون موارد تعریف شده برای `string` کار می کنند.

## چند متد دیگر

`stringbuffer` علاوه بر متدهای بررسی شده متدهای دیگری نیز دارد که برخی از آنها به وسیله `J2SE` افزوده شده اند.

`stringbuffer appendCodePoint(int ch)` ، کد پونیت یونی کد را به انتهای شیء اضافه می کند. نشانی شیء برگردانده می شود. به وسیله `J2SE 5` اضافه شده است.

`Int codePointAt(int i)` ، کد پونیت یونی کدمحل `i` را برمیگرداند. به وسیله `J2SE 5` اضافه شده است.

`Int codepointBeffor (int start ,int end)` ، تعداد کد پونیت های بین `start` و `end-1` شیء را برمی گرداند به وسیله `J2SE 5` اضافه شده است.

`Int indexOf(String str)` ، `StringBuffer` را برای یافتن نخستین نمونه از `str` جستجو می کند. ایندکس کاراکتر پیدا شده و یا `1-` را در صورت عدم موفقیت بر می گرداند.

StringBuffer، Int indexOf(String str,int startIndex) را از startIndex-1 برای یافتن نخستین نمونه از str جستجو می کند. ایندکس کاراکتر پیدا شده و یا -1- را در صورت عدم موفقیت بر می گرداند.

StringBuffer ، Int lastIndexOf(String str) را برای یافتن آخرین نمونه از str جستجو می کند. ایندکس کاراکتر پیدا شده و یا -1- را در صورت عدم موفقیت بر می گرداند.

StringBuffer ، Int lastIndexOf(String str,int startIndex) را از startIndex-1 برای یافتن آخرین نمونه از str جستجو می کند. ایندکس کاراکتر پیدا شده و یا -1- را در صورت عدم موفقیت بر می گرداند.

Int offsetByCodepoints(int start,int num) ، ایندکس محلی از رشته فراخوان را بر می گرداند که num کد پس از ایندکس start قرار دارد به وسیله J2SE 5 اضافه شده است . CharSequense subsequence(int startIndex,int stopIndex) ، زیررشته از رشته فراخوان را از startIndex تا stopIndex برمی گرداند. این متد مورد نیاز رابط Charsequence است که اینک بوسیله StringBuffer پیاده سازی شده است.

## Void trimToSize()

اندازه بافر کاراکترها را برای شیء فراخوان کاهش می دهد تا مقدار جاری در آن جای گیرد. بوسیله J2SE 5 اضافه شده است.

به غیر از subSequense() که متد مورد نیاز رابط Charsequense را پیاده سازی میکند متدهای دیگر امکان جستجوی نمونه ای از یک string را برای stringBuffer فراهم می سازند. برنامه زیر کاربرد دو متد indexOf() و lastIndexOf() را نشان میدهد:

```
Class IndexOfDemo{
Static void main (String args[]){
stringBuffer sb = new StringBuffer (" one two one");
int I;
I = sb.indexOf("one");
System.out.println("First indrx:" + I);
I = sb.lastIndexOf("one");
System.out.println("Last indrx:" + I);
}
```

```
}  
}
```

خروجی:

```
First index: 0  
Last index: 8
```

## StringBuilder

J2SE 5 کلاس جدیدی را به قابلیت های جاری قدرتمند جاوا برای مدیریت رشته ها افزوده است این کلاس جدید `StringBuilder` نام دارد. این کلاس مشابه `StringBuffer` است اما یک تفاوت مهم دارد : سنکرون شده است یعنی `"thread-safe"` به شمار نمی آید. مزیت کلاس `StringBuilder` کارایی بیشتر است. اما در مواردی که از `multithreading` استفاده می کنید می بایست به جای `StringBuffer` از `StringBuilder` استفاده کنید.

## آرایه

آرایه در جاوا با کلاس پیاده سازی می شود. در جاوا هر آرایه ای که ایجاد میشود یک فیلد داده ای به نام length به طور خود کار تخصیص می یابد که اندازه آرایه را نگهداری می کند. در آرایه یک بعدی برای تعیین طول آرایه از فیلد length استفاده می شود. این فیلد توسط جاوا برای شیء آرایه منظور می گردد. در آرایه های دو بعدی نیز می توان از همین فیلد برای تعیین طول و عرض آرایه استفاده کرد. دستور زیر را ببینید:

```
Int [] [] x = new int [4] [5];
```

- x.length ، تعداد سطر ها را مشخص می کند
  - x[i].length ، تعداد ستون های سطر i ام را مشخص می کند ( تعداد ستون های سطر های مختلف می تواند متفاوت باشد ، البته در این مثال یکسان و برابر ۵ است.)
- کلاس آرایه در پکیج java.lang قرار دارد که در همه برنامه های جاوا به طور خودکار اضافه میشود. اما برای انجام کارهای اضافی روی آرایه ها می توان از کلاس های دیگری نیز استفاده کرد.

## کلاس Arrays

این کلاس محل مناسبی را برای انجام کارهای متداول بر روی آرایه ها فراهم می کند مثل مرتب سازی عناصر آرایه ، پر کردن عناصر آرایه با یک مقدار ، بررسی مساوی بودن محتویات دو آرایه و جستجو یک مقدار در آرایه.

کلاس Arrays در پکیج java.util قرار دارد ، برای استفاده از آن عبارت زیر به ابتدای برنامه باید اضافه شود:

```
Import java.util.*;
```

کلاس Arrays متدهای گوناگونی را فراهم می کند که هنگام کار با آرایه ها مفید واقع می شوند.

## متدها :

- `binarySearch()`: از یک جستجوی دودویی برای پیدا کردن مقدار مشخص شده استفاده می کند. این متد برای آرایه های مرتب شده به کار می رود. فرم کلی آن به صورت زیر است:

```
Static int binarySearch(type[] array, type key)
```

`array` آرایه ای است که باید جستجو شود ، `key` مقداری است که باید پیدا شود ، `type` یک از انواع داده اولیه یا انتزاعی است. چنانچه `array` حاوی عناصر غیر قابل مقایسه باشد (مثل `StringBuffer` ) یا نوع `key` با نوع عناصر `array` سازگار نباشد فراخوانی متد منجر به استثنای `ClassCastException` می شود. چنانچه `key` در آرایه موجود باشد ایندکس عنصر مربوطه برگردانده می شود.

- `equals()`: دو آرایه را مقایسه می کند چنانچه دو آرایه معادل باشند `true` برگردانده می شود. فرم کلی آن به صورت زیر است:

```
Static Boolean equals(type array1[], type array2)
```

- `fill()`: مقداری را به تمام عناصر موجود در آرایه تخصیص می دهد. فرم کلی آن به دو صورت زیر است:

```
static void fill(type array[], type value)
```

`value` به تمام عناصر موجود در `array` تخصیص می یابد.

```
static void fill(type array[], int from, int to, type value)
```

بخشی از آرایه را از اندیس `from` تا `to` با مقدار `value` پر میکند.

- `Sort()`: یک آرایه را به ترتیب صعودی مرتب می کند. که دو فرم دارد:

```
Static void sort(type array[])
```

```
Static void sort(type array[], int start, int end)
```

فرم اول کل آرایه را مرتب می کند. فرم دوم آن دسته از عناصر `array` که در محدوده `start` تا `end-1` قرار دارند مرتب می کند.



## کلاس Vector

Vector یک آرایه پویا را پیاده سازی می کند. در بردار نیاز به تعیین اندازه بردار نیست بلکه در صورت لزوم کوچک یا بزرگ می شود. متدهایی در این کلاس وجود دارند که برای افزودن ، دستیابی، حذف و درج عناصر در بردار به کار می آیند . کلاس Vector در پکیج java.util قرار دارد، برای استفاده از آن عبارت زیر به ابتدای برنامه باید اضافه شود:

```
Import java.util.*;
```

Vector سازنده های مختلفی دارد:

```
Vector()
```

```
Vector(int size)
```

```
Vector(int size,int incr)
```

فرم نخست بردار پیش فرضی ایجاد می کند که اندازه اولیه آن ۱۰ است. فرم دوم برداری ایجاد میکند که اندازه اولیه آن به وسیله size مشخص می شود. فرم سوم برداری ایجاد می کند که اندازه اولیه آن با size و نمو آن با incr مشخص می شود. مقدار نمو مشخص می کند که هر بار به هنگام افزایش اندازه چه تعداد عنصر به بردار تخصیص داده شود. البته نوع عناصر را می توان با سه سازنده بالا بیان کرد:

```
Vector<type>()
```

```
Vector<type> (int size)
```

```
Vector<type> (int size,int incr)
```

### متدها

- Void addElement(type element)

شیئی ای که به وسیله element مشخص می شود به بردار اضافه میشود.

- Int capacity()

اندازه بردار را برمی گرداند.

- Boolean contains(Object element)

چنانچه element در بردار باشد true برمی گرداند.

- Type elementAt(int index)

عنصر موجود در موقعیت ایندکس را بر می گرداند.

- Boolean isEmpty()

چنانچه بردار خالی باشد true بر می گرداند.

- `Void removeAllElements()`

بردار را خالی می کند. پس از اجرای متد اندازه بردار صفر می شود.

- `Boolean removeElement(Object element)`

Element را از بردار حذف می کند در صورت موفقیت `true` بر می گرداند.

## کلاس Stack

Stack زیر کلاسی از Vector است که یک پشته LIFO استاندارد را پیاده سازی می کند. به صورت زیر تعریف می شود:

`Class Stack<E>`

E مشخص کننده نوع عنصری است که در پشته ذخیره می شود. Stack علاوه بر متد هایی که در Vector تعریف شده اند متد های خاص خود را نیز دارد.

## متدها

- `Boolean empty()`

چنانچه پشته خالی باشد `true` بر می گرداند.

- `E peek()`

عنصر بالای پشته را برمی گرداند اما آن را حذف نمی کند.

- `E pop()`

عنصر بالای پشته را برمی گرداند و آن را حذف می کند.

- `E push(E element)`

Element را به پشته اضافه می کند.

- `Int search(Object element)`

Element را در پشته جستجو میکند چنانچه پیدا شود افسر آن نسبت به بالای پشته برگردانده می شود. در غیر این صورت -1 بر می گرداند.

جاوا کلاس های مختلفی برای پیاده سازی انواع ساختمان داده ها دارد.

- ArrayList •
- LinkedList •
- HashSet •
- LinkedHashSet •
- TreeSet •
- PriorityQueue •
- ... •

## فایل ها

### ورودی/خروجی

بیشتر برنامه های کاربردی واقعی جاوا برنامه های کنسولی مبتنی بر متن نیستند. بلکه در عوض برنامه های گرافیک گزایی هستند که برای برقراری ارتباط با کاربر بر AWT(Abstract Window Toolkit) یا Swing جاوا اتکا دارند. برنامه های مبتنی بر متن از کاربردهای مهم جاوا در کارهای واقعی به شمار نمی آیند. پشتیبانی جاوا از I/O کنسولی محدود بوده و استفاده از آن نیز قدری مشکل ساز است. حتی در برنامه های نمونه ساده I/O کنسولی مبتنی بر متن در برنامه سازی جاوا چندان مهم نیست. اما در عین حال جاوا در رابطه با فایل ها و شبکه ها امکانات قوی و انعطاف پذیری برای I/O دارد. سیستم I/O جاوا نوعی پیوستگی و یکپارچگی دارد.

### استریم ها

برنامه های جاوا عملیات I/O را از طریق استریم ها انجام میدهند. منظور از استریم سطحی انتزاعی است که اطلاعات را تولید یا مصرف میکند. هر استریم به وسیله سیستم I/O جاوا به یک وسیله ی فیزیکی مرتبط میشود. تمام استریم ها به یک شکل عمل می کنند حتی اگر وسایل فیزیکی مرتبط با آنها متفاوت باشند. از این رو کلاس ها و متدهای I/O یکسانی را میتوان برای هر نوع وسیله به کاربرد. این به این معناست که هر استریم ورودی میتواند انواع مختلف زیادی از ورودی ها را از یک دیگر مجزا سازد: فایلی از یک دیسک، صفحه کلید یا سوکتی از شبکه. همین طور هر استریم خروجی نیز ممکن است با کنسول فایلی از یک دیسک یا اتصال شبکه مرتبط باشد. استریم ها روش شفافی برای مدیریت I/O به شمار می آیند و دیگر نیازی به آگاهی از قسمت های مختلف برنامه نسبت به تفاوت بین یک صفحه کلید و شبکه نیست. جاوا استریم ها را در کلاس های تعریف شده در پکیج java.io پیاده سازی می کند.

### استریم های بایتی و کاراکتری

دو نوع استریم در جاوا تعریف شده است: بایتی و کاراکتری. استریم های بایتی روش مناسبی را برای مدیریت I/O داده های بایتی فراهم میسازد. به عنوان مثال از این استریم ها برای خواندن یا نوشتن داده های باینری استفاده می شود. استریم های کاراکتری روش مناسبی برای مدیریت I/O کاراکترها فراهم ساخته اند. این استیم ها از یونی کد استفاده می کنند و بنابراین می توانند برای انواع زبان های بین المللی مورد استفاده قرار گیرند. این استریم ها در برخی از موارد کارآمد تر از استریم های بایتی هستند.

استریم های کاراکتری در نگارش نخست جاوا (1.0) تعریف نشده بودند و از این رو تمام عملیات I/O بایت گرا بودند. استریم های کاراکتری به جاوا 1.1 افزوده شدند و برخی از متد ها و کلاس های بایت گرا کنار گذاشته شدند. به همین دلیل است که برنامه های قدیمی که از استریم های کاراکتری استفاده نمی کنند می بایست جهت استفاده از آن ها به روز رسانی شوند.

**نکته دیگر:** تمام عملیات I/O هنوز هم در پایین ترین سطح بایت گرا هستند. استریم های مبتنی بر کاراکتر صرفاً روش مناسب و کارآمدی برای مدیریت کاراکتر ها فراهم می سازد.

### کلاس های استریم های بایتی

استریم های بایتی به وسیله دو شاخه از ساختار سلسله مراتبی کلاس ها تعریف شده اند. دو کلاس انتزاعی در بالا ترین سطح این کلاس ها قرار دارند: `OutputStream`, `InputStream`. هر یک از این کلاس های انتزاعی چندین زیر کلاس دارند که تفاوت های بین وسایل I/O مختلف از جمله فایل های روی دیسک ها , اتصالات شبکه و حتی بافرهای حافظه را مدیریت می کنند. برخی از این کلاس ها در این قسمت مورد بررسی قرار می گیرند. به خاطر داشته باشید که برای استفاده از کلاس های استریم ها باید پکیج `java.io` را وارد کنید.

در دو کلاس انتزاعی `OutputStream`, `InputStream` چندین متد کلیدی تعریف شده اند که به وسیله کلاس های دیگر پیاده سازی می شوند. و مهمترین این متد ها `read()`, `write()` هستند که برای خواندن و نوشتن بایت هایی از داده ها به کار برده می شوند. هر دو متد به صورت انتزاعی در `OutputStream`, `InputStream` تعریف شده اند. این متد ها به وسیله زیر کلاس ها `override` می شوند.

## کلاس های استریم های بایتی:

BufferedInputStream : استریم ورودی بافر شده.  
BufferedOutputStream : استریم خروجی بافر شده.  
ByteArrayInputStream : استریم ورودی که از یک آرایه بایتی می خواند.  
ByteArrayOutputStream : استریم خروجی که در یک آرایه می نویسد.  
DataInputStream : استریم ورودی که متدهایی برای خواندن انواع داده های استاندارد جاوا دارد.  
DataOutputStream : استریم خروجی که متدهایی برای نوشتن انواع داده های استاندارد جاوا دارد.  
FileInputStream : استریم ورودی که از یک فایل می خواند.  
FileOutputStream : استریم خروجی که در یک فایل می نویسد.  
FilterInputStream : Inputstream را پیاده سازی می کند.  
FilterOutputStream : Outputstream را پیاده سازی می کند.  
InputStream : کلاس انتزاعی که استریم ورودی را تشریح می کند.  
ObjectInputStream : استریم ورودی برای شی ها.  
ObjectOutputStream : استریم خروجی برای شی ها.  
OutputStream : کلاس انتزاعی که استریم خروجی را تشریح می کند.  
PipedInputStream : پایپ ورودی.  
PipedOutputStream : پایپ خروجی.  
PrintStream : استریم خروجی متضمن print(),println() .  
PushbackInputStream : استریم ورودی که از unget تک بایتی پشتیبانی می کند. یک بایت را به استریم ورودی بر می گرداند.  
RandomAccessFile : از I/O تصادفی در فایل پشتیبانی می کند.  
SequenceInputStream : استریم ورودی که ترکیبی از دو استریم ورودی است که به طور متوالی و یکی پس از دیگری خوانده می شوند.

## کلاس های استریم های کاراکتری

استریم های کاراکتری به وسیله دو شاخه از سلسله مراتبی کلاس ها تعریف شده اند. دو کلاس انتزاعی در بالاترین سطح این کلاس ها قرار دارند: Reader, Writer. این کلاس های انتزاعی استریم های کاراکتری یونی کد را مدیریت می کنند. چندین زیر کلاس از هر یک از این دو کلاس در جاوا مشتق شده اند. کلاس های استریم های کاراکتری در زیر ذکر شده اند. چندین متد کلیدی در دو کلاس انتزاعی Reader, Writer تعریف شده اند که به وسیله کلاس های دیگر پیاده سازی میشوند. مهمترین این متد ها read(), write() هستند که برای خواندن و نوشتن کاراکتر هایی از داده ها به کار برده می شوند. این متد ها به وسیله زیر کلاس ها override می شوند.

## کلاس های I/O استریم کاراکتری

BufferedReader : استریم کاراکتری ورودی بافر شده.  
BufferedWriter : استریم کاراکتری خروجی بافر شده.  
CharArrayReader : استریم ورودی که از یک آرایه کاراکتری می خواند.  
CharArrayWriter : استریم خروجی که در یک آرایه کاراکتری می نویسد.  
FileReader : استریم ورودی که از یک فایل می خواند.  
FileWriter : استریم خروجی که در یک فایل می نویسد.  
FilterReader : استریم ورودی فیلتر شده.  
FilterWriter : استریم خروجی فیلتر شده.  
InputStreamReader : استریم ورودی که بایت ها را به کاراکتر ها تبدیل می کند.  
LineNumberReader : استریم ورودی که سطر ها را شمارش می کند.  
OutputStreamWriter : : استریم خروجی که کاراکتر ها را بایت ها تبدیل می کند.  
PipedReader : پایپ ورودی.  
PipedWriter : پایپ خروجی.  
PrintWriter : استریم خروجی متضمن print(), println() .  
PushbackReader : استریم ورودی که امکان بازگرداندن کاراکتر ها به استریم ورودی را فراهم می سازد.

Reader: کلاس انتزاعی که استریم های ورودی کاراکتری را تعریف می کند.  
StringReader: استریم ورودی که از یک رشته می خواند.  
StringWriter: : استریم خروجی که در یک رشته می نویسد.  
Writer: کلاس انتزاعی که استریم های خروجی کاراکتری را تعریف می کند.

### استریم های از پیش تعریف شده

همان گونه که می دانید تمام برنامه های جاوا پکیج java.lang را به طور خود کار وارد می کنند. در این پکیج کلاسی به نام System تعریف شده است که جنبه های مختلفی از محیط زمان اجرا را نهان می کند. به عنوان مثال با استفاده از برخی از متد های این کلاس می توانید زمان جاری و تنظیمات خصوصیات مربوط به سیستم را به دست آورید. سه متغیر استریم از پیش تعریف شده نیز در کلاس System وجود دارد: err, out, in. این فیلد ها به صورت ایستا و عمومی در System تعریف شده اند.

System.out به استریم خروجی استاندارد ارجاع دارد. این استریم طبق پیش فرض همان کنسول است. System.in نیز به ورودی استاندارد ارجاع دارد که طبق پیش فرض صفحه کلید است. System.err هم به استریم خطاهای استاندارد ارجاع دارد که طبق پیش فرض کنسول است. اما این استریم ها را می توان به هر وسیله I/O سازگار دیگری هدایت نمود.

System.in شی ای از نوع InputStream است. System.out, System.err نیز شی هایی از نوع PrintStream هستند. گرچه این استریم ها عموماً برای خواندن/نوشتن کاراکترها به/در کنسول استفاده می شود اما استریم های بایتی به شمار می آیند. همان گونه که خواهید دید در صورت نیاز می توانید آن ها را در استریم های مبتنی بر کاراکتر پیوشانید.

### خواندن ورودی های کنسول

تنها روش خواندن ورودی های کنسول در جاوا 1.0 استفاده از یک استریم بایتی بود از این رویه هنوز هم در برنامه های قدیمی تر متداول است. امروزه استفاده از استریم های بایتی برای خواندن ورودی های کنسول هنوز هم از نظر فنی میسر است اما انجام این کار توصیه نمی شود. روش از



پیش تعریف شده برای خواندن ورودی های کنسول استفاده از استریم کاراکترگرا است که استفاده از آن ها نگره داشت و بین المللی کردن برنامه ها را آسان تر ساخته است.

خواندن ورودی های کنسول در جاوا از طریق خواندن از System.in انجام می شود. برای خواندن استریم های مبتنی بر کاراکتر مرتبط با کنسول می بایست System.in را در شی ای از نوع BufferedReader بپوشانید. BufferedReader از استریم ورودی بافر شده پشتیبانی می کند. متداول ترین Constructor مورد استفاده به شکل زیر است :

`BufferedReader(Reader inputReader)`

inputReader استریم مرتبط با نمونه ای از BufferedReader است که ایجاد میشود. Reader نوعی کلاس انتزاعی است. یکی از زیر کلاس های آن InputStreamReader است. که بایت ها را به کاراکتر تبدیل میکند. برای ایجاد شی ای از نوع InputStreamReader که با System.in مرتبط باشد از constructor ذیل استفاده کنید:

`InputStreamReader(InputStream inputStream)`

چون Stream.in به شی ای از نوع InputStream ارجاع دارد از آن میتوان برای استریم ورودی استفاده نمود. در سطر زیر یک BufferedReader ایجاد می شود که به صفحه کلید متصل میشود :

```
BufferedReader br= new BufferedReader (new InputStreamReader (System.in));
```

پس از اجرای سطر بالا br یک استریم مبتنی بر کاراکتر خواهد بود که از طریق System.in با کنسول مرتبط است.

## خواندن کاراکتر ها

برای آن که کاراکتری را از یک BufferedReader بخوانید از Read() استفاده کنید. نگارشی از Read() که به کار خواهیم برد به صورت زیر است:

`int Read()throws IOException`

هر بار که Read() فراخوانده می شود کاراکتری را از استریم ورودی می خواند و آن را به صورت یک مقدار صحیح بر میگرداند. چنانچه به انتهای استریم ۱- را بر میگرداند. همان گونه

که ملاحظه می کنید استثنای `IOException` را نیز می تواند پرتاب کند.

برنامه ذیل روش استفاده از `read()` را با خواندن کاراکتر ها از کنسول نشان می دهد. این کار را آنقدر ادامه می دهد تا کلید حرف "q" فشار داده شود:

```
//use a BufferedReader to read characters from the console.
import java.io.*;
class BRRead{
    public static void main(String args[]) throws IOException
    {
        char c;
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter characters,'q' to quit.");
        //read characters
        do{
            c=(char)br.read();
            System.out.println(c);
        }while(c!='q');
    }
}
```

نمونه ای از خروجی برنامه در ذیل نشان داده شده است:

```
Enter characters,'q' to quit.
123abcq
1
2
3
A
B
C
Q
```

خروجی بالا ممکن است با آنچه انتظارش را دارید قدری تفاوت داشته باشد چرا که `System.in` طبق پیش فرض سطر را به صورت با فر شده می خواند. این بدین معناست که تا وقتی کلید `enter` زده نشود ورودیها به برنامه ارسال نمیشوند. این امر سبب میشود تا `read()` برای دریافت ورودیها از کنسول چندان ارزشمند نباشد.

**خواندن رشته ها**

برای آنکه رشته ای را از صفحه کلید بخوانید از `readline()` استفاده کنید که عضوی از کلاس `bufferedReader` است. فرم کلی آن به صورت زیر است.

```
String readline() throws IOException
```

همان گونه که ملاحظه می کنید حاصل آن یک شیء `string` است. برنامه زیر عملکرد `bufferedReader` و متد `readline()` را نشان میدهد این برنامه سطرها را آن قدر یک به یک می خواند و نمایش می دهد تا کلمه `word` را تایپ کنید:

```
//read a string from console using a bufferedreader.
import java . io .*;
class breadlines {
    public static void main (string args[] )
    throws IOException
    {
        //create a bufferedreader using system. In
        BufferedReader br = new bufferedreader (new inputstreamreader( system.in
        ));
        String str;

        System .out. println( enter lines of text. );
        system .out. println(enter `stop` to quit.``);
        do {
            str =br .readline();
            system.out. println(str);
        } while(!str.equals (``stop``));
    }
}
```

در برنامه زیر یک ویراستار متنی ساده ایجاد شده است. آرایه ای از شیء های `string` ایجاد و سپس سطرها خوانده شده و هر سطر در آرایه ذخیره میشود. حداکثر ۱۰۰ سطر خوانده می شود. یا اینکه کلمه ```stop``` را تایپ کنید برای خواندن از کنسول `BufferRead` استفاده شده است:

```
//A tiny editor
```

```

import java.io.*;
class TinyEdit{
public static void main(String args[])throws IOException{
//creat a bufferreader using System.in
BufferedReader br = new BufferedReader(new

InputStreamReader(System.in));
String str[] = new String[100];
System.out. println("enter lines of text.");
System.out. println("enter 'stop' to quit.");
for(int i=0;i<100;i++){
str[i]=br.readLine();
if(str[i].equals("stop"))break;
}
System.out. println("\nHere is your file:");
//display the lines
for(int i=0;i<100;i++){
if(str[i].equals("stop"))break;
System.out. println(str[i]);
}
}
}
}

```

نمونه ای از خروجی برنامه در ذیل نشان داده شده است:

```

Enter lines of text.
Enter 'stop' to quit.
This is line one.
This is line two.
Java makes working with strings easy.
Just creat String objects.
Stop
Here is your file:
This is line one.
This is line two.
Java makes working with strings easy.
Just creat String objects.

```

نوشتن خروجی کنسول

خروجی کنسول می تواند با `print()`, `println()` انجام شود. این متد ها به وسیله کلاس `PrintStream` (نوعی شی که `System.out` به آن ارجاع دارد) تعریف شده اند. اگرچه `System.out` نوعی استریم بایتی است اما استفاده از آن برای خروجی برنامه ای ساده هنوز قابل پذیرش است.

چون `PrintStream` نوعی استریم خروجی است که از `OutputStream` مشتق شده است متد سطح پایین `write()` را نیز پیاده سازی می کند. از این رو از `write()` می توان برای نوشتن در کنسول استفاده نمود. ساده ترین فرم `write()` که به وسیله `PrintSystem` تعریف شده است به صورت زیر می باشد:

```
Void write (int byteval)
```

این متد بایت مشخص شده با `byteval` در استریم می نویسد. اگر چه `byteval` به عنوان عدد صحیح تعریف شده است اما تنها هشت بیت سمت راست نوشته می شوند. در مثال کوتاه زیر از `write` برای نوشتن کاراکتر A و سپس کاراکتر انتقال به سطر بعد استفاده شده است:

```
//Demonstrate System.out.write()  
Class WriteDemo{  
Public static void main(String args[]){  
Int b;  
B='A';  
System.out.write(b);  
System.out.write('\n');  
}  
}
```

برای انجام خروجی های کنسول اغلب از `write()` استفاده نمی شود چرا که استفاده از `print()`, `println()` آسان تر است.

## کلاس `PrintWriter`

اگر چه استفاده از `System.out` برای نوشتن در کنسول قابل پذیرش است اما استفاده از آن عمدتاً برای مقاصد اشکال زدایی یا نمونه برنامه های ساده توصیه می شود. روشی که در برنامه های واقعی برای نوشتن در کنسول توصیه می شود از طریق استریم `PrintWriter` است.

PrintWriter یکی از کلاس های مبتنی بر کاراکتر است. استفاده از کلاس مبتنی بر کاراکتر جهت نوشتن خروجی ها در کنسول بین المللی ساختن برنامه ها را آسان تر می کند.

چند سازنده در PrintWriter تعریف شده است. یکی از این متد ها به صورت زیر است:

```
PrintWriter(OutputStream outputstream, Boolean flushOneNewLine)
```

OutputStream شی ای از نوع OutputStream است و flushOneNewLine مشخص می کند که استریم خروجی هر بار هنگام فراخوانی println() تخلیه شود یا خیر. چنانچه مقدار آن true باشد عمل تخلیه به طور خودکار انجام می شود. اما اگر مقدار آن false باشد این کار به طور خودکار انجام نخواهد شد.

برای آنکه PrintWriter برای نوشتن در کنسول استفاده نمایم زیر نمونه ای از از System.out برای استریم خروجی استفاده نماییم. استریم را پس از هر سطر جدید تخلیه کنید. به عنوان مثال PrintWriter را ایجاد و با کنسول خروجی مرتبط می کند:

```
PrintWriter pw= new PrintWriter(System.out ,true);
```

برنامه زیر نشان می دهد که چگونه از PrintWriter برای مدیریت کنسول خروجی استفاده میشود:

```
//demonstrate PrintWriter
Import java.io.*;

Public class PrintWriterdemo{
Public static void main(String args[]){
PrintWriter pw = new PrintWriter(System.out ,true);
Pw.println("this is a string");
Int I =-7;
Pw.println(i);
Double d=4.5e-7;
Pw.println(d);
}
}
```

خروجی برنامه به صورت زیر است:

```
This is a string
-7
```

## خواندن از /نوشتن در فایل ها

جاوا چندین کلاس و متد را برای خواندن از/نوشتن در فایل ها فراهم کرده است. تمام فایل ها در جاوا بایت گرا هستند و جاوا متد هایی را برای خواندن/نوشتن بایت ها از / به فایل ها فراهم ساخته است. با این وجود جاوا امکان پوشاندن استریم های بایت گرای فایل ها را در شی های مبتنی بر کاراکتر فراهم ساخته است.

دو مورد از کلاس هایی که بیشتر مورد استفاده قرار می گیرند `FileOutputStream`, `FileInputStream` هستند که استریم های بایتی مرتبط با فایل ها را ایجاد می کنند. برای آن که فایلی را باز کنید کافی است شی ای از این نوع کلاس ها را ایجاد و نام فایل را به عنوان آرگومان سازنده مشخص نمایید. اگر چه هر دو کلاس از سازنده های اضافی `override` شده پشتیبانی می کنند اما ما فقط دو فرم زیر را توضیح می دهیم:

```
FileInputStream(String fileName) throws FileNotFoundException
FileOutputStream(String fileName) throws FileNotFoundException
```

`filename` نام فایلی را مشخص می کند که باید باز شود. وقتی یک استریم ورودی ایجاد می کنید چنانچه فایل موجود نباشد در آن صورت استثنای `FileNotFoundException` پرتاب می شود. برای استریم های خروجی نیز چنانچه فایل موجود نباشد در آن صورت استثنای `FileNotFoundException` پرتاب می شود.

وقتی کارتان با فایلی به پایان رسید می بایست آن را با فرا خوانی `close()` ببندید. هر دو کلاس `FileOutputStream`, `FileInputStream` آن را تعریف کرده اند:

```
Void close() throws IOException
```

برای خواندن از هر فایل می توانید از یکی از نگارش های `read()` استفاده کنید که در کلاس `FileInputStream` تعریف شده است یکی از این نگارش ها به صورت زیر است:

```
Int read() throws IOException
```

هر بار که این متد فرا خوانده می شود یک بایت واحد را از فایل می خواند و آن را به صورت یک عدد صحیح بر می گرداند. پس از رسیدن به انتهای فایل نیز 1- برگردانده می شود. در صورت بروز هر گونه خطا نیز استثنای IOException پرتاب می شود.

در برنامه زیر از read() برای خواندن ورودی ها استفاده شده و محتوای فایل نمایش داده می شود. نام فایل به عنوان آرگومان در خط فرمان مشخص می شود. به کاربرد بلوک های try/catch برای مدیریت دو خطایی که ممکن است در حین اجرای برنامه پیش بیایند توجه کنید-خطای پیدا نشدن فایل مورد نظر با فراموش کردن تایپ نام فایل در خط فرمان. از این رویه می توان همیشه هنگام استفاده از آرگومان های خط فرمان استفاده نمایید.

```
/*display a text file.
To use this program ,specify the name
Of the file that you want to see.
For example to see a file called TEST.TXT,
Use the following command line:

Java showFile TEST.TXT
*/

Import java.io.*;

Class showFile{
Public static void main(String args[])throws IOException{
Int I;
FileInputStream fin;
Try{
Fin = new FileInputStream(args[0]);
}catch(FileNotFoundException e){
System.out. println("file not found");
Return;
}catch(ArrayIndexOutOfBoundsException e){
System.out. println("usage: showfile file");
Return;
}
//read charactors until EOF is encountered
Do{
I=fin.read();
```



```

If( I != -1)System.out.print((char)i);
}while(I != -1);
Fin.close();
}
}

```

برای نوشتن در فایل نیز می توانید از متد `write()` که به وسیله کلاس `FileOutputStream` تعریف شده است استفاده کنید. ساده ترین فرم آن به صورت زیر است:

```
Void write(int byteval) throws IOException
```

بایت مشخص شده با `byteval` در فایل نوشته می شود. اگرچه `byteval` به صورت `int` تعریف شده است اما تنها هشت بیت سمت راست در فایل نوشته می شوند. در صورت بروز هر گونه خطا نیز استثنای `IOException` پرتاب می شود. در مثال زیر از `write()` برای کپی کردن یک فایل متنی استفاده شده است.

```

/*copy a text file.
To use this program specify the name of
The source file and the destination file.
For example to copy a file called EIRST.TXT
To a file called SECOND.TXT ,use the following
Command line.

Java CopyFile FIRST.TXT SECOND.TXT
*/
Import java.io.*;

Class CopyFile{
Public static void main(String args[])throws IOException{
Int I;
FileInputStream fin;
FileOutputStream fout;
Try{
//open input file
Try{
Fin=new FileInputStream(arg[0]);
}catch(FileNotFoundException e){
System.out. println("file not found");
}
}
}
}

```

```

Return;
}
//open output file
Try{
Fout =new FileOutputStream(arg[1]);
}catch(FileNotFoundException e){
System.out. println("error opening output file");
Return;
}
}catch(ArrayIndexOutOfBoundsException e){
System.out. println("usage: CopyFile  from to");
Return;
}
//copy file
Try{
Do{
I=fin.read();
If(I != -1)fout.write(i);
}while(I != -1);
}catch(IOException e){
System.out. println("file error");
}
Fin.close();
Fout.close();
}
}

```

جاوا بر خلاف سایر زبان های کامپیوتری از جمله c,c++ که از کد خطا ها برای گزارش خطاهای فایل ها استفاده می کنند، از مکانیسم مدیریت استثناها استفاده میکند. این امر نه تنها سبب شفافیت مدیریت فایل ها میشود بلکه به جاوا امکان می دهد تا به آسانی بین خطاهای رسیدن به انتهای فایل و خطاهای در حین خواندن از فایل تمایز قائل شود. در دو زبان c,c++ بسیاری از توابع هنگام بروز خطا در حین خواندن و هنگام رسیدن به انتهای فایل کدهای یکسانی را برمی گردانند (یعنی شرط EOF در C/C++ اغلب با همان مقدار خطای خواندن اعلام می شود). این امر معمولاً بدین معناست که برنامه ساز باید عبارات بیشتری در برنامه بگنجاند تا مشخص شود که کدام رویداد واقعاً پیش آمده است. خطا ها در زبان جاوا از طریق استثنا ها به برنامه هایتان

اعلام می شوند و نه از طریق کدهایی که `read()` بر می گرداند. از این رو وقتی `read` مقدار 1- را بر می گرداند معنای آن همیشه واحد است وضعیت EOF پیش آمده است.

توجه: نام فایل های مورد نظر مثل `FIRST.TXT` و `SECOND.TXT` باید به عنوان آرگومان متد `main()` در خط فرمان باشد و به `main()` ارسال شود یعنی برای اجرای برنامه بالا به صورت زیر در خط فرمان اعلام میکنیم:

```
Java CopyFile FIRST.TXT SECOND.TXT
```

# مدیریت حافظه

## عناوین این بخش :

مدیریت حافظه در جاوا

نحوه عملکرد garbage collector

## مدیریت حافظه در Java

یکی از تفاوت های Java با زبانی مثل c و ++c چگونگی مدیریت حافظه آن است. مدیریت حافظه در Java به گونه ای است که این زبان نیازی به اشاره گر ندارد. در واقع خود ماشین مجازی Java ، کار با اشاره گرها را به عهده گرفته و این عمل حساس و خسته کننده را از اختیار کاربر خارج کرده است. همچنین یکی از مهمترین مزایای مدیریت حافظه در Java ، آسغال جمع کن (garbage collector) آن است. اگر با زبان های c و ++c کار کرده باشید مطمئناً با تخصیص حافظه پویا آشنایی دارید. در مواقعی مجبور به تخصیص حافظه به شیئی یا ساختار خود شده، در برنامه خود از این شیئی استفاده کرده، و سپس آن را از بین برده اید. هنگامی که حافظه ای را به شیئی اختصاص می دهید، خود نیز وظیفه مدیریت آن حافظه (یا حافظه ها) و بازگرداندن آن به سیستم را بر عهده دارید. عمل مدیریت حافظه در هنگام زیاد شدن این اشیاء (مثلاً در لیست های پیوندی) عملی بسیار خسته کننده و دشوار است. هنگامی که حافظه یک شیئی را می گیرید، باید دقت کنید که تمامی منابعی را که به آن دسترسی دارد نیز آزاد کنید. مسلماً این عمل بسیار خسته کننده بوده و نیاز به دقت زیادی دارد. در Java می توانید این اعمال را به garbage collector (gc) محول کنید. بدین ترتیب نیازی نیست که نگران حافظه دینامیک مصرفی برنامه خود باشید. gc به طور اتوماتیک، عمل گرفتن حافظه را از اشیاء غیر قابل دسترس انجام می دهد. شکل کلی عملکرد آن را می توان بدین گونه دانست که هنگامی که هیچ منبعی به شیئی نداشته باشیم، آن شیئی غیر قابل مصرف در نظر گرفته شده و حافظه آن آزاد شده و به سیستم برگردانده می شود. پس نیازی نیست که به طور واضح و مشخص مانند ++c، یک شیئی را از بین ببریم. gc را می توان یک thread در نظر گرفت که به طور موازی با برنامه اجرا شده و اشیاء ایجاد شده توسط آن را ردیابی و کنترل می کند و در موقع لزوم، حافظه را از آنها گرفته و به سیستم بر می گرداند.

## نحوه عملکرد garbage collector

ابداع کنندگان Java، اولین افرادی نبودند که به فکر gc برای زبان خود افتادند. در واقع می توان گفت که این عمل به دهه های پیش باز می گردد. هنگامی که سازندگان زبان های Lisp و

Small Talk متوجه شدند که گرفتن حافظه از سیستم نیز مانند تخصیص آن در هنگام برنامه نویسی، بسیار پر ارزش و مهم بوده و این عمل مهم، به علت عدم توانایی سیستم بر عهده خود برنامه نویس می باشد. از آن هنگام بود که gc ها ایجاد و کار بر روی آنها انجام شد. یکی از ساده ترین روشها در gc های اولیه، ایجاد یک شمارنده برای هر شیئی است. با هر بار اختصاص منبع (reference) به یک شیئی، یکی به آن اضافه و با هر بار گرفتن آن، یکی از شمارنده کم می کنیم. اگر شمارنده از ابتدا دارای مقدار یک باشد، هنگامی که مقدار آن صفر شد، یعنی تمامی منابع از آن گرفته شده و دیگر به آن نیازی نداریم. پس gc می تواند حافظه را از آن بگیرد. تنها کاری که باید انجام دهیم، بررسی مقدار شمارنده بعد از هر بار کاهش آن است. این روش که یکی از ساده ترین روشهاست، مشکلاتی نیز به همراه دارد. کافی است کمی به متغیر های برنامه و تعداد آنها مانند متغیرهای محلی، آرگومان های توابع، مقادیر بازگشتی توابع و ... فکر کنیم. در هر لحظه از دوران زندگی برنامه، بارها این اعمال انجام می شوند. اگر برای هر کدام از این متغیرها نیاز به یک جمع اضافی (در بهترین حالت) و یک تفریق، بررسی و گرفتن حافظه (در بدترین حالت) داشته باشیم. تصور کنید که چه تعداد عمل اضافه بر عهده برنامه خود قرار داده ایم. در واقع مهمترین عیب این روش، سرعت بسیار کم آن است. به طوری که gc های اولیه که از این روش استفاده می کردند، غیر قابل استفاده بودند. ولی خوشبختانه روش های بسیار دیگری برای برطرف کردن این مشکل ارائه شده است. یکی دیگر از مشکلاتی که در gc داریم، مشکل قسمت شدن حافظه است. فرض کنید قسمتی از حافظه سیستم، به اشیاء برنامه اختصاص داده شده اند. حال در این قسمت ما اشیائی را پاک کرده ایم و می توانیم فضای آنها را خالی فرض کنیم. مطمئناً همیشه بدین گونه نیست که این فضاهای خالی در کنار یکدیگر باشند. اکنون اگر یک شیئی با حافظه بزرگ بخواد در سیستم قرار گیرد، ممکن است یک فضای خالی به تنهایی برای آن نداشته باشیم. ولی مجموع فضاهای خالی ما، حتی برای چندین شیئی از آن نوع نیز کافی باشد. پس gc ما باید علاوه بر گرفتن حافظه از اشیاء غیر لازم، بتواند فضاهای خالی را نیز مدیریت کند. و مثلاً آن ها را در کنار یکدیگر قرار دهد. در تغییر مکان شیئی، باید توجه کنیم که تمامی منابعی که از آن شیئی داریم، آدرس خود را تغییر داده و به مکان جدید شیئی اشاره کنند.

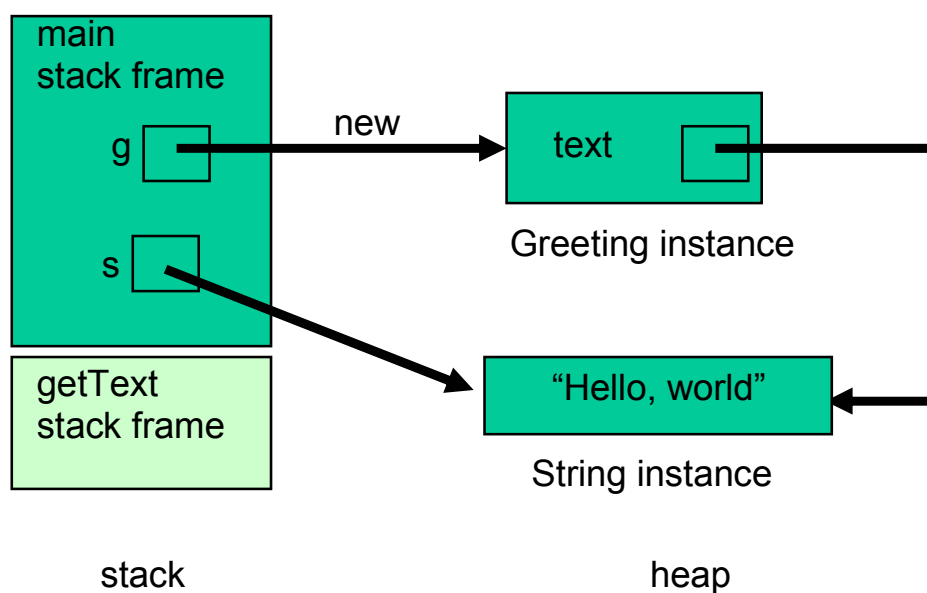
موارد ذکر شده ، تنها مواردی ساده از وظایف gc است. خوشبختانه gc در Java تمامی این موارد را در نظر گرفته و مانند یک thread موازی در هنگام اجرای برنامه، اجرا می شود و

به گونه ای که کاربر متوجه نمی شود، اشیاء و حافظه های غیر دسترس را در یک زمان کم و با تخصیص حافظه کم برای خود، جمع آوری می کند. به گونه ای که هیچ گاه متوجه حضور آن نمی شویم.

ولی اگر خود ما بخواهیم قبل از حضور gc یک شیئی را از بین ببریم تکلیف چیست؟ این عمل را می توانیم به راحتی و با null کردن آن شیئی (myobject=null) و سپس فراخوانی gc توسط متد system.gc() انجام دهیم.

### Java memory model

Implicit pointer semantics  
new, no delete (garbage collection)



# امکانات ویژه

## عناوین این بخش :

مدل زبان و کاربردهای آن

اصول ریزبرنامه ها applets

برنامه نویسی چند نخ کشی شده

Multithreaded programming

نخ اصلی The Main Thread

ایجاد يك نخ

پیاده سازی Runnable

بسط نخ

ایجاد نخ های چندگانه

استفاده از isAlive() و join()

استفاده از suspend() و resume()

تقدمهای نخ



## اصول ریز برنامه ها applets

پشتیبانی از I/O و ریز برنامه ها از هسته کتابخانه های API جاوا ناشی شده اند نه از واژه کلیدی این زبان.

کلیه مثالهای قبلی از برنامه های (applications) جاوا بودند. اما برنامه ها فقط یک کلاس از برنامه های جاوا می باشند. نوع دیگر برنامه applet یا همان ریز برنامه است. همانطوریکه قبلاً اشاره شده ، " ریز برنامه ها " ، برنامه های کوچکی هستند که روی یک سرویس دهنده اینترنت قابل دسترس بوده و سرتاسر شبکه حمل و نقل شده و بطور خودکار نصب می شوند و بعنوان بخشی از یک سند وب اجرا می شوند .

یکبار که applet روی سرویس گیرنده می رسد ، دسترسی محدود شده ای به منابع دارد بطوریکه می تواند یک رابط کاربر چند رسانه ای دلخواه را ایجاد نموده و بدون ایجاد ریسک ویروسها یا نقض تمامیت داده ها محاسبات پیچیده را اجرا نماید . اجازه دهید با ریز برنامه ساده بصورت زیر شروع کنیم :

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
    public void paint(Graphics g ){
        g.drawString("A Simple Applet"/ 20/ 20);
    }
}
```

این ریز برنامه با د و دستور import آغاز می شود. اولین دستور کلاسهای جعبه ابزار پنجره مجرد یا AWT را وارد می کند. ریز برنامه ها از طریق AWT با کاربرشان فعل و انفعال دارند نه از طریق کلاسهای I/O بر مبنای کنسول AWT . دربرگیرنده پشتیبانی برای یک رابط پنجره ای و گرافیکی است. همانطوریکه ممکن است انتظار داشته باشید ، این جعبه ابزار بسیار بزرگ و پیچیده است.

خوشبختانه این ریز برنامه ساده استفاده بسیار محدودی از AWT دارد. دستور بعدی import بسته Applet را وارد می کند. این بسته دربرگیرنده کلاس applet است. هر ریز برنامه ای که تولید می کنید باید زیر کلاسی از Applet باشد.

خط بعدی در برنامه ، کلاس Simple Applet را اعلان می کند. این کلاس باید بعنوان public اعلان شود ، چون باید توسط کدهای خارج از برنامه نیز قابل دسترس باشد . روش paint() درون SimpleApplet اعلان شده است. این روش توسط AWT تعریف شده و باید توسط ریز برنامه انباشته شود. هرگاه که ریز برنامه باید خروجی اش را دوباره نمایش دهد ، paint() فراخوانی می شود . این شرایط ممکن است به چند دلیل اتفاق بیفتد. بعنوان مثال ، روی پنجره ای که ریز برنامه داخل آن در حال اجراست ممکن است توسط پنجره دیگری نوشته شود و سپس آشکار شود . یا پنجره applet می تواند حداقل رسیده و سپس ذخیره شود . وقتی که ریز برنامه اجرا را شروع می کند ، paint() نیز فراخوانی می شود. دلیل هر چه باشد ، هر وقت ریز برنامه مجبور به نمایش مجدد خروجی خود باشد ، paint () فراخوانی می شود. روش paint() یک پارامتر از نوع Graphics دارد. این پارامتر دربرگیرنده متن گرافیکی است که محیط گرافیکی که applet در آن در حال اجراست را توصیف می کند.

هر گاه که خروجی applet مورد نیاز باشد ، این متن مورد استفاده قرار می گیرد. درون paint() یک فراخوانی به drawstring() وجود دارد که عضوی از کلاس Graphics است. این روش یک رشته شروع شده در مکانهای x و y و مشخص را خارج می کند. شکل عمومی آن بصورت زیر است:

```
Void drawstring( string message, int x, int y )
```

در اینجا message رشته ای است که در x و y و شروع شده و باید خارج شود. در یک پنجره جاوا ، گوشه سمت چپ بالایی مکان o و o و است . فراخوانی drawstring() در ریز برنامه سبب می شود پیام "A simple Applet" بنمایش درآید و در نقطه ۲۰ و ۲۰ آغاز شود. دقت کنید که ریز برنامه فاقد روش main() است . برخلاف برنامه های جاوا ، ریز برنامه ها اجرای خود را در main() شروع نمی کنند. در حقیقت ، اکثر ریز برنامه ها حتی یک روش main() هم ندارند. در عوض ، یک ریز برنامه شروع با اجرا می کند هرگاه که نام کلاس آن به یک مشاهده گر ریز برنامه (applet viewer) یا یک مرورگر شبکه گذر داده شود . بعد از اینکه کد منبع SimpleApplet را وارد کرده اید، بهمان روشی که برنامه ها را کامپایل می کنید، ریز برنامه ها را کامپایل می کنید. اما اجرای SimpleApplet شامل یک پردازش متفاوت است. در حقیقت ، دو راه برای اجرای ریز برنامه وجود دارد:

اجرانمودن applet داخل يك مرورگر وب سازگار با جاوانظير Netscape Navigator استفاده از يك مشاهده گر ريزبرنامه (applet viewer) نظير ابزار JDK استاندارد يعني appletviewer. يك مشاهده گر ريز برنامه ، ريز برنامه شما را در يك پنجره اجرا مي كند. اين روش عموما سريعترين و آسانترين راه براي آزمون ريزبرنامه ها است. اجازه دهيد هر كدام اين روشها را بررسي نماييم . براي اجراي يك applet در مرورگر وب ، لازم است يك فايل متني HTML کوتاه بنويسيد كه دربرگيرنده دنباله APPLET مناسب باشد. در اينجا فايل HTML را كه Simple Applet را اجرا مي كند ، مشاهده مي نماييد.

```
<html>
<body>
<applet code=YOURFILENAME.class width=200 height=200>
</applet>
</body>
</html>
```

دستورات width و height و مشخص كننده ناحيه شروع نمايشي هستند كه توسط ريز برنامه مورد استفاده قرار مي گيرد. دنباله APPLET دربرگيرنده چندين گزينه يگر است. بعد از اينكه اين فايل را ايجاد كرديد ، مي توانيد مرورگرتان را اجرا نموده و سپس اين فايل را بار گذاري كنيد . انجام اينكار باعث مي شود كه SimpleApplet اجرا شود . براي اجراي SimpleApplet با يك مشاهده گر ريز برنامه ، ممكن است همچنين فايل HTML قبلي را اجرا نماييد. بعنوان مثال ، اگر فايل HTML قبلي را RunApp.html بخوانيم ، آنگاه خط فرمان بعدي ، SimpleApplet را اجرا مي كند :

```
C:\>appletviewer Runnapp.html
```

اما ، يك روش بسيار راحت تر وجود دارد كه سرعت آزمائش را افزايش مي دهد. در بالاي فايل كد منبع جاواي خود يك توضيح (comment) بگنجانيد كه دربرگيرنده دنباله APPLE باشد. بدین ترتيب ، كد شما با يك الكوي دستورات HTML ضروري مستند سازي مي شود و مي توانيد ريز برنامه كامپايل شده خود را با شروع مشاهده گر ريزبرنامه با فايل كد منبع جاواي خود مورد آزمائش قرار دهيد. اگر از اين روش استفاده مي كنيد ، فايل منبع Simple Applet بقرار زير خواهد بود:

```
import java.awt.*;
import java.applet.*;

public class SimpleApplet extends Applet {
    public void paint(Graphics g ){
        g.drawString("A Simple Applet"/ 20 .20);
    }
}
```

در کل می توانید از طریق توسعه ریز برنامه با استفاده از این سه مرحله سرعت تکرار را اجرا نمایید :

1 : (iterate) يك فایل منبع جاوا ویرایش نمایید .

۲: برنامه اتان را کامپایل کنید .

۳: مشاهده گر ریزبرنامه را اجرا نموده ، تا نام فایل منبع ریزبرنامه اتان را مشخص نمایید .  
مشاهده گر ریز برنامه ، داخل توضیح با دنباله APPLET مواجه شده و ریز برنامه شما را اجرا خواهد نمود. پنجره تولید شده توسط SimpleApplet ، آنطوریکه توسط مشاهده گر زیربرنامه بنمایش درآمده بصورت زیر می باشد :

```
|
>< |< | _ |_ ...| pplet Viewer SSimp
| Applet |

| A Simple Applet |
| |
| |
| |
| Applet started .|
```

نکات اصلی که باید در این موضوع بیاد داشته باشید ، عبارتند از:

ریز برنامه ها نیازی به روش main () ندارند. ریز برنامه ها باید تحت يك مشاهده گر ریز برنامه (applet viewer) یا يك مرورگر سازگار با جاوا اجرا شوند. ارباب با کلاسهای I/o جریان جاوا اجرا نمی شود. در عوض ، ریز برنامه ها.

## برنامه نویسی چند نخ کشی شده Multithreaded programming

برخلاف سایر زبانهای کامپیوتری، جاوا، پشتیبانی توکار از برنامه نویسی چند نخ کشی شده را فراهم می کند. یک برنامه چند نخ کشی شده (multithreaded) دربرگیرنده دو یا چند بخش است که می توانند بصورت متقارن و همزمان اجرا شوند. هر بخش از چنین برنامه ای را یک نخ یا thread می نامند که هر نخ می تواند یک مسیر جداگانه از اجرا تعریف نماید. بنابراین، چند نخ کشی شده یک شکل تخصصی تر از همان چند وظیفه ای (multitasking) است. بطور حتم با مفهوم چند وظیفه ای آشنا هستید، زیرا کلیه سیستم های عامل مدرن از این موضوع پشتیبانی می کنند.

دو نوع مجزا از چند وظیفه ای وجود دارد:

بر مبنای پردازش (process-based) و بر مبنای نخ (thread-based) لازم است که تفاوت بین این دو را درک نمایید. یک پردازش از نظر مفهومی یک برنامه در حال اجرا است. بدین ترتیب، چند وظیفه ای بر مبنای پردازش به کامپیوتر شما امکان می دهد تا دو یا چند برنامه را بطور متقارن و همزمان اجرا نماید. بعنوان مثال چند وظیفه ای بر مبنای پردازش به شما امکان می دهد تا در حین اجرای کامپایلر جاوا و در همان زمان بتوانید از یک ویرایشگر متن نیز استفاده نمایید. در چند وظیفه ای بر مبنای پردازش، یک برنامه، کوچکترین واحد کدی است که توسط زمانبند (scheduler) توزیع می شود (dispatched).

در یک محیط چند وظیفه ای بر مبنای نخ، کوچکترین واحد کد قابل توزیع، همان نخ است. این بدان معنی است که یک برنامه منفرد می تواند دو یا چندین وظیفه را در آن واحد انجام دهد. بعنوان نمونه، یک ویرایشگر متن می تواند در همان زمانی که مشغول چاپ گرفتن است، به فرمت کردن یک متن نیز بپردازد، البته مادامیکه این دو عمل توسط دو نخ جداگانه اجرا شوند. بدین ترتیب، چند وظیفه ای بر مبنای پردازش با "تصاویر بزرگ" سر و کار دارد در حالیکه چند وظیفه ای بر مبنای نخ جزئیات کارها را اداره می کند نخ های چند وظیفه کننده مستلزم انباشتگی کمتری در مقایسه با پردازشهای چند وظیفه کننده هستند. پردازشها در اصل وظایف سنگینی هستند که نیازمند فضاهای آدرس جداگانه خاص خودشان می باشند. ارتباطات بین پردازشی (Interprocess) اغلب بر هزینه و محدود است چند کارگی و راه گزینی متن (context switching) از یک پردازش به پردازش دیگر نیز بر هزینه است. در عوض نخها سبک هستند.

آنها يك فضاي يكسان آدرس رابه اشتراك گذاشته و بصورت مشاركتي همان پردازش با وظيفه سنگين را به اشتراك مي گذارند. ارتباطات بين نخ ها (Interthread) ارزان است و راه گزيني متن از يك نخ به نخ ديگر كم هزينه است. اگرچه برنامه هاي جاوا از محيطهاي چند وظيفه اي بر مبناي پردازش هم استفاده مي كنند ، اما چند وظيفه اي بر مبناي پردازش تحت كنترل جاوا نيست . اما چند وظيفه اي چند نخ كشي شده تحت كنترل قرار مي گيرد . چند نخ كشي كردن به شما اجازه مي دهد تا برنامه هايي بسيار موثر و كارا بنويسيد كه حداكثر استفاده از cpu را داشته باشند ، زيرا آنها زمان خالي idle time را به حداقل ممكن كاهش مي دهند. اين امر بخصوص در محيط فعل و انفعالي و شبكه اي شده اي كه جاوا در آن كار مي كند حائز اهميت است ، زيرا در اين گونه محيط ها زمان خالي بسيار زياد است. بعنوان مثال ، نرخ انتقال داده روي يك شبكه در مقايسه با نرخ پردازش كامپيوتر بسيار كندتر است. حتي منابع فايلهاي سيستم با سرعت كم تري نسبت به آنچه cpu قادر است پردازش نمايد، خوانده و نوشته ميشوند و البته ، ورودي كاربر نيز خيلي كندتر انجام خواهد گرفت. در يك محيط سنتي تك نخي ، برنامه شما قبل از حركت بطرف وظيفه بعدي مجبور است منتظر اتمام اجراي وظايف قبلي بماند حتي اگر cpu زمان نسبتا" زيادي را بيقار باشد. چند نخ كشي به شما اجازه ميدهد تا اين زمان بيقاري را تحت كنترل گرفته و از آن بنحو مطلوبي استفاده نماييد. اگر تابلال براي سيستم هاي عامل نظير windows 95 يا windows NT برنامه نويسي کرده باشید ، پس حتما با برنامه نويسي چند نخ كشي آشنا هستيد . اما بخصوص اين حقيقت كه جاوا نخ ها را مديريت مي كند، چند نخ كشي كردن را آسان مي سازد، زيرا شوند.

### نخ اصلي The Main Thread

وقتي يك برنامه جاوا شروع مي شود ، حتما قبل از آن ، يك نخ در حال اجرا وجود دارد. اين نخ را معمولا نخ اصلي يا main thread برنامه شما مي نامند. زير وقتي برنامه شما مي خواهد شروع شود ، اين نخ اجرا شده است. نخ اصلي به دودليل بسيار مهم است. اين همان نخي است كه ساير نخ هاي فرزند (child) از آن تكثير مي شوند. اين نخ بايد آخرين نخي باشد كه اجرا را تمام مي كند. وقتي كه نخ اصلي متوقف مي شود ، برنامه شما نيز خاتمه خواهد يافت.

اگرچه هنگامیکه برنامه اتان را آغاز می کنید ، نخ اصلی بطور خودکار ایجاد می شود ، اما می توان آن را از طریق یک شی Thread کنترل نمود. برای انجام اینکار ، باید با فراخوانی روش `currentthread()` که یک عضو `public static` از `thread` است ، یک ارجاع به آن بدست آورید. شکل عمومی آن بصورت زیر می باشد:

```
staticThreadcurrentThread()
```

این روش یک ارجاع به نخ که در آن فراخوانی شده است را بر می گرداند. هربار که ارجاعی به نخ اصلی ایجاد کنید ، می توانید آن را مثل هر نخ دیگری تحت کنترل در آورید. اجازه دهید با یک مثال شروع کنیم :

```
// Controlling the main Thread.
class CurrentThreadDemo {
public static void main(String args[] ){
Thread t = Thread.CurrentThread();
System.out.println("Current thread :" + t);
// change the name of the thread
t.setName("My Thread");
System.out.println("After name change :" + t);
try {
for(int n = 5; n > 0; n-- ) {
System.out.println(n);
Thread.sleep(1000);
}
} catch( InterruptedException e ){
System.out.println("Main thread interrupted");
}
}
}
```

در این برنامه ، یک ارجاع به نخ جاری ( در این حالت ، همان نخ اصلی ) بوسیله فراخوانی `currentThread()` بدست آمده و در متغیر محلی `t` ذخیره می شود سپس برنامه اطلاعات درباره نخ را نمایش می دهد. برنامه آنگاه `etName()` را فراخوانی می کند تا نام داخلی نخ را تعویض نماید. اطلاعات درباره نخ مجدداً بنمایش در می آیند. سپس ، یک حلقه از عدد ۵ شمارش معکوس می کند و بین هر دو خط یک ثانیه مکث می کند. مکث فوق توسط روش `sleep()` انجام می شود .

آرگومان به `sleep()` مشخص کننده دوره تاخیر برحسب میلی ثانیه است. دقت کنید که بلوک `try/catch` این حلقه را احاطه کرده است. روش `sleep()` در `Thread` ممکن است یک `InterruptedException` را پرتاب نماید. اگر برخی از نخ های دیگر بخواهند در این نخ معوق شده اختلال نمایند، چنین حالتی اتفاق می افتد. این مثال اگر دچار وقفه شود، یک پیام را چاپ می کند. در یک برنامه واقعی، باید این حالت را طور دیگری اداره نمایید. خروجی تولید شده توسط این برنامه بقرار زیر می باشد:

```
Current thread :Thread[main/5/main]
After name change :Thread[My Thread/5/main]
5
4
3
2
1
```

دقت کنید که وقتی `t` بعنوان یک آرگومان به `println()` استفاده می شود، خروجی تولید می شود. این خروجی بترتیب موارد بعدی را نمایش می دهد:

نام نخ، حق تقدم آن، و نام گروه مربوطه آن. بطور پیش فرض، نام نخ اصلی `main` است. تقدم آن ۵ است که مقداری پیش فرض می باشد، همچنین نام گروهی از نخ ها که این نخدان متعلق است، همان `main` می باشد. یک گروه نخ `Threadgroup` یک نوع ساختار داده است که حالت یک مجموعه از نخ ها را بطور کلی کنترل می کند. این پردازش محیط حین اجرای خاصی مدیریت شده و در اینجا مورد بررسی قرار نمی گیرد. بعد از اینکه نام نخ تغییر می یابد، `t` مجدداً حاصل می شود. در این زمان، نام جدید نخ بنمایش درمی آید. اجازه دهید نگاهی دقیق تر به روشهای تعریف شده توسط `Thread` که در برنامه استفاده شده اند، داشته باشیم. روش `sleep()` سبب میشود تا نخ که از آن فراخوانی شده، اجرا را برای مدت مشخصی از میلی ثانیه بطور موقت متوقف نماید. شکل عمومی آن بصورت زیر است:

```
static void sleep( long milliseconds )Throws InterruptedException
```

تعداد میلی ثانیه هایی که باید تعلیق انجام گیرد بر حسب میلی ثانیه مشخص می شود. این روش ممکن است یک `InterruptedException` را پرتاب نماید. روش `sleep()` یک شکل دوم هم دارد



که بعداً نشان می‌دهیم و به شما اجازه می‌دهد تا مدت زمان را بر حسب میلی ثانیه و nanoseconds مشخص نماید.

```
static void sleep( long milliseconds/ int nanoseconds )Throws Interrupt  
Exception
```

این شکل دوم فقط برای محیط‌هایی مناسب است که امکان زمانبندی دوره های زمانی را بر حسب nanoseconds دارند. همانطوریکه برنامه قبلی نشان می دهد ، می توانید با استفاده از setName() نام یک نخ را تعیین کنید. با فراخوانی getName() می توانید نام یک نخ را بدست آورید ( اما توجه کنید که این رویه در برنامه نشان داده نشده است ). این روشها اعضا کلاس Thread هستند و بصورت زیر اعلان می شوند:

```
final void setName( string ThreadName)  
final string getName
```

### ایجاد یک نخ

زبان خیلی ساده ، شما با نمونه سازی یک شی از نوع Thread می توانید یک نخ را بوجود آورید . جاوا دو شیوه برای انجام اینکار تعریف مینماید : می توانید رابط Runnable را پیاده سازی نمایید. می توانید خود کلاس Thread را بسط دهید. هر کدام از شیوه های فوق را بررسی می کنیم.

### پیاده سازی Runnable

آسانترین شیوه ایجاد یک نخ ایجاد یک کلاس است که رابط Runnable را پیاده سازی نماید . Runnable یک واجد از کد اجرایی را مجرد می کند. می توانید روی هر شیئی که Runnable را پیاده سازی می نماید ، یک نخ بسازید. برای پیاده سازی Runnable یک کلاس فقط لازم است یک روش تکی موسوم به run() را که بصورت زیر اعلان شده پیاده سازی نماید:

```
public abstract void run()
```

كدي كه نخ جديد را مي سازيد را داخل run() تعريف نماييد. مهم است بدانيد كه run() مي تواند ساير روشها را فراخواني كند ، همچنين از ساير كلاسها استفاده نمايد و متغيرهايي درست مثل نخ اصلي را اعلان نمايد . تنها تفاوت در اين است كه run () نقطه ورودي براي نخ ديگر همزمان اجرا داخل برنامه شما را تثبيت مي كند. اين نخ وقت run() برمي گردد ، پايان مي گيرد. بعد از اينكه يك كلاس كه Runnable را پياده سازي مي كند، ايجاد نموديد، بايد يك شي از نوع Thread از داخل همان كلاس نمونه سازي كنيد Thread . چندين سازنده را تعريف مي كند . يكي از آنها كه مورد استفاده ما قرار گرفته بصورت زير است:

```
Thread( Runnable threadOb/ string threadName)
```

در اين سازنده ، threadOb يك نمونه از كلاس است كه رابط Runnable را پياده سازي مي كند . اين سازنده ، جايي را كه نخ شروع خواهد شد ، تعريف مي كند. نام نخ جديد بوسيله ThreadName مشخص مي شود. نخ جديد پس از ايجاد شدن تا زمانيكه روش start() آن را كه داخل Thread اعلان شده ، فراخواني نكنيد ، شروع با اجرا نمي كند. از نظر ذاتي ، start() يك فراخواني به run() را اجرا مي كند. روش start() را در زير نشان داده ايم :

```
synchronized void start()
```

در اينجا مثالي وجود دارد كه يك نخ جديد ايجاد نموده و اجراي آن را شروع مي كند:

```
// Create a second thread.
class NewThread implements Runnable {
    Thread t;
    NewThread () {
        // Create a new/ second thread
        t = new Thread(this/ "Demo Thread");
        System.out.println("Child thread :" + t);
        t.start(); // Start the thread
    }
    // This is the entry point for the second thread.
    public void run () {
        try {
            for(int i = 5; i > 0; i )--{
                System.out.println("Child Thread :" + i);
```

```

Thread.sleep(500);
}
} catch( InterruptedException e ){
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}

class ThreadDemo {
public static void main(String args[] ){
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i )--{
System.out.println("Main Thread : " + i);
Thread.sleep(1000);
}
} catch( InterruptedException e ){
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}

```

درون سازنده New Thread يك شي Thread جديد با استفاده از دستور بعدي ايجاد شده است:

```
t = new Thread(this/ "Demo Thread");
```

گذر دادن this بعنوان اولين آرگومان نشان مي دهد كه شما مي خواهيد نخ جديد روش run() را روي شي this فراخواني نمايد. سپس start() فراخواني مي شود ، كه نخ اجراي شروع كار را در روش run() آغاز مي كند. اين باعث مي شود كه حلقه for نخ فرزند آغاز شود . بعد از فراخواني start() ، سازنده مربوط به New Thread به main() برمي گردد. وقتي كه نفاصلي از سر گرفته مي شود ، حلقه for خود را وارد مي كند. هر دو نخ اجرا را ادامه مي دهند ، cpu را با شتراك گذاشته تا اينكه حلقه هایشان پايان گيرند. خروجي توليد شده توسط اين برنامه بقرار زير است :

```

Child thread :Thread[Demo Thread/5/main]
Main Thread :5

```

```

Child Thread :5
Child Thread :4
Main Thread :4
Child Thread :3
Child Thread :2
Main Thread :3
Child Thread :1
Exiting child thread.
Main Thread :2
Main Thread :1
Main thread exiting.

```

همانطوریکه قبلاً گفتیم ، در يك برنامه چند نخ کشی شده ، نخ اصلي بايد آخرين نخي باشد که اجرا را پايان مي دهد . اگر نخ اصلي قبل از اینکه يك نخ فرزند کامل شود ، پايان گیرد ، آنگاه ممکن است سیستم حين اجراي جاوا بحالت "hang" درآيد. برنامه قبلي اطمینان مي دهد که نخ اصلي آخرين نخي است که پايان مي گیرد زیرا نخ اصلي براي ۱۰۰۰ ميلي ثانيه بين تکرارها معوق مي ماند در حالیکه نخ فرزند فقط ۵۰۰ ميلي ثانيه معوق مي ماند. اين باعث مي شود که نخ فرزند زودتر از نخ اصلي پايان گیرد. خلاصه ، راه بهتري براي اطمینان از اینکه نخ اصلي آخر از همه پايان گیرد خواهید يافت.

## بسط نخ

دومین شیوه ایجاد يك نخ ، ایجاد يك کلاس جديد است که Thread را بسط داده و سپس يك نمونه از همان کلاس ایجاد مي کند . کلاس بسط دهنده بايد روش run() را لغو نمايد ، که نقطه مدخل (entry) براي نخ جديد است. اين کلاس همچنين بايد start() را فراخواني کند تا اجراي نخ جديد را آغاز نمايد . در اینجا برنامه قبلي را دوباره برنامه نویسی نموده ایم تا Thread را بسط دهد .

```

// Create a second thread by extending Thread.
class NewThread extends Thread {
    NewThread () {
        // Create a new/ second thread
        super("Demo Thread");
    }
}

```

```

System.out.println("Child thread :" + this);
start(); // Start the thread
}
// This is the entry point for the second thread.
public void run () {
    try {
        for(int i = 5; i > 0; i )--{
            System.out.println("Child Thread :" + i);
            Thread.sleep(500);
        }
    } catch( InterruptedException e ){
        System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
}
}

class ExtendThread {
    public static void main(String args[] ){
        new NewThread(); // create a new thread
        try {
            for(int i = 5; i > 0; i )--{
                System.out.println("Main Thread :" + i);
                Thread.sleep(1000);
            }
        } catch( InterruptedException e ){
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}

```

این برنامه همان خروجی برنامه قبلی را تولید می کند. همانطوریکه می بینید نخ فرزند بوسیله نمونه سازی یک شی NewThread که از Thread مشتق شده ، ایجاد می شود. به فراخوانی super () داخل NewThread دقت نمایید. این امر شکل بعدی سازنده Trhead را فعال می کند

```

public thread( string threadName)

```

در اینجا Thread Name ، نام نخ را مشخص می کند. انتخاب يك شیوه ممکن است تعجب کنید که چرا جاوا دو شیوه برای ایجاد نخ فرزند دارد و اینکه کدام شیوه بهتر است. کلاس Thread چندین روش را تعریف می کند که بوسیله يك کلاس مشتق شده می توانند لغو شوند. از این روشها ، آنکه باید لغو شود فقط run() است. این روش البته همان روشی است که وقتی شما Runnable را پیاده سازی میکنید لازم است. بسیاری از برنامه نویسان جاوا احساس می کنند که آن کلاسها فقط وقتی در حال افزایش یا اصلاح شدن هستند ، باید بسط یابند. بنابراین اگر نمی خواهید هیچیک از سایر روشهای thread را لغو نمایید ، احتمالاً بهتر است خیلی ساده بستگی دارد.

### ایجاد نخ های چندگانه

تا بحال فقط دو نخ را استفاده نموده اید: نخ اصلی و نخ فرزند. اما ، برنامه شما می تواند به تعداد مورد نیاز از نخ ها تکثیر نماید . بعنوان مثال ، برنامه بعدی سه نخ فرزند ایجاد می کند :

```
// Create multiple threads.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    NewThread(String threadname ){
        name = threadname;
        t = new Thread(this/ name);
        System.out.println("New thread :" + t);
        t.start(); // Start the thread
    }
    // This is the entry point for thread.
    public void run () {
        try {
            for(int i = 5; i > 0; i --){
                System.out.println(name + " :" + i);
                Thread.sleep(1000);
            }
        } catch( InterruptedException e ){
            System.out.println(name + " Interrupted.");
        }
    }
}
```

```

System.out.println(name + " exiting.");
}
}
class MultiThreadDemo {
public static void main(String args[] ){
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
try {
// wait for other threads to end
Thread.sleep(10000);
} catch( InterruptedException e ){
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}

```

خروجي حاصل از اين برنامه بقرار زير مي باشد :

```

New thread :Thread[One/5/main]
New thread :Thread[Two/5/main]
New thread :Thread[Three/5/main]
One :5
Two :5
Three :5
One :4
Two :4
Three :4
One :3
Three :3
Two :3
One :2
Three :2
Two :2
One :1
Three :1
Two :1

```

```
One exiting.  
Two exiting.  
Three exiting.  
Main thread exiting.
```

همانطوریکه می بینید ، هر بار که آغاز شوند ، هر سه نخ فرزند ، cpu را به اشتراک می گذارند به فراخوانی sleep ۱۰۰۰۰ در main ر دقت نمایید. این باعث می شود که نخ اصلی برای ۱۰ ثانیه معوق مانده و اطمینان می دهد که آخر از همه یان می یابد.

### استفاده از isAlive() و join()

همانطوریکه ذکر شد ، نخ اصلی باید آخرین نخي باشد که پایان می گیرد. در مثالهای قبلی ، اینکار را با فراخوانی sleep() داخل main() با يك تاخير بحد كافي طولاني براي اطمینان از اینکه کلیه نخ های فرزند قبل از نخ اصلی پایان می گیرند ، انجام دادیم . البته این بك راه حل بسختي قانع کننده است. این راه حل در ضمن يك سوال را برمي انگیزد : چگونه يك نخ می تواند از پایان گرفتن نخ دیگر آگاهی یابد ، خوشبختانه Thread وسیله ای فراهم نموده که توسط آن می توانید پاسخ این پرسش را بدهید. دو شیوه وجود دارد تا تعیین کنیم که آیا يك نخ پایان گرفته است یا نه . اول می توانید isAlive() را روی نخ فراخوانی کنید. این روش توسط Thread تعریف شده و شکل کلی آن بصورت زیر می باشد :

```
final boolean isAlive (Throws InterruptedException)
```

اگر نخ بالایی آنکه فراخوانی شده همچنان در حال اجرا باشد ، روش isAlive() مقدار true را برمي گرداند . در غیر اینصورت false را برمي گرداند . در حالیکه isAlive() گهگاه سودمند است ، روشی که بطور رایج مورد استفاده قرار می گیرد تا برای پایان یافتن يك نخ منتظر بمانید ، join() است که بصورت زیر می باشد :

```
final void join (throws InterruptedException)
```



این روش منتظر می ماند تا نخ‌هایی که روی آن فراخوانی شده پایان گیرند. شکلهای دیگری از `join()` وجود دارند که به شما اجازه می دهند تا همچنین حداکثر زمانی که می خواهید برای پایان یافتن یک نخ خاص صبر کنید را تعیین نمایید .

در اینجا یک روایت اصلاح شده از مثال قبلی وجود دارد که از `join()` استفاده کرده تا اطمینان دهد که نخ اصلی آخرین نخ‌هایی است که متوقف می شود . این برنامه همچنین روش `isAlive()` را نشان می دهد .

```
// Using join () (to wait for threads to finish.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    NewThread(String threadname ){
        name = threadname;
        t = new Thread(this/ name);
        System.out.println("New thread : " + t);
        t.start(); // Start the thread
    }
    // This is the entry point for thread.
    public void run () {
        try {
            for(int i = 5; i > 0; i-- ) {
                System.out.println(name + " : " + i);
                Thread.sleep(1000);
            }
        } catch( InterruptedException e ){
            System.out.println(name + " Interrupted.");
        }
        System.out.println(name + " exiting.");
    }
}

class DemoJoin {
    public static void main(String args[] ){
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        NewThread ob3 = new NewThread("Three");
        System.out.println("Thread One is alive : " + ob1.t.isAlive());
    }
}
```

```

System.out.println("Thread Two is alive :" + ob2.t.isAlive));
System.out.println("Thread Three is alive :" + ob3.t.isAlive));
// wait for threads to finish
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch( InterruptedException e ){
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive :" + ob1.t.isAlive));
System.out.println("Thread Two is alive :" + ob2.t.isAlive));
System.out.println("Thread Three is alive :" + ob3.t.isAlive));
System.out.println("Main thread exiting.");
}
}

```

خروجي حاصل از اين برنامه بقرار زیر مي باشد :

```

New thread :Thread[One/5/main]
New thread :Thread[Two/5/main]
New thread :Thread[Three/5/main]
Thread One is alive :true
Thread Two is alive :true
Thread Three is alive :true
One :5
Two :5
Three :5
One :4
Two :4
Three :4
One :3
Two :3
Three :3
One :2
Two :2
Three :2

```

```

One :1
Two :1
Three :1
One exiting.
Two exiting.
Three exiting.
One exiting.
Thread One is alive :false
Thread Two is alive :false
Thread Three is alive :false
Main thread exiting.

```

### استفاده از suspend() و resume()

گاهی لازم است اجرای یک نخ را لغو نماییم . بعنوان مثال با استفاده از یک نخ جداگانه می توان وقت را نشان داد. اگر کاربر تمایلی به استفاده از ساعت نداشته باشد ، پس نخ مربوط به آن باید لغو شود. موضوع آن هر چه باشد ، لغو نمودن یک نخ کار ساده ای است. همچنین بکار انداختن مجدد یک نخ لغو شده نیز کار ساده ای است. روشهایی که ایندو وظیفه را انجام میدهند عبارتند از resume() و suspend() . آنها توسط Thread تعریف شده و بصورت زیر می باشند :

```

final void resume()

final void suspend()

```

برنامه بعدی این روشها را نشان می دهد :

```

// Using suspend ()and resume.()

class NewThread implements Runnable {}

String name; // name of thread
Thread t;
NewThread(String threadname ){
    name = threadname;
    t = new Thread(this/ name);
    System.out.println("New thread :" + t);
}

```

```

t.start()// Start the thread
}
// This is the entry point for thread.
public void run (){

    try {
        for(int i = 5; i > 0; i )--{
            System.out.println(name + " :" + i);
            Thread.sleep(200);
        }
        } catch( InterruptedException e ){
            System.out.println(name + " Interrupted .");
        }
        System.out.println(name + " exiting.");
    }
}

class SuspendResume {
    public static void main(String args[] ){
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        try {
            Thread.sleep(1000);
            ob1.t.suspend) (
            System.out.println("Suspending thread One");
            Thread.sleep(1000);
            ob1.t.resume) (
            System.out.println("Resuming thread One");
            ob2.t.suspend) (
            System.out.println("Suspending thread Two");
            Thread.sleep(1000);
            ob2.t.resume) (
            System.out.println("Resuming thread Two");
        } catch( InterruptedException e ){
            System.out.println("Main thread Interrupted");
        }
        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");

```

```

ob1.t.join();
ob2.t.join();
} catch( InterruptedException e ){
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}

```

این برنامه خروجی بعدی را تولید می کند :

```

New thread :Thread[Two/5/main]
One :15
New thread :Thread[Three/5/main]
Two :15
One :14
Two :14
One :13
Two :13
One :12
Two :12
One :11
Two :11
Suspending thread One
Two :10
Two :9
Two :8
Two :7
Two :6
Resuming thread One
Suspending thread Two
One :10
One :9
One :8
One :7
One :6
Resuming thread Two
Waiting for threads to finish.

```

```
Two :5
One :5
Two :4
One :4
Two :3
One :3
Two :2
One :2
Two :1
One :1
Two exiting.
One exiting.
Main thread exiting.
```

## تقدمهاي نخ

تقدمهاي نخ توسط زمانبند نخ استفاده مي شود تا مشخص شود كه کدام نخ بايد اجازه اجرا پيدا نمايد . از نظر تئوري ، نخهاي داراي تقدم بيشتر نسبت به نخهاي داراي تقدم كمتر ، زمان بيشتري از cpu را مي گيرند . در عمل ، ميزان وقتي كه يك نخ از cpu مي گيرد ، علاوه بر تقدم بستگي به عوامل ديگري هم دارد ( . بعنوان مثال ، اينكه چگونه يك سيستم عامل چند وظيفه اي را پياده سازي مي كند مي تواند روي دسترسي نسبي به زمان cpu تاثير داشته باشد . ) يك نخ داراي تقدم بيشتر مي تواند از يك نخ با تقدم كمتر پيشدستي نمايد. بعنوان نمونه ، وقتي يك نخ با تقدم كمتر در حال اجرا باشد و يك نخ با تقدم بيشتر از سر گرفته شود ( مثلا از حالت تعليق يا انتظار روي ( I/o اين نخ با تقدم بيشتر به نخ با تقدم كمتر پيشدستي مي كند.

در تئوري ، نخ هاي با تقدم برابر بايد دسترسي معادلي به cpu داشته باشند . اما لازم است مراقب باشيد . بياد داشته باشيد كه جاوا براي كار در طيف وسيعي از محيطها طراحي شده است . برخي از محيطها ، چند وظيفه اي را كاملا متفاوت از ساير محيطها ، پياده سازي مي كنند . بخاطر ايمني ، نخ هايي كه تقدم يكساني را به اشتراك مي گذارند بايد هر چند گاه يكبار كنترل شوند . اين امر اطمينان مي دهد كه كليخ نخ ها يك فرصت براي اجرا شدن تحت سيستم عامل غير وابسته به پيشدستي (non-preemptive) را خواهند داشت. در عمل ، حتي در محيطهاي غير وابسته به پيشدستي ، اكثر نخ ها همچنان يك شانس اجرا شدن دارند ، زيرا اكثر نخ ها بناچار با برخي

شرایط بلوک سازی نظیر انتظار برای I/O مواجه خواهند شد. وقتی این اتفاق می افتد، نخ بلوک شده لغو شده و سایر نخ ها می توانند اجرا شوند. اما اگر می خواهید اجرای چند نخي شده را بنرمي انجام دهید، نباید روی این اصل متكي باشید. همچنین، برخی انواع وظایف وجود دارند که گرایش به cpu دارند چنان نخ هايي بر cpu چیره خواهند شد. روی این نوع از نخ ها، باید گهگاه كنترلي داشته باشید تا سایر نخ ها بتوانند اجرا شوند.

برای تعیین تقدم يك نخ، از روش `setpriority()` استفاده نمایید، که عضوي از Thread است. شکل عمومي آن بقرار زیر است:

```
final void setpriority( in level)
```

در اینجا level توصیفگر تعیین تقدم جديد برای فراخواننده است. مقدار level باید داخل محدوده MIN-PRIORITY و MAX-PRIORITY باشد. در حال حاضر، این مقادیر 1 و ۱۰ میباشند. برای برگرداندن يك نخ به تقدم پيش فرض، NORM-PRIORITY را مشخص می کنید که فعلا 5 است. این تقدمها بعنوان متغيرهاي final داخل Thread تعریف شده اند. می توانید تعیین تقدم جاري را با فراخواني روش `getpriority()` در Thread ر بدست آورید، که بصورت زیر می باشد:

```
final int getpriority()
```

فعلا پیاده سازی جاوا، وقتی که زمانبندي پيش می آید، بصورت شدیدی رفتار متفاوتي نشان می دهد. روایت ویندوز ۹۵ کمابیش همانطوریکه انتظار دارید کار می کند. اما روایت solaris چگونه اي متفاوت کار می کند. بسیاری از ناسازگاریها هنگامی بروز می کنند که شما نخ هايي داشته باشید که بجای اینکه زمان cpu را بصورت اشتراكي مصرف نمایند، متكي به رفتار وابسته به پيش دستي باشند. برای کسب رفتار ارجاع متقابل قابل پيش بيني با جاواي امروز، باید از نخهايي استفاده کنید که بطور اختياري از كنترل نمودن cpu دست برمي دارند.

مثال بعدي دو نخ با تقدمهاي مختلف را نشان می دهد، که روی دو محیط زیربنايي ذکر شده اجرا خواهند شد. يك نخ دو سطح بالاتر از تقدم معمولي تعیین شده که بوسیله Thread-NORM-PRIORITY تعریف شده است و دیگری دو سطح پایین تر از تقدم معمولي تعیین شده است. نخ ها آغاز شده و بمدت ۱۰ ثانیه اجازه اجرا دارند. هر نخ يك حلقه را اجرا می کند، که تعداد

تکرارها را شمارش می کند. بعد از ده ثانیه ، نخ اصلی هر دو نخ را متوقف می کند. سپس تعداد دفعاتی که هر نخ از طریق حلقه ساخته شده ، بنمایش در می آید.

```
// Demonstrate thread priorities.
class clicker implements Runnable {
    int click = 0;
    Thread t;
    private boolean running = true;
    public clicker(int p ){
        t = new Thread(this);
        t.setPriority(p);
    }
    public void run (){
        while( running ){
            click++;
        }
    }
    public void stop (){
        running = false;
    }
    public void start (){
        t.start();
    }
}

class HiLoPri {
    public static void main(String args[] ){
        Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
        clicker hi = new clicker(Thread>NORM_PRIORITY + 2);
        clicker lo = new clicker(Thread>NORM_PRIORITY - 2);
        lo.start();
        hi.start();
        try {
            Thread.sleep(10000);
        } catch( InterruptedException e ){
            System.out.println("Main thread interrupted.");
        }
        lo.start();
    }
}
```



```

hi.start();
// Wait for child threads to terminate.
try {
hi.t.join();
lo.t.join();
} catch( InterruptedException e ){
System.out.println("InterruptedException caught");
}
System.out.println("Low-priority thread :" + lo.click);
System.out.println("High-priority thread :" + hi.click);
}
}

```

خروجي اين برنامه كه بعدا نشان داده ايم هنگاميكه تحت ويندوز ۹۵ اجرا ميشود نشان مي دهد كه نخ ها context switch را انجام داده اند ، حتي اگر هيچكدامشان بطور اختياري از cpu دست نكشند و يا براي I/O بلوكه نشوند .

```
Low-priority thread :434104
```

```
High-priority thread :4860791
```

البته خروجي دقيق توليد شده توسط اين برنامه به سرعت cpu شما و تعداد ساير وظايف در حال اجرا در سيستم بستگي خواهد داشت.

خروجي بعدي از همان كلاس جاوا ، كه اينبار روي يك ماشين solaris اجرا مي شود نشان مي دهد كه نخ با تقدم بيشتر صد در صد زمان اجرا شده است . اين بلحاظ آن است كه نخ با تقدم بيشتر بر cpu مسلط شده است .

```
Low-priority thread :0
```

```
High-priority thread :3062507
```

در آينده اي نزديك ، بسيار مهم است كه شما كدهايي كه بستگي به رفتار براساس پيش دستي در ويندوز ۹۵ يا هر نوع سيستم عامل ديگري داشته باشند، ننويسيد .

## استفاده از چند نخ کشی کردن

اگر مثل سایر برنامه نویسان باشید ، پس در اختیار داشتن پشتیبانی توکار از چند نخ کشی شدن در زبان برنامه نویسی برای شما تازگی خواهد داشت. کلید استفاده موثر از این پشتیبانی این است که بصورت " همزمانی " تفکر نمایید نه بصورت سریالی. بعنوان مثال ، وقتی دو زیر سیستم داخل یک برنامه دارید که می توانند همزمان اجرا شوند ، آنها را به نخ های منفرد تقسیم نمایید. با استفاده محافظه کارانه از چند نخ کشی کردن ، می توانید برنامه های بسیار موثری بنویسید. اما اگر تعداد زیادی از نخ ها ایجاد نمایید، سبب افت عملکرد برنامه اتان خواهید شد. بیاد داشته باشید که برخی بالاسریها با conext switching همراه هستند. اگر تعداد زیادی از نخ ها ایجاد کنید، آنگاه وقت cpu بیشتر از آنکه برای اجرای اص یابد ، صرف تغییر context ها خواهد شد.

## پيوست

در اين بخش شما ، گام به گام آنچه را در فصل هاي پيشين آموخته ايد به صورت عملي به كار خواهيد گرفت. سعي شده است با ارائه مثال هايي مناسب زيبايي جاوا را درك كنيد.

موفق باشيد

برنامه ای که قصد توضیح آن را داریم ، بسیار ساده و مختصر است . برنامه فوق محیط لازم برای رسم یک خط قطری را ایجاد می نماید. بدین منظور عملیات زیر را می بایست انجام داد :

- برنامه Notepad را فعال و برنامه مورد نظر را در آن تایپ نمائید.
- برنامه را ذخیره نمائید.
- برنامه نوشته شده را با استفاده از کمپایلر جاوا ترجمه تا یک اپلت جاوا ایجاد گردد.
- در صورت گزارش خطاء ، نسبت به رفع آنها اقدام گردد.
- یک صفحه وب Html ایجاد و از اپلت ایجاد شده در آن استفاده نمائید.
- اپلت جاوا را اجرا نمائید.

متن برنامه اشاره شده بصورت زیر است :

متن برنامه
<pre>import java.awt.Graphics;  public class FirstApplet extends java.applet.Applet {      public void paint(Graphics g)     {         g.drawLine(0, 0, 200, 200);     } }</pre>

#### مرحله یک : تایپ برنامه

بمنظور ذخیره نمودن برنامه ، فولدری با نام دلخواه ایجاد تا برنامه در آن ذخیره گردد. در ادامه ویرایشگر Notepad ( و یا هر ادیتور متنی دیگری که قادر به ایجاد فایل های با انشعاب TXT باشد ) را فعال و برنامه فوق را تایپ ( و یا Copy و Paste ) نمائید. در زمان تایپ برنامه فوق می بایست در رابطه با حروف بزرگ و کوچک دقت لازم صورت پذیرد. در این رابطه لازم است که حروف بزرگ و کوچک دقیقا مشابه جدول فوق ، تایپ گردند.

#### مرحله دوم : ذخیره کردن فایل

برنامه تایپ شده را با نام فایل FirstApplet.Java در فولدری که در مرحله یک ایجاد کرده اید ، ذخیره نمائید. نسبت به استفاده از حروف بزرگ و کوچک در نام فایل دقت گردد چراکه در آینده فایل با همین نام مورد دستیابی قرار خواهد گرفت .

**مرحله سوم :** کمپایل برنامه

پنجره MS-DOS را فعال و با استفاده از دستور CD ، در فولدری که فایل FirstApplet.java قرار دارد ، مستقر شده و دستور زیر را بمنظور ترجمه برنامه نوشته شده ، تایپ نمائید :

```
javac FirstApplet.java
```

نام فایل حاوی برنامه را بدرستی تایپ نمائید ( دقت لازم در رابطه با حروف بزرگ و کوچک )  
**مرحله چهارم :** تصحیح و برطرف کردن خطاء ، در صورت وجود خطاء ، می بایست نسبت به رفع اشکالات موجود اقدام کرد.

**مرحله پنجم :** ایجاد یک صفحه Html ، بمنظور نگهداری و استفاده از اپلت ایجاد شده ، یک صفحه وب ایجاد و اطلاعات زیر را در آن قرار دهید :

فایل Html
<html> <body> <applet code=FirstApplet.class width=200 height=200> </applet> </body> </html>

فایل فوق را با نام applet.htm و در فولدری با نام مشابه ذخیره نمائید.

**مرحله ششم :** اجرای اپلت ، پنجره MS-DOS را فعال و دستور زیر را بمنظور اجرای اپلت تایپ نمائید :

```
appletviewer applet.htm
```

پس از اجرای اپلت ، یک خط قطری از گوشه بالای سمت چپ بسمت گوشه پائین سمت راست را مشاهده خواهید کرد. بدین ترتیب اولین برنامه جاوا نوشته و اجراء گردید.

**توضیحات و تشریح برنامه**

برنامه نوشته شده یک اپلت ساده جاوا است. اپلت ، نوع خاصی از برنامه های جاوا بوده که می توان آنها را در یک مرورگر اجراء کرد. اپلت های جاوا در مقابل برنامه های کاربردی جاوا مطرح شده اند. برنامه های کاربردی جاوا ، برنامه هایی بوده که می توان آنها را بر روی یک ماشین محلی اجراء نمود. برای کمپایل نمودن اپلت از برنامه javac استفاده شده است. در ادامه بمنظور نگهداری اپلت و فراهم نمودن محیط لازم برای اجرای آن ، یک صفحه وب ایجاد و اپلت در صفحه فوق صدا زده شده است. برای اجرای یک اپلت می توان از برنامه appletviewer نیز استفاده کرد.

برنامه نوشته شده صرفا دارای ده خط برنامه است. برنامه فوق ساده ترین نوع اپلتی است که می توان ایجاد کرد. بمنظور شناخت کامل عملکرد برنامه فوق ، لازم است با تکنیک های برنامه نویسی شی گراء آشنائی لازم وجود داشته باشد. بدین منظور بر روی یکی از خطوط برنامه متمرکز و عملکرد آن توضیح داده می شود :

```
g.drawLine(0, 0, 200, 200);
```

خط فوق مسئول انجام عملیات مورد نظر در برنامه است. دستور فوق ، خط قطری را رسم خواهد کرد. سایر خطوط برنامه در ارتباط با خط اصلی فوق می باشند. با دستور فوق به کامپیوتر گفته شده است که ، خطی را از گوشه سمت چپ بالا ( مختصات صفر و صفر ) به گوشه سمت راست پائین ( مختصات ۲۰۰ و ۲۰۰ ) رسم کند.

در صفحه وب ، اندازه پنجره مربوط به اجراء و نمایش اپلت ( در مرحله پنج ) به ابعاد ۲۰۰ و ۲۰۰ مشخص شده است. در برنامه فوق از متدی ( تابع ) با نام drawLine استفاده شده است. متد فوق ، چهار پارامتر را بعنوان ورودی اخذ می نماید ( ۰،۰،۲۰۰،۲۰۰ ). انتهای خط با استفاده از کاراکتر ";" مشخص شده است. نقش کاراکتر فوق نظیر استفاده از نقطه در انتهای جملات است. ابتدای خط با حرف g شروع شده است. بدین ترتیب مشخص شده است که قصد فراخوانی متدی با نام drawLine با نام شی g وجود دارد.

یک متد ، نظیر یک دستور است. متد ها به کامپیوتر اعلام می نمایند که می بایست یک کار خاص انجام گیرد. drawLine ، به کامپیوتر اعلام می نماید که ، خطی افقی با مختصات مشخص شده را رسم نماید. با تغییر مختصات مربوطه ( پارامترهای متد drawLine ) می توان خطوط متعدد و با استفاده از مختصات مشخص شده را رسم نمود.

از چه توابع دیگری بجز drawLine می توان استفاده کرد ؟ بدین منظور لازم است که به مستندات مربوط به کلاس Graphice مراجعه گردد. در زمان نصب محیط پیاده سازی جاوا و مستندات مربوطه ، یکی از فایل هائی که بر روی سیستم شما نصب خواهد شد ، فایل `java.awt.Graphics.html` است . فایل فوق کلاس Graphic را تشریح می نماید. drawLine صرفاً یکی از متدهای کلاس Graphic بوده و در این زمینه متدهای متعدد دیگر بمنظور رسم خطوط ، کمان ، چند ضلعی، تغییر و ... وجود دارد.

جاوا دارای کلاس های متعدد بوده و هر کلاس نیز دارای متدهای فراوانی است . مثلاً کلاس Color دارای مجموعه ای از متدها بمنظور تعریف و تنظیمات مربوط به رنگ است. SetColor. نمونه ای در این زمینه است . در زمان استفاده هاز هر یک از متدهای مربوط به کلاس های جاوا می بایست در ابتدای برنامه با استفاده از دستور import زمینه استفاده از آنان را فراهم کرد.

## اشکال زدائی

در زمان نوشتن برنامه های کامپیوتری ، ممکن است به خطاهای متفاوت برخورد نمایم . خطاهای برنامه نویسی دارای انواع متفاوتی نظیر : خطای گرامری ، خطای زمان اجراء و خطای منطقی می باشند. تمام خطاهای فوق صرفنظر از ماهیت مربوطه را ، اشکال (Bugs) گفته و عملیات مربوط به برطرف کردن اشکال را اشکال زدائی (debugging) می گویند. اشکال زدائی برنامه های کامپیوتری همواره زمان زیادی از وقت برنامه نویسان را بخود اختصاص خواهد داد.

در زمان نوشتن یک برنامه در صورتیکه مجموعه قوانین موجود در رابطه با زبان برنامه نویسی رعایت نگردد ( مثلاً عدم استفاده از کاراکتر ";" در انتهای جملات در جاوا ) ، کمپایلر در زمان ترجمه برنامه ، یک خطای گرامری را تشخیص و اعلام می نماید. در چنین مواردی می بایست قبل از هر اقدام دیگر ، نسبت به برطرف نمودن اشکال گزارش داده شده ، اقدام کرد. پس از ترجمه موفقیت آمیز یک برنامه ( عدم وجود خطای گرامری ) ، برنامه اجراء می گردد. در زمان اجرای یک برنامه ممکن است با نوع دیگری از خطاء مواجه گردیم . خطاهای فوق را ، خطای زمان اجراء می نامند. در صورتیکه برنامه دارای خطای زمان اجراء نباشد و بطور کامل اجراء گردد ، ممکن است خروجی تولید شده توسط برنامه متناسب با خواسته تعریف شده نباشد. خطاهای

فوق را خطاهای منطقی گویند و به علت عدم استفاده درست از دستورات و یا استفاده نامناسب از الگوریتم ها در یک برنامه بوجود می آیند. در چنین مواردی لازم است برنامه نویس ، برنامه نوشته شده را مجدداً بازبینی نموده و با دنبال نمودن بخش های مربوطه و در صورت لزوم الگوریتم های استفاده شده ، خطای موجود را تشخیص و نسبت به رفع آن اقدام و مجدداً برنامه را کمپایل و اجراء نماید.

## متغیرها

تمام برنامه های کامپیوتری ، بمنظور نگهداری موقت اطلاعات از متغیر ها استفاده می کنند. مثلاً در صورتیکه برنامه ای نوشته شده است که عددی را بعنوان ورودی خوانده و جذر آنرا محاسبه و در خروجی نمایش دهد ، از یک متغیر بمنظور ذخیره عدد وارد شده توسط کاربر استفاده و پس از ذخیره کردن عدد مورد نظر امکان عملیات دلخواه بر روی آن فراهم خواهد شد.

متغیرها را می بایست قبل از استفاده ، تعریف کرد. در زمان تعریف یک متغیر می بایست نوع داده هائی که قرار است در آن نگهداری گردد را نیز مشخص کرد. مثلاً می توان متغیری تعریف کرد که در آن ، اعداد نگهداری شده و یا متغیر دیگری را تعریف کرد که بتوان در آن نام و نام خانوادگی را ذخیره کرد. در زبان برنامه نویسی جاوا تمام متغیرها قبل استفاده می بایست تعریف و همزمان نوع داده هائی که می توان در آنها نگهداری گردد را نیز مشخص کرد .

مثال : در برنامه زیر ، دو متغیر width و height تعریف شده اند. نوع متغیرهای فوق ، int تعریف شده است . یک متغیر از نوع int ، قادر به نگهداری یک عدد صحیح ( مثلاً ۱ ، ۲ ، ۳ ) است . مقدار اولیه هر یک از متغیرهای فوق ، مقدار ۲۰۰ در نظر گرفته شده است .

عملیات مربوط به نسبت دهی یک مقدار اولیه به متغیرها " مقدار دهی اولیه " می گویند. یکی از اشکالاتی که ممکن است در برخی از برنامه ها اتفاق افتد ، عدم مقدار دهی اولیه متغیرها است . بنابراین توصیه می گردد در زمان تعریف یک متغیر ، مقدار دهی اولیه آن را انجام تا از بروز برخی خطاهای احتمالی در آینده پیشگیری گردد.

در زبان جاوا دو نوع متغیر وجود دارد : متغیرهای ساده (Primitive) و کلاس ها. نوع int ساده بوده و قادر به نگهداری یک عدد است . تمام عملیاتی که می توان با متغیرهای ساده انجام داد ، صرفاً نگهداری یک مقدار با توجه به نوع متغیر است.



**کلاس ها** ، قادر به دارا بودن چندین بخش بوده و با استفاده از متدهای مربوط به هریک ، امکان استفاده آسان آنها فراهم می گردد. Rectangle یک نمونه از کلاس های فوق است.

متن برنامه
<pre>import java.awt.Graphics; import java.awt.Color;  public class FirstApplet extends java.applet.Applet {      public void paint(Graphics g)     {         int width = 200;         int height = 200;         g.drawRect(0, 0, width, height);         g.drawLine(0, 0, width, height);         g.drawLine(width, 0, 0, height);     } }</pre>

در برنامه ارائه شده ، همواره یک خط قطری در پنجره ای با ابعاد ۲۰۰ \* ۲۰۰ پیکسل ، رسم می گردد. عدم امکان پویایی ابعاد پنجره ، یکی از محدودیت های برنامه فوق است . فرض کنید در این رابطه ، امکانی در برنامه پیش بینی گردد که از کاربر درخواست شود ابعاد پنجره را مشخص نماید. پس از مشخص نمودن ابعاد پنجره توسط کاربر ، خط قطری بر اساس ابعاد ارائه شده ، رسم گردد. با مراجعه به صفحه مستندات مربوط به کلاس Graphic ، ( موجود در فایل java.awt.Graphic.html ، فایل فوق شامل لیست تمام توابع مربوط به عملیات گرافیکی است ) ، با تابع getClipBounds برخورد خواهیم کرد. تابع فوق پارامتری را بعنوان ورودی اخذ نکرده و یک مقدار از نوع Rectangle را برمی گرداند. Rectangle برگردانده شده ، شامل طول و عرض محدوده مورد نظر برای رسم است.

کلاس Rectangle دارای چهار متغیر به اسامی x,y,width,height است. بنابراین بمنظور امکان پویا نمودن ابعاد پنجره ، با استفاده از getClipBounds ، محدوده Rectangle را اخذ و پس از استخراج مقادیر مربوط به width و height از Rectangle ، آنها را در متغیرهای width و height ذخیره می نمائیم.

#### متن برنامه

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Rectangle;

public class FirstApplet extends
java.applet.Applet
{

    public void paint(Graphics g)
    {
        int width;
        int height;
        Rectangle r;

        r = g.getClipBounds();
        width = r.width - 1;
        height = r.height - 1;

        g.drawRect(0, 0, width, height);
        g.drawLine(0, 0, width, height);
        g.drawLine(width, 0, 0, height);
    }
}
```

پس از اجرای برنامه فوق ، مشاهده خواهد شد که Rectangle و قطرهای بصورت کامل در محدوده مربوطه قرار خواهند گرفت . پس از تغییر اندازه پنجره ، قطرهای Rectangle براساس مقادیر جدید بصورت خودکار مجدداً رسم خواهند شد. در رابطه با برنامه فوق ، ذکر نکات زیر ضروری است :

- با توجه به استفاده از کلاس Rectangle ، لازم است از java.awt.Rectangle استفاده گردد.
- در برنامه فوق سه متغیر تعریف شده است. دو متغیر ( width و height ) از نوع int و یک متغیر ( r ) ، از نوع Rectangle است.
- تابع getClipBounds ، پارامتری را بعنوان ورودی اخذ نکرده و صرفاً یک Rectangle را برمی گرداند. دستور : r=g.getClipBounds() ، یک Rectangle را برگردانده و آن را در متغیر r ذخیره می نماید.
- متغیر r ، از نوع کلاس Rectangle بوده و دارای چهار متغیر است. (x,y,width,height). بمنظور دستیابی به هر یک از متغیرها ، از عملگر نقطه استفاده

می گردد. مثلا `r.width` ، عنوان می نماید که در متغیر `r` ، مقداری با نام `width` بازیابی می گردد. مقدار مورد نظر در متغیر محلی با نام `width` ذخیره می گردد.

- در نهایت از `width` و `height` در توابع مربوط به رسم ، استفاده می گردد.

لازم به توضیح است که در برنامه فوق می توانستیم از متغیرهای `width` و `height` استفاده نکرده و مقدار `1 - r.width` را مستقیماً در اختیار توابع مربوطه قرار داد.

جاوا دارای چندین نوع متغیر ساده است . سه نمونه رایج در این زمینه عبارتند از :

- نوع صحیح (`int`)
- نوع اعشاری (`float`)
- نوع کاراکتری (`char`)

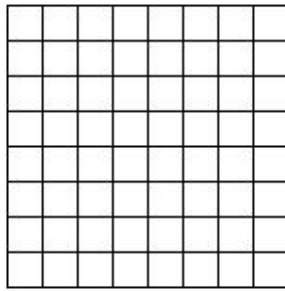
بر روی داده های نوع ساده می توان عملیات محاسباتی متفاوتی را انجام داد. در این راستا از عملگر `+` برای جمع ، `-` برای تفریق ، `*` برای ضرب و `/` برای تقسیم استفاده می گردد. برنامه زیر نحوه استفاده از عملگرهای فوق را نشان می دهد .

متن برنامه
<pre>float diameter = 10; float radius; float volume;  radius = diameter / 2.0; volume = 4.0 / 3.0 * 3.14159 * radius * radius * radius;</pre>

### حلقه های تکرار

یکی از عملیاتی را که کامپیوتر بخوبی انجام می دهد ، امکان انجام عملیات و یا محاسبات تکراری است . در بخش های قبل با نحوه نوشتن " کدهای ترتیبی " آشنا شدیم . در ادامه با نحوه تکرار مجموعه ای از کدها بمنظور تحقق عملیات و محاسبات تکراری آشنا خواهیم شد.

مثال : فرض کنید می خواهیم شکل زیر توسط کامپیوتر رسم گردد :



در ابتدا و بمنظور رسم شکل فوق ، مناسب است خطوط افقی بصورت زیر رسم گردند.



یکی از روش های رسم خطوط فوق ، ایجاد مجموعه ای از کدهای ترتیبی است که یکی پس از دیگری اجراء خواهند شد. (فقط یک مرتبه).

#### متن برنامه

```
import java.awt.Graphics;

public class FirstApplet extends
java.applet.Applet
{
    public void paint(Graphics g)
    {
        int y;
        y = 10;
        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
        y = y + 25;
    }
}
```

```

        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
    }
}

```

با مشاهده به خطوط برنامه فوق ، مشاهده می گردد که دو خط در برنامه بدفعات تکرار شده اند. با استفاده از حلقه های تکرار می توان دو خط تکرار شونده را صرفاً " یک مرتبه تکرار و فرایند تکرار عملیات را بر عهده حلقه تکرار قرار داد. بدین ترتیب یک حلقه تکرار (loop) ایجاد می گردد.

متن برنامه
<pre> import java.awt.Graphics;  public class FirstApplet extends java.applet.Applet {      public void paint(Graphics g)     {         int y;         y = 10;         while (y &lt;= 210)         {             g.drawLine(10, y, 210, y);             y = y + 25;         }     } } </pre>

پس از اجرای برنامه فوق ، نه خط افقی که هر یک دارای طولی به اندازه ۲۰۰ پیکسل می باشند ، رسم خواهد گردید.

عبارت while باعث ایجاد یک حلقه تکرار در زبان جاوا می گردد. حلقه تکرار مادامیکه مقدار y کوچکتر و یا مساوی ۲۰۰ باشد ، ادامه خواهد یافت . شرط موجود در ابتدای حلقه while در هر مرتبه بررسی می گردد ، در صورتیکه شرط درست باشد ، دستورات موجود در حلقه مجدداً " تکرار ( دستورات محصور بین { و } ) می گردند. در صورتیک شرط موجود در ابتدای حلقه while ، نادرست باشد ، دستورات مربوطه به حلقه تکرار اجراء نشده و بلافاصله اولین دستور پس از انتهای حلقه اجراء خواهد شد.

در زمان اجرای برنامه فوق در ابتدا مقدار  $y$  معادل ۱۰ است. چون مقدار ده کمتر از ۲۱۰ می باشد، دستورات موجود در حلقه تکرار اجراء و خطی از نقطه (۱۰، ۱۰) تا (۲۱۰، ۱۰) رسم خواهد شد. در ادامه مقدار  $y$ ، سی و پنج شده و مجدداً به ابتدای حلقه (بررسی شرط) مراجعه می گردد. مقدار ۳۵ از ۲۱۰ کوچکتر بوده و شرط همچنان درست بوده و مجدداً دستورات موجود در حلقه تکرار، اجراء خواهند شد. فرآیند فوق مادامیکه مقدار  $y$  کوچکتر از ۲۱۰ می باشد، تکرار خواهد شد. پس از اینکه مقدار  $y$  از ۲۱۰ بیشتر گردید، حلقه تکرار اجراء نشده و با توجه به عدم وجود دستوری دیگر، برنامه نیز خاتمه خواهد یافت.

برای ایجاد خطوط عمودی، می توان از یک حلقه تکرار دیگر استفاده کرد.

متن برنامه
<pre>import java.awt.Graphics;  public class FirstApplet extends java.applet.Applet {      public void paint(Graphics g)     {         int x, y;         y = 10;         while (y &lt;= 210)         {             g.drawLine(10, y, 210, y);             y = y + 25;         }         x = 10;         while (x &lt;= 210)         {             g.drawLine(x, 10, x, 210);             x = x + 25;         }     } }</pre>

در زبان جاوا، می توان از عبارت for برای ایجاد حلقه های تکرار نیز استفاده کرد. نمونه برنامه زیر نحوه استفاده از حلقه for در مقابل حلقه while را نشان می دهد.

حلقه تکرار با استفاده از For
<pre>for (y = 10; y &lt;= 210; y = y + 25) {     g.drawLine(10, y, 210, y); }</pre>
حلقه تکرار با استفاده از While
<pre>y = 10; while (y &lt;= 210) {     g.drawLine(10, y, 210, y);     y = y + 25; }</pre>

جاوا یکی از بهترین زبانهای برنامه نویسی در حال حاضر بوده و دارای امکانات فراوانی است . در این مقاله صرفا هدف آشنائی اولیه و عمومی با زبان برنامه نویسی جاوا بود. علاقه مندان می توانند از سایر منابع موجود خصوصا مستندات ارائه شده به همراه جاوا برای تکمیل اطلاعات خود استفاده نمایند

## منابع

آموزش جاوا در ۲۱ روز – لورالمی ، چارلز پرکینز – مترجم : علیرضا زارع پور  
جاوا – هربرت شیلد – مترجم : فرهاد قلی زاده نوری  
آموزش گام به گام برنامه نویسی جاوا – عین ا... جعفرنژاد  
جاوا رهیافتی شی گرا – مترجم : علیرضا منتظرالقائم  
ساختمان داده ها و الگوریتم ها در جاوا – مترجم : کیارش بحرینی

### Persian Refrence :

[www.dev.ir](http://www.dev.ir)

[www.java.schoolnet.ir](http://www.java.schoolnet.ir)

[www.sarzemine-it.com](http://www.sarzemine-it.com)

[www.sohail2d.com/forum](http://www.sohail2d.com/forum)

[www.paradise19791979.persianblog.com](http://www.paradise19791979.persianblog.com)

[www.qomcse.com/forum](http://www.qomcse.com/forum)

### English Refrence :

Java programming primer.pdf (e-book)

[www.cs.umb.edu/~serl/java/ppt/](http://www.cs.umb.edu/~serl/java/ppt/)

[www.iut-orsay.fr/~fournier/Cork/OOP.pdf](http://www.iut-orsay.fr/~fournier/Cork/OOP.pdf)



**راهنمای برنامه نویسی جاوا  
برای  
مهندسين نرم افزار**



Java programming  
for  
Software Engineers