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

مقدمه

Java به وسیله Ed، Chris World، Patrick Haughton، James Gosling Mike Shridon Frank در سال ۱۹۹۱ در لابراتوار شرکت sun پایه ریزی شد ه 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 قوي 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 : اصلاً اینگونه نیست. شاید برای افراد راحت طلب درست باشد.

بدون رابط کاربر : اگر منظور IDE باشد، بیشتر از انگشتان دست براي این زبان IDE وجود دارد. کافی است کمی در مورد آن تحقیق کرد تا با IDE های آن مانند Eclipse ، NetBeans ، JBuilder ، IntelliJ ، Sun studio ، 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

فهرست

عنوان ..... صفحه

تاریخچه Java

عناوین این بخش :

مدل زبان و کاربردهای آن

ایجاد java

Appletها و Applicationها

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 هاي آن آشنا میشوید.

تاریخچه Java

در اواخر دهه ۸۰ و اوایل دهه ۹۰ میلادی، زبان c++که یک زبان ش یئ گرا بود، جلودار زبان های برنامه نویس ي 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 کارآیی 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 مایکرویوها 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 ۱۹۹٤ 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 الکترونیکي به برنامه نویسي اینترنت معطوف کردند. Java به طور گسترده از سال ۱۹۹٦ 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 شیئ گرایی ++ c باعث توجه برنامه نویسان پیشرفته و با تجربهC++/ 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 تفاوتهایی باC++ 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++ طراحي نشده است. Java براي حل یکسري مشکلات خاص و C++ نیز براي 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 شئ گرایی در Java به غایت خود رسیده است.

Appletها و Applicationها

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 نمي 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 مرور گر 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، Solarisو...) 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 کارآیی 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 کامپایل

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 " Write Once ,run any where" 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- زبانJava 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) برسی می کند که کلاس ها به محدودیت ها دسترسی نداشته باشند.

٤- 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 گرها 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 نگهبان 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)

بارکنده کلاس، نوعی دیگر از نگهبان 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 کیرد. 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.

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

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

٦- 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، SocketImplFactory یا URLStreamHandlerFactoryرا 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 پورتی پایین تر از ۱۰۲٤ 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

عناوين اين بخش :

نصب برنامه

نوشتن يك برنامه ساده و آماده كردن آن براي اجرا

نحوه كامپايل برنامه اصلي

اجراي فايل اصلي (Virtual Java Machine)

نحوه اجراي فايل اصلي

خطای گرامر (Grammatical Error)

اشكال (Bug)

توضيحات (Command)

پيغام خطا

ساير كامپايلرها

* به همراه CD ، نرم افزار مربوطه و چند نمونه كد ضميمه شده است.

نصب برنامه

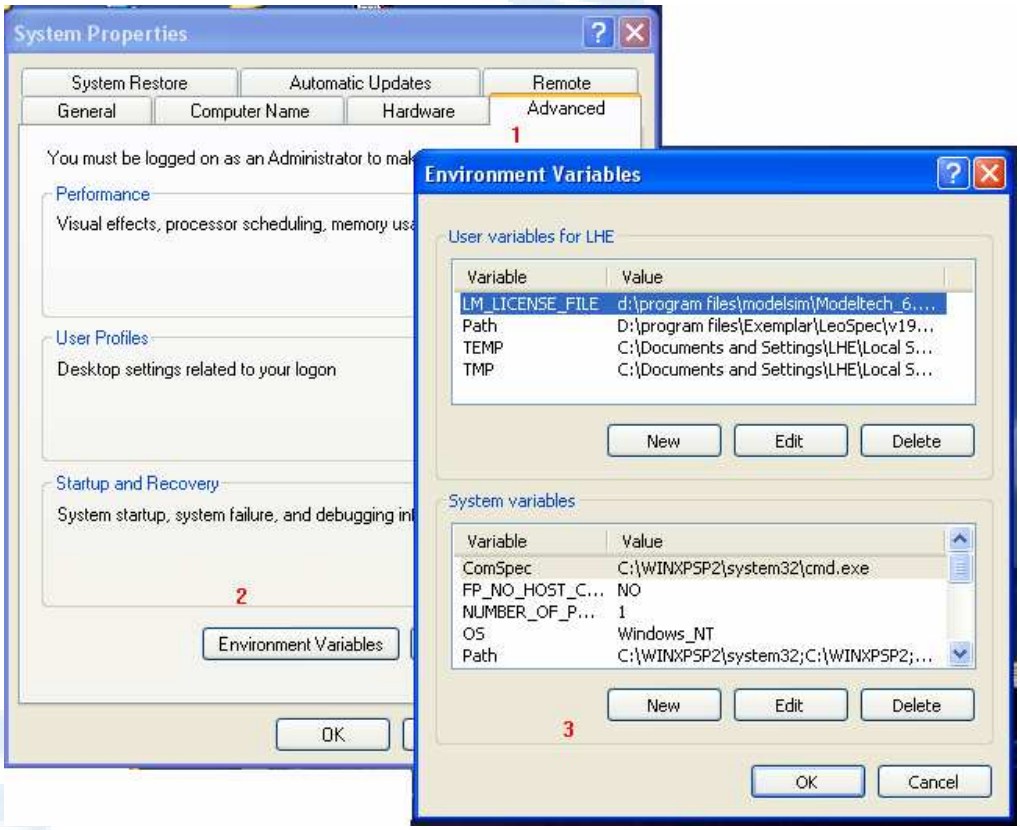
جهت اجراي برنامه هاي جاوا بسته نرم افزازي j2sdk-1_3_1_01-win كه به منظور استفاده بر روي سيستم عامل ويندوز مي باشد را مانند ساير نرم افزار ها در مسير دلخواه نصب مي كنيم.

بعد از اتمام نصب برنامه مي بايست مسير شاخه اي كه فايل هاي java و javac در آنها مي باشد طبق روال زير تنظيم شود.

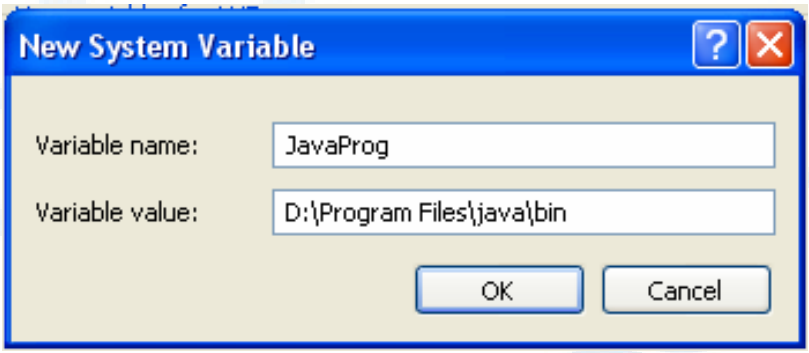
۱ – برروي My Computer كليك راست كرده و Properties را انتخاب مي كنيد.

۲- به تب Advanced رفته و بر روي Environment Variables كليك كنيد.

۳ – در پنجره فوق در بخش System Variables دكمه New را انتخاب كنيد.



٤ – در اين مرحله مي بايست مسيري كه فايل هاي java و javac در آن مي باشد تعريف كنيم. اين فايل ها در شاخه bin مي باشند. مسير بايد به طور كامل طبق شكل زير بيان شود. براي نام نيز مي توانيد هر نامي را انتخاب كنيد.



بعد از ok كردن ، Variable فوق به ليست اضافه مي شود. پنجره ها را ok كنيد.

در اين مرحله نصب برنامه به اتمام رسيد.

حال با چگونگي اجراي يك برنامه آشنا مي شويم.

قبل از آن مي بايست به اين نكته توجه داشت كه توضيحات زير مربوط به نسخه اي است كه ما آن را نصب كرده ايم. چنانچه شما از محيط جاواي ديگري استفاده مي كنيد ، قطعا روش نصب و اجراي برنامه و امكانات محيط برنامه متفاوت خواهد بود. لذا ، اطلاعات بيشتر را در مستندات كامپايلر خود مشاهده نماييد.

نوشتن يك برنامه ساده و آماده كردن آن براي اجرا

براي آماده سازي يك برنامه مسير زير را طي كنيد.

```
Start > Programs > Accessories > Notepad
```

برنامه notepad را باز كنيد. طبق آنچه آموخته ايد و بر اساس برنامه هاي نمونه اي كه توضيح داديم ، برنامه اي را بنويسيد.

در اين مرحله ما برنامه اي ساده را انتخاب كرده ايم :

```
class Example {

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

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

بعد از نوشتن برنامه از گزينه File > Save as برنامه را با پسوند java. ذخيره نماييد.

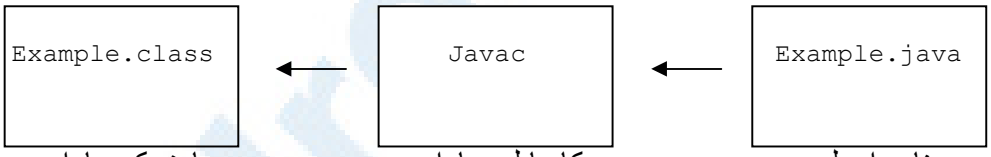
نكته :

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

۲ – برنامه را بهتر است در همان مسير فايل هاي java و javac يعني فولدر bin ذخيره نماييد.

۳ – مي توانيد نام فايل به همراه پسوند آن را در بين دو علامت " " بنويسيد ، مانند "Example.java" ، زين پس اين فايل source file ما شناخته ميشود.

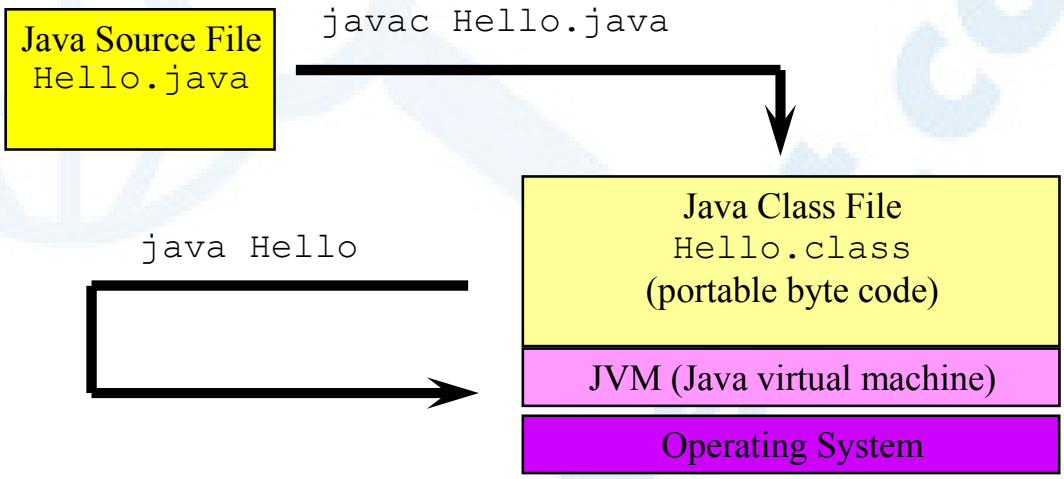
تا اين مرحله شما با استفاده از ويرايشگر Notepad يك فايل text ساخته ايد كه كدهاي جاوا را به صورت حروف يا character در آن ذخيره كرده ايد. محتويات اين فايل را مي توانيد بر روي مانيتور مشاهده يا از آن پرينت بگيريد و يا تغيير دهيد. كامپيوتر چنين فايلي را نمي تواند مستقيما اجرا كند ، چراكه به صورت بايت ذخيره شده است ، لذا بايد آن را به byte code تبديل كرد.عمل تبديل برنامه اصلي به بايت كد توسط كامپايلر جاوا مي باشد.



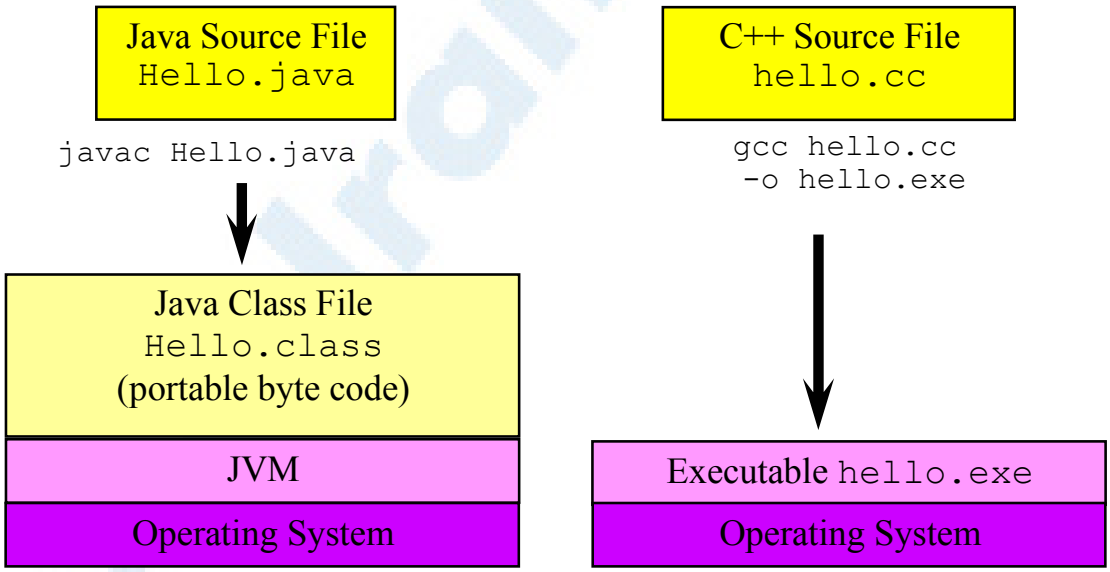
همان طور كه ملاحظه كرديد حاصل تبديل برنامه به بايت كد ، فايل با پسوند class. مي باشد.

مطالب بالا به صورت زير گويايي بيشتري خواهد داشت :

Java Computation Model



Byte Code vs. Machine Code



نحوه كامپايل برنامه اصلي

از مسير زير Command Prompt را اجرا كنيد :

```
Start > run > cmd
```

با استفاده از دستور DIR *.java تمامي فايل هاي جاواي موجود در دايركتوري فعلي نمايش داده ميشود.

نكته :

۱ – براي تغيير درايو به صورت زير عمل كنيد :

نام درايو:

۲ – براي رفتن به دايركتوري هاي يك درايو از دستور زير استفاده كنيد :

Cd نام دايركتوري\ نام دايركتوري\ ...

۳ – براي خروج از يك دايركتوري دستور زير كاربرد دارد :

```
cd..
```

با دستوراتي كه عنوان شد به شاخه bin برويد. با استفاده از دستور

```
Javac Exmaple.java
```

برنامه را كامپايل كنيد. در صورتيكه هيچ گونه خطايي در برنامه نداشته باشيد ، بعد از فشردن enter ، فايل بايت كد (.class) ساخته ميشود.

نكته : چنانچه مسير path را درست وارد نكرده و يا برنامه را درست نصب نكرده باشيد بعد از وارد كردن دستور ... javac يك پيغام خطا ظاهر ميشود. بهتر است يك بار ديگر برنامه را نصب كنيد.

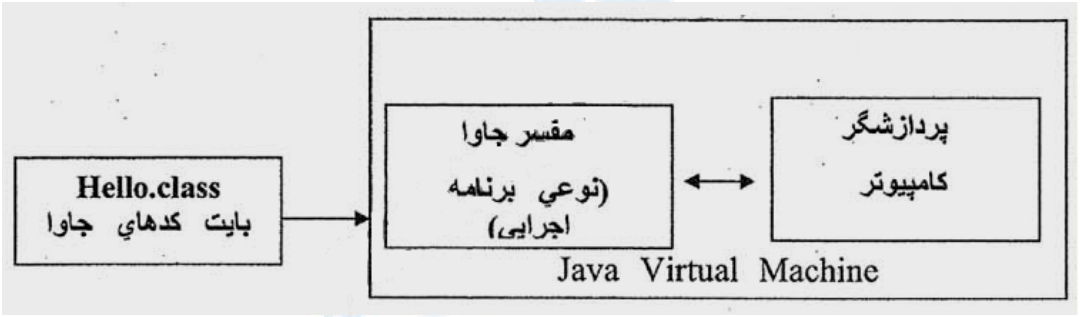
```
'javac' is not recognized as an internal or external command,
operable program or batch file.
```

اجراي فايل اصلي (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 ، بايت كدهاي جاوا (فايل class.) را خوانده ، فرامين آن ها را اجرا مي كند. اين برنامه را اصطلاحا مفسر interpreted و عمليات آن را interpretation مي نامند. مفسر بايت كدهاي جاوا نوعي برنامه اجرايي است كه در هر سيستم كامپيوتري قابل اجراست. هر يك از شركتهاي 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 يكديگر فرق دارند اما هنگامي كه يك برنامه جاوا به بايت كد تبديل مي شود ، اين بايت كدها در تمام سخت افزارها يكسان هستند. در واقع حاصل ترجمه يك برنامه جاوا در تمام سخت افزارها بايت كد يكساني است كه در تمام سيستمهاي سخت افزاري بدون هيچ اشكالي قابل اجراست ، حتي اين بايت 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 اجراي فايل اصلي

براي اجراي برنامه اصلي از دستور java استفاده ميشود.

```
java Example
```

بعد از اجراي اين دستور شما متن خود را مشاهده مي كنيد و خط فرمان در اختيار شما قرار ميگيرد :

```
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 نكته :

نكته١ : توجه داشته باشيد كه جاوا به حروف بزرگ و كوچك حساس مي باشد ، حتي در نام فايل ها. بنابراين اگر مي نوشتيم java example برنامه اجرا نمي شد. اما آنچه بين " " نوشته ميشود چاپ ميشود و بزرگ و كوچكي آن مهم نيست.

نكته٢ : JVM جهت اجراي برنامه به سراغ متد main ميرود ، در واقع محل آغاز برنامه متد main مي باشد.

نكته٣ : كليه دستوراتي كه اجرايي هستند با ; در انتهاي خط آنها مشخص ميشود.

نكته٤ : سعي كنيد برنامه ها را به صورت دندانه دار بنويسيد. اين امر سبب خوانايي برنامه شما ميشود. اما اگر برنامه را به طور خطوط پشت سر هم بنويسيد نيز اجرا ميشود.

نكته٥ : قرار دادن خطوط خالي بين سطور و همچنين بين كلمات خللي به كامپايل و اجرا وارد نمي كند.

خطای گرامر (Grammatical Error)

اگر حین تایپ کدهای جاوا اشتباهی صورت گیرد آن را اصطلاحا خطای گرامری می نامند ، در این موارد کامپایلر با ذکر شماره سطری 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. نتیجتا 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 صرفا جهت دادن اطلاعات به خوانندگان کد است. در هر جای برنامه که به توضیحات نیاز باشد اگر جمله کوتاه و یک سطری بود توضیحات را پس از // و اگر چند سطری بود پس از */ و /* قرار دهید.
توجه : توضیحات اصلا 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.long. NoClassDefenitionFoundError : Hello / java
```

۲ – اگر به جای javac Example.java ، javac Example 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.

٤ – اگر در هنگام تعریف کلاس اولین } را فراموش کنید پیغامی مشابه زیر صادر میشود.

```
Hello.java:1'{'expected
class Hello
          ^
1 error
```

٥ – اگر ; را فراموش کنید چنین پیغامی نوشته میشود.

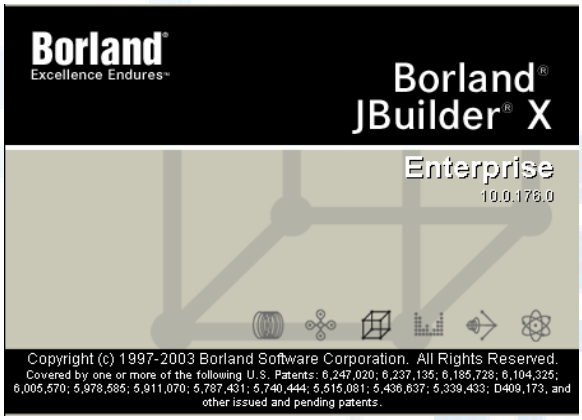
```
Hello. Java:5:";" expected
          System.out.println (" Hello world!")
                                              ^
1 error
```

سایر کامپایلرها

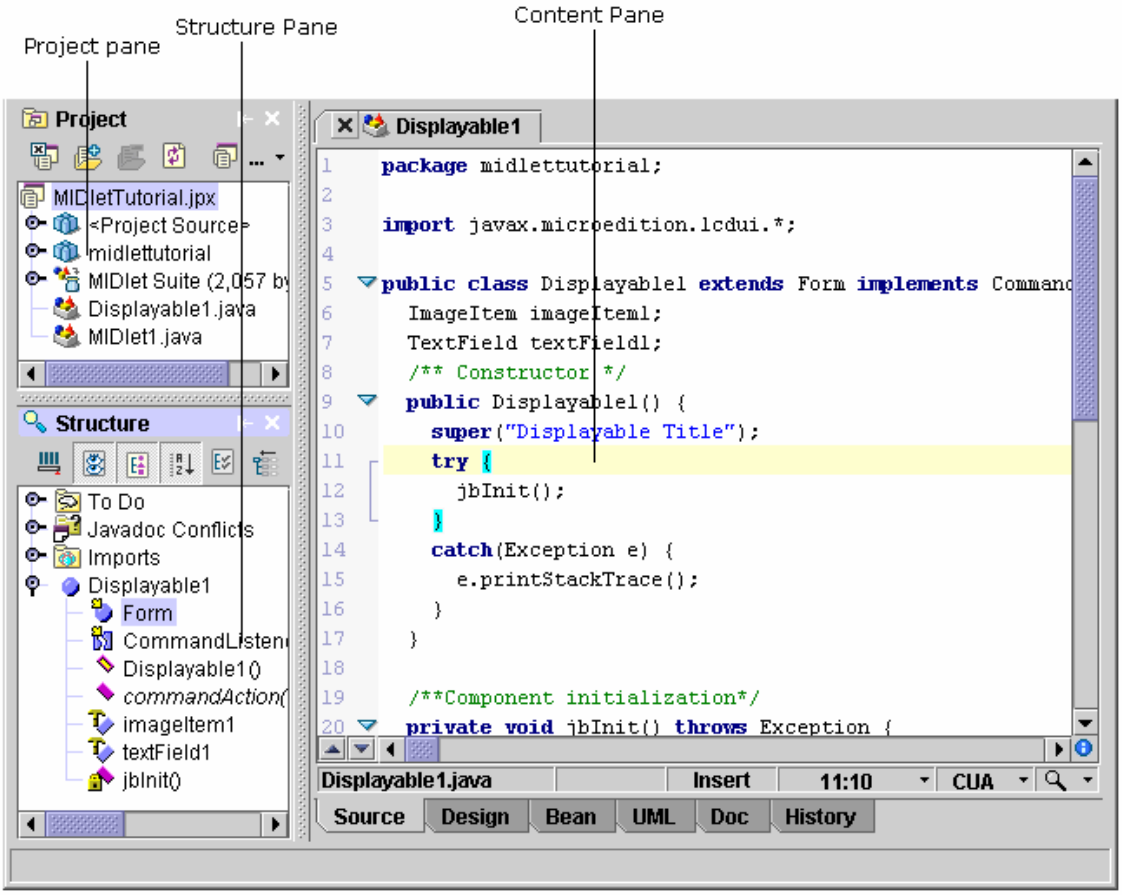
کامپایلرهای جاوا انواع مختلفی دارند ، که نرم افزارهای زیر دارای محیطهای ویژوالی هستند و بالاترین ورژن آن قیمتی حدود ٥-٦ هزار دلار می باشد. ولی ورژنهای پایین تر را می توان از طریق اینترنت دانلود کرد.

Jbuilder X – ۱

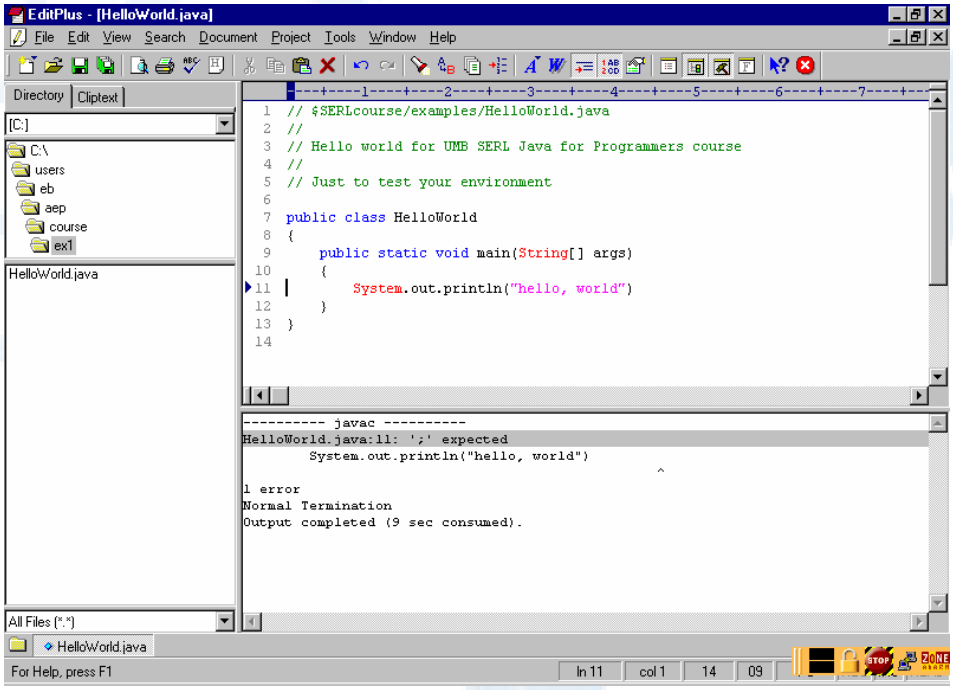
این برنام ساخت شرکت Borland می باشد.



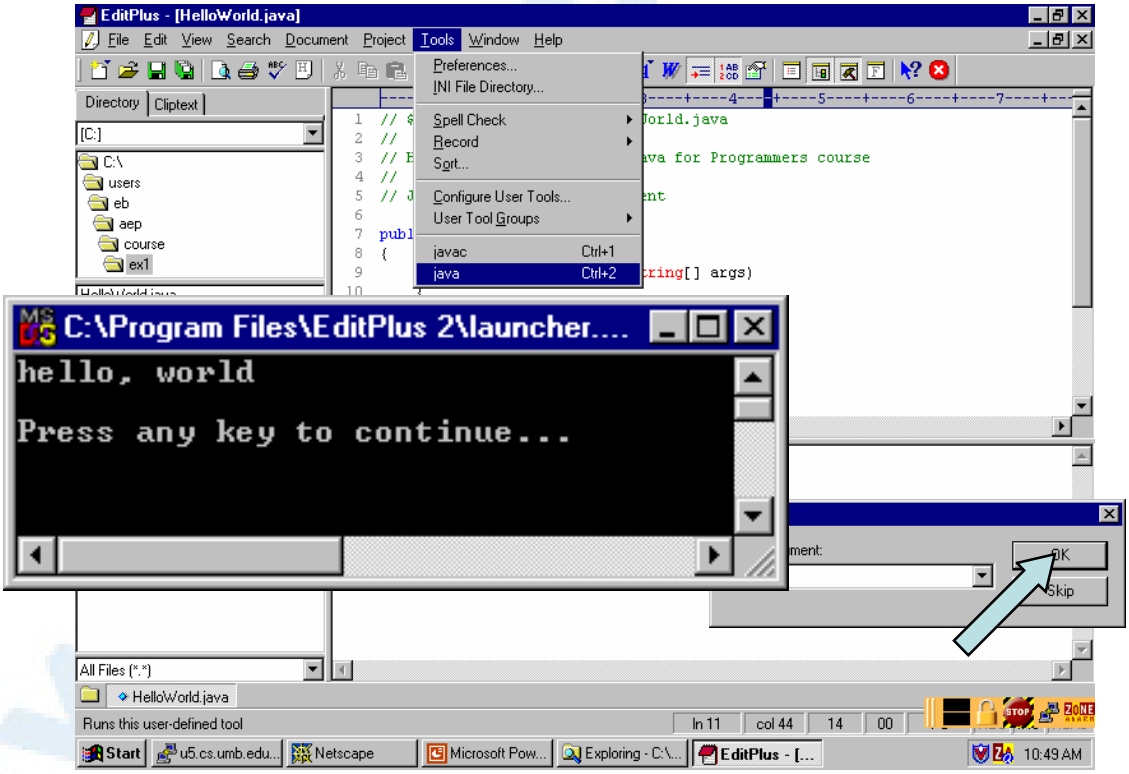
نمایشی از محیط برنامه



EditPlus – ۲



نمونه اي از اجراي برنامه به صورت تصوير ملاحظه مي كنيد :



انواع داده ها ، متغییرها

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

متغییرها

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 اشاراتي به آرایه ها شده است که در بخش ساختمان داده ها نیز توضیحاتي داده شده است.

انواع داده ها،متغيرهاوآرايه ها در جاوا

اين بخش به بررسي سه مورداز عناصرپايه جاواا ختصاص يافته است : انواع داده ها ، متغيرها و آرايه ها. جاوا نيزهمچون تمام زبان هاي برنامه سازي مدرن ، از انواع داده ها پشتيباني مي كند. ازاين نوع داده ها مي توانيد براي تعريف متغيرها و ايجاد آرايه ها استفاده كنيد. همان گونه كه خواهيد ديد ، نگرش جاوا نسبت به اين مورد ، شفاف ، كارآمد و پيوسته است.

كنترل شديد نوع داده هاي جاوا

لازم است درهمين ابتداي كارگفته شود كه جاوا زباني است كه نوع دادها به شدت درآن كنترل مي شود. درواقع ، بخشي ازامنيت و استحكام جاوا از اين امر ناشي مي شود. اينك ببينيم كه اين كاريعني چه. نخست اينكه ، هر متغير نوعي دارد ، هرعبارت نوعي دارد ، و هر يك از انواع داده ها به دقت تعريف شده اند. دوم اينكه ، هنگام تخصيص تمام مقادير، چه به طورصريح وچه از طريق ارسال پارامترها درحين فراخواني متدها ، سازگاري نوع داده ها بررسي مي شود. برخلاف برخي اززبان ها ، تبديل خودكار انواع داده هاي سازگاردر جاوا انجام نمي گيرد. كامپايلر جاوا تمام عبارت ها و پارامترها را جهت حصول اطمينان از سازگاري انواع داده ها بررسي مي كند. عدم تطبيق نوع داده ها ، خطاهايي هستند كه مي بايست پيش ازپايان كامپايل شدن هر كلاس تصحيح شوند.

انواع داده هاي پايه

هشت نوع داده پايه درجاوا تعريف شده است : byte ، short ، int ، long ، char ، float ، double و Boolean
به اين نوع داده هاي پايه ، داده هاي ساده نيز گفته مي شود واز هر دو واژه دراين كتاب استفاده شده است. اين نوع داده ها رامي توان به چهارگروه تقسيم نمود:

- اعداد صحيح – اين گروه شامل byte , short , int و long است كه اعداد كامل علامت دار مي باشند.
- اعداد اعشاري باممیز شناور- اين گروه شامل float و double است كه نمايانگر اعداد اعشاري مي باشند.
- كاراكترها- اين گروه شامل char است كه نمايانگر انواع نمادها در مجموعه كاراكترهاست ؛ ازقبيل حروف و ارقام.
- بولي- اين گروه شامل boolean است كه نوع ويژه اي براي نمايش مقاديرtrue/ false است.

موارد پيش گفته را مي توانيد به گونه اي كه هستند بكار بريد و يا اينكه آرايه ها يا انواع كلاس هاي خاص خودتان را بسازيد. ازاين رو ، انواع داده هاي پيش گفته ، پايه واساس انواع داده هاي ديگري راكه ايجاد مي كنيد ، تشكيل مي دهند.

انواع داده هاي پايه ، نمايانگر مقادير واحد هستند و نه شي هاي مركب. اگرچه جاوا كاملا شي گراست ،اما انواع داده هاي پايه اين گونه نيستند. آنها مشابه انواع داده هاي ساده اي هستند كه در بيشتر زبانهاي غير شي گرا يافت مي شوند. دليل اين امر، كارايي و بازدهي است. تبديل انواع داده هاي پايه به شي ها ، كارايي رابيش اندازه كاهش مي داد.

انواع داده هاي پايه به گونه اي تعريف مي شوند تا يك محدوده و رفتار رياضي معين داشته باشند. زبان هايي چون C و C++ امكان تغيير اندازه اعداد صحيح رابر اساس شرايط محيط اجرا فراهم مي آورند.

اما ، جاوا اين گونه نيست. به دليل نياز به داشتن قابليت انتقال برنامه هاي جاوا ، محدوده انواع داده ها ثابت است. به عنوان مثال ، int هميشه و بدون توجه به محيط اجرا ، ۳۲ بيتي است. اين امرامكان نوشتن برنامه هايي را فراهم مي سازد كه يقينا بدون هرگونه انتقال در معماريهاي ماشين ها اجرا مي شوند. اگرچه تعيين اندازه اعداد صحيح ممكن است موجب كاهش كارآيي در برخي از محيطها شود، اما انجام اين كار براي رسيدن به قابليت انتقال ضروري است.

اينك به بررسي هريك ازانواع داده ها مي پردازيم .

اعداد صحيح

درزبان جاوا چهار نوع عدد صحيح تعريف شده است : byte ، short ، int و long. تمام اين نوع داده ها علامت دار هستند؛ مقادير مثبت و منفي. جاوا از اعداد صحيح غيرعلامت دار« فقط- مثبت» پشتيباني نمي كند. بسياري از زبان هاي كامپيوتري ديگر، هم ازاعداد صحيح مثبت وهم منفي پشتيباني مي كنند. اما طراحان جاوا احساس كردند كه اعداد صحيح فاقد علامت غير ضروري هستند. بخصوص اينكه ، آنها تصور مي كردند كه مفهوم اعداد فاقد علامت عمدتا براي مشخص شدن رفتار بيت منتهي اليه سمت چپ ، كه علامت داده هاي نوع int راتعيين مي كند ، به كار برده مي شود.

طول اعداد صحيح ، مقدار فضاي مصرفي آنها در حافظه نيست و در عوض رفتار متغيرها و عبارتهاي آن نوع را مشخص مي كند. محيط زمان اجراي جاوا مي تواند ازهر اندازه اي استفاده كند ، مشروط بر اين كه متغيرها وعبارتها متناسب با نوع داده تعريف شده رفتار نمايند. درحقيقت، حداقل يكي از نگارشها ، byte وshort را به صورت مقادير ۳۲ بيتي (۸ و۱۶بيتي) ذخيره مي كند تا كارآيي افزايش يابد، چرا كه اندازه "word " در بيشتر كامپيوترهاي مورد استفاده امروزي ، ۳۲ بيت است.

همان گونه كه در جدول ذيل نشان داده شده است ، طول و محدوده مقادير انواع داده هاي صحيح متغير است :

نام	طول	محدوده مقادير
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

اينك به بررسي هريك ازآنها مي پردازيم.

Byte

کوچکترین نوع اعداد صحیح ، byte است. byte ، نوعي عدد صحيح ۸ بيتي علامت دار است که محدوده آن از ۱۲۸- تا ۱۲۷است. متغيرهاي نوع byte بخصوص هنگام کاربا جرياني از داده هاي يك شبکه يا فايل مفيد واقع مي شوند. آنها همچنين هنگام کار با داده هاي باينري خاص که ممکن است مستقيما با ساير انواع داده هاي توکار جاوا سازگار نباشند ، مفيد واقع مي شوند.
اين نوع متغيرها با استفاده از کلمه کليدي byte تعريف مي شوند. به عنوان مثال ، درسطر ذيل متغير به نام b و c از نوع byte تعريف مي شوند :

```
byte b, c ;
```

Short

short ، نوعي عدد صحيح ۱۶ بيتي علامت دار است. محدوده آن از ۳۲۷۶۸- تا۳۲۷۶۷ مي باشد. اين نوع داده ها احتمالا کمترين استفاده را در جاوا دارند ، چرا که تعريف آنها به گونه است که نخستين بايت سمت راست آنها ، بيشترين ارزش را دارد (فرمتي به نام big-endian) اين نوع داده ها عمدتا درکامپيوترهاي ۱۶ بيتي که به طور فزاينده اي کمياب مي شوند ، کاربرد دارند. به مثالهايي از تعريف متغيرهاي short توجه کنيد:

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

توجه : " Endianness" واژه اي است که چگونگي ذخيره سازي داده هاي چند بايتي ، از قبيل short ، int و long ، در حافظه را توصيف مي کند. اگر۲ بايت براي داده هاي نوع short لازم باشد ، کداميك درابتدا قرار مي گيرد؛ بايت با ارزش بيشتر يا بايت با ارزش کمتر؟ اينکه گفته مي شود کامپيوتري big-endian است ، بدين معناست که ابتدا بايت با ارزش بيشتر ذخيره مي شود و سپس بايت با ارزش کمتر قرار مي گيرد. کامپيوترهايي چون SPARS و PowerPC از نوع "big-endian" هستند و کامپيوترهاي مبتني بر سري x86 اينتل ، از نوع "little-endian".

int

متداولترين نوع داده هاي صحيح ، int است. اين نوع داده ها ، ۳۲ بيتي علامت دار بوده و محدوده آنها از ۶۴۸ , ۴۸۳ , ۱۴۷ , ۲- تا ۶۴۷ ,۴۸۳ ,۱۴۷ ,۲ مي باشد. متغيرهاي نوع int

علاوه بر كاربردهاي ديگر، براي كنترل حلقه ها و به عنوان شاخص آرايه ها نيز عموما به كار برده مي شوند. هرگاه با عبارتي متشكل از داده هاي نوع byte ، short وint و اعداد صحيح ليترال سر و كار داشته باشيد ، كل عبارت پيش ازانجام محاسبات به int ارتقاء مي يابد.
int ، متنوعترين و كارآمدترين نوع داده هاست و در بيشتر مواقعي كه اعدادي براي شمارش يا شاخص آرايه ها نياز داريد و يا نياز به محاسبات اعداد صحيح داريد ، مي بايست از آن استفاده نماييد. ممكن است اين گونه به نظر رسد كه استفاده از short يا byte موجب صرفه جويي درفضاي حافظه مي شود ، اما هيچ تضميني وجود ندارد كه جاوا آنها را به طور خودكار به int ارتقاء ندهد. به خاطر داشته باشيد كه « نوع داده» ، تعيين كننده رفتاراست و نه اندازه (تنها استثناي موجود ، آرايه ها هستند كه چنانچه از نوع byte باشند ، براي هرعنصرشان از يك بايت استفاده مي شود و براي آرايه هاي نوع short و int به ترتيب از ۲ و ۴ بايت به ازاي هرعنصر استفاده مي شود).

Long

Long ، نمايانگرداده هاي ۶۴ بيتي علامت دار است و براي شرايطي مفيد واقع مي شود كه int براي نگهداري مقدار مورد نظر، به اندازه كافي بزرگ نباشد. محدوده داده هاي نوع long كاملا بزرگ است. اين موضوع ، long رابراي مواقعي كه اعداد صحيح بزرگ لازم باشند ، مفيد ساخته است. به عنوان مثال ، برنامه ذيل مسافتي كه نور در مدت روزهاي مشخص شده طي خواهد كرد را محاسبه مي كند.

```
// 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 شكل ذيل خواهد بود:

```
In 1000 days light will travel about 16070400000000 miles.
```

واضح است كه نتيجه برنامه فوق را نمي توان در متغيري از نوع int نگهداري كرد.

انواع داده هاي اعشاري با ممیز شناور

اعداد اعشاري با ممیز شناور، اعداد real نيز ناميده مي شوند و براي ارزيابي عباراتي مفيد هستند كه نگهداري قسمت اعشاري نيز ضروري باشد. به عنوان مثال ، حاصل محاسباتي چون جذر و محاسبات مثلثاتي چون سينوس و كسينوس ، اعدادي هستند كه نگهداريشان مستلزم استفاده از متغيرهاي اعشاري با ممیز شناور است. مجموعه استاندارد (IEEE-754) عملگرها و داده هاي اعشاري با ممیز شناور در جاوا پياده سازي شده است. float و double دو نوع متداولي هستند كه نمايانگر اعداد اعشاري با دقت ساده و مضاعف مي باشند. طول و محدوده آنها در ذيل نشان داده شده است :

نام	طول بر حسب بيت	محدوده تقريبي مقادير
double	٦٤	$1/8\times e^{308}$ تا $4/9\times e^{-324}$
float	٣٢	$3/4\times e^{38}$ تا $1/4\times e^{-45}$

اين دو نوع در ذيل بررسي شده اند.

Float

float ، نمايانگر مقادير اعشاري با دقت ساده است كه از ۳۲ بيت براي ذخيره سازي استفاده مي كند. «دقت ساده» در برخي ازپردازنده ها سريعتر است و به اندازه نصف «دقت مضاعف» به فضاي ذخيره سازي نياز دارد ، اما وقتي كه مقادير بسيار بزرگ يا بسيار كوچك باشند ، نتايج نادرست خواهند شد.

متغيرهاي نوع float زماني مفيد واقع مي شوند كه جز اعشاري لازم باشد و دقت بالا ضرورت نداشته باشد. مثلا ، float براي نمايش مقادير ارزي برحسب دلار و سنت مفيد خواهد بود.

مثالي از تعريف دو متغير float در ذيل نشان داده شده است:

```
float hightemp,  lowtemp;
```

double

براي «دقت مضاعف» ، كه توسط كلمه كليدي double مشخص مي شود ، از۶۴ بيت براي ذخيره سازي مقادير استفاده مي شود. واقعيت امر آن است كه «دقت مضاعف» در برخي از پردازنده هاي مدرني كه براي محاسبات رياضي با سرعت بالا بهينه شده اند ، سريعتر است. حاصل توابع رياضي چون()sin،()cos و()sqrt ، مقدار double است. وقتي نياز به حفظ دقت محاسبات در محاسبات تكراري داشته باشيد و يا هنگام پردازش و مديريت اعداد بسيار بزرگ ، double بهترين گزينه خواهد بود.

برنامه كوتاه ذيل از متغيرهاي double براي محاسبه مساحت يك دايره استفاده مي كند :

```
// 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);
      }
  }
```

كاراكترها

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

char در C ، C++ ،عدد صحيحي است كه طول آن ۸ بيت است. اما درجاوا اين گونه نيست. در عوض جاوا از يونـي كد براي نمايش كاراكترها استفاده مي كند. يونـي كد ، مجموعه كاراكترهاي كاملا بين المللي است كه مي توانند نمايانگر تمام كاراكترهاي موجود در همه زبانهاي طبيعي باشند. يونـي كد درواقع اجتماع مجموعه كاراكترهاي بيشماري ازقبيل لاتين، يوناني، عربي، اسلاو، عبري، ژاپني، مجاري وغيره، به شمار مي آيد. به همين دليل به ۱۶ بيت نياز دارد. از اين رو char در جاوا ،۱۶بيتي است. محدوده ازصفرتا ۶۵۵۳۶ است. char منفي وجود ندارد. مجموعه كاراكترهاي استاندارد آسكي هنوز درمحدوده صفر تا ۱۲۷ قرار دارد و مجموعه كاراكترهاي ۸ بيتي گسترش يافته (ISO-Latin-1) در محدوده صفر تا ۲۵۵ قرار دارد. از آنجايي كه جاوا براي فراهم ساختن امكان نوشتن آپلت جهت استفاده در سرتاسر جهان طراحي شده است ، طبيعي است كه از يونـي كد براي كاراكترها استفاده نمايد. البته ، استفاده از يونـي كد براي زبانهايي چون انگليسي ، آلماني ، اسپانيايي يا فرانسوي كه كاراكترهايشان به آساني با ۸ بيت قابل نمايش هستند، قدري ناكارآمد است. اما اين گونه موارد، بهاي قابليت انتقال عمومي وجهاني است.

توجه : براي كسب اطلاعات بيشتر درباره يونـي كدبه http://www.unicode.org رجوع كنيد.

برنامه زير كاربرد متغيرهاي 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 اين برنامه به شكل زير است :

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

توجه داشته باشيد كه مقدار ۸۸ كه نمايانگر مقدار آسكي (ويوني كد) متناظر با حرف X است ، به ch1 تخصيص مي يابد. همان گونه كه گفته شد ، مجموعه كاراكترهاي آسكي ،۱۲۷ مقدار نخست را در مجموعه كاراكترهاي يوني كد اشغال مي كنند. به همين دليل تمام «حقه هاي قديمي» كه در گذشته در خصوص كاراكترها به كار برده ايد ، در جاوا نيز قابل استفاده خواهند بود.

اگرچه داده هاي نوع char ،عدد صحيح نيستند ، اما در بسياري از موارد مي توانيد به گونه اي با آنها كار كنيد كه گويي عدد صحيح هستند. اين امر به شما امكان مي دهد تا دو كاراكتر را با هم «جمع» (ادغام) كنيد و يا مقدار يك متغير كاراكتري را افزايش دهيد. به عنوان مثال ، برنامه ذيل را در نظر بگيريد:

```
// char variables behave like integers.
class charDemo2 {
   public staticvoid main (string args[]) {
       char ch1;
       ch1 =' x';
       system. out.println ("ch1 contans " +  ch1);
       ch1++; // increment ch1
       system. out.println("ch1 is now" + ch1);
  }
}
```

خروجي برنامه به شكل زير است:

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

در برنامه بالا ، ابتدا مقدار X به ch1 تخصيص مي يابد. سپس ، ch1 افزايش داده مي شود. اين كار موجب ذخيره شدن Y در ch1 مي شود؛ يعني كاراكتر پس از X در مجموعه آسكي (يوني كد).

داده هاي بولي

يكي ديگر از انواع داده هاي پايه در جاوا، boolean نام دارد براي مقادير منطقي است. تنها يكي از دو مقدارfalse يا true را مي توان به اين نوع متغيرها نسبت داد. حاصل تمام عملگرهاي رابطه اي همين است؛ از جمله a < b.

جملات شرطي كه بر عبارات كنترلي چون if و for نظارت دارند ، بايد boolean باشند.

برنامه ذيل كاربرد 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 شكل زير است:

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

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

دوم اينكه ، مقدار يك متغير بولي به تنهايي براي كنترل عبارت if كفايت مي كند. نيازي به نوشتن عبارت به صورت زير نيست :

```
if (b == true) …
```

سوم اينكه ، خروجي هر عملگر رابطه اي ، از جمله > ، يك مقدار بولي است. به همين دليل است كه حاصل عبارت 9<10،true است. به علاوه ، پرانتزهاي اضافي پيرامون ۹<۱۰ 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 و۴۲ 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 ، اُكتال (مبناي۸) و هگزادسيمال (مبناي ۱۶). در جاوا رقم صفر در ابتداي اعداد اُكتال 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 كامپايلر خواهد شد ، چرا كه ۹ در خارج از محدوده صفر تا ۷ اُكتال قرار دارد. يك مبناي متداولتر براي اعداد مورد استفاده برنامه سازان ، هگزادسيمال است كه به خوبي با " word"هاي مضرب ۸ ، مثلاً۸، ۱۶ ،۳۲ و۶۴ بيت ، مطابقت دارد.

مشخصه اعداد هگزادسيمال ، وجود0x در ابتداي آنهاست. محدوده ارقام هگزادسيمال از صفر تا ۱۵است ، بنابراين A تا F (يا a تا f) جايگزين ۱۰ تا ۱۵ مي شوند.

ليترال هاي موجب ايجاد مقدارint 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 متغيرlong ، 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 بااضافه 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.

ليترال هاي اعشاري

اعداد اعشاري با ممیز شناور، نمایانگر اعداد دسیمال با بخش اعشاري مي باشند. این اعداد را مي توان به صورت استاندارد یا با نماد علمي نمایش داد. در حالت استاندارد، یك عدد كامل و سپس علامت اعشار و در آخر نیز بخش اعشاري نمایش داده مي شود.

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

نمایانگر اعداد اعشاري با ممیز شناور به صورت استاندارد هستند.در نماد علمي ، از مقدار استاندارد ، عدد اعشاري با ممیز شناور وپسوندي كه نمایانگر تواني از ۱۰ است ، استفاده مي شود. توان به صورت E(یا)e ویك عدد دسیمال نمایش داده مي شود كه یا مثبت است یا منفي.

۶,۰۲۲E23 ، 314159-05 و 2e+100 چند نمونه از این اعداد هستند.

در جاوا به طور پیش فرض از «دقت مضاعف» براي لیترال اعشاري با ممیز شناور استفاده مي شود. براي مشخص كردن لیترال هاي float ، باید F یا f را به انتهاي آنها بیفزایید. با افزودن D یا d نیزمي توانید لیترال هاي double را به طور صریح مشخص كنید. البته انجام این كار اضافي است. در پیش فرض double از ۶۴ بیت استفاده مي شود ، درصورتي كه در مقادیر float از ۳۲ بیت استفاده مي شود.

ليترال هاي بولي

لیترال هاي بولي بسیار ساده هستند. تنها دو مقدار منطقي بولي وجود دارد: true وfalse . مقادیر true وfalse به هیچ گونه نمایش عددي تبدیل نمي شوند. لیترال true در جاوا برابر با "۱ " نیست و لیترال false نیز با "۰"برابر نیست. این مقادیر در جاوا تنها به متغیرهاي بولي قابل تخصیص مي باشند و یا در عبارتهاي بولي همراه با عملگرهاي بولي قابل استفاده هستند.

ليترال هاي كاراكتري

كاراكترها در جاوا ، ایندكس هاي مجموعه كاراكترهاي یوني كد به شمار مي آیند. كاراكترها ، مقادیر۱۶بیتي هستند كه به اعداد صحیح قابل تبدیل بوده و با عملگرهایي چون "+" و"-" قابل

پردازش و مديريت مي باشند. هر ليترال كاراكتري دربين علائم نقل قول نمايش داده مي شود. تمام كاراكترهـاي آسـكي قابـل رؤيـت را مـي تـوان مسـتقيما در بـين علائـم نقـل قـول نوشـت ، مـثلا 'a'،'z'و'@'. بـراي كاراكترهـايي هم كـه مستقيما قابل نوشتن نيستند ، مـي تـوان از " escape sequence"استفاده نمود كه امكان وارد كردن

كاراكترهاي مورد نياز را فراهم سازد؛ مثلا "\'به جاي كاراكتر علامت نقل قول تكي.\n\ 'به جاي كاراكتر نمايانگر سطر جديد. مكانيزمي هم براي وارد كردن مستقيم مقدار يك كاراكتر به صورت أكتال يا هگزادسيمال وجود دارد. براي كاراكترهاي أكتال،از'\' وسپس عدد سه رقمي وارد كنيد. بـه عنـوان مثـال،'\141' نمايـانگر'a'اسـت. بـراي هگزادسـيمال نيـز'\u'، و سـپس چهـار رقـم هگزادسيمال وارد مي شود. به عنوان مثال ،'\u0061'، حرف'a'در مجوعه كاراكترهـاي ISO-Latin-1 است ، چرا كه بايت نخست ۰ است.'\ua432' هم يك كاراكتر ژاپني است. فهرست اين كاراكترها در جدول 1 نشان داده شده است.

ليترال هاي رشته اي

ليترال هاي رشته اي در جاوا همچون بيشتر زبانهاي ديگر مشخص مي شوند - با نوشتن يك سري كاراكتر در بين علائم نقل قول جفتي ، چند مثال از ليترال هاي رشته اي در ذيل نشان داده شده است.

```
"Hello Word"
" two\nlines"
"\"This is in quotes\""
```

قراردادهاي أكتال\ هگزادسيمال و "escape sequence "هايي كـه بـراي ليترال هـاي كـاراكتري تعريف شده اند ، به همان صورت براي ليترال هاي رشته اي نيز قابل استفاده اند. نكتـه مهمـي كـه بايد درباره رشته هاي جاوا توجه كنيد ، آن است كه آن است كه اين رشته بايد در يك سطر آغاز و خاتمه يابند. بر خلاف زبانهاي ديگر، هيچ گونه كاراكتر ادامه خط در اينجا وجود ندارد.

توجه : همان گونه كه مي دانيد،رشته ها در برخي زبانهاي ديگر، از جمله C/C++ ، بـه صـورت آرايه اي از كاراكترها پياده سازي مي شوند. اما در جاوا اين گونه نيست. رشته ها در واقع شي

هستند. همان گونه که خواهید دید ، چون جاوا رشته ها را به صورت شي پياده سازي مي کند ، قابليتهاي زيادي براي مديريت رشته ها دارد که هم قدرتمند هستند و هم استفاده از آنها آسان است.

کاراکترهاي Escape Sequence

Escape Sequence	شرح
\ddd	Octal character (ddd)
\uxxxx	Hexadcimal 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

متغيرها

متغيرها ، واحد پايه ذخيره سازي در برنامه هاي جاوا هستند. هر متغير به وسيله ترکيبي از يك شناسه ، نوع و مقدار اوليه (دلخواه) تعريف مي شوند. به علاوه ، همه متغيرها داراي محدوده و دوره حيات هستند. محدوده هر متغير مشخص مي کند که آن متغير در چه قسمتهايي قابل استفاده است. عناصر پيش گفته در ذيل بررسي شده اند.

شيوه تعريف کردن متغيرها

همه متغيرها را بايد در جاوا تعريف وسپس به كار برد. شكل كلي تعريف كردن متغيرها در ذيل آورده شده است:

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

type ، يكي از انواع داده هاي تجزيه ناپذير جاوا ، نام يك كلاس يا رابط است. identifier ، نام متغير است. متغيرها را مي توان از طريق مشخص كردن يك علامت تساوي و يك مقدار، مقداردهي نمود. به خاطرداشته باشيد كه حاصل عبارت مقداردهي بايد مقداري از همان نوع مشخص شده براي متغير باشد.

براي آنكه بيش از يك متغير از نوع مورد نظر تعريف كنيد ، نام آنها را به وسيله كاما از يكديگر جدا كنيد. به چند مثال ذيل از تعريف متغيرهاي گوناگون توجه كنيد. دقت كنيد كه برخي از آنها داراي مقدار اوليه 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 هايي كه انتخاب مي كنيد ، چيزي در نامشان ندارند كه بيانگر نوعشان باشد. جاوا امكان تخصيص هر يك ازانواع داده ها را به گونه شناسه مي دهد.

مقداردهي اوليه پويا

اگرچه در مثالهاي قسمت پيش ، از ثابتها به عنوان مقدار اوليه استفاده شده است ، اما جاوا امكان مقداردهي پوياي متغيرها را به وسيله عبارتهاي معتبر در زمان تعريف آنها نيز فراهم كرده است. به عنوان مثال ، به برنامه كوتاه ذيل توجه كنيد كه طول وتر يك مثلث قائم الزاويه را با در دست داشتن دو ضلع ديگر محاسبه مي كند :

```
// 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 اينجا سه متغير محلي (a،b و c) تعريف شده است. دو متغير نخست ،a و b ، به وسيله دو ثابت مقداردهي شده اند. اما ، c به طور پويا با طول وتر مقدار دهي مي شود (با استفاده از قضيه فيثاغورث).

در برنامه از يكي ديگر از متدهاي توكار جاوا sqrt() ،استفاده شده است كه عضوي از كلاس Math مي باشد و جذر آرگومان خودش را محاسبه مي كند. نكته كليدي اين كار آن است كه در عبارت مقداردهي مي توان از هر عنصري كه در زمان مقدار دهي معتبر است استفاده نمود ، از جمله متدها ، متغيرها ي ديگر يا ليترال ها.

محدوده دستيابي و دوره حيات متغيرها

تا به حال تمام متغيرهاي قابل استفاده ، در ابتداي متد main() تعريف شده اند. اما ، جاوا امكان تعريف متغيرها را در هر بلوكي فراهم مي سازد. همان گونه كه مي دانيم هر بلوك با '{' آغاز و به '}' ختم مي شود. هر بلوك ، محدوده اي را تعيين مي كند. از اين رو ، هر بار كه بلوك جديدي را آغاز مي كنيد ، محدوده جديدي ايجاد مي شود. هر محدوده تعيين مي كند كه كدام شي ها براي ساير بخشهاي برنامه قابل رؤيت هستند. دوره حيات آن شيءها نيز تعيين مي شود.

بسياري از زبانهاي كامپيوتري ديگر، دو دسته عمومي از محدوده ها را تعريف مي كنند : عمومي و محلي. اما، اين محدوده هاي قديمي به خوبي با مدل شي گراي دقيق جاوا مطابقت ندارند. اگرچه امكان ايجاد متغيرهايي وجود دارد كه در نهايت به داشتن يك محدوده عمومي منجر مي شود ، اما انجام اين كارعموما يك استثنا به شمار مي آيد و نه يك قاعده كلي. در زبان جاوا ، دو محدوده اصلي ، محدوده هايي هستند كه توسط يك كلاس تعريف مي شوند و نيز محدوده هايي كه توسط يك متد تعريف مي شوند. حتي اين تمايز نيز قدري مصنوعي است ، اما ، چون محدوده كلاس ، چندين

خصوصيات و ويژگي منحصر به فرد دارد كه به محدوده تعريف شده توسط يك متد اعمال نمي شوند ، اين تمايز معقول به نظر مي رسد.

محدوده اي كه توسط هر متد تعريف مي شود ، با آكولاد باز آن آغاز مي شود. اما اگر آن متد پارامترهايي داشته باشد ، آنها نيز در محدوده آن متد قرار مي گيرند.

به طور كلي ، متغير هايي كه در يك محدوده تعريف مي شوند ، براي قسمتهايي كه در خارج از آن محدوده قرار دارند ، قابل رؤيت (دستيابي) نيستند. از اين رو، وقتي متغيري را درون يك محدوده تعريف مي كنيد ، در واقع آن را محلي مي سازيد و آن را در مقابل دستيابي و يا تغييرات غير مجاز محافظت مي كنيد. در حقيقت ، قوانين محدوده ، پايه و اساس لازم براي نهان سازي را فراهم مي سازند.

محدوده ها ممكن است تودرتو باشند. به عنوان مثال ، هر بار كه بلوكي را ايجاد مي كنيد ، در واقع محدوده جديد تودرتويي را ايجاد مي كنيد. وقتي چنين اتفاقي مي افتد، محدوده بيروني، محدوده دروني را در بر مي گيرد. اين بدين معناست كه شي هاي تعريف شده در محدوده بيروني ، براي دستورالعمل هاي محدوده دروني قابل رؤيت خواهند بود. اما، عكس اين مطلب صادق نيست. شي هايي كه در محدوده دروني تعريف مي شوند ، براي محدوده بيروني قابل رؤيت نخواهند بود.

براي درك تأثير محدوده هاي تودرتو ، برنامه ذيل را در نظر بگيريد :

```
// 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 andy: "+ 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 كه در توضيحات مشخص است ، متغيرx در ابتداي محدوده main() تعريف شده است و براي كل دستورالعمل هاي درون main() قابل دستيابي است. درون بلوك if ،متغير y تعريف شده است. چون در اين بلوك متغيري تعريف شده است ، آن متغير (y) تنها در همان بلوك خودش قابل رؤيت است. به همين دليل است كه سطر y=100 در خارج آن بلوك به « توضيح » مبدل شده است. اگر نماد « توضيح » را از ابتداي آن سطر برداريد، با خطاي زمان كامپايل مواجه خواهيد شد ، چرا كه y در بيرون محدوده اش قابل رؤيت (دستيابي) نيست. در بلوك if مي توان از x استفاده كرد ، چرا كه دستورالعمل هاي درون يك بلوك (يعني ، محدوده تودرتو) ، به متغيرهاي تعريف شده در محدوده محيطي خود ، دستيابي دارند.

در هر بلوك مي توان متغيرها را در هر نقطه اي تعريف كرد، اما متغيرها تنها پس از تعريف شدن قابل استفاده خواهند بود. از اين رو، اگر متغيري را در ابتداي يك متد تعريف كنيد، درآن صورت در كل متد قابل رؤيت خواهد بود. اما اگر متغيري را در انتهاي بلوكي تعريف كنيد ، در آن صورت عملا بي فايده خواهد بود ، چرا كه هيچ دستورالعملي به آن دستيابي نخواهد داشت. به عنوان مثال ، عبارت زير نادرست است ، چرا كه متغيرcount پيش از تعريف شدن قابل استفاده نيست:

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

به اين نكته نيز بايد توجه كنيد : متغيرها هنگام ورود به محدوده شان ايجاد مي شوند و هنگام خروج از محدوده شان نيز از بين برده مي شوند. اين بدين معناست كه هيچ متغيري مقدارش را پس از خروج ازمحدوده اش نخواهد داشت. بنابراين متغيرهايي كه درون يك متد تعريف مي شوند ، مقدارشان را در فواصل زماني بين فراخواني متد نخواهند داشت. همچنين ، متغيري كه در يك بلوك تعريف مي شود ، مقدار

را پس از خروج از بلوك از دست خواهد داد. از اين رو، دوره حيات هر متغير به محدوده اش وابسته است. اگر متغيري در حين تعريف مقدار دهي شود، در آن صورت هر بار به هنگام ورود به آن بلوك ، از نو مقداردهي مي شود. به عنوان مثال ، برنامه بعدي را در نظر بگيريد :

```
// 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 اين برنامه در ذيل 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 كه مي بينيد ، هر بار به هنگام ورود به حلقه for ، متغير y با ١- مقدار دهي مي شود. با وجود آنكه سپس مقدار ١٠٠ به آن تخصيص مي يابد ، اما مقدار جديد را از دست مي دهد.

و آخرين نكته : اگرچه بلوك ها مي توانند تودرتو باشند، اما نمي توان متغيري همنام با يكي از متغيرهاي محدوده بيروني تعريف كرد. به عنوان مثال ، برنامه ذيل معتبر نيست :

```
// 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!
        }
    }
}
```

تبديل و casting

اگر تجربه برنامه سازي داشته باشيد ، درآن صورت مي دانيد كه تخصيص يك نوع مقدار به متغيري از يك نوع ديگر نسبتا متداول است. اگر آن دو نوع سازگار باشند ، در آن صورت جاوا عمل تبديل را به صورت خودكار انجام خواهد داد. به عنوان مثال ، هميشه امكان تخصيص يك مقدار int به يك متغير long وجود دارد. اما تمام داده هاي مختلف سازگار نيستند و از اين رو، همه تبديلات به طور ضمني ممكن نيست. به عنوان نمونه ، هيچ تبديلي براي double به byte تعريف نشده است. خوشبختانه ، باز هم امكان كسب رويه هاي تبديل براي انواع داده هاي ناسازگار وجود دارد. براي انجام اين كار بايد از"casting " استفاده كنيد؛ تبديل صريح بين انواع داده هاي ناسازگار. اينك به بررسي تبديل خودكار و casting مي پردازيم.

تبديلات خودكار جاوا

وقتي يك نوع داده به متغيري از نوع ديگر تخصيص مي يابد ، چنانچه شرايط ذيل مهيا باشند ، عمل تبديل خودكار انجام مي شود :

- آن دو نوع سازگار باشند.
- نوع مقصد بزرگتر از نوع مبدأ باشد.

هرگاه اين دو شرط برقرار باشد ، نوعي «تبديل همراه با بزرگ سازي» انجام مي شود. به عنوان مثال ، نوع int هميشه براي نگهداري مقادير byte به اندازه كافي بزرگ است ، بنابراين استفاده از عبارت casting به صورت صريح ضرورت ندارد.

براي « تبديلات همراه با بزرگسازي» ، انواع داده هاي عددي ، از جمله اعداد صحيح واعشاري با مميز شناور، با يكديگر سازگار هستند. اما ، انواع داده هاي عددي با char يا boolean سازگار نيستند. همچنين ، char و boolean با يكديگر سازگار نيستند.

همان گونه كه پيش از اين گفته شد ، وقتي يك مقدار ثابت ليترال صحيح در متغيرهاي نوع byte ،short يا long ذخيره مي شوند ، جاوا باز هم عمل تبديل خودكار را انجام مي دهد.

انجام casting براي انواع داده هاي ناسازگار

اگرچه تبديلات خودكار مفيد واقع مي شوند ، اما اين تبديلات تمام نيازها را برطرف نمي سازند. به عنوان مثال ، اگر بخواهيد يك مقدار int را به يك متغير byte تخصيص دهيد ، چه مي كنيد؟ اين تبديل به طور خودكار انجام نمي شود ، چرا كه هرbyte كوچكتر ازint است. اين نوع تبديلات گاهي اوقاتَ ، « تبديل همراه با كوچك سازي» ناميده مي شود ، چرا كه مقدار int به طور صريح كوچكتر مي شود تا در مقصد قابل ذخيره باشد.

براي آنكه تبديل بين دو نوع ناسازگار را انجام دهيد ، بايد از casting استفاده كنيد، يعني تبديل صريح انواع داده ها. شكل كلي انجام كار به صورت زير است:

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

target-type ، مشخص كننده نوع داده اي است كه مقدار مورد نظر بايد به آن تبديل شود. به عنوان مثال ، در عبارت زير، عمل casting ازint به byte انجام مي شود. چنانچه مقدار صحيح بزرگتر از محدوده byte باشد ، مقدارش از طريق تقسيم بر محدوده byte و به دست آوردن باقيمانده تقسيم كوچك مي شود.

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

هنگام تخصيص يك مقدار اعشاري با مميز شناور به يك نوع صحيح ، تبديل به گونه اي ديگر انجام مي شود : برش.

همان گونه كه مي دانيد ، اعداد صحيح فاقد قسمت اعشاري هستند. از اين رو، وقتي يك مقدار اعشاري با مميز شناور به يك نوع صحيح تخصيص مي يابد، قسمت اعشاري از دست مي رود. به عنوان مثال ، اگر مقدار ۱/۲۳ به يك متغير صحيح تخصيص يابد ، مقدار حاصل ، ۱ خواهد بود. ۰/۲۳ برش داده خواهد شد. البته ، اگر اندازه عدد صحيح حاصل بزرگتر از آن باشد كه در متغير مقصد ذخيره شود ، در آن صورت مقدارش از طريق تقسيم به محدوده نوع مقصد كاهش مي يابد.

برنامه ذيل چند نوع تبديل را نشان مي دهد كه نياز 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 زير 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
```

اينك هر يك از تبديلات را بررسي مي كنيم. وقتي عمل"casting" جهت ذخيره مقدار ۲۵۷ در متغيرbyte انجام مي شود ، نتيجه ، باقيمانده تقسيم ۲۵۷ بر ۲۵۶ (محدودهbyte) مي شود كه در اين مثال ، يك است. وقتي d به يك int تبديل مي شود ، قسمت اعشاري آن از دست مي رود و مقدارش به باقيمانده تقسيم به ۲۵۶ كاهش مي يابد كه در اين مثال، ۶۷ است.

ارتقاء خودكار انواع داده ها در عبارتها

علاوه بر عمليات تخصيص ، تبديل نوع داده ها در شرايط ديگر هم محتمل است : در عبارتها. براي درك دليل اين امر، مثال ذيل را در نظر بگيريد. در برخي از عبارات ، دقت مورد نياز براي يك مقدار ميانجي گاهي اوقات از محدوده يكي از عملوندها تجاوز مي كند. به عنوان مثال، عبارت زير را بررسي كنيد :

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

نتيجه جمله ميانجي a*b از محدوده عملوندهاي نوع byte تجاوز مي كند. براي مديريت اين گونه مسائل ، جاوا در حين ارزيابي عبارتها ، عملوندهاي نوع byte يا short را به طور خودكار به int ارتقاء مي دهد. اين بدين معناست كه جمله ميانجي a*b با استفاده ازint انجام مي شود و نه byte . از اين رو، نتيجه جمله ميانجي (٤٠ *٥٠ =٢٠٠٠) حتي با وجود اينكه a و b هر دو از نوع byte هستند ، معتبر خواهد بود.

گرچه عمل ارتقاء خودكار مفيد است ، اما ممكن است موجب بروز خطاي زمان كامپايل نيز بشود. به عنوان نمونه ، مثال به ظاهر درست زير موجب بروز خطا مي شود :

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

در اين مثال نتيجه كاملاً معتبر ٥٠*٢ قرار است در متغير byte ذخيره شود. اما چون عملوندها هنگام ارزيابي عبارت به طور خودكار به int ارتقاء مي يابند ، نتيجه نيز به int ارتقاء مي يابد. از اين رو نتيجه عبارت كه از نوع int است ، بدون استفاده ازcasting در byte قابل ذخيره سازي نيست. اين امر حتي اگر مقدار در دست تخصيص، همچون همين مثال، در متغير مقصد قابل ذخيره باشد، باز هم صادق است.

در مواقعي كه از عواقب مسئله سرريز آگاهي داريد ، بايد از عمل casting صريح استفاده كنيد؛ همچون مثال زير:

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

كه نتيجه آن مقدار معتبر و درست ١٠٠ است.

قوانين ارتقاء انواع داده ها

جاوا علاوه بر ارتقاء byteها و shortها به int ، چندين قانون براي ارتقاء دارد كه در عبارات رعايت مي شوند. اين قوانين به شرح ذيل هستند: نخست اينكه ، تمام مقادير byte و short به گونه اي كه گفته شد ، بهint ارتقاء مي يابند. سپس ، اگر يكي از عملوندها از نوع long باشد ، كل عبارت به long ارتقاء مي يابد. اگر يكي از عملوندها از نوع float باشد، در آن صورت كل عبارت به float ارتقاء مي يابد و اگر يكي از عملوندها از نوع double باشد، نتيجه هم double خواهد بود.

برنامه زير نشان مي دهد كه چگونه هر يك از مقادير موجود در عبارت ارتقاء داده مي شود تا با آرگومان ديگر 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);
    }
 }
```

اينك به بررسي دقيق ارتقا انواع داده ها در اين سطر از برنامه مي پردازيم :

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

در جمله نخست f*b،b به float ارتقاء مي يابد نتيجه جمله نيزfloat مي شود. سپس در جمله i /c ،c به int ارتقاء مي يابد و نتيجه جمله نيز به نوع int تبديل مي شود. سپس در جمله d*s، مقدار s به double ارتقاء مي يابد و نوع جمله نيزdouble مي شود و بالاخره اينكه اين سه مقدار ميانجي float ، int و double بررسي مي شود. نتيجه float به علاوه يك int وfloat خواهد بود. سپس نتيجه float منهاي آخرينdouble به double ارتقاء مي يابد كه نوع نتيجه نهايي عبارت است.

آرايه ها

هر آرايه، گروهي از متغيرهاي همنوع است كه ارجاع به آنها از طريق يك نام مشترك صورت مي گيرد. در جاوا مي توان آرايه هايي از انواع مختلف را به صورت تك بعدي يا چند بعدي ايجاد نمود. هر يك از عناصر هر آرايه از طريق ايندكس خودشان قابل دستيابي هستند. آرايه ها روش مناسبي براي گروه بندي اطلاعات مرتبط به هم مي باشند.
توجه : اگر با C/C++ آشنايي داريد،مراقب باشيد. عملكرد آرايه ها در جاوا با عملكردشان در آن زبانها تفاوت دارد.

آرايه هاي تك بعدي

هرآرايه « تك بعدي» ، اساسا فهرستي از متغيرهاي همنوع است. براي آنكه آرايه اي ايجاد كنيد، ابتدا بايد يك متغير آرايه اي از نوع مورد نظر ايجاد نماييد. فرم كلي تعريف آرايه هاي تك بعدي به صورت زير است:

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

type در اينجا نوع آرايه را مشخص مي كند. نوع آرايه، نوع داده يكايك عناصر تشكيل دهنده آرايه را مشخص مي كند. از اين رو ، نوع آرايه مشخص مي كند كه چه نوع داده هايي در آرايه ذخيره خواهند شد. به عنوان مثال، آرايه اي به نام month-days از نوع int در زير تعريف مي شود:

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

اگرچه تعريف بالا موجب تثبيت متغير آرايه اي month-days مي شود، اما در واقع هنوز هيچ آرايه اي وجود ندارد. در حقيقت ، آرايه month-days كه با null مقداردهي مي شود، نمايانگر آرايه بدون مقدار است. براي آنكه month-days را با آرايه اي فيزيكي از مقادير صحيح مرتبط سازيد، بايست آرايه را با استفاده از new ايجاد و آن را به month-days تخصيص دهيد. new ، عملگر ويژه اي براي تخصيص حافظه است.

فرم كليnewبراي آرايه هاي تك بعدي به صورت زير است :

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

type مشخص كننده نوع داده ها ، size مشخص كننده تعداد عناصر آرايه وarry-var ، متغير آرايه است كه با آرايه مرتبط مي شود. يعني ، براي آنكه ازnew براي تخصيص آرايه استفاده كنيد، بايد نوع و تعداد عناصري كه بايد تخصيص داده شوند را مشخص نماييد. عناصري كه به وسيله new به آرايه تخصيص مي يابند ، به طور خودكار با صفر مقداردهي مي شوند. در اين مثال يك آرايه ۱۲ عنصري از اعداد صحيح و با month-days مرتبط مي شود.

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

پس از اجراي عبارت بالا، month-days به آرايه اي از ۱۲ عدد صحيح اشاره خواهد داشت. به علاوه، تمام عناصر آرايه با صفر مقداردهي خواهند شد.

اينك مطالب پيش گفته را مرور مي كنيم : ايجاد هر آرايه نوعي فرآيند دو مرحله اي است. نخست اينكه ، بايد متغيري از آرايه مورد نظر تعريف كنيد. دوم اينكه، بايد حافظه محل آرايه را با استفاده از new تخصيص دهيد وآن را به متغير آرايه نسبت دهيد.

ازاين رو، تمام آرايه ها در جاوا به صورت پويا تخصيص مي يابند. اگر مفهوم تخصيص پويا برايتان نا آشناست، نگران نباشيد. اين موضوع به تفصيل بررسي خواهد شد.

پس از تخصيص آرايه، با مشخص كردن ايندكس عناصر در بين كروشه، به راحتي مي توانيد به هر يك از آنها دستيابي داشته باشيد. ايندكس تمام آرايه ها از صفر آغاز مي شود. به عنوان مثال، عبارت زير مقدار ۲۸ را به دومين عناصرmonth-days تخصيص مي دهد.

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

سطر زير هم سبب نمايش مقدار ذخيره شده در ايندكس شماره ۳ (عنصر چهارم) مي شود.

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

تمام مطالب پيش گفته در برنامه زير به كار برده شده اند. اين برنامه، آرايه اي متشكل از تعداد روزهاي هر ماه ايجاد مي كند.

```
// 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-days4] = 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، بنابراين تعداد روزهاي ماه آوريل ، mounth-days[3] يا ۳۰ است.

امكان تركيب تعريف متغير آرايه با تخصيص خودِ 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 خواهيد ديد.

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 يكديگر جدا مي شوند. كاماها، مقادير عناصر آرايه را از يكديگر 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 improvied 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 خواهيد ديد.

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 mounth-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 ميانگين 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 خواهيد ديد، 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 ، درعبارت زير يك آرايه دو بعدي به نام twoD تعريف شده است.

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

عبارت بالا يك آرايه ۴×۵ 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 ۳-۱ خواهد بود.

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

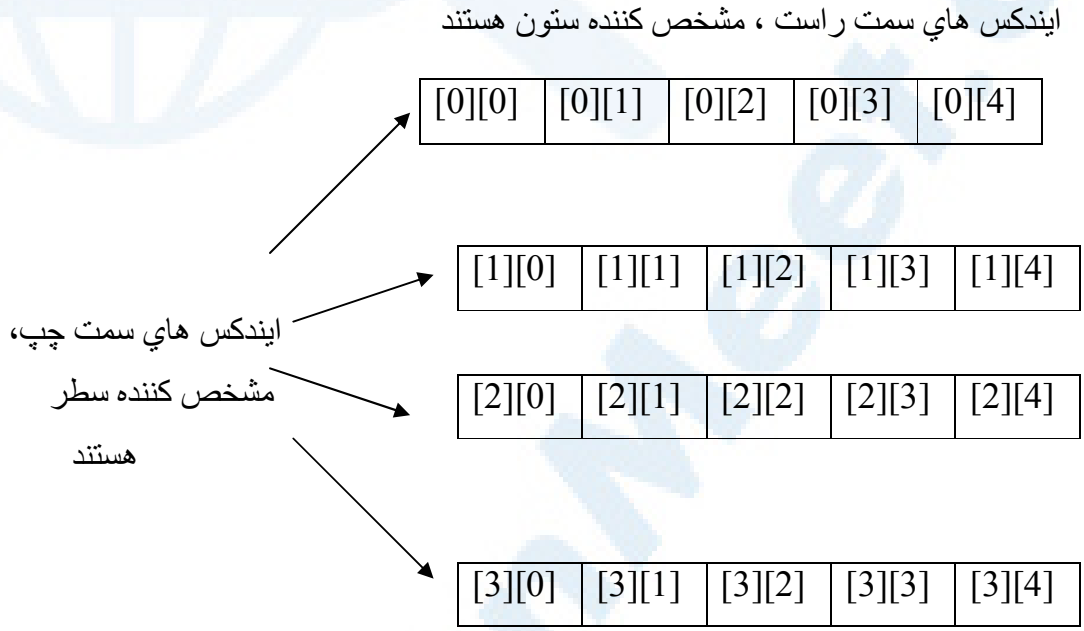
```
// Demonstrate a two-dimensional array.
class TwoDArray   {
   public static void main(string args[ ]) {
       int twoD[ ] [ ]= new int[4] [5];
       int i, j, k = 0;
       for(i=0; i < 4; i++)
           for(j=0; j < 5; j++)  {
               twoD[i] [j] = k;
               k++;
            }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



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

اگرچه در اين مثال تخصيص جداگانه بعد دوم آرايه ها هيچ مزيتي ندارد ، اما انجام اين كار در شرايط ديگر ممكن است سودمند باشد. به عنوان مثال ، وقتي ابعاد آرايه ها را به طور دستي تخصيص مي دهيد ، نياز به تخصيص همان تعداد عنصر براي هر يك از ابعاد نخواهيد داشت. همان گونه كه پيش از اين گفته شد ، از آنجا كه آرايه هاي چند بعدي درواقع آرايه هايي از آرايه ها هستند ، طول هر آرايه تحت كنترل شماست. به عنوان مثال ، برنامه زير يك آرايه دو بعدي ايجاد مي كند كه در آن اندازه هاي بعد دوم مساوي نيستند.

```
// 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 ذيل نشان داده شده است :

```
0
1 2
3 4 5
6 7 8 9
```

آرايه اي كه به وسيله برنامه ايجاد مي شود به صورت شكل زير است.

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

استفاده از آرايه هاي چند بعدي غير معمول (يا نامنظم) ممكن است براي بسياري از برنامه هاي كاربردي مناسب نباشد ، چرا كه وجود آرايه هاي چند بعدي در برنامه ها خلاف انتظار كاربران خواهد بود. با اين وجود ، آرايه هاي نامنظم ممكن است در برخي از شرايط به طور كارآمدي مورد استفاده قرار گيرند. به عنوان مثال ، اگر نياز به آرايه دو بعدي بسيار بزرگي داشته باشيد كه تنها برخي از عناصر آنها مقدار داشته باشند (يعني ، آرايه اي كه تمام عناصرش مورد استفاده نباشند) ، در آن صورت آرايه هاي نامنظم راه حل كاملي به شمار مي آيند.

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 هرعنصر، 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 خواهيد ديد:

```
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 ، مقدارمي گيرد.

اينك به بررسي چند مثال ديگر از كاربرد آرايه هاي چند بعدي مي پردازيم. در برنامه زير يك آرايه سه بعدي ۵×۴×۳ 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 اين روش ، كروشه ها پس از مشخصه نوع آرايه قرار مي گيرند و نه نام متغير آرايه.

```
int a1 = new int[3];
int a2 = new int[3];
```

تعاريف ذيل نيز معادل هستند:

```
char twod1[ ] [ ] = new char[3] [4];
char[ ] [ ] twod2 = new char[3] [4];
```

اين روش براي مواقعي مفيد است كه همزمان چندين آرايه همنوع تعريف مي شوند. مثلا :

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

عبارت بالا سه متغير آرايه اي از نوع int تعريف مي كند. نتيجه مثال بالا همچون عبارت زير است :

```
int name[ ], name2[ ], nume3[ ]; // create three arrays
```

اين روش همچنين براي مشخص كردن يك آرايه به عنوان مقدار حاصل از يك متد مفيد است.

چند نكته درباره رشته ها

همان گونه كه ممكن است متوجه شده باشيد ، در مباحث انواع داده ها و آرايه ها هيچ مطلبي درباره رشته ها يا داده هاي نوع string مطرح نشد. دليل اين مطلب آن است كه جاوا از چنين نوعي پشتيباني نمي كند- البته نه به شكل معمول. واقعيت امر آن است كه داده هاي رشته اي جاوا كهstring ناميده مي شوند ، يكي از انواع داده هاي پايه و ساده به شمار نمي آيند و همين طور، آرايه اي از كاراكترها نيز به شمار نمي آيند. بلكه در عوض شي مي باشند و ارائه شرح كاملي از اين نوع داده ها ، مستلزم آشنايي با برخي از ويژگي هاي مرتبط با شيءهاست. بدين ترتيب ، اين موضوع در آينده و پس از بررسي شي ها مورد بررسي قرار خواهد گرفت. با اين وجود ، براي آنكه بتوانيد در برنامه هاي نمونه از رشته هاي ساده استفاده كنيد، شرح خلاصه ذيل در اينجا 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 تخصيص داد. متغيرهاي نوع string را مي توان به ساير متغيرهاي نوع string تخصيص 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 اينجا شيئي از نوع string است كه رشته "this is a test" به آن تخصيص مي يابد. اين رشته به وسيله عبارت println() نمايش داده مي شود.

نكته 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 پشتيباني مي كنند. اما، در اينجا هيچ اشاره اي به نشانه روها نشده است. دليل اين امر ساده است : جاوا از نشانه روها پشتيباني نمي كند و استفاده از آنها مجاز نيست (يا به عبارت درست تر، جاوا از نشانه روهايي كه توسط برنامه سازان قابل دستيابي يا تغيير باشند ، پشتيباني نمي كند). جاوا نمي تواند از نشانه روها پشتيباني كند، چرا كه انجام اين كار به برنامه هاي جاوا امكان مي دهد تا شكافي بين محيط اجراي جاوا وكامپيوتر 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 عملگرهاي غني زيادي دارد. بيشتر عملگرهاي آن را مي توان به چهار گروه زير تقسيم كرد : حسابي ، بيتي ، رابطه اي و منطقي. جاوا همچنين چند عملگر اضافي دارد كه شرايط ويژه را مديريت مي كنند.
توجه : بيشتر عملگرها در جاوا همچون زبانهاي C/C++/C# كار ميكنند. اما تفاوتهاي جزئي هم وجود دارد.

عملگرهاي حسابي

عملگرهاي حسابي به همان صورتي كه در جبر به كار مي رود ، در جملات حسابي به كار برده مي شوند.

عملگر	نتيجه
+	جمع
-	تفريق
*	ضرب
/	تقسيم
%	باقيمانده تقسيم
++	افزايش
+=	جمع و تخصيص
-=	تفريق و تخصيص
*=	ضرب و تخصيص
/=	تقسيم و تخصيص
%=	باقيمانده و تخصيص
--	كاهش

عملوندهاي عملگرهاي حسابي بايد از نوع عددي باشند. آنها را نمي توان همراه با داده هاي نوع Boolean به كار برد ، اما مي توان بر روي داده هاي نوع char به كار برد ، چون char در جاوا اساسا زيرمجموعه int مي باشد.

عملگرهاي حسابي پايه

عملگرهاي حسابي پايه – جمع، تفريق، ضرب، تقسيم – همان گونه اي عمل مي كنند كه براي داده هاي عددي انتظار داريد. عملگر "-" يك فرم يگانگي دارد كه تنها عملوندش را منفي مي كند. به خاطر داشته باشيد كه وقتي عملگر تقسيم (/) با داده هاي صحيح به كار برده مي شود ، حاصل تقسيم فاقد بخش اعشاري خواهد بود.

برنامه ساده زير ، عملگرهاي حسابي را نشان مي دهد. اين برنامه همچنين تفاوت بين تقسيم اعداد صحيح و تقسيم اعداد اعشاري را نشان مي 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 زير 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 باقيمانده تقسيم

عملگر باقيمانده تقسيم ، % ، باقيمانده عمل تقسيم را برميگرداند. اين عملگر را مي توان هم براي اعداد اعشاري با ممیز شناور ، و هم براي انواع داده هاي صحيح به كار 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 mod 10 = 2
Y mod 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. عملكرد هر دو عبارت يكسان است : مقدار a را به اندازه ٤ 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.

١ – در حجم كار تايپ قدري صرفه جويي مي شود ، چرا كه "فرم كوتاه" معادل خود هستند.

٢ – 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 خواهيد ديد.

افزايش و كاهش

"++" و "--" ، عملگرهاي افزايش و كاهش هستند. همانگونه كه خواهيد ديد، اين عملگرها چند خصوصيت ويژه دارند كه آنها را كاملا جذاب ساخته اند. بحث خود را با مرور دقيق عملكرد اين دو عملگر آغاز مي كنيم.

عملگر افزايش ، يك واحد به عملوند خود مي افزايد. عملگر كاهش نيز يك واحد از عملوند خود مي كاهد. به عنوان مثال ، عبارت زير :

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

را مي توان با استفاده از عملگر افزايش به صورت ذيل بازنويسي كرد :

```
x++;
```

همچنين :

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

اين عملگرها از اين جهت كه مي توانند به صورت postfix (عملگر پس از عملوند) و همينطور prefix (عملوند پس از عملگر) باشند ، منحصر به فرد هستند. در مثالهاي بالا هيچ تفاوتي بين فرم هاي postfix و prefix وجود ندارد. اما ، وقتي عملگرهاي افزايش و يا كاهش بخشي از يك جمله جبري بزرگتر باشند، در آن صورت تفاوتي جزئي ، وليكن تاثيرگذار ، بين اين دو فرم وجود خواهد داشت. در فرم prefix ، پيش از آنكه از مقدار متغيير در جمله جبري استفاده شود ، عملوند افزايش يا كاهش مي يابد. در فرم postfix ، مقدار متغيير در جمله جبري به كار برده مي شود ، و سپس عملوند تغيير داده مي شود. به عنوان مثال :

```
x = 42;
y = ++x;
```

در اين حالت مقدار ٤٣ به y نخصيص مي يابد، چرا كه عمل افزايش پيش از تخصيص مقدار x به y انجام مي شود. از اين رو سطر y= ++x معادل دو عبارت زير مي باشد :

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

اما وقتي مثال بالا به صورت زير باشد :

```
x = 42;
y = x++;
```

مقدار x پيش از اجراي عملگر افزايش تخصيص مي يابد، بنابراين مقدار ٤٢ به y تخصيص مي يابد. البته ، در هر دو حالت مقدار ٤٣ در x ذخيره مي شود. در اينجا ، سطر y=x++ معادل دو عبارت ذيل است :

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

عملگرهاي بيتي

جاوا چندين عملگر بيتي دارد كه مي توان با انواع داده هاي صحيح long ، int ، short ، char و byte به كار برد. اين عملگر بر روي يكايك بيت هاي عملوند خود عمل مي كنند. فهرست اين عملگرها در جدول صفحه بعد آورده شده است.

عملگر	نتيجه
~	not يكاني بيتي
&	and بيتي
|	or بيتي
^	xor بيتي
>>	شيفت به راست
>>>	شيفت به راست با صفر اضافي
<<	شيفت به چپ
&=	and بيتي و تخصيص
!=	or بيتي و تخصيص
^=	xor بيتي و تخصيص
>>=	شيفت به راست و تخصيص
>>>=	شيفت به راست و تخصيص با صفر اضافي
<<=	شيفت به چپ و تخصيص

از آنجايي كه عملگرهاي بيتي ، بيت هاي اعداد صحيح را پردازش و مديريت مي كنند ، مهم است كه با تاثير اين چنين پردازشها بر روي مقادير آشنا باشيد. بخصوص اينكه ، دانستن اينكه جاوا چگونه مقادير صحيح را ذخيره مي كند و چگونه اعداد منفي را نمايش مي دهد، مفيد واقع مي شود.

تمام اعداد صحيح به وسيله اعداد بايناري با طول متغيير نمايش داده مي شوند. به عنوان مثال ، مقدار ٤٢ نوع byte ، 00101010 است كه ارزش مكاني هر رقم ، تواني از ٢ مي باشد.

انواع داده هاي صحيح (به غير از char) ، اعداد صحيح علامت دار هستند. اين بدين معناست كه مي توانند نمايانگر مقادير مثبت و منفي باشند. جاوا از نوعي روش رمزگذاري به نام " مكمل ۲ " استفاده مي كند كه در آن اعداد منفي از طريق معكوس كردن تمام بيت ها (تبديل يك ها به صفر و بلعكس) و سپس افزودن يك واحد به حاصل ، نمايش داده مي شوند. به عنوان مثال ، 42- با معكوس كردن تمام بيت ها عدد ٤٢ (يا 00101010) ، كه نتيجه 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 معكوس كنيد، و سپس يك واحد به نتيجه بيافزاييد.

عملگرهاي منطقي بيتي

عملگرهاي منطقي بيتي عبارتند از & ، | ، ^ و ~ . نتيجه اين عمليات در جدول زير نشان داده شده است. در بحث ذيل به خاطر داشته باشيد كه عملگرهاي بيتي به يكايك 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 (~) كه مكمل بيتي نيز ناميده مي شود، تمام بيت هاي عملوند خود را معكوس مي كند. به عنوان مثال ، عدد ٤٢ كه معادل باينري آن به شكل زير است :

```
00101010
```

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

```
11010101
```

AND بيتي

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

```
        00101010    42
&       00001111    15
-------------------------
        00001010    10
```

OR بيتي

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

```
        00101010    42
|       00001111    15
-------------------------
        00101111    47
```

XOR بيتي

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

```
        00101010    42
^       00001111    15
-------------------------
        00100101    37
```

به چگونگي تغيير بيت ها دقت كنيد. هرگاه عملوند دوم صفر باشد ، عملوند نخست تغيير نمي كند. اين ويژگي براي برخي از عمليات پردازش بيت ها مفيد واقع مي شود.

كاربرد عملگرهاي منطقي بيتي

برنامه زير عملگرهاي منطقي بيتي را نشان مي 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 اين مثال بيت هايي دارند كه نمايانگر هر چهار حالت ممكن براي دو رقم باينري هستند : ۱-۱ ، ۱-۰ ، ۰-۱ ، ۰-۰ . با نگاه كردن به نتايج c و d مي توانيد به چگونگي عملكرد | و & بر روي هر يك از بيت ها دريابيد. رشته آرايه اي كه binary نام دارد ، حاوي نمايش باينري اعداد صفر تا ۱۵ است. ايندكس هاي آرايه در اين مثال به گونه اي هستند تا نمايش باينري هر يك از نتايج را نشان دهند. آرايه به گونه اي ساخته شده است تا نمايش رشته اي مقدار باينري n ، در binary[n] ذخيره شود. مقدار a– با 0x0f (00001111) AND ، مي شود تا مقدارش به كمتر از ۱۶ كاهش يابد تا بتوان با استفاده از آرايه binary نمايش داد. خروجي برنامه در ذيل نشان داده شده است :

```
a = 0011
```

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

شيفت به چپ

عملكرد شيفت به چپ (<<) ، تمام بيت هاي عملوندش را به اندازه تعداد دفعات مشخص شده به سمت چپ انتقال مي دهد. شكل كلي آن در زير نشان داده شده است :

```
Value << num
```

Num مشخص كننده تعداد دفعاتي است كه مقدار value بايد به چپ انتقال يابد. يعني ، "<<" تمام بيت هاي مقدار مورد نظر را به اندازه num مرتبه به سمت چپ انتقال مي دهد. به ازاي هر بار انتقال (شيفت) ، بيت منتهي اليه سمت چپ از دست مي رود و يك بين صفر از سمت راست اضافه مي شود. اين بدين معناست كه وقتي عملگر "شيفت به چپ" به يك عملوند int اعمال مي شود، بيت ها پس از گذشت از موقعيت سي و يكم از دست مي روند. چنانچه عملوند از نوع long باشد، در آن صورت بيت ها پس از گذشت از موقعيت شصت و سوم از دست مي روند.

وقتي مقادير نوع byte و short را شيفت مي دهيد ، عمل ارتقا خودكار جاوا منجر به توليد نتايج غيرمنتظره مي شود. همان گونه كه مي دانيد ، وقتي يك جمله جبري ارزيابي مي شود، مقادير نوع byte و short به int ارتقا مي يابند. به علاوه نتيجه يك چنين جمله اي نيز int مي شود. اين بدين معناست كه نتيجه شيفت به چپ در مقادير byte ، short و int مي شود و بيت هاي شيفت شده هم تا زماني از موقعيت سي و يكم نگذرند ، از دست نمي روند. به علاوه ، مقادير byte و short منفي نيز هنگام ارتقا به int از سمت بيت علامتشان بزرگ مي شوند. از اين رو، بيت هاي سمت چپ با يك پر مي شوند. به همين دليل است كه انجام عمل شيفت به چپ در مقادير نوع byte و short بدين معناست كه بايد بايت هاي سمت چپ نتيجه نوع int را ناديده بگيرد. به عنوان مثال ، اگر يك مقدار نوع byte را به سمت چپ شيفت بدهيد ، آن مقدار ابتدا به int ارتقا مي يابد و سپس شيفت داده مي شود. اين بدين معناست كه اگر منظورتان نتيجه "شيفت به چپ" مقدار byte باشد،

باید سه بایت سمت چپ نتیجه را نادیده بگیرید. آسانترین روش براي انجام این کار ، آن است که نتیجه را مجددا با استفاده از "casting" به byte تبدیل کنید. این مفهموم 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);
}
}
```

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

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

چون a براي ارزیابي به int ارتقا مي یابد ، دو مرتبه "شیفت به چپ" مقدار ٦٤ (0000 0000) منجر به مقدار ٢٥٦ (0000 0000 1) براي i مي شود. اما ، مقدار b پس از شیفت صفر مي شود، چون بایت سمت چپ صفر مي شود. تنها بیت یك 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 یك جایگزین کارآمدتر براي ضریب در ٢ استفاده مي کنند. اما باید مواظب هم باشید. اگر یك بیت "یك" را به موقعیت آخرین بیت سمت چپ (بیت ٣١ یا ٦٣) انتقال دهید، مقدارتان منفي 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 = 0xFFFFFFE;
for(i=0; i<4; i++ ){
num = num << 1;
System.out.println(num);
}
```

```
}
}
```

خروجي برنامه به صورت زير است :

```
536870908
1073741816
2147483632
-32
```

مقدار اوليه به دقت انتخاب شده است تا پس از ٤ مرتبه "شيفت به چپ" به ٣٢- تبديل شود. همان گونه كه مي بينيد وقتي يك بيت "يك" به موقعيت ٣١ انتقال مي يابد ، عدد حاصل منفي مي شود.

شيفت به راست

عملگر شيفت به راست (>>) ، تمام بيتهاي عملوندش را به اندازه تعداد دفعات مشخص شده به سمت راست انتقال مي دهد. شكل كلي آن در زير نشان داده شده است :

```
value >> num
```

num مشخص كننده تعداد دفعاتي است كه مقدار value بايد به سمت راست انتقال يابد. يعني ، ">>" تمام بيت هاي مقدار مورد نظر را به اندازه num مرتبه به سمت راست انتقال مي دهد.

مثال زير مقدار ٣٢ را ٢ مرتبه به سمت راست شيفت مي دهد تا مقدار ٨ در a ذخيره شود :

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

وقتي بيت هاي يك مقدار به خارج از محدوده اش شيفت داده شوند، تمام آن بيت ها از دست مي روند. مثلا ، مثال زير مقدار ٣٥ را ٢ مرتبه به راست شيفت مي دهد و در نتيجه با از دست رفتن بيت هاي سمت راست ، مجددا مقدار ٨ در a ذخيره مي شود.

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

بررسي همان عمل با اعداد باينري ، اتفاقات را به وضوح نشان مي دهد :

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

هر بار كه مقداري را به راست شيفت مي دهيد ، آن مقدار بر ٢ تقسيم مي شود – و باقيمانده ناديده انگاشته مي شود. از اين ويژگي مي توانيد براي انجام عمل تقسيم بر ٢ به شكل كارآمدتر استفاده كنيد. البته ، بايد اطمينان حاصل كنيد كه بيت ها از سمت راست خارج نمي شوند.

وقتي مقداري را به سمت راست شيفت مي دهيد، سمت چپ ترين بيت ها كه توسط عمل شيفت راست نمايان شده اند، به وسيله محتواي پيشين سمت چپ ترين بيت جايگزين مي شوند. اين عمل "sign extension" ناميده شده و سبب حفظ علامت اعداد منفي به هنگام شيفت دادن آنها به سمت راست مي شود. به عنوان مثال ، 1>>8- ، برابر 4- است كه بايناري اين عمل در ذيل نشان داده شده است :

```
11111000          -8
>>1
11111100          -4
```

جالب است بدانيد كه اگر ١- را به سمت راست شيفت دهيد ، نتيجه هميشه ١- باقي مي ماند، چرا كه فرآيند "sign extension" سبب مي شود تا همواره بيت هاي يك در سمت چپ جايگزين شوند.

گاهي اوقات انجام فرآيند، پيش گفته به هنگام شيفت دادن مقادير به سمت راست ضرورت ندارد. به عنوان مثال ، برنامه زير يك مقدار نوع byte را به رشته هگزادسيمال معادلش تبديل مي كند. دقت كنيد كه مقدار حاصل از شيفت ، از طريق AND كردن با 0x0f ماسك مي شود تا نتيجه "sign extension" ناديده انگاشته شود و در نتيجه ، بتوان از مقدار حاصل به عنوان ايندكس آرايه كاراكترهاي هگزادسيمال 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 )oxf1
System.out.println("b = ox" + hex[(b >> 4 )& oxof] + hex[b & oxof]);
}
}
```

خروجي برنامه در زير نشان داده شده است :

```
b = 0xf1
```

شيفت به راست بدون علامت

همان گونه كه در قسمت پيش ديديد، عملگر >> ، بيت منتهي اليه سمت چپ را به طور خودكار هر بار پس از شيفت با مقدار پيشين آن پر مي كند. اين كار موجب حفظ علامت مقدار مورد نظر مي شود. اما، گاهي اين امر لازم نمي شود. به عنوان مثال ، اگر چيزي را شيفت مي دهيد كه نمايانگر يك مقدار عددي نيست ، ممكن است نياز به "sign extension" نداشته باشيد. اين شرايط هنگام كار با مقادير مبتني بر پيكسل و گرافيك متداول است. در اين گونه موارد ، عموما بايد يك صفر را به بيت منتي اليه سمت چپ انتقال مي دهد.

عملكرد ">>>" در مثال زير نشان داده شده است. در اين مثال ، a با ١- مقداردهي شده و بنابراين ، كل ٣٢ بيت با يك پر مي شود. اين مقدار سپس ٢٤ مرتبه به سمت راست انتقال يافته و ٢٤ بيت سمت چپ با صفر پر شده و عمل sign extension ناديده انگاشته مي شود. بدين ترتيب ٢٥٥ در a ذخيره ميشود.

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

انجام عمل بالا در ذيل با ارقام باينري تكرار شده است تا بيشتر با اتفاقاتي كه رخ مي دهند آشنا شويد :

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

عملكرد >>> اغلب به اندازه اي كه دوست داريد مفيد واقع نمي شود ، چراكه تنها براي مقادير ٣٢ و ٦٤ بيتي معنا خواهد داشت. به خاطر داشته باشيد كه مقادير كوچكتر در جملات جبري به طور خودكار به int ارتقا مي يابند. اين بدين معناست كه sign-extension رخ مي دهد و عمل شيفت به جاي مقادير ٨ يا ١٦ بيتي ، در مقدار ٣٢ بيتي انجام مي شود. يعني ، ممكن است برخي

افراد انتظار داشته باشند كه عمل شيفت به راست بدون علامت در مقادير نوع byte به گونه اي انجام شود تا مقداردهي بيت ها با صفر ، از بيت موقعيت ۷ آغاز شود. اما اين گونه نيست ، چراكه در واقع يك مقدار ۳۲ بيتي شيفت داده مي شود. برنامه زير نشانگر اين 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 )oxf1
byte c =( byte( )b >> 4);
byte d =( byte( )b >>> 4);
byte e =( byte(( )b & oxff )>> 4);
System.out.println(" b = ox"
+ hex[(b >> 4 )& oxof] + hex[b & oxof]);
System.out.println(" b >> 4 = ox"
+ hex[(c >> 4 )& oxof] + hex[c & oxof]);
System.out.println(" b >>> 4 = ox"
+ hex[(d >> 4 )& oxof] + hex[d & oxof]);
System.out.println("(b & oxof )>> 4 = ox"
+ hex[(e >> 4 )& oxof] + hex[e & oxof]);
}
}
```

خروجي اين برنامه نشان مي دهد كه چگونه عملگر >>> در حين كار با مقادير نوع byte ، به ظاهر كاري انجام نمي دهد. در اينجا ، متغيير b با يك مقدار دلخواه نوع byte مقداردهي شده است. حاصل شيفت به راست b (به اندازه ۳ بيت)، به خاطر انجام عمل مورد انتظار sign extension در c ذخيره مي شود (يعني 0xff). سپس حاصل همان عمل شيفت، البته بدون علامت، كه انتظار مي رود 0x0f باشد، به d تخصيص مي يابد. اما در حقيقت چون عمل sign extension هنگام ارتقا b به int (پيش از عمل شيفت) رخ داده است، 0xff در d ذخيره مي شود. در جمله آخر نيز مقدار نوع byte متغيير b از طريق AND شدن با ۸ بيت ماسك شده، چهار مرتبه به راست شيفت داده ميشود، و حاصل آنكه بايد 0x0f باشد، در c ذخيره مي شود.

توجه داشته باشيد كه براي متغيير d از عملگر >>> استفاده نشده است، چرا كه وضعيت بيت علامت پس از AND كردن مشخص مي شود.

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

عملگرهاي بيتي و تخصيص

تمام عملگرهاي بيتي باينري فرم كوتاهي دارند كه مشابه عملگرهاي جبري است كه عمل تخصيص را با عمل بيتي تركيب مي كنند. به عنوان مثال، دو عبارت زير كه مقدار a را به اندازه ٤ بيت به سمت راست شيفت مي دهند، معادل يكديگرند :

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

همين طور دو عبارت زير هم كه جمله بيتي a OR b به a تخصيص مي يابد، معادل يكديگرند :

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

در برنامه زير ٤ فيلد صحيح ايجاد شده و سپس فرم كوتاه عملگرهاي بيتي و تخصيص براي پردازش و مديريت آنها استفاده مي 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 ذيل 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 ذيل نشان داده شده است :

عملگر	نتيجه
==	مساوي است با
=!	مخالف است با
>	بزرگتر از
<	كوچكتر از
>=	بزرگتر يا مساوي است با
<=	كوچكتر 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 مميز شناور ، كاراكترها 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 مميز شناور و كاراكتري را مي توان مقايسه نمود تا مشخص شود كه كداميك بزرگتر و كداميك كوچكترند.

همان گونه كه گفته شد، نتيجه حاصل از عملگر رابطه اي ، يك مقدار بولي است. مثلا عبارت ذيل كاملا معتبر هستند :

```
int a = 4;
int b = 1;
```

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

در اين حالت نتيجه a<b (كه false است) در c ذخيره مي شود.

اگر تجربه كار با C/C++ را داريد ، لطفا به نكته زير دقت كنيد. انواع عبارت زير در C/C++ بسيار متداولند :

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

عبارت بالا را بايد در جاوا به صورت زير نوشت :

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

دليل اين امر آن است كه تعريف true و false در جاوا همچون C/C++ نيست. True در C/C++ ، هر مقدار مخالف صفر است و false نشانگر مقدار صفر. True و false در جاوا مقادير غير عددي هستند كه ارتباطي با صفر و غير صفر ندارند. بنابراين ، براي مقايسه با صفر و غير صفر ، بايد از يك يا چند عملگر رابطه اي استفاده كنيد.

عملگرهاي منطقي بولي

عملگرهاي منطقي بولي كه در اينجا نشان داده شده اند، تنها براي عملوندهاي نوع Boolean قابل استفاده هستند. تمام عملگرهاي منطقي بولي دو مقدار نوع Boolean را با هم تركيب مي كنند تا نتيجه يك مقدار نوع Boolean شود.

عملگر	نتيجه
&	AND منطقي
|	OR منطقي
^	XOR منطقي
||	OR اتصال كوتاه
&&	AND اتصال كوتاه
!	NOT يگاني منطقي
&=	AND و تخصيص
|=	OR و تخصيص

^=	XOR و تخصيص
==	مساوي است با
!=	مخالف است با
?:	if-then-else سه تايي

عملكرد عملگرهاي بولي منطقي (& ، | و ^) بر روي مقادير بولي همچون عملكردشان بر روي بيت هاي يك عدد صحيح است. عملكرد منطقي "!" وضعيت بولي را معكوس مي كند : !true == false و !false == true . نتيجه هر يك از عمليات منطقي در جدول زير 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 زير تقريبا معادل مثال BitLogic است كه پيش از اين مطرح شده است. با اين تفاوت كه به جاي بيت هاي باينري بر روي مقادير بولي عمل مي كند :

```
// 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 اجراي اين برنامه ، خواهيد ديد كه همان قوانين منطقي بيت ها ، به مقادير نوع Boolean نيز اعمال مي شوند. همان گونه كه از خروجي ذيل مشخص است ، نمايش رشته اي مقادير بولي جاوا ،يكي از دو مقدار ليترال true يا false است :

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

عملگرهاي منطقي اتصال كوتاه

جاوا دو عملگر بولي جالب دارد كه در بسياري از زبانهاي كامپيوتري ديگر وجود ندارد. اين عملگرها ، نگارشهاي ثانويه دو عملگر AND و OR بولي هستند و تحت عناون عملگرهاي منطقي "اتصال كوتاه" شناخته شده اند. همان گونه كه از جدول قسمت پيش مشخص است، نتيجه عملگر OR زماني true است كه A مقدارش true باشد ، صرفنظر از مقدار B. همين طور ، حاصل عملگر AND زماني false مي شود كه A مقدارش false باشد ، صرفنظر از مقدار B. اگر به جاي فرم هاي | و & اين عملگرها از دو فرم || و && استفاده كنيد ، در آن صورت چنانچه جاوا بتواند نتيجه جمله شرطي را تنها به وسيله عملوند سمت چپ تعيين كند ، در آن صورت ديگر عملوند سمت راست را ارزيابي نخواهد كرد. اين امر براي مواقعي كه عملوند سمت راست به true يا false بودن عملوند سمت چپ وابسته باشد ، بسيار مفيد است. به عنوان مثال ، عبارت زير نشان مي دهد كه چگونه با استفاده از ارزيابي منطقي اتصال كوتاه مي توانيد پيش از يك عملوند تقسيم اطمينان حاصل كنيد كه حاصل آن درست خواهد بود يا خير :

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

چون ار فرم اتصال كوتاه AND (&&) استفاده شده است ، احتمال بروز استثناي "زمان-اجرا" به دليل صفر بودن denom از بين مي رود. اگر اين سطر با استفاده از نگارش & عملگر AND

نوشته مي شد ، مي بايست هر دو طرف عملگر منطقي ارزيابي مي شد و در نتيجه ، استثناي "زمان-اجرا" در صورت صفر بودن denom پيش مي آيد.
خوب است در مواقعي كه با منطق بولي سر و كار داريد ، از فرم هاي اتصال كوتاه AND و OR استفاده كنيد ، و نگارشهاي تك نمادي را صرفا براي عمليا بيتي باقي بگذاريد. اما ، اين قانون استثنا هم دارد. به عنوان مثال ، عبارت زير را در نظر بگيريد :

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

استفاده از & در اينجا تضمين مي كند كه افزايش مقدار e صرفنظر از يك بودن مقدار c انجام مي شود.

عملگر تخصيص

عملگر تخصيص با علامت "=" نشان داده مي شود. عملكرد عملگر تخصيص در جاوا همچون ساير زبانهاي كامپيوتري است. شكل كلي آن به صورت زير است :

```
Var = expression ;
```

نوع داده var بايد با نوع expression سازگار باشد.
عملكرد تخصيص ويژگي جالبي دارد كه ممكن است با آن آشنا نباشيد : امكان ايجاد زنجيره اي از عمليات تخصيص را فراهم مي سازد. به عنوان مثال ، به عبارت زير توجه كنيد :

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

در عبارت بالا تنها با استفاده از يك عبارت مقدار هر سه متغيير ۱۰۰ ميشود. دليل ميسر بودن اين ويژگي آن است كه "=" عملگري است كه نتيجه آن ، سمت راست عبارت است. از اين رو ، مقدار z=100 ، ۱۰۰ است كه به y تخصيص مي يابد ، و آن نيز به نوبه خود به x تخصيص مي يابد. استفاده از "زنجير تخصيص" روش آساني براي نسبت دادن يك مقدار مشترك به گروهي از متغيير هاست.

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 : experssion3
```

experssion1 مي تواند هر جمله اي با حاصل نوع Boolean باشد. اگر حاصل آن true باشد ، در آن صورت expersion2 ارزيابي مي شود ، در غير اين صورت experssion3 ارزيابي خواهد شد. نتيجه عملگر ؟ ، همان نتيجه ارزيابي جملات آن است. به مثالي از كاربرد ؟ 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 اين برنامه بصورت زير مي باشد :

```
Absolute value of 10 is 10
Absolute value of- 10 is 10
```

تقدم عملگرها

جدول زير ترتيب حق تقدم عملگرهاي جاوا را از بالاترين اولويت تا پايين ترين نشان مي دهد. دقت كنيد كه در سطر اول اقلامي وجود دارد كه معمولا بعنوان عملگر درباره آنها فكر نميكنيد : پرانتزها ، كروشه ها و عملگر نقطه (.) ، اين موارد جداساز ناميده مي شوند ، اما عملكرد آنها در جملات جبري همچون عملگرهاست. پرانتزها براي تغيير تقدم يك عمل به كار برده مي شوند. كروشه امكان مشخص كردن ايندكس آرايه ها را فراهم مي سازد. از عملگر نقطه براي ارجاع به شي ها استفاده ميشود.

بالاترين			
()	[]		
++	--	~	~
*	/	%	
+	-		
>>	>>>	<<	
>	=>	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
پايين ترين			

كاربرد پرانتزها

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

```
a >> 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 >>( 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 << 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 جلوگيري نمايد. بعنوان مثال ، كداميك از عبارات زير راحت تر خوانده و درك مي شوند ؟

```
a | 4 + c >> b & 7
( a |(( 4 + c )>> b )& 7 ))
```

يك نكته ديگر : پرانتزها (بطور كلي خواه اضافي باشند يا نه) سطح عملكرد برنامه شما را كاهش نمي دهند. بنابراين ، اضافه كردن پرانتزها براي كاهش ابهام تاثيري 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 كنترلي

هر زبان برنامه سازي از عبارات كنترلي براي هدايت جريان اجراي برنامه بر اساس تغييرات وضعيت آن استفاده مي كند. عبارات كنترلي برنامه هاي جاوا را مي توان به دسته هاي ذيل طبقه بندي نمود : انتخاب ، اكرار و پرش. عبارات انتخاب ، به برنامه هايتان امكان مي دهند تا مسيرهاي اجرايي مختلفي را بر اساس نتيجه يك جمله جبري يا وضعيت يك متغيير انتخاب كنند. عبارات تكرار ، به برنامه ها امكان مي دهند تا يك يا چند عبارت را تكرار كنند (يعني ، عبارات تكرار ، حلقه ها را تشكيل مي دهند.) عبارات پرش ، به برنامه هايتان امكان مي دهند تا به صورت غير خطي اجرا شوند.

توجه : عبارات كنترلي 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 پشتيباني مي كند : if و switch. با اين دستورات شما اجراي برنامه را براساس شرايطي كه فقط حين اجراي برنامه اتفاق مي افتند كنترل مي كنيد. 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. شكل كلي اين دستور بصورت زير است :

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

دراينجا هر statement ممكن است يك دستور منفرد يا يك دستور مركب قرار گرفته در آكولاد (يعني يك بلوك) باشد. condition (شرط) هر عبارتي است كه يك مقدار boolean را برمي گرداند . جمله else اختياري است.

if بصورت زير كار مي كند : اگر شرايط محقق باشد (حاصل condition ، مقدار true باشد.) آنگاه statement1 اجرا مي شود. در غير اينصورت statement2 (در صورت وجود) اجرا خواهد شد. تحت هيچ شرايطي هر دو دستور با هم اجرا نخواهند شد. بعنوان مثال ، در نظر بگيريد :

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

اگر a كوچكتر از b باشد ، آنگاه a برابر صفر مي شود. در غير اينصورت b برابر صفر خواهد شد. در هيچ شرايطي اين دو متغير در آن واحد برابر صفر نمي شوند.

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

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

به خاطر داشته باشيد كه فقط يك دستور مي تواند مستقيما بعد از if يا else قرار گيرد. اگر بخواهيد دستورات بيشتري داخل نماييد ، نيازي به ايجاد يك بلوك نداريد ، نظير اين قطعه كه در زير آمده است :

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

```
} else
  waitForMoreData();
```

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

برخي از برنامه نويسان استفاده از آكولاد را هنگام استفاده از if ، حتي زمانيكه فقط يك دستور در هر جمله وجود داشته باشد را مناسب ميدانند. اين امر سبب مي شود تا بعدا بتوان براحتي دستور ديگري را اضافه نمود و نگراني از فراموش كردن آكولادها نخواهيد داشت. در حقيقت ، فراموش كردن تعريف يك بلوك هنگامي كه نياز است ، يكي از دلايل رايج بروز خطاها ميباشد. بعنوان مثال قطعه زير از يك كد را در نظر بگيريد :

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

بنظر خيلي روشن است كه دستور bytesAvailable = n; طوري طراحي شده تا داخل جمله else اجرا گردد ، و اين بخاطر سطح طراحي آن است. اما حتما بياد داريد كه فضاي خالي براي جاوا اهميتي ندارد و راهي وجود ندارد كه كامپايلر بفهمد چه مقصودي وجود دارد. اين كد بدون مشكل كامپايل خواهد شد ، اما هنگام اجرا بطور ناصحيح اجرا خواهد شد. مثال بالا به صورت زير اصلاح ميشود :

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

```
bytesAvailable = n;
}
```

if هاي تو در تو

منظور از if تو در تو ، آن است كه عبارت if ديگري ، در يك if يا else ديگر قرار مي گيرد. if هاي تودرتو در برنامه سازي بسيار متداولند. وقتي if ها را به صورت تو در تو مي نويسيد ، مطلب اصلي كه بايد به خاطر بسپاريد آن است كه هر عبارت else هميشه به نزديكترين عبارت if موجود در همان بلوك else كه با else ديگري هم مرتبط نيست ، ارجاع دارد. به مثال زير توجه كنيد :

```
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 كه توضيحات نشان مي دهند ، else نهايي با if(j<20) مرتبط نيست ، چراكه در همان بلوك قرار ندارد (با وجود آنكه نزديكترين if بدون else است). بلكه در عوض ، آخرين else با if(i==10) مرتبط است. else داخلي به if(k>100) ارجاع دارد ، چرا كه نزديكترين if در همان بلوك است.

نردبان if-else-if

يك ساختار برنامه نويسي رايج براساس يك ترتيب از if هاي تودرتو بنا شده است. اين ساختار نردبان if-else-if است و بصورت زير مي باشد :

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

```
  .
  .
  .
else
  statement;
```

دستورات if از بالا به پايين اجرا مي شوند. ماداميكه يكي از شرايط كنترل كننده if صحيح باشد (true) ، دستور همراه با آن if اجرا مي شود ، و بقيه نردبان رد خواهد شد. اگر هيچكدام از شرايط صحيح نباشند ، آنگاه دستور else نهايي اجرا خواهد شد. else نهايي بعنوان شرط پيش فرض عمل مي كند ، يعني اگر كليه شرايط ديگر صحيح نباشند ، آنگاه آخرين دستور else انجام خواهد شد. اگر else نهايي وجود نداشته باشد و ساير شرايط ناصحيح باشند ، آنگاه هيچ عملي انجام نخواهد گرفت.

در زير ، برنامه اي را مشاهده مي كنيد كه از نردبان if-else-if استفاده كرده تا تعيين كند كه يك ماه مشخص در كدام 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 + ".");
  }
}
```

خروجي اين برنامه بقرار زير مي باشد :

```
April is in the Spring.
```

ممكن است بخواهيد اين برنامه را آزمايش كنيد. خواهيد ديد كه هيچ فرقي ندارد كه چه مقداري به month بدهيد ، يك و فقط يك دستور انتساب داخل نردبان اجرا خواهد شد.

Switch

دستور switch ، دستور انشعاب چند راهه در جاوا است. اين دستور راه ساده اي است براي تغيير مسير اجراي بخشهاي مختلف يك كد براساس مقدار يك عبارت مي باشد. اين روش يك جايگزين مناسب تر براي مجموعه هاي بزرگتر از دستورات if-else-if است. شكل كلي دستور switchبه قرار زير مي باشد :

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

expression بايد از نوع byte ، short ، int يا char باشد ، هر يك از مقادير values در دستورات case بايد از نوع سازگار با expression باشند. هر يك از مقادير case بايد يك ليترال منحصر بفرد باشد (يعني بايد يك ثابت ، نه متغير ، باشد.) مقادير case نبايد تكراري باشند.
دستور switch بشرح فوق عمل مي كند : مقدار expression با هر يك از مقادير ليترال عبارت case مقايسه مي شوند. اگر مقدار متناظري پيدا شود ، كد سلسله اي تعقيب كننده

آن دستور case اجرا خواهد شد. اگر هيچيك از ثابت ها با مقدار expression برابر نباشند ، آنگاه دستور پيش فرض (default) اجرا خواهد شد ، اما دستور default اختياري است. اگر هيچيك از case ها تطابق نيابد و default وجود نداشته باشد آنگاه عمل اضافي ديگري انجام نخواهد شد.

از عبارت break داخل دستور switch استفاده شده تا سلسله يك دستور را پايان دهد. هنگاميكه با يك دستور break مواجه مي شويم ، اجرا به خط اول برنامه كه بعد از كل دستورswitch قرار گرفته ، منشعب خواهد شد. بدين ترتيب كنترل از عبارت switch خارج ميشود.

در زير مثال ساده اي را مشاهده مي كنيد كه 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 then 3.");
      }
  }
}
```

خروجي اين برنامه بقرار زير مي باشد :

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

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

همانطوريكه مشاهده مي كنيد ، داخل حلقه ، دستوراتي كه همراه ثابت case بوده و با i مطابقت داشته باشند ، اجرا خواهند شد. ساير دستورات پشت سر گذاشته مي شوند. بعد از اينكه i بزرگتر از ۳ بشود ، هيچ يك از مقادير case برابر مقدار expression نخواهد بود ، و بنابراين عبارت default اجرا مي شود.

دستور default اختياري است. اگر break را حذف كنيد ، اجراي برنامه با case بعدي ادامه خواهد يافت. گاهي بهتر است چندين case بدون دستورات break در بين آنها داشته باشيم. بعنوان مثال ، برنامه بعدي را در نظر بگيريد :

```
//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:٨
        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 than۰
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 10 or more
i is 10 or more
```

همانطوريكه مشاهده مي كنيد، اجرا طي هر case ، به محض رسيدن به يك دستور break (يا انتهاي switch) متوقف مي شود.

در حاليكه مثال قبلي براي توصيف نظر خاصي طراحي شده بود ، اما بهر حال حذف دستور break كاربردهاي عملي زيادي در برنامه هاي واقعي دارد. براي نشان دادن كاربردهاي واقعي تر اين موضوع ، دوباره نويسي برنامه نمونه مربوط به فصول سال را مشاهده نماييد. اين روايت جديد همان برنامه قبلي از switch استفاده مي كند تا پياده سازي موثرتري 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

مي توانيد از يك switch بعنوان بخشي از ترتيب يك دستور switch خارجي تر استفاده نماييد. اين حالت را switch تودرتو مي نامند. از آنجاييكه دستور switch تعريف كننده بلوك مربوط به خودش مي باشد، هيچ تلاقي بين ثابتهاي case در switch داخلي و آنهايي كه در switch خارجي قرار گرفته اند ، بوجود نخواهد آمد . بعنوان مثال ، قطعه بعدي كاملا 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 اينجا دستور case 1 در switch داخلي با دستور case 1 در switch خارجي تلاقي نخواهد داشت. متغير count فقط با فهرست case ها در سطح خارجي مقايسه مي شود. اگر count برابر ۱ باشد، آنگاه target با فهرست case هاي داخلي مقايسه خواهد شد. به طور خلاصه ، سه جنبه مهم از دستور switch قابل توجه هستند :

۱ - switch با if متفاوت است چون switch فقط آزمايش كيفيت انجام مي دهد ، در حاليكه if هر نوع عبارت بولي را ارزيابي مي كند. يعني كه switch فقط بدنبال يك تطابق بين مقدار عبارت و يكي از ثابت هاي case خودش مي گردد .

۲ - دو ثابت case در switch نمي توانند مقادير يكسان داشته باشند . البته ، يك دستورswitch قرار گرفته داخل يك switch خارجي تر مي تواند ثابتهاي case مشترك داشته باشد.

۳ - يك دستور switch معمولا بسيار كارآمدتر از يك مجموعه از if هاي تودرتو شده است.

آخرين نكته بخصوص جالب توجه است زيرا روشنگر نحوه كار كامپايلر جاوا مي باشد. كامپايلر جاوا هنگاميكه يك دستور switch را كامپايل مي كند ، به هر يك از ثابتهاي case سركشي نموده و يك جدول jump table مي سازد كه براي انتخاب مسير اجرا براساس مقدار موجود در عبارت استفاده مي شود. بنابراين ، اگر بايد از ميان گروه بزرگي از مقادير انتخاب نماييد ، يك دستور switchنسبت به يك ترتيب از if-else ها كه بطور معادل و منطقي كد بندي شده باشد ، بسيار سريعتر اجرا خواهد شد. كامپايلر قادر است اينكار را انجام دهد چون مي داند كه ثابتهاي caseهمه از يك نوع بوده و بايد خيلي ساده با عبارت switch براي كيفيت مقايسه شوند. كامپايلر چنين شناسايي را نسبت به يك فهرست طولاني از عبارات if ندارد.

دستورات تكرار (Iteration Statements)

دستورات تكرار در جاوا عبارتند از for ، while وdo-while. اين دستورات آن چه را ما حلقه مي ناميم ، ايجاد مي كنند. احتمالا مي دانيد كه حلقه يك مجموعه از دستورالعملها را بطور تكراري

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

While

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

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

شرط يا condition ممكن است هر عبارت بولي باشد. مادامیكه 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 تكرار باشد ، استفاده از ابروها غير ضروري است .

در اينجا يك حلقه while وجود دارد كه تا ۱۰ را محاسبه كرده و دقيقا 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 آنجاييكه حلقه while عبارت شرطي خود را در بالاي حلقه ارزيابي ميكند ، اگر شرط ابتدايي ناصحيح باشد ، بدنه حلقه اجرا نخواهد شد. بعنوان مثال ، در قطعه زير ، فراخواني println() هرگز اجرا نخواهد شد.

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

بدنه while يا هر حلقه ديگر در جاوا ممكن است تهي باشد. زيرا دستور تهي دستوري كه فقط شامل ; باشد ، از نظر قواعد تركيبي در جاوا معتبراست. بعنوان مثال ، برنامه زير را در نظر بگيريد :

```
//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 );
}
}
```

اين برنامه نقطه مياني (midpoint) بين i و j را پيدا مي كند و خروجي زير را توليد خواهد كرد :

```
Midpoint is  150
```

در اينجا چگونگي كار حلقه while را مي بينيد. مقدار i افزايش و مقدار j كاهش مي يابد. سپس اين دو مقدار با يكديگر مقايسه مي شوند. اگر مقدار جديد i همچنان كمتر از مقدار جديد j باشد ، آنگاه حلقه تكرار خواهد شد . اگر i مساوي با يا بزرگتر از j بشود ، حلقه متوقف خواهد شد. تا

هنگام خروج از حلقه ، i مقداري را مي گيرد كه بين مقادير اوليه i و j مي باشد. (بديهي است كه اين رويه هنگامي كار مي كند كه i كوچكتر از مقدار اوليه j باشد.) همانطوريكه مي بينيد ، نيازي به بدنه حلقه نيست ، كليه عمليات داخل خود عبارت شرطي اتفاق مي افتد. در كدهاي حرفه اي نوشته شده ديگر جاوا ، وقتي كه عبارت كنترل كننده توانايي مديريت كليه جزئيات خود را داشته باشد ، حلقه هاي كوتاه غالبا بدون بدنه كد بندي مي شوند.

Do-While

گفتيم اگر عبارت شرطي كنترل كننده يك حلقه while در ابتدا ناصحيح باشد آنگاه بدنه حلقه اصلا اجرا نمي شود. اما گاهي مايليم در چنين شرايطي ، بدنه حلقه حداقل يكبار اجرا شود. بعبارت ديگر ، در حالات خاصي مايليد تا عبارت پايان دهنده در انتهاي حلقه را آزمايش كنيد. خوشبختانه ، جاوا حلقه اي را عرضه مي كند كه دقيقا همين كار را انجام مي دهد. حلقه do-while همواره حداقل يكبار بدنه خود را اجرا مي كند ، زيرا عبارت شرطي آن در انتهاي حلقه قرار گرفته است. شكل كلي آن بصورت زير است :

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

هر تكرار از حلقه do-while ابتدا بدنه حلقه را اجرا نموده ، سپس به ارزيابي عبارت شرطي خود مي پردازد. اگر اين عبارت صحيح (true) باشد ، حلقه اجرا خواهد شد. درغيراينصورت حلقه پايان مي گيرد. نظير كليه حلقه هاي جاوا ، شرط بايد يك عبارت بولي باشد. اينجا يك روايت ديگر از برنامه (tick) وجود دارد كه حلقه do-while را نشان مي دهد. خروجي اين برنامه مشابه برنامه قبلي 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 تكنيكي صحيح است ، اما مي توان آن را به شكل كاراتري بصورت زير دوباره نويسي نمود :

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

در اين مثال ، عبارت (--n>0) ، عمل كاهش n و آزمايش براي صفر را در يك عبارت گنجانده است. عملكرد آن بقرار بعدي است. ابتدا دستور n اجرا مي شود و n را كاهش داده و مقدار جديد را به n برمي گرداند. اين مقدار سپس با صفر مقايسه مي شود. اگر بزرگتر از صفر باشد ، حلقه ادامه مي يابد. در غير اينصورت حلقه پايان مي گيرد.

حلقه do-while بويژه هنگام پردازش انتخاب منو بسيار سودمند است ، زيرا معمولا مايليد تا بدنه يك حلقه منو حداقل يكبار اجرا شود. برنامه بعدي را كه يك سيستم Help ساده را براي دستورات تكرار و انتخاب در جاوا پياده سازي مي كند در نظر بگيريد :

```
//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.prinln("Help on:”);
System.out.prinln(" 1 .if”);
System.out.prinln(" 2 .switch”);
System.out.prinln(" 3 .while”);
System.out.prinln(" 4 .do-while”);
System.out.prinln(" 5 .for\n”);
System.out.prinln("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;
}
}
}
```

اكنون 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 براي تصديق اينكه كاربر يك گزينه معتبر را وارد كرده باشد ، استفاده مي شود. در غير اينصورت ، به كاربر مجددا اعلان خواهد شد. از آنجاييكه منو بايد حداقل يكبار بنمايش درآيد ، do-while لقه كاملي براي انجام اين مقصود است. چند نكته ديگر درباره اين مثال : دقت كنيد كه كاراكترهااز صفحه كليد بوسيله فراخواني system.in.read() خوانده مي شوند. اين يكي از توابع ورودي كنسول در جاوا است.

اگر چه بررسي تفصيلي روشهاي I/O جاوا به بحثهاي بعدي موكول شده ، اما از system.in.read() در اينجا براي بدست آوردن گزينه كاربر استفاده شده است. اين تابع كاراكترها را از ورودي استاندارد مي خواند. (كاراكترها بعنوان عدد صحيح برگردانده می شوند ، و به همين دليل از طريق casting به char تبديل می شوند.) وسيله ورودی طبق پيش فرض ، همچون يک بافر خطی است ، بنابراين برای ارسال كاراكترهای تايپی شده به برنامه ، می بايست كليد ENTER را فشار دهيد.

كار كردن با كنسول ورودی جاوا می تواند قدری ناراحت كننده باشد. به علاوه بيشتر برنامه های واقعی و اپلت های جاوا ، گرافيكی و مبتنی بر پنجره خواهند بود. نكته ديگر اينكه چون از system.in.read() استفاده شده است ، برنامه بايد از throws.java.io.IOExeption استفاده كند.

for

خواهيد ديد كه حلقه for يك ساختار قدرتمند و بسيار روان است. شكل كلي دستورfor بصورت زير است :

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

اگر فقط يك دستور بايد تكرار شود ، نيازي به آكولادها نيست. عملكرد حلقه for بشرح بعدي است. وقتي كه حلقه براي اولين بار شروع مي شود مقدار دهي اوليه در حلقه اجرا مي شود.

معمولا ، اين بخش يك عبارت است كه مقدار متغير كنترل حلقه را تعيين مي كند ، كه بعنوان يك شمارشگر ، كنترل حلقه را انجام خواهد داد. مهم است بدانيم كه عبارت مقدار دهي اوليه فقط يكبار اجرا مي شود. سپس شرط مورد ارزيابي قرار مي گيرد . اين شرط بايد يك عبارت بولي باشد. اين بخش معمولا مقدار متغير كنترل حلقه را با مقدار هدف مقايسه مي كند. اگر عبارت صحيح (true) باشد، آنگاه بدنه حلقه اجرا خواهد شد. اگر ناصحيح باشد حلقه پايان مي گيرد. بعد، بخش تكرار (iteration) حلقه اجرا مي شود . اين بخش معمولا عبارتي است كه مقدار متغير كنترل را افزايش يا كاهش مي دهد. آنگاه حلقه تكرار خواهد شد ، ابتدا عبارت شرطي را ارزيابي مي كند ، سپس بدنه حلقه را اجرا مي كند و سرانجام عبارت تكرار را در هر گذر (pass) اجرا ميكند. اين روال آنقدر دادمه مي يابد تا عبارت شرطي ناصحيح (false) گردد .

در زير روايت جديدي از برنامه "tick" را مي بينيد كه از يك حلقه for استفاده كرده 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);
}
}
```

تعريف متغييرهای كنترل حلقه در حلقه for

غالبا متغيري كه يك حلقه for را كنترل مي كند ، فقط براي همان حلقه مورد نياز بوده و كاربرد ديگري ندارد. در چنين حالتي ، مي توان آن متغير را داخل بخش مقدار دهي اوليه حلقه for اعلان نمود. بعنوان مثال در اينجا همان برنامه قبلي را مشاهده مي كنيد كه متغير كنترل حلقه يعني 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);
}
}
```

هنگاميكه يك متغيير را داخل يك حلقه for اعلان مي كنيد ، يك نكته مهم را بايد به ياد داشته باشيد : قلمرو آن متغيير هنگاميكه دستور for انجام مي شود ، پايان مي يابد. (يعني قلمرو متغير محدود به حلقه for است.) خارج از حلقه for حيات آن متغير متوقف مي شود. اگر بخواهيد از اين متغير كنترل حلقه در جاي ديگري از برنامه اتان استفاده كنيد ، نبايد آن متغير را داخل حلقه for اعلان نماييد.

درشرايطي كه متغير كنترل حلقه جاي ديگري موردنياز نباشد، اكثر برنامه نويسان جاوا آن متغير را داخل for اعلان مي كنند. بعنوان مثال ، در اينجا يك برنامه ساده را مشاهده مي كنيد كه بدنبال اعداد اول مي گردد. دقت كنيد كه متغير كنترل حلقه ، چون جاي ديگري مورد نياز نيست ، 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 كاما Comma

شرايطي پيش مي آيد كه مايليد بيش از يك دستور در بخش مقدار دهي اوليه initialization و تكرار (iteration) بگنجانيد. بعنوان مثال ، حلقه موجود در برنامه بعدي را در نظر بگيريد :

```
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--;
}
}
}
```

همانطوريكه مي بينيد ، حلقه توسط ارتباط متقابل دو متغير كنترل مي شود. از آنجاييكه حلقه توسط دو متغير اداره مي شود ، بجاي اينكه b را بصورت دستي اداره كنيم ، بهتر است تا هر دو را در دستور for بگنجانيم. خوشبختانه جاوا راهي براي اينكار دارد براي اينكه دو يا چند متغير بتوانند يك حلقه for را كنترل كنند ، جاوا به شما امكان مي دهد تا چندين دستور را در بخشهاي مقدار دهي اوليه و تكرار حلقه for قرار دهيد. هر دستور را بوسيله يك كاما از دستور بعدي جدا مي كنيم.

حلقه for قبلي را با استفاده از كاما ، خيلي كاراتر از قبل مي توان بصورت زير كد بندي 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 اين مثال ، بخش مقدار دهي اوليه ، مقاديرa وb و را تعيين مي كند. هربار كه حلقه تكرار مي شود ، دو دستور جدا شده توسط كاما در بخش تكرار (itration) اجرا خواهند شد. خروجي اين برنامه بقرار زير مي باشد :

```
a=1
b=4
```

```
a=2
b=3
```

نكته : اگر با C++/C آشنايي داريد ، حتما مي دانيد كه در اين زبانها ، علامت كاما يك عملگر است كه در هر عبارت معتبري قابل استفاده است. اما در جاوا اينطور نيست. در جاوا ، علامت كاما يك جدا كننده است كه فقط در حلقه for قابل اعمال مي باشد.

برخي از اشكال حلقه for

حلقه for از تعدادي گوناگونيها پشتيباني مي كند كه قدرت و كاربري آن را افزايش مي دهند. دليل انعطاف پذيري آن است كه لزومي ندارد كه سه بخش مقداردهي اوليه ، آزمون شرط و تكرار ، فقط براي همان اهداف مورد استفاده قرار گيرند. در حقيقت ، سه بخش حلقه for براي هر هدف مورد نظر شما قابل استفاده هستند. به چند مثال توجه فرمائيد.

يكي از رايجترين گوناگونيها مربوط به عبارت شرطي است. بطور مشخص ، لزومي ندارد اين عبارت ، متغير كنترل حلقه را با برخي مقادير هدف آزمايش نمايد. در حقيقت ، شرط كنترل كننده حلقه for ممكن است هر نوع عبارت بولي باشد. بعنوان مثال ، قطعه زير را در نظر بگيريد :

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

در اين مثال ، حلقه for تا زمانيكه متغير بولي done معادل true بشود ، اجرا را ادامه خواهد داد. اين مثال مقدار i را بررسي نمي كند . اكنون يكي ديگر از گوناگونيهاي جالب حلقه for را مشاهده مي كنيد. ممكن است يكي يا هر دو عبارت مقدار دهي اوليه و تكرار قابل حذف باشند ، نظير 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 اينجا عبارتهاي مقدار دهي اوليه و تكرار به خارج از for انتقال يافته اند. برخي از بخشهاي حلقه for تهي هستند. اگر چه در اين مثال ساده چنين حالتي هيچ ارزشي ندارد ، اما در حقيقت شرايطي وجود دارد كه اين روش بسيار كارا و سودمند خواهد بود. بعنوان مثال ، اگر شرط اوليه بصورت يك عبارت پيچيده و در جاي ديگري از برنامه قرار گرفته باشد و يا تغييرات متغير كنترل حلقه بصورت غير ترتيبي و توسط اعمال اتفاق افتاده در داخل بدنه حلقه تعيين شود ، پس بهتر است كه اين بخشها را در حلقه for تهي بگذاريم . اكنون يكي ديگر از گوناگونيهاي حلقه for را مشاهده مي كنيد. اگر هر سه بخش حلقه for را تهي بگذاريد ، آنگاه بعمد يك حلقه نامحدود (حلقه اي كه هرگز پايان نمي گيرد) ايجاد كرده ايد. بعنوان مثال :

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

اين حلقه تا ابد ادامه خواهد يافت ، زيرا هيچ شرطي براي پايان گرفتن آن تعبيه نشده است. اگر چه برخي برنامه ها نظير پردازشهاي فرمان سيستم عامل مستلزم يك حلقه نامحدود هستند ، اما اكثر حلقه هاي نامحدود در واقع حلقه هايي هستند كه ملزومات پايان گيري ويژه اي دارند. بزودي خواهيد ديد ، راهي براي پايان دادن به يك حلقه حتي يك حلقه نامحدود نظير مثال قبلي وجود دارد كه از عبارت شرطي معمولي حلقه استفاده نمي كند .

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 متغییر 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 );
}
}
```

خروجي به صورت زير 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
```

تكرار در آرايه هاي چند بعدي

اين كاربرد با يك مثال مشخص مي 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 زير 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 كليه زبانهاي برنامه نويسي ، جاوا نيز امكان تودرتو كردن 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 ;١٠>i}( ++
for(j=i; j ;١٠>j(++
System.out.print;(".")
System.out.println;)(
}
}
}
```

عبارات پرش

جاوا از سه عبارت پرش break ، continue ، return پشتيباني ميكند. اين عبارات ، كنترل را به بخش ديگري از برنامه تان انتقال مي دهند.

Break

اين دستور چهار كاربرد دارد :

١ – همانطور كه ديديد به اجراي متوالي عبارات در يكي از حالات عبارت switch پايان مي بخشد.

٢ – استفاده از break براي خروج از حلقه ، با استفاده از break مي توانيد يك حلقه را فورا پايان بخشيد و از جمله شرطي و مابقي علارات موجود در قسمت body صرف نظر كنيد. وقتي نوبت به اجراي عبارت Break در يك حلقه مي شود ، اجراي حلقه به پايان رسيده و كنترل اجراي برنامه به نخستين علارت پس از حلقه هدايت مي شود.

اين دستور مختص قطع جريان حلقه نيست ، بلكه دستورات شرطي وظيفه اين كار را دارند و در شرايط خاص قابل استفاده است.

۳ – استفاده از break به عنوان نوعي goto : جاوا فاقد عبارت goto است ، با وجود آنكه درك و نگهداشت روتين هاي goto دشوار است و از برخي از بهينه سازي هاي كامپايلر جلوگيري مي كند ، جاوا سعي كرده است با اين نوع از دستور break به گونه اي ديگر همان كار goto را انجام دهد. با اجرا اين دستور از اجراي يك يا چند بلوك صرفنظر مي شود.اين كار براي هر بلوكي امكانپذير است. ضمنا مي توانيد مشخص كنيد اجرا از كجا متوقف شود. break مزاياي goto را بدون مشكلات آن فراهم ميكند. اين دستور به فرم زير استفاده مي شود :

```
Break label;
```

Lable نشان دهنده بلوك مورد نظر مي 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.");
     }
   }
}
```

خروجي به فرم زير است :

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

٤ - يكي ديگر از كاربرد ها خروج از حلقه هاي 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 زير است :

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

Continue

گاهي لازم مي شود اجراي حلقه پيش از رسيدن به انتها تكرار شود. يعني ممكن است بخواهيد اجراي حلقه ادامه بايد ، اما پردازش مابقي عبارات قسمت body آن براي تكرار جاري متوقف شود. اين عبارت در حلقه هاي while و do-while سبب انتقال مستقيم كنترل به جمله شرطي كنترل كننده حلقه ميشود. در حلقه for كنترل نخست به قسمت iteration و سپس به جمله شرطي عبارت for هدايت مي شود. در هر سه حلقه ، عبارات پس از continue ناديده انگاشته ميشود.

```
// 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 % بررسي مي شود كه مقدار I زوج است يا خير. اگر چنين باشد ، حلقه بدون چاپ كاراكتر سطر جديد ادامه مي يابد. خروجي مانند زير است :

```
0 1
2 3
4 5
6 7
8 9
```

مانند دستور break مي توان از برچسب نيز استفاده كرد.
كاربردهاي درست براي عبارت break نادر است چون جاوا مجموعه غني از عبارات ايجاد حلقه دارد كه براي بيشتر كاربردها مناسب مي باشند ، اما براي شرايط ويژه اي كه تكرار زودتر از حد تعيين شده حلقه نياز باشد ، عبارت continue روش ساختاريافته اي براي انجام اين كار فراهم نموده است.

Return

براي بازگشت صريح از يك متد به كار مي رود. يعني سبب بازگرداندن كنترل اجرا به برنامه فراخوان متد مي شود. بدين ترتيب ، اين عبارت به عنوان پرش طبقه بندي شده است.
با استفاده از عبارت return در متدها مي توان كنترل اجرا را در هر لحظه به روتين فراخوان متد بازگرداند. از اين رو اجراي متدي كه در آن اجرا مي شود را فورا پايان مي دهد.

استفاده از استثنائات

اداره كردن استثنائات يك مكانيسم قدرتمند براي كنترل برنامه هاي پيچيده فراهم آورده كه داراي چندين ويژگي پوياي حين اجرا است . مهم اين است كه try ، throw ، throws ، finally و catch را بعنوان شيوه هاي واضح براي اداره خطاها و شرايط محدوده غير طبيعي در منطق برنامه هايتان در نظر بگيريد.
اگر مثل اكثر برنامه نويسان باشيد، آنگاه وقتي يك روش عقيم مي شود، احتمالا سعي مي كنيد تا يك كد خطا را برگردانيد . وقتي در جاوا مشغول برنامه نويسي هستيد ، بايد اين عادت را كنار

بگذاريد. وقتي يك روش مي تواند عقيم باشد، بهتر است يك استثنائ پرتاب نماييد . اين شيوه بمراتب بهتري براي اداره حالات عقيمي است.
نكته : 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 كرده 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 درپردازش جلوگيري كرده و مديريت خطاي حين اجرا را در دنياي شي گرايي نمايد.

اصول اداره استثنا

يك استثنا در جاوا، شيئي است كه يك شرط استثنايي (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 كليدي اعمال مي شود :try ، catch،throw ، throws و finally. بطور خلاصه عملكرد آنها را توضيح مي دهيم . دستورات برنامه اي كه مايليد براي استثنائات نشان دهيد داخل يك بلوك try گنجانده ميشوند. اگر داخل اين بلوك يك استثنائ حادث شود ، پرتاب خواهد شد . كد شما مي تواند اين استثنا را توسط catch گرفته و آن را بروشي منطقي اداره نمايد. استثنائات توليد شده توسط سيستم بطور خودكار توسط سيستم حين اجراي جاوا پرتاب مي شوند .

براي اينكه يك استثنائ را بصورت دستي پرتاب كنيم ، از واژه كليدي throw استفاده مي كنيم . هر استثنايي كه بيرون از يك روش پرتاب مي شود بايد توسط يك جملهthrows مشخص شود . هر كدي كه بايد كاملا" قبل از برگردانهاي يك روش اجرا شود در يك بلوك finally قرار داده مي شود. شكل عمومي يك بلوك اداره استثنائ بصورت زير مي 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
}
```

استثنايي است كه حادث شده است.

انواع استثنا

كليه انواع استثنا زير كلاسهايي از كلاس توكار throwable مي باشند. بنابراين throwable در بالاي سلسله مراتب كلاس استثنائ (exception) قرار دارد . بلافاصله پس از throwable دو

زير كلاس وجود دارند كه استثنائات را به دو شاخه مجزا تقسيم مي كنند ، سر عنوان يك شاخه Exception است . اين كلاس براي شرايط استثنايي كه برنامه هاي كاربر بايد بگيرد ، استفاده مي شود . همچنين از اين كلاس ، زير كلاسي مي سازيد تا انواع استثنائ سفارشي خودتان را ايجاد نماييد . يك زير كلاس با اهميت از Exception تحت عنوان RuntimeException وجود دارد . استثنائات اين نوع براي برنامه هايي كه شما مي نويسيد و مواردي نظير " تقسيم بر صفر " و " نمايه سازي غير معتبر آرايه " را در آن مي گنجانيد ، بطور خودكار تعريف مي شود .
شاخه ديگر تحت عنوان Error است كه استثنائاتي را تعريف مي كند كه انتظار نداريم . تحت شرايط عادي توسط برنامه شما گرفته شوند. استثنائات از نوع Error توسط سيستم حين اجرايجاوا براي نشان دادن خطاهايي كه با خود محيط حين اجراي جاوا سر و كار دارند ، استفاده مي شود . سرريزي پشته نمونه اي از اين خطاهاست . استثنائات نوع Error نوعا در پاسخ به شكستهاي مصيبت باري كه معمولا توسط برنامه اره آنها نيست ، بوجود مي آيند.

استثنائات گرفته نشده Uncaught Exceptions

قبل از اينكه بياموزيد كه چگونه استثنائات را در برنامه تان اداره نماييد بهتر است كه بفهميد در صورت عدم اداره استثنائات چه اتفاقي مي افتد . برنامه ساده بعدي دربرگيرنده يك عبارت است كه بعمد باعث يك خطاي تقسيم بر صفر مي شود.

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

وقتي سيستم حين اجراي جاوا تلاش خود براي انجام تقسيم برصفر را آشكار ميسازد يك شي ، استثنا جديدي ساخته و سپس آن را پرتاب مي كند . (throws) اين كار باعث توقف اجراي EXC0 مي گردد ، زيرا يكبار كه يك استثنا پرتاب شود ، بايد توسط يك اداره كننده استثنا گرفته شده (caught by an exception handler) و بلافاصله براي آن كاري انجام گيرد. دراين مثال

، ما اداره كننده استثنا خودمان را عرضه نكرده ايم ، بنابراين 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 رديابي 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.javaو شماره خط ، ٤ ، 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 توكار را عرضه مي كند كه با انواع گوناگون خطاهاي حين اجرا كه احتمالا توليد ميشوند، مطابقت مي يابند .

رديابي 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)(;
}
}
```

ردياب رشته بدست آمده از اداره كننده پيش فرض استثنائ نشان مي دهد كه چگونه پشته فراخواني كل بنمايش درآمده است:

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

همانطوريكه مي بينيد ، پايين پشته خط ٧ روش main وجود دارد كه تحت عنوانSubroutine است و سبب بروز خطا در خط ٤ شده است. پشته فراخوان براي اشكال زدايي (debugging) كاملا مفيد است ، زيرا سلسله دقيق مراحلي را كه منجربه خطا شده اند.

استفاده از try وcatch

اگرچه اداره كننده پيش فرض استثنا فراهم شده توسط سيستم حين اجراي جاوا براي اشكال زدايي مفيد است ، اما معمولا بدنبال آن هستيد تا يك استثنا خودتان اداره نماييد. انجام اين كار دو مزيت دارد. اول اينكه امكان تثبيت خطا را داريد. دوم اينكه ، اينكار مانع ختم خودكار برنامه خواهد شد . اگر هر بار كه در برنامه شما خطايي بروز مي كند ، برنامه اتان متوقف شده و يك رديـاب پشته چاپ نمايد ، آنگاه اكثر كاربران برنامه اتان گيج و سردرگم ميشوند. خوشبختانه جلوگيري از چنين حالتي بسيار ساده است.

براي جلوگيري از اين وضعيت و اداره يك خطاي حين اجرا ، خيلي ساده كدي را كه مي خواهيد نمايش دهيد (monitor) داخل يك بلوك try قرار دهيد. بلافاصله بعد از اين بلوك ، يك جمله catch قرار دهيد كه نوع استثنائي را كه مايليد بگيرد مشخص مي كند . براي اينكه سهولت اينكار را نشان دهيم ، برنامه بعدي را نگاه كنيد كه دربرگيرنده يك بلوك try و يك جمله catch ميباشد كه ArithmeticException توليد شده توسط خطاي تقسيم بر صفر را پردازش مي كند .

```
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.");
}
}
```

خروجي اين برنامه بقرار زير مي باشد :

```
Division by zero.
After catch statement.
```

دقت نماييد كه فراخواني println() داخل بلوك try هرگز اجرا نمي شود. هر بار كه يك استثنائ پرتاب مي شود، برنامه ، انتقالات خارج از بلوك try به بلوك catch را كنترل مي كند . بعبارت بهتر، catchفراخواني نمي شود بنابراين اجرا هرگز از يك catch بـه بلوك try برنمي گردد . بنابراين ، خط "This will not be printed" بنمايش در نمي آيد . هر بار كه دستور catch اجرا شود ، كنترل برنامه با خط بعدي برنامه تعقيب كننده مكانيسم كل try/catch ادامه خواهد يافت .

يك دستورtry و catch و يك واحد (unit) تشكيل مي دهند . قلمرو جمله catch محدود شده به آن دستوراتي است كه بلافاصله قبل از دستور try مشخص شده اند . يك دستور catch نمي تواند استثنائ پرتاب شده توسط يك دستور ديگر try را بگيرد ، مگر در حالت دستورات تودرتو شده try كه باختصار توضيح داده ايم . دستوراتي كه بوسيله try محافظت مي شوند . بايد توسط ابروها احاطه شوند . (يعني آنها بايد داخل يك بلوك قرار گيرند.) نمي توانيد از tryروي يك دستور منفرد استفاده نماييد . اين تغيير در روايت1.0.2 ازJDK ز معرفي شده است .

هدف اكثر جملات خوش ساخت catch بايد اين باشد كه شرط استثنايي رااز سر گرفتـه و سپس طوري برنامه را ادامه دهد كه گويا خطا هرگز اتفاق نيفتاده است . بعنوان مثال ، در برنامه بعدي هر تكرار حلقه for دو عدد صحيح تصادفي را كسب مي كند . آن دو عدد صحيح بر يكديگر تقسيم شده و جواب آن براي تقسيم عدد ١٢٣٤٥ استفاده مي شود . جواب نهايي در a قرار مي گيرد .

اگر هر يك از عمليات تقسيم منجر به خطاي " تقسيم بر صفر " شود ، آن خطا گرفته شده و مقدار a برابر صفر قرار گرفته و برنامه ادامه مي يابد .

```
// 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( ArithneticException e ){
System.out.println("Division by zero.");
a = 0; // set a to zero and continue
}
System.out.println("a :" + a);
}
}
}
```

نمايش توصيفي از يك استثنا

Throwable روش toString() ، (بوسيله object تعريف شده) را لغو ميكند ، بطوريكه يك رشته دربرگيرنده توصيفي از استثنائ را برمي گرداند. مي توانيد خيلي ساده با گذر دادن استثنا بعنوان يك آرگومان ، اين توصيف را در يك دستور println() به نمايش بگذاريد. بعنوان مثال ، بلوك catch در برنامه قبلي را مي توان بصورت زير دوباره نويسي نمود .

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

وقتي اين روايت را در برنامه جايگزين نماييد ، و برنامه تحت مفسر JDK جاوا اجرا شود ، هر خطاي تقسيم بر صفر ، پيام بعدي را نمايش خواهد داد :

```
Exception :java.lang.ArethmeticException :/ by zero
```

توانايي نمايش توصيفي از يك استثنا اگرچه در اين متن ارزش خاصي ندارد، اما در ساير شرايط بسيار ارزشمند است بخصوص هنگام كار با استثنائات يا هنگام اشكال زدايي.

جملات catch چند گانه

در بعضي مواقع از يك قطعه كوچك كد بيش از يك استثنا بوجود مي آيد . براي اداره چنين شرايطي ، مي توانيد دو يا چند جمله catch مشخص نماييد كه هر كدام يك نوع متفاوت از استثنائ بگيرند . وقتي يك استثنائ پرتاب مي شود ، هر دستور catch بترتيب امتحان مي شود، و اولين دستوري كه نوع آن با نوع استثنا مطابقت داشته باشد ، اجرا خواهد شد . بعد از اجراي يك دستور catch ، ساير دستورات پشت سر گذاشته مي شوند و اجراي بعد از بلوك try/catch ادامه خواهد يافت . برنامه بعدي دو نوع استثنائ مختلف را بدام مي 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 خط‌فرمان آغاز شود منجر به يك استثنا " تقسيم بر صفر " خواهد شد. اگر يك آرگومان خط‌فرمان 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 كند تا مقداري را به C[42] منسوب نمايد . در اينجا خروجي توليد شده بوسيله اجراي هر دو راه را مشاهده مي كنيد :

```
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 كنيد ، مهم است بدانيد كه زير كلاسهاي استثنائ بايد قبل از هر يك از كلاس بالاهاي مربوطه قرار گيرند . اين بدان دليل است كه يك دستور catch كه از يك كلاس بالا استفاده مي كند، استثنائات آن نوع و زير كلاسهاي آن نوع را خواهد گرفت . بنابراين ، اگر يك زير كلاس بعد از كلاس بالايش بيايد ، هرگز به آن زير كلاس نمي رسد. بعلاوه ، در جاوا ، كد غير قابل دسترس نوعي خطا است . بعنوان مثال ، برنامه بعدي را در نظر بگيريد :

```
/* This program contains an error.
A subclass must come befor its superclass in
a series of catch statements .If not/
unreachable code will be created and a
comple-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 بخواهيد اين برنامه را كامپايل كنيد ، يك پيغام خطا دريافت مي كنيد كه ميگويد دومين دستور catch غيرقابل رسيدن است . از آنجاييكه ArithneticException يك زيركلاس از Exception است ، اولين دستور catch كليه خطاهاي برمبناي Exception از جمله ArithneticException را اداره ميكند. بدين ترتيب دومين دستور catch هرگز اجرا نخواهد شد ، براي برطرف كردن اين مشكل ترتيب دو عبارت catch را تغيير دهيد.

دستورات تودرتو شدهtry

دستورات try را مي توان تودرتو نمود . يعني يك دستور try را مي توان داخل بلوك يك try ديگر قرار داد. هر بار كه يك دستور try وارد مي شود ، متن آن عبارت روي پشته نشانده مي شود. اگر يك دستور try داخلي تر فاقد يك اداره كننده catch براي يك استثنا خاص باشد، پشته دور زده نشده و اداره كننده catch مربوط به دستور بعدي try براي يك تطبيق مورد جستجو قرار مي گيرد . اين حالت تداوم مي يابد تااينكه يكي ازدستورات catch موفق شود و يا تا زماني كه كليه دستورات تودرتو شده try تمام شوند ، اگر هيچيك از دستورات catch با استثنا مطابقت نداشته باشند ، آنگاه سيستم حين اجراي جاوا خودش استثنائ را اداره مي كند . در زير مثالي را مشاهده ميكنيد كه 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);
}
}
}
```

همانطوريكه مشاهده مي كنيد، يك بلوك try را داخل ديگري جاي مي دهد . برنامه بصورت زير كار مي كند . وقتي برنامه را بدون آرگومانهاي خط فرمان اجرا مي كنيد يك استثنا " تقسيم بر صفر " توسط بلوك خارجي تر try ايجاد مي شود . اجراي برنامه توسط يك آرگومان خط فرمان يك استثنا " تقسيم بر صفر " را از داخل بلوك تودرتو شده try توليد مي كند. چون بلوك داخلي تر اين استثنائ را نمي گيرد استثنا به بلوك خارجي تر try گذر داده مي شود و در آنجا اداره خواهد شد . اگر برنامه را بدون آرگومانهاي خط فرمان اجرا نماييد، يك استثنا " محدوده آرايه " از داخل بلوك داخلي تر try توليد خواهد شد. در زير اجراهاي نمونه اي وجود دارند كه هر يك از حالات فوق را نشان مي 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 فراخواني روش هم وجود داشته باشد، آنگاه تودرتو نمودن دستورات try ممكن است به روشهايي پنهان تر اتفاق بيفتد . بعنوان مثال ، مي توانيد فراخواني به يك روش را درون يك بلوك try انجام دهيد . درون آن روش يك دستور ديگر try وجود دارد . در اين حالت ، tryدرون روش همچنان داخل يك بلوك خارجي تر try كه روش را فراخواني مي كند تودرتو مي شود . در زير برنامه قبلي را مشاهده مي كنيد كه در آن بلوك تودرتو شده try بداخل روش nesttry() نقل مكان كرده است.

```
/* Tyr 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 استثنائ 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 ، نظيرint يا char ، همچون كلاسهاي غير ازthrowable 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 دستورcatch

پيدا شود كه با نوع استثنا مطابقت داشته باشد . اگر مطابقت حاصل شود كنترل به آن دستور منتقل مي شود . اگر مطابقت پيدا نشود ، آنگاه دستور بعدي بسته شده try مورد جستجو قرار مي گيرد ، و همينطور الي آخر . اگر هيچ catch مطابقت كننده اي پيدا نشود ، آنگاه اداره كننده پيش فرض استثنائ مكثي به برنامه داده و رديابِ پشته را چاپ مي كند . در اينجا يك برنامه نمونه مشاهده مي كنيد كه يك استثنا را ايجاد و پرتاب مي نمايد . اداره كننده اي كه اين استثنائ را مي گيرد ، مجددا آن را به اداره كننده بيروني 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);
}
}
}
```

اين برنامه دو شانس براي كار با يك خطاي مشترك دارد . اول ، main() يك متن استثنا تنظيم مي كند و سپس demoproc() را فراخواني مي كند . روش demoproc() آنگاه يك متن ديگر اداره كننده استثنا را تنظيم نموده و بلافاصله يك نمونه جديد از NullpointerException را پرتاب مي كند كه روي خط بعدي گرفته خواهد شد . مجددا استثنا پرتاب خواهد شد . حاصل بقرار زير خواهد شد .

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

برنامه همچنين نشان مي دهد چگونه يكي از اشيا استثنا استاندارد جاوا ايجاد مي شود توجه بيشتري به اين خط داشته باشيد :

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

در اينجا ، new استفاده شده تا يك نمونه از NullpointerException ساخته شود .كليه استثنائات حين اجراي توكار جاوا دو سازنده دارند: يكي بدون پارامتر و يكي با يك پارامتر رشته اي . وقتي از شكل دوم استفاده مي شود ، آرگومان يك رشته را مشخص مي كند كه استثنا. را توصيف مي نمايد . وقتي كه شي ئ بعنوان يك آرگومان به print() يا println() استفاده مي شود ، اين رشته بنمايش درمي آيد. اين رشته را همچنين مي توان بوسيله يك فراخواني getMessage() كه توسط Throwable توصيف شده بدست آورد .

Throws

اگر يك روش ظرفيت ايجاد يك استثنائ را دارد و آن را اداره نمي كند، آن روش بايد اين رفتار خود را مشخص نمايد بگونه اي كه فراخوانان آن روش خودشان را در مقابل استثنائات محافظت نمايند . اينكار با گنجاندن يك جمله Throws در اعلان روش انجام ميگيرد. يك جمله throws انواع استثنائاتي كه يك روش ممكن است پرتاب نمايد و فهرست بندي مي كند . اينكار براي كليه استثنائات شروري است ، بغير از انواع Error يا RuntimeException ، يا استثنائات مربوط به زير كلاسهاي آنها . كليه ساير استثنائاتي كه يك روش مي تواند پرتاب نمايد ، بايد در جمله throws اعلان شوند . اگر اينكار انجام نشود ، يك خطاي (compile-time) حاصل خواهد شد . شكل عمومي يك اعلان روش كه دربرگيرنده جمله throws باشد ، بصورت زير است :

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

در اينجا exception-list يك فهرست جدا شده با كاما 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

caugh 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 مايل 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 استثناي 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 درابتداي 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 catchيا 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 اين مثال ، procA() با پرتاب يك استثنائ، 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 توكار جاوا

داخل بسته نرم افزاري استاندارد java.lang ، جاوا چندين كلاس استثنائ را تعريف مي كند . چند تا از اين كلاسها در مثالهاي قبلي استفاده شدند. عمومي ترين اين استثنائات زير كلاسهاي 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 كنترل شده (unchecked exception) مي نامند . زيرا كامپايلر كنترل نمي كند كه آيا يك روش اين استثنائات را اداره مي كند يا آنها را پرتاب مي نمايد . آنها را در جدول 1 فهرست نموده ايم . جدول 2 آن استثنائاتي را فهرست مي كند كه توسط java.lang تعريف شده اند و بايد در يك فهرست throws مربوط به روش گنجانده شوند ، اگر آن روش بتواند يكي از اين استثنائات را توليد نموده ، اما خودش آن را اداره نكند . اين استثنائات را استثنائات كنترل شده (checked exceptions) مي نامند . جاوا چندين نوع ديگر از استثنائات را تعريف مي كند كه با كتابخانه هاي گوناگون كلاس جاوا مرتبط هستند.

زير كلاسهاي كنترل نشده RuntimeException در جاوا

توضيح	استثنا
خطاي جبري ، مثل تقسيم بر صفر	**ArithmeticException**

نمايه آرايه خارج از محدوده است	ArrayIndexOutOfBoundsException
انتساب به يك عضو آرايه از يك نوع سازگار	ArrayStoreException
تبديل cast فير معتبر	ClassCastException
آرگومان غير مجاز استفاده شده براي فراخواني مجدد يك روش	IllegalArgumentException
عمليات ناشي غير مجاز ، نظير منتظر ماندن روي يك بند قفل نشده . (unlocked thread)	IllegalMonitorStateException
عمليات درخواست شده ناسازگار با وضعيت بند جاري	IllegalThreadStateException
برخي انواع نمايه خارج از محدوده است	IndexOutOfBoundsException
آرايه ايجاد شده با يك اندازه منفي	NegativeArrayException
كاربرد غير معتبر از يك مرجع تهي	NullpointerException
تبديل غير معتبر يك رشته به يك فرمت رقمي	NumberFormatException
تلاش براي نقض امنيت	SecurityException

استثنائات کنترل شده توکار در جاوا

توضيح	استثنا
كلاس پيدا نشده است	ClassNotFoundException
تلاش براي توليد مثل يك شي كه رابط Cloneable را پياده سازي نميكند.	CloneNotSupportedException.
دسترسي به يك كلاس انكار شده است	IllegalAccessException
تلاش براي ايجاد يك شي از يك كلاس يا يك رابط	InstantiationException
توقف يك thread توسط يك thread ديگر	InterruptedException
عدم وجود فيلد درخواستي	NoSuchFieldException
عدم وجود متد درخواستي	NoSuchMethodException

ايجاد نمودن زير كلاسهاي استثناء مربوط به خودتان

اگرچه استثنائات توكار جاوا اكثر خطاهاي رايج را اداره مي كنند ، احتمال دارد بخواهيد انواع استثنائ مربوط به خودتان را ايجاد كنيد تا شرايط مشخص پيش آمده در برنامه هاي شما را اداره كنند. انجام اينكار بسيار ساده است : فقط يك زيركلاس از Exceptionتعريف نماييد (كه البته يك زيركلاس از Throwableمي باشد .(زير كلاسهاي شما لزومي ندارند تا واقعا" چيزي را پياده سازي كنند.

كلاس Exception هيچگونه روشي براي خود تعريف نمي كند . البته اين كلاس از روشهاي فراهم شده توسط Throwable ارث مي برد . بدين ترتيب كليه استثنائات شامل آنهاييكه شما ايجاد كرده ايد ، داراي روشهايي هستند كه توسط Throwable تعريف شده و در دسترس آنها مي باشند آنها را در جدول زير نشان داده ايم . همچنين ممكن است بخواهيد يك يا چند تا از اين روشها را در كلاسهاي استثنايي كه ايجاد كرده ايد ، لغو نماييد.

روشهاي تعريف شده توسط Throwable

متد	شرح
ThrowablefillinStackTrace()	**يك شي Throwable را كه شامل يك ردياب تكميل شده پشته است برمي گرداند . اين شي ممكن است مجددا پرتاب شود .**
StringgetMessage()	**توصيفي از استثنائ را برمي گرداند**
VoidprintStackTrace()	**ردياب پشته را نمايش مي دهد**
StringtoString()	**شي از نوع string را برمي گرداند كه متضمن شرحي از استثناست. اين متد هنگام نمايش محتواي شي throwable به وسيله println() فراخوانده ميشود.**

مثال بعدي يك زير كلاس جديد از Exception اعلان مي كند و سپس از آن زير كلاس استفاده مي كند تا علامت يك شرط را به يك روش ارسال نمايد . اين زير كلاس ، روش()tostring را لغو مي كند ، و با استفاده از println() اجازه مي دهد تا توصيف استثنائ بنمايش درآيد .

```
// 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);
}
}
}
```

اين مثال يك زير كلاس از Exception تحت عنوان MyException تعريف مي كند. اين زير كلاس خيلي ساده است : اين زير كلاس فقط يك سازنده بعلاوه يك روش tostring() انباشته شده دارد كه مقدار استثنائ را نمايش مي دهد . كلاس ExceptionDemo يك روش تحت نام compute() معرفي مي كند كه يك شي ئ MyException را پرتاب مي كند . وقتي كه پارامتر عدد صحيح مربوط به compute() بزرگتر از ١٠ باشد ، استثنائ پرتاب خواهد شد . روش main() يك اداره كننده استثنائ براي MyException قرار مي دهد ، سپس compute() را با يك مقدار مجاز (كمتر از ١٠) و نيز يك مقدار غير مجاز براي نشان دادن دو مسير موجود در كد فراخواني مي كند . حاصل بصورت زير است :

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

آشنایی با کلاس ها

عناوین این بخش :

اصول کلاس ها
شيوه تعريف كردن شي
تخصيص متغييرهاي ارجاع به شي
معرفي متدها (جايگزين توابع)
Constructor ها
كلمه كليدي this
باز پس گيري حافظه بلا استفاده
متد finalize()
كلاس Stack

در اين فصل علاوه بر نوع داده انتزاعي با نحوه تعريف و فراخواني زيربرنامه (متد) ، كنترل زيربرنامه ، نحوه اختصاص حافظه به انها ، ارسال پارامتر و زيربرنامه بازگشتي آشنا ميشويد

اصول كلاس‌ها

كلاس‌ها در هسته مركزي جاوا جاي دارند. كلاس‌ها ، ساختار منطقي هستند كه كل زبان جاوا بر روي آن ساخته شده است، چرا كه شكل و ماهيت شي ها را تعريف مي‌كنند. بدين ترتيبت، كلاس‌ها پايه و اساس برنامه‌سازي شي گرا را در جاوا تشكيل مي‌دهند. هر موضوعي كه بخواهيد در برنامه‌هاي جاوا پياده سازي كنيد، مي‌بايست در يك كلاس نهان[1] شود.

شايد مهمترين نكته‌اي كه بايد درباره كلاس‌ها ياد گرفت آن است كه نوع جديدي از داده‌ها را تعريف مي كنند. داده‌هاي نوع جديد را پس از تعريف شدن مي‌توان براي ايجاد شيء‌هاي نوع مورد نظر به كار رد. از اين رو، هر كلاس، الگويي[2] براي يك شيء است. و هر شيء هم نمونه-اي[3] از يك كلاس به شمار مي‌آيد. چون هر شيء، نمونه‌اي از يك كلاس است، اغلب خواهيد ديد كه دو واژه شي و نمونه به جاي يكديگر به كاربرده مي‌شوند.

وقتي كلاسي را تعريف مي‌كنيد، ماهيت و فرم دقيق آن معرفي مي‌شود. اين كار با مشخص كردن داده‌هاي درون آن و روتين‌هايي كه بر روي آن داده‌ها عمل مي‌كنند، انجام مي‌شود. اگر چه بسياري از كلاس‌هاي ساده ممكن است تنها دربرگيرنده روتين‌ها يا داده‌ها باشند، اما بيشتر كلاس-هاي مطرح در كارهاي واقعي، هر دو را شامل مي‌شوند. همان گونه كه خواهيد ديد، روتين‌هاي هر كلاس، رابط منتهي به داده‌هاي آن را تعريف مي‌كنند.

هر كلاس با استفاده از كلمه كليدي Class تعريف مي‌شود. كلاس‌ها مي‌توانند بسيار پيچيده‌تر باشند (و معمولاً هستند) شكل عمومي تعريف هر كلاس در ذيل نشان داده شده است.

```
Class classname {
Type instrance-variable1;
Type instance-variable2;
II…
Type instance-variableN;

Type methodname1(parameter-list){
//body of method
}
Type methodname2(parameter-list){
```

[1] - encapsulated.
[2] - template.
[3] - instance.

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

داده‌هايي، يا متغييرهايي، كه در هر كلاس تعريف مي‌شوند، «نمونه متغير» [4] ناميده مي‌شوند. روتين‌ها نيز در متدها جاي مي‌گيرند. به طور كلي، به متدها و متغيرهايي كه در هر كلاس تعريف مي‌شوند، اعضاي[5] كلاس گفته مي‌شود. در بيشتر كلاس‌ها، متدهاي تعريف شده براي هر كلاس هستند كه بر روي نمونه متغيرها كار مي‌كنند و به آنها دستيابي دارند. از اين رو، اين متدها هستند كه چگونگي استفاده از داده‌هاي هر كلاس را تعيين مي‌كنند.

دليل اينكه متغيرهاي هر كلاس، نمونه متغير خوانده مي‌شوند، آن است كه هر نمونه از يك كلاس (يعني، هر شيء از يك كلاس)، كپي خاص خود را متغيرها دارد. از اين رو، داده‌هاي هر شيء، جداگانه و خاص خود آن بوده و با داده‌هاي يك شيء ديگر يكسان نيستند.

كلاس‌هاي جاوا نياز به متد main() ندارند. تنها زماني چنين متدي مشخص مي‌شود كه كلاس مورد نظر، نقطه آغازين برنامه‌تان باشد. به علاوه، آلت‌ها اصلاً نياز به متدي به نام main() ندارند.

معرفي كلاس و پياده سازي متدها در يكجا ذخيره مي‌شوند و به طور جداگانه تعريف نمي‌شوند. اين امر گاهي اوقات سبب ايجاد فايل‌هاي java بسيار بزرگ مي‌شود، چه آنكه هر كلاس بايد كلاً در يك فايل واحد تعريف شود.

يك كلاس ساده

مطالعه كلاس‌ها را با يك مثال ساده آغاز مي‌كنيم، براي اين كار كلاسي به نام Box تعريف مي‌كنيم كه داراي سه نمونه متغير به نام height, width و depth است. در حال حاضر، Box فاقد هر گونه متد است (اما در آينده متدهايي به آن افزوده خواهد شد).

[4] - instance variable.
[5] - member.

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

همان گونه كه گفته شد، هر كلاس، نوع جديدي از داده‌ها را تعريف مي‌كند. در اين مثال خاص، نوع جديدي كه ايجاد مي‌شود، Box ناميده شده است. از اين نام براي تعريف شيء‌ها نوع Box استفاده خواهد شد. مهم است به خاطر بسپاريد كه تعريف هر كلاس جديد تنها سبب ايجاد يك الگو[6] مي‌شود؛ يك شيء واقعي ايجاد نمي‌شود.

براي آنكه يك شيء Box ايجاد شود، مي‌بايست از عباراتي همچون سطر زير استفاده كنيد.

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

پس از آنكه عبارت بالا اجرا شد، mybox به عنوان نمونه‌اي از Box ايجاد خواهد شد. از اين رو، يك واقعيت «فيزيكي» از كلاس Box ايجاد خواهد شد.

باز هم لازم به ذكر است كه هر گاه نمونه‌اي از يك كلاس را ايجاد مي‌كنيد شيئي ايجاد مي‌شود كه نسخه خاص خودش را از هر يك از نمونه متغيرهاي تعريف شده در آن كلاس خواهد داشت. از اين رو، هر شيء Box ، نسخه‌هاي خاص خودش را از نمونه متغيرهاي height, width و depth خواهد داشت. براي دستيابي به اين متغيرها بايد از عملگر نقطه[7] (.) استفاده كنيد . اين عملگر، نام شيء را به نام «نمونه متغير» مرتبط مي‌كند. به عنوان مثال، براي آنكه مقدار ۱۰۰ را به متغير width از mybox تخصيص دهيد، از عبارت زير استفاده كنيد:

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

عبارت بالا براي كامپايلر مشخص مي‌كند كه مقدار ۱۰۰ را به نسخه‌اي از width كه در شيء mybox است تخصيص دهد.به طور كلي، از عملگر نقطه (.) براي دستيابي به نمونه متغيرها و متدهاي موجود در يك شيء استفاده مي‌شود.

هر شيء نسخه‌هاي خاص خودش را از نمونه متغيرها خواهد داشت. اين بدين معناست كه اگر دو شيء نوع Box داشته باشيد، هر يك از آنها، نسخه‌هاي خاص خودشان را از height, width و depth خواهند داشت. مهم است بدانيد كه تغييراتي كه در نمونه متغيرهاي يك شيء ايجاد مي-شوند. هيچ تأثيري بر نمونه متغيرهاي شيء ديگر نخواهند داشت.

---

[6] . Template.
[7] . dot.

شيوه تعريف كردن شيءها

همان گونه كه در بالا شرح داده شد، وقتي كلاسي را ايجاد مي‌كنيد، در واقع يك نوع جديد براي داده‌ها ايجاد مي‌شود. از اين نوع جديد مي‌توانيد براي تعريف كردن شيءهايي از آن نوع استفاده كنيد. اما ، رسيدن به شيءهاي يك كلاس، نوعي فرآيند دو مرحله‌اي است. نخست اينكه، بايد متغيري از نوع كلاس تعريف كنيد. اين متغير سب تعريف يك شيء نمي‌شود. بلكه در عوض، متغيري است كه مي‌تواند به يك شيء ارجاع داشته باشد. دوم اينكه، مي‌بايست يك نسخه فيزيكي واقعي از شيء به دست آوريد و آن را به آن متغير تخصيص دهيد. اين كار را مي‌توانيد با استفاده از عملگر new انجام دهيد. عملگر new ، حافظه‌اي را به طور پويا (يعني در زمان اجرا) به شيء تخصيص مي‌دهد و نشاني آن را برمي‌گرداند. اين نشاني سپس در متغير ذخيره مي‌شود. از اين رو، تمام شيءهاي نوع كلاس در جاوا بايد به طور پويا تخصيص يابند.

در نمونه برنامه‌هاي زير از سطري مشابه عبارت زير براي تعريف شيئي از نوع Box استفاده خواهدشد:

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

دو مرحله پيش گفته در عبارت بالا تركيب شده‌اند. عبارت بالا را مي‌توان براي نشان دادن هر يك از مراحل به صورت زير بازنويسي كرد:

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

در سطرح نخست، mybox به عنوان نشاني شيئي از نوع Box تعريف مي‌شود. پس از اجراي اين خط، مقدار null در mybox ذخيره خواهد شد كه نشانگر آن است كه متغير هنوز به هيچ شيء واقعي ارجاع ندارد. هر گونه اقدام براي استفاده از mybox در اين مرحله منجر به بروز خطاي زمان كامپايل خواهد شد. سطر دوم هم موجب تخصيص شيء واقعي و تخصيص نشاني آن به mybox مي‌شود. پس از اجراي سطر دوم، مي‌توانيد از mybox به گونه‌اي استفاده كنيد كه گويي يك شيء Box است. اما mybox صرفاً نشاني حافظة شيء Box واقعي را نگهداري مي-كند.

تخصيص متغيرهاي ارجاع به شي

وقتي عمل تخصيص انجام مي‌گيرد، عملكرد متغيرهاي ارجاع به شيء با آنچه انتظار داريد تفاوت دارد. فكر مي‌كنيد دو عبارت زير چه عملي انجام مي‌دهند؟

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

ممكن است چنين تصور كنيد كه نشاني نسخه‌اي از شيئي كه b1 به آن ارجاع دارد. به b2 تخصيص مي‌يابد. يعني، ممكن است چنين فكر كنيد كه b1 و b2 به شيء‌هاي جداگانه و متمايزي ارجاع دارند. اما، اين تصور درست نيست. بلكه در عوض، پس از اجراي عبارات بالا، b1 و b2 هر دو به يك شيء ارجاع خواهند داشت. تخصيص b1 و b2 موجب تخصيص حافظه يا كپي كردن بخشي از شيء اوليه نمي‌شود. بلكه صرفاً سبب مي‌شود كه b2 نيز به همان شيئي كه b1 به آن ارجاع دارد، ارجاع داشته باشد. از اين رو، هر گونه تغيير در شيء از طريق b2 ، بر شيئي كه b1 به آن ارجاع دارد، تأثير خواهد گذاشت، چرا كه هر دو آنها يك شيء هستند.

معرفي متدها

كلاس‌ها معمولاً از دو چيز تشكيل مي‌شوند: نمونه متغيرها[8] و متدها[9]. موضوع متدها بسيار گستردهاست، چرا كه جاوا قدرت و انعطاف‌پذيري زيادي را در آنجا جاي داده است.

شكل كلي هر متد به صورت زير است:

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

Type، نوع داده‌هايي را مشخص مي‌كند كه متد بازمي‌گرداند. Type مي‌تواند هر يك از انواع مورد بررسي قبلي باشد، از جمله انواع كلاس‌هايي كه خودتان ايجاد مي‌كنيد. چنانچه متد چيزي را برنگرداند، type بايد void باشد. نام متد نيز به وسيله name مشخص مي‌شود. از هر شناسه معتبري مي‌توانيد به عنوان نام استفاده كنيد؛ البته به غير از مواردي كه براي اقلام موجود در همان محدوده جاري استفاده شده اند. Parameter-list، فهرست زوجهايي (نوع و شناسه) است كه با كاما از يكديگر جدا مي‌شوند. پارامترها اساساً متغيرهايي هستند كه مقدار آرگومان‌هاي

[8] - instance variable.
[9] – method.

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

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

```
Return value;
```

Value ، مقداري است كه برگردانده مي‌شود.

افزودن متد به كلاس Box

اگر چه ايجاد كلاس‌هايي كه تنها حاوي داده باشند كاملاً درست است، اما اين امر به ندرت رخ مي‌دهد. در بيشتر مواقع از متدها براي دستيابي به نمونه متغيرهاي تعريف شده در كلاس‌ها استفاده خواهد شد. در حقيقت، متدها، رابط دستيابي به بيشتر كلاس‌ها را تعريف مي‌كنند. اين امر به ايجاد كننده كلاس‌ها امكان مي‌دهد تا شماي ساختارهاي داده‌اي مرتبط با كلاس را پشت سر متدهاي شفاف‌تر پنهان نمايد. علاوه بر تعريف متدهايي كه دستيابي به داده‌ها را فراهم مي‌سازند، امكان تعريف متدهايي كه توسط خود كلاس‌ها به طور داخلي مورد استفاده قرار مي‌گيرند نيز فراهم شده است.

اينك كار خود را با افزودن متدي به كلاس Box آغاز مي‌كنيم.

```
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 ( ) ;
   }
}
```

خروجي برنامه در ذيل آورده شده است كه البته با خروجي مثال پيش يكسان است.

```
Volume is 3000.0
Volume is 162.0
```

وقتي دستيابي به نمونه متغيرها به وسيله روتيني انجام مي‌گيرد كه در همان كلاس تعريف متغيرها تعريف نشده است، در آن صورت بايد اين كار از طريق نام شيء و عملكرد نقطه(.) انجام شود. اما، وقتي اين كار به وسيله روتيني انجام مي‌شود كه بخشي از همان كلاس مربوط به متغيرهاست، در آن صورت متغيرها به طور مستقيم قابل ارجاع مي‌باشند. اين مطلب درباره متدها نيز صادق است.

بازگرداندن مقادير

دو نكته مهم درباره مقادير حاصل از فراخواني متدها وجود دارد كه بايد به خوبي با آنها آشنا باشيد:

نوع داده‌هاي حاصل از فراخواني متدبايد با نوعي كه در تعريف متد مشخص شده است، سازگار باشد. به عنوان مثال، اگر نوع مقداري كه يك متد بازمي‌گرداند، Boolean باشد، نمي‌توانيد مقدار صحيحي را بازگردانيد.

متغير دريافت كننده مقدار حاصل از فراخواني متد (مثلاً vol در اين مثال)، بايد با نوعي كه در تعريف متد مشخص شده است، سازگار باشد.

افزودن متدهاي پارامتريك

اگر چه برخي از متدها نياز به پارامتر ندارند، اما بيشتر متدها اين گونه نيستند. پارامترها، امكان عموميت بخشيدن به متدها را فراهم مي‌سازند. يعني، متدهاي پارامتريك مي‌توانند بر روي انواع داده‌ها عمل كنند، و يا در شرايط نسبتاً مختلف مورد استفاده قرار گيرند. براي درك اين نكته به مثال بسيار ساده زير توجه كنيد. متد زير، مجذور عدد ۱۰را بازمي‌گرداند:

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

اگر چه اين متد واقعاً مجذور عدد ۱۰ را بازمي‌گرداند، اما كاربرد آن بسيار محدود است، اما اگر متد را به گونه‌اي تغيير دهيم تا پارامتري را همچون مثال زير دريافت كنيد، در آن صورت square() بسيار مفيدتر مي‌شود.

```
Int square (int  i)
{

  }
```

بدين ترتيب square() اينك مجذور هر مقداري كه با آن فراخوانده مي‌شود را بازمي‌گرداند. يعني، square() به متد همه منظوره‌اي مبدل شده است كه به جاي عدد ۱۰۰ ، مجذور هر عدد صحيح را محاسبه مي كنيد.

حفظ تمايز بين دو واژه ارامتر و آرگومان از اهميت خاصي برخوردار است. منظور از پارامتر، متغيري است كه توسط متد تعريف مي‌شود و وقتي متد فراخوانده مي‌شود، مقداري را دريافت مي-كند. به عنوان مثال، در متد بالا، i پارامتر به شمار مي‌آيد. منظور از آرگومان، مقداري است كه

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 مي‌كند.

Constructor ها

جاوا اين امكان را فراهم ساخته تا شيءها خودشان را به هنگام ايجاد، مقدار دهي كنند اين مقداردهي خودكار، از طريق استفاده از يك constructor انجام مي‌شود.

constructor ، يك شيء را به محض ايجاد مقدار دهي مي‌كند. نام آن با نام كلاسي كه در آن قرار دارد يكسان بوده و از نظر ساختار گرامري نيز مشابه متدهاست. هر constructor پس از تعريف، به طور خودكار به محض ايجاد شيء فراخوانده مي‌شود.

اين وظيفه constructorهاست كه وضعيت داخلي يك شيء را در همان ابتداي كار تعيين كنند (مقدار دهي اوليه)، تا روتيني كه نمونه‌اي از كلاس را ايجاد مي‌كند، فوراً شيء قابل استفاده و مقدار دهي 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.printIn("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 كه ملاحظه مي‌كنيد ، mybox1 و mybox2 هر دو هنگام ايجاد به وسيلة constructor كه Box() نام دارد، مقدار دهي شده‌اند. از آنجايي كه Constructor ابعاد ۱۰×۱۰×۱۰ را به تمام مكعب‌ها نسبت مي‌دهد، حجم mybox1 و mybox2 برابر خواهد شد. عبارت printLn() در Box() صرفاً به خاطر نمايش صحت عملكرد آن است. بيشتر constructor ها چيزي را نمايش نخواهند داد. آنها صرفاً عمل مقداردهي اوليه را براي شيء انجام مي‌دهند.

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

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

newBox() موجب فراخواني constructor كلاس مي‌شود كه همنام با خود كلاس است (يعني Box()) وقتي constructorاي را صريحاً براي كلاسي تعريف نمي‌كنيد. جاوا اين كار را به طور پيش فرض انجام مي‌دهد. به همين دليل است كه سطر بالا در نگارشهاي پيشين مثال Box ، كه فاقد تعريف constructor بودند، به خوبي كار مي‌كرد. constructor پيش فرض، تمام نمونه متغيرها را به طور خودكار با صفر مقداردهي مي‌كند. Constructor پيش فرض اغلب براي كلاس‌هاي ساده كفايت مي‌كند، اما معمولاً براي كلاس‌هاي پيچيده‌تر كفايت نمي‌كند. وقتي constructor خاص خودتان را تعريف مي‌كنيد، constructor پيش فرض ديگر به كاربرده نمي‌شود.

Construdtor هاي پارامتريك

اگر چه constructor مثال پيش (Box()) عمل مقدار دهي اوليه را براي شيء Box انجام مي-دهد، اما چندان مفيد نيست ـ ابعاد تمام مكعبها يكسان خواهد بود. بايد به دنبال روشي براي ساخت شيءهاي Box با ابعاد گوناگون باشيم. راه حل آسان براي انجام اين كار، افزودن پارامترهايي به constructor است. همان گونه كه احتمالاً حدس زده‌ايد، انجام اين كار موجب مفيدتر شدن آن مي‌شود. به عنوان مثال، در نگارش جديد Box ، يك constructor پارامتريك تعريف شده است كه ابعاد هر مكعب را براساس تعداد پارامترها تعيين مي‌كند.

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

كلمه كليدي this

گاهي اوقات متدها نياز به ارجاع به شيئي دارند كه آنها را فعال كرده است. جاوا براي فراهم ساختن اين امكان، كلمه كليدي this را تعريف كرده است. با استفاده از this در هر متد مي توان

به شيء جاري ارجاع نمود. يعني ، this هميشه ارجاع به شيئي دارد كه متد براي آن فعال شده است. هر جا كه ارجاع به شيئي از كلاس جاري مجاز باشد، مي‌توان 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 كار مي‌كند. استفاده از this بي‌مورد است، اما كاملاً صحيح است. this در اين نگارش، هميشه به شيئي كه متد را فرامي‌خواند، ارجاع خواهد داشت. اگر چه كاربرد آن در اين مثال بي‌مورد است، اما در ساير مواردمفيد واقع مي‌شود.

بازپس‌گيري حافظه بلا استفاده

از آنجايي كه شيء‌ها با استفاده از عملگر new به طور پويا تخصيص مي‌يابند، ممكن است از خود بپرسيد كه چگونه از بين برده مي‌شوند و حافظه آنها چگونه براي استفاده‌هاي آتي آزاد مي-شود. در برخي از زبانها، از قبيل C++ ، شيء‌هايي كه به طور پويا تخصيص داده مي‌شوند را بايد به صورت دستي با استفاده از عملگر delete ، آزاد نمود. جاوا از رويه ديگري استفاده مي-كند؛ آزاد سازي را به طور خودكار برايتان انجام مي‌دهد. تكنيكي كه از آن براي انجام اين كار استفاده مي‌شود، garbage collection نام دارد. عملكرد آن به اين شرح است؛ وقتي هيچ گونه ارجاعي به يك شيء وجود نداشته باشد، فرض مي‌شود كه شيء ديگر مورد نياز نبوده و حافظه آن نيز بازپس گرفته مي‌شود. در زبان جاوا، برخلاف C++ ، ديگر نيازي به از بين بردن شيء‌ها نيست. اين كار تنها به صورت نامنظم و گاه و بيگاه در طي اجراي برنامه انجام مي‌شود.

متد Finalize()

گاهي اوقات برخي از شيءها نياز به انجام عمليات خاص پيش از بين بردن دارند. به عنوان مثال، اگر شيئي از منابع غير جاوا ، از قبيل handle يك فايل يا فونت خاص، استفاده مي‌كند، در آن صورت بهتر است پيش از آزادسازي آن شيء، از آزاد شدن آن منابع اطمينان حاصل نماييد. جاوا براي مديريت اين گونه شرايط، مكانيزمي به نام funalization دارد. با استفاده از اين مكانيزم مي‌توانيد عمليات خاصي را مشخص كنيد تا درست پيش از آزاد سازي يك شيء، تماماً انجام شوند.

براي پياده سازي اين مكانيزم در هر كلاس، كافي است متد finalize() را تعريف كنيد. محيط زمان اجراي جاوا اين متد را هنگام بازيافت شيئي از آن كلاس فرا مي‌خواند. در متد finalize() بايد آن عملياتي را مشخص كنيد كه بايد پيش از بين بردن يك شيء انجام شوند. قسمتي كه از مسئوليت بازپس‌گيري حافظه بلا استفاده را دارد، به طور متناوب اجرا شده و شيءهايي را جستجو مي‌كند كه ديگر ارجاعي به آنها صورت نمي‌گيرد و بهطور غير مستقيم نيز از طريق ساير شيءها به آنها ارجاع نمي‌شود. درست پيش از آزاد كردن هر شيء، سيستم «زمان اجراي» جاوا متد finalize() را براي آن شيء فرا مي‌خواند.

شكل كلي اين متد در زير نشان داده شده است:

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

كلمه كليدي protected ، مشخصه‌اي[10] است كه از دستيابي به finalize() توسط روتين‌هاي خارج از همان كلاس جلوگيري مي‌كند.

مهم است بدانيد كه finalized() تنها پيش از بازپس گيري حافظه شيءها فراخوانده مي‌شود. مثلاً زماني كه يك شيء در خارج از محدوده‌اش قرار مي‌گيرد، اين متد فراخوانده نمي‌شود.

Overload كردن متدها

در زبان جاوا اين امكان فراهم شده تا دو و يا بيش از دو متد همنام در يك كلاش تعريف نمود، مشروط بر اينكه تعريف پارامترهاي آنها متفاوت مي‌باشد در اين گونه موارد گفته مي‌شود كه

[10] - specifier.

متدها، Overload شده اند و به این فرآیند، method overload گفته مي‌شود. این فرآیند یکي از روشهایي است که جاوا از طریق آن پلي مورفیزم پشتیباني مي‌کند.

وقتي متد overload شده اي فعال مي‌شود، جاوا از انواع و یا تعداد آرگومان ها براي 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 مقادیري که این متدها بر مي‌گردانند ممکن است متفاوت باشد، اما نوع مقادیر به تنهایي براي تمایز بین آنها کفایت نمي کند.

وقتي که جاوا با عبارت فراخواني این گونه متدها مواجه مي‌شود، متدي را اجرا مي‌کند که پارامترهاي آن با آرگومان هاي مورد استفاده در عبارت فراخواني مطابق داشته باشد.

مثال ساده زیر فرآیند overload کردن متدها را نشان مي‌دهد:

```
/ / 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 كه ملاحظه مي‌كنيد، test ( ) چهار مرتبه overload شده است. نخستين نگارش آن پارامتري ندارد، نگارش دوم تنها يك پارامتر از نوع اعداد صحيح دارد، نگارش سوم دو پارامتر از نوع اعداد صحيح دريافت مي‌كند، و نگارش چهارم نيز تنها يك پرامتر از نوع double دارد.

وقتي متد overload شده اي فراخوانده مي‌شود، جاوا تطابق بين آرگومان هاي مورد استفاده براي فراخواني، و پارامترهاي متد را بررسي مي‌كند. اما، نيازي نيست كه اين تطابق هميشه دقيق باشد. تبديل خودكار انواع داده ها در برخي از شرايط، نقش مهمي را تعيين متدي كه بايد فعال شود، ايفا مي‌كند.

دليل اينكه فرآيند overload كردن متدها از پلي مورفيزم پشتيباني مي‌كند، آن است كه اين فرآيند يكي از راههايي است كه جاوا از طريق آن، مدل «يك رابط، چند متد» را پياده سازي مي‌كند. در زبانهايي كه از overload كردن پشتيباني نمي كنند، معمولاً دو يا سه نگارش از اين تابع وجود دارد كه نامشان قدري با يكديگر تفاوت دارد. به عنوان مثال، تابع abs ( ) در زبان C قدر مطلق يك عدد صحيح را بر مي‌گرداند، و labs ( ) قدر مطلق يك عدد صحيح نوع long را بر مي‌گرداند، و fabs ( ) نيز قدر مطلق يك مقدار اعشاري با مميز شناور را بر مي‌گرداند. از آنجايي كه زبان C از overload كردن متدها پشتيباني نمي كند، با وجود آنكه هر سه تابع اساساً يك كار انجام مي‌دهند، اما هر يك از توابع پايه خاص خودشان را داشته باشند. وضعيت كار را از نظر مفهمي پيچيده تر از آنچه كه واقعاً هست، مي‌كند. اگرچه مفهوم سه تابع يكسان است، اما بايد سه نام مختلف را به خاطر بسپاريد. اين وضعيت در جاوا پيش نمي آيد، چرا كه متدهاي قدر مطلق مي‌توانند از نام مشتركي استفاده كنند. متدي به نام abs ( ) را در كتابخانه استاندارد كلاس هاي

جاوا وجود دارد. اين متد در كلاس match براي مديريت انواع مختلف داده ها overload شده است. جاوا بر اساس نوع آرگومان تصميم مي‌گيرد كه كدام نگارش تابع فراخوانده شود.
ارزش overload به خاطر آن است كه امكان دستيابي به متدهاي مرتبط به هم را از طريق كاربرد نام مشترك فراهم ساخته است. از اين رو، نام abs نمايانگر عمل عمومي است كه انجام مي‌شود. اين وظيفه كامپايلر است كه نگارش خاص مورد نظر را براي هر يك از شرايط انتخاب كند، و شما به عنوان برنامه ساز تنها بايد عمل عمومي كه انجام مي‌شود را به خاطر بسپاريد.
وقتي متدي را overload مي‌كنيد، هر يك از نگارشهاي آن مي‌توانند يكي از كارهاي مورد نظرتان را انجام دهند. هيچ قانوني مبني بر اينكه متدهاي overload شده بايد با يكديگر مرتبط باشند وجود ندارد. اما از منظر سبك كار، فرآيند overload كردن متدها به خوبي خود القاء كننده نوعي رابطه است. از اين رو، اگرچه با استفاده از نام مشترك مي‌توانيد متدهاي غير مرتبط را overload كنيد، اما توصيه مي‌شود اين كار را انجام ندهيد.

Overload كردن Constructorها

متدهاي Constructor را نيز مي‌توانيد همچون متدهاي معمولي overload كنيد. در حقيقت، در بيشتر كلاس هاي مربوط به كارهاي واقعي constructor هاي overload شده، نه تنها استثنا به شمار نمي آيند، بلكه كاملاً معمول خواهند بود.

استفاده از شيءها به عنوان پارامتر

تا به حال تنها از انواع داده هاي پايه و ساده به عنوان پارامتر متدها استفاده كرده ايم. اما، ارسال شيءها به متدها هم درست و هم متداول است. به عنوان مثال، برنامه كوتاه زير را در نظر بگيريد:

```
/ / 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   {
    publics 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 كه ملاحظه مي‌كنيد، متد ( ) equals در برنامة Test، برابر بودن دو شيء را مقايسه و نتيجه را بر مي‌گرداند. يعني، شيء فعال كننده متد را با شيء ارسالي مقايسه مي‌كند. اگر مقادير آنها يكسان باشد، در آن صورت مقدار حاصل متد، true خواهد بود. در غير اين صورت، حاصل آن false خواهد بود.

يكي از متداولترين كاربردهاي پارامترهاي نوع شيء، به استفاده از constructorها مربوط مي‌شود. غالباً نياز به ايجاد شيءهاي جديدي خواهد داشت كه مقدار دهي اوليه آنها مي‌بايست همچون يكي از شيءهاي موجود باشد. براي انجام اين كار بايد constructorاي تعريف كنيد تا شيئي از نوع كلاس خودش را به عنوان پارامتر دريافت كند. به عنوان مثال، نگارش زير از كلاس Box امكان مقدار دهي اوليه يك شيء با استفاده از يك شيء ديگر را فراهم مي‌سازد:

```
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 indicte
    height = -1; /  /  an uninitialized
    depth = -1;  /  / box
}

/  / constructor used when cube is created
Box (double 1en)  {
    width = height = depth = 1en;
}

/  / compute and return volume
double volume ( )   {
     return with * 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 vo1;

/ / get volume of first box
vo1 = mybox1.volume ( ) ;
system.out.println ("volume of mybox1 is " + vo1) ;
/ / get volume of second box
vo1= mebox2.volume ( ) ;
system.out.println ("volume of mybox2 is " + vo1) ;

/ / get volume of cube
vo1 = mycube.volume ( ) ;
system.out.println ("volume of clone is " +vo1) ;
   }
}
```

نگاهي دقيقتر به روند ارسال پارامترها

بطور كلي، در هر زبان برنامه سازي دو روش براي ارسال آرگومان ها به يك سابروتين وجود دارد. روش نخست، "call-by-value" نام دارد. در اين روش، مقدار آرگومان به پارامتر سابروتين كپي مي‌شود. بنابراين، تغييراتي كه در پارامتر اعمال مي‌شوند، هيچ تأثيري بر آرگومان نخواهند داشت. روش دوم، "call-by-reference" نام دارد. در اين روش، نشاني آرگومان (و نه مقدار آرگومان) به پارامتر ارسال مي‌شود. از اين نشاني در سابروتين براي دستيابي به خود آرگومان مشخ شده در عبارت فراخواني استفاده مي‌شود. اين بدين معناست كه تغييراتي كه در پارامتر اعمال مي‌شوند، بر آرگومان مورد استفاده براي فراخواني سابروتين تأثير خواهند گذاشت. همان گونه كه خواهيد ديد، جاوا بسته به چيزي كه ارسال مي‌شود، از هر دو روش استفاده مي‌كند. وقتي در جاوا يكي از انواع داده هاي پايه را به متدي ارسال مي‌كنيد، از روش نخست استفاده مي‌شود. از اين رو، آنچه براي پارامتر دريافت كننده آرگومان رخ مي‌دهد، بازتابي در خارج از متد نخواهد داشت.

وقتي شيئي را به متدي ارسال مي‌كنيد، وضعيت به طور چشمگيري تغيير مي‌كند، چرا كه شيءها با روش "call-by-reference" ارسال مي‌شوند. به خاطر داشته باشيد كه وقتي متغيري از نوع كلاس ايجاد مي‌كنيد، تنها نوعي نشاني به يك شيء ايجاد مي‌شود. از اين رو، وقتي اين نشاني را به متدي ارسال مي‌كنيد، پارامتر دريافت كننده آن، به همان شيئي ارجاع خواهد شد كه آرگومان متناظرش به آن ارجاع دارد. اين بدين معناست كه شيءها با روش "call-by-reference" به متدها ارسال مي‌شوند. تغييراتي كه در متد به شيء اعمال مي‌شوند، 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 satic void main (straing arges [  ])  {
      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.pritln ("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 نكته اي جالب بد نيست بدانيد كه وقتي نشاني شيئي به يك متد ارسال مي‌شود، خود نشاني به صورت "call-by-value" ارسال مي‌شود. اما، از آنجايي كه مقدار در دست ارسال، به شيء ارجاع دارد، كپي مقدار آن نيز به همان شيء متناظر با آرگومان ارجاع خواهد شد.

شيءهايي كه به عنوان خروجي متدها باز گردانده مي‌شوند

هر متد انواع مختلفي از داده ها را بازگرداند؛ از جمله انواع كلاس هايي را كه ايجاد مي‌كنيد. به عنوان مثال، متد ( ) incrByTen در برنامه صفحه بعد، شيئي را به عنوان خروجي باز مي‌گرداند 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 آنجايي كه تمام شيءها با استفاده از new به طور پويا تخصيص داده مي‌شوند، نيازي نيست كه به دليل پايان يافتن اجراي متدي كه شيء در آن ايجاد شده، نگران خارج شدن شيء از محدود دستيابي باشيد. تا زماني كه نشاني شيء را در جايي از برنامه خود داشته باشيد، دوره حيات آن ادامه خواهد يافت. هرگاه ديگر ارجاعي به شيء نباشد. حافظه آن در مرتبه آتي اجراي «سيستم باز پس گيري حافظه بلا استفاده» ، آزاد مي‌شود.

قابليت بازگشت

جاوا از مسئله قابليت بازگشت[11] پشتيباني مي‌كند. منظور از قابليت بازگشت، فرآيند تعريف كردن يك چيز بر حسب خودش مي‌باشد. «قابليت بازگشت» خصوصيتي است كه متدها امكان مي‌دهد تا خودشان را فرا بخوانند. متدي كه خودش را فرا بخواند، بازگشتي[12] ناميده مي‌شود.
مثال كلاسيك «قابليت بازگشت»، محاسبه فاكتوريل اعداد است. روش محاسبه فاكتوريل هر عدد با استفاده از روش «قابليت بازگشت» در ذيل نشان داده شده است:

```
/ / A simple example of recursion.
Class Factorial  {
      /  / this is a recusive 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 (staring args [  ])  {
         Factorial f = new Factorial ( );
```

[11]- Recursion.
[12]- Recursive.

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

وقتي متدي خود را فرا مي‌خواند، فضاي مورد نياز متغيرهاي محلي جديد و پارامترها در پشته تخصيص مي‌يابد، و قسمت اجرايي متد با همين متغيرهاي جديد از ابتدا اجرا مي‌شود. با پايان رسيدن نتيجه هر عبارت فراخواني، متغيرهاي محلي قديمي و پارامترها از روي پشته برداشته مي‌شوند، و اجرا از نقطه آغاز فرا خواني در خود متد ادامه مي‌يابد. اصطلاحاً گفته مي‌شود كه متدهاي بازگشتي همچون تلسكوپ‌ها، حركت رو به جلو و عقب دارند.

نگارشهاي بازگشتي بسياري از روتين ها ممكن است به دليل «سربار»[13] اضافي حاصل از فراخوانيهاي اضافي، قدري كندتر از معدلهاي تكراري آنها اجرا شود. بسياري از اين گونه فراخوانيها (براي متدهاي بازگشتي) ممكن است به سر ريز پشته منجر شود. چون فضاي لازم براي پارامترها و متغيرهاي محلي از پشت تأمين مي‌شود و هر عبارت فراخواني جديد موجب ايجاد نسخه جديدي از اين متغيرها مي‌شود، امكان دارد پشته با كمبود فضا مواجه شود. اگر چنين اتفاقي رخ دهد، سيستم زمان اجراي جاوا بروز استثنا خواهد شد. اما، احتمالاً نبايد نگران اين مسئله باشيد، مگر آنكه روتين بازگشتي مورد استفاده تان، درست عمل نكند.

مزيت اصلي متدهاي بازگشتي آن است كه با استفاده از آنها مي‌توان نگارشهاي شفافتر و ساده تري از الگوريتم ها را نسبت به معادلهاي تكراري خودشان ايجاد نمود. به عنوان مثال، پياده سازي الگوريتم مرتب سازي QuickSort با روش تكراري كاملاً دشوار است. به نظر مي‌رسد كه حل برخي از مسائل، به ويژه مسائل مرتبط با AI، با راه حلهاي بازگشتي آسانتر باشد. و بالاخره اينكه، روش تفكر بازشگتي براي بخري از افراد آسانتر از روش تفكري تكراري است.

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

---

[13]- Overhead.

مقدمه اي بر كنترل دستيابي

همان گونه كه مي‌دانيد، «نهان سازي»[14]، داده ها را با روتين هايي كه آنها را پردازش و مديريت مي‌كنند، مرتبط مي‌سازد. اما، «نهان سازي» خصوصيت مهم ديگري هم دارد: كنترل دستيابي. از طريق نهان سازي مي‌توانيد كنترل كنيد كه كدام قسمتهاي برنامه مي‌توانند به اعضاي كلاس مورد نظر دستيابي داشته باشند. با تحت كنترل در آوردن دستيابي مي‌توانيد از كار برد نادرست جلوگيري كنيد.

چگونگي دستيابي به هر كي از اعضاي يك كلاس به وسيله «مشخصه دستيابي»[15] مورد استفاده براي تعريف كردن آن تعيين مي‌شود. 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 كرده است. مشخصه دسيابي protected، تنها زماني اعمال مي‌شود كه وراثت مطرح باشد.

وقتي عضوي از يك كلاس به وسيله مشخصة public تعريف مي‌شود، در آن صورت آن عضو توسط هر روتين ديگري قابل دستيابي خواهد بود. وقتي عضوي از يك كلاس به وسيله مشخصة private تعريف مي‌شود، 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 آشنايي با چگونگي كاربرد 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.
```

[14]- encapsulation.
[15]- 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 كه ملاحظه مي‌كنيد، اينك هم stack، كه براي نگهداري پشته است، و هم tos، كه به عنوان شاخص عنصر روي پشته به كار برده مي‌شود، به صورت private تعريف شده اند. اين بدين معناست كه بدون استفاده از ( ) push و ( ) pop قابل دستيابي يا تغيير نمي باشند. Private ساختن tos، از دستيابي آن از ديگر بخشهاي برنامه و تغيير عمدي مقدار آن به مقداري كه در خارج از محدوده آراية stack باشد، جلوگيري مي‌كند.

عموماً، هر عضو از يك كلاس بايد تنها در ارتباط با شيئي از همان كلاس خودش مورد دستيابي قرار گيرد. اما، اين امكان وجود دارد كه بتوان اعضايي را ايجاد نمود كه به تنهايي و بدون ارجاع به نمونه خاصي از كلاس، قابل استفاده باشند. براي ايجاد اين گونه اعضا از كلمه كليدي static در ابتداي سطر تعريف آنها استفاده كنيد. وقتي عضوي از يك كلاس به صورت static تعريف مي‌شود، دستيابي به آن پيش از ايجاد شيئي از همان كلاس، و بدون ارجاع هر گونه شي مقدور

خواهد بود. هم متدها و هم متغيرها را مي‌توانيد، به صورت static تعريف مي‌شود، آن است كه بايد پيش از ايجاد هر شيئي قابل فراخواني باشد.

نمونه متغيرهايي كه به صورت static تعريف مي‌شود عملاً متغيرهاي عمومي مي‌شوند. وقتي شي هايي از كلاس يك متغير static تعريف مي‌شوند هيچ نسخه اي از آن متغير ايجاد نمي شود. در عوض، تمام نمونه هاي آن كلاس، متغيرهاي static يكساني را به اشتراك مي‌گذارند.

متدهايي كه به صورت static تعريف مي‌شوند، چندين محدوديت دارند:

تنها ساير متدهاي static را مي‌توانند فرابخوانند.

بايد تنها با داده هاي static كار كنند.

به هيچ عنوان نمي توانند از this يا super استفاده كنند (كلمه كليدي supper به وراثت مربوط مي‌شود و در فصل آتي شرح داده شده است.

اگر براي تعيين مقدار اوليه متغيرهاي static نياز به انجام محاسبات داريد، مي‌توانيد بلوكي را به صورت static تعريف كنيد تا دقيقاً تنها يك مرتبه هنگام بارگذاري اوليه كلاس اجرا شود. مثال زير كلاسي را نشان مي‌دهد كه يك متد static چند متغير static و يك بلوك static (براي مقدار دهي اوليه) 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 اينكه كلاس UseStatic بارگذاري[16] مي‌شود. تمام عبارات static اجرا مي‌شوند. نخست، مقدار ۳ به a تخصيص مي‌يابد، سپس بلوك static اجرا مي‌شود (پيامي را نمايش مي‌دهد)، و آخر مقدار حاصل a*4 به b تخصيص مي‌يابد. سپس ( ) main فراخوانده مي‌شود، و ۴۲ را هنگام فراخواني ( )meth براي x ارسال مي‌كند. سه عبارت println ( )، دو متغير static (a و b) و همين طور متغير محلي x را نمايش مي‌دهند.

خروجي برنامه در ذيل نشان داده شده است:

```
Static block initialized
x = 42
a = 3
b = 12
```

متدها و متغيرهاي ايستا را مي‌توان در خارج از كلاسي كه تعريف شده اند، مستقل از هر شيئي به كار برد. براي انجام اين كار كافي است نام كلاس و سپس عملگر نقطه (.) را پيش از نامشان بنويسيد. به عنوان مثال، چنانچه بخواهيد متد ايستايي را از خارج كلاس خودش فرا بخوانيد، مي‌توانيد اين كار را با استفاده از فرم كلي زير انجام دهيد:

```
Classname.method ( )
```

مروري بر آرايه ها

آرايه ها به صورت شيء پياده سازي مي‌شوند. به همين دليل است كه خصوصيت ويژه اي در رابطه با آرايه ها وجود دارد كه بد نيست از آن بهره مند شويد. اندازه هر آرايه ـ يعني، تعداد عناصري كه در آرايه قابل ذخيره اند، در نمونه متغير length آن است. تمام آرايه ها اين متغير را دارند، و اندازه آرايه هميشه در آن خواهد بود. برنامه زير اين خصوصيت را نشان مي‌دهد:

```
/  / 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.printl ("length of 1 is " + al.length) ;
      system.out.printl ("length of 2 is " + al.length) ;
      system.out.printl ("length l of 3 is " + al.length) ;
```

[16]- Load.

```
        }
}

length of a1 is 10
length of a2 is 8
length of a3 is 4
```

همان گونه كه ملاحظه مي‌كنيد، اندازه هر يك از آرايه ها نمايش داده شده است. به خاطر داشته باشيد كه مقدار length هيچ ارتباطي با تعداد عناصري كه در آرايه وجود دارند، ندارد. مقدار آن تنها نمايانگر تعداد عناصري است كه مي‌توان در آرايه ذخيره كرد.

مقدمه اي بر كلاس‌هاي داخلي و تودرتو

امكان ايجاد يك كلاس در هر كلاس ديگر فراهم شده است؛ به اين گونه كلاس ها، كلاس هاي تودرتو گفته مي‌شود. محدود اين كلاس ها به محدوده كلاسي كه در آن قرار دارند، محدود مي‌شود. از اين رو، اگر كلاس B و در كلاس A تعريف شود، در آن صورت B براي A شناخته شده خواهد بود، اما در خارج از آن خير. اين كلاس‌ها به اعضاي كلاسي كه در آن تعريف شده اند، دستيابي دارند، از جمله اعضاي private. اما در كلاس بيروني به اعضاي كلاسي بيروني به اعضاي كلاسي كه در خودش تعريف شده است، دستيابي ندارد.

دو نوع كلاس تودرتو وجود دارد: ايستا و غير ايستا. كلاس هاي تودرتوي ايستا، كلاس هايي هستند كه از static براي تعريف آنها استفاده مي‌شود. چون اين كلاس ها ايستا هستند، بايد از طريق يك شيء به اعضاي كلاسي كه در آن تعريف مي‌شوند، دستيابي داشته باشند. يعني، نمي توانند مستقيماً به اعضاي آن كلاس دستيابي داشته باشند. كلاس هاي تودرتوي ايستا به دليل محدوديت به ندرت مورد استفاده قرار مي‌گيرند.

مهمترين نوع از كلاس هاي تودرتو، كلاس هاي داخلي[17] هستند اين كلاس ها، غير ايستا مي‌باشند و به تمام متغيرها و متدهاي كلاس خارجي خود دستيابي دارند، و مي‌توانند با همان روش خاص اعضاي غير ايستا، مستقيماً به آنها ارجاع داشته باشند. از اين رو، هر كلاس داخلي كاملاً در محدوده كلاس در برگيرنده خود است.

[17]- Inner.

برنامه زير چگونگي تعريف و استفاده از يك كلاس داخلي را نشان مي‌دهد. كلاس Outer يك «نمونه متغير» به نام outer-x، يك «نمونه متد» به نام test () دارد و كلاسي به نام Inner در آن تعريف مي‌شود.

```
/ / 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.printl ("display: outer-x = " + outer - x) ;
    }
  }
}

class InnerClassDemo {
    public static void main (string args [ ] )  {
      outer outer = new outer ( ) ;
      outer.test ( );
    }
}

display: outer-x = 100
```

در اين برنامه، يك كلاس داخلي به نام Inner در محدوده كلاس outer تعريف مي‌شود. بنابراين، روتين هاي كلاس Inner مي‌توانند مستقيماً به متغير outer-x دستيابي داشته باشند.

مهم است بدانيد كه كلاس Inner تنها در محدوده كلاس outer شناخته شده است. چنانچه روتيني در خارج از كلاس outer اقدام به ايجاد نمونه اي از كلاس Inner نمايد، كامپايلر جاوا خطايي را نمايش خواهد داد. به طور كلي، كلاس هاي تودرتو، تفاوتي با ساير عناصر برنامه ندارند: تنها در محدوده اي كه در آن تعريف شده اند، شناخته شده مي‌باشند.

هر كلاس داخلي به تمام اعضاي كلاسي كه در آن تعريف شده است، دستيابي دارد، اما عكس اين مطلب صادق نيست. اعضاي كلاس داخلي تنها در محدوده همان كلاس شناخته شده اند و در كلاس خارجي قابل استفاده نيستند.

اگرچه كلاس هاي تودرتو در بيشتر برنامه هاي روزمره به كار برده نمي شود، اما هنگام مديريت رويدادها در اپلت ها واقعاً مفيد واقع مي‌شوند.

استفاده از كلاس هاي تودرتو در مجموعه مشخصات 1.0 جاوا مجاز نبود. استفاده از آنها از جاوا ۱/۱ آغاز شده است.

بررسي كلاس String

String احتمالاً متداولترين كلاس در كتابخانه كلاس هاي جاوا به شمار مي‌آيد. دليل بارز اين مطلب آن است كه رشته ها بخش بسيار مهمي از برنامه سازي به شمار مي‌آيند.

نخستين چيزي كه بايد درباره رشته ها بدانيد، آن است كه هر رشته اي كه ايجاد مي‌كنيد، در واقع شيئي از كلاس String است. حتي ثابتهاي رشت هاي هم شيء به شمار مي‌آيند. به عنوان مثال در عبارت زير،

```
System.out.println ("This is a string, too");
```

رشتة "This is a string, too" نوعي ثابت رشته اي به شمار مي‌آيد. خوشبختانه، روش مديريت ثابتهاي رشته اي در جاوا همچون مديريت رشته هاي «معمولي» در زبانهاي كامپيوتري ديگر است، بنابراين از اين جهت مشكلي نخواهيم داشت.

دومين مطلبي كه بايد درباره رشته ها بدايند آن است كه شيءهاي نوع String، تغيير ناپذير هستند. يعني پس از ايجاد شيءهاي String، محتواي آنها قابل تغيير نخواهد بود. اگرچه اين موضوع ممكن است محدوديت جدي به نظر آيد، اما به دو دليل اين گونه نيست:

- اگر نياز به تغيير رشته اي داشته باشيد، هميشه مي‌توانيد نمونه جديدي ايجاد كنيد كه متضن تغييرات مورد نظر باشد.
- كلاسي نظير String به نام StringBuffer در جاوا تعريف شده است كه امكان تغيير رشته ها را فرهم مي‌سازد، بنابراين تمام كارهاي پردازش مربوط به رشته ها هنوز در جاوا قابل انجام هستند (StringBufeer در بخش دوم كتاب بررسي 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 مي‌توانيم در هر شرايطي كه كاربرد رشته ها مجاز است، به كار بريد.

عملگر "+" در جاوا براي شيءهاي نوع String تعريف شده است. از آن براي ادغام دو رشته استفاده مي‌شود. به عنوان مثال، نتيجه عبارت زير،

```
String my string = "I" + "like" + "Java" ;
```

ذخيره شدن "I like Java" در MyString مي‌شود.

با استفاده از equals () مي‌توانيد تساوي دو رشته را بررسي كنيد. با فراخواني متد length () مي‌توانيد طول يك رشته به دست آوريد. با استفاده از charAt () هم مي‌توانيد كاراكتر 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 مي‌شود كه اطلاعاتي را هنگام اجراي يك برنامه براي آن ارسال كنيم. اين كار با استفاده از ارسال آرگومان هاي خط فرمان[18] به main () انجام مي‌شود. منظور از آرگومان خط فرمان، اطلاعاتي است كه هنگام اجراي برنامه، مستقيماً پس از نام برنامه در خط فرمان نوشته مي‌شوند. دستيابي به آرگومان هاي خط فرمان در برنامه هاي جاوا كاملاً آسان است ـ اين اطلاعات به صورت رشته اي در آراية String ارسالي به main () نگهداري مي‌شوند.

Varargs: آرگومان هاي با طول متغير

[18]- Command-line.

J2SE 5، ويژگي جديدي را به جاوا افزوده است كه ايجاد متدهايي را ساده مي‌كند كه نياز به تعداد متغيري آرگومان دارند. اين ويژگي، varargs ناميده شده است و از كلمات variable-length arguments گرفته شده است. متدي كه تعداد آرگومان هايش متغير است، متد variable-arity يا صرفاً متد varargs ناميده مي‌شود.

شرايطي كه تعداد آرگومان هاي يك متد متغير باشد، غير معمول به شمار مي‌آيند. به عنوان مثال، متدي كه باري اتصال به اينترنت به كار مي‌رود، نام كاربري، كلمه عبور، نام فايل، پروتكل و غيره را نياز خواهد داشت، اما در صورت عدم تأمين برخي از آنها مي‌توانيد از مقادير پيش فرض استفاده كنيد. در چنين شرايطي مي‌توانيد تنها آرگومان هايي كه مقادير پيش فرض در خصوص آنها قابل استفاده نيستند را ارسال نمود.

تا پيش از عرضة J2SE 5، دو روش براي مديريت آرگومان هاي با طول متغير وجود داشت كه هيچكدام از آنها خوشايند بود. نخست اينكه، اگر حداكثر تعداد آرگومان ها كوچك و مشخص بود، در آن صورت نگارشهاي مختلفي از متد به صورت overload شده باري هر يك از حالات فراخواني متد ايجاد مي‌شد. اگرچه اين روش براي برخي از شرايط عملياتي بود، اما تنها براي شرايط بسيار محدودي كاربرد داشت.

در شرايط كه حداكثر تعداد آرگومان ها، بزرگتر يا نامعين بود، از روش ديگري استفاده مي‌شد. در آن روش، آرگومان ها در آرايه اي نگهداري مي‌شدند و سپس آرايه به متد ارسال مي‌شد.

آرگومان ها با طول متغير به سه نقطه (...) مشخص مي‌شوند.

هر متد مي‌تواند پارامترهاي «متعارفي» همراه با يك پارامتر با طول متغير داشته باشد. اما پارامتر با طول متغير بايد هنگام تعريف متد، آخرين پارامتر باشد. به عنوان مثال، تعريف زير كاملاً قابل قبول است:

```
int doIt (int a, int b, double c, int … vals) {
```

به خاطر داشته باشيد كه varargs بايد آخرين پارامتر باشد.

محدوديت ديگري نيز وجود دارد كه بايد نسبت به آن آگاه باشيد: تنها يك پارامتر varargs مجاز است. به عنوان مثال، عبارت زير نادرست است:

```
int doIt (int a, int b, double c, int … vals, double … morevals)
{ / / Error!
```

تعريف پارامتر Varargs دوم غير مجاز است.

Overload کردن متدهاي Varargs

متدهايي كه آرگومان با طول متغير دارند را نيز مي‌تواند Overload نمود. به عنوان مثال، در برنامه زير، 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
```

اين برنامه هر دو روش overload كردن متدهاي varargs را نشان مي‌دهد. نخست اينكه، نوع داده هاي پارامتر varargs ممكن است مختلف باشد. در خصوص متدهاي vaTest (int …) و vaTest (Boolean …) همين گونه است. به خاطر داشته باشيد كه وجود "..." سبب مي‌شود تا با پارامتر به صورت آرايه اي از نوع مشخص شده برخورد شود.

روش دوم براي overload كردن متدهاي varargs، افزودن يك پارامتر معمولي است.

وراثت

Inheritance

عناوين اين بخش :

مباني وراثت

كاربرد كلمه كليدي super

Multilevel

زمان فراخواني Constructor ها

Override كردن متدها

توزيع (dispatch) پوياي متدها

چرا متدهاي لغو شده ؟

استفاده از كلاسهاي مجرد (abstract)

استفاده از Final با وراثت

كلاس Object

وراثت را يكي از سنگ بناهاي برنامه نويسي شي گراست ، زيرا امكان ايجاد طبقه بنديهاي سلسله مراتبي را بوجود مي آورد . با استفاده از وراثت ، مي توانيد يك كلاس عمومي بسازيد كه ويژگيهاي مشترك يك مجموعه اقلام بهم مرتبط را تعريف نمايد . اين كلاس بعدا ممكن است توسط ساير كلاسها بارث برده شده و هر كلاس ارث برنده چيزهايي را كه منحصر بفرد خودش باشد به آن اضافه نمايد . در روش شناسي جاوا ، كلاسي كه بارث برده مي شود را كلاس بالا (superclass)مي نامند . كلاسي كه عمل ارث بري را انجام داده و ارث برده است را زير كلاس (subclass)مي نامند . بنابراين ، يك " زير كلاس " روايت تخصصي تر و مشخص تر از يك " كلاس بالا " است . زير كلاس ، كليه متغيرهاي نمونه و روشهاي توصيف شده توسط كلاس بالا را بارث برده و منحصر بفرد خود را نيز اضافه مي كند.

مباني وراثت

براي ارث بردن از يك كلاس ، خيلي ساده كافيست تعريف يك كلاس را با استفاده از واژه كليدي extendsدر كلاس ديگري قرار دهيد . براي فهم كامل اين مطلب ، مثال ساده اي را نشان مي دهيم . برنامه بعدي يك كلاس بالا تحت نام A و يك زير كلاس موسوم به B ايجاد مي كند . دقت كنيد كه چگونه از واژه كليدي extends استفاده شده تا يك زير كلاس از A ايجاد 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 اين برنامه ، بقرار زير مي 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
```

همانطوريكه مي بينيد ، زير كلاس B دربرگيرنده كليه اعضا كلاس بالاي مربوطه يعني A است . بهمين دليل است كه subob مي تواند به i و j دسترسي داشته و showij() را فراخواني نمايد . همچنين داخل sum() مي توان بطور مستقيم i وj و همانگونه كه قبلا بخشي از B بودند ، ارجاع نمود . اگرچه A كلاس بالاي B مي باشد ، اما همچنان يك كلاس كاملا" مستقل و متكي بخود است .كلاس بالا بودن براي يك زير كلاس بدان معني نيست كه نمي توان خود آن كلاس بالا را بتنهايي مورد استفاده قرار داد . بعلاوه ، يك زير كلاس مي تواند كلاس بالاي يك زير كلاس ديگر باشد. شكل عمومي اعلان يك class كه از يك كلاس بالا ارث مي برد ، بصورت زير است :

```
class subclass-name extends superclass-name {
// body of class
}
```

براي هر زير كلاسي كه ايجاد مي كنيد ، فقط يك كلاس بالا مي توانيد تعريف كنيد .جاوا از انتقال وراثت چندين كلاس بالا به يك كلاس منفرد پشتيباني نمي كند . (از اين نظر جاوا باC++ متفاوت است كه در آن وراثت چند كلاسه امكان پذير است. (قبلا گفتيم كه مي توانيد يك سلسله مراتب از وراثت ايجاد كنيد كه در آن يك زير كلاس ، كلاس بالاي يك زير كلاس ديگر باشد . اما ، هيچ كلاسي نمي تواند كلاس بالاي خودش باشد .

دسترسي به اعضا و وراثت

اگرچه يك زير كلاس دربرگيرنده كليه اعضائكلاس بالاي خود مي باشد، اما نميتواند به اعضايي از كلاس بالا كه بعنوان private اعلان شده اند ، دسترسي داشته باشد . بعنوان مثال ، سلسله مراتب ساده كلاس زير را در نظر بگيريد :

```
/* 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);
}
}
```

اين برنامه كامپايل نخواهد شد زيرا ارجاع به j داخل روش sum() در B ر سبب نقض دسترسي خواهد شد. از آنجاييكه j بعنوان private اعلان شده ، فقط توسط ساير اعضائ كلاس خودش قابل دسترسي است و زير كلاسها هيچگونه دسترسي به آن ندارند.

يادآوري : يك عضو كلاس كه بعنوان private اعلان شده براي كلاس خودش اختصاصي خواهد بود . اين عضو براي كدهاي خارج از كلاسش از جمله زير كلاسها ، قابل دسترسي نخواهد بود .

يك مثال عملي تر

اجازه دهيد به يك مثال عملي تر بپردازيم كه قدرت واقعي وراثت را نشان خواهد داد . در اينجا ، روايت نهايي كلاس Box بنحوي گسترش يافته تا يك عنصر چهارم تحت نام weight را دربرگيرد . بدين ترتيب ، كلاس جديد شامل width ، height، depth و weight ويك 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 زير مي باشد :

```
Volume of mybox1 is 3000
Weight of mybox1 is 34.3

Volume of mybox2 is 24
Weight of mybox2 is 0.076
Boxweight
```

كليه مشخصات Box را بارث برده و به آنها عنصر weight را اضافه مي كند . براي Boxweight ضرورتي ندارد كه كليه جوانب موجود در Box را مجددا ايجاد نمايد . بلكه مي تواند بسادگي Box را طوري گسترش دهد تا اهداف خاص خودش را تامين نمايد.

يك مزيت عمده وراثت اين است كه كافيست فقط يكبار يك كلاس بالا ايجاد كنيد كه خصلتهاي مشترك يك مجموعه از اشيائ را تعريف نمايد ، آنگاه مي توان از آن براي

ايجاد هر تعداد از زير كلاسهاي مشخص تر استفاده نمود . هر زير كلاس مي تواند دقيقا" با طبقه بندي خودش تطبيق يابد . بعنوان مثال ، كلاس بعدي ، از Box بارث برده و يك خصلت رنگ (color) نيز 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;
}
}
```

بياد آوريد كه هرگاه يك كلاس بالا ايجاد نماييد كه وجوه عمومي يك شي را تعريف كند ، مي توان از آن كلاس بالا براي تشكيل كلاسهاي تخصصي تر ارث برد . هر زير كلاس خيلي ساده فقط خصلتهاي منحصر بفرد خودش را اضافه مي كند . اين مفهوم كلي وراثت است . يك متغير كلاس بالا مي تواند به يك شي زير كلاس ارجاع نمايد يك متغير ارجاع مربوط به يك كلاس بالا را مي توان به ارجاعي ، به هر يك از زير كلاسهاي مشتق شده از آن كلاس بالا ، منتسب نمود . در بسياري از شرايط ، اين جنبه از وراثت كاملا مفيد و سودمند است . بعنوان مثال ، مورد زير را در نظر بگيريد :

```
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 اينجا weightbox يك ارجاع به اشيائ Boxweight است و plainbox يك ارجاع به اشيائ Boxاست . از آنجاييكه Boxweight يك زير كلاس از Box است ، مي توانplainbox را بعنوان يك ارجاع به شي ئ weightbox منتسب نمود . نكته مهم اين است كه نوع متغير ارجاع و نه نوع شيئي كه به آن ارجاع شده است كه تعيين مي كند كدام اعضائ قابل دسترسي هستند . يعني هنگاميكه يك ارجاع مربوط به يك شي ئ زير كلاس ، به يك متغير ارجاع كلاس بالا منتسب مي شود ، شما فقط به آن بخشهايي از شي ئ دسترسي داريد كه توسط كلاس بالا تعريف شده باشند . بهمين دليل است كه plainbox نمي تواند به weight دسترسي داشته باشد حتي وقتي كه به يك شي Boxweight ارجاع مي كند . اگر به آن فكر كنيد ، آن را احساس مي كنيد زيرا يك كلاس بالا آگاهي و احاطه اي نسبت به موارد اضافه شده به زيركلاس مربوطه اش نخواهد داشت . بهمين دليل است كه آخرين خط از كد موجود در قطعه قبلي از توضيح رج شده است . براي يك ارجاع Box امكان ندارد تا به فيلد weight دسترسي داشته باشد ، چراكه فيلدي با اين نام توسط Box تعريف نشده است.

كاربرد كلمه كليدي super

در مثالهاي قبلي كلاسهاي مشتق شده از Box به كارايي و قدرتمندي كه امكان داشت ، پياده سازي نشدند . بعنوان مثال ، سازنده Boxweight بطور صريحي فيلدهاي width ، height و depthو در Box() را مقدار دهي اوليه مي كند . اين امر نه تنها كدهاي پيدا شده در كلاس بالاي آنها را دو برابر مي كند كه غير كاراست ، بلكه دلالت دارد بر اينكه يك زير كلاس بايد دسترسي به اين اعضائ داشته باشد . اما شرايطي وجود دارند كه مي خواهيد يك كلاس بالا ايجاد كنيد كه جزئيات پياده سازي خودش را خودش نگهداري كند . در اين شرايط ، راهي براي يك زير كلاس

وجود ندارد تا مستقيما به اين متغيرهاي مربوط به خودش دسترسي داشته و يا آنها را مقداردهي اوليه نمايد . از آنجاييكه كپسول سازي يك خصلت اوليه oop است ، پس باعث تعجب نيست كه جاوا راه حلي براي اين مشكل فراهم كرده باشد . هرگاه لازم باشد تا يك زير كلاس به كلاس بالاي قبلي خودش ارجاع نمايد ، اينكار را با استفاده از واژه كليدي super انجام مي دهيم. super دو شكل عمومي دارد . اولين آن سازنده كلاس بالا را فراخواني مي كند . دومين آن بمنظور دسترسي به يك عضو كلاس بالا كه توسط يك عضو زير كلاس مخفي مانده است ، استفاده مي شود .

استفاده از super

يك زيركلاس ميتواند روش سازنده تعريف شده توسط كلاس بالاي مربوطه را بااستفاده از اين شكل super فراخواني نمايد :

```
super( parameter-list);
```

در اينجا parameter-list مشخص كننده هر پارامتري است كه توسط سازنده در كلاس بالا مورد نياز باشد. super() بايد همواره اولين دستور اجرا شده داخل يك سازنده زير كلاس باشد.

بنگريد كه چگونه از super() استفاده شده ، و همچنين اين روايت توسعه يافته از كلاس Boxweight() را در نظر بگيريد :

```
// 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 اينجا Boxweight() فراخواني super() را با پارامترهاي w، h و d و انجام مي دهد. اين كار سبب فراخوانده شدن سازنده Box() شده با استفاده از اين مقادير width ، height،وdepth و را مقدار دهي اوليه مي كند . ديگر Boxweight خودش اين مقادير اوليه را مقدار دهي نمي كند . فقط كافي است تا مقدار منحصر بفرد خود weightرا مقدار دهي اوليه نمايد . اين عمل Box را آزاد مي گذارد تا در صورت تمايل اين مقادير را private بسازد .

در مثال قبلي ، super() با سه آرگومان فراخواني شده بود . اما چون سازندگان ممكن است انباشته شوند ، مي توان super() را با استفاده از هر شكل تعريف شده توسط كلاس بالا فراخواني نمود . سازنده اي كه اجرا مي شود ، همانی است كه با آرگومانها مطابقت داشته باشد . بعنوان مثال ، در اينجا يك پياده سازي كامل ازBoxweight وجود دارد كه سازندگان را براي طرق گوناگون و ممكن ساخته شدن يك box فراهم مي نمايد. در هر حالت super() با استفاده از آرگومانهاي تقريبي فراخواني ميشود. دقت كنيد كهwidth , height و depth و داخل Box بصورت اختصاصي درآمده اند .

```
// A complete implementation of BoxWeight.
class Box {
private double width;
private double heght;
private double deoth;
// 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 construtors.
class BoxWeight extends Box {
double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob ){ // pass object to consructor
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();
}
}
```

اين برنامه خروجي زير را توليد مي كند :

```
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 بيشتري نسبت به اين سازنده در Boxweight() داشته باشيد:

```
// construct clone of an object
BoxWeight(BoxWeight ob ){ // pass object to constructor
super(ob);
weight = ob.weight;
}
```

توجه كنيد كه super() با يك شي ئاز نوع Boxweight نـه از نـوع Box فراخـواني شده است و نيز سازنده Box (Box ob) را فراخواني مي كند . همانطوريكه قبلا ذكر شد ، يك متغير كلاس بالا را مي توان براي ارجاع به هر شي مشتق شده از آن كلاس مورد استفاده قرار داد . بنابراين ، ما قادر بوديم يك شي Boxweightرا بـه سازنده Box گذر دهيم . البتـه Box فقط نسبت بـه اعضائ خودش آگاهي دارد .

اجازه دهيد مفاهيم كليدي مربوط بـه super() را مرور نماييم . وقتي يك زيركلاس super() را فراخواني مي كند ، در اصل سازنده كلاس بالاي بلافصل خود را فراخواني مـي كند . بنـابراين super()همواره به كلاس بالاي بلافصل قرار گرفته در بالاي كلاس فراخوانده شده ، ارجاع مـي كند . اين امر حتي در يك سلسله مراتب چند سطحي هم صادق است . همچنين super بايد همواره اولين دستوري باشد كه داخل يك سازنده زير كلاس اجرا مي شود .

دومين كاربرد super

دومين شكل super تا حدودي شبيه this كار مي كند، بجز اينكه super همواره بـه كـلاس بـالاي زير كلاسي كه در آن استفاده مي شود ، ارجاع مي كند . شكل عمومي اين كـاربرد بصـورت زير است :

```
Super .member
```

در اينجا ، member ممكن است يك روش يا يك متغير نمونه باشد . اين دومين شكل super براي شرايطي كاربرد دارد كه در آن اسامي اعضائ يك زير كلاس ، اعضا با همان اسامي را در كلاس بالا مخفي مي سازند . اين سلسله مراتب ساده كلاس را در نظر بگيريد :

```
// 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)(;
}
}
```

اين برنامه خروجي زير را نمايش مي دهد :

```
i in superclass :1
i in subclass :2
```

اگرچه متغير نمونه i در B ر متغير i در A ر را پنهان مي سازد ، اما super امكان دسترسي به i تعريف شده در كلاس بالا بوجود مي آورد . همانطوريكه خواهيد ديد همچنين ميتوان از super براي فراخواني روشهايي كه توسط يك زير كلاس مخفي شده اند.

ايجاد يك سلسله مراتب چند سطحي (Multilevel)

مي توانيد سلسله مراتبي بسازيد كه شامل چندين لايه وراثت بدلخواه 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 وC داشته باشيم آنگاه C مي تواند يك زير كلاس ازB و يك زير كلاس از A باشد . وقتي چنين شرايطي اتفاق مي افتد ، هر زير كلاس كليه خصلتهاي موجود در كليه كلاس بالاهاي خود را بارث مي برد . در اين شرايط ، C كليه جنبه هايB وA 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 كليه 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 كاربرد 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 نكته 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 كليه 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 .

نكته : 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 ترتيبي 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
```

همانطوريكه مشاهده مي كنيد، سازندگان بترتيب مشتق شدنشان فراخواني مي شوند. اگر درباره آن تفكر كنيد ، مي فهميد كه توابع سازنده بترتيب مشتق شدنشان اجرا مي شوند . چون يك كلاس بالا نسبت به زير كلاسهاي خود آگاهي ندارد ، هر گونه مقدار دهي اوليه كه براي اجرا شدن نياز داشته باشد، جدا از و احتمالا پيش نياز هر گونه مقدار دهي اوليه انجام شده توسط زير كلاس بوده و بنابراين ، بايد اول اين كار انجام شود.

Override كردن متدها

در يك سلسله مراتب كلاس ، وقتي يك روش در يك زير كلاس همان نام و نوع يك روش موجود در كلاس بالاي خود را داشته باشد، آنگاه ميگويند آن روش در زير كلاس ، روش موجود در كلاس بالا را لغو نموده يا از پيشروي آن جلوگيري مي نمايد . وقتي يك روش لغو شده از داخل يك زير كلاس فراخواني مي شود ، همواره به روايتي از آن روش كه توسط زير كلاس تعريف شده ، ارجاع خواهد نمود و روايتي كه كلاس بالا از همان روش تعريف نموده ، پنهان خواهد شد . مورد زير را در نظر بگيريد :

```
// 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 توليد شده توسط اين برنامه بقرار زير مي باشد :

```
K : 3
```

وقتي show() روي يك شي ئ از نوع B فراخواني مي شود ، روايتي از show كه داخل B تعريف شده مورد استفاده قرار ميگيرد. يعني كه ، روايت show() داخل B ، روايت اعلان شده در A را لغو مي كند .

اگر مي خواهيد به روايت كلاس بالاي يك تابع لغو شده دسترسي داشته باشيد ، اين كار را با استفاده از super انجام دهيد . بعنوان مثال ، در اين روايت از B روايت كلاس بالاي show() داخل روايت مربوط به زير كلاس فراخواني خواهد شد . اين امر به كليه متغيرهاي نمونه اجازه مي دهد تا بنمايش درآيند .

```
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 اين روايت از A را در برنامه قبلي جايگزين نماييد، خروجي زير را مشاهده مي كنيد :

```
i and j :1 2
k:3
```

در اينجا super.show()، روايت كلاس بالاي show() را فراخواني مي كند . لغو روش فقط زماني اتفاق مي افتد كه اسامي و نوع دو روش يكسان باشند. اگر چنين نباشد ، آنگاه دو روش خيلي ساده انباشته (overioaded) خواهند شد . بعنوان مثال ، اين روايت اصلاح شده مثال قبلي را در نظر بگيريد:

```
// 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 توليد شده توسط اين برنامه بقرار زير مي باشد :

```
This is k:3
i and j :1 2
```

روايت show()در B ر يك پارامتر رشته اي مي گيرد. اين عمل سبب متفاوت شدن تاييديه نوع آن از نوع موجود در A شده ، كه هيچ پارامتري را نمي گيرد. بنابراين نباشتگي (يا مخفي شدن اسم) اتفاق نمي افتد.

توزيع (dispatch) پوياي متدها

اگر در لغو روشها چيزي فراتر از يك قرارداد فضاي نام وجود نداشت ، آنگاه اين عمل در بهترين حالت ، ارضا نوعي حس كنجكاوي و فاقد ارزش عملي بود. اما اين چنين نيست . لغو روش تشكيل دهنده اساس يكي از مفاهيم پرقدرت در جاوا يعني" توزيع پوياي روش " است . اين يك مكانيسم است كه توسط آن يك فراخواني به تابع لغو شده در حين اجرا ، در عوض زمان كامپايل ، از سر گرفته مي شود. توزيع پوياي روش مهم است چون طريقي است كه جاوا با آن چند شكلي را درست حين اجرا پياده سازي مي نمايد.

توضيح را با تكرار يك اصل مهم شروع ميكنيم : يك متغير ارجاع كلاس بالا ميتواند به يك شي ئ زير كلاس ارجاع نمايد . جاوا از اين واقعيت استفاده كرده و فراخواني به روشهاي لغو شده را حين اجرا از سر مي گيرد . وقتي يك روش لغو شده از طريق يك ارجاع كلاس بالا فراخواني مي شود، جاوا براساس نوع شي ئ ارجاع شده در زماني كه فراخواني اتفاق مي افتد ، تعيين مي كند كه كدام روايت از روش بايد اجرا شود .

بنابراين ، عمل تعيين روايت خاص از يك روش ، حين اجرا انجام مي گيرد . وقتي به انواع مختلف اشيائارجاع شده باشد، روايتهاي مختلفي از يك روش لغو شده فراخواني خواهند شد . بعبارت ديگر ، اين نوع شي ئ ارجاع شده است) نه نوع متغير ارجاع (كه تعيين مي كند كدام

روايت از روش لغو شده بايد اجرا شود . بنابراين اگر يك كلاس بالا دربرگيرنده يك روش لغو شده توسط يك زير كلاس باشد ، آنگاه زماني كه انواع مختلف اشيائ از طريق يك متغير ارجاع كلاس بالا مورد ارجاع قرار مي گيرند روايتهاي مختلف آن روش اجرا خواهند شد .
در اينجا مثالي را مشاهده ميكنيد كه توزيع پوياي روش را به شما نشان ميدهد :

```
// 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 را ايجاد مي كند. زير كلاسهاي B و C 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 شيئي 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 .

نكته : كساني كه با C++ آشنا هستند تشخيص مي دهند كه روشهاي لغو شده در جاوا مشابه توابع مجازي (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 كليه مشتقات آن كلاس مشترك باشند ، و به زير كلاس ها اجازه مي دهد تا پياده سازيهاي مشخص برخي يا كليه 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، كليه اجزائي كه يك زير كلاس مي تواند بطور مستقيم استفاده نمايد ، تعريف مي كند. اين امر به زير كلاس قابليت انعطاف تعريف روشهاي خودش را مي دهد ، كه همچنان يك رابط منسجم را بوجود مي آورد . بنابراين ، بوسيله تركيب وراثت با روشهاي لغو شده ، يك كلاس بالا مي تواند شكل عمومي روشهايي را كه توسط كليه زير كلاسهاي مربوطه استفاده خواهند شد را تعريف نمايد.

چند شكلي پويا و حين اجرا يكي از قدرتمندترين مكانيسمهايي است كه طراحي شي گرايي را مجهز به استفاده مجدد و تنومندي كدها نموده است . اين ابزار افزايش دهنده قدرت كتابخانه هاي كدهاي موجود براي فراخواني روشهاي روي نمونه هاي كلاسهاي جديد بدون نياز به كامپايل مجدد مي باشد در حاليكه يك رابط مجرد و زيبا را نيز حفظ مي كنيم .

بكار بردن لغو روش

اجازه دهيد تا به يك مثال عملي تر كه از لغو روش استفاده مي كند ، نگاه كنيم . برنامه بعدي يك كلاس بالا تحت نام Figure را ايجاد مي كند كه ابعاد اشيا مختلف دو بعدي را ذخيره مي كند . اين برنامه همچنين يك روش با نام area() را تعريف مي كند كه مساحت يك شي را محاسبه مي كند. برنامه ، دو زير كلاس از Figure مشتق مي كند . اولين آن Rectangle و دومين آن Triangle است . هر يك از اين زير كلاسها area() را طوري لغو ميكنند كه بترتيب مساحت يك مستطيل و مثلث را برگردان كنند.

```
// 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 طريق مكانيسم دوگانه وراثت و چند شكلي حين اجرا ، امكان تعريف يك رابط منسجم كه براي چندين نوع اشيائ مختلف ، اما بهم مرتبط ، استفاده مي شود ، وجود دارد . در اين حالت ، اگر از Figureيك شي ئ مشتق شود ، پس با فراخواني area() مي توان مساحت آن شي ئ را بدست آورد . رابط مربوط به اين عمليات صرفنظر از نوع شكل هندسي مورد استفاده ، هميشه يكسان است.

استفاده از كلاسهاي مجرد (abstract)

شرايطي وجود دارد كه ميخواهيد يك كلاس بالا تعريف نماييد كه ساختار يك انتزاع معين را بدون يك پياده سازي كامل از هر روشي ، اعلان نمايد. يعني گاهي مي خواهيد يك كلاس بالا ايجاد كنيد كه فقط يك شكل عمومي شده را تعريف كند كه توسط كليه زير كلاسهايش باشتراك گذاشته خواهد شد و پر كردن جزئيات اين شكل عمومي بعهده هر يك از زير كلاس ها واگذار مي شود . يك چنين كلاسي طبيعت روشهايي كه زير كلاسها بايد پياده سازي نمايند را تعريف مي كند . يك شيوه براي وقوع اين شرايط زماني است كه يك كلاس بالا توانايي ايجاد يك پياده سازي با معني براي يك روش را نداشته باشد . تعريف area() خيلي ساده يك نگهدارنده مكان(place holder) است . اين روش مساحت انواع شي ئ را محاسبه نكرده و نمايش نمي دهد . هنگام ايجاد كتابخانه هاي خاص كلاس خود ، خواهيد ديد كه غير معمول نيست اگر يك روش هيچ تعريف بامعني در متن (context) كلاس بالاي خود نداشته باشد . اين شرايط را بدو طريق مي توانيد اداره نماييد . يك طريق اين است كه يك پيام هشدار(warning) گزارش نماييد . اگرچه اين روش در برخي شرايط خاص مثل اشكال زدايي (debugging) مفيد است ، اما روش دائمي نيست . ممكن است روشهايي داشته باشيد كه بايد توسط زير كلاس لغو شوند تا اينكه آن زيركلاس معنادار بشود.

كلاس Triangle را در نظر بگيريد . اگر area() تعريف نشود، اين كلاس هيچ معنايي ندارد . در اين حالت ، شما بدنبال راهي هستيد تا مطمئن شويد كه يك زير كلاس در حقيقت كليه روشهاي ضروري را لغو مي كند . راه حل جاوا براي اين مشكل روش مجرد abstract method است . مي توانيد توسط زير كلاسها و با مشخص نمودن اصلاح كننده نوع abstract ، روشهاي معيني را لغو نماييد . به اين روشها گاهي subclasser responsibilty اطلاق ميشود ، زيرا آنها هيچ پياده سازي مشخص شده اي در كلاس بالا ندارند . بنابراين يك زيركلاس بايد آنها را لغو نمايد چون نمي تواند بسادگي روايت تعريف شده در كلاس بالا را استفاده نمايد . براي اعلان يك روش مجرد ، از شكل عمومي زير استفاده نماييد .

```
abstract type name( parameter-list);
```

همانطوريكه مشاهده مي كنيد در اينجا بدنه روش معرفي نشده است . هر كلاسي كه دربرگيرنده يك يا چند روش مجرد باشد ، بايد بعنوان مجرد اعلان گردد . براي اعلان يك كلاس بعنوان مجرد ، بسادگي از واژه كليدي abstract در جلوي واژه كليدي class در ابتداي اعلان كلاس استفاده مي نماييد . براي يك كلاس مجرد هيچ شيئي نمي توان ايجاد نمود . يعني يك كلاس مجرد نبايد بطور مستقيم با عملگرnew نمونه سازي شود . چنان اشيائي بدون استفاده هستند ، زيرا يك كلاس مجرد بطور كامل تعريف نشده است . همچنين نمي توانيد سازندگان مجرد يا روشهاي ايستاي مجرد اعلان نماييد . هر زير كلاس از يك كلاس مجرد بايد يا كليه روشهاي مجرد موجود در كلاس بالا را پياده سازي نمايد ، و يا خودش بعنوان يكabstract اعلان شود . در اينجا مثال ساده اي از يك كلاس با يك روش مجرد مشاهده مي كنيد كه بعد از آن يك كلاس قرار گرفته كه آن روش را پياده سازي مي كند :

```
// 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 كنيد كه هيچ شيئي از كلاس A در برنامه اعلان نشده است . همانطوريكه ذكر شد ، امكان نمونه سازي يك كلاس مجرد وجود ندارد . يك نكته ديگر : كلاس A يك روش واقعي با نام callmetoo()را پياده سازي مي كند . اين امر كاملا مقبول است. كلاسهاي مجرد مي توانند ماداميكه تناسب را حفظ نمايند ، دربرگيرنده پياده سازيها باشند.

اگرچه نمي توان از كلاسهاي مجرد براي نمونه سازي اشيائ استفاده نمود، اما از آنها براي ايجاد ارجاعات شي مي توان استفاده نمود زيرا روش جاوا براي چند شكلي حين اجرا از طريق استفاده از ارجاعات كلاس بالا پياده سازي خواهد شد . بنابراين ، بايد امكان ايجاد يك ارجاع به يك كلاس مجرد وجود داشته باشد بطوريكه با استفاده از آن ارجاع به يك شي زير كلاس اشاره نمود . شما استفاده از اين جنبه را در مثال بعدي خواهيد ديد.

با استفاده از يك كلاس مجرد، مي توانيد كلاس Figure را توسعه دهيد. چون مفهوم با معنايي براي مساحت يك شكل دو بعدي تعريف نشده وجود ندارد ، روايت بعدي اين برنامه area()را بعنوان يك مجرد داخل Figure اعلان مي كند . اين البته بدان معني است كه كليه كلاسهاي مشتق شده از Figure بايد area()را لغو نمايند .

```
// 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
abstruct double area();
}
class Rectangle extends Figure {
Rectangle(duoble 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))(;
}
}
```

همانطوريكه توضيح درون main() نشان مي دهد ، ديگر امكان اعلان اشيائ از نوع Figure وجود ندارد ، چون اكنون بصورت مجرد است . كليه زير كلاسهاي Figure بايدarea() را لغو نمايند . براي اثبات اين امر، سعي كنيد يك زير كلاس ايجاد نماييد كه area() را لغو نمي كند . حتما يك خطاي complle-time در زمان كامپايل دريافت مي كنيد.

اگرچه امكان ايجاد يك شي از نوع Figure وجود ندارد، اما مي توانيد يك متغير ارجاع از نوع Figureايجاد نماييد . متغير fighref بعنوان ارجاعي به Figure اعلان شده و بدان معني است كه با استفاده ازآن مي توان به يك شي از هر كلاس مشتق شده از Figure ، ارجاع نمود . همانطوريكه توضيح داديم ، تعيين اينكه كدام نگارش از متدهاي override شده بايد در زمان اجرا فعال شوند ، از طريق متغيرهاي ارجاع به فوق كلاس ها صورت ميگيرد.

استفاده از Final با وراثت

واژه كليدي final سه كاربرد دارد . اول براي ايجاد مشابه يك ثابت اسم دار. دو كاربرد ديگر مربوط به وراثت هستند و بررسي خواهند شد . استفاده از final براي ممانعت از لغو كردن در حاليكه لغو كردن روش يكي از جنبه هاي قدرتمند جاوا است ، اما زمانهايي وجود دارند كه مي خواهيد مانع وقوع لغو روش گرديد. براي غير مجاز كردن لغو يك روش ، final را بعنوان يك اصلاحگر در شروع اعلان آن مشخص نماييد . روشهاي اعلان شده بعنوان final نمي توانند لغو شوند . قطعه بعدي 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 ، نمي توان آن را در B لغو نمود . اگر چنين تلاشي بكنيد ، نتيجه يك خطاي complle-time خواهد بود . روشهاي اعلان شده بعنوان final گاهي مي توانند باعث افزايش عملكرد شوند:

چون كامپايلر مي داند كه آنها توسط يك زير كلاس لغو نخواهند شد، كامپايلر از فراخوانيinline به آنها آزاد مي شود. وقتي يك تابع كوچك final فراخواني ميشود اغلب كامپايلر جاوا مي تواند كد بايتي را براي زير روال مستقيم درون خطي (inline) كد كامپايل شده در روش فراخواننده كپي نمايد، و بدين ترتيب بالاسريهاي پر هزينه همراه يك فراخوان روش را حذف مي نمايد Inlining . فقط يك گزينه با روشهاي final است . بطور طبيعي ، جاوا فراخواني به روشها را بصورت پويا و در زمان حين اجرا حل و فصل مي كند. اين عمل را late binding مي نامند . اما از آنجاييكه روشهاي final را نمي توان لغو نمود ، يك فراخواني را مي توان در زمان كامپايل حل و فصل نمود . اين را early binding مي نامند.

استفاده از final براي جلوگيري از وراثت

گاهي ممكن است بخواهيد مانع ارث بردن از يك كلاس بشويد . براي انجام اينكار قبل از اعلان كلاس از واژه كليدي final استفاده نماييد. اعلان نمودن يك كلاس بعنوان final بطور ضمني كليه روشهاي آن را نيز بعنوان final اعلان ميكند. حتما مي دانيد كه اعلان يك كلاس بعنوان هم abstractو هم final غير مجاز است چون يك كلاس abstract بتنهايي كامل نبوده و براي تكميل پياده سازيها متكي به زيركلاسهاي خود مي باشد . در اينجا يك مثال از كلاس final را مشاهده مي كنيد :

```
final class A {
//...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
//...
}
```

همانطوريكه از " توضيحات " مشخص شده ، ارث بردن B ازA ز غير مجاز شده چون A بعنوان final اعلان شده است.

كلاس Object

كلاس شي The object class يك كلاس مخصوص يعني object توسط جاوا تعريف شده است. كليه كلاسهاي ديگر زير كلاسهاي object هستند. يعني object كلاس بالاي كليه كلاسهاي ديگر است. بدين ترتيب يك متغير ارجاع نوع object مي تواند به يك شي از هر كلاس ديگري ارجاع نمايد. همچنين ، نظريه اينكه آرايه ها بعنوان كلاس پياده سازي مي شوند ، يك متغير از نوع objectمي تواند به هر آرايه اي ارجاع نمايد. Object روشهاي بعدي را تعريف مي كند. كه بدين ترتيب همه آنها در هر شيئي قابل دسترسي هستند .

عملكرد	متد
شي جديدي ايجاد مي كند كه تكثير شده همان شي است .	**object clon()**
تعيين مي كند كه آيا شي با شي ديگر مساوي است يا نه.	**boolean equals (Object object)**
قبل از اينكه يك شي بدون استفاده در چرخه مجدد قرار گيرد ، فراخواني ميشود .	**void finalize()**
كلاس يك شي را حين اجرا نگهداري ميكند.	**class get class()**
كد hash همراه با اشيائ فراخوان را برگردان مي كند .	**int hash code()**
اجراي يك رشته در حال انتظار روي شي فراخوان كننده را از سر مي گيرد .	**void notify()**
اجراي كليه رشته هاي در حال انتظار روي شي ئ فراخوان كننده را از سر مي گيرد .	**void notify All()**
رشته اي كه توصيف كننده شي ئ است را برمي گرداند .	**string to string()**

void wait() **روي يك نخ (thread) ديگر از اجرا منتظر 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. ممكن است ساير موارد را لغو نماييد . فعلا به دو روش دقت نماييد :

equals() و tostring() . متد equals() محتوي دو شي را با يكديگر مقايسه نموده و اگر دو شي معادل هم باشند ، اين روش true را برمي گرداند و در غير اينصورت false را برمي گرداند . روش to string() رشته اي كه دربرگيرنده توصيفي از شي كه روي آن فراخواني شده ، را برمي گرداند . اين روش همچنين هرگاه يك شي ئ حاصل استفاده از println() باشد بطور خودكار فراخواني خواهد شد . بسياري از كلاسها اين روش را لغو مي كنند. انجام اين كار به آن كلاسها امكان مي دهد تا شرحي را مشخصا براي نوع شي هايي كه ايجاد مي كنند ، آماده سازند.

بسته ها و رابطها

عناوين اين بخش :

مدل زبان و كاربردهاى آن

بسته ها و رابطها

بسته ها packages

تعريف يك بسته

درك مفهوم CLASS PATH

محافظت دسترسي Access protection

وارد كردن بسته ها

رابطها interfaces

تعريف نمودن يك رابط

پياده سازي رابطها

دسترسي به پياده سازيها از طريق ارجاعات رابط

پياده سازي نسبي(partial)

بكار بردن رابطها

متغيرها در رابطها

بسته ها و رابطها

بسته ها (packages) ، ظروفي براي كلاسها هستند كه براي نگهداري فضاهاي تقسيم بندي شده اسامي كلاسها استفاده مي شوند . بعنوان مثال ، يك بسته به شما اجازه مي دهد تا يك كلاس با نام list ايجاد نموده كه آن را در بسته خود ذخيره نماييد بدون نگراني از اينكه اين كلاس با يك كلاس ديگر با نام list و ذخيره شده در جاي ديگري تلاقي داشته باشد. بسته ها بصورت سلسله مراتبي ذخيره شده و بطور صريح به تعريف كلاس جديد وارد مي شوند .

از طريق استفاده از واژه كليدي interface ، جاوا به شما اجازه مي دهد تا رابط را از پياده سازي مربوط به آن كاملا مجرد نماييد. با استفاده از interface مي توانيد يك مجموعه از روشهايي كه قادرند براي يك يا چندين كلاس پياده سازي شوند را مشخص نماييد .خود interface بواقع هيچ نوع پياده سازي را تعريف نميكند . اگر رابطها مشابه با كلاسهاي مجرد abstract هستند ، اما يك ظرفيت بيشتر دارند : يك كلاس مي تواند بيش از يك رابط را پياده سازي نمايد. اما، يك كلاس فقط ميتواند از يك كلاس بالاي تكي (مجرد يا غيره) ارث ببرد.

بسته ها (packages) و رابطها (interfaces) دو عنصر اصلي در يك برنامه جاوا مي باشند . بطور كلي ، يك فايل منبع جاوا مي تواند دربرگيرنده يكي از (يا كليه) بخشهاي داخلي چهارگانه زير باشد:

يك دستور بسته اي تكي (اختياري) A single package statement

به هر تعداد دستورات وارده (اختياري) Any number of import statements

يك اعلان كلاس تكي بعنوان public (اجباري) A single public class declaration

به هر تعداد كلاسهاي اختصاصي براي بسته(اختياري)

Any number of classes private to the package

تا بحال در مثالهاي مشاهده شده فقط يكي از اين موارد يعني اعلان كلاس تكيه است.

بسته ها packages

قبلا نام هر مثالي از كلاس را از يك فضاي نام مخصوص (name space) مي گرفتيم . يعني براي هر كلاس بايد يك نام منحصر بفرد استفاده مي شد تا از اختلاط نامها جلوگيري 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 اعضائ 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 يكديگر داشته باشند اما آن اطلاعات را براي ديگران افشاگري نكنند .

تعريف يك بسته

ايجاد يك بسته بسيار آسان است :

خيلي ساده يك فرمان package بعنوان اولين دستور در فايل منبع جاوا بگنجانيد. هركلاس 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 كوتاه و نمونه مناسب است ، اما براي برنامه هاي كاربردي 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 نامMypackage ايجاد مي كند :

```
package MyPackage;
```

جاوا از دايركتوريهاي فايل سيستم براي ذخيره سازي بسته ها استفاده مي كند . بعنوان مثال فايلهاي .class براي كلاسهايي كه بعنوان بخشي از Mypackage اعلان مي كنيد ، بايد در يك دايركتوري تحت نام Mypackage ذخيره شوند. بياد آوريد كه جاوا بسيار حساس است ، بنابراين اسم دايركتوري بايد دقيقا با اسم بسته مطابقت داشته باشد .

اين امكان وجود دارد كه بيش از يك فايل شامل يك دستور يكسان package باشند .دستور package خيلي ساده مشخص مي كند كه كلاسهاي تعريف شده در يك فايل به كدام بسته تعلق دارند. اين دستور ساير كلاسهاي موجود در فايلهاي ديگر را از اينكه بخشي از همان بسته اول باشند ، منع نمي كند . بسياري از بسته هاي دنياي واقعي روي تعداد زيادي از فايلها گسترده مي شوند .

مي توانيد يك سلسله مراتب از بسته ها بسازيد . براي انجام اينكار ، خيلي ساده اسم هر بسته را از اسم بالايي اش و با استفاده از يك نقطه جدا سازيد . شكل عمومي يك دستور package چند سطحي بقرار زير است :

```
package pkg1[.pkg2[.pkg3]];
```

سلسله مراتب بسته بايد در فايل سيستم توسعه جاوا منعكس شود . بعنوان مثال يك بسته اعلان شده بعنوان

```
package java.awt.image;
```

بايد در java/awt/image ، java\awt\image يا java:awt:image بترتيب روي فايل سيستم unix ، windows ، يا Macintosh ذخيره شود. اسم بسته خود را خيلي با دقت انتخاب نماييد. نمي توانيد اسم يك بسته را تغيير دهيد مگر اينكه اسم دايركتوري كه كلاسها در آن ذخيره شده اند را تغيير دهيد .

درك مفهوم CLASS PATH

قبل از اينكه مثالي براي استفاده از يك بسته معرفي كنيم ، لازم است بحث مختصري درباره متغير محيطي CLASS PATH داشته باشيم . اگرچه بسته ها بسياري از مشكلات از نقطه نظر كنترل

دسترسي و اختلاط فضاي نام را حل مي كنند ، اما هنگام كامپايل و اجرا نمودن آنها با مشكلات عجيب و غريبي مواجه مي شويد . زيرا مكان مشخص در نظر گرفته شده بعنوان ريشه سلسله مراتب بسته توسط كامپايلر 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 (.) 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 راحتي نخواهد بود .

فرض كنيد كه يك كلاس با نام packTest در يك بسته با نام test ايجاد كرده 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.javaرا 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 شبيه زير را دربرمي گيرد :

```
. ;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

كه دايركتوري test در دايركتوري در حال كار جاري وجود ندارد زيرا در خود دايركتوري test قرار داريد. در اين نقطه دو انتخاب خواهيد داشت :

دايركتوريهاي يك سطح بالاتر را تغيير داده و javatest.packTest را آزمايش كنيد و يا اينكه بالاي سلسله مراتب توسعه كلاس را به متغير محيطي CLASS PATH اضافه نماييد . آنگاه مي توانيد packTest javatest را از هر دايركتوري مورد استفاده قرار دهيد و جاوا فايل درست class. را پيدا خواهد نمود. بعنوان مثال ، اگر روي كد منبع خودتان در يك دايركتوري تحت نام C:mmyjava در حال كار باشيد ، آنگاه CLASS PATH خود را روي ;C:\myjava; . C:\java\classes قرار دهيد.

يك مثال كوتاه از بسته

اگر بحث قبلي را در نظر داشته باشيد ، مي توانيد اين بسته ساده را آزمايش كنيد :

```
// 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)(;
}
}
```

اين فايل را Mypack/java ناميده و آن را در يك دايركتوري تحت نام Mypack قرار دهيد .
سپس فايل را كامپايل كنيد. مطمئن شويد كه فايل حاصل.class نيز در دايركتوري Mypack قرار گرفته باشد. سپس اجراي كلاس Account Balance را آزمايش كنيد، البته با استفاده از خط فرمان بعدي

```
java MyPack.AccountBalance
```

بياد داشته باشيد كه وقتي مي خواهيد اين فرمان را اجرا كنيد، لازم است تا در دايركتوري بالاي Mypack باشيد و يا اينكه متغير محيطي CLASS PATH خود را بطور مناسبي قرار داده باشيد.
همانطوريكه توضيح داديم ، Account Balance اكنون بخشي از بسته Mypack است . اين بدان معني است كه نمي تواند بوسيله خودش اجرا شود. يعني نمي توانيد از اين خط فرمان استفاده نماييد.

```
Java AccountBalance
```

محافظت دسترسي Access protection

قبلا مي دانستيد كه دسترسي به يك عضو private در يك كلاس فقط به ساير اعضاء همان كلاس واگذار شده است . بسته ها بعد ديگري به كنترل دسترسي مي افزايند . همانطوريكه خواهيد ديد ، جاوا سطوح چندي از محافظت براي اجازه كنترل خوب طبقه بندي شده روي رويت پذيري متغيرها و روشهاي داخل كلاسها ، زير كلاسها و بسته ها فراهم مي نمايد.
كلاسها و بسته ها هر دو وسايلي براي كپسول سازي بوده و دربرگيرنده فضاي نام و قلمرو متغيرها و روشها مي باشند. بسته ها بعنوان ظروفي براي كلاسها و ساير بسته هاي تابعه هستند . كلاسها بعنوان ظروفي براي داده ها و كدها مي باشند. كلاس كوچكترين واحد مجرد در جاوا

است. بلحاظ نقش متقابل بين كلاسها و بسته ها ، جاوا چهار طبقه بندي براي رويت پذيري اعضاي كلاس مشخص كرده است:

زير كلاسها در همان بسته

غير زير كلاسها در همان بسته

زير كلاسها در بسته هاي مختلف

كلاسهايي كه نه در همان بسته و نه در زير كلاسها هستند.

سه مشخصگر دسترسي يعنيprivate ، public و protected و فراهم كننده طيف گوناگوني از شيوه هاي توليد سطوح چند گانه دسترسي مورد نياز اين طبقه بنديها هستند. جدول زير اين ارتباطات را يكجا نشان داده است.

public	protected	Nomodifier	private	
Yes	Yes	Yes	Yes	همان كلاس
Yes	Yes	Yes	No	زير كلاس همان بسته
Yes	Yes	Yes	No	غير زير كلاس همان بسته
Yes	Yes	No	No	زير كلاس بسته هاي مختلف
Yes	No	No	No	غير زير كلاس بسته هاي مختلف

اگرچه مكانيسم كنترل دسترسي در جاوا ممكن است بنظر پيچيده باشد، اما ميتوان آن را بصورت بعدي ساده گويي نمود . هر چيزي كه بعنوان public اعلان شود از هر جايي قابل دسترسي است. هر چيزي كه بعنوان private اعلان شود خارج از كلاس خودش قابل رويت نيست. وقتي يك عضو فاقد مشخصات دسترسي صريح و روشن باشد ، آن عضو براي زير كلاسها و ساير كلاسهاي موجود در همان بسته قابل رويت است . اين دسترسي پيش فرض است . اگر مي خواهيد يك عضو ، خارج از بسته جاري و فقط به كلاسهايي كه مستقيما از كلاس شما بصورت زير كلاس درآمده اند قابل رويت باشد ، پس آن عضو را بعنوان protected اعلان نماييد.

يك كلاس فقط دو سطح دسترسي ممكن دارد : پيش فرض و عمومي . (public) وقتي يك كلاس بعنوان public اعلان مي شود ، توسط هر كد ديگري قابل دسترسي است. اگر يك كلاس دسترسي پيش فرض داشته باشد ، فقط توسط ساير كدهاي داخل همان بسته قابل دسترسي خواهد بود.

يك مثال از دسترسي :

مثال بعدي كليه 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 مثالp1 وp2 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. متغيرn با حفاظت پيش فرض اعلان شده است. m-priبعنوان 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 بنابراين به كليه 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);
```

```
}
}
```

فايل 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);
}
}
```

فايل 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);
}
}
```

اكنون كد منبع يك بسته ديگر يعني p2 را مشاهده مي كنيد. دو كلاس تعريف شده در p2 دو شرايطي را كه توسط كنترل دسترسي تحت تاثير قرار گرفته اند را پوشش داده است . اولين كلاس يعني protection 2 يك زير كلاس p1.protection است . اين كلاس دسترسي به كليه متغير هاي مربوط به p1.protection را بدست مي آورد غير از n-pri (چون private است) و n_ متغيري كه با محافظت پيش فرض اعلان شده است . بياد داشته باشيد كه پيش فرض فقط اجازه دسترسي از داخل كلاس يا بسته را مي دهد نه از زير كلاس هاي بسته هاي اضافي . در نهايت ،

کلاس otherpackage فقط به يك متغير n-pub كه بعنوان public اعلان شده بود دسترسي خواهد داشت .

فايل 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);
}
}
```

فايل 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 مايليد تا اين دو بسته را آزمايش كنيد ، در اينجا دو فايل آزمايشي وجود دارد كه مي توانيد از آنها استفاده نماييد. يكي از اين فايلها براي بسته p1 را در زير نشان داده ايم :

```
// 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 Drived();
SamePackage ob3 = new SamePackage();
}
}
```

فايل آزمايشي براي p2 بقرار زير مي 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 كردن بسته ها

با قبول اين امر كه بسته ها وجود داشته و مكانيسم خوبي براي جدا كردن كلاسهاي متمايز از يكديگر هستند، آنگاه براحتي مي فهميد كه چرا كليه كلاسهاي توكار جاوا در بسته ها ذخيره مي شوند. در بسته پيش فرض بدون نام ، كلاسهاي هسته اي جاوا وجود ندارند . كليه كلاسهاي استاندارد در برخي بسته هاي داراي نام ذخيره مي شوند . چون كلاسهاي داخل بسته ها بايد كاملا با نام يا اسامي بسته هاي خود كيفيت دهي شوند ، ممكن است تايپ نام مسير بسته كه با نقطه از هم جدا مي شود براي هر يك كلاسي كه مايليد استفاده نماييد ، كاري كسل كننده باشد . بهمين دليل ، جاوا دربرگيرنده دستور import (وارد كردن) براي آوردن كلاسهاي مشخص يا كل بسته ها در معرض رويت مي باشد. هر كلاسي كه يكبار وارد شود ، فقط با استفاده از اسم آن بطور مستقيم قابل ارجاع است. دستور import يك وسيله راحت براي برنامه نويسان است و از نظر تكنيكي ديگر نيازي نيست تا يك برنامه كامل جاوا نوشته شود . اگر قصد داريد تا به چند ده كلاس در برنامه اتان ارجاع نماييد ، دستور import حجم زيادي از كار تايپي را كاهش مي 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 مواردي 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 عملكرد حين اجرا يا اندازه كلاسهاي شما نخواهد داشت . كليه 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 كليه برنامه ها وارد خواهد شد. اين حالت معادل آن است كه خط بعدي در بالاي كليه برنامه هاي شما قرار بگيرد :

```
import java.lang.*;
```

اگر يك كلاس با همان اسم در دو بسته متفاوتي كه با استفاده از ستاره وارد كرده ايد ، قرار داشته باشد ، كامپايلر ساكت (silent) مي ماند ، مگر اينكه سعي كنيد يكي از آن كلاسها را استفاده نماييد. درآن 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 اقلامي 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 كاربردهاي عمومي خارج از Mypack در دسترس باشد ، پس بايد آن را بعنوان public اعلن 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);
}
}
```

همانطوريكه مي بينيد ، كلاس Balance اكنون بعنوان public مي باشد. همچنين سازنده اين كلاس و روش show() در آن نيز بعنوان public مي باشند . اين بدان معني است كه همه آنها توسط هر نوع كدي خارج از بسته Mypack قابل دسترسي هستند . بعنوان مثال ، در اينجا Test Balance وارد كننده Mypack شده و سپس قادر است از كلاس Balance استفاده نمايد :

```
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 كلاس Balance برداشته و سپس سعي كنيد ا بروز مي كنند.

رابطها Interfaces

بااستفاده از واژه كليدي interface ، مي توانيد رابط يك كلاس را از پياده سازي آن كلاس بطور كامل مجرد نماييد. يعني با استفاده از interface مي توانيد مشخص نماييد يك كلاس چكاري بايد انجام دهد ، اما چگونگي آنرا مشخص نخواهيد كرد. رابطها از نظر قواعد صرف و نحو مشابه كلاسها هستند ، اما فاقد متغيرهاي نمونه هستند و روشهاي آنها بدون بدنه اعلان مي شود . در عمل ، اين بدان معني است كه مي توانيد رابطهايي تعريف كنيد كه درباره چگونگي پياده سازي خود فرضيه اي نمي سازند. هر بار كه رابط تعريف شود ، هر تعدادي از كلاسها مي توانند آن رابط interface را پياده سازي نمايند. همچنين ، يك كلاس مي تواند هر تعداد رابطها را پياده سازي نمايد.

براي پياده سازي يك رابط، كلاس بايد مجموعه كامل روشهاي تعريف شده توسط رابط را ايجاد نمايد. اما ، هر كلاسي آزاد است تا جزئيات پياده سازي خودش را تعيين نمايد. با استفاده از واژه

كليدي interface ، جاوا به شما امكان مي دهد تا حداكثر بهره از جنبه " يك رابط و چندين روش " در چند شكلي را بدست آوريد . رابطها بمنظور حمايت از سرگيري پوياي روش در حين اجرا طراحي مي شوند. بطور معمول ، براي اينكه يك روش از يك كلاس به كلاس ديگر فراخواني شود ، هر دو كلاس بايد در زمان كامپايل حضور داشته باشند، بطوريكه كامپايلر جاوا بتواند آنها را كنترل نموده و سازگاري روشها را تاييد نمايد. اين امر بتنهايي منجر به يك محلط كلاس دهي ايستا و غير قابل توسعه خواهد شد. در يك چنين سيستمي بناچار عملگرايي (functionality) در سلسله مراتب كلاس بالاتر و بالاتر مي رود بطوريكه مكانيسم فوق ، در دسترس زير كلاسهاي بيشتري قرار خواهد گرفت. رابطها طراحي ميشوند تا از بروز چنين مشكلي جلوگيري بعمل آورند. آنها قطع ارتباط بين تعريف يك روش يا مجموعه اي از روشها و سلسله مراتب وراثت را بوجود مي آورند. از آنجاييكه رابطها در سلسله مراتب متفاوتي از كلاسها قرار گرفته اند، براي كلاسهايي كه بر حسب سلسله مراتب كلاس نامرتبط هستند ، امكان پياده سازي همان رابط وجود دارد. اين نقطه اي است كه قدرت واقعي رابطها را نمايان مي سازد .

نكته :

رابطها ، آن ميزان عملگرايي لازم براي بسياري از كاربردها را بوجود مي آورند كه در غير اينصورت ناچار از متوسل شدن به وراثت چند گانه در زباني نظير C++ خواهيم شد.

تعريف نمودن يك رابط

يك رابط مشابه يك كلاس تعريف مي شود . شكل عمومي يك رابط بصورت زير مي 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 اينجا ، دسترسي (access) يا public است يا اصلا بكار نمي رود. وقتي كه مشخصگر دسترسي را نگنجانيم ، آنگاه دسترسي پيش فرض اعمال شده و رابط فقط براي ساير اعضا بسته اي كه در آن اعلان انجام گرفته ، در دسترس خواهد بود. وقتي كه بعنوان public اعلان مي شود ، رابط ، توسط هر كد ديگري قابل استفاده خواهد بود. name همان نام رابط است و بجاي آن هر شناسه معتبري را مي توان بكار برد . توجه كنيد كه روشهاي اعلان شده داراي بدنه نيستند. انتهاي آنها بعداز parameter-list يك (;) قرار ميگيرد و آنها ضرورتا روشهاي مجرد هستند: ممكن است هيچ پياده سازي پيش فرضي براي روشهاي مشخص شده داخل يك رابط وجود نداشته باشد . هر كلاسي كه دربرگيرنده يك رابط باشد ، بايد كليه روشها را پياده سازي نمايد . متغيرها را مي توان داخل اعلانات رابط ، اعلان نمود. آنها بطور ضمني final وstatic هستند. بدين معني كه آنها توسط كلاس پياده سازي قابل تغيير نيستند . آنها همچنين بايد با يك مقدار ثابت ، مقدار دهي اوليه شوند . اگر رابط خودش بعنوان يك public اعلان شده باشد ، كليه روشها و متغيرها بطور ضمني public خواهند بود.

اكنون مثالي از يك تعريف رابط را مشاهده مي كنيد. اين مثال يك رابط ساده را تعريف مي كند كه دربرگيرنده روشي تحت نام callback() است كه يك پارامتر تكي عدد صحيح را مي گيرد .

```
interface Callback {
void callback(int param);
}
```

پياده سازي رابطها

هر بار رابطي را تعريف نماييد ، يك يا چندين كلاس مي توانند آن رابط را پياده سازي نمايند . براي پياده سازي يك رابط ، جمله implements را در تعريف كلاس بگنجانيد و سپس روشهاي تعريف شده توسط رابط را ايجاد نماييد. شكل عمومي يك كلاس كه دربرگيرنده جمله implementsاست ، مشابه مورد زير مي باشد :

```
access class classname [extends superclass]
[implements interface [/interface...]] {
```

```
// class-body
}
```

در اينجا (access) يا public است يا اصلا" استفاده نمي شود. اگر يك كلاس بيش از يك رابط را پياده سازي نمايد ، رابطها را با يك علامت كاما از يكديگر جدا مي كنيم . اگر يك كلاس دو رابط را كه روش يكساني را اعلان مي كنند ، پياده سازي نمايد ، آنگاه آن روش يكسان توسط سرويس گيرندگان هر يك از رابطها استفاده خواهد شد. روشهايي كه يك رابط را پياده سازي مي كنند بايد بعنوان public اعلان شوند .همچنين تاييد نوع روش پياده سازي بايد دقيقا با تاييد نوع مشخص شده در تعريف interface مطابقت و سازگاري داشته باشد . در اينجا يك مثال از كلاس ساده اي مشاهده مي كنيد كه رابط callback را پياده سازي مي نمايد :

```
class Client implements Callback {
// Implement Callback's interface
public void callback(int p ){
System.out.println("callback called with " + p);
}
}
```

دقت داشته باشيد كه callback() با استفاده از مشخصگر دسترسي public اعلان شده است.
يادآوري : وقتي كه يك روش رابط را پياده سازي مي كنيد ، بعنوان public اعلان خواهد شد .
براي كلاسهايي كه رابطها را پياده سازي مي كنند ، هم مجاز و هم رايج است كه اعضا اضافي براي خودشان تعريف نمايند . بعنوان مثال ، روايت بعدي از client پياده سازي callback را انجام داده و روش nonIfaceMeth() را اضافه مي نمايد :

```
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 ميكند، 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 ميكند :

```
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 اعضائ كلاس client دسترسي داشته باشد . يك متغير ارجاع رابط فقط از روشهاي اعلان شده توسط اعلان interface خود ، آگاهي دارد . بدين 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 ئ دسترسي داشته باشد ، اما قدرت چند شكلي يك چنين ارجاعي را نمايش نمي دهد . براي مشاهده اين كاربرد ، 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));
}
}
```

اكنون كلاس بعدي را امتحان كنيد :

```
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 اين برنامه بقرار زير مي باشد :

```
callback called with 42
Another version of callback
p squared is 1764
```

همانطوريكه مي بينيد ، روايتي از callback() كه فراخواني شده توسط نوع شيئي كه C به آن در حين اجرا ارجاع مي كند ، تعيين مي شود . اگرچه اين مثال بسيار ساده است ، يك برنامه عملي تر و كوتاهتر را مشاهده خواهيد نمود.

پياده سازي نسبي (partial)

اگر يك كلاس دربرگيرنده يك رابط باشد ، اما روش تعريف شده توسط آن رابط را كاملا پياده سازي نكند ، آنگاه آن كلاس بايد بعنوان abstract اعلان شود . بعنوان مثال :

```
abstract class Incomplete implements Callback {
int a, b;
void show (){
System.out.println(a + " " + b);
}
//...
}
```

در اينجا كلاس Incomplete روش callback() را پياده سازي نميكند و بايد بعنوانabstract اعلان شود. هر كلاسي كه از Incomplete ارث ميبرد بايد يا روش callback() را پياده سازي نمايد و يا خودش بعنوان abstract اعلان شود .

بكار بردن رابطها

براي درك قدرت رابطها ، اجازه دهيد تا به يك مثال عملي تر بپردازيم . قبلا يك كلاس موسوم به stack را توسعه داديم كه يك پشته ساده و با اندازه ثابت را پياده سازي مي كرد . اما ، راههاي متعددي براي پياده سازي يك پشته وجود دارد . بعنوان مثال ، يك پشته ممكن است داراي اندازه ثابت و يا قابل گسترش باشد .
همچنين مي توان پشته را در يك آرايه ، يك فهرست پيوندي (kinded list) ، يك درخت دودوئي و امثالهم نگهداري نمود . مهم نيست كه پشته چگونه پياده سازي مي شود رابط به پشته يكسان مي ماند. يعني روشهاي push() و pop() معرف رابط به پشته هستند و اين رابطها

مستقل از جزئيات پياده سازي مي باشند. چون رابط به يك پشته از پياده سازي آن جدا مي باشد ، تعريف رابط يك پشته خيلي ساده است ، و هر پياده سازي مشخصات خاص خود را تعريف خواهد كرد . به دو مثال نگاه كنيد . اول اينكه در اينجا رابطي وجود دارد كه يك پشته عدد صحيح را تعريف مي كند . آن را در يك فايل تحت نام IntStack.java قرار دهيد . اين رابط توسط هر دو نوع پياده سازي پشته استفاده خواهد شد .

```
// Define an integer stack interface.
interface IntStack {
void push(int item); // store an item
int pop)(; // retrieve an item
```

برنامه بعدي يك كلاس تحت نام FixedStack ايجاد مي كند:

```
// 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 زير 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 بيشتري 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) ();
}
}
```

كلاس بعدي هم از پياده سازي FixedStack و هم از DynStack استفاده مي كند . اين كلاس اينكار را طريق تعريف يك ارجاع رابط انجام مي دهد. اين بدان معني است كه فراخواني هاي push() و pop() در حين اجرا (بجاي زمان كامپايل) 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);
mystak = ds;
System.out.println("Values in dynamic stack:");
for(int i=0; i<12; i++)
System.out.println(mystack.pop) ();
mystak = fs;
System.out.println("Values in fixed stack:");
for(int i=0; i<8; i++)
System.out.println(mystack.pop) ();
}
}
```

در اين برنامه ، mystackيك ارجاع است به رابط .IntStack بدين ترتيب ، هرگاه آن به ds ارجاع ميكند، از روايتهاي ()push و ()pop تعريف شده بوسيله پياده سازي DynStack استفاده مي كند . وقتي كه به fs ارجاع مي كند ، از روايتهاي ()push و ()pop كه بوسيله FixedStack تعريف شده ، استفاده مي كند. همانطوريكه توضيح داديم ، تمام اسن تعيين كنندگيها در حين اجرا انجام مي گيرند. دسترسي به پياده سازيهاي چندگانه از يك رابط از طريق يك متغير ارجاع رابط يكي از شيوه هاي كاملا قدرتمند جاوا براي رسيدن به چند شكلي در حين اجرا مي باشد .

متغيرها در رابطها

مي توانيد از رابطها براي وارد كردن ثابتهاي باشتراك گذاشته شده به كلاسهاي چند گانه بسادگي

از اعلان يك رابط كه دربرگيرنده متغيرهايي باشد كه با مقادير دلخواه مقداردهي اوليه شده باشند، استفاده نماييد . وقتي كه آن رابط را در يك كلاس مي گنجانيد (يعني وقتي كه رابط را پياده سازي مي كنيد) كليه اسامي آن متغيرها بعنوان ثابت ها در قلمرو خواهند بود . اين كار مشابه استفاده از فايل feader درC++/Cر براي ايجاد يك رقم بزرگ از ثابتهاي #defined و با اعلانات const مي باشد. اگر يك رابط دربرگيرنده هيچ روشي نباشد، آنگاه هر كلاسي كه دربرگيرنده آن رابط باشد در واقع چيزي را پياده سازي نمي كند . مثل اين است كه آن كلاس متغيرهاي ثابت را به فضاي اسم كلاس بعنوان متغيرهاي final وارد مي كرده است .

مثال بعدي از اين تكنيك براي پياده سازي يك تصميم گيرنده خودكار (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 باشيد كه اين برنامه از يكي ازكلاسهاي استاندارد جاوا يعني Random استفاده ميكند. اين كلاس فراهم كننده اعداد شبه تصادفي است . اين كلاس دربرگيرنده چندين روش است كه به شما امكان نگهداري ارقام تصادفي در شكل مورد نياز برنامه آن را مي دهد . در اين مثال ، از

روش nextDouble() استفاده شده است . اين روش اعداد تصادفي در محدوده ۰,۰تا ۱,۰ را برمي گرداند.

در اين برنامه نمونه دو كلاسQuestion و AskMe و هر دو رابط SharedConstants را پياده سازي مي كنند ، جايي كهNo ، Yes، MAYBE، SOON، LATER و NEVER و تعريف شده اند . داخل هر يك كلاس ، كد به اين ثابت ها مراجعه مي كند بطوريكه گويا هر كلاس آنها را بطور مستقيم تعريف نموده يا مستقيما از آنها ارث برده است . در اينجا حاصل يك اجراي نمونه از اين برنامه را مشاهده مي كنيد . دقت كنيد كه نتايج در هر بار اجرا متفاوت خواهد بود .

```
Later
Soon
No
Yes
```

رابطها را مي توان گسترش داد

يك رابط با استفاده از واژه كليدي extends مي تواند از يك رابط ديگر ارث ببرد. دستور زبان مشابه كلاسهاي ارث برنده است. وقتي يك كلاس رابطي را پياده سازي مي كند كه از رابط ديگري ارث برده است ، بايد پياده سازيهاي كليه روشهاي تعريف شده داخل زنجيره وراثت را فراهم نمايد . مثالي را مشاهده مي كنيد :

```
// 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 قبلا گفتيم ، هر كلاسي كه يك رابط را پياده سازي مي كند بايد كليه 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 سازي انواع ساختمان داده ها

عناوین این بخش :

رشته ها

آرایه ها

فایل ها

در این فصل با نحوه پیاده سازي 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 شئ 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.شئ 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 شیئ 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 رشتهای در جاوا همچون مدیریت رشته های معمولی در زبان های کامپیوتری دیگر است.

دومین مطلبی که باید درباره رشته ها بدانید ان است که شیئ های نوع string تغییرناپذیر هستند.یعنی پس از ایجاد شیئ 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 شیئ ُفقهدل 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 شیئ 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 توانیدکاراکتر 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 شیئ های توکار این امکان را برای جاوا فراهم ساخته است تا ویژگی های زیادی در اختیارتان بگذارد که مدیریت رشته ها را اسان می سازند .به عنوان مثال جاوا متد هایی برای مقایسه دو رشته ,جستجوی یک زیر رشته,ادغام دو رشته و تغییر بزرگی وکوچکی حروف هر رشته دارد.همچنین شیئ های String را با چند روش می توان ایجاد نمود.بنابر این ایجاد رشته ها به هنگام نیاز آسان می شود.وقتی یک شیئ نوع String ایجاد میکنیم,رشته ای ایجاد می شود که قابل تغییر نخواهد بود.یعنی پس ازایجادشیءهای string کاراکترهای تشکیل دهنده آنها رانمی توانید تغییردهید. 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 ان است که هر بار نیاز به نگارش تغییر یافته ای از هر رشته مطرح باشد شیئ 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 تغییرناپذیر 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 درjava.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 واین 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 شیئ های نوع String غیر قابل تغییر هستند,بدین معناست که محتوای نمونه های String را نمیتوان پس از ایجاد تغییر داد. اما متغیری که به صورت نشانی Stringها تعریف می شود در هر لحظه می تواند به شیئ String دیگری ارجاع داشته باشد.

Constructor کلاس String

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 آغازکارکترهای 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 میتوانید شئ String ای با مقدار یک شئ String دیگر ایجاد کنید.

```
String(String strObj)
```

strObj شئ 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 s2, s1 وجود دارد.

اکر 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 ازیک آرایه نوع byte فراهم ساخته است.فرم آنها در زیر نشان داده شده است:

```
String(byte askiichars[])
String(byte askiichars[],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 شئ 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 میتوانید 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 Stringای 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 زیر۳ است چرا که سه کاراکتر در رشتهs 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 ومهمی از برنامه سازی با جاوا به شمار می آیند جاوا امکانات ویزه 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 شئ String با استفاده از عملگر"+" وتبدیل انواع داده های دیگربه 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 عملگرnew 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 شئ 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 شئ String برای هر یک از لیترال های رشته ای ایجاد می شود هر جا که شئ 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 مستقیما برای رشته هایی که بین علائم نقل قول نوشته می شوند فرا بخوانید درست همچون فرا خوانی آن ها با متغییرهای ارجاع به شئ 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 شئ های String را فراهم نمی سازد.تنها استثنا این موضوع عملگر "+" است که دو رشته را ادغام می کند و نتیجه آن یک شئ 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 واز سطر بعد ادامه یابند می توانید آن ها را به چند قسمت تقسیم وبا "+" ادغام کنید.به مثال های زیر توجه کنید:

```
//using concatentionto prevent long lines.
Class concat{
Public static void main(String args[]){
String longstr=”this coud 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 نسبتا متفاوتی از مثال پیش توجه کنید:

```
Int age = 9;
String s = “he is”+age+”years old.”;
System.out.println(s);
```

Age در این مثال متغییر 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 مستقیما 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)
```

targetStartمشخص کننده ایندکس ابتدای زیر رشته,وsourceEndهم مشخص کننده ایندکس پس ازآخرین کاراکتر زیر رشته مورد نظر است.از این رو,زیر رشته مورد نظرمتضمن کاراکترهای targetStartتاsourceEnd-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( )equalsاستفاده کنید.فرم کلی آن به صورت زیر است:

Bolean equals(Object str)

Str شئ string ای است که با شئ String فعال کننده متد مقایسه میشود.چنانچه کاراکتر های موجود در رشته ها برابر باشند و ترتیب شان نیز یکسان باشد حاصل فراخوانی trueخواهد 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 وکوچک 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:

Bolean equalsIgnoreCase(String str)

Str شئ Stringای است که با شئ string فعال کننده متد مقایسه می شود.حاصل این متد نیز در صورت یکسان بودن کاراکتر ها و ترتیبشان true ودر غیر این صورت falseخواهد بود.

Equals() در مقابل= =

لازم است بدانید که متد equals() وعملگر "= =" دو عمل متفاوت انجام می دهند .همان گونه که پیش از این شرح داده شد متدequals() کاراکترهای موجود در شئ String مورد نظر را مقایسه می کند.عملگر "= =" دو نشانی را مقایسه میکند تا مشخص شود که هر دو به نمونه یکسانی از یک شئ ارجاع دارند یا خیر.

compareTo()

اغلب اوقات دانستن اینکه دورشته مشابه هستتند یا خیر.کفایت نمی کند. برای برنامه های مرتب سازی باید بتوان تشخیص داد که کدام کاراکتر کوچکتر مساوی یا بزرگتر ازمورد بعدی است.

رشته ای کوچکتر ازیک رشته د یگرتلقی میشودکه ازنظرترتیب پیش ازآن جایی داشته باشد.

ورشته ای بزرگترازیک رشته دیگر تلقی میشودکه ازنظرترتیب پس ازآن جای بگیرد.متد compareTo()کلاس stringاین کارراانجام میدهد.فرم کلی آن درذیل نشان داده شده است:

Int compraeto(string str)

Str همان string ای است که با string فعال کننده ی متد مقایسه می شود.نتیجه مقایسه برگردانده شده وبه صورت زیرتفسیرمی شود:

کوچکترازصفر: رشته فعال کننده متد کوچکترازstr است.

بزرگترازصفر: رشته فعال کننده متد بزرگترازstr است.

صفر: دورشته برابرهستند.

Compareto() بزرگی و کوچکی حروف رامقایسه درنظرمی گیرد.کلمه ای که به آن اضافه شده است با یک حرف بزرگ شروع شده است.

اگر می خواهیدبزرگی وکوچکی 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 وکوچک 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 موفقیتبرگردانده 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 اخر 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
```

تغییررشته ها

چون شئ های string تغییر ناپذیرهستند هرگاه بخواهید یکstring را تغییر دهید میبایست آنرا بهStringBuffer یا StringBuilder کپی کنید ویا از یکی ازمتدهای زیر استفاده کنید

Substring

با استفاده از این متد می توانید یک زیررشته را استخراج نمایید.این متد دو فرم دارد.فرم نخست:

```
String substring(int startIndex)
```

startIndex ایندکس محل شروع زیررشته را مشخص میکند.حاصل این فرم ازمتدزیررشته ای است که از محل startIndex آغازوتاانتهای رشته ادامه مییابد.

فرم دوم نیز امکان مشخص کردن ایندکس ابتداوانتهای زیررشته را فراهم میسازد.

```
Strng 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 زیررشته با یک زیررشته دیگراستفاده 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 شئ جدیدی می شودکه حاوی رشته فعال کننده متد است که محتوای str به آن اضافه می شود.concat() عمل " + " راانجام می دهد.به عنوان مثال عبارت زیر سبب ذخیره شدن "onetwo" در s2 می شود.

```
String s1 = "one"
String s2 = s1.cotcat("two");
```

Replace()

این متد دو فرم دارد

فرم نخست : تمام نمونه های یک کاراکتر در رشته فعال کننده متد با کاراکتری دیگرجایگزین می شوند.

```
String replace(char original,char replacement)
```

Original مشخص کننده کاراکتری است که باید بوسیله کاراکتری جایگزین شود که با replacement مشخص می شودرشته حاصل برگردانده می شود.به عنوان مثال عبارت زیر سبب ذخیره"Heww" در s میشود.

```
String a = "Hello".replace('1','w');
```

در فرم دیگر این متد یک سری کاراکتر با یک سری دیگر جایگزین می شود.

```
String replace(Charsequense original,Charsequese replacement)
```

Trim()

این متد نسخه ای از رشته فعال کننده خودرا برمی گرداند که تمام فاصله های ابتدایی وانتهایی ازآن 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 متدداده هارا از فرمت داخلی خودشان به فرم خوانا تبدیل می کند.این متد یک متد ایستا ست که در کلاس string برای انواع داده های مختلف جاوا overload شده است تا تمامشان بدرستی به یک رشته قابل تبدیل با شند.valueof برای نوع object نیزoverload شده است تا شئ های هر کلاسی به عنوان آرگومان این متد قابل استفاده باشند

چند فرم این متددرزیرنشان 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 دادهای دیگر فراخوانده می شود-به عنوان مثال در طی عملیات ادغام داده هاست.این متد مستقیما با هر نوع داده ای فرا خوانده می شودو حاصل آن نیز نمایش string گونه آرگومانش است.تمام انواع داده های پایه به نمایش رشته ای معادل خودشان تبدیل می شوند.هر شئ که به valueof ارسال میشود نتیجه فراخوانی متد tostring شئ را برمی گرداند

Valueof() برای بیشتر آرایه ها رشته نسبتا مرموزی را برمی گرداندکه نشان می دهد نوعی آرایه است.اما در مورد آرایه های نوع char یک شئ string ایجاد میکند که شامل کاراکترهای موجوددرآرایه char است.

فرم کلی این متد:

```
static String valueOf (char chars[],int startIndex,int  numchars)
```

chars آرایه ایست که کاراکترها درآن قراردارندstartIndex مشخص کننده ایندکس آرایه کاراکتر هایی است که زیررشته مورد نظر از آنجا آغاز می شودو numchars نیز طول رشته را مشخص می کند.

تغییر کوچکی وبزرگی 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 دومتداستفاده 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 J2SE 5 اضافه شده است.

intcodePointBefore (int i) ، کد پونیت یونی کد موقعیت پیش از i را برمی گرداند.بوسیله J2SE 5 اضافه شده است.

Int codepointCount(int start,int end) ، تعداد کدپونیت های موجوددربین start و end-1 رادر String فعال کننده متدبرمی گرداند بوسیله J2SE 5 اضافه شده است.

Boolean contains(CharSequense ,str) ، چنانچه رشته str درشئ فعال کننده متد وجود داشته باشد حاصل آن true ودرغیراین صورت false خواهدبود.بوسیله J2SE 5 اضافه شده است.

Boolean contentEquals(Charsequense str) ، چنانچه رشته فعال کننده متد با رشته str یکسان باشد حاصل آن true ودرغیراین صورت falseخواهدبود. بوسیله J2SE 5 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 طورخودکاررشدمی کند تا فضای کافی برای این گونه اضافات فراهم شود و اغلب کاراکترهایی بیش ازآنچه لازم باشد پیشاپیش به آن اختصاص یابد تا فضا برای رشد وجود داشته باشد. جاوا ازهر دو کلاس زیاد استفاده می کند اما بسیاری از برنامه نویسان تنها با string سروکار دارند وبا استفاده ها ازعملگر"+" کار با StringBuffer درپشت پرده را به جاوا واگذار میکنند.

سازنده های StringBuffer

```
StringBuffer()
StringBuffer(int size)
StringBuffer(String str)
StringBuffer(CharSequence char)
```

سازنده پیش فرضکه فاقد پارامتر است فضابرای۱۶کراکتر را بدون تخصیص مجدد رزرو میکند. دومی یک آرگومان ازنوع صحیح داردکه اندازه بافررا مشخص میکند. سومی آرگومانی از نوع string دارد که مقدار اولیه شیئ stringbuffer را مشخص و فضا برای ۱۶کاراکتر را بدون تخصیص مجدد رزرو میکند. وقتی طول مشخصی برای بافردرخواست نشود stringbuffer فضا برای۱۶کاراکتراضافی را پیش بینی می کند چرا که تخصیص مجدد از نقطه نظرزمانی فرایندی پرهزینه به شمارمی آید. همچنین تخصیص مجدد به طور مکررسبب چند تکه شدن حافظه می شود stringbuffer با پیش بینی فضا برای چند کاراکتراضافی تعداد تخصیص های مجدد احتمالی راکاهش می دهد. سازنده ی چهارم شیئ ایجاد می کند که حاوی رشته ی مشخص شده درchar خواهد بود.

Length(),capacity()

اندازه جاری هر stringBuffer را میتوان بامتد length() وظرفیت کامل آنرا ازطریق متد capacity() بدست آورد فرم کلی به صورت زیراست.

```
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 آن۵ است. اما ظرفیت آن۲۱ است چرا که فضا برای ۱۶کاراکتراضافی بطور خودکاربه آن اضافه میشود.

ensureCapacity()

اگر میخواهید فضا برای چند کاراکتر را پس از ایجاد stringbuffer پیشاپیش تخصیص دهید میتوانید ازensureCapacity() استفاده کنید. اگر از قبل بدانید که تعداد زیادی از رشته های کوچک رابه StringBuffer اضافه خواهید کرداین امر مفید واقع خواهد شد. فرم کلی این متد به صورت زیراست:

```
Void ensureCapacituy(int capacity)
```

Capacity,اندازه بافررامشخص میکند.

setLength()

برای آنکه اندازه بافر شئ های StringBuffer رامشخص کند ازsetLength استفاده نمایید. فرم کلی آن به صورت زیراست.

```
Void setLength(int len)
```

Len اندازه بافر را مشخص می کند. مقدارآن باید مثبت باشد.
وقتی اندازه بافررا افزایش میدهید کاراکترهای تهی به انتهای بافر موجود اضافه میشود. اگر setLength() را با مقداری کوچکترازمقدارجاری حاصل ازفراخوانی length() فرا بخوانید در آن صورت کارکترهای پس ازطول جدید ازبین خواهند رفت. برنامه ساده setCharAtDemo در قسمت زیر ازsetLengh() برای کوتاه کردن stringBuffer استفاده میکند.

charAt(),setCharAt()

مقدار هریک ارکاراکترهای یک stringBuffer رامی توان ازطریق متد charAt() بدست آورد. با استفاده از بدست آورد. با استفاده از setCharAt() نیز میتواند مقدارهرکاراکتر را در هر stringBuffer تعیین کنید. فرم کلی آن به صورت زیراست:

```
Char charAt(int where)
Void setCharAt(int where,char ch)
```

Where برای charAt() ایندکس کاراکتر مورد نظر را مشخص می کند و ch نیزمقدار جدید کاراکتر را مشخص میکند where در هر دو متد باید مثبت باشد و نباید محلی پس ازانتهای 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 getChrs() 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 رامشخص میکند که به اندازه یک واحد بیش از انتهای زیررشته موردنظراست. یعنی زیر رشته ازکاراکترهای 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 نظرفضای کافی داشته باشد.

Append()

متد append معادل رشته هرنوع داده دیگررابه انتهای شیئ 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 شئ StringBuffer جاری اضافه میشود. تمام نگارشهای append() خود بافر را برمی گردانند. این امر سبب می شود تا فراخوانیهای متد مزبوربه طورزنجیره 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() بیشتراوقات هنگام استفاده از عملگر"+" برای شئ های string فراخوانده می شود. جاوا تغییرات مربوط به نمونه های هر string را بطور خودکار به عملیات مشابه درنمونه های stringBuffer تغییر میدهد از این رو عمل ادغام سبب فعال شدن append() برای شئ StringBuffer میشود. کامپایلرپس ازانجام عمل ادغام متد tostring() را یک مرتبه دیگر فرامی خواند تا StringBuffer را به یک ثابت String تبدیل کند.

تمام این کارها ممکن است به شکل غیرمعقولی پیچیده به نظررسد. 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 شئ 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 استتا علاوه بر string ها ، object ها و Charsequense 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 فرامی خواند.

این رشته سپس درشئ 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 شئ 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 هرشئ StringBuffer رامعکوس کنید:

```
StringBuffer reverse()
```

این متد شئ معکوس شده ای را برمیگرداند که هنگام فراخوانی ارسال شده است. برنامه زیرکاربرد 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

بااستفاده ازاین دو متد میتوانید کاراکترهای StringBuffer راحذف کنید.

```
StringBuffer delete(int startIndex,int endIndex)
StringBuffer deleteCharAt(int loc)
```

متد delete() یکسری کاراکتررا ازشئ 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. شئ StreingBuffer حاصل برگردانده می شود.

متد deleteCharAt() کاراکتر موجود در موقعیت loc راحذف می کند. این متد نیز شئ StringBuffer حاصل رابرمی 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 کاراکترهارا در یک شئ StringBuffer جایگزین مجموعه دیگری بکنید.

```
StringBuffer replace(int startIndex,int endIndex,String str)
```

زیررشته ای که جایگزین می شودبه وسیله ایندکس های startIndex,endIndex مشخص می شود از این رو زیررشته ای که ازstartIndex آغاز شده وتا endIndex-1 ادامه دارد جایگزین می شود. رشته جایگزین هم درstr ارسال می شود. شئ 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 هرstringBuffer را بدست آورید. این متد در دو فرم قابل استفاده است:

```
String substring (int startIndex)
Sting substring ( int startIndex,int endIndex)
```

فرم نخست زیررشته ای را برمی گرداند که از این startIndex آغازشده و تا انتهای شئ StringBuffer ادامه می یابد. فرم دوم نیز زیر رشته ای را بر می گرداند که از startIndex آغازشده و تا endIndex-1 ادامه می یابد. این متدها دقیقا همچون موارد تعریف شده برای string کارمی کنند.

چند متد دیگر

stringbuffer علاوه بر متدهای بررسی شده متدهای دیگری نیز داردکه برخی از آنها به وسیله J2SE افزوده شده اند.

stringbuffer appendCodePoint(int ch) ، کد پونیت یونی کدرا به انتهای شئ اضافه می کند. نشانی شئ برگرداندهمی شود .به وسیله J2SE 5 اضافه شده است.

Int codePointAt(int i) ، کد پونیت یونی کدمحلi رابرمیگرداند. به وسیله J2SE 5 اضافه شده است.

Int codepointBeffor (int start ,int end) ، تعداد کد پونیتهای بین start و end-1 شئ 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 پیداشده ویا 1- رادرصورت عدم موفقیت بر می گرداند.

Int indexOf(String str,int startIndex)، StringBuffer را از startIndex-1 برای یافتن نخستین نمونه از str جستجو می کند.ایندکس کاراکتر پیداشده ویا 1- رادرصورت عدم موفقیت بر می گرداند.

Int lastIndexOf(String str) ، StringBuffer را برای یافتن آخرین نمونه از str جستجو می کند.ایندکس کاراکتر پیداشده ویا 1- رادرصورت عدم موفقیت بر می گرداند.

Int lastIndexOf(String str,int startIndex) ، StringBuffer 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 subsequense(int startIndex,int stopIndex) ، زیررشته ازرشته فراخوان رااز startIndex تا stopIndex برمی گرداند.این متدمورد نیازرابط Charsequence است که اینک بوسیله StringBuffer پیاده سازی شَده است.

Void trimTosize()

اندازه بافر کاراکترها را برای شئ فراخوان کاهش می دهد تا مقدار جاری درآن جای گیرد. بوسیله J2SE 5 اضافه شده است.

به غیراز 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() وlastIndexOf() 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:" + 1);
I = sb.lastindexof("one");
System.out.println("Last indrx:" + 1);
```

```
}
}
```

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 کارآیی بیشتر است. اما در مواردی که از multithreading استفاده می کنید می بایست به جای StringBuilder ازStringBuffer 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 شئ 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 سطرi ام را مشخص می کند (تعداد ستون های سطر های مختلف می تواند متفاوت باشد ، البته در این مثال یکسان و برابر۵ 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 عنا صر 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.

متد ها :

- 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 (مثلStringBuffer) 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 عناصرarray 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 ودرج 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)
```

متد ها

- `Void addElement(type element)`

شئ 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 پشه را برمی گرداندااما آن را حذف نمی کند.

- `E pop()`

عنصر بالای پشه را برمی گرداندو آن را حذف می کند.

- `E push(E element)`

Element را به پشته اضافه می کند.

- `Int search(Object element)`

Element رادر پشته جستجو میکند چنانچه پیدا شود افست 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

بیشتربرنامه های کاربردی واقعی جاوا برنامه های کنسولی مبتنی بر متن نیستند .بلکه در عوض برنامه های گرافیک گرایی 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 درکارهای واقعی به شمار نمی آیند. پشتیبانی جاوا از I/O کنسولی محدود بوده واستفاده 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 وشبکه 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 فیزکی مرتبط میشود.تمام استریم ها به یک شکل عمل می کنند حتی اگروسایل فیزیکی مرتبط با آنها متفاوت باشند.از این رو کلاس ها ومتدهای 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 دیگرمجزا 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 شمارمی آیندو 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 کارامد 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 جاوا1.1 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 رسانده شوند.

نکته دیگر: تمام عملیات I/O هنوز هم در پایین ترین سطح بایت گرا هستند.استریم های مبتنی بر کاراکتر صرفا روش مناسب و کارامدی 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 وحتی 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 اندکه به وسیله کلاس های دیگر پیاده سازی می شوند.ومهمترین 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 شئ ها.

ObjectOutputStream : استریم خروجی برای شئ 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 کنندچندین 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. ومهمترین 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 هایI/O استریم کاراکتری

BuffereReader : 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 کاراکترها 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 متغییر 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 شئ ای از نوع InputStream است.System.out,System.err نیز شئ هایی از نوع PrintSystem هستند.گرچه این استریم ها عموما 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 کاراکتربپوشانید.

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 ان ها نگه 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 شئ ای از نوع BufferReader بپوشانید.BufferReader از استریم ورودی بافر شده پشتیبانی می کند .متداول ترین Constructor مورد استفاده به شکل زیر است :

BufferReader(Reader inputReader)

inputReader استریم مرتبط با نمونه ای از BufferReader 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 شئ ای از نوع InputStreamReader که باSystem.in مرتبط باشد از constructor ذیل استفاده کنید:

InputStreamReader(InputStream inputStream)

چون Stream.in به شئ ای از نوع InputStream ارجاع دارد از آن میتوان برای استریم ورودی استفاده نمود. در سطر زیر یک BufferReader ایجاد می شود که به صفحه کلید متصل میشود :

BufferReader br= new BufferReader (new InputStreamReader (System.in));

پس از اجرای سطر بالا br یک استریم مبتنی بر کاراکتر خواهد بود که از طریق System.inبا کنسول مرتبط است.

خواندن کاراکتر ها

برای آن که کاراکتری را از یک BufferReader بخوانید از ()Read استفاده کنید.نگارشی از ()Read که به کار خواهیم بردبه صورت زیر است:

int Read()throws IOExeption

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 ioexeption را نیز می تواندپرتاب کند.

برنامه ذیل روش استفاده از ()read رابا 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 IOExeption
{
Char c;
BufferedReader br=new BufferReader(new InputstreamReader(System.in));
System.out.println("Enter charactors,'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 charactors,'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 رااز صفحه کلید بخوانیداز()readline استفاده کنید که عضوی از کلاس bufferedreader است. فرم کلی آن به صورت زیر است.

```
String readline() throws loexception
```

همان گونه که ملاحظه می کنید حا صل ان یک شیئ string است .

برنامه زیر عملکرد bufferedreader ومتد ()readline را نشان میدهد این برنامه سطرهاراآن قدر یک به یک می خواند ونمایش می دهدتاکلمه یwordراتایپ 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 bufferedeader (new inputsteamreader( 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 شیئ های string ایجاد وسپس سطرها خوانده شده وهر سطردر آرایه ذخیره میشود.حداکثر ۱۰۰ سطرخوانده می شود.یا اینکه کلمه ی``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 شئ که System.out به آن ارجاع دارد) تعریف شدهاند.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 OutputStreamمشتق 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 کاراکترA وسپس 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 عمدتا برایمقاصد 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 درPrintWriter تعریف شده است.یکی از این متد ها به صورت زیر است:

```
PrintWriter(OutputStream outputstream,Boolean flushOneNewLine)
```

Outputstream شئ 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 نماییدطر زیر نمونه ای از System.out برای استریم خروجی استفاده نماییدو 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
```

```
4.5e-7
```

خواندن از /نوشتن در فایل ها

جاوا چندین کلاس ومتد را برای خواندن از/نوشتن در فایل ها فراهم کرده است.تمام فایل ها در جاوا بایت گرا هستندو 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 شئ 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 شئ 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 FileNotFoundEXeption
FileOutputStream(String fileName)throws FileNotFoundEXeption
```

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 FileNotFoundEXeption پرتاب می شود,برای استریم های خروجی نیز چنانچه فایل موجود نباشد در آن صورت استثنای FileNotFoundEXeption 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 وآن 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 اجرا ی 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 استفادهمیکند.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 امرمعمولا بدین معناست که برنامه ساز باید عبارات بیشتری در برنامه بگنجاند تا مشخص شود که کدام رویداد واقعا 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 گرداندمعنای آن همیشه واحد است وضعیت EOF پیش آمده است.

توجه:نام فایل های مورد نظر مثل FIRST.TXT وSECOND.TXT باید به عنوان آرگومان متد ()main در خط فرمان باشدو 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 :

مدریت حافظه در جاوا

نحوه عملکرد garbage collector

مدیریت حافظه در Java

یکی از تفاوت های Java با زبانی مثل c وc++ 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 ، آشغال 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 شیئ یا ساختار خود شده، در برنامه خود از این شیئ 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 رابر 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 شیئ 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 شیئ 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 شیئ 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

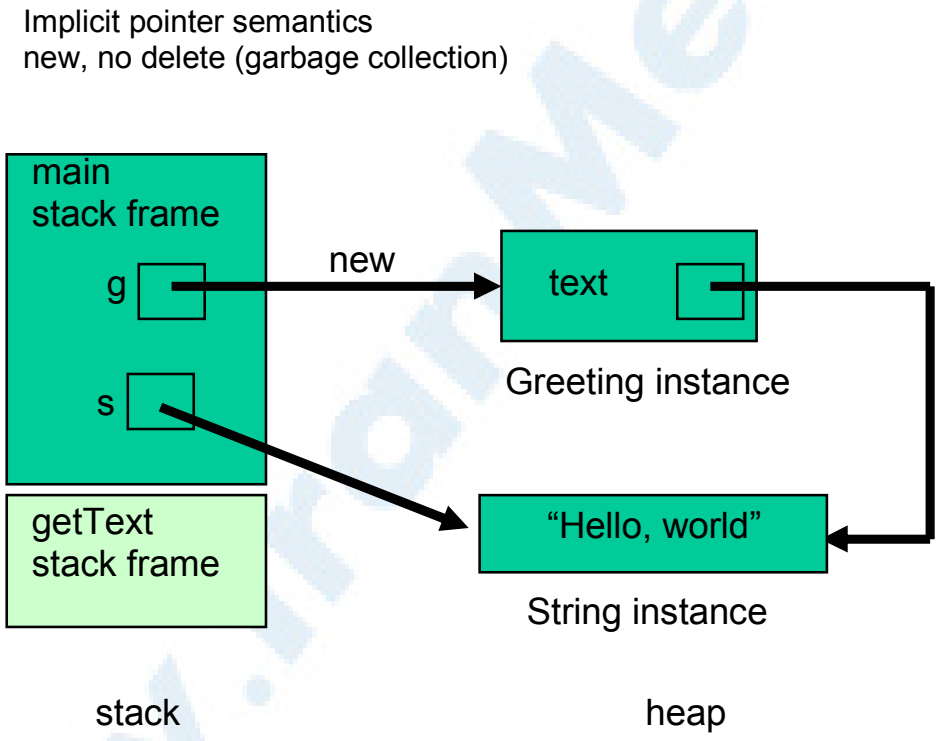
Small Talk متوجه شدند که گرفتن حافظه از سیستم نیز مانند تخصیص آن در هنگام برنامه نویسی، بسیار پر ارزش و مهم بوده و این عمل مهم، به علت عدم توانایی سیستم بر عهده خود برنامه نویس می باشد. از آن هنگام بود که gc ها ایجاد و کار بر روی آنها انجام شد. یکی از ساده ترین روشها در gcهای اولیه، ایجاد یک شمارنده برای هر شیئ است. با هر بار اختصاص منبع(reference) به یک شیئ، 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 gcهای 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 شیئ با حافظه بزرگ بخواهد در سیستم قرار گیرد، ممکن است یک فضای خالی به تنهایی برای آن نداشته باشیم.ولی مجموع فضاهای خالی ما، حتی برای چندین شیئ 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 شیئ، باید توجه کنیم که تمامي منابعی که از آن شیئ داریم، آدرس خود را تغییر داده و به مکان جدید شیئ 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 براي 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 شیئ را از بین ببریم تکلیف چیست؟ این عمل را می توانیم به راحتی و با null کردن آن شیئ (myobject=null) و سپس فراخوانی gc توسط متد system.gc() انجام دهیم.

Java memory model



امکانات ویژه

عناوین این بخش :

مدل زبان و کاربردهای آن

اصول ریزبرنامه ها applets

برنامه نويسي چند نخ كشي شده

Multithreaded programming

نخ اصلي The Main Thread

ايجاد يك نخ

پياده سازي Runnable

بسط نخ

ايجاد نخ هاي چندگانه

استفاده از isAlive() و join()

استفاده از suspend()و resume()

تقدمهاي نخ

اصول ريزبرنامه ها applets

پشتيباني از I/o و ريز برنامه ها از هسته كتابخانه هاي API جاوا ناشي شده اند نه از واژه كليدي اين زبان.

كليه مثالهاي قبلي از برنامه هاي (applications) جاوا بودند. اما برنامه ها فقط يك كلاس از برنامه هاي جاوا مي باشند. نوع ديگر برنامه applet يا همان ريز برنامه است. همانطوريكه قبلا اشاره شده ، " ريز برنامه ها " ، برنامه هاي كوچكي هستند كه روي يك سرويس دهنده اينترنت قابل دسترس بوده و سرتاسر شبكه حمل و نقل شده و بطور خودكار نصب مي شوند و بعنوان بخشي از يك سند وب اجرا مي شوند .

يكبار كه applet روي سرويس گيرنده مي رسد ، دسترسي محدود شده اي به منابع دارد بطوريكه مي تواند يك رابط كاربر چند رسانه اي دلخواه را ايجاد نموده و بدون ايجاد ريسك ويروسها يا نقض تماميت داده ها محاسبات پيچيده را اجرا نمايد . اجازه دهيد با ريز برنامه ساده بصورت زير شروع كنيم :

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
public void paint(Graphics g ){
g.drawString("A Simple Applet"/ 20/ 20);
}
}
```

اين ريز برنامه با د و دستور import آغاز مي شود. اولين دستور كلاسهاي جعبه ابزار پنجره مجرد يا AWT را وارد مي كند. ريزبرنامه ها از طريق AWT با كاربرشان فعل و انفعال دارند نه از طريق كلاسهاي I/o برمبناي كنسول AWT . دربرگيرنده پشتيباني براي يك رابط پنجره اي و گرافيكي است. همانطوريكه ممكن است انتظار داشته باشيد ، اين جعبه ابزار بسيار بزرگ و پيچيده است.

خوشبختانه اين ريز برنامه ساده استفاده بسيار محدودي از AWT دارد. دستور بعدي import بسته Applet را وارد مي كند. اين بسته دربرگيرنده كلاس applet است. هر ريز برنامه اي كه توليد مي كنيد بايد زير كلاسي از Applet باشد.

خط بعدي در برنامه ، كلاس Simple Applet را اعلان مي كند. اين كلاس بايد بعنوان public اعلان شود ، چون بايد توسط كدهاي خارج از برنامه نيز قابل دسترس باشد . روش paint() درون SimpleApplet اعلان شده است. اين روش توسط AWT تعريف شده و بايد توسط ريز برنامه انباشته شود. هرگاه كه ريز برنامه بايد خروجي اش را دوباره نمايش دهد ، paint() فراخواني مي شود . اين شرايط ممكن است به چند دليل اتفاق بيفتد. بعنوان مثال ، روي پنجره اي كه ريز برنامه داخل آن در حال اجراست ممكن است توسط پنجره ديگري نوشته شود و سپس آشكار شود . يا پنجره applet مي تواند بحداقل رسيده و سپس ذخيره شود . وقتي كه ريز برنامه اجرا را شروع مي كند ، paint() نيز فراخواني مي شود. دليل هر چه باشد ، هر وقت ريز برنامه مجبور به نمايش مجدد خروجي خود باشد ، paint () فراخواني مي شود. روش paint() يك پارامتر از نوع Graphics دارد. اين پارامتر دربرگيرنده متن گرافيكي است كه محيط گرافيكي كه applet در آن در حال اجراست را توصيف مي كند.

هر گاه كه خروجي applet مورد نياز باشد ، اين متن مورد استفاده قرار مي گيرد. درون paint() يك فراخواني به drawstring() وجود دارد كه عضوي ازكلاس Graphics است. اين روش يك رشته شروع شده در مكانهاي y وx و مشخص را خارج مي كند. شكل عمومي آن بصورت زير است:

```
Void drawstring( string message, int x, inty )
```

در اينجا message رشته اي است كه درy وx و شروع شده و بايد خارج شود. در يك پنجره جاوا ، گوشه سمت چپ بالايي مكانo وo و است . فراخواني drawstring() در ريز برنامه سبب مي شود پيام "A simple Applet" بنمايش درآيد و در نقطه ٢٠ و ٢٠ آغازشود. دقت كنيد كه ريز برنامه فاقد روش main() است . برخلاف برنامه هاي جاوا ، ريز برنامه ها اجراي خود را در main() شروع نمي كنند. در حقيقت ، اكثر ريز برنامه ها حتي يك روش main() هم ندارند. در عوض ، يك ريز برنامه شروع باجرا مي كند هرگاه كه نام كلاس آن به يك مشاهده گر ريز برنامه (applet viewer) يا يك مرورگر شبكه گذر داده شود . بعداز اينكه كد منبع SimpleApplet را وارد كرده ايد، بهمان روشي كه برنامه ها را كامپايل مي كنيد، ريز برنامه ها را كامپايل مي كنيد. اما اجراي SimpleApplet شامل يك پردازش متفاوت است. در حقيقت ، دو راه براي اجراييك ريز 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 رابررسي نماييم . براي اجراي يك applet در مرورگر وب ، لازم است يك فايل متني HTML كوتاه 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 وheight 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 آزمايش را افزايش مي دهد در بالاي فايل كد منبع جاواي خود يك توضيح (comment) بگنجانيد كه دربرگيرنده دنباله APPLE باشد. بدين ترتيب ، كد شما با يك الگوي 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 آزمايش 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.drwString:"A Simple Applet"/ 20 .20);
}
}
```

در كل مي توانيد از طريق توسعه ريز برنامه با استفاده از اين سه مرحله بسرعت تكرار را اجرا نماييد :

1 : (iterate) يك فايل منبع جاوا ويرايش نماييد .

۲: برنامه اتان را كامپايل كنيد .

۳: مشاهده گر ريزبرنامه را اجرا نموده ، تا نام فايل منبع ريزبرنامه اتان را مشخص نماييد . مشاهده گر ريز برنامه ، داخل توضيح با دنباله APPLET مواجه شده و ريز برنامه شما را اجرا خواهد نمود. پنجره توليد شده توسط SimpleApplet ، آنطوريكه توسط مشاهده گر زيربرنامه بنمايش درآمده بصورت زير مي باشد :

```
|
>< |< | _ |_ ...| pplet Viewer SSimp
| Applet |

| A Simple Applet |
| |
| |
| |
| Applet started .|
```

نكات اصلي كه بايد در اين موضوع بياد داشته باشيد ، عبارتند از:

ريز برنامه ها نيازي به روش main() ندارند. ريز برنامه ها بايد تحت يك مشاهده گر ريز برنامه (applet viewer) يا يك مرورگر سازگار با جاوا اجرا شوند. اربر با كلاسهاي I/o جريان جاوا اجرا نمي شود. در عوض ، ريز برنامه ها.

برنامه نويسي چند نخ كشي شده Multithreaded programming

برخلاف ساير زبانهاي كامپيوتري ، جاوا ، پشتيباني توكار از برنامه نويسي چند نخ كشي شده را فراهم مي كند. يك برنامه چند نخ كشي شده (multithreaded) دربرگيرنده دو يا چند بخش است كه مي توانند بصورت متقارن و همزمان اجرا شوند . هربخش از چنين برنامه اي را يك نخ يا thread مي نامند كه هر نخ مي تواند يك مسير جداگانه از اجرا تعريف نمايد. بنابراين ، چند نخ كشي شده يك شكل تخصصي تر از همان چند وظيفه اي (multitasking) است. بطور حتم با مفهوم چند وظيفه اي آشنا هستيد ، زيرا كليه سيستم هاي عامل مدرن از اين موضوع پشتيباني مي كنند .

دو نوع مجزا از چند وظيفه اي وجود دارد :

بر مبناي پردازش (process-based) و بر مبناي نخ . (thread-based) لازم است كه تفاوت بين اين دو را درك نماييد . يك پردازش از نظر مفهومي يك برنامه در حال اجرا است . بدين ترتيب ، چند وظيفه اي بر مبناي پردازش به كامپيوتر شما امكان مي دهد تا دو يا چند برنامه را بطور متقاون و همزمان اجرا نمايد . بعنوان مثال چند وظيفه اي بر مبناي پردازش به شما امكان مي دهد تا در حين اجراي كامپايلر جاوا و در همان زمان بتوانيد از يك ويرايشگر متن نيز استفاده نماييد . در چند وظيفه اي بر مبناي پردازش ، يك برنامه ، كوچكترين واحد كدي است كه توسط زمانبند (scheduler) توزيع مي شود (dispatched) .

در يك محيط چند وظيفه اي بر مبناي نخ ، كوچكترين واحد كد قابل توزيع ، همان نخ است . اين بدان معني است كه يك برنامه منفرد مي تواند دو يا چندين وظيفه رادر آن واحد انجام دهد. بعنوان نمونه ، يك ويرايشگر متن مي تواند در همان زماني كه مشغول چاپ گرفتن است ، به فرمت كردن يك متن نيز بپردازد ، البته مادامكه اين دو عمل توسط دو نخ جداگانه اجرا شوند . بدين ترتيب ، چند وظيفه اي بر مبناي پردازش با " تصاوير بزرگ" سر و كار دارد در حاليكه چند وظيفه اي بر مبناي نخ جزئيات كارها را اداره مي كند نخ هاي چند وظيفه كننده مستلزم انباشتگي كمتري در مقايسه با پردازشهاي چند وظيفه كننده هستند. پردازشها در اصل وظايف سنگيني هستند كه نيازمند فضاهاي آدرس جداگانه خاص خودشان مي باشند. ارتباطات بين پردازشي (Interprocess) اغلب پر هزينه و محدود است چند كارگي و راه گزيني متن (context switching) از يك پردازش به پردازش ديگر نيز پر هزينه است. در عوض نخها سبك 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 كمتري 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 بيكار باشد. چند نخ كشي به شما اجازه ميدهد تا اين زمان بيكاري را تحت كنترل گرفته و ازآن بنحو مطلوبي استفاده نماييد. اگر تابحال براي سيستم هاي عاملي نظير95 windows يا windows NT برنامه نويسي كرده باشيد ، 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 هنگاميكه برنامه اتان را آغاز مي كنيد ، نخ اصلي بطور خودكار ايجاد مي شود ، اما مي توان آن را از طريق يك شي Thread كنترل نمود. براي انجام اينكار ، بايد با فراخواني روش currentthread() كه يك عضو public static از thread است ، يك ارجاع به آن بدست آوريد. شكل عمومي آن بصورت زير مي باشد:

```
staticThreadcurrentThread()
```

اين روش يك ارجاع به نخي كه در آن فراخواني شده است را بر مي گرداند. هربار كه ارجاعي به نخ اصلي ايجاد كنيد ، مي توانيد آن را مثل هر نخ ديگري تحت كنترل درآوريد. اجازه دهيد با يك مثال شروع كنيم :

```
// 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 اين برنامه ، يك ارجاع به نخ جاري (در اين حالت ، همان نخ اصلي) بوسيله راخواني currentThread () بدست آمده و در متغير محلي t ذخيره مي شود سپس برنامه اطلاعات درباره نخ را نمايش مي دهد. برنامه آنگاه etName() را فراخواني مي كند تا نام داخلي نخ را تعويض نمايد. اطلاعات درباره نخ مجددا بنمايش در مي آيند. سپس ، يك حلقه از عدد ٥ شمارش معكوس مي كند و بين هر دو خط يك ثانيهمكث مي كند. مكث فوق توسط روش sleep() انجام مي شود .

آرگومان به sleep() مشخص کننده دوره تاخير برحسب ميلي ثانيه است. دقت کنيد که بلوك try/catch اين حلقه را احاطه کرده است. روش sleep() در Thread ممکن است يك InterruptedException را پرتاب نمايد. اگر برخي از نخ هاي ديگر بخواهند در اين نخ معوق شده اختلال نمايند، چنين حالتي اتفاق مي افتد . اين مثال اگر دچار وقفه شود ، يك پيام را چاپ مي کند. در يك برنامه واقعي ، بايد اين حالت را طور ديگري اداره نماييد. خروجي توليد شده توسط اين برنامه بقرار زير مي باشد :

```
Current thread :Thread[main/5/main]
After name change :Thread[My Thread/5/main]
5
4
3
2
1
```

دقت کنيد که وقتي t بعنوان يك آرگومان به println () استفاده مي شود ، خروجي توليد مي شود. اين خروجي بترتيب موارد بعدي را نمايش مي دهد :

نام نخ ، حق تقدم آن ، و نام گروه مربوطه آن. بطور پيش فرض ، نام نخ اصلي main است . تقدم آن ٥ است که مقداري پيش فرض مي باشد ، همچنين نام گروهي از نخ ها که اين نخبدان متعلق است ، همان main مي باشد. يك گروه نخ Threadgroup يك نوع ساختار داده است که حالت يك مجموعه از نخ ها را بطور کلي کنترل مي کند. اين پردازش محيط حين اجراي خاصي مديريت شده و در اينجا مورد بررسي قرار نمي گيرد. بعد از اينکه نام نخ تغيير مي يابد ، tمجددا حاصل مي شود . در اين زمان ، نام جديد نخ بنمايش درمي آيد. اجازه دهيد نگاهي دقيق تر به روشهاي تعريف شده توسط Thread که در برنامه استفاده شده اند، داشته باشيم. روش sleep() سبب ميشود تا نخي که از آن فراخواني شده ، اجرا را براي مدت مشخصي از ميلي ثانيه بطور موقت متوقف نمايد. شکل عمومي آن بصورت زير است:

```
static void sleep( long miliseconds )Throws InterruptedException
```

تعداد ميلي ثانيه هايي که بايد تعليق انجام گيرد بر حسب ميلي ثانيه مشخص مي شود . اين روش ممکن است يك InterruptedException را پرتاب نمايد. روش sleep () يك شکل دوم هم دارد

که بعدا نشان ميدهيم و به شما اجازه ميدهد تا مدت زمان را بر حسب ميلي ثانيه و nanoseconds مشخص نمايد.

```
static void sleep( long miliseconds/ int nanoseconds )Throws Interupt
Exception
```

اين شكل دوم فقط براي محيطهايي مناسب است كه امكان زمانبندي دوره هاي زماني را بر حسب nanoseconds دارند. همانطوريكه برنامه قبلي نشان مي دهد ، مي توانيد با استفاده از setName() نام يك نخ را تعيين كنيد. با فراخواني getName() مي توانيد نام يك نخ را بدست آوريد (اما توجه كنيد كه اين رويه در برنامه نشان داده نشده است). اين روشها اعضا كلاس Thread هستند و بصورت زير اعلان مي شوند:

```
final void setName( string ThreadName)
final string getName
```

ايجاد يك نخ

زبان خيلي ساده ، شما با نمونه سازي يك شي از نوع Thread مي توانيد يك نخ را بوجود آوريد . جاوا دو شيوه براي انجام اينكار تعريف مينمايد : مي توانيد رابط Runnable را پياده سازي نماييد. مي توانيد خود كلاس Thread را بسط دهيد. هر كدام از شيوه هاي فوق را بررسي مي كنيم.

پياده سازي Runnable

آسانترين شيوه ايجاد يك نخ ايجاد يك كلاس است كه رابط Runnable را پياده سازي نمايد . Runnable يك واجد از كد اجرايي را مجرد مي كند. مي توانيد روي هر شيئي كه Runnable را پياده سازي مي نمايد ، يك نخ بسازيد. براي پياده سازي Runnable يك كلاس فقط لازم است يك روش تكي موسوم به run() را كه بصورت زير اعلان شده پياده سازي نمايد:

```
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)
```

دراين 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 باجرا نمي كند. از نظر ذاتي ، ()startيك 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 نخاصلي 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 هايشان 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.
```

همانطوريكه قبلا گفتيم ، در يك برنامه چند نخ كشي شده ، نخ اصلي بايد آخرين نخي باشد كه اجرا را پايان مي دهد . اگر نخ اصلي قبل از اينكه يك نخ فرزند كامل شود ، پايان گيرد ، آنگاه ممكن است سيستم حين اجراي جاوا بحالت "hang" درآيد. برنامه قبلي اطمينان مي دهد كه نخ اصلي آخرين نخي است كه پايان مي گيرد زيرا نخ اصلي براي ۱۰۰۰ ميلي ثانيه بين تكرارها معوق مي ماند در حاليكه نخ فرزند فقط ۵۰۰ ميلي ثانيه معوق مي ماند. اين باعث مي شود كه نخ فرزند زودتر از نخ اصلي پايان گيرد. خلاصه ، راه بهتري براي اطمينان از اينكه نخ اصلي آخر از همه پايان گيرد خواهيد يافت.

بسط نخ

دومين شيوه ايجاد يك نخ ، ايجاد يك كلاس جديد است كه Thread را بسط داده و سپس يك نمونه از همان كلاس ايجاد مي كند . كلاس بسط دهنده بايد روش run() را لغو نمايد ، كه نقطه مدخل (entry) براي نخ جديد است. اين كلاس همچنين بايد start() را فراخواني كند تا اجراي نخ جديد را آغاز نمايد . در اينجا برنامه قبلي را دوباره برنامه نويسي نموده ايم 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.");
}
}
```

اين برنامه همان خروجي برنامه قبلي را توليد مي كند. همانطوريكه مي بينيد نخ فرزند بوسيله نمونه سازي يك شي NewThread كه از Thread مشتق شده ، ايجاد مي شود. به فراخواني super () داخل NewThread دقت نماييد. اين امر شكل بعدي سازنده Trhead را فعال مي كند .

```
public thread( string threadName)
```

در اينجا Thread Name ، نام نخ را مشخص مي كند. انتخاب يك شيوه ممكن است تعجب كنيد كه چرا جاوا دو شيوه براي ايجاد نخ فرزند دارد و اينكه كدام شيوه بهتر است. كلاس Thread چندين روش را تعريف مي كند كه بوسيله يك كلاس مشتق شده مي توانند لغو شوند. از اين روشها ، آنكه بايد لغو شود فقط run() است. اين روش البته همان روشي است كه وقتي شما Runnable را پياده سازي ميكنيد لازم است. بسياري از برنامه نويسان جاوا احساس مي كنند كه آن كلاسها فقط وقتي در حال افزايش يا اصلاح شدن هستند ، بايد بسط يابند. بنابراين اگر نمي خواهيد هيچيك از ساير روشهاي thread را لغو نماييد ، احتمالا بهتر است خيلي ساده بستگي دارد.

ايجاد نخ هاي چندگانه

تا بحال فقط دو نخ را استفاده نموده ايد: نخ اصلي و نخ فرزند. اما ، برنامه شما مي تواند به تعداد مورد نياز از نخ ها تكثير نمايد . بعنوان مثال ، برنامهبعدي سه نخ فرزند ايجاد مي كند :

```
// 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( InterruptdException 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.
```

همانطوريكه مي بينيد ، هر بار كه آغاز شوند ، هر سه نخ فرزند ، cpu را به اشتراك مي گذارند به فراخواني sleep١٠٠٠٠ در main ر دقت نماييد. اين باعث مي شود كه نخ اصلي براي ١٠ثانيه معوق مانده و اطمينان مي دهد كه آخر از همه يان مي يابد.

استفاده از isAlive() وjoin()

همانطوريكه ذكر شد ، نخ اصلي بايد آخرين نخي باشد كه پايان مي گيرد. در مثالهاي قبلي ، اينكار را با فراخواني sleep() داخل main() با يك تاخير بحد كافي طولاني براي اطمينان از اينكه كليه نخ هاي فرزند قبل از نخ اصلي پايان مي گيرند ، انجام داديم . البته اين يك راه حل بسختي قانع كننده است. اين راه حل در ضمن يك سوال را برمي انگيزد : چگونه يك نخ مي تواند از پايان گرفتن نخ ديگر آگاهي يابد ، خوشبختانه Thread وسيله اي فراهم نموده كه توسط آن مي توانيد پاسخ اين پرسش را بدهيد .دو شيوه وجود دارد تا تعيين كنيم كه آيا يك نخ پايان گرفته است يا نه . اول مي توانيد isAlive() را روي نخ فراخواني كنيد. اين روش توسط Thread تعريف شده و شكل كلي آن بصورت زير مي باشد :

```
final boolean isAlive (Throws InterruptedException)
```

اگر نخ بالايي آنكه فراخواني شده همچنان در حال اجرا باشد ، روش isAlive() مقدار true را برمي گرداند . در غير اينصورت false را برمي گرداند . در حاليكه isAlive() گهگاه سودمند است ، روشي كه بطور رايج مورد استفاده قرار مي گيرد تا براي پايان يافتن يك نخ منتظر بمانيد ، join() است كه بصورت زير مي باشد :

```
final void join (throws InterruptedException)
```

اين روش منتظر مي ماند تا نخي كه روي آن فراخواني شده پايان گيرد. شكلهاي ديگري از join() وجود دارند كه به شما اجازه مي دهند تا همچنين حداكثر زماني كه مي خواهيد براي پايان يافتن يك نخ خاص صبر كنيد را تعيين نماييد .

در اينجا يك روايت اصلاح شده از مثال قبلي وجود دارد كه از join() استفاده كرده تا اطمينان دهد كه نخ اصلي آخرين نخي است كه متوقف مي شود . اين برنامه همچنين روش isAlive() را نشان مي 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( InterruptdException 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 زير 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()

گاهي لازم است اجراي يك نخ را لغو نماييم . بعنوان مثال با استفاده از يك نخ جداگانه مي توان وقت را نشان داد. اگر كاربر تمايلي به استفاده از ساعت نداشته باشد ، پس نخ مربوط به آن بايد لغو شود. موضوع آن هر چه باشد ، لغو نمودن يك نخ كار ساده اي است. همچنين بكار انداختن مجدد يك نخ لغو شده نيز كار ساده اي است. روشهايي كه ايندو وظيفه را انجام ميدهند عبارتند از suspend()وresume() . آنها توسط Thread تعريف شده و بصورت زير مي باشند :

```
final void resume()

final void suspend()
```

برنامه بعدي اين روشها را نشان مي 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( InterruptdException 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.");
}
}
```

اين برنامه خروجي بعدي را توليد مي كند :

```
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 كدام 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 كليه 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

شرايط بلوك سازي نظير انتظار براي I/o مواجه خواهند شد . وقتي اين اتفاق مي افتد ، نخ بلوكه شده لغو شده و ساير نخ ها مي توانند اجرا شوند . اما اگر مي خواهيد اجراي چند نخي شده را بنرمي انجام دهيد، نبايد روي اين اصل متكي باشيد . همچنين ، برخي انواع وظايف وجود دارند كه گرايش به cpu دارند چنان نخ هايي بر cpu چيره خواهند شد . روي اين نوع از نخ ها ، بايد گهگاه كنترلي داشته باشيد تا ساير نخ ها بتوانند اجرا شوند.

براي تعيين تقدم يك نخ ، از روش setpriority() استفاده نماييد ، كه عضوي ازThread است . شكل عمومي آن بقرار زير است :

```
final void setpriority( in level)
```

در اينجا level توصيفگر تعيين تقدم جديد براي فراخواننده است. مقدار level بايد داخل محدوده MIN-PRIORITY و MAX-PRIORIT باشد. در حال حاضر، اين مقاديرl و ۱۰ ميباشند . .براي برگرداندن يك نخ به تقدم پيش فرض ، NORM-PRIORITY را مشخص مي كنيد كه فعلا 5 است . اين تقدمها بعنوان متغيرهاي final داخل Thread تعريف شده اند .

مي توانيد تعيين تقدم جاري را با فراخواني روش getpriority()درThread ر بدست آوريد ، كه بصورت زير مي باشد :

```
final int getpriority()
```

فعلا پياده سازي جاوا ، وقتي كه زمانبندي پيش مي آيد ، بصورت شديدي رفتار متفاوتي نشان مي دهد . روايت ويندوز ۹۵ كمابيش همانطوريكه انتظار داريد كار مي كند. اما روايت solaries بگونه اي متفاوت كار مي كند. بسياري از ناسازگاريها هنگامي بروز مي كنند كه شما نخ هايي داشته باشيد كه بجاي اينكه زمان cpu را بصورت اشتراكي مصرف نمايند ، متكي به رفتار وابسته به پيش دستي باشند . براي كسب رفتار ارجاع متقابل قابل پيش بيني با جاواي امروز، بايد از نخهايي استفاده كنيد كه بطور اختياري از كنترل نمودن cpu دست برمي دارند.

مثال بعدي دو نخ با تقدمهاي مختلف را نشان مي دهد، كه روي دو محيط زيربنايي ذكر شده اجرا نخواهند شد . يك نخ دو سطح بالاتر از تقدم معمولي تعيين شده كه بوسيله-Thread NORM PRIORITY تعريف شده است و ديگري دو سطح پايين تر از تقدم معمولي تعيين شده است . نخ ها آغاز شده و بمدت ۱۰ ثانيه اجازه اجرا دارند. هر نخ يك حلقه را اجرا مي كند ، كه تعداد

تکرارها را شمارش مي کند. بعد از ده ثانيه ، نخ اصلي هر دو نخ را متوقف مي کند. سپس تعداد دفعاتي که هر نخ از طريق حلقه ساخته شده ، بنمايش در مي آيد.

```
// 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 l/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 كشي كردن

اگر مثل ساير برنامه نويسان باشيد ، پس در اختيار داشتن پشتيباني توكار از چند نخ كشي شدن در زبان برنامه نويسي براي شما تازگي خواهد داشت. كليد استفاده موثر از اين پشتيباني اين است كه بصورت " همزماني " تفكر نماييد نه بصورت سريالي. بعنوان مثال ، وقتي دو زير سيستم داخل يك برنامه داريد كه مي توانند همزمان اجرا شوند ، آنها را به نخ هاي منفرد تقسيم نماييد. با استفاده محافظه كارانه از چند نخ كشي كردن ، مي توانيد برنامه هاي بسيار موثري بنويسيد .اما اگر تعداد زيادي از نخ ها ايجاد نماييد، سبب افت عملكرد برنامه اتان خواهيد شد . بياد داشته باشيد كه برخي بالاسريها با conext switching همراه هستند . اگر تعداد زيادي از نخ ها ايجاد كنيد، آنگاه وقت cpu بيشتر از آنكه براي اجراي اص يابد ، صرف تغيير context ها خواهد شد.

پیوست

در این بخش شما ، گام به گام آنچه را در فصل هاي پیشین آموخته اید به صورت عملي به کار خواهید گرفت. سعي شده است با ارائه مثال هایي مناسب زیبایي جاوا را درك کنید.

موفق باشید

برنامه اي که قصد توضيح آن را داريم ، بسيار ساده و مختصر است . برنامه فوق محيط لازم برای رسم يک خط قطری را ايجاد می نمايد.بدين منظور عمليات زير را می بايست انجام داد :

- برنامه Notepad را فعال و برنامه مورد نظر را در آن تايپ نمائيد.
- برنامه را ذخيره نمائيد.
- برنامه نوشته شده را با استفاده از کمپايلر جاوا ترجمه تا يک اپلت جاوا ايجاد گردد.
- در صورت گزارش خطاء ، نسبت به رفع آنها اقدام گردد.
- يک صفحه وب Html ايجاد و از اپلت ايجاد شده در آن استفاده نمائيد.
- اپلت جاوا را اجراء نمائيد.

متن برنامه اشاره شده بصورت زير است :

متن برنامه

```
import java.awt.Graphics;

public class FirstApplet extends
java.applet.Applet
{

    public void paint(Graphics g)
    {
        g.drawLine(0, 0, 200, 200);
    }
}
```

مرحله يک : تايپ برنامه

بمنظور ذخيره نمودن برنامه ، فولدری با نام دلخواه ايجاد تا برنامه در آن ذخيره گردد. در ادامه ويرايشگر Notepad (و يا هر اديتور متنی ديگری که قادر به ايجاد فايل های با انشعاب TXT باشد) را فعال و برنامه فوق را تايپ (و يا Copy و Paste) نمائيد. در زمان تايپ برنامه فوق می بايست در رابطه با حروف بزرگ و کوچک دقت لازم صورت پذيرد. در اين رابطه لازم است که حروف بزرگ و کوچک دقيقا مشابه جدول فوق ، تايپ گردند.

مرحله دوم : ذخيره کردن فايل

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 هائی 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 تگهداری 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 (۲۰۰،۲۰۰،۰،۰). 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.Graphice.html است . فایل فوق کلاس Graphic را تشریح می نماید. drawLine صرفا" 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 . مثلا" 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 نمائیم . 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 (مثلا 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 مجددا 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 مجددا برنامه را کمپایل و اجراء نماید.

متغیرها

تمام برنامه های کامپیوتری ، بمنظور نگهداری موقت اطلاعات از متغیر ها استفاده می کنند. مثلا 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. مثلا 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 (مثلا 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 ، صرفا 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

```
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);
    }
}
```

در برنامه ارائه شده ، همواره یک خط قطری در پنجره ای با ابعاد ۲۰۰ * ۲۰۰ پیکسل ، رسم می گردد. عدم امکان پویائی 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 قطرها بصورت کامل در محدوده مربوطه قرار خواهند گرفت . پس از تغییر اندازه پنجره ، قطرها و Rectangle براساس مقادیر جدید بصورت خودکار مجددا 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 صرفا" 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 ازمتغیرها ، 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 r.width - 1 را مستقیما" 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

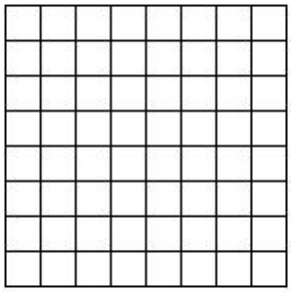
```
float diameter = 10;
float radius;
float volume;

radius = diameter / 2.0;
volume = 4.0 / 3.0 * 3.14159 * radius * radius *
radius;
```

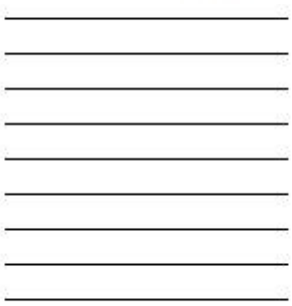
حلقه های تکرار

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 محموعه 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);
        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 صرفا" یک مرتبه تکرار و فرآیند تکرار عملیات را بر عهده حلقه تکرار قرار داد. بدین ترتیب یک حلقه تکرار (loop) ایجاد می گردد.

متن برنامه

```
import java.awt.Graphics;

public class FirstApplet extends
java.applet.Applet
{

    public void paint(Graphics g)
    {
        int y;
        y = 10;
        while (y <= 210)
        {
            g.drawLine(10, y, 210, y);
            y = y + 25;
        }
    }
}
```

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 مجددا" تکرار (دستورات محصور بین { و }) می گردننده. 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 شد ه و مجددا به ابتدای حلقه (بررسی شرط) مراجعه می گردد. مقدار ۳۵ از ۲۱۰ کوچکتر بوده و شرط همچنان درست بوده و مجددا" دستورات موجود در حلقه تکرار ، اجراء خواهند شد. فرآیند فوق مادامیکه مقدار y کوچمتر 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

```
import java.awt.Graphics;

public class FirstApplet extends
java.applet.Applet
{

    public void paint(Graphics g)
    {
        int x, y;
        y = 10;
        while (y <= 210)
        {
            g.drawLine(10, y, 210, y);
            y = y + 25;
        }
        x = 10;
        while (x <= 210)
        {
            g.drawLine(x, 10, x, 210);
            x = x + 25;
        }
    }
}
```

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

```
for (y = 10; y <= 210; y = y + 25)
    {
      g.drawLine(10, y, 210, y);
    }
```

حلقه تکرار با استفاده از While

```
y = 10;
   while (y <= 210)
   {
      g.drawLine(10, y, 210, y);
      y = y + 25;
   }
```

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 بهمراه جاوا برای تکمیل اطلاعات خود استفاده نمایند

منابع

آموزش جاوا در ۲۱ روز – لورالمي ، چارلز پرکينز – مترجم : عليرضا زارع پور

جاوا – هربرت شيلد – مترجم : فرهاد قلي زاده نوري

آموزش گام به گام برنامه نويسی جاوا – عين ا... جعفرنژاد

جاوا رهيافتی شی گرا – مترجم : عليرضا منتظرالقائم

ساختمان داده ها و الگوريتم ها در جاوا – مترجم : کيارش بحرينی

Persian Refrence :

www.dev.ir

www.java.schoolnet.ir

www.sarzemine-it.com

www.sohail2d.com/forum

www.paradise19791979.persianblog.com

www.qomcse.com/forum

English Refrence :

Java programming primer.pdf (e-book)

www.cs.umb.edu/~serl/java/ppt/

www.iut-orsay.fr/~fournier/Cork/OOP.pdf

راهنمای برنامه نویسی جاوا
برای
مهندسین نرم افزار

Java programming
for
Software Engineers