باسمه تعالی

Java

تألیف و ترجمه : عسکر قندچی

مقدمه

زبان برنامه نویسی جاوا بوسیله شرکت سان میکروسیستمز(Sun Microsystems) در اوایل دهه ۱۹۹۰ ساخته شده و یک زبان شئ گراست. این زبان حاصل چندین سال تلاش برای ایجاد زبان برنامه نویسی بهتر و محیط ساده تر برای تولید برنامه های مطمئن تر است. وجود کتابخانه ای غنی از کلاسها ، تفکر شئ گرایی و کاربرد فراوان آن در شبکه های کامپیوتری (بویژه اینترنت) ، باعث شده است که از محبوبیت خاصی برخوردار شده و کاربرد آن بطور فزاینده ای در حال گسترش باشد.

برنامه نویسی شئ گرا Object-Oriented Programming(OOP)

برنامه نویسی شئ گرا به یک زبان خاص بستگی ندارد بلکه یک سبک برنامه نویسی است. زبان Simula67 ، اولین زبان طراحی شده بر اساس شئ گرایی است و زبان Smalltalk اولین زبان کاملاً شئ گراست. زبانهای شئ گرا دارای چهار قابلیت زیر هستند:

1. Encapsulation (بسته بندی)
2. Abstraction (ایزوله کردن یا پنهان سازی جزئیات)
3. Inheritance (وراثت)
4. Polymorphism (چند ریختی)

در برنامه نویسی ساخت یافته ، برنامه نویس به این سؤالها پاسخ می دهد: "داده ها چگونه در برنامه جابجا می شوند؟ و ورودیها و خروج های برنامه چیستند؟" ، اما در برنامه نویسی به روش شئ گرا با دو سؤال دیگر مواجه است: "با چه چیزهایی باید کار کنم؟ و انتظار دارم این چیزها چه کار کنند؟". لذا این دو شیوه برنامه نویسی در واقع دو نوع طرز تفکر یا به عبارت دیگر دو دیدِ مختلف هستند.

در اغلب موارد مفاهیم بصورت یک چیز انتزاعی و مستقل مطرح نمی شوند و اکثر مفاهیم به یکدیگر وابسته هستند. به عنوان مثال مفاهیم هواپیما و اتومبیل ، هر دو به مفهوم وسیله نقلیه و مفهوم حمل و نقل وابسته هستند و یا اینکه مفاهیم دایره ، مستطیل و چند ضلعی با مفهوم شکل در ارتباط هستند. اگر مفاهیم در برنامه ها مستقیماً به عنوان نوع (class) مطرح شوند می توان متعاقباً رابطه بین انواع مختلف را نمایش داد. شئ گرایی ، برنامه نویس را قادر می سازد که کلاسهایی (انواعی) تعریف کند که خواص خود را از

کلاسهای دیگر به ارث می برند. این مسأله ، کل چیزی است که در برنامه نویسی شئ گرا رخ می دهد. سازماندهی برنامه ها براساس سلسله مراتب میراثی ، یک روش بسیار مهم برای انجام کاربرد های پیچیده است.

در تفکر شئ گرایی ، قطعات برنامه ها به عنوان اشیایی مثل دنیای واقعی در نظر گرفته می شوند. سپس برای رسیدن به نتیجه مطلوب ، اشیاء دستکاری می شوند. اشیای موجود در برنامه نویسی شئ گرا همانند اشیای موجود در دنیای واقعی دارای صفات (attributes) و رفتار هایی (behavior) هستند. اتومبیل دارای صفاتی مثل مدل ، رنگ ، قیمت ، دنده ، سرعت و غیره است. تمام اتومبیل ها صفات یکسانی دارند ولی مقادیر آنها متفاوت است. بنابراین اتومبیل ها تشکیل یک کلاس (class) را می دهند که مثلاً اتومبیل پیکان نمونه ای (instance) از آن کلاس است. این تفکر موجب می شود که برنامه نویس ، دانش خود را در مورد یک کلاس ، درمورد هریک از نمونه های آن کلاس به کار گیرد. هر نمونه ای از یک کلاس ، صفات خود را از کلاس به ارث (inherit) می برد. به همین ترتیب ، در محیطی مثل ویندوز ، هر قطعه دارای صفاتی مثل نوار منو و نوار عنوان می باشد ، زیرا هر قطعه این صفات را از یک کلاس کلیِ قطعات به ارث می برد.

اشیاء برای انجام وظایف خود دارای رفتار هایی هستند که آنها را متُد (method) می نامیم. به عنوان مثال ، اتومبیل می تواند به جلو برود ، به عقب برود ، و ترمز کند. اتومبیل ها می توانند سوختگیری کنند و شسته شوند که منجر به تغییر صفات آنها می شود. اتومبیل ها متُد هایی برای تعیین صفاتی مثل سرعت و میزان سوخت نیز دارند. بدین ترتیب ، برنامه های شیءگرا همانند برنامه های ساخت یافته ، دارای متغیر ها(صفات) و رَویه ها (متُد ها) هستند ، ولی صفات و رَویه ها در اشیاء بسته بندی (Encapsulation) می شوند. بسته بندی تکنیکی است که موجب می شود شئ به عنوان یک واحد ، دستیابی شود.

برای برنامه نویسی در جاوا ، باید کلاسهایی را برای مدلسازی اشیای دنیای خارج ایجاد کنیم برای این منظور باید صفات و متُد های آن را تعیین کرده ، متُد ها را برای دستکاری صفات آنها بنویسیم. اما صفات کلاسها همان متغیر هایی هستند که باید بر روی آنها عملیاتی صورت گیرد. خلاصه اینکه ، همه چیز در برنامه نویسی جاوا ، جزئی از یک کلاس است. در واقع کلاس یک قالب است که اشیاء از آن ساخته می شوند. به عنوان مثال ، اگر نقشه ساختمان را یک کلاس در نظر بگیریم ، ساختمان هایی که از آن نقشه ساخته می شوند ، اشیایی از آن کلاس محسوب می شوند. بنابراین شئ را در یک زبان برنامه نویسی شیءگرا مثل جاوا ، نمونه ای از کلاس می دانیم.

ویژگی های جاوا **Java Features**

موارد زیر از جمله ویژگی های این زبان است:

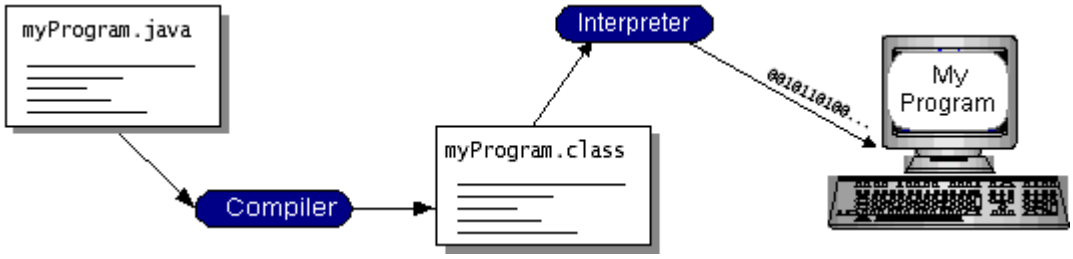
1. Simple (C/C++ Syntax)
2. Portable (Write once, Run anywhere)
3. Secure
4. Object-Oriented
5. Robust (Automatic Memory Management ,Exception Handling)
6. Multithreaded
7. Architecture-neutral
8. Interpreted and High Performance
9. Distributed (RMI: Remote Method Invocation)
10. Dynamic

۱- ساده و آسان بودن یادگیری

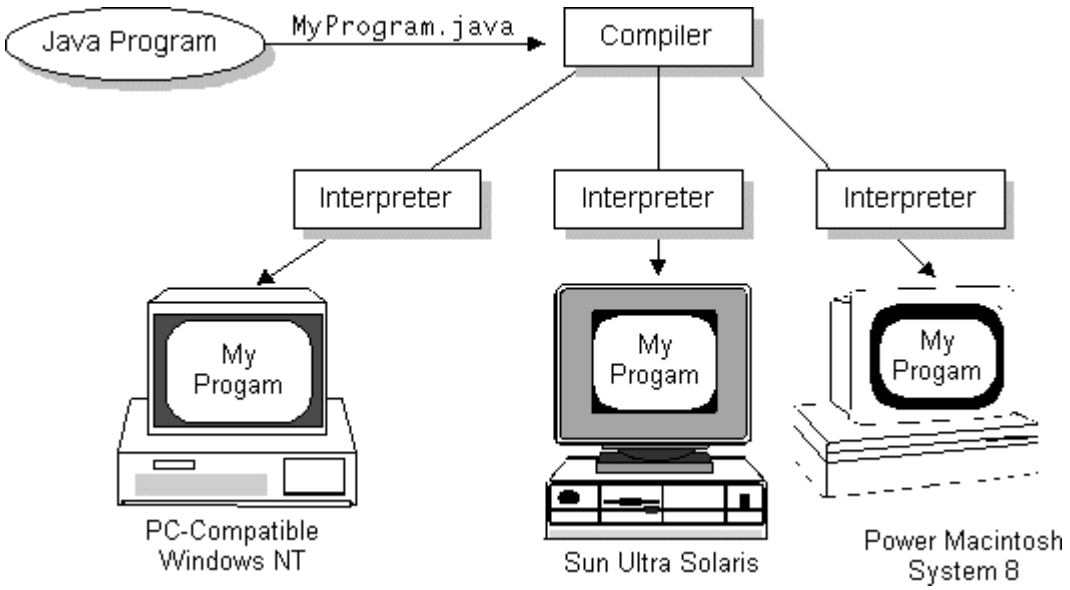
طراحی این زبان بگونه ای است که یادگیری آن برای برنامه نویسان (کسانی که تجربه برنامه نویسی دارند) آسان است. اگر با مفاهیم برنامه نویسی شئ گرا آشنا باشد آسان تر خواهد بود بویژه اگر تجربه برنامه نویسی با C++ را داشته باشد. زیرا جاوا ، Syntax و بسیاری از ویژگیهای شئ گرایی را از C++ به ارث برده است. همچنین اشکالات شناخته شده آنرا نیز مرتفع کرده است.

۲- قابلیت حمل

جاوا مستقل از محیط است یعنی در محیط ها و سیستم عامل های مختلف قابل اجراست و این ویژگی یکی از مهمترین امتیازات جاوا می باشد. جاوا ، استقلال از محیط را با استفاده از ماشین مجازی جاوا (Java Virtual Machine) انجام می دهد. در واقع ، ماشین مجازی ، کامپیوتری در کامپیوتر دیگر است. ماشین مجازی برنامه های کامپایل شده جاوا را گرفته و دستورات آن را به فرمان هایی تبدیل می کند که سیستم عامل می تواند آنها را اجرا کند. ماشین مجازی جاوا را ، مفسر(interpreter) جاوا نیز می گویند.



کد ماشین مجازی را بایت کد (Bytecode) گویند. برنامه کامپایل شده که به صورت بایت کد است ، می تواند بر روی هر کامپیوتر و سیستم عاملی که ماشین مجازی جاوا (JVM) را دارد اجرا شود.



بنابراین برنامه های کاربردی جاوا فقط در سیستم هایی می توانند اجرا شوند که ماشین مجازی جاوا بر روی آنها نصب شده باشد. با توجه به اینکه ماشین مجازی جاوا یک سطح اضافی بین برنامه جاوا (کد منبع) و کد ماشین ایجاد می کند ، اجرای برنامه

های جاوا کند است. در صورتی که لازم است سرعت اجرای برنامه های نوشته شده به زبان جاوا بیش از ماشین مجازی باشد ، می توان به یکی از دو روش زیر عمل کرد:

الف- در برنامه جاوا از کد ماشین خاصی استفاده کرد که منجر به برنامه ای می شود که وابسته به محیط است.

ب - از کامپایلر های بی درنگ استفاده کرد که بایت کد جاوا را به کدی برای ماشین های خاص تبدیل می کند.

در هر یک از این راه حل ها ، قابلیت حمل جاوا ، فدای سرعت شده است. به عنوان مثال برنامه کاربردی که از فراخوانی های ویندوز برای دستیابی به دیسک استفاده می کند ، بدون انجام تغییرات ، در مکینتاش کار نمی کند.

۳ - امنیت

Download کردن برنامه های اجرایی از اینترنت و اجرای آنها با ریسک هایی مانند ویروسی بودن برنامه ها و یا جستجوی پنهانِ کامپیوتر بوسیله آن برنامه ها ، برای جمع آوری اطلاعات شخصی مانند شماره کارت اعتباری ، مانده حساب بانکی ، Password و ... همراه است. اما برنامه های اجرایی جاوا در اینترنت که applet نامیده می شوند اینطور نیستند با توجه به ویژگی قابلیت حمل برنامه های جاوا ، برای اجرای applet ها ، وجود JVM الزامی است و چون اجرای هر برنامه جاوا تحت کنترل JVM است ، اجازه دسترسی به سایر بخش های کامپیوتر را به این گونه برنامه ها نخواهد داد.

۴ - شئ گرایی

شاید بتوان سیر تکاملی برنامه نویسی را در مراحل زیر خلاصه کرد.

قدیمی ترین انواع آن که حتی فاقد ساختار برنامه فرعی هستند شیوه *کد اسپاگتی* (*Spaghetti code*) می باشد و نمونه آن ، انواع اولیه بیسیک است و با صفت هرج و مرج قابل توصیف است.

اولین پیشرفت نسبت به شیوه هرج و مرج ، الگوی برنامه نویسی *پروسه ای* (*Procedural*) بود. زبان هایی از قبیل فرترن و کوبول در این دسته جای می گیرند که در این شیوه برنامه به چندین قسمت (تابع) تقسیم می شود. تغییر بعدی ، الگوی برنامه نویسی *ساخت یافته* (*Structured*) است در این الگو بجای آنکه بین توابع هرج و مرج برقرار باشد قوانینی ارایه شده است که هر ساختار ، نظیر for ، while ، if و ... را می توان متعاقباً ایزوله کرد. در این شیوه توصیه می شود که انواع داده ای جدیدی تعریف شود که توصیفی دقیق تر از شئ حقیقی آنها ایجاد شود. یک نمونه مناسب از این گونه زبانها پاسکال است. البته شیوه ای که بوسیله اکثر برنامه نویسان مورد استفاده قرار می گرفت چیزی بین روش پروسه ای و ساخت یافته بود.

برنامه نویسی *مدولار* (*Modular*) یا *قطعه – قطعه* نوع توسعه یافته ای از برنامه نویسی ساخت یافته است در این الگو توابع به فایل های برنامه ای مجزای سازگار با هم تقسیم می شوند که به هریک از این تکه برنامه ها ، یک مدول یا ماژول می گویند که هر ماژول دارای مجموعه ای از توابع و داده هاست که تنها خود به آنها دسترسی دارد. در بهترین برنامه های به زبان C از این الگوی برنامه نویسی استفاده می شود متأسفانه تبعیت از این الگو در C اختیاری است.

آخرین الگوی برنامه نویسی که امروزه از آن استفاده می شود برنامه نویسی *شئ گرا* (*Object-Oriented*) است و به جُرأت می توان گفت که جاوا ، شئ گرایی را در زبانهای برنامه نویسی به اوج خود رسانده است.

۵ - قابلیت اطمینان

تنوع فناوری ها در محیط های مختلف وب که برنامه های جاوا باید در تمامی آن محیط ها با اطمینان و درست اجرا شوند ، قابلیت تولید برنامه های قدرتمند و قابل اطمینانی را می طلبد که این مطلب را جاوا با کنترل کردن برنامه در چند ناحیه کلیدی بدست

می آورد و برنامه نویس را ملزم به رفع خطا ها در زمان توسعه سیستم می کند. جاوا یک زبان برنامه نویسی سخت گیر است و برنامه را هم در زمان کامپایل و هم در زمان اجرا بررسی می کند. همچنین از جمله مواردی که باعث خطا می شوند مدیریت اشتباه حافظه و عدم بررسی استثنا هاست. که این دو مشکل با مدریت خودکار حافظه و Exception Handling شئ گرای جاوا حل شده است.

۶ – چند بندی

این ویژگی ، قابلیت انجام (اجرای) چند کار بوسیله یک برنامه جاوا را به طور هم زمان فراهم می آورد. البته این قابلیت با موضوع Multitasking سیستم عامل متفاوت است.

۷ – معماری بی طرف (مستقل از فناوری)

یکی از مسایلی که همواره ذهن برنامه نویسان را به خود مشغول می کند این است که هیچ تضمینی برای اینکه برنامه نوشته شده در کامپیوتر خودشان در سایر محیط ها و سخت افزار ها بدرستی عمل کند وجود ندارد. طراحان جاوا حل این مسأله را جزو اهداف اصلی خود قرار دادند و در محدوده وسیعی به این هدف رسیدند. شعار آنها Write once; Run anywhere, Any time, Forever. است.

۸ – مفسری و کارآیی

جاوا امکان تولید برنامه هایی با قابلیت اجرا در بسترهای متنوع را با کامپایل کردن برنامه به بایت کد فراهم می آورد و بایت کد بوسیله JVM تفسیر شده و به کد ماشینِ مناسب محیط تبدیل می شود. با توجه به اینکه برنامه هایی که بوسیله مفسرها اجرا می شوند ذاتاً کند هستند لذا اجرای کنْدِ برنامه های جاوا مورد انتظار است اما باید گفت که این کنْدی تا حدودی با اضافه شدن یک کامپایلر بایت کد به JVM حل شده است به این ترتیب که بایت کد ها قطعه – قطعه به کد ماشین محلی کامپایل شده و اجرا می شوند و با این تکنیک مزایای مربوط به مفسر (JVM) ، مانند امنیت و قابلیت حمل پا برجا می ماند.

۹ – امکان توزیع یک برنامه بر روی بیش از یک ماشین

این ویژگی امکان اجرای اشیای یک برنامه را در چند کامپیوتر فراهم می آورد. در واقع متُد های اشیاء می توانند از کامپیوتر های دیگر فراخوانی شوند.

۱۰ – پویایی

برنامه های جاوا بطور ذاتی مقداری از اطلاعات زمان اجرا را برای بررسی و حل مسایل دسترسی به اشیاء در زمان اجرا ، با خود حمل می کنند. این موضوع امکان اتصال پویای کد ها را با روشی مناسب و امن فراهم می کند. این مسأله در محیط Applet سخت است که تکه های کوچک بایت کد ، بصورت پویا روی یک سیستم در حال اجرا Update شوند که این مشکل بوسیله JVM حل شده است.

انواع داده اولیه Primitive Data Types

در جاوا ، چهار نوع اولیه وجود دارد که عبارتند از صحیح ، بولین ، اعشاری و کاراکتری.

Type	Size (Bit)	نوع
byte	8	صحیح
short	16	
int	32	
long	64	
float	32	اعشاری
double	64	
char	16	کاراکتری
boolean	8	بولین

*- متغیر های عددی مقادیر مثبت و منفی را با توجه با اندازه شان می پذیرند.

تمرین

با توجه به اینکه جاوا بخش اَعظمی از Syntax خود را از C به ارث برده است لذا یادآوری مباحث زیر و بررسی تفاوت های جزئی آن با جاوا به دانشجویان واگذار می شود:

اعلان متغیر ها ، مقدار دادن به متغیر ها ، ثوابت ، مقادیر ثابت (Literals) ، عملگر ها (محاسباتی ، رابطه ای ، منطقی ، بیتی ، ترکیبی و ?) ، تقدم عملگر ها ، تبدیل انواع (Type casting) ، دستورات مرکب ، ساختار های تکرار ، ساختار های تصمیم گیری ، کاراکتر های کنترلی و افزودن توضیحات.

مقدمات

تمام تعاریف داده ها و متُد ها داخل کلاس ها انجام می شود و خارج از کلاس چیزی قابل تعریف نیست. هر کلاس جاوا در یک فایل متنی بصورت جداگانه ذخیره می شود که نام فایل ، همان نام کلاس ، و پسوند آن java می باشد. با توجه به Case Sensitive بودن جاوا ، بزرگ و کوچک بودن حروف نام فایل باید عیناً مانند نام کلاس تعریف شده در داخل آن باشد.

هر برنامه جاوا حداقل از یک کلاس تشکیل می شود. و هر کلاس باید داخل پکیجی قرار گرفته باشد ، اگر در برنامه ای کلاسی ایجاد شود ولی در داخل پکیجی قرار نگیرد ، جاوا یک پکیج پیش فرض برای آن در نظر می گیرد.

نقطه شروع اجرای برنامه جاوا از کلاسی است که حاوی متُد main است و در کلاس مذکور نیز با اجرای متُد main آغاز می شود. هر نامی را می توان به عنوان نام کلاسی که حاوی متُد main است انتخاب کرد ولی توصیه می شود که نامی متناسب با عملکردش برای آن انتخاب کرد ، همچنین برای نامگذاری سایر کلاس ها نیز بهتر است از اسامی با مُسَما استفاده کرد. الگوی یک برنامه ساده جاوا به شکل زیر است:

```
تعریف کلاس ----------◄ نام کلاس public class
{
   public static void main (String[] arguments)  ◄---------- تعریف متُد
   {
      دستورات متُد
   }
}
```

مثال: برنامه ای که جمله ای را در خروجی چاپ می کند.

```
public class FirstProgram
{
   public static void main (String[] args)
   {
      System.out.println ("This is a simple java program.");
   }
}
```

برای چاپ چیزی در صفحه نمایش از متُد های print و println استفاده می کنیم. توضیحات دقیق آن پس از مطالب مربوط به پکیج ارایه خواهد شد. برنامه های جاوا را می توان با یکی از نرم افزار های زیر کامپایل و اجرا کرد:

1. JDK (Java Development Kit)
2. NetBeans
3. Sun Java Studio
4. Forte for Java
5. JDeveloper
6. Jbuilder
7. Eclipse
8. IntelliJ IDEA

برای اجرای برنامه بوسیله JDK به روش زیر عمل می کنیم:

```
javac  FirstProgram.java
```

javac، برنامه FirstProgram.java را کامپایل کرده و در صورت عدم وجود خطا ، فایل بایت کد آن را به نام FirstProgram.class تولید می کند. سپس آن را بوسیله مفسر جاوا اجرا می کنیم:

```
java  FirstProgram
```

*- می توان محل فایل های javac.exe و java.exe را با استفاده از دستور path مشخص کرد. مانند:

```
path=%path%;JavaHome\bin
```

و در دایرکتوری دیگری که برای نگهداری Source code ها در نظر گرفته ایم کار کرد.

استاندارد های نامگذاری

۱- نام کلاس با حرف بزرگ آغاز می شود.

۲- نام پکیج ها ، خصوصیت ها و متُد ها با حرف کوچک شروع می شوند.

۳- اگر نامی بیش از یک کلمه داشت ، کلمات دوم به بعد با حرف بزرگ شروع می شوند.

۴- در نامگذاری کلاسها ، خصوصیت ها و متُد ها از اسامی مخفف استفاده نمی کنند اما انجام این کار در مورد نامگذاری پکیج ها مجاز است.

کلاس ها و اشیاء **Classes and Objects**

کلاس ، اساس جاواست و کل جاوا بر روی ساختار منطقی آن ساخته شده است. هر مفهومی برای پیاده سازی در جاوا باید در یک کلاس ، بسته بندی (Encapsulate) شود. شاید مهمترین مطلبی که لازم است در مورد کلاس بدانیم این است که کلاس یک نوع داده جدید ایجاد می کند. در تعریف یک کلاس ، داده ها و عملیاتی که روی آن داده ها صورت می گیرد مشخص می شوند. هرچند که کلاس می تواند فقط حاوی داده یا فقط حاوی کد (متُد) باشد.

الگوی کلی تعریفِ کلاس به شکل زیر است:

```
class Classname
{
   type instance-variable1;
   type instance-variable2;
   // ...
   type instance-variableN;

   type methodname1(parameter-list)
   {
      // body of method
   }
   type methodname2(parameter-list)
   {
      // body of method
   }
   // ...
   type methodnameN(parameter-list)
   {
      // body of method
   }
}
```

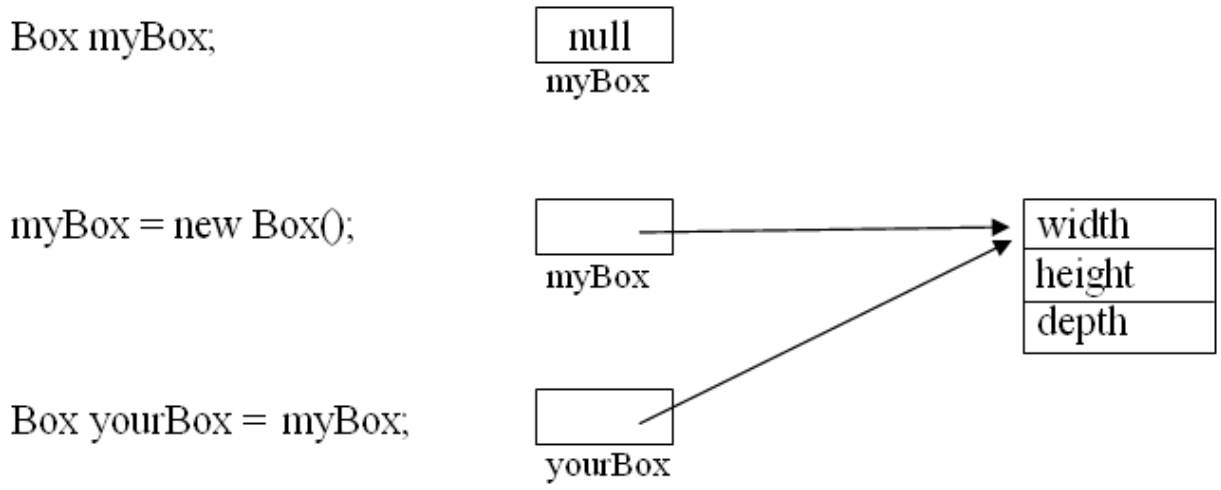
مثال: کلاس زیر ، شئ Box را شبیه سازی می کند.

```
class Box
{
   double width;
   double height;
   double depth;

   double volume()
   {
       return width * height * depth;
   }
}
```

برای ایجاد اشیاء ، مانند ایجاد یک متغیر ، از نوع و نام استفاده می شود. پس از انتخاب نوع و نام شئ ، باید نمونه ای از آن شئ را بسازیم بعبارت دیگر باید حافظه ای به آن شئ اختصاص دهیم. نام هر شئ ، محلی از حافظه کامپیوتر بوده و یک ارجاع به حافظه است. وقتی شیئی از یک کلاس new می شود فقط برای متغیر های آن فضا گرفته می شود.

مثال:

```
class Example1
{
   public static void main(String[] args)
   {
       Box myBox;
       myBox = new Box();
       myBox.width = 5;
       myBox.height = 7;
       myBox.depth = 4;
       System.out.println("Volume = "+ myBox.volume());
   }
}
```



* - Reference ها در جاوا چهار بایت فضا اشغال می کنند.

** - وقتی یک Reference ، null است نمی توانیم از اعضای آن استفاده کنیم در غیر این صورت NullPointerException رخ می دهد (Runtime Error).

کلاس های تو در تو Nested and Inner Classes

می توان کلاسی را داخل کلاس دیگر تعریف کرد که به چنین کلاسهایی کلاسهای تو در تو (Nested Classes) گفته می شود. حوزه اعتبار چنین کلاسهایی داخل کلاسهایی است که آنها را در میان گرفته اند. بنابراین اگر کلاس B در داخل کلاس A تعریف شده باشد ، B در داخل A قابل استفاده و معتبر است و خارج از آن قابل Instantiate نیست. کلاس Nested به تمامی اعضای (از جمله private) کلاسی که در داخل آن تعریف شده است دسترسی دارد ولی عکس آن صحیح نیست یعنی کلاسی که Nested Class را در میان گرفته است به اعضای کلاس Nested دسترسی ندارد.

کلاسهای Nested به دو صورت static و غیر static تعریف می شوند کلاس Nested ی که در تعریف آن ، قبل از کلمه class از واژه کلیدی static استفاده شده باشد بصورت static تعریف شده است این کلاسها مانند کلاسهای غیر static به اعضای کلاسی که در داخل آن تعریف شده اند دسترسی ندارند و بدلیل این محدودیت بندرت استفاده می شوند.

نوع غیر sta tic کلاسهای Nested ، مهم هستند که به آنها Inner Class گفته می شود. Inner Class بدون محدودیت به تمامی اعضای کلاسی که در داخل آن تعریف شده است دسترسی مستقیم دارد.

مثال:

```
public class Outer
{
   int outer_x = 100;

   void test()
   {
      MyInner myInner = new MyInner();
      myInner.display();
   }

   // this is an inner class
   class MyInner
   {
      int y = 10;

      void display()
      {
         System.out.println("display: outer_x = " + outer_x);
      }
   }

   void showy()
   {
//       System.out.println(y);    // error, y not known here!
   }
}

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

همانطور که در مثال نیز مشاهده می شود متُد display از کلاس MyInner به فیلد outer_x که Instance Variable کلاس Outer است دسترسی دارد اما متُد showy از کلاس Outer به فیلد y که Instance Variable کلاس MyInner است دسترسی ندارد.

Inner Class را می توان داخل یک بلوک ، حتی بلوک حلقه تکرار نیز تعریف کرد که حوزه اعتبار آن فقط همان بلوک خواهد بود. همچنین Inner Class را می توان بدون نام (Anonymous Inner Class) نیز تعریف کرد.

Inner Class ها در برنامه نویسی روزمره کاربرد کمی دارند اما در مواردی مانند Event Handling و Control Framework کاربرد های اساسی و مهمی دارند.

Nested Class ها از Java 1.1 به بعد اضافه شده اند.

سازنده ها Constructors

جاوا اجازه می دهد که اشیاء ، هنگام ایجاد ، خود را مقداردهی کنند. این کار با استفاده از متُدی انجام می شود که سازنده نامیده می شود. سازنده ، همنام کلاس بوده و هیچ مقداری را بر نمی گرداند (حتی void). متُد سازنده بلافاصله پس از ایجاد شئ بطور خودکار اجرا می شود. متُد های سازنده می توانند دارای پارامتر نیز باشند که در این صورت هنگام ایجاد شئ بوسیله new ، لازم است پارامترها را به سازنده پاس کنیم.

متُد های سازنده برای مقدار اولیه دادن به صفات (متغیر های) شئ بکار می روند. هر کلاسی دارای یک سازنده پیش فرض است و هنگام ایجاد شئ ، اگر متغیر های آن شئ مقداردهی اولیه شده باشند آن مقادیر را به متغیر ها نسبت خواهد داد و اگر متغیر یا متغیر هایی مقداردهی اولیه نشده باشند به شکل زیر عمل خواهد کرد:

- فیلد های عددی را برابر صفر قرار می دهد.
- فیلد های کاراکتری را برابر یونیکد "\u0000" قرار می دهد.
- فیلد های بولین را برابر false قرار می دهد.
- فیلد های نوع شئ را برابر null قرار می دهد.

مثال:

```
class Box
{
   double width;
   double height;
   double depth;

   Box(double  width, double height, double depth)
   {
       this.width = width;
       this.height = height;
       this.depth = depth;
   }

   double volume()
   {
       return width * height * depth;
   }
```

}

کلمه کلیدی this this keyword

همانطور که می دانید برای دسترسی به فیلدِ یک شئ ، پس از ذکر نام شئ و یک نقطه نام آن فیلد را می نویسیم. در مثال فوق نیز به شکل myBox.width عمل کردیم ، اما در مواقعی لازم است که متُد به این صفات از داخل خود شئ دسترسی پیدا کند در حالی که متغیری محلی ، همنام با آن در داخل متُد تعریف شده است ، برای تفکیک آن دو ، Instance variable را با کلمه کلیدی this و یک نقطه آغاز می کنیم مانند :

this.width

اصل محلیت می گوید که در صورت تشابه اسمی ، آن متغیر یا متُدی را که نزدیک تر است را انتخاب کن.

کلمه کلیدی this کاربرد دیگری هم دارد که در بحث Method Overloading مطرح خواهد شد.

مدیریت خودکارِ حافظه Garbage Collection

هر شئ که ایجاد می شود مقداری از حافظه را به صورت دینامیکی اشغال می کند و لازم است پس از پایان کار آن شئ ، حافظه اخذ شده به سیستم عامل بازگردانده شود. برای انجام این کار در زبانهای برنامه نویسی مانند C++ ، باید دستورات لازم بوسیله برنامه نویس در محل مناسب نوشته شود اما در جاوا این کار بطور خودکار بوسیله تکنیکی به نام *Garbage Collection* انجام می شود بدین ترتیب که وقتی هیچ ارجاعی به یک شئ وجود نداشت Garbage Collector آن را به عنوان یک شئ غیر لازم فرض کرده و حافظه مربوطه را آزاد می کند. البته JVM های مختلف روشهای متنوعی را برای انجام این کار انجام می دهند. در هر صورت برنامه نویس زبان جاوا نباید نگران آزاد شدن حافظه های اشغال شده باشد. همچنین می توان Garbage Collector را با استفاده از متُد ()gc از کلاس System به صورت دستی اجرا کرد. System.gc();

متُد ()finalize Finalization

در زبانهای شئ گرا ، آزاد کردن حافظه اشغال شده بوسیله شئ را اصطلاحاً *destroy* کردن (یا خراب کردن) شئ می گویند. برخی مواقع لازم است قبل از خراب شدن شئ ، یک یا چند کار انجام شود برای این منظور می توان از متُد *finalize* استفاده کرد. الگوی کلی این متُد به شکل زیر است:

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

Garbage Collector قبل از خراب کردن شئ ، متُد مذکور را فراخوانی خواهد کرد. البته این بدان معنی نیست که هر وقت شئ از حوزه تعریف شده اش خارج شد بلافاصله متُد مذکور اجرا می شود.

آرایه ها Arrays

برای تعریف آرایه می توان به یکی از دو روش زیر عمل کرد:

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

با نوشتن یکی از دو دستور فوق ، فقط نوع و نام متغیر آرایه را مانند نوع و نام متغیر شئ تعریف کرده ایم و برای اختصاص حافظه لازم است مانند ایجاد اشیاء ، از کلمه new استفاده کنیم.

array-var = new type[size];

مثال:

```
int[] monthDays;
monthDays = new int[ 12];
```

همچنین می توان آرایه را در زمان تعریف مقداردهی اولیه کرد که در این صورت عمل new کردن به طور خودکار انجام می شود مانند:

```
int[] monthDays = {31, 31, 31, 31, 31, 31,30, 30, 30, 30, 30,29};
```

مقداردهی اولیه به شکل زیر نیز انجام می شود:

```
int[] monthDays;
monthDays = new int[]  {31, 31, 31, 31, 31, 31,30, 30, 30, 30, 30,29};
```

برای تعریف آرایه های دو یا چند بعدی می توان مانند زیر عمل کرد.

مثال:

```
int[][] twoDim = new int[4][5];
```

یا

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

مقداردهی اولیه آرایه فوق که new کردن آن بطور خودکار انجام خواهد گرفت به شکل زیر است:

```
int[][] twoDim =
{
   {2,3,8,5,1},
   {8,6,7,9,0},
   {3,0,5,6,7},
   {9,3,4,2,1}
};
```

در مثال زیر آرایه ای دو بعدی با طول سطر های متفاوت تعریف شده است:

```
int myArr[][] =
{
   {2},
   {8,6},
   {3,0,5},
   {9,3,4,2}
};
```

چند نکته

۱- با توجه به اینکه تعریف آرایه و اختصاص فضای حافظه به آن در مراحل جداگانه ای انجام شود می توان اندازه آرایه را با توجه به میزان مورد نیاز در حین اجرای برنامه مشخص کرد.

۲- هر آرایه ای که ایجاد می شود ، بطور خودکار یک فیلد به نام ***length*** به آن اختصاص می یابد که اندازه آرایه را نگهداری می کند ، همانطور که بعداً توضیح داده می شود طول String ها بوسیله متُدی به نام ***length()*** بدست می آید.

۳- در برنامه نویسی شئ گرا به سه روش می توان از آرایه استفاده کرد:

الف- تعریف در داخل متُد

ب – تعریف در خارج از متُد و داخل کلاس

ج – تعریف آرایه ای از اشیاء

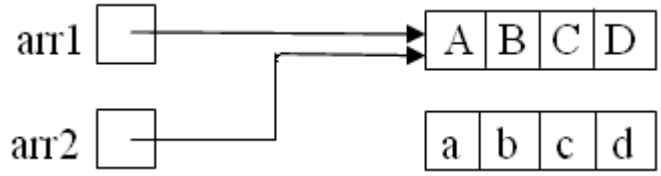
۴- در صورتی که آرایه هنگام ایجاد ، مقداردهی نشود عناصر آن صفر یا null خواهند بود.

۵- در مورد انتصاب آرایه ها اگر دو آرایه زیر را در نظر بگیریم :

```
char[] arr1 = {65,66,67,68};
char[] arr2 = {97,98,99,100};
```

با توجه به اینکه اختصاص حافظه به آرایه ها مانند اشیاء صورت می گیرد قبل از انتصاب دو آرایه نمایش آنها در حافظه به شکل زیر می باشد:

بعد از دستور انتصاب arr2 = arr1; خواهیم داشت:



بنابراین پس از اجرای دستور انتصاب ، arr1 و arr2 به یک محل از حافظه اشاره خواهند کرد. محلی را که arr2 قبلاً به آن اشاره می کرده به عنوان حافظه مازاد تلقی شده و بوسیله Garbage Collector آزاد می شود. برای کپی کردن آرایه ها می توان از متُد arraycopy() که در کلاس System تعریف شده است استفاده کرد:

```
System.arraycopy(sourceArray,sourcePosition,desArray,desPosition,count);
```

۶- وقتی آرایه ای از اشیاء ایجاد می شود ، به طور خودکار هیچ شیئی در آن قرار نمی گیرد ، بلکه تمام عناصر آن برابر null می شود. در واقع آرایه ای از ارجاعات بوجود می آید که در ابتدا ، به هیچ شیئی اشاره نمی کنند. مثال:

```
Box[] myObj;
myObj = new Box[10];
```

در دستور اول ، آرایه ای به نام myObj اعلان می شود و در دستور دوم حافظه ای برای اشاره به ده شئ از نوع کلاس Box اختصاص می یابد و چون مقداری در آرایه قرار نگرفته هر ده ارجاع ، null است.

۷- کلاس Array از پکیج java.util ، حاوی متُد هایی است که اکثر عملیات متُداول بر روی آرایه ها را انجام می دهند.

۸- در جاوا Pointer ها ، مانند C/C++ وجود ندارند ، چرا که در صورت وجود Pointer ها ، برنامه های Java applet ، خواهند توانست در Firewall بین محیط اجرایی جاوا و کامپیوتر میزبان نفوذ کنند. همچنین Pointer می تواند حاوی هر آدرسی در حافظه ، از جمله خارج از سیستم JVM باشد.

کلاس String — String Class

در جاوا ، رشته مانند C/C++ یک نوع اولیه یا آرایه ای از کاراکتر ها نیست بلکه String ، شئ رشته را اعلان می کند و حاوی تعداد زیادی متُد برای انجام عملیات مختلف بر روی رشته هاست. کلاس String در پکیج java.lang تعریف شده است.

مثال:

```
String str1 = new String() ;
String str2 = new String(" Test String");
String str3 = "Java programming";          //تخصیص خودکار حافظه

String str4 ,str5;
str4 = new String("Test sentence");
str5 = "Ali";                              //تخصیص خودکار حافظه
```

نکته

رشته ها تغییر ناپذیرند(Immutable) و وقتی مقدار جدیدی به آنها نسبت داده می شود محل جدیدی از حافظه به آنها اختصاص می یابد. عملگر + در مورد رشته ها ، به عنوان عملگر الحاق عمل می کند.

بررسی کلاس StringBuffer که دارای قابلیت های دیگری می باشد به دانشجویان واگذار می شود.

پکیج package

پکیج ابزاری برای دسته بندی کلاسها و واسط ها(interface)ی مرتبط به هم است در واقع پکیج کتابخانه ای از کلاسهایی است که به نحوی با یکدیگر در ارتباطند. همچنین مفهوم Encapsulation را در سطح کلاسها پوشش می دهد.

جاوا هزاران کلاس را تدارک دیده است که تعداد اندکی از آنها ، مثل کلاسهایی که در پکیج java.lang قرار دارند ، بطور خودکار در هر برنامه ای قابل استفاده اند (مانند کلاس System). اما سایر کلاسها باید با استفاده از دستور import در ابتدای برنامه ، ضمیمه شوند تا قابل استفاده باشند. مثال:

```
import java.util.Date;
import javax.swing.*;
```

دستور اول کلاس Date را از پکیج java.util و دستور دوم همه کلاسهای موجود در پکیج javax.swing را به برنامه ضمیمه می کند. البته می توان بدون استفاده از دستور import و با ذکر مسیر کاملِ کلاس مورد نظر ، از آن استفاده کرد مانند:

```
java.util.Date myDate = new java.util.Date();
```

که در این صورت لازم است در هر بار استفاده از آن کلاس ، مسیر کاملِ آن را ذکر کرد.

*- استفاده از ستاره ممکن است زمان کامپایل را افزایش دهد بخصوص اگر چندین پکیج بزرگ ضمیمه شده باشند به همین دلیل بهتر است بطور صریح نام کلاس هایی که مورد نیاز است ذکر شود ، اما استفاده از ستاره هیچ تأثیری در عملکرد حین اجرا یا اندازه کلاس ها نخواهد داشت.

همچنین هر پکیج می تواند حاوی پکیج های دیگری باشد که سطح دیگری از سلسله مراتب کلاسها را بوجود می آورد.

برای قراردادن کلاسی در یک پکیج ، لازم است به عنوان اولین دستور و قبل از تعریف کلاس از دستور package استفاده شود الگوی کلی آن به شکل زیر است :

package packageName;

مثال:

```
package myPackage;
```

کلاسهایی که دارای دستور پکیج مشابه هستند بعبارت دیگر دارای نام پکیج یکسان هستند در یک پکیج با آن نام قرار می گیرند. نام پکیج با حرف کوچک آغاز می شود. جاوا برای نگهداری پکیج از سیستم دایرکتوریِ سیستم عامل استفاده می کند لذا کلاسهای هم پکیج در یک دایرکتوری به نام پکیج نگهداری می شوند.

برای ایجاد سلسله مراتبی از پکیج ها (پکیج هایِ تودرتو) ، نام هر پکیج را با استفاده از یک نقطه از پکیج بالایی خود جدا می کنیم و الگوی کلی آن به شکل زیر است:

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

مثال:

package myPackage.io.file;

که در این مثال کلاسهای مربوط به پکیج file در دایرکتوری با ساختار myPackage\io\file ذخیره خواهند شد.

فایل های Source Code مربوط به کلاس ها را می توان در یک دایرکتوری و یا در دایرکتوری هایی مانند ساختار پکیج ها نگهداری کرد ولی بایت کد های تولید شده (فایل های با پسوند class) لزوماً در ساختار دایرکتوری توضیح داده شده قرار خواهند گرفت.

برای کامپایل و اجرا ، می توان به روش زیر عمل کرد:

javac –classpath *CompiledClassesPath* –d *DestinationPath ClassName.java*
java –cp *CompiledClassesPath pkg1[.pkg2[.pkg3[...]]].ClassName*

متُد های همنام Method Overloading

در جاوا می توان در یک کلاس متُد هایی تعریف کرد که نامشان یکسان بوده ولی تعداد و یا نوع پارامترهای آنها متفاوت باشد. در این حالت می گوییم که متُد ها Overload شده اند متُد های همنام به دو طریق از یکدیگر تمیز داده می شوند : تعداد پارامترها و نوع هریک از پارامترها. این دو را امضای متُد گویند. بنابراین امضای متُد های همنام باید متفاوت باشند. متفاوت بودن نوع داده های برگشتی متُد ها برای Overloading کافی نبوده و نقشی در تجزیه و تحلیل متُد های Overload شده ندارند. وقتی جاوا با فراخوانی یک متُد Overload شده مواجه می شود ، نسخه ای از متُد را اجرا می کند که پارامترهای آن با آرگومان های استفاده شده در فراخوانی مطابقت داشته باشند. مثال:

```
class OverloadDemo
{
  void test()
  {
     System.out.println("No parameters");
  }
  void test(int a)
  {
     System.out.println("a: " + a);
  }
  void test(int a, int b)
  {
     System.out.println("a and b: " + a + " " + b);
  }
  double test(double a)
  {
     System.out.println("double a: " + a);
     return  a * a;
  }
}
```

```
class Overload
{
  public static void main(String args[])
  {
     OverloadDemo ob = new OverloadDemo();
     double result;
     ob.test();
     ob.test(10);
     ob.test(10, 20);
     result = ob.test(123.25);
     System.out.println("Result of ob.test( 123.25): " + result);
  }
}
```

لازم به ذکر است که در برخی موارد ، جاوا از تبدیل نوع خودکار (Automatic Type Casting) در Overload کردن متُد ها استفاده می کند. مثلاً اگر متُدی دارای یک پارامتر double باشد وبا یک مقدار int فراخوانی شود و متُدی همنام با آن Overload نشده باشد ، جاوا بطور خودکار نوع int را به double تبدیل کرده و آن را فراخوانی می کند.

علاوه بر Overload کردن متُد های معمولی ، متُد های سازنده را نیز می توان Overload کرد. یعنی یک کلاس می تواند دارای چندین سازنده با امضا های متفاوت باشد که باعث انعطاف پذیری هرچه بیشتر کلاس می شود. همچنین سازنده ها می توانند همدیگر را فراخوانی کنند این کار با استفاده از کلمه کلیدی this انجام می شود. وقتی this به عنوان نام یک متُد مورد استفاده قرار می گیرد و به دنبال آن نام پارامترها می آیند بدین معنی است که یکی از سازنده های دیگر فراخوانی می شود.کامپایلر با توجه به امضای بکار رفته ، سازنده مورد نظر را شناسایی می کند. البته این روش فراخوانی سازنده ، صرفاً در داخل سازنده مجاز است و باید به عنوان اولین دستورِ سازنده فراخوان باشد. مثال:

```
class Box
{
  int width;
  int height;
  int depth;
  int Color;

  Box(int w,int h,int d)
  {
      width = w;
      height = h;
      depth = d;
      Color = 1;
  }

  Box(int w,int h,int d,int c)
  {
      this(w,h,d);
      Color = c;
  }
```

```
  int volume()
  {
      return width * height * depth;
  }
}
```

Inheritance وراثت

وراثت یکی از اصول مهم برنامه نویسی شئ گراست زیرا امکان ایجاد طبقه بندی های سلسله مراتبی را بوجود می آورد. در جاوا ، کلاسی را که از آن ارث برده می شود ، superclass و کلاسی که عمل ارث بری را انجام داده و ارث می برد ، subclass می نامند . بنابراین ، subclass نسخه تخصصی تر از یک superclass است . subclass ، همه فیلد ها و متُد های تعریف شده در superclass را به ارث برده و موارد خاص خود را نیز به آن اضافه می کند.

برای ارث بردن از یک کلاس ، کافی است در تعریف subclass پس از نوشتن نام کلاس جدید ، کلمه کلیدی extends و سپس نام کلاسی(superclass) را که می خواهیم کلاس جدید از آن ارث ببرد را ذکر کنیم.

الگوی کلی :

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

در مثال زیر دو کلاس A و B تعریف شده است کلاس A ، superclass و کلاس B ، subclass می باشد بعبارت دیگر کلاس B ویژگیهای کلاس A را به ارث می برد.

```
class A
{
  int i, j;

  void showij()
  {
     System.out.println("i and j: " + i + " " + j);
  }
}

class B extends A
{
  int k;

  void showk()
  {
     System.out.println("k: " + k);
  }

  void sum()
  {
```

```
    System.out.println("i+j+k: " + (i+j+k));
  }
}
```

همانطوری که از مثال نیز مشخص است کلاس B شامل اعضای کلاس A ، یعنی superclass است بنابراین ، نمونه ساخته شده از کلاس B شامل فیلد های i و j و متُد ()showij بوده و می تواند متُد ()showij را فراخوانی کند.

لازم به ذکر است که superclass بودن برای یک subclass ، به این معنی نیست که superclass نمی تواند بطور مستقل بکار رود لذا می توان از کلاس A بطور مستقل نمونه(Instance) ایجاد کرد. بعلاوه یک subclass می تواند یک superclass برای subclass دیگر باشد.

هر کلاسی صرفاً از یک کلاس می تواند ارث ببرد و جاوا مانند C++ از ارث بری چندگانه پشتیبانی نمی کند. ولی چندین کلاس می توانند از یک کلاس ارث ببرند همچنین هیچ کلاسی نمی تواند superclass خودش باشد.

*- یک متغیر از نوع superclass می تواند به یک شئ از نوع subclass مراجعه کند بعبارت دیگر می توان آدرسِ شئ از نوع subclass را در متغیری از نوع superclass خودش قرار داده و بکار برد که در این صورت توجه به نکات زیر ضروری است:

الف – اعضایی (متغیر و متُد) از شئ که در subclass اضافه شده اند قابل دسترس نخواهند بود چون متغیر از نوع superclass دانش لازم برای دسترسی به آنها را ندارد.

ب – اگر متُدی از superclass که در subclass ، Override شده باشد فراخوانی شود نسخه مربوط به subclass (شئ) اجرا خواهد شد. (Dynamic Method Dispatch)

**- در بالای سلسله مراتب کلاس جاوا ، کلاس Object قرار دارد ؛ یعنی اولین superclass همه کلاسهای جاوا کلاس Object است پس هر کلاسی که در جاوا به صورت superclass تعریف می شود به صورت تلویحی (غیر صریح) از کلاس Object ارث بری می کند. اعضای این کلاس در قسمتهای بعدی توضیح داده خواهند شد.

لغو متُد Method Overriding

وقتی که یک متُد در subclass ، دارای نام و امضای یکسان با متُدی در superclass خودش باشد در این حالت می گویند که متُد همنام خود در superclass را Override یا لغو کرده است. Override کردن یک متُد از پیشروی آن در نسل های بعدی این کلاس جلوگیری می کند. وقتی یک متُد Override شده از subclass فراخوانی می شود متُدی که در subclass تعریف شده اجرا شده و متُد superclass پنهان خواهد شد. مثال:

```
class A
{
  int i, j;

  A(int a, int b)
  {
    i = a;
    j = b;
  }
```

```
  void show()
  {
     System.out.println("i and j: " + i + " " + j);
  }
}

class B extends A
{
  int k;

  B(int a, int b, int c)
  {
     i = a;
     j = b;
     k = c;
  }

  void show()
  {
  //  super.show(); // this calls A's 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
  }
}
```

لغو متُد ، فقط زمانی اتفاق می افتد که اسامی و امضای دو متُد یکسان باشند. اگر چنین نباشد ، آنگاه دو متُد Overload خواهند شد.

کلمه کلیدی super **super keyword**

این کلمه ، کاربردی مشابه کلمه کلیدی this دارد با این تفاوت که کلمه super ، همان نقش را در مورد superclass بازی می کند. بنابر این وقتی super به عنوان نام یک متُد مورد استفاده قرار می گیرد و به دنبال آن نام پارامترها می آیند بدین معنی است که یکی از سازنده های superclass آن کلاس فراخوانی می شود.کامپایلر با توجه به امضای بکار رفته ، سازنده مورد نظر را شناسایی می کند. البته این روش فراخوانی سازنده superclass ، صرفاً در داخل سازنده مجاز است و باید به عنوان اولین دستورِ سازنده فراخوان باشد.

وقتی شیئی از نوع subclass ایجاد می گردد ، ابتدا متُد سازنده superclass و سپس متُد سازنده subclass اجرا می شود. بنابراین اگر متُد سازنده superclass دارای پارامتر باشد باید این پارامترها بوسیله متُد سازنده subclass در اختیار متُد سازنده superclass قرار گیرد. حتی اگر در subclass نیازی به استفاده از متُد سازنده نباشد باید سازنده ای در subclass نوشته شود و پارامترهای موردنیاز متُد سازنده superclass بوسیله کلمه super در اختیارش قرار گیرند. البته اگر superclass دارای چند سازنده باشد و در subclass مشخص نشده باشد که کدام سازنده superclass اجرا شود ، سازنده پیش فرض (در صورت وجود) ، که همان سازنده بدون آرگومان است اجرا خواهد شد.

اگر در کلاسی ، سازنده بوسیله برنامه نویس نوشته شود دیگر جاوا ، سازنده پیش فرض را ایجاد نمی کند. پس اگر سازنده های نوشته شده بوسیله برنامه نویس ، آرگومان ورودی داشته باشند حتماً در زمان new کردن باید آرگومان های لازم را به آن پاس کنیم چون دیگر متُد سازنده بدون آرگومان نداریم.

متُد سازنده subclass ، همیشه سازنده پیش فرض (سازنده بدون آرگومان) superclass را call می کند و اگر سازنده بدون آرگومان نداشته باشیم کلاسِ فرزند ، قادر به فراخوانی سازنده کلاس پدر نخواهد شد و Error رخ خواهد داد. پس باید سازنده superclass (سازنده دارای آرگومان) را خودمان call کنیم.

مثال:

```
class Box
{
  private double width;
  private double height;
  private double depth;

  Box(Box ob)  // construct clone of an object with passing object to constructor
  {
     width = ob.width;
     height = ob.height;
     depth = ob.depth;
  }

  Box(double w, double h, double d)
  {
     width = w;
     height = h;
     depth = d;
  }

  Box()   // use -1 to indicate an uninitialized box
  {
     this(1,-1,-1);
  }

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

```
  double volume()
  {
     return width * height * depth;
  }
}

class BoxWeight extends Box
{
  double weight;  // weight of box

  BoxWeight(BoxWeight ob)
  {
     super(ob);  // call superclass constructor
     weight = ob.weight;
  }

 BoxWeight(double w, double h, double d, double m)
 {
     super(w, h, d);
     weight = m;
  }

  BoxWeight()    // default constructor
  {
     super();
     weight = -1;
  }

  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);

     System.out.println("Volume of mybox1 is " + mybox1.volume());
     System.out.println("Weight of mybox1 is " + mybox1.weight);
  }
```

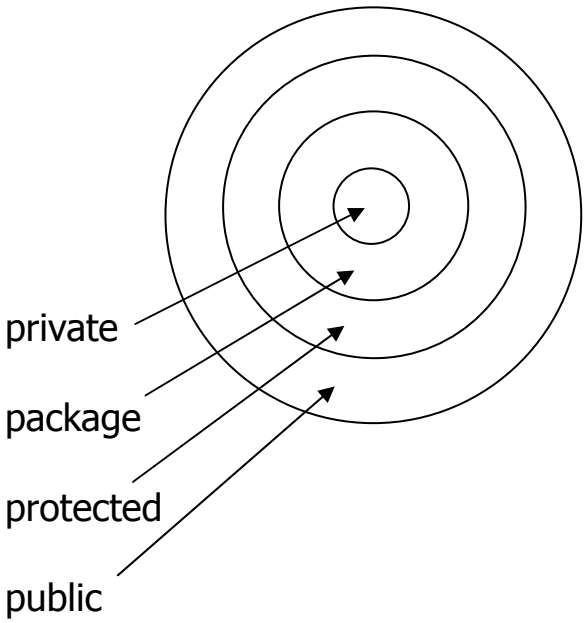
```
}
```

حُسْنِ استفاده از سازنده superclass در این است که اگر بعداً در کُد superclass تغییری ایجاد شد نیاز به تغییر subclass ها نخواهد بود.

کاربرد دیگر کلمه super برای دسترسی به اعضای پنهان شده superclass می باشد. متُد ها و فیلد های Override شده با قرار دادن کلمه super و یک نقطه قبل از نام آنها در subclass در دسترس قرار می گیرند. مانند مثال ارایه شده در بخش Method Overriding (متُد ()super.show). در واقع واژه super ، ابزار دسترسی به صفات و متُد های superclass است.

Access Protection محافظت دسترسی

کلاسها و پکیج ها ابزارهایی برای Encapsulation بوده و در برگیرنده نام و حوزه متغیر ها و متُد ها می باشند. پکیج ها به عنوان ظروفی برای کلاس ها و سایر پکیج های وابسته هستند. کلاس ها نیز به عنوان ظروفی برای داده ها و کد ها می باشند. کلاس ، کوچکترین واحد مجرد در جاوا است. کنترل یا محافظت دسترسی یکی دیگر از ویژگیهای بسیار مهم Encapsulation می باشد به لحاظ نقش متقابل بین کلاس ها و پکیج ها ، چهار طبقه بندی برای دسترسی به اعضای کلاس در جاوا مشخص شده است.



سه مشخصگر(**Modifier**) دسترسی ، یعنی public ، private و protected طیف گوناگونی از شیوه های تولید سطوح چندگانه دسترسی مورد نیازِ این طبقه بندی ها را فراهم می کنند. برخی جوانب کنترل دسترسی بشدت با وراثت و پکیج ها مرتبط اند. جدول زیر ، این ارتباطات را به صورت یکجا نشان می دهد:

	private	No modifier	Protected	Public
Same class	✓	✓	✓	✓
Same package, Subclass	✘	✓	✓	✓
Same package, Non-Subclass	✘	✓	✓	✓
Different package, Subclass	✘	✘	✓	✓
Different package, Non-Subclass	✘	✘	✘	✓

ممکن است مکانیسم کنترل دسترسی در جاوا پیچیده بنظر برسد ، اما می توان آن را بصورت ساده زیر بیان کرد:

هر چیزی که به عنوان public اعلان شود از هر جایی قابل دسترس است. هر چیزی که به عنوان private اعلان شود خارج از کلاس خودش قابل دسترس نیست. بعبارت دیگر عضوی که private تعریف می شود مانند چیزی است که در فکر شخص وجود دارد و از بیرون مشخص و قابل دسترس نیست ولی عضوی که public تعریف می شود مانند صحبت کردن شخص است که از بیرون مشخص است.

وقتی یک عضو فاقد مشخصات دسترسی باشد (No modifier) ، از سطح دسترسی package استفاده می کند (دسترسی پیش فرض). یعنی خود کلاس ، subclass ها و سایر کلاس های موجود در همان پکیج به آن عضو دسترسی خواهند داشت.

اگر لازم است یک عضو ، خارج از پکیج جاری و فقط برای کلاس هایی که مستقیماً از آن کلاس بصورت subclass درآمده اند قابل دسترس باشد ، آن عضو به عنوان protected اعلان می شود.

*- در Override کردن یک متُد می توان سطح دسترسی آن را افزایش داد مثلاً می توان سطح دسترسی متُدی را که در superclass ، protected بوده هنگام Override کردن به public تغییر داد و کلاً حالات زیر امکان پذیر است:

package to protected
package to public
protected to public

**- مطالب ذکر شده در مورد اعضای داخل کلاس می باشد و خود کلاس فقط دو سطح دسترسی دارد: پیش فرض و public. وقتی یک کلاس به عنوان public اعلان می شود ، بوسیله هر کد دیگری قابل دسترس است و اگر دسترسی پیش فرض (No modifier) داشته باشد ، فقط بوسیله کد های داخل همان پکیج قابل دسترس خواهد بود.

مثال زیر کلیه ترکیبات مربوط به Modifier های کنترلِ دسترسی را نشان می دهد. این مثال حاوی دو پکیج و پنج کلاس است. پکیج p1 کلاس Protection ، کلاس Derived که از کلاس Protection مشتق شده و کلاس SamePackage را تعریف می کند و پکیج p2 کلاس Protection2 را که از p1.Protection مشتق شده و کلاس OtherPackage را تعریف می کند.

```
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);
  }
}
```

```
package p1;
class Derived extends Protection
{
  Derived()
  {
     System.out.println("derived constructor");
     System.out.println("n = " + n);
  //  System.out.println("n_pri = " + n_pri);    //  class only
     System.out.println("n_pro = " + n_pro);
     System.out.println("n_pub = " + n_pub);
  }
}
```

```
package p1;
class SamePackage
{
  SamePackage()
  {
     Protection p = new Protection();
     System.out.println("same package constructor");
     System.out.println("n = " + p.n);
  //  System.out.println("n_pri = " + p.n_pri);     //  class only
     System.out.println("n_pro = " + p.n_pro);
     System.out.println("n_pub = " + p.n_pub);
  }
}
```

```
package p2;
class Protection2 extends p1.Protection
{
  Protection2()
  {
     System.out.println("derived other package constructor");
  //  System.out.println("n = " + n);    //  class or package only
  //  System.out.println("n_pri = " + n_pri);  //  class only
     System.out.println("n_pro = " + n_pro);
     System.out.println("n_pub = " + n_pub);
  }
}
```

```
package p2;
class OtherPackage
{
  OtherPackage()
  {
     p1.Protection p = new p1.Protection();
     System.out.println("Other Package constructor");
  //  System.out.println("n = " + p.n);   //  class or package only
```

```
  //  System.out.println("n_pri = " + p.n_pri);    //  class only
  //  System.out.println("n_pro = " + p.n_pro);    // class, subclass or package only
      System.out.println("n_pub = " + p.n_pub);
  }
}
```

برای آزمایش دو پکیج نوشته شده کلاسهای زیر را می توان بکار برد:

```
// Demo package p1.
package p1;
public class Demo
{
  public static void main(String args[])
  {
     Protection ob1 = new Protection();
     Derived ob2 = new Derived();
     SamePackage ob3 = new SamePackage();
  }
}
```

```
// Demo package p2.
package p2;
public class Demo
{
  public static void main(String args[])
  {
     Protection2 ob1 = new Protection2();
     OtherPackage ob2 = new OtherPackage();
  }
}
```

متُد های دستیاب و مقدار دهنده getter and setter Methods

متُد هایی که برای دستیابی به فیلد های private یک کلاس نوشته می شوند متُد های دستیاب (getter) ، و متُد هایی که برای تغییر این فیلد ها نوشته می شوند متُد های مقدار دهنده (setter) نامیده می شوند. از آنجائیکه حفظ امنیت و برقراری حالت اختصاصی ، در برنامه نویسی شئ گرا از اهمیت ویژه ای برخوردار است توصیه می شود که حتی الامکان از تعریف فیلد های public خودداری کرده و برای دسترسی به آنها از متُد های getter و setter استفاده شود همچنین بهتر است داخل خود کلاس نیز برای دسترسی به فیلد ها از این متُد ها استفاده شود.

برخی از متُد های مهم را نیز در صورت لزوم می توان بصورت private تعریف کرده و آنها را از طریق متُد های public دیگر پس از تنظیمات لازم فراخوانی کرد. رعایت موارد فوق نه تنها کلاس را از دستیابی های غیرمجاز محافظت می کند ، بلکه جزئیات کلاس را نیز پنهان می کند این کار را *پنهان سازی اطلاعات* (Information Hiding) می گویند.

یکی از وظایف اولیه هر شئ این است که جامعیت خود را حفظ کند ، یعنی اطمینان حاصل کند که تمام فیلد هایش دارای مقادیر معتبری هستند که ترکیب آنها وضعیت پایداری از شئ را نشان می دهد. هر شئ می تواند در مقابل درخواستی که جامعیت آن را نقض می کند مقاومت کند. بخش مهمی از این وظیفه را می توان بوسیله متُد های getter و setter انجام داد.

حل تمرین

۱- برنامه ای بنویسید که جدول ضربی را با توجه به بُعد داده شده چاپ کند.(بین ۲ تا ۱۰)

```
public class MultiplicationTable
{
   private int side;
   public MultiplicationTable(int side)
   {
      if( ide > 10)
        this.side = 10;
      else if(side < 2)
        this.side = 2;
      else
        this.side = side;
   }

   public MultiplicationTable()
   {
       this(5);
   }

   public int getSide()
   {
      return side;
   }

   public void printMultiplicationTable()
   {
      int i ,j;
      for(i = 1;i <= side;i++)
      {
         for(j = 1;j <= side;j++)
            System.out.print((i*j)+" ");
         System.out.println();
      }
   }
}
```

۲- برنامه ای بنویسید که دارای یک Stack بوده و آن Stack حداکثر n عدد صحیح را در خود نگهداری کند و اعمالی مانند push ، pop و top بر روی آن قابل انجام باشد. همچنین تعداد عناصر داخل آن قابل دسترس باشد.

```
public class Stack
{
   private int[] stackArr;
   private int stackPointer;
   public Stack(int sSize)
   {
      if(sSize < 100)
```

```
        sSize = 100;
    if(sSize > 1000)
        sSize = 1000;
    stackArr = new int[sSize];
    stackPointer = -1;
}

public Stack()
{
    this(200);
}

public int getStackSize()
{
    return stackArr.length;
}

public boolean push(int value)
{
    if(stackPointer == stackArr.length-1)
        return false;
    stackArr[++stackPointer] = value;
    return true;
}

public int pop()
{
    if(stackPointer == -1)
    {
        System.err.println("Stack is empty");
        return 0;
    }
    return stackArr[stackPointer--];
}

public int top()
{
    if(stackPointer == -1)
    {
        System.err.println("Stack is empty");
        return 0;
    }
    return stackArr[stackPointer];
}

public int getStackPointer()
```

```
    {
        return stackPointer + 1;
    }
}
```

برای آزمایش دو کلاس فوق می توان مانند زیر عمل کرد:

```
public class Examine
{
    public static void main(String[] args)
    {
        MultiplicationTable t = new MultiplicationTable(20);
        t.printMultiplicationTable();
        Stack s = new Stack();
        for (int i = 0; i < 10; i++)
            s.push(i+100);
        while(s.getStackPointer() != 0)
            System.out.println("Stack item = "+s.pop());
    }
}
```

نکاتی در مورد تعریف کلاس

الف- چیزهایی را که قابل محاسبه است نباید متغیر گرفت مانند محیط شکل ، ولی خصوصیات اصلی مانند ضلع را باید متغیر تعریف کرد ، موارد قابل محاسبه را متُد تعریف می کنیم. کلاسها باید به گونه ای ساخته شوند که مثلاً در مورد اشکال هندسی ، ضلع آن یک بار گرفته شده و در داخل شئ قرار گیرد و بارها بتوان بوسیله متُد های مساحت و محیط ، مقادیر موردنیاز را بدست آورد بدون آنکه نیاز به پاس کردن اندازه اضلاع باشد. پس اگر بخواهیم یک کلاس برای اطلاعات پرسنلی بنویسیم سن شخص را جزو متغیر ها (attributes) نمی آوریم بلکه تاریخ تولد او را attribute می گیریم. موارد متغیر را هرچند که محاسبات پیچیده ای لازم داشته باشد جزو attribute ها نمی آوریم.

ب- یکی از اصول شئ گرایی این است که هر کلاس هر کاری در ارتباط با خودش را که خودش می تواند انجام دهد در آن گنجانده شود همچنین هر کلاس ، فقط متُد های مرتبط با کار خود را باید داشته باشد و حتی الامکان متُد ها فقط کار محاسباتی انجام دهند و خروجی چاپ نکنند.

ج- در نوشتن کلاسها باید دقت کافی به خرج داد و از تولید کلاسهای غیر ضروری اجتناب کرد مثلاً مربع ، یک مستطیل خاص است و می توان از همان کلاس مستطیل برای رسم مربع نیز استفاده کرد یا دایره یک بیضی خاص است یعنی اینکه می توانیم از کلاس بیضی برای کارهای مربوط به دایره نیز استفاده کنیم.

اگر نیاز داشته باشیم که بفهمیم مستطیل موجود مربع نیز هست یا خیر یک متُد در کلاس مستطیل می گذاریم که با یک خروجی boolean مشخص کند که شکل مربع است یا مستطیل. در مورد دایره و بیضی هم به همین روش می توان عمل کرد.

د- کلاسها را باید اصولی نوشت یعنی اینکه نباید سعی کنیم که آنها را کوتاه و پیچیده بنویسیم که کار بیشتری هم انجام دهند بلکه باید آنها را اصولی و ساده نوشت تا نگهداری و اشکال زدایی آنها راحت باشد.

چند نکته در مورد متُد ها

بطور کلی زبانهای برنامه نویسی به دو روش ، آرگومان ها را به زیر برنامه ها پاس می کنند: call-by-value و call-by-reference.

در جاوا وقتی یک Primitive Type به عنوان آرگومان به متُد پاس می شود call-by-value ، و وقتی یک Object به عنوان آرگومان به متُد پاس می شود call-by-reference اتفاق می اُفتد.

متُد می تواند هر نوع داده ، از جمله Object (داده از نوع کلاس) را برگرداند. مثال:

```
class Test
{
  int a;

  Test(int i)
  {
     a = i;
  }

  Test incrByTen()
  {
     Test temp = new Test(a+10);
      return temp;
  }
}

class RetOb
{
  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("ob2.a after second increase: "+ ob2.a);
  }
}
```

با توجه به واگذاری آدرس شئ به متغیر دیگر ، خروج شئ از حوزه تعریف شده باعث آزاد شدن حافظه اختصاص یافته نخواهد شد ، و Garbage Collector حافظه هایی را آزاد می کند که هیچ ارجاعی به آنها وجود نداشته باشد.

مفهوم static

برخی موارد لازم است یک یا چند عضو کلاس را طوری تعریف کنیم که مستقل از هر شئ آن کلاس مورد استفاده قرار گیرد. بطور معمول یک عضو کلاس باید فقط همراه یک شئ از همان کلاس مورد دسترسی قرار گیرد. اما می توان عضوی را تعریف کرد که بوسیله خودش استفاده شود ، بدون آنکه نیازی به ایجاد یک Instance از آن کلاس باشد. برای ایجاد چنین عضوی ، قبل از اعلان آن ، واژه کلیدی static را قرار می دهیم وقتی یک عضو به عنوان static اعلان می شود ، می توان بدون ایجاد شئ از آن کلاس و بدون ارجاع به هیچ یک از اشیای تولید شده از آن کلاس ، آن را مورد استفاده قرار داد . هم متُد ها و هم متغیر ها را می توان به عنوان static اعلان کرد. متُد ()main مثال خوبی برای یک عضو static است. متُد ()main بصورت static اعلان می شود چون باید قبل از وجود هر نوع شیئی فراخوانی شود. Instance Variable های اعلان شده به عنوان static ، مانند متغیر های سراسری هستند. هنگامیکه اشیائی از کلاسی که حاوی عضو static است ایجاد می شوند ، هیچ کپی از متغیر (های) static آن ساخته نمی شود در عوض ، کلیه نمونه های آن کلاس همان متغیر (های) static را به اشتراک می گذارند. متُد های اعلان شده به عنوان static دارای محدودیت های زیر هستند:

- مجاز به فراخوانی متُد های غیر static کلاس خود نیستند.
- به داده های غیر static کلاس خود دسترسی ندارند.
- امکان ارجاع به this و super را ندارند.
- این متُد ها قابل Override شدن نیستند.

اگر لازم است محاسباتی انجام شود تا متغیر های static مقدار دهی اولیه شوند ، می توان یک بلوک static تعریف کرد که فقط یک بار ، آن هم زمانی که کلاس برای اولین مرتبه بارگذاری می شود(در زمان ایجاد اولین Instance یا اولین مراجعه به عضو static آن کلاس) ، اجرا گردد. این بلوک قبل از متُد سازنده اجرا می شود. در بلوک static فقط می توان از متُد ها و متغیر های static استفاده کرد. همچنین اگر کلاسی که دارای متغیر static است بلوک static نداشته باشد یک بلوک static پیش فرض اجرا شده و متغیر های static را مقداردهی اولیه می کند چون سازنده کلاس ، آنها را مقدار دهی نمی کند. مثال:

```
import common.ReadNumber;

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 main(String args[])
    {
        int num;
        meth(42);
        num = ReadNumber.readInt();
    }
}
```

همانطور که از مثال فوق نیز مشخص می شود الگوی کلی مراجعه به یک متغیر یا متُد static به شکل زیر است:

ClassName.StaticMember

استفاده از متُد ها و متغیر های static با مفاهیم شئ گرایی تضاد دارد و حتی الامکان نباید از آنها استفاده کرد. بنابراین با احتیاط تمام از static استفاده می کنیم خصوصاً در Application های چند کاربره مانند Web base.

کلمه کلیدی final — final keyword

اگر در جاوا بخواهیم چیزی تغییر نکند از کلمه کلیدی final استفاده می کنیم. کاربرد های این کلمه بشرح زیر است:

الف- ایجاد constant

اگر این کلمه قبل از تعریف یک متغیر بیاید محتوای آن متغیر را غیر قابل تغییر می کند بنابر این لازم است در همان زمانِ تعریف ، مقداردهی اولیه شود. این حالت مشابه const در C/C++ می باشد. مثال:

```
final double PI = 3.141592654;
```

در نامگذاری متغیر های final ، تمام حروف آن را بزرگ در نظر می گیرند. متغیر های final در هر بار که از آن کلاس Instance ایجاد می شود فضای جدیدی از حافظه را اشغال نمی کنند.

ب- جلوگیری از Method Overriding

مواقعی لازم است که از لغو شدن یک متُد جلوگیری شود برای این کار قبل از تعریف متُد از کلمه کلیدی final استفاده می کنیم. مانند:

```
final int sum(int x,int y)
{
    // body of method
}
```

ج- جلوگیری از ارث بری

برای آنکه از ارث بردن کلاسهای دیگر از یک کلاس جلوگیری کنیم قبل از اعلان کلاس از واژه کلیدی final استفاده می کنیم. مثال:

```
final public class myClass
{
     // body of class
}
```

*- وقتی که یک کلاس final تعریف می شود بطور ضمنی همه متُد های آن نیز final می شوند.

پارامترهای خط فرمان — Command-Line Arguments

برخی مواقع لازم است اطلاعاتی را در زمانی که برنامه را اجرا می کنیم از محیط سیستم عامل به برنامه پاس کنیم. برای انجام این کار ، از آرایه از نوع شئ String که در متُد main در نظر گرفته شده است استفاده می کنیم. مثالِ زیر نحوه استفاده از این پارامترها را نشان می دهد.

```
class CommandLine
{
  public static void main(String args[])
  {
      for(int i=0; i<args.length; i++)
          System.out.println("args[" + i + "]: " + args[i]);
  }
}
```

کلاسهای abstract — abstract Classes

شرایطی وجود دارد که لازم است یک superclass تعریف شود تا ساختار یک موضوع معین را بدون پیاده سازی کامل از برخی متُد های آن ، اعلان نماید. یعنی گاهی می خواهیم یک superclass ایجاد کنیم که فقط یک شکل عمومی را تعریف کند که بوسیله همه subclass هایش به اشتراک گذاشته خواهد شد و پر کردن جزئیات این شکل عمومی بعهده هر یک از subclass ها واگذار می شود. چنین کلاسی طبیعت متُد هایی که subclass ها باید پیاده سازی نمایند را تعریف می کند.

یکی از مواردِ وقوع این شرایط زمانی است که superclass توانایی پیاده سازی معنی دار برای یک متُد را نداشته باشد. به عنوان مثال در تعریف کلاسی به نام Figure که انواع شکل هندسی را پیاده سازی می کند ، متُد area() ، مساحت اشکال را بدلیل متفاوت بودن طریقه محاسبه مساحت آنها ، محاسبه نمی کند بلکه یک امضای متُد را ارایه می کند که فاقد بدنه است. بدنه این متُد در هر یک از کلاسهایی که از این کلاس برای تعریف اشکال مختلف مشتق می شوند پیاده سازی (*implement*) خواهد شد چنین متُدی را *متُد abstract* گویند و کلاسی که حاوی یک یا چند متُد abstract باشد *کلاس abstract* نامیده می شود و باید قبل از کلمه کلیدی class در تعریف کلاس و نوع متُد در تعریف متُد از واژه کلیدی abstract استفاده کرد. الگوی کلی تعریف متُد abstract به شکل زیر است:

abstract type methodName(parameter-list);

روش دیگر انجام چنین کاری این است که متُد area() در superclass دارای بدنه ای باشد که مثلاً یک پیغام warning را چاپ کند و هر کلاسی که برای پیاده سازی یک شکل هندسی از این کلاس مشتق می شود متُد area() را Override کرده و کد مربوط به محاسبه مساحت آن شکل را در آن متُد قرار دهد. اگر متُد area() در subclass ، Override نشده و کد لازم برای محاسبه مساحت را نداشته باشد کلاس تعریف شده دارای متُد بی معنی بوده و اشکال دار خواهد بود. در این حالت ، باید بدنبال راهی بود تا مطمئن شد که subclass ، همه متُد های ضروری را Override می کند. راه حل جاوا برای این مشکل ، متُد abstract است. کلاسهایی که از کلاس abstract مشتق می شوند باید همه متُد های abstract آن را implement کنند و یا خودشان به عنوان یک abstract اعلان شوند. لازم به ذکر است که نام ، امضا و نوع بازگشتی متُد هایی که متُد های abstract را implement می کنند باید دقیقاً مانند هم باشند.

از یک کلاس abstract هیچ شیئی نمی توان ایجاد کرد یعنی کلاس abstract نباید بطور مستقیم با عملگر new نمونه سازی شود. به عبارت دیگر کلاس های abstract بیشتر به درد ارث بری می خورند زیرا کلاس های abstract بطور کامل تعریف نشده اند. همچنین نمی توان متُد های سازنده یا متُد های static را abstract اعلان کرد. مثال :

```
public abstract class Figure
```

```
{
   protected double dim1;
   protected double dim2;

   public Figure(double a,double b)
   {
      dim1 = a;
      dim2 = b;
   }
   public abstract double area();
}

public class Rectangle extends Figure
{
   public Rectangle(double a,double b)
   {
      super(a,b);
   }

   public double area()   // Implement area for right Rectangle
   {
      System.out.println("Inside Area for Rectangle.");
      return dim1 * dim2;
   }
}

public class Triangle extends Figure
{
   public Triangle(double a,double b)
   {
      super(a,b);
   }

   public double area()   // Implement area for right Triangle
   {
      System.out.println("Inside Area for Triangle.");
      return dim1 * dim2 / 2;
   }
}

public class AbstractAreas
{
   public static void main(String[] args)
   {
      // Figure f = new Figure(10, 10);   // illegal now
```

```
        Rectangle r = new Rectangle(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());
    }
}
```

همانطور که در مثال بالا نیز دیده می شود از کلاس Figure هیچ نمونه ای ساخته نشده است اما می توان از آن کلاس ، یک متغیر اعلان کرد و آدرس اشیاءِ از نوع subclass آن را به آن واگذار کرد به عبارت دیگر از کلاسهای abstract می توان برای اعلان Object Reference ها استفاده کرد. کاربرد این ویژگی در polymorphism است.

از abstract زمانی استفاده می کنیم که رفتار مشخصی وجود نداشته باشد و گرنه اگر رفتاری را داشته باشیم می توانیم آن را غیر abstract تعریف کرده و در کلاسهای مشتق آن را Override کنیم.

با توجه به آنچه گذشت می توان گفت که در سلسله مراتب کلاسها هرچه به طرف بالا پیش می رویم کلاس حالت کلی تری پیدا می کند کلاس موجود در بالای سلسله مراتب ، فقط می تواند صفات (متغیر ها) و رفتار هایی (متُد هایی) را تعریف کند که در همه کلاسها متُداول اند و صفات و رفتار های خاص ، در سطوح پایین تر اضافه می شوند.

واسط ها interfaces

با استفاده از interface می توان مشخص کرد که یک کلاس چه کاری باید انجام دهد ، اما چگونگی آنرا مشخص نمی کند. interface ها از نظر قواعد صرف و نحو مشابه کلاس ها هستند ، اما فاقدInstance Variable هستند و متُد های آنها بدونِ بدنه اعلان می شود. این بدان معنی است که interface ها درباره چگونگی پیاده سازی خود فرضیه ای نمی سازند. چندین کلاس می توانند یک interface را implement کنند. همچنین ، یک کلاس می تواند هر تعداد interface را implement کند. در implement کردن یک interface ، کلاس باید تمام متُد های تعریف شده بوسیله interface را implement کند. اما هر کلاسی آزاد است تا جزئیات پیاده سازی را خودش تعیین نماید. الگوی کلی :

```
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;
}
```

access ، سطح دسترسی را مشخص می کند که می تواند public باشد یا اصلاً بکار نرود(پیش فرض). name همان نام واسط است و هر شناسه معتبری را می توان بکار برد. لازم به یادآوری است که متُد های اعلان شده دارای بدنه نیستند و در انتهای

آنها بعد از parameter-list یک ; قرار می گیرد. متغیر ها را می توان در واسط اعلان نمود ولی آنها بطور ضمنی final و static هستند. همچنین باید با یک مقدار ثابت ، مقدار دهی اولیه شوند. اگر interface خودش public اعلان شده باشد ، همه اعضای آن ، بطور ضمنی public خواهند بود. اعضای interface را نمی توان private یا protected اعلان کرد. مثال :

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

الگوی کلی implement کردن یک interface به قرار زیر است:

```
access class Classname [extends superclass]
                [implements interface [,interface...]]
{
  // class body
}
```

اگر یک کلاس بیش از یک interface را implement می کند نام interface ها با یک کاما از هم جدا می شوند. در مثال زیر واسط Callback ، implement شده است:

```
class Client implements Callback
{
  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.");
  }
}
```

همانطور که در مثال نیز مشاهده می شود متُد Callback ، public تعریف شده است. هنگامی که یک interface method ، implement می شود باید public اعلان شود.

اگر کلاسی یک interface را implement کند و تمامی متُد های مشخص شده در آن interface را implement نکند لازم است آن کلاس abstract اعلان شود. مانند:

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

مانند کلاسهای abstract ، interface ها نیز قابل نمونه سازی نیستند ولی می توان متغیری از نوع interface را اعلان کرد و اشیای از نوع کلاسهایی که از آن interface پیاده سازی شده اند را به آنها واگذار کرد. هنگام فراخوانی متُد های

implement شده بوسیله متغیر از نوع interface ، آن نسخه از متُد اجرا می شود که در کلاس آن شئ پیاده سازی شده است. این روش انجام کار ، یکی از مهمترین روشهای انجام Run-time polymorphism است.

از interface ها می توان برای به اشتراک گذاشتن ثابت ها استفاده کرد این کار مشابه تعریف مقادیر ثابت در برنامه نویسی ساخت یافته است. بعبارت دیگر همه متغیر های تعریف شده در interface که بطور ضمنی final و static نیز هستند در سلسله interface ها و کلاسهایی که از این interface ارث بری می کنند وجود خواهند داشت.

اگر interface ی فقط حاوی متغیر بوده و هیچ متُدی نداشته باشد و کلاسی این interface را implement کند در واقع چیزی را پیاده سازی نمی کند و مثل این است که آن کلاس متغیر های ثابت را در کلاس به عنوان متغیر های final وارد کرده باشد.

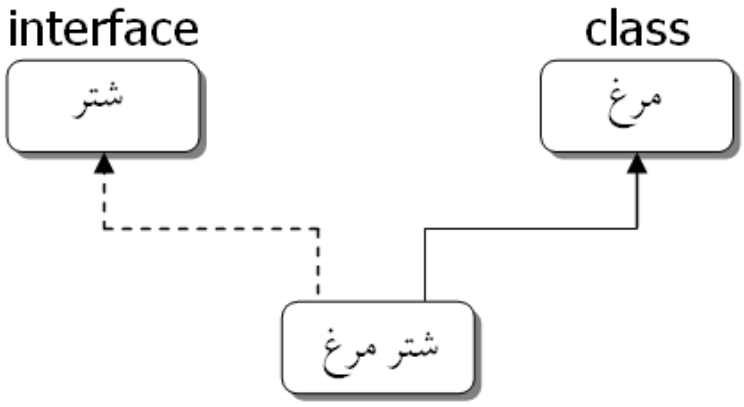
interface ها با استفاده از کلمه کلیدی extends می توانند از هم ارث بری کنند (مانند کلاس ها). اگر کلاسی یک interface را که خود از interface دیگری ارث بری می کند implement کند آن کلاس باید همه متُد هایی را که در زنجیره ارث آن interface قرار دارند را implement کند. مثال:

```
public interface A
{
   void meth1();
   void meth2();
}

public interface B extends A
{
   void meth3();
}

public 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 Examine
 {
```

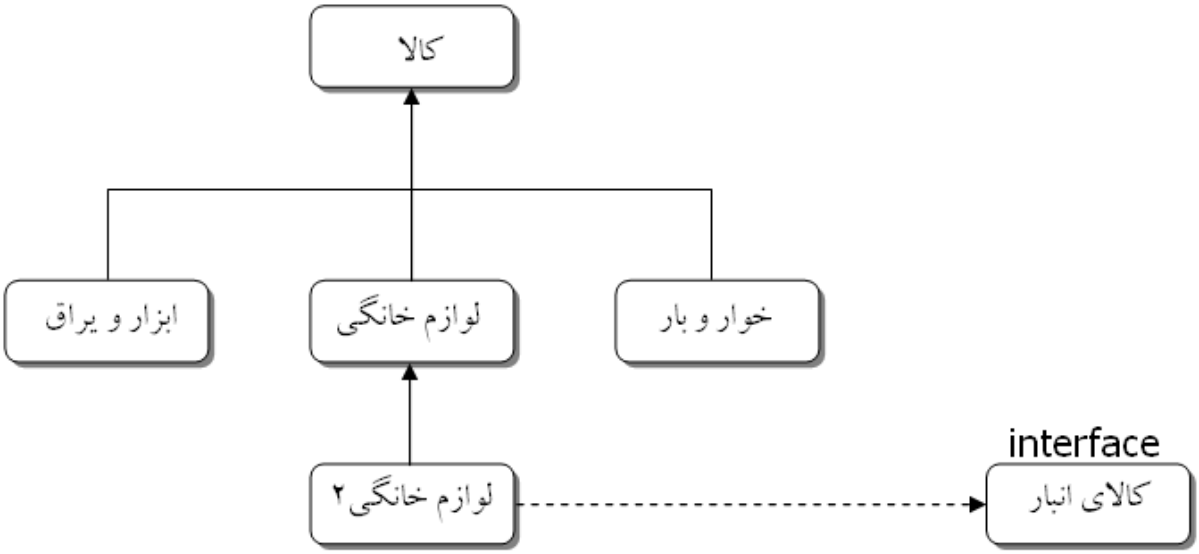
```
    public static void main(String arg[])
    {
        MyClass ob = new MyClass();
        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}
```

وراثت چندگانه در جاوا وجود ندارد و هر کلاسی فقط یک superclass می تواند داشته باشد اما می تواند مجموعه ای از رفتار های انتزاعی را از چندین interface به ارث ببرد.

مثال۱ : مدل کردن شترمرغ



مثال۲ : مدل کردن لوازم خانگی انبار



اگر کلاسی چند interface را implement کند ممکن است حاوی متُد ها یا فیلد های همنام باشند ، در این صورت موارد زیر مشخص می کنند که در هر حالت چه اتفاقی خواهد افتاد؟

- متُدی که در چند interface با یک امضا ظاهر شده است ، در کلاس پیاده سازی کننده دارای یک بدنه است طوری که همه آنها را implement می کند. البته اگر نوعی که بوسیله آنها برگردانده می شود متفاوت باشد ، خطای زمان کامپایل صادر می شود.
- اگر نام فیلد ها در interface های مختلف یکسان باشد برای استفاده از آنها باید از الگوی interfaceName.fieldName استفاده کرد.

چند ریختی Polymorphism

در دو مدل static و dynamic وجود دارد:

Static Polymorphism ⟶ Overloading (قبل از کامپایل)

Dynamic Polymorphism ⟶ Overriding (در زمان اجرا)

منظور از polymorphism بیشتر Dynamic Polymorphism است.

مثلاً وقتی سازنده ها Overload می شوند Static Polymorphism اتفاق می افتد.

تبدیل نوع کلاس Class Type Casting

کلاس Box دارای سه فیلد x ، y و z است و کلاس WeightBox که یک subclass از کلاس Box می باشد دارای فیلد دیگری به نام weight است. در دستور زیر:

```
Box plainBox = new WeightBox();
```

چون plainBox از جنس Box است فقط می تواند به اعضای تعریف شده در کلاس Box دسترسی داشته باشد. برای اینکه یک superclass بتواند از attribute ها و method های یک subclass استفاده کند باید Type Casting انجام شود یعنی superclass باید cast شود.

```
((WeightBox)plainBox).weight
```

در این حالت با مسؤولیت برنامه نویس Compiler اجازه می دهد که Type Casting انجام شود و اگر درست نبود Runtime-Error می دهد.(ClassCastException)

زمانی که ما مطمئن هستیم Type ها درست هستند ولی Compiler نمی داند ، از Type Casting استفاده می کنیم انواع Cast کردن عبارت است از:

Down Casting : پدر به فرزند یا superclass به subclass

Up Casting : فرزند به پدر یا subclass به superclass

*- Up Casting نیازی به تصریح ندارد (نیازی به نوشتن آن نیست) و بصورت خودکار انجام می شود ولی Down Casting حتماً باید انجام شود.

عملگر های منطقی && و || Short Circuit Logical Operators

همانطور که می دانید عملگر های & و | همچنین عملگر های && و || برای انجام عملیات AND و OR منطقی به کار می روند که کاربردی مشابه دارند. تفاوت عملگر های && و || (که به Short Circuit Logical Operators معروف هستند) با عملگر های & و | در این است که عملگر های && و || ، عملوند اول را ارزیابی می کنند و بر اساس آن نتیجه مشخص می شود

(البته عملوند دوم درصورت لزوم ارزیابی می شود.) ولی عملگر های & و | ، هر دو عملوند را ارزیابی می کنند و بعد نتیجه مشخص می شود در مورد عملگر && اگر اولین عملوند دارای ارزش false باشد ، عملوند بعدی هر ارزشی داشته باشد حاصل false خواهد بود همچنین در مورد عملگر || ، اگر اولین عملوند دارای ارزش true باشد بدون توجه به ارزش عملوند دوم حاصل ارزیابی true خواهد بود. کاربرد آن را می توان در مثال زیر مشاهده کرد:

```
public class ShortCircuitAndOr
{
   public static void main(String[] args)
   {
      int x, d;
      x = 40;
      d = 0;
      if((d != 0) && (x %d ) == 0)
         System.out.println(d + " is Factor");
      x = 40;
      d = 5;
      if((d != 0) && (x %d ) == 0)
         System.out.println(d + " is Factor");
      x = 40;
      d = 0;
      if((d != 0) & (x %d ) == 0)
         System.out.println("is Factor");
   }
}
```

در کلاس فوق طی سه دستور if ، بررسی می شود که آیا x بر d بخش پذیر است یا خیر؟ در دو دستور اول از شرط منطقی && و در if آخری از شرط منطقی & استفاده شده است. با توجه به اینکه ممکن است تقسیم بر صفر نیز اتفاق بیافتد در دو دستور اول از آن جلوگیری می شود ولی در دستور آخری چون هر دو عملوند ارزیابی می شوند امکان بوجود آمدن Exception وجود دارد.

دستورات break و continue

یکی از کاربرد های دستور break ، قطع ادامه اجرای دستورات حلقه تکرار (داخلی ترین حلقه تکرار) و واگذاری کنترل اجرا به اولین دستور بعد از بلوک حلقه تکرار می باشد اگر چندین بلوک حلقه تکرار بصورت تودرتو نوشته شده باشند و در داخلی ترین بلوک از این دستور استفاده شده باشد کنترل اجرا را به اولین دستور بعد از داخلی ترین بلوک (حلقه ای که دستور break در آن نوشته شده) می برد. اما break به تنهایی نمی تواند کنترل اجرا را به بعد از بلوک حلقه تکرار دلخواهی در این سری از حلقه های تودرتو ببرد. ویژگی جدید break این امکان را فراهم آورده است که بلوک مورد نظر را با یک label مشخص کرده و در دستور break آن را اعلام کنیم تا کنترل اجرا به اولین دستور بعد از بلوک برچسب خورده برود. مثال:

```
public class BreakUsage1
{
   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.");
      }
   }
```

خروجی مثال فوق به شکل زیر است:

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

بلوک ها(ی تودرتو) می توانند حلقه تکرار نباشند. مثال:

```
public class BreakUsage2
{
   public static void main(String[] args)
   {
      boolean t = true;
      first:
      {
         second:
         {
            third:
            {
               System.out.println("Before the break.");
               if (t)
                  break second; // break out of second block
               System.out.println("This won't execute");
            }
            System.out.println("This won't execute");
         }
         System.out.println("This is after second block.");
      }
   }
}
```

خروجی مثال فوق به شکل زیر است:

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

*- لازم به ذکر است که دستور break نمی تواند کنترل را به خارج از بلوک (هایی) ببرد که خود در داخل آن(ها) قرار ندارد. مثال:

```
public class BreakErr
{
   public static void main(String[] args)
   {
      one:
```

```
      for (int i = 0; i < 3; i++)
      {
         System.out.print("Pass " + i + ": ");
      }
      for (int j = 0; j < 100; j++)
      {
         if (j == 10)
            break one; // WRONG !!!
         System.out.print(j + " ");
      }
   }
}
```

به دستور continue نیز ویژگی مشابه break اضافه شده است اگر دستور continue را بدون بکار بردن برچسب در یک بلوک حلقه تکرار استفاده کنیم باقیمانده بدنه بلوک جاری (داخلی ترین بلوک) اجرا نشده و کنترل به بخش بررسی شرط تکرار بلوک حلقه تکرار واگذار می شود. در صورتی که چندین بلوک حلقه تکرار بصورت تودرتو نوشته شده باشد با برچسب زدن به بلوک مورد نظر و اعلام آن در دستور continue، می توان کنترل تکرار را به بلوک دلخواه واگذار کرد. مثال:

```
public class ContinueLabel
{
   public static void main(String args[])
   {
      outer:
      for (int i = 0; i < 10; i++)
      {
         for (int j = 0; j < 10; j++)
         {
            if (j > i)
            {
               System.out.println();
               continue outer;
            }
            System.out.print(" " + (i * j));
         }
      }
      System.out.println();
   }
}
```

خروجی مثال فوق به شکل زیر است:

```
 0
 0 1
 0 2 4
 0 3 6   9
 0 4 8   12 16
 0 5 10 15 20 25
 0 6 12 18 24 30 36
 0 7 14 21 28 35 42 49
 0 8 16 24 32 40 48 56 64
```

0 9 18 27 36 45 54 63 72 81

*- استاندارد نامگذاری label در جاوا ، همان استاندارد نامگذاری متغیر هاست و برای برچسب زدن به یک بلوک کافی است نام مورد نظر را قبل از شروع آن بلوک نوشته و پس از آن یک علامت colon (:) قرار داد.

Exception Handling اداره استثناء

استثناء یا Exception شرایط غیر عادی است که در زمان اجرای کد رخ می دهد. بعبارت دیگر Exception ، خطای حین اجرا است. مانند تقسیم بر صفر یا سعی در خواندن از فایلی که وجود ندارد. در زبان های کامپیوتری که Handling Exception را پشتیبانی نمی کنند ، خطا ها باید بصورت دستی کنترل و اداره شوند این کار معمولاً از طریق کد های خطا (error codes) انجام می شود که دارای اشکالاتی است.

در مثالِ زیر Exception تقسیم بر صفر رخ می دهد که Handle نشده است:

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

وقتی سیستمِ run-time جاوا تلاش خود برای انجام تقسیم بر صفر را آشکار می سازد یک شئ Exception ساخته و آن را پرتاب (throw) می کند. این کار باعث توقف اجرای EXC0 می شود ، زیرا هرExceptionی که پرتاب می شود ، باید بوسیله یک اداره کننده استثناء(Exception Handler) گرفته شده و بلافاصله برای آن کاری انجام گیرد. در مثال فوق ، Exception Handler مناسب وجود ندارد ، بنابراین Exception ، بوسیله Handler پیش فرض جاوا گرفته (catch) می شود. هر Exceptionی که بوسیله برنامه catch نشود ، در نهایت بوسیلهHandler پیش فرض پردازش خواهد شد. Handler پیش فرض ، یک رشته (stack trace) را که Exception را توصیف می کند نمایش می دهد و برنامه را خاتمه می دهد. برنامه فوق خروجی (stack trace) زیر را نمایش می دهد:

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

در خروجی برنامه فوق ، نام کلاس (EXC0) ، نام متُد (main) ، نام فایل (EXC0.java) و شماره خط حاوی خطا (۴) ، همگی درstack trace گنجانده شده اند. نوع Exception پرتاب شده یک subclass از کلاس RuntimeException به نام ArithmeticException می باشد که توضیح می دهد دقیقاً چه نوع خطایی اتفاق افتاده است. stack trace همیشه سلسله فراخوانی های متُد که منجر به بروز خطا شده اند را نمایش می دهد. مثال:

```
public class Exc1
{
  static void subroutine()
  {
    int d = 0;
    int a = 10 / d;
```

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

stack trace مثال فوق سلسله فراخوانی های متُد که منجر به بروز خطا شده اند را نمایش می دهد.

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

stack trace برای اشکال زدایی(debug) برنامه مفید است ، زیرا سلسله دقیق مراحلی را که منجر به Exception شده اند را نشان می دهد.

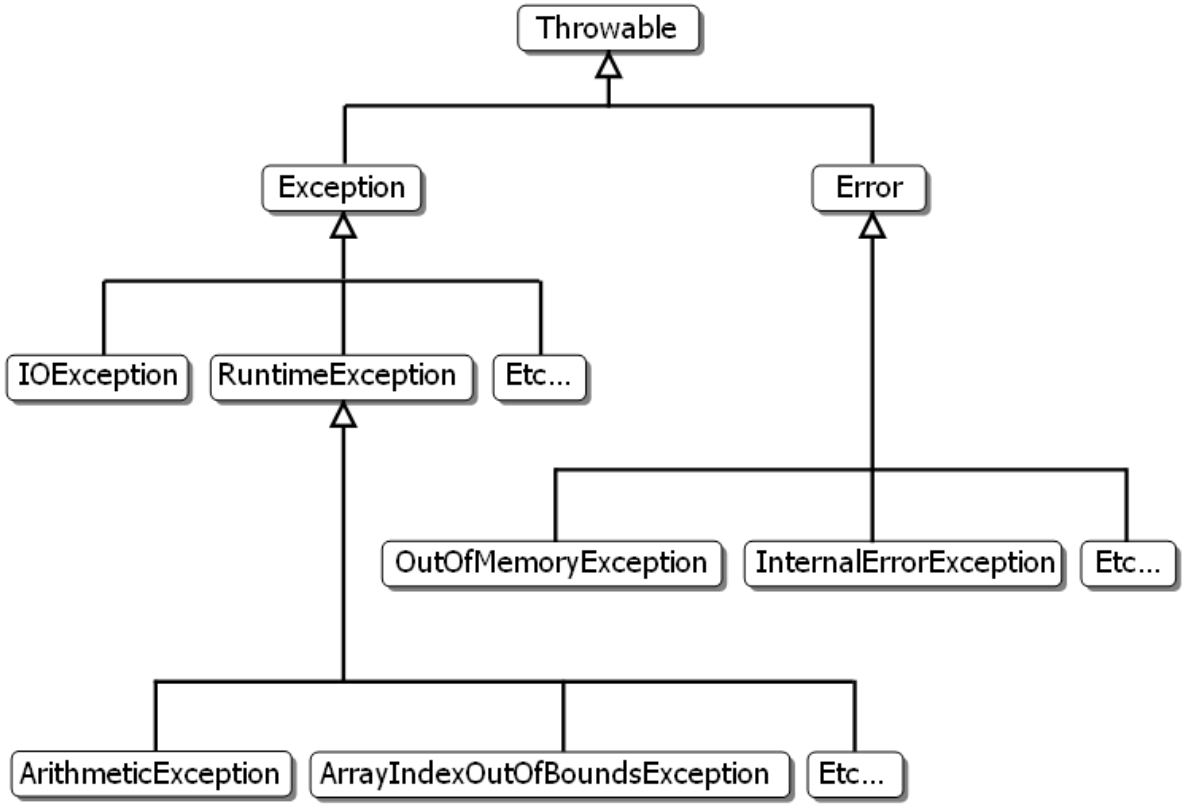
اصول اداره استثناء

Exception در جاوا ، شیئی است که شرط Exceptionی (یعنی یک خطا) را که در قطعه ای از کد حادث شده ، توصیف می کند. وقتی یک شرط Exception ایجاد می شود یک شئ که آن Exception را معرفی می کند ایجاد شده و در متُدی که آن خطا را ایجاد نموده ، پرتاب می شود. آن متُد ممکن است Exception را خودش اداره کند و یا آن را به متُد فراخواننده پرتاب کند. در هر صورت ، باید در نقطه ای Exception گرفته شده و پردازش شود . Exception ها ممکن است بوسیله سیستم run-time جاوا تولید (پرتاب) شوند یا بصورت دستی بوسیله کد های نوشته شده بوسیله برنامه نویس بوجود آیند.

Exception های پرتاب شده بوسیله جاوا با خطا های اصلی که از قوانین زبان جاوا تخطی می کنند مرتبط هستند و یا زمانی ایجاد می شوند که محدودیت های محیط اجرایی جاوا رعایت نشود. Exception های تولید شده دستی ، برای گزارش نمودن برخی شرایط خطا به فراخواننده یک متُد استفاده می شوند.

انواع Exception

کلیه انواع Exception ، subclass هایی از کلاس Throwable می باشند. بنابراین Throwableدر بالای سلسله مراتب کلاس Exception قرار دارد. بلافاصله پس از Throwable دو subclass وجود دارند که Exception ها را به دو شاخه مجزا تقسیم می کنند :



کلاس Exception برای شرایط Exceptionی که کد های برنامه نویس باید بگیرد ، استفاده می شود. همچنین از این کلاس ، برای نوشتن subclass ی که انواع Exception سفارشی را ایجاد می کند استفاده می شود.

شاخه دیگر تحت عنوان Error است که Exception هایی را تعریف می کند که انتظار نداریم. Exception های از نوع Error بوسیله سیستم run-time جاوا برای نشان دادن خطا هایی که با خود محیط run-time جاوا سر و کار دارند ، استفاده می شود. سرریزی پشته نمونه ای از این خطا هاست.

با توجه به اینکه Exception Handler پیش فرض جاوا در نهایت به اجرای برنامه خاتمه می دهد برای جلوگیری از خاتمه برنامه در مواقع بروز Exception ، باید آن را در برنامه Handle کرد.

Exception Handling در جاوا بوسیله پنج واژه کلیدی try ، catch ، throw ، throws و finally اِعمال می شود. قطعه برنامه ای را که لازم است به لحاظ Exception مورد بررسی قرار گیرد در داخل یک بلوک try قرار می دهند. اگر داخل این بلوک Exception ای پرتاب شود می توان آن را بوسیله catch گرفته و به روشی منطقی با کد نوشته شده در بلوک catch ، Handle کرد. Exception های تولید شده بوسیله سیستم run-time جاوا ، بطور خودکار پرتاب می شوند و برای آنکه یک Exception را بصورت دستی پرتاب کنیم ، از واژه کلیدی throw استفاده می کنیم. هر Exception ی که بیرون از یک متُد پرتاب می شود باید بوسیله یک دستور throws مشخص شود. هر کدی که لازم است قبل از برگشتن از یک متُد اجرا شود در یک بلوک finally قرار داده می شود.

الگوی کلی یک بلوک Exception Handling به شکل زیر است:

```
try
{
 // block of code to monitor for errors
}
```

```
catch(ExceptionType1 exOb)
{
 // exception handler for ExceptionType1
}
catch(ExceptionType2 exOb)
{
 // exception handler for ExceptionType2
}
 // ...
finally
{
 // block of code to be executed before try block ends
}
```

مثال:

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

هر بار که یک Exception پرتاب می شود، کنترل اجرا به بلوک catch منتقل می شود و هرگز از بلوک catch به بلوک try بر نمی گردد. بنابراین در مثال فوق ، خط "This will not be printed" به نمایش در نمی آید. پس از هر بار که دستور catch اجرا شود ، کنترل برنامه به خط بعدی مجموعه try/catch واگذار می شود و برنامه خاتمه نمی یابد.

کلاس Throwable متُد toString (تعریف شده در Object) را Override می کند که یک رشته را که حاوی توصیفی از Exception بوجود آمده می باشد بر می گرداند بنابراین در مثال فوق می توان با استفاده از دستور زیر ، آن را چاپ کرد:

```
catch(ArithmeticException e)
{
  System.out.println("Exception: " + e);
```

```
}
```

برخی مواقع ممکن است بیش از یک Exception در یک بلوک try رخ دهد برای Handle کردن آنها می توان از دو یا چند catch استفاده کرد که هر catch ی ، Exception متفاوتی را در بر می گیرد. پس از پرتاب شدن هر Exceptionی همه catch ها آن را بررسی می کنند و اگر یکی از catch ها آن Exception را Handle کند بقیه catch ها اجرا نشده و کنترل اجرا به دستور پس از مجموعه try/catch واگذار می شود. مثال:

```
public 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 blocks.");
  }
}
```

اگر در یک دستور try از چند catch استفاده شده باشد لازم است Exception های subclass ، قبل از superclass شان قرار گیرند در غیر این صورت Exception بوسیله superclass گرفته شده و catch مربوط به subclass تبدیل به unreachable code می شود که باعث خطای compile-time می شود زیرا هر کدی که unreachable باشد از نظر جاوا یک Error می باشد.

دستورات try می توانند بصورت تودرتو نیز بکار روند که در این صورت catch یا catch های مربوط به داخلی ترین try ، داخل بلوک try قبل از خود قرار می گیرند و الی آخر.

Exception ها معمولاً بوسیله سیستم run-time جاوا پرتاب (throw) می شوند اما می توان آنها را بصورت دستی نیز throw کرد. الگوی کلی :

```
throw ThrowableInstance;
```

اینجا ThrowableInstance یک Object از نوع کلاس Throwable یا subclass آن می باشد.

اگر متُدی استعداد صدور Exception را دارد و در داخل خود آن را Handle نمی کند لازم است آن را به متُد فراخواننده پرتاب کند تا آنجا Handle شود و بطور کلی برنامه نویس است که تصمیم می گیرد که Exception در کدام متُد Handle شود. الگوی کلی :

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

مثال:

```
public 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);
    }
  }
}
```

بسته به اینکه متُد چگونه نوشته شده باشد ، امکان دارد که Exception باعث شود تا اجرای متُد متوقف شده و به متُد فراخواننده برگردد. به عنوان مثال اگر یک متُد فایلی را هنگام ورود باز کرده و هنگام خروج آن را می بندد مشکل فوق باعث بسته نشدن فایل می شود. واژه کلیدی finally برای رفع مشکلاتی از این نوع طراحی شده است. finally یک بلوک کد ایجاد می کند که بعد از بلوک try اجرا می شود. بلوک finally چه یک استثناء پرتاب شود و چه نشود ، اجرا خواهد شد. دستورات finally همچنین قبل از برگشت های متُد اجرا خواهد شد. بلوک finally اختیاری است اما هر دستور try مستلزم حداقل یک catch یا یک finally می باشد. مثال:

```
public class FinallyDemo
{
  static void procA()
  {
    try
    {
      System.out.println("inside procA");
      throw new RuntimeException("demo");
    }
    finally
    {
      System.out.println("procA's finally");
    }
  }
```

```
  static void procB()
  {
    try
    {
      System.out.println("inside procB");
      return;
    }
    finally
    {
      System.out.println("procB's finally");
    }
  }

  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(Exception e)
    {
      System.out.println("Exception caught");
    }
    procB();
    procC();
  }
}
```

Thread

برنامه Multithread در برگیرنده دو یا چند بخش است که می توانند بصورت متقارن و همزمان اجرا شوند. هر بخش از چنین برنامه ای را یک Thread می نامند که هر Threadمی تواند یک مسیر جداگانه از اجرا را تعریف کند. با Multithread کردن برنامه می توان برنامه هایی بسیار موثر و کارا نوشت که حداکثر استفاده از CPU را داشته باشند ، زیرا آنها زمان خالی (idle time) را به حداقل ممکن کاهش می دهند.

وقتی یک برنامه جاوا شروع می شود ، حتماً قبل از آن ، یک Threadدر حال اجرا وجود دارد. این Thread را معمولاً Thread اصلی یا Main Thread برنامه می نامند. Thread اصلی به دو دلیل بسیار مهم است:

- Thread اصلی ، همان Threadی است که سایر Threadهای فرزند (child) از آن تکثیر می شوند .
- این Thread باید آخرین Threadی باشد که اجرا را تمام می کند. وقتی که Thread اصلی متوقف می شود ، برنامه نیز خاتمه خواهد یافت.

اگرچه هنگامیکه برنامه آغاز می شود ، Thread اصلی بطور خودکار ایجاد می شود ، اما می توان آن را از طریق یک شئ Thread کنترل کرد. برای انجام این کار ، باید با فراخوانی متُد currentThread() که یک عضوpublic static از کلاس Thread است ، یک ارجاع به آن بدست آورد. این متُد یک ارجاع به Threadی که در آن فراخوانی شده است را بر می گرداند. پس از ایجاد ارجاع به Thread اصلی ، می توان آن را مثل هر Threadدیگری تحت کنترل درآورد. مثال:

```
public class CurrentThreadDemo
{
   public static void main(String args[])
   {
      Thread t = Thread.currentThread();

      System.out.println("Current thread: " + t);

      t.setName("My Thread");  // change the name of the 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");
      }
   }
}
```

خروجی تولید شده بوسیله برنامه فوق به قرار زیر است:

```
Current thread :Thread[main,5,main]
After name change :Thread[My Thread,5,main]
5
4
3
2
1
```

خروجی فوق به ترتیب موارد زیر را نشان می دهد : نام Thread، حق تقدم آن ، و نام گروه مربوطه آن. بطور پیش فرض ، نام Thread اصلی main است ، تقدم آن ۵ است که مقداری پیش فرض می باشد ، همچنین نام گروهی از Threadها که این Thread به آن متعلق است ، که همان main می باشد. ThreadGroupیک نوع ساختار داده ای است که یک مجموعه از Threadها را بطور کلی کنترل می کند.

روش های ایجاد Thread

در ساده ترین حالت ، با ایجاد Instance از کلاس Thread می توان یک Threadرا بوجود آورد. جاوا دو شیوه برای انجام این کار تعریف می کند:

- extend کردن کلاس Thread.
- implement کردن واسط Runnable.

extend کردن کلاس Thread

برای ایجاد یک Thread می توان کلاس Thread را extend کرد. در extend کردن کلاس Thread باید متُد public void run(); را Override کرد ، که نقطه ورودی برای Threadجدید است. این کلاس همچنین باید متُد start() را فراخوانی کند تا اجرای Threadجدید آغاز شود. لازم به ذکر است که متُد run() می تواند سایر متُد ها را فراخوانی کند ، همچنین از سایر کلاس ها استفاده نماید و متغیر هایی درست مثل Thread اصلی را اعلان نماید. تنها تفاوت در این است که متُد run() نقطه ورودی برای Threadجدید است. این Thread وقتی اجرای متُد run() خاتمه می یابد (بر می گردد) ، پایان می گیرد. مثال:

```
public class NewThread extends Thread
{
   NewThread()
   {
      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.");
    }
}

public 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.");
    }
}
```

خروجی تولید شده بوسیله این برنامه بقرار زیر 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.
```

در یک برنامه Multithreaded ، Thread اصلی باید آخرین Thread ی باشد که اجرا را پایان می دهد. اگر Thread اصلی قبل از اینکه یک Thread فرزند خاتمه یابد ، پایان گیرد ، ممکن است سیستم run-time جاوا بحالت hang در آید. در برنامه فوق ، Thread اصلی ، آخرین Thread ی است که پایان می گیرد زیرا Thread اصلی به مدت ۱۰۰۰ میلی

ثانیه بین تکرار ها معوق می ماند در حالیکه Thread فرزند ۵۰۰ میلی ثانیه معوق می ماند. این کار باعث مـی شـود کـه Thread فرزند زودتر از Thread اصلی پایان گیرد.

پیاده سازی Runnable

واسط Runnable یک واحد از کد اجرایی را بسته بندی می کند. همچنین می توان از روی هر شـیئی کـه Runnable را implement می کند ، یک Thread ساخت.

این interface فقط حاوی متُد public void run(); مـی باشـد متُـدی کـه run() را implement مـی کنـد نقطـه ورودی Thread جدید خواهد بود. یکی از سازنده های کلاس Thread دارای شکل کلی زیر است:

```
Thread(Runnable target,String name);
```

که اولین پارامتر آن شیئی از کلاسی است که Runnable را implement کـرده باشـد و پـارامتر دوم آن نـام Thread جدید است. بنابراین برای ایجاد یک Thread جدید ، می توان مانند مثال زیر اقدام کرد:

```
public class NewThread implements Runnable
{
   Thread t;
   NewThread()
   {
      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.");
   }
}

public 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.");
    }
}
```

خروجی این برنامه مانند برنامه قبلی می باشد.

برنامه های قبلی حداکثر حاوی دو Thread بودند : Thread اصلی و Thread فرزند. در صورت لزوم می توان به تعداد مورد نیاز از Thread ها را تکثیر کرد. مثال:

```
public class ThirdThread implements Runnable
{
    String name; // name of thread
    Thread t;

    ThirdThread(String threadName)
    {
        name = threadName;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }

    public void run()
    {
        try
        {
            for (int i = 5; i > 0; i--)
            {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
```

```
        {
            System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}

public class MultiThreadDemo
 {
    public static void main(String args[])
    {
        new ThirdThread("One"); // start threads
        new ThirdThread("Two");
        new ThirdThread("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.");
    }
}
```

در مثال فوق سه Thread فرزند و Thread اصلی ، CPU را به اشتراک می گذارند. فراخوانی متُد sleep(10000) در Main Thread باعث می شود که Thread اصلی برای ده ثانیه معوق مانده و اطمینان دهد که آخر از همه پایان می یابد.

متُد های isAlive() و join()

همانطوری که ذکر شد ، Thread اصلی باید آخرین Threadی باشد که پایان می گیرد. در مثالهای قبلی ، این کار را با فراخوانی متُد sleep() در Thread اصلی که یک تأخیر کافی برای اطمینان از اینکه کلیه Thread های فرزند قبل از Thread اصلی پایان می گیرند ، انجام دادیم. که راه حل قانع کننده ای نیست. همچنین با این روش یک Thread نمی تواند از پایان یافتن Thread های دیگر آگاهی یابد. برای این کار می توان از متُد isAlive() که شکل کلی آن بصورت زیر می باشد استفاده کرد:

```
final boolean isAlive() throws InterruptedException
```

اگر Thread مربوطه در حال اجرا باشد ، متُد isAlive() مقدار true و در غیر این صورت false را بر می گرداند. متُد join() نیز کاری مشابه isAlive() انجام می دهد و استفاده از آن رایج تر است. اگر متُد join() یک Thread از

دیگری فراخوانی شود Thread فراخواننده تا خاتمه یافتن Thread مربوط به join() منتظر می ماند و پس از خاتمه آن ، Thread فراخواننده به کار خود ادامه می دهد. الگوی کلی متُد join() به شکل زیر است:

```
final void join() throws InterruptedException
```

متُد join() یک متُد Overload شده می باشد و شکل های دیگری از متُد join() وجود دارند که اجازه می دهند تا حداکثر زمانی را که می خواهیم برای پایان یافتن یک Thread خاص صبر کند را تعیین کنیم. در مثال زیر برنامه قبلی اصلاح شده و کاربرد این متُد ها را نشان می دهد:

```
public 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
   }

   public void run()
   {
      try
      {
         for (int i = 5; i > 0; i--)
         {
            System.out.println(name + ": " + i);
            Thread.sleep(1000);
         }
      }
      catch (InterruptedException e)
      {
         System.out.println(name + " interrupted.");
      }
      System.out.println(name + " exiting.");
   }
}

public 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.");
    }
}
```

اولویت بندی Thread ها

به لحاظ تئوری ، Thread های دارای اولویت بیشتر نسبت به Thread های دارای اولویت کمتر ، زمان بیشتری از CPU را می گیرند. اما در عملْ ، میزانِ وقتی که یک Thread از CPU می گیرد ، علاوه بر اولویت به عوامل دیگری هم بستگی دارد. به عنوان مثال ، اینکه چگونه یک سیستم عامل Multitasking را پیاده سازی می کند می تواند روی دسترسی نسبی به زمان CPU تأثیر داشته باشد. همچنین برخی کارها گرایش به CPU دارند چنین Thread هایی بر CPU چیره خواهند شد بنابراین لازم است روی این نوع از Thread ها ، گه گاه کنترلی اِعمال شود تا سایرThread ها بتوانند اجرا شوند. بطور کلی برای داشتن یک برنامه Multithread خوب ، نباید روی اصل اولویت متکی بود و برای کسب رفتار قابل پیش بینی با جاوای امروز ، باید از Thread هایی استفاده کرد که از کنترل کردن CPU دست بردارند.

برای تعیین اولویت یک Thread ، ازمتُد ()setPriority استفاده می شود ، که عضوی از کلاس Thread است. الگوی کلی آن به شکل زیر است:

```
public final void setPriority(int newPriority)
```

در اینجا newPriority اولویت جدید برای فراخواننده است مقدار newPriority باید در محدوده MIN_PRIORITY و MAX_PRIORITY باشد. در حال حاضر ، این مقادیر ۱ و ۱۰ می باشند برای برگرداندن یک

Thread به اولویت پیش فرض از NORM_PRIORITY استفاده می شود که فعلاً ۵ است. این اولویت ها به عنوان متغیر های final، در کلاس Thread تعریف شده اند. برای تعیین اولویت جاری از متُد getPriority() که الگوی کلی آن بصورت زیر می باشد استفاده می شود:

```
public final int getPriority()
```

مثال:

```
public class clicker implements Runnable
{
   int click = 0;
   Thread t;
   private volatile 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();
   }
}

public class HiLoPri
{
   public static void main(Sting 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.stop();
        hi.stop();

        // 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);
    }
}
```

Event Handling

Event ها ، انواع متعددی دارند. اکثر Event های Handle شده ، بوسیله ماوس و صفحه کلید تولید می شوند. Event ها در پکیج java.awt.event پشتیبانی شده اند.

دو مکانیزم Event Handling وجود دارد شیوه ای که در نسخه ۱ جاوا ارایه شده و هنوز پشتیبانی می شود و روش مدرنی که از نسخه ۱.۱ جاوا به بعد ارایه شده است و توصیه می شود برنامه های جدید از این روش استفاده کنند.

روش مدرن Event Handling بر پایه Delegation Event Model بنا شده که استانداردها و مکانیزم های تولید و پردازش Event را تعریف می کند. در این مدل یک Source (منبع) Event را تولید و به یک یا چند Listener ارسال می کند در این شیوه Listener منتظر دریافت Event می ماند و پس از دریافت ، آن را پردازش کرده و بر می گرداند. مزیّت این مدل در این است که منطق برنامه ای که Event ها را پردازش می کند از منطق برنامه رابط کاربر که Event ها را تولید می کند کاملاً جداست. عناصر رابط کاربر می توانند پردازش Event را به کد مجزّایی واگذار (Delegate) کنند.

در این مدل ، Listener ها باید با یک تولید کننده Event (Source) ثبت شوند تا از Event مطّلع شوند. حُسْنِ این روش آن است که Event ها فقط بوسیله Listener هایی دریافت می شوند که مورد نظر ماست. در نسخه ۱.۰ جاوا Event در

میان همه Listener ها پخش می شد تا بوسیله یک Component اداره شود. و Component ها باید Event هایی را دریافت می کردند که نباید پردازش کنند که مستلزم صرف زمان و کاهش سرعت بود. Delegation Event Model ، این Overhead را حذف کرده است.

Event ها

در DEM ، Event شیئی است که تغییر در وضعیت Source را توصیف می کند که می تواند در اثر تعامل شخص با یکی از عناصر رابط گرافیکی کاربر(GUI) تولید شده باشد. کار هایی مثل فشردن یک کلید ، وارد کردن یک کاراکتر بوسیله صفحه کلید ، انتخاب یک مورد از لیست ، کلیک ماوس یا خیلی از موارد مشابه که بوسیله کاربر انجام می شود می توان باعث تولید Event شود.

همچنین Event ها ممکن است بدون تعامل مستقیم کاربر با GUI تولید شوند. مثلاً وقتی که یک زمان منقضی می شود ، یک شمارنده از یک مقدار معیّن تجاوز می کند ، نقصی در نرم افزار یا سخت افزار رخ می دهد یا عملیّاتی کامل می شود. برنامه نویس می تواند هر Event ی را که مناسب برنامه خود می داند تعریف کند.

منابع Event ها

منبع شیئی است که Event تولید می کند هنگامی که وضعیت داخلی شئ به طریقی تغییر کند Event رخ می دهد منابع ممکن است بیش از یک نوع Event تولید کنند.

منبع باید شنودگر هایی را برای مطلع ساختن آنها از وقوع نوع خاصی از Event ثبت کند هر نوع Event ی ، متد ثبت کننده خاص خود را دارد الگوی کلی این متد ها بشکل زیر است :

public void add*Type*Listener(*Type*Listener *el*)

Type ، نام Event و *el* ، یک ارجاع (Reference) به شنودگر Event است. برای مثال متدی که شنودگر Event صفحه کلید را ثبت می کند ()addKeyListener می باشد همچنین متدی که شنودگر حرکت ماوس را ثبت می کند ()addMouseMotionListener می باشد هنگامی که یک Event رخ می دهد همه شنودگر های ثبت شده مطلع می شوند و یک کپی از شئ Event را دریافت می کنند که به آن *Event multicasting* گویند در هر صورت فقط شنودگر هایی از وقوع Event مطلع می شوند که برای این منظور ثبت شده باشند.

برخی منابع ، فقط اجازه ثبت یک شنودگر را می دهند. الگوی کلی چنین متد هایی به شکل زیر است:

public void add*Type*Listener(*Type*Listener *el*)
    throws java.util.TooManyListenersException

Type ، نام Event و *el* ، یک ارجاع (Reference) به شنودگر Event است. هنگامی که چنین Event ی رخ می دهد شنودگر ثبت شده مطلع می گردد که به آن *Event unicasting* گویند.

همچنین منبع باید متدی داشته باشد که بتواند شنودگر ثبت شده در خود را حذف کند. الگوی کلی چنین متد هایی به شکل زیر است:

public void remove*Type*Listener(*Type*Listener *el*)

Type ، نام Event و *el* ، یک ارجاع (Reference) به شنودگر Event است. به عنوان مثال برای حذف شنودگر صفحه کلید باید متد ()removeKeyListener فراخوانی شود.

متد های حذف و اضافه کردن شنودگر ها ، بوسیله منبع تولید کننده Event ها تأمین می شود. برای مثال ، کلاس Component متد های حذف و اضافه کردن شنودگر های Event های صفحه کلید و ماوس را در خود دارد.

شنودگر های Event

شنودگر شیئی است که وقتی Event ی رخ می دهد مطلع می شود به شرطی که دو کار ، بر روی آن انجام شده باشد اولاً بوسیله یک یا چند منبع ، برای دریافت انواع مشخصی از Event ها ثبت شده باشد ثانیاً متد هایی را برای دریافت و پردازش این Event ها implements کرده باشد.

متد هایی که Event ها را دریافت و پردازش می کنند در یک سری از interface ها تعریف شده اند که در پکیج java.awt.event قرار دارند. برای مثال ، واسط MouseMotionListener دو متد را تعریف می کند که وقتی ماوس drag می شود یا حرکت می کند Event های مربوطه را دریافت می کنند. هر شیئی که این واسط را implements کرده باشد می تواند یک یا هر دو این Event ها را دریافت و پردازش کند.

کلاسهای Event

کلاسهای مربوط به Event ها ، در هسته مکانیزم Event Handling جاوا قرار دارند. لذا در ادامه به بررسی کلاسهای Event می پردازیم. این کلاسها ابزارهایی با کاربرد آسان و سازگار برای بسته بندی Event ها هستند.

در ریشه سلسله مراتب کلاسهای Event جاوا ، کلاس EventObject قرار دارد که متعلق به پکیج java.util می باشد. این کلاس superclass همه کلاسهای Event است و تنها Constructor آن دارای الگوی زیر است:

EventObject(Object *src*)

src ، شیئی است که این Event را تولید می کند. کلاس EventObject دو متد به نام های ()getSource و ()toString دارد. متد ()getSource ، منبع Event را بر می گرداند. الگوی کلی این متد به شکل زیر است:

Object getSource()

متد ()toString نیز یک رشته را که کلاس Event را توصیف می کند را بر می گرداند.

کلاس AWTEvent یک subclass از کلاس EventObject بوده و در پکیج java.awt تعریف شده است. این کلاس superclass همه کلاسهای Event بر پایه AWT می باشد (مستقیم یا غیر مستقیم) که در Delegation Event Model مورد استفاده قرار گرفته اند. از متد ()getID موجود در این کلاس می توان برای معین کردن نوع Event استفاده کرد. امضای این متد به شکل زیر است :

int getID()

بررسی جزئیات بیشتر در مورد کلاس AWTEvent به دانشجویان واگذار می شود آنچه که دانستن آن در این بخش مهم است این است که همه کلاسهای Event مورد بررسی در این بخش subclass هایی از این کلاس می باشند و بطور خلاصه :

- EventObject ، superclass همه کلاسهای Event است.
- کلاس AWTEvent ، superclass همه کلاسهای Event بر پایه AWT می باشد که در Delegation Event Model مورد استفاده قرار گرفته اند.

پکیج java.awt.event انواع مختلفی از Event ها را که بوسیله عناصر GUI تولید می شوند تعریف می کند. جدول زیر مهمترین کلاسهای Event موجود در این پکیج و شرح مختصری از نحوه اتفاق افتادن آن Event ها را ارایه می کند.

Constructor ها و متد های پر کاربرد یکی از این کلاسها بعد از جدول شرح داده شده اند. بررسی سایر کلاسها بـه دانشـجویان واگذار می شود.

نام کلاس **Event**	شرح
ActionEvent	وقتی که button فشرده می شود ، list item ، دابل کلیک می شـود یـا menu item انتخاب می شود تولید می شود.
AdjustmentEvent	وقتی که scroll bar دستکاری می شود تولید می شود.
ComponentEvent	وقتی Component مخفی یا آشکار می شود یا تغییر اندازه می یابـد یـا منتقـل مـی شود تولید می شود.
ContainerEvent	وقتی که از Container ، یک Component حذف می شود یا به آن اضافه مـی شود تولید می شود.
FocusEvent	وقتی که Component ، Focus صفحه کلید را بدست می آورد یا از دسـت مـی دهد تولید می شود.
InputEvent	این کلاس که abstract می باشد superclass همه کلاسـهای input Event مربوط به Component ها می باشد.
ItemEvent	این Event در موارد زیر تولید می شود: وقتی که check box یا list item کلیک شود. وقتی که choice انتخاب می شود. وقتی که menu item قابل انتخاب ، انتخاب یا از حالت انتخاب خارج می شود.
KeyEvent	وقتی که کلیدی از صفحه کلید فشرده می شود تولید می شود.
MouseEvent	وقتی که ماوس ، drag ، move یا click شود یا دکمه ماوس فشار داده شود یا رها شود همچنین وقتی که ماوس از یک Component خارج می شـود یـا در آن وارد می شود تولید می شود.
MouseWheelEvent	وقتی که چرخ ماوس حرکت داده شود تولید می شود.
TextEvent	وقتی که مقدار text area یا text field تغییر کند تولید می شود.
WindowEvent	وقتـی کـه پنجـره ای فعـال ، غیرفعـال ، بـاز ، بسـته ، Minimized یـا از حالـت Minimized برگردانده می شود تولید می شود.

کلاس **ActionEvent**

وقتی که button فشرده می شود ، list item ، دابل کلیک می شود یا menu item انتخاب می شود تولید می شـود. این کلاس چهار مقدار ثابت صحیح تعریف می کند که بوسیله آنها می توان معین کرد که در حین وقوع این Event ، کدام کلیـد های مبـدل صـفحه کلیـد فشـرده شـده بودنـد. ایـن مقـادیر عبارتنـد از: ALT_MASK ، CTRL_MASK ، META_MASK و SHIFT_MASK. یک ثابت صحیح دیگر به نام ACTION_PERFORMED نیز وجود دارد که بـرای معـین کـردن Event های آن عمل بکار می رود.

کلاس ActionEvent دارای سه سازنده زیر است:

ActionEvent(Object *src*, int *type*, String *cmd*)
ActionEvent(Object *src*, int *type*, String *cmd*, int *modifiers*)
ActionEvent(Object *src*, int *type*, String *cmd*, long *when*, int *modifiers*)

در این سازنده ها ***src*** یک ارجاع به شیئی است که این Event را تولید کرده است. نوع Event بوسیله ***type*** و فرمان بوسیله رشته ***cmd*** مشخص می شود. پارامتر ***modifiers*** مشخص می کند که در زمان وقوع Event ، کدام کلید(های) مبدل (ALT ، CTRL ، META و/یا SHIFT) فشار داده شده بود(ند). پارامتر ***When*** نیز معین می کند که Event در چه زمانی رخ داده است.

نام فرمان را می توان با فراخوانی متد ()getActionCommand از شئ کلاس ActionEvent بدست آورد الگوی کلی این متد به شکل زیر است:

String getActionCommand()

برای مثال ، وقتی که button فشار داده می شود یک ActionEvent تولید می شود که دارای نام فرمانی برابر با برچسب button می باشد.

مقداری که بوسیله متد ()getModifiers برگردانده می شود مشخص کننده کلید(های) مبدلی (ALT ، CTRL ، META و/یا SHIFT) است که در زمان وقوع Event فشرده شده بودند. الگوی کلی این متد به شکل زیر است:

int getModifiers()

متد ()getWhen نیز با مقداری که بر می گرداند مشخص می کند که Event در چه زمانی رخ داده است که به آن ***timestamp*** می گویند. الگوی کلی این متد به شکل زیر است:

long getWhen()

منابع Event ها

در جدول زیر ، تعدادی از Component های رابط کاربر ارایه شده است این عناصر Event هایی را که در بخش قبل توضیح داده شدند را تولید می کنند علاوه بر عناصر GUI ارایه شده در جدول زیر عناصر دیگری مانند Applet ها نیز Event تولید می کنند. همچنین برنامه نویس نیز می تواند عناصری را بسازد که Event تولید کنند.

منبع Event	توضیح
Button	وقتی که button فشرده می شود یک ActionEvent تولید می کند.
CheckBox	وقتی که CheckBox ، انتخاب یا از حالت انتخاب خارج می شود یک ItemEvent تولید می کند.
List	وقتی یکی از item های List ، انتخاب می شود یا از حالت انتخاب خارج می شود یا دابل کلیک می شود یک ActionEvent تولید می کند.
Menu Item	وقتی که یکی از گزینه های Menu Item کلیک می شود یک ActionEvent تولید می کند ، همچنین اگر یکی از گزینه های Menu Item که قابلیت علامت زدن (checkable) را دارد علامت زده شود یا علامت

	آن برداشته شود یک ItemEvent تولید می کند.
Scrollbar	وقتی که Scrollbar دستکاری می شود یک AdjustmentEvent تولید می کند.
Text Components	وقتی که کاربر کاراکتری را وارد می کند یک TextEvent تولید می کند.
Window	وقتی که پنجره فعال ، غیر فعال ، باز یا بسته می شود WindowEvent تولید می کند.

واسط های شنودگر Event ها

همانطور که قبلاً نیز توضیح داده شد Delegation Event Model دارای دو بخش منابع (Sources) و شنودگر ها (Listeners) است. شنودگر ها با implements کردن یک یا چند interface که در پکیج java.awt.event تعریف شده اند ایجاد می شوند. هنگامی که یک Event رخ می دهد منبع Event ، متد مناسب آن Event را که در شنودگر تعریف شده است را فراخوانی می کند و یک کپی از شئ Event تولید شده را بوسیله پارامتر آن متد در اختیارش قرار می دهد. جدول زیر ، لیستی از واسط های پرکاربرد شنودگر ها و توضیح مختصری از متد های تعریف شده بوسیله آنها را ارایه می کند.

واسط	توضیح
ActionListener	یک متد برای دریافت ActionEvent تعریف می کند.
AdjustmentListener	یک متد برای دریافت AdjustmentEvent تعریف می کند.
ComponentListener	چهار متد برای تشخیص آشکار یا مخفی شدن و انتقال یا تغییر اندازه یافتن Component تعریف می کند.
ContainerListener	دو متد را برای تشخیص اضافه یا حذف شدن Component ها در یک Container تعریف می کند.
FocusListener	دو متد را برای تشخیص دریافت و از دست دادن Focus صفحه کلید بوسیله Component تعریف می کند.
ItemListener	یک متد را برای تشخیص تغییر وضعیت item تعریف می کند.
KeyListener	سه متد را برای تشخیص وقتی که کلیدی از صفحه کلید فشرده می شود ، رها می شود یا تایپ می شود تعریف می کند.
MouseListener	پنج متد را برای تشخیص وقتی که کلیدی از ماوس فشرده می شود ، رها می شود کلیک می شود یا اشاره گر آن از یک Component خارج یا در آن وارد می شود تعریف می کند.

MouseMotionListener	دو متد را برای تشخیص حرکت دادن یا drag کردن ماوس تعریف می کند.
MouseWheelListener	یک متد را برای تشخیص حرکت دادن چرخ ماوس تعریف می کند.
TextListener	یک متد را برای تشخیص تغییر مقدار یک Text تعریف می کند.
WindowFocusListener	دو متد را برای تشخیص وقتی که پنجره ای Focus را دریافت می کند یا از دست می دهد تعریف می کند.
WindowListener	هفت متد را برای تشخیص مواردی مانند فعال ، غیر فعال ، باز یا بسته شدن پنجره تعریف می کند.

واسط ActionListener

این واسط ، متد actionPerformed() را تعریف می کند و زمانی که ActionEvent رخ می دهد متد actionPerformed() بوسیله منبع تولید کننده Event ، که این شنودگر را نیز ثبت کرده است فراخوانی می شود الگوی کلی این متد به شکل زیر است:

void actionPerformed(ActionEvent *ae*)

واسط AdjustmentListener

این واسط ، متد adjustmentValueChanged() را تعریف می کند و وقتی که AdjustmentEvent رخ می دهد این متد فراخوانی می شود الگوی کلی این متد به شکل زیر است:

void adjustmentValueChanged(AdjustmentEvent *ae*)

*- بررسی متد های سایر واسط های شنودگر به دانشجویان واگذار می شود.

بکار بردن Delegation Event Model

مباحث تئوری مربوط به Delegation Event Model توضیح داده شد و مختصری نیز در مورد انواع GUI ها (Sources) بحث شد در این قسمت کاربرد عملی موارد گفته شده با چند مثال مطرح می شود با توجه به اینکه در مثال ها نیاز به استفاده از پنجره (Window) می باشد در مثال اول یکی از Event های مربوط به عنصر JFrame مطرح می شود و در دو مثال بعدی به دو تولید کننده مهم Event ، یعنی ماوس و صفحه کلید پرداخته می شود.

جاوا ، دو سری از کلاسها را برای ساختن GUI (واسط گرافیکی کاربر) ارایه کرده است: AWT(Abstract Window Toolkit) و Swing. با توجه به اینکه همه Component های ارایه شده در AWT در Swing نیز ارایه شده و تعداد قابل توجهی Component جدید مانند table ، tree ، tab و ... را نیز به این مجموعه افزوده است و از طرف دیگر Component های ارایه شده در Swing ، قدرتمند تر ، انعطاف پذیر تر و سریع تر هستند لذا از بررسی کلاسهای پکیج AWT صرف نظر کرده و به Component های ارایه شده در Swing می پردازیم.

تعداد Component های ارایه شده در Swing خیلی زیاد است و امکان بررسی همه آنها خارج از حوصله این بحث است البته عناصر مهم و مطالب پایه ای مربوطه مطرح خواهد شد لذا برای استفاده از هر یک از عناصر مطرح نشده زمان اندکی باید صرف شود.

مثال ۱: ایجاد پنجره و امکان بستن آن.

```
public class MyWindowListener implements WindowListener
{
   public void windowClosing(WindowEvent e)
   {
      System.out.println("Window Closing ...");
      System.exit(0);
   }
   public void windowOpened(WindowEvent e) {}
   public void windowClosed(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
   public void windowDeiconified(WindowEvent e) {}
   public void windowActivated(WindowEvent e) {}
   public void windowDeactivated(WindowEvent e) {}
}
//-------------------------------------------------------------------------------

public class Main
{
   public static void main(String[] args)
   {
      JFrame jFrame = new JFrame();
      MyWindowListener myWindowListener = new MyWindowListener();
      jFrame.addWindowListener(myWindowListener);
      jFrame.setBounds(200,200,600,400);
      jFrame.setVisible(true);
   }
}
```

در مثال فوق و در کلاس Main ، برای ایجاد پنجره از کلاس JFrame استفاده شده است پس از ایجاد یک نمونه از کلاس JFrame ، مختصات پنجره (دو رقم اول ، آدرس گرافیکی گوشه چپ و بالا پنجره در صفحه نمایش و دو رقم دوم طول و عرض پنجره) بوسیله متد setter آن به نام setBounds() تنظیم شده و سپس بوسیله متد setter دیگری به نام setVisible() و پارامتر true ، پنجره به نمایش در آمده است.

با توجه به مطالبی که در بحث Event Handling مطرح شد در این مثال ، jFrame یک منبع (Source) ، یعنی تولید کننده Event است یکی از Event هایی که این منبع می تواند تولید کند Event بستن پنجره است. برای دریافت و پردازش این Event لازم است یک شئ شنودگر بوسیله این شئ (jFrame) ثبت شود تا پس از وقوع Event ، شئ شنودگر مطلع و متد مورد نظر آن فراخوانی شود.

متد addWindowListener() شئ jFrame برای ثبت شنودگر این نوع Event نوشته شده است. شئ شنودگر باید به عنوان پارامتر به این متد پاس شود و با فراخوانی این متد شنودگر در منبع ثبت می شود اما فقط اشیایی را می توان به این متد پاس کرد که واسط WindowListener را implements کرده باشند چون نوع پارامتر آن از نوع واسط WindowListener تعریف شده است.

این واسط حاوی تمام متد های مربوط به Event های پنجره می باشد و در صورت وقوع هریک از آنها متد مورد نظر از شئ شنودگر ، بوسیله منبع فراخوانی می شود و بدین ترتیب شنودگر ، Event را در پارامتر متد فراخوانی شده دریافت می کند.

با توجه به اینکه تا زمانی که به Event بسته شدن پنجره پاسخ داده نشود ، پنجره بسته نخواهد شد لذا کلاسی به نام MyWindowListener که واسط WindowListener را پیاده سازی می کند نوشته شده است و همه متد های آن implements شده اند اما در این مثال فقط متد windowClosing() مورد نیاز است ، با توجه به اجباری بودن پیاده سازی همه متد های واسط ، متد هایی را که موردنظر مان نیستند را با بدنه خالی و متد windowClosing() را با دستورات مورد نظر implements می کنیم (دستور خروج از برنامه) سپس یک نمونه از این کلاس (شنودگر) را ایجاد کرده و بوسیله متد ثبت کننده شنودگر مربوطه ثبت می کنیم این کار در خط های دوم و سوم متد main انجام شده است. با انجام این کار می توان پنجره مورد نظر را بست.

برای Encapsulate کردن کارهای مربوط به تولید پنجره و ثبت Event های مربوطه می توان برنامه فوق را به شکل زیر نیز نوشت:

```
public class MyWindow extends JFrame
{
   public MyWindow() throws HeadlessException
   {
      MyWindowListener myWindowListener = new MyWindowListener();
      setBounds(200,200,600,400);
      addWindowListener(myWindowListener);
   }

   class MyWindowListener implements WindowListener
   {
      public void windowClosing(WindowEvent e)
      {
         System.out.println("Window Closing ...");
         System.exit(0);
      }
      public void windowOpened(WindowEvent e){}
      public void windowClosed(WindowEvent e){}
      public void windowIconified(WindowEvent e){}
      public void windowDeiconified(WindowEvent e){}
      public void windowActivated(WindowEvent e){}
      public void windowDeactivated(WindowEvent e){}
   }
}
//-----------------------------------------------------------------------------
```

```
public class Main
{
   public static void main(String[] args)
   {
      MyWindow myWindow = new MyWindow();
      myWindow.setVisible(true);
   }
}
```

برای این کار لازم است که کلاس JFrame را برای نوشتن کلاس خودمان extends کنیم. در مثال فوق ، کلاس MyWindow ، یک subclass از کلاس JFrame است و در کلاس Main ، یک instance از آن ساخته شده و یکی از متد های setter آن به نام setVisible() (که از کلاس JFrame به ارث رسیده است) ، با پارامتر true برای نمایش داده شدن آن بر روی صفحه نمایش فراخوانی شده است. در اینجا شئ myWindow ، منبع (Source) می باشد.

تنظیمات پنجره و ثبت Listener که در برنامه قبلی در متد main() انجام شده بودند در سازنده کلاس MyWindow ، انجام شده اند همچنین کلاس مربوط به شنودگر هم در داخل کلاس MyWindow به صورت Inner نوشته شده است.

کلاس MyWindow را می توان به صورت زیر نیز نوشت :

```
public class MyWindow extends JFrame implements WindowListener
{
   public MyWindow(String windowTitle) throws HeadlessException
   {
      super(windowTitle);
      setBounds(200,200,600,400);
      addWindowListener(this);
   }

   public void windowClosing(WindowEvent e)
   {
      System.out.println("Window Closing ...");
      System.exit(0);
   }

   public void windowOpened(WindowEvent e) {}
   public void windowClosed(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
   public void windowDeiconified(WindowEvent e) {}
   public void windowActivated(WindowEvent e) {}
   public void windowDeactivated(WindowEvent e) {}
}
//--------------------------------------------------------------------------------

public class Main
```

```
{
   public static void main(String[] args)
   {
      MyWindow myWindow = new MyWindow("My Swing Window");
      myWindow.setVisible(true);
   }
}
```

در این برنامه شئ myWindow ، هم منبع (Source) و هم شنودگر (Listener) می باشد به عبارت دیگر کلاس MyWindow ، علاوه بر extends کردن کلاس JFrame ، واسط WindowListener را نیز implements کرده است یعنی متد های موجود در WindowListener در کلاس MyWindow پیاده سازی شده اند. بنابر این کلاس MyWindow می تواند به عنوان شنودگر نیز مورد استفاده قرار گیرد. به همین دلیل در متد سازنده این کلاس ، آدرس شئ جاری (this) ، به عنوان شئ شنودگر در پارامتر متد addWindowListener() قرار داده شده است.

همچنین کلاس JFrame که superclass کلاس MyWindow می باشد دارای سازنده ای است که بوسیله آن می توان عنوان پنجره را اعلام کرد که با واژه کلیدی super ، فراخوانی شده است.

مثال۲: حال در برنامه فوق ، به کلاس MyWindow ، شنودگر های مربوط به Event های ماوس را نیز اضافه می کنیم تا به Event های ماوس عکس العمل نشان دهد.

```
public class MyWindow extends JFrame implements WindowListener
{
   public MyWindow(String windowTitle) throws HeadlessException
   {
      super(windowTitle);
      setBounds(200,200,600,400);
      addWindowListener(this);

      MyMouseEventsListener myMouseEventsListener =
                                    new MyMouseEventsListener();

      addMouseListener(myMouseEventsListener);
      addMouseMotionListener(myMouseEventsListener);
   }

   public void windowClosing(WindowEvent e)
   {
      System.out.println("Window Closing ...");
      System.exit(0);
   }

   public void windowOpened(WindowEvent e) {}
   public void windowClosed(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
```

```
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}

    class MyMouseEventsListener implements MouseListener
                                          , MouseMotionListener
    {
        public void mouseClicked(MouseEvent e)
        {
            System.out.println("mouse Clicked at x = "
                        + e.getX() + ", y = " + e.getY());
        }

        public void mousePressed(MouseEvent e)
        {
            System.out.println("mouse Pressed");
        }

        public void mouseReleased(MouseEvent e)
        {
            System.out.println("mouse Released");
        }

        public void mouseEntered(MouseEvent e)
        {
            System.out.println("mouse Entered");
        }

        public void mouseExited(MouseEvent e)
        {
            System.out.println("mouse Exited");
        }

        public void mouseDragged(MouseEvent e)
        {
            System.out.println("mouse Dragged");
        }

        public void mouseMoved(MouseEvent e)
        {
            System.out.println("mouse Moved");
        }
    }
}
//--------------------------------------------------------------------------------
```

```
public class Main
{
   public static void main(String[] args)
   {
      MyWindow myWindow = new MyWindow("My Swing Window");
      myWindow.setVisible(true);
   }
}
```

در مثال فوق به کلاس MyWindow ، کلاس MyMouseEventsListener به صورت Inner اضافه شده است که واسط های MouseListener و MouseMotionListener را پیاده سازی کرده است. این واسط ها حاوی متد هایی هستند که Event های مربوط به ماوس را Handle می کنند. در سازنده کلاس MyWindow ، یک Instance از این کلاس ساخته شده و بوسیله متد های ()addMouseListener و ()addMouseMotionListener به عنوان شنودگر Event های ماوس ثبت شده است. با توجه به اینکه کلاس MyMouseEventsListener هر دو واسط را یک جا implements کرده است لذا می توان Instance این کلاس را به هر دو متد ثبت کننده شنودگر پاس کرد.

وقتی که ماوس کلیک می شود متد ()mouseClicked فراخوانی می شود بنابر این می توان دستورات مورد نظر را در این متد نوشت همچنین ویژگیهای Event ، مانند محل کلیک ماوس را می توان از شئ Event ی که به عنوان پارامتر به این متد پاس شده است بدست آورد.

وقتی که یکی از دکمه های ماوس فشار داده می شود متد ()mousePressed ، و وقتی رها می شود متد ()mouseReleased ، وقتی که اشاره گر ماوس وارد پنجره (یا هر Component ی که این شنودگر را ثبت کرده باشد) می شود متد ()mouseEntered ، و وقتی از آن خارج می شود متد ()mouseExited ، وقتی ماوس drag می شود متد ()mouseDragged ، و وقتی ماوس حرکت داده می شود متد ()mouseMoved فراخوانی می شود.

مثال۳: برنامه ای که به Event های صفحه کلید پاسخ می دهد.

```
public class MyWindow extends JFrame implements WindowListener
{
   public MyWindow(String windowTitle) throws HeadlessException
   {
      super(windowTitle);
      setBounds(200,200,600,400);
      addWindowListener(this);
      MyKeyEventsListener myKeyEventsListener =
                  new MyKeyEventsListener();
      addKeyListener(myKeyEventsListener);
   }

   public void windowClosing(WindowEvent e)
   {
      quit();
   }
```

```
public void windowOpened(WindowEvent e) {}
public void windowClosed(WindowEvent e) {}
public void windowIconified(WindowEvent e) {}
public void windowDeiconified(WindowEvent e) {}
public void windowActivated(WindowEvent e) {}
public void windowDeactivated(WindowEvent e) {}

private void quit()
{
   System.out.println("Window Closing ...");
   System.exit(0);
}

class MyKeyEventsListener implements KeyListener
{
   public void keyTyped(KeyEvent e)
   {
      System.out.print(e.getKeyChar());
   }

   public void keyPressed(KeyEvent e)
   {
      System.out.println("Key Down");
      switch(e.getKeyCode())
      {
         case KeyEvent.VK_F1 :
            System.out.println("F1");
            break;
         case KeyEvent.VK_DELETE :
            System.out.println("Delete");
            break;
         case KeyEvent.VK_RIGHT :
            System.out.println("Right Arrow");
            break;
         case KeyEvent.VK_ALT :
            System.out.println("Alt");
            break;
         case KeyEvent.VK_ESCAPE :
            System.out.println("Escape");
            quit();
         case KeyEvent.VK_CAPS_LOCK :
            System.out.println("Caps Lock");
      }
   }
```

```
        public void keyReleased(KeyEvent e)
        {
            System.out.println("Key Up");
        }
    }
}
//-------------------------------------------------------------------------------

public class Main
{
    public static void main(Sting[] args)
    {
        MyWindow myWindow = new MyWindow("My Swing Window");
        myWindow.setVisible(true);
    }
}
```

Handle کردن Event های مربوط به صفحه کلید مانند ماوس است تفاوت آنها در این است که کلاس شنودگر Event های صفحه کلید ، واسط KeyListener را implements می کند.

وقتی کلیدی فشار داده می شود متد keyPressed() ، و وقتی رها می شود متد keyReleased() فراخوانی می شود اگر فشردن کلید از صفحه کلید باعث تولید کاراکتر شود ، متد keyTyped() نیز فراخوانی می شود. بنابراین هر وقت کاربر کلیدی را فشار می دهد دو یا سه Event رخ می دهد اگر Event مورد نظر ما دریافت کاراکتر تایپ شده بوسیله کاربر باشد می توان دو Event فشردن و رها کردن کلید را نادیده گرفت و صرفاً Handler مربوطه یعنی متد keyTyped() را مورد استفاده قرار داد. اگر برنامه نیاز به Handle کردن کلید هایی مانند Arrow Key و Function Key داشته باشد باید از متد keyPressed() استفاده کرد. برای مشخص کردن کلید زده شده می توان از ثوابت تعریف شده در کلاس KeyEvent استفاده کرد.

در مثال های فوق ، Event های ماوس و صفحه کلید که بر روی Component پنجره اتفاق می افتد Handle شده اند Handle کردن این Event ها در مورد سایر Component ها نیز مشابه پنجره است. Handle کردن سایر Event ها نیز مانند Event های ماوس و صفحه کلید است.

Swing

در بالای سلسله مراتب کلاس های Swing ، کلاس Component قرار دارد این کلاسِ abstract ، ویژگیهای عناصر Visual را Encapsulate کرده است تمام عناصری که در صفحه ، نمایش داده می شوند و با کاربر تعامل دارند subclass این کلاس می باشند. در این کلاس بیش از صد متد برای مدیریت Event ها و سایر موارد تعریف شده است.

کلاس Container که یک subclass از کلاس Component است متد هایی را به کلاس Component اضافه کرده است که قابلیت نگهداری اشیایی از نوع کلاس Component را در خود دارد. همچنین کلاس Container قابلیت نگهداری اشیایی از نوع خود (Container) را نیز دارد این امکان توانایی تولید سیستم Container چند لایه را به ما می دهد. این کلاس وظیفه چیدن و مرتب کردن عناصر اضافه شده به خود را نیز به عهده دارد.

کلاس JFrame، بصورت غیر مستقیم یکی از subclass های کلاس Container و کلاس هایی مانند JButton، بصورت غیر مستقیم subclass هایی از کلاس Component می باشند.

برای قرار دادن Component ها در پنجره چند نوع قاب (Pane) وجود دارد مانند content pane، glass pane و root pane. در این مبحث از content pane برای قرار دادن Component ها در پنجره استفاده خواهیم کرد. content pane شیئی از کلاس Container است که بصورت Instance Variable در کلاس JFrame تعریف شده است برای بدست آوردن یک ارجاع به این شئ می توان از متد getter آن به نام getContentPane() استفاده کرد الگوی کلی این متد به شکل زیر است:

```
Container getContentPane();
```

کلاس Container (که content pane شیئی از آن کلاس است) متدی به نام add() را تعریف می کند که بوسیله آن می توان Component هایی مانند JButton را به آن اضافه کرد. الگوی کلی این متد به شکل زیر است:

```
Component add(Component compObj)
```

compObj یک Instance از Component ی است که می خواهیم به پنجره اضافه شود. همچنین این متد یک ارجاع به این Component را نیز بر می گرداند. هر وقت یک Component اضافه می شود بصورت خودکار در پنجره به نمایش در می آید.

به Component، کنترل (Control) نیز گفته می شود.

کلاس Container متد دیگری به نام remove() را نیز برای برداشتن کنترل ها از پنجره تعریف می کند الگوی کلی این متد به شکل زیر است:

```
void remove(Component obj)
```

در این الگو، *obj* یک ارجاع به کنترلی است که باید برداشته شود برای برداشتن همه کنترل های اضافه شده به پنجره می توان از متد removeAll() استفاده کرد.

در مثال زیر یک برچسب (Label) به پنجره اضافه شده است:

```
public class MyWindow extends JFrame implements WindowListener
{
   public MyWindow(String windowTitle) throws HeadlessException
   {
      super(windowTitle);
      setBounds(200,200,600,400);
      addWindowListener(this);

      Container cp = getContentPane();
      JLabel jl = new JLabel("First Label");
      cp.add(jl);
   }

   public void windowClosing(WindowEvent e)
   {
      System.out.println("Window Closing ...");
      System.exit(0);
   }
```

```
    public void windowOpened(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
}
```

در Constructor مثال فوق پس از دریافت شئ content pane بوسیله متد getContentPane() و قرار دادن آن در cp ، یک Instance از کلاس JLabel ایجاد شده و سپس بوسیله متد add() شئ cp به قاب (Pane) اضافه شده است. کنترل ها بلافاصله پس از افزوده شدن به pane ، در پنجره به نمایش در می آیند.

در مثال فوق ، برچسب اضافه شده بطور خودکار در ناحیه پیش فرضی از پنجره قرار گرفت این کار بوسیله Layout Manager انجام می شود کلاس های BorderLayout ، FlowLayout ، GridLayout و چند کلاس دیگر که بوسیله جاوا تعریف شده اند Layout Manager نامیده می شوند در واقع Layout Manager مشخص می کند که کنترل اضافه شده در کدام بخش از پنجره قرار گیرد. هر کدام از Layout Manager ها الگوریتم و قالب خاصی برای قرار دادن کنترل ها در پنجره دارند برای مشخص کردن Layout Manager یک Pane ، از متد setLayout() آن استفاده می شود. الگوی کلی این متد به شکل زیر است:

void setLayout(LayoutManager *layoutObj*)

layoutObj یک ارجاع به شئ Layout Manager مورد نظر است. به صورت پیش فرض یک Instance از کلاس BorderLayout به عنوان Layout Manager قاب کلاس JFrame تنظیم شده است برای غیرفعال کردن Layout Manager تنظیم شده و تنظیم دستی محل قرار گرفتن کنترل ها در پنجره ، متد setLayout() را با پارامتر null فراخوانی می کنیم برای تنظیم دستی محل قرار گرفتن کنترل ها در پنجره ، از متد setBounds() کنترل مربوطه می توان استفاده کرد.

در ادامه به شرح مختصری از سه Layout Manager متداول می پردازیم:

در BorderLayout ، پنجره به پنج ناحیه به شکل زیر تقسیم می شود:

North		
West	Center	East
South		

برای تنظیم Layout Manager قاب پنجره به این نوع ، می توان مانند زیر عمل کرد:

```
setLayout(new BorderLayout());
```

اگر چند کنترل را به Pane اضافه کنیم همه آنها در ردیف وسط و بر روی هم قرار می گیرند بنابراین پس از اجرای برنامه ، فقط آخرین کنترل اضافه شده مشاهده خواهد شد برای قرار دادن کنترل ها در محل مورد نظر می توان از متد زیر استفاده کرد:

void add(Component *compObj*, int *index*)

compObj یک Instance از کنترلی است که می خواهیم به Pane اضافه کنیم و *index* یک عدد صحیح است که مشخص کننده محلی است که می خواهیم کنترل در آنجا قرار گیرد مقادیر معتبر برای BorderLayout ، بصورت مقادیر ثابت در کلاس مربوطه تعریف شده است مثال :

```
JLabel jl = new JLabel("Name");
```

```
add(jl,BorderLayout.North);
```

اگر Layout Manager یک Pane به FlowLayout تنظیم شده باشد کنترل ها به ترتیب از چپ به راست و از بالا به پایین در پنجره قرار می گیرند. برای انتخاب این Layout Manager می توان مانند زیر عمل کرد:

```
setLayout(new FlowLayout());
```

بصورت پیش فرض ، کنترل ها از وسط هر ردیف اضافه می شوند که می توان با استفاده از مقادیر ثابت تعریف شده در این کلاس نحوه چیده شدن آنها را تغییر داد مانند:

```
setLayout(new FlowLayout(FlowLayout.RIGHT));
```

اگر بخواهیم پنجره به سطرها و ستون های مساوی تقسیم شده و کنترل ها داخل آنها قرار گیرند Layout Manager را به GridLayout تنظیم می کنیم:

```
setLayout(new GridLayout(3,4));
```

این دستور باعث می شود که پنجره به سه سطر و چهار ستون تقسیم شود در این حالت کنترل ها به ترتیب از چپ به راست و از بالا به پایین در خانه های پنجره قرار می گیرند.

در مثال زیر چند کنترل در پنجره قرار گرفته و Event مربوط به کنترل JButton ، پردازش می شود:

```
public class MyWindow extends JFrame implements WindowListener
{
   private JLabel jl;
   private JTextField jtf;
   private JButton jb;
   private JLabel printLabel;

   public MyWindow(String windowTitle) throws HeadlessException
   {
      super(windowTitle);
      setBounds(200,200,600,400);
      addWindowListener(this);

      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());

      jl = new JLabel("Name : ");
      jtf = new JTextField(40);
      jb = new JButton("Print");
      printLabel = new JLabel();

      cp.add(jl);
      cp.add(jtf);
      cp.add(jb);
      cp.add(printLabel);

      jb.addActionListener(new MyActionListener());
```

```
    }

    class MyActionListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            printLabel.setText("You typed : " + jtf.getText());
        }
    }

    public void windowClosing(WindowEvent e)
    {
        System.out.println("Window Closing ...");
        System.exit(0);
    }

    public void windowOpened(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
}
```

در این مثال ، پس از انتخاب FlowLayout به عنوان Layout Manager قاب (با دستور cp.setLayout(new FlowLayout())) ، Instance هایی از کلاس های کنترل JLabel ، JTextField و JButton ایجاد شده و بوسیله متد cp.add() به قاب پنجره اضافه شده اند. سپس یک نمونه از کلاس MyActionListener ، به عنوان شنودگر منبعِ JButton (jb) در آن ثبت شده است. شنودگر Event فشار دادن دکمه JButton ، باید واسط ActionListener را implements کند تنها متد این واسط actionPerformed() می باشد که پس از فشار دادن button فراخوانی می شود. در این متد ، از شئ jtf (کنترل JTextField) ، متد getText() فراخوانی شده و رشته تایپ شده در آن اخذ شده و به متد setText() شئ printLabel (کنترل JLabel) پاس می شود تا بر روی پنجره نمایش داده شود.

اشاره به چند امکان دیگر Swing

برای افزودن MenuBar به پنجره ، از سه کلاس JMenuBar ، JMenu و JMenuItem استفاده می شود. هر JMenuBar از تعدادی شئ JMenu ، و هر JMenu از تعدادی شئ JMenuItem تشکیل می شود برای ایجاد MenuBar مورد نظر لازم است یک Instance از کلاس JMenuBar را بوسیله متد setJMenuBar() در JFrame ثبت کرد لازم به ذکر است که این متد متعلق به کلاس JFrame است و در شئ content pane ثبت نمی شود. سپس به تعداد مورد نیاز از کلاس JMenu ، Instance ایجاد کرده و بوسیله متد add() کلاس JMenuBar به آن اضافه می کنیم و بدین ترتیب منو های مورد نظر ایجاد می شوند برای ایجاد گزینه های منو ها ، به تعداد لازم برای هر منو از کلاس JMenuItem ،

Instance ایجاد کرد و بوسیله متد add() کلاس JMenu به آن اضافه می کنیم. الگوی کلی Constructor های متداول کلاس های فوق به شکل زیر است :

```
JMenuBar()
JMenu(String name)
JMenuItem(String name)
```

در الگو های فوق ، *name* نام منو یا گزینه منو می باشد.

برای مطلع شدن از وقوع Event کلیک شدن گزینه های منو ، شئ شنودگری را که واسط ActionListener را implements کرده باشد بوسیله متد addActionListener() کلاس JMenuItem ثبت می کنیم.

برای ایجاد کادر های مکالمه (Dialog Box) ، از کلاس JDialog استفاده می کنیم نحوه اضافه کردن کنترل ها به این نوع پنجره ، مانند کلاس JFrame است الگوی کلی Constructor این کلاس به شکل زیر است:

```
JDialog(Frame owner, String title, boolean modal)
```

در الگوی فوق ، *owner* یک Reference به پنجره ای است که این کادر مکالمه از آن ایجاد می شود. *title* ، رشته ای است که در نوار عنوان کادر مکالمه ظاهر می شود و *modal* برای مشخص کردن نوع کادر مکالمه است اگر مقدار آن true باشد به معنی آن است که تا بسته نشدن کادر مکالمه ، کاربر اجازه کار با بخش های دیگر Application را ندارد.

کلاس JDialog ، ده Constructor دیگر نیز دارد که عملکرد آنها مشابه سازنده ای است که توضیح داده شد.

کلاس JPanel یکی از subclass های کلاس Container است و می تواند مانند content pane تعدادی کنترل را در خود نگهداری کند. Layout Manager پیش فرض این Container ، FlowLayout می باشد که قابل تغییر است. با توجه به اینکه این کلاس می تواند تعدادی کنترل را در خود نگهداری کند می توان کنترل های مورد نظر را به آن اضافه کرد سپس همه آنها را به عنوان یک Component به قاب افزود. مهمترین کاربرد این کلاس ، دسته بندی کنترل ها و چیدن راحت تر آنها در قاب (content pane) می باشد برای مثال ، اگر Layout Manager قابی ، BorderLayout باشد قاب به پنج ناحیه Center ، North ، South ، East و West تقسیم شده است حال می توان پنج دسته از کنترل ها را در پنج JPanel قرار داد سپس هریک از JPanel ها را در یکی از پنج ناحیه BorderLayout اضافه کرد.

برای ایجاد پنجره ، تحت پنجره اصلی ، از کلاس های JInternalFrame و JDesktopPane استفاده می شود. قطعه برنامه زیر دو پنجره ، تحت پنجره اصلی ایجاد می کند:

```
JDesktopPane desktop = new JDesktopPane();
setContentPane(desktop);

JInternalFrame jif1=new JInternalFrame("Int. Win1",true,true,true, true);
jif1.setBounds(20, 20, 200, 150);
desktop.add(jif1);
jif1.setVisible(true);

JInternalFrame jif2= new JInternalFrame("Int. Win2",true,true,true, true);
jif2.setBounds(50, 50, 200, 150);
desktop.add(jif2);
jif2.setVisible(true);
```

برای اضافه کردن پنجره ، باید قاب جاری (content pane) ، به شیئی از کلاس JDesktopPane تغییر یابد سپس پنجره های ایجاد شده به آن افزوده شوند لذا در قطعه برنامه فوق ابتدا یک Instance از کلاس مذکور ایجاد شده و سپس بوسیله متد setContentPane() ، که در کلاس JFrame تعریف شده است قاب جاری به شیئی از کلاس JDesktopPane تنظیم شده است. برای ایجاد پنجره در قاب جدید (desktop) ، از کلاس JInternalFrame ، نمونه ایجاد شده و به desktop اضافه شده است.

الگوی کلی Constructor متداول کلاس JInternalFrame به شکل زیر است:

JInternalFrame(String *title*, boolean *resizable*, boolean *closable*,
boolean *maxzimizable*, boolean *iconifiable*)

در این الگو ، *title* رشته ای است که عنوان پنجره را مشخص می کند پارامترهای بعدی در صورت true بودن ، به ترتیب باعث می شوند که پنجره امکان تغییر اندازه یافتن ، بسته شدن ، Maximize شدن (در پنجره اصلی) و Minimize شدن (در پنجره اصلی) را داشته باشد.

در جدول زیر لیستی از Event های مهم ، Listener های مربوطه و Source های متداولی که این Event ها را تولید می کنند ارایه شده است:

Event ، واسط Listener و متد های اضافه کردن و حذف کردن شنودگر مربوطه	**Component هایی که این Event را تولید می کنند**
ActionEvent **ActionListener** **addActionListener( )** **removeActionListener( )**	**JButton, JList, JTextField,** **JMenuItem** and its subclasses, including **JCheckBoxMenuItem,** **JMenu,** and **JpopupMenu.**
AdjustmentEvent **AdjustmentListener** **addAdjustmentListener( )** **removeAdjustmentListener( )**	**JScrollbar** and anything that implements the **Adjustable interface.**
ComponentEvent **ComponentListener** **addComponentListener( )** **removeComponentListener( )**	**Component** and its subclasses, including **JButton, JCheckBox, JComboBox,** **Container, JPanel, JApplet, JScrollPane,** **Window, JDialog, JFrame, JLabel, JList,** **JScrollbar, JTextArea,** and **JTextField.**
ContainerEvent **ContainerListener** **addContainerListener( )** **removeContainerListener( )**	**Container** and its subclasses, including **JPanel, JApplet,** **JScrollPane, Window, JDialog** and **JFrame.**
FocusEvent **FocusListener** **addFocusListener( )** **removeFocusListener( )**	**Component** and **subclasses.**
KeyEvent **KeyListener** **addKeyListener( )** **removeKeyListener( )**	**Component** and **subclasses.**
MouseEvent (for both clicks and motion) **MouseListener** **addMouseListener( )** **removeMouseListener( )**	**Component** and **subclasses.**
MouseEvent (for both clicks and motion) **MouseMotionListener** **addMouseMotionListener( )** **removeMouseMotionListener( )**	**Component** and **subclasses.**
WindowEvent	**Window** and its subclasses,

WindowListener **addWindowListener( )** **removeWindowListener( )**	including **JDialog and JFrame.**
ItemEvent **ItemListener** **addItemListener( )** **removeItemListener( )**	**JCheckBox, JCheckBoxMenuItem, JComboBox, JList,** and anything that implements the **ItemSelectable interface.**
TextEvent **TextListener** **addTextListener( )** **removeTextListener( )**	Anything subclassed from **JTextComponent,** including **JTextArea** and **JTextField.**

همچنین در جدول زیر لیستی از واسط های Listener مهم و متد هایشان ارایه شده است:

واسط **Listener** (و **Adapter** مربوطه)	متد(ها)
ActionListener	**actionPerformed(ActionEvent)**
AdjustmentListener	**adjustmentValueChanged(AdjustmentEvent)**
ComponentListener **ComponentAdapter**	**componentHidden(ComponentEvent)** **componentShown(ComponentEvent)** **componentMoved(ComponentEvent)** **componentResized(ComponentEvent)**
ContainerListener **ContainerAdapter**	**componentAdded(ContainerEvent)** **componentRemoved(ContainerEvent)**
FocusListener **FocusAdapter**	**focusGained(FocusEvent)** **focusLost(FocusEvent)**
KeyListener **KeyAdapter**	**keyPressed(KeyEvent)** **keyReleased(KeyEvent)** **keyTyped(KeyEvent)**
MouseListener **MouseAdapter**	**mouseClicked(MouseEvent)** **mouseEntered(MouseEvent)** **mouseExited(MouseEvent)** **mousePressed(MouseEvent)** **mouseReleased(MouseEvent)**
MouseMotionListener **MouseMotionAdapter**	**mouseDragged(MouseEvent)** **mouseMoved(MouseEvent)**
WindowListener **WindowAdapter**	**windowOpened(WindowEvent)** **windowClosing(WindowEvent)** **windowClosed(WindowEvent)** **windowActivated(WindowEvent)** **windowDeactivated(WindowEvent)** **windowIconified(WindowEvent)** **windowDeiconified(WindowEvent)**
ItemListener	**itemStateChanged(ItemEvent)**

*- Adapter به کلاس abstract ی گفته می شود که تمام متد های یک interface را با بدنه خالی implements کرده باشد برای مثال کلاس MouseMotionAdapter بصورت زیر است:

```
public abstract class MouseMotionAdapter
                                    implements MouseMotionListener
{
   public void mouseDragged(MouseEvent e) {}
   public void mouseMoved(MouseEvent e) {}
```

```
}
```

کاربرد کلاس های Adapter برای مواقعی است که کلاس شنودگر ، لازم است یک یا چند متد از متد های یک واسط را implements کند و نیاز به implements کردن سایر متد ها نداشته باشد در این صورت ، کلاس شنودگر ، کلاس Adapter را extends کرده و فقط متد های مورد نیاز را Override می کند.

JavaDoc

جاوا سه نوع Comment را پشتیبانی می کند. دو نوع آنها // و /* */ می باشند. سومین نوع آنها *توضیحات مستند سازی* یا *documentation comments* یا اصطلاحاً *JavaDoc* نامیده می شود. که با /** شروع شده و به */ ختم می شود. با JavaDoc می توان توضیحات مربوط به برنامه را در خودش جاسازی کرد و سپس با استفاده از JavaDoc Utility ، اطلاعات و توضیحات برنامه را استخراج و در فایل HTML قرار داد. JavaDoc مستندسازی برنامه ها را آسان و راحت می کند. مستندات تولید شده بوسیله JavaDoc تقریباً قابل اطمینان هستند چون این روشی است که Sun از آن برای مستندسازی Java API library استفاده کرده است.

دو نمونه از Tag های قابل تشخیص بوسیله JavaDoc Utility عبارتند از:

@author

نویسنده کلاس را مشخص می کند.

@deprecated

اعلام می کند که استفاده از کلاس ، یا عضوی از کلاس بد دانسته شده یا منسوخ شده است.

پایان