

معرفی کلاسها در جاوا

کلاس هسته اصلی جاوا است. کلاس یک ساختار منطقی است که تمامیت زبان جاوا بر آن استوار شده، زیرا شکل (shape) و طبیعت یک شیء را روشن می کند. کلاس همچنین شکل دهنده اساس برنامه نویسی شیء گرا در جاوا می باشد. هر مفهومی که مایلید در یک برنامه جاوا پیاده سازی نمایید باید ابتدا داخل یک کلاس کپسول سازی شود. از این پس می آموزید چگونه یک کلاس را برای تولید اشیاء استفاده کنید. همچنین درباره روشها (methods) و سازنده ها (constructors) و واژه کلیدی this مطالبی می آموزید.

بنیادهای کلاس در جاوا

کلاسهای تولید شده در بحثهای گذشته فقط برای کپسول سازی روش main() استفاده می شد، که برای نشان دادن اصول دستور زبان جاوا مناسب بودند. شاید بهترین چیزی که باید درباره یک کلاس بدانید این است که کلاس یک نوع جدید داده را تعریف می کند. هر بار که این نوع تعریف شود، می توان از آن برای ایجاد اشیائی از همان نوع استفاده نمود. بنابراین، یک کلاس قالبی (template) برای یک شیء است و یک شیء نمونه ای (instance) از یک کلاس است. چون شیء یک نمونه از یک کلاس است غالباً کلمات شیء (object) و نمونه (instance) را بصورت مترادف بکار می بریم.

شکل عمومی یک کلاس

هنگامیکه یک کلاس را تعریف می کنید، در حقیقت شکل و طبیعت دقیق آن کلاس را اعلان می کنید. ابتکار را با توصیف داده های موجود در آن کلاس و کدهایی که روی آن داده ها عمل می کنند، انجام می دهید. در حالیکه کلاسها ممکن است خیلی ساده فقط شامل داده یا فقط کد باشند، اکثر کلاسهای واقعی هردو موضوع را دربرمیگیرند. بعداً خواهید دید که کد یک کلاس، رابط آن به داده های همان کلاس را توصیف میکند. یک کلاس را با واژه کلیدی class اعلان می کنند. کلاسهایی که تا بحال استفاده شده اند، نوع بسیار محدود از شکل کامل کلاسها بوده اند. خواهید دید که کلاسها می توانند (و معمولاً هم) بسیار پیچیده تر باشند. شکل عمومی توصیف یک کلاس به شرح زیر است:

```
type methodName2(parameter-list){
// body of method
}
//...
type methodNameN(parameter-list){
// body of method
}
}
```

داده یا متغیرهایی که داخل یک کلاس تعریف شده اند را متغیرهای نمونه (instance variables) می نامند . کدها ، داخل روشها (methods) قرار می گیرند . روشها و متغیرهای تعریف شده داخل یک کلاس را اعضای (members) یک کلاس می نامند . در اکثر کلاسها ، متغیرهای نمونه یا روی روشهای تعریف شده برای آن کلاس عمل کرده یا توسط این روشها مورد دسترسی قرار می گیرند . بنابراین ، روشها تعیین کننده چگونگی استفاده از داده های یک کلاس هستند . متغیرهای تعریف شده داخل یک کلاس ، متغیرهای نمونه خوانده شده زیرا هر نمونه از کلاس (یعنی هر شیء یک کلاس) شامل کپی خاص خودش از این متغیرهاست . بنابراین داده مربوط به یک شیء ، جدا و منحصر بفرد از داده مربوط به شیء دیگری است . ما بزودی این نکته را بررسی خواهیم نمود ، اما فعلاً باید این نکته بسیار مهم را یاد داشته باشید . کلیه روشها نظیر main() همان شکل عمومی را دارند که تاکنون استفاده کرده ایم . اما ، اکثر روشها را بعنوان static یا public توصیف نمی کنند . توجه داشته باشید که شکل عمومی یک کلاس ، یک روش main() را توصیف نمی کند . کلاسهای جاوا لزومی ندارد که یک روش main() داشته باشند . فقط اگر کلاس ، نقطه شروع برنامه شما باشد ، باید یک روش main() را توصیف نمایید . علاوه بر این ، ریز برنامه ها (applets) اصولاً نیازی به روش main() ندارند . نکته : برنامه نویسان ++C آگاه باشند که اعلان کلاس و پیاده سازی روشها در یک مکان ذخیره شده و بصورت جداگانه تعریف نمی شوند . این حالت گاهی فابلهای خیلی بزرگ java ایجاد می کند ، زیرا هر کلاس باید کاملاً در یک فایل منبع تکی تعریف شود . این طرح در جاوا رعایت شد زیرا احساس می شد که در بلند مدت ، در اختیار داشتن مشخصات ، اعلانات و پیاده سازی در یک مکان ، امکان دسترسی آسانتر کد را بوجود می آورد . یک کلاس ساده بررسی خود را با یک نمونه ساده از کلاسها شروع می کنیم . در اینجا کلاسی تحت عنوان Box وجود دارد که سه متغیر نمونه را تعریف می کند width : ، height ، و depth و فعلاً " ، کلاس Box دربرگیرنده روشها نیست .

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

قبلاً هم گفتیم که یک کلاس نوع جدیدی از داده را توصیف می کند . در این مثال نوع جدید داده را Box نامیده ایم . از این نام برای اعلان اشیاء از نوع Box استفاده می کنید . نکته مهم این است که اعلان یک کلاس فقط یک الگو یا قالب را ایجاد می کند ، اما یک شیء واقعی بوجود نمی آورد . بنابراین ، کد قبلی ، شیئی از نوع Box را بوجود نمی آورد . برای اینکه واقعاً یک شیء Box را بوجود آورید ، باید از دستوری نظیر مورد زیر استفاده نمایید :

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

پس از اجرای این دستور ، mybox نمونه ای از Box خواهد بود . و بدین ترتیب این شیء وجود فیزیکی و واقعی پیدا می کند . مجدداً یاد داشته باشید که هر بار یک نمونه از کلاسی ایجاد می کنید ، شیئی ایجاد کرده اید که دربرگیرنده کپی (نسخه خاص) خود از هر متغیر نمونه تعریف شده توسط کلاس خواهد بود . بدین ترتیب ، هر شیء Box دربرگیرنده کپی های خود از متغیرهای نمونه width ،

depth و height و می باشد . برای دسترسی به این متغیرها از عملگر نقطه (.) استفاده می کنید . عملگر نقطه ای ، نام یک شی ئ را با نام

یک متغیر نمونه پیوند می دهد . بعنوان مثال ، برای منتسب کردن مقدار 100 به متغیر width در mybox ر ، از دستور زیر استفاده نمایید :

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

این دستور به کامپایلر می گوید که کپی width که داخل شی ئ mybox قرار گرفته را معادل عدد 100 قرار دهد . بطور کلی ، از

عملگر نقطه ای برای دسترسی هم به متغیرهای نمونه و هم به روشهای موجود در یک شی ئ استفاده می شود . در اینجا یک برنامه کامل را

مشاهده میکنید که از کلاس Box استفاده کرده است :

```
/* A program that uses the Box class.  
Call this file BoxDemo.java  
/*  
class Box {  
double width;  
double height;  
double depth;  
}
```

```
// This class declares an object of type Box.  
class BoxDemo {  
public static void main(String args[] ){  
Box mybox = new Box();  
double vol;  
  
// assign values to mybox's instance variables  
mybox.width = 10;  
mybox.height = 20;  
mybox.depth = 15;  
  
// compute volume of box  
vol = mybox.width * mybox.height * mybox.depth;  
  
System.out.println("Volume is " + vol);
```

```
}  
}
```

فایلی را که دربرگیرنده این برنامه است باید با نام `BoxDemo.java` بخوانید زیرا روش `main()` در کلاس `BoxDemo` و نه در کلاس `Box` قرار گرفته است. هنگامیکه این برنامه را کامپایل می کنید، می بینید که دو فایل `class` ایجاد شده اند، یکی برای `Box` و دیگری برای `BoxDemo`. کامپایلر جاوا بطور خودکار هر کلاس را در فایل `class` مربوط به خودش قرار می دهد. ضرورتی ندارد که کلاس `Box` و `BoxDemo` و هر دو در یک فایل منبع قرار گیرند. می توانید هر کلاس را در فایل خاص خودش گذاشته و آنها را بترتیب `Box.java` و `BoxDemo.java` و بنامید. برای اجرای این برنامه باید `BoxDemo.class` را اجرا کنید. پس از اینکار حاصل زیر را بدست می آورید:

Volume is 3000

قبلاً هم گفتیم که هر شیء دارای کپی های خاص خودش از متغیرهای نمونه است. یعنی اگر دو شیء `Box` داشته باشید، هر کدام بتهایی کپی (یا نسخه ای) از `width`، `height` و `length` خواهند داشت. مهم است بدانید که تغییرات در متغیرهای نمونه یک شیء تاثیری روی متغیرهای نمونه کلاس دیگر نخواهد داشت. بعنوان مثال، برنامه بعدی دو شیء `Box` را اعلان می کند:

```
// This program declares two Box objects.  
  
class Box {  
    double width;  
    double height;  
    double depth;  
}  
  
class BoxDemo2 {  
    public static void main(String args[]) {  
  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double vol;  
  
        // assign values to mybox1's instance variables  
        mybox1.width = 10;  
        mybox1.height = 20;  
        mybox1.depth = 15;
```

```
/* assign different values to mybox2's  
instance variables */  
mybox2.width = 3;  
mybox2.height = 6;  
mybox2.depth = 9;  
  
// compute volume of first box  
vol = mybox1.width * mybox1.height * mybox1.depth;  
System.out.println("Volume is " + vol);  
  
// compute volume of second box  
vol = mybox2.width * mybox2.height * mybox2.depth;  
System.out.println("Volume is " + vol);  
}  
}
```

خروجی تولید شده توسط این برنامه بقرار زیر می باشد :

Volume is 3000

Volume is 162

تعریف کنترل دسترسی

می دانید که کپسول سازی ، داده ها را با کدی که با آن داده ها سر و کار دارد پیوند میدهد. اما کپسول سازی یک حصلت بسیار مهم دیگر هم دارد: کنترل دستیابی . (access control) . از طریق کپسول سازی ، می توانید کنترل کنید چه بخشهایی از یک برنامه می توانند به اعضای یک کلاس دسترسی داشته باشند . با کنترل نمودن دستیابی ، می توانید از سوء استفاده جلوگیری نمایید ، بعنوان مثال ، اجازه دادن برای دسترسی به داده ها فقط از طریق یک مجموعه خوش تعریف از روشها ، مانع سوء استفاده از آن داده ها می شود . بنابراین ، وقتی این کار بخوبی پیاده سازی شود یک کلاس یک "جعبه سیاه (black box)" تولید می کند که امکان دارد مورد استفاده قرار گیرد، اما عملکرد درونی آن جعبه در معرض دسترسی دیگران قرار نخواهد داشت . اما کلاسهایی که قبلاً تعریف شده اند این هدف را بطور کامل برآورده نمی سازند . بعنوان مثال ، کلاس stack را در نظر بگیرید . اگرچه این امر حقیقت دارد که روشهای push() و pop() رابطهای کنترل شده ای به پشته هستند ، اما این رابط تاکید نشده

است . یعنی این امکان وجود دارد که بخش دیگری از برنامه این روشها را دور زده و مستقیماً به پشته دسترسی یابد . البته این خاصیت در دستان افراد ناجور، سبب مشکلاتی خواهد شد . در این قسمت مکانیسمی به شما معرفی می کنیم که توسط آن با دقت تمام دسترسی به اعضای مختلف یک کلاس را کنترل می کنید .

چگونگی دسترسی به یک عضو توسط "توصیفگر دسترسی (access specifier)" که اعلان آن عضو را تغییر میدهد، تعریف خواهد شد. جاوا مجموعه غنی از "توصیفگرهای دسترسی" را عرضه می کند. برخی جوانب کنترل دسترسی بشدت با وراثت و بسته ها (packages) مرتبط اند. (یک بسته ضرورتاً یک نوع گروه بندی از کلاسهها است) . اگر یکبار بنیادهای کنترل دسترسی را فرا گیرید ، آنگاه بقیه مطالب را براحتی درک خواهید نمود . توصیفگرهای دسترسی در جاوا public، private، و protected هستند. جاوا همچنین یک سطح دسترسی پیش فرض را تعریف کرده است protected . زمانی استفاده می شود که وراثت وجود داشته باشد .

با تعریف public و private و شروع می کنیم. اگر یک عضو کلاسی را با توصیفگر public توصیف می کنیم ، آن عضو توسط هر کد دیگری در برنامه قابل دسترسی خواهد بود. اگر یک عضو کلاسی را بعنوان private مشخص می کنیم ، پس آن عضو فقط توسط سایر اعضای همان کلاس قابل دسترسی است. اکنون می فهمید که چرا همیشه قبل از main() مشخصگر public قرار می گرفت. این روش توسط کدی خارج از برنامه یعنی توسط سیستم حین اجرای جاوا فراخوانی خواهد شد. اگر هیچ توصیفگر دسترسی استفاده نشده باشد ، آنگاه بصورت پیش فرض عضو یک کلاس ، داخل بسته مربوط به خود public است ، اما خارج از بسته مربوط به خود قابل دسترسی نمی باشد .

در کلاس‌هایی که تاکنون توسعه داده ایم، کلیه اعضای یک کلاس از حالت دسترسی پیش فرض که ضرورتاً "public" است، استفاده کرده اند. اما همیشه این حالت مطلوب شما نیست. معمولاً "مایلید تا دسترسی به اعضای داده ای یک کلاس را محدود نمایید و فقط از طریق برخی روشها امکان پذیر سازید". همچنین، شرایطی وجود دارند که مایلید روشهایی را که برای یک کلاس اختصاصی هستند، تعریف نمایید.

یک توصیفگر دسترسی قبل از سایر مشخصات نوع عضو یک کلاس قرار می گیرد. یعنی که این توصیفگر باید شروع کننده دستوراعلان یک عضو باشد، یک مثال را مشاهده می کنید:

```
public int i;  
private double j;  
private int myMethod(int a/ char b ){ //...
```

برای درک تأثیرات دسترسی عمومی (public) و اختصاصی (private) برنامه بعدی را در نظر بگیرید:

```
/* This program demonstrates the difference between  
public and private.  
*/  
class Test {  
    int a; // default access  
    public int b; // public access  
    private int c; // private access  
  
    // methods to access c  
    void setc(int i ){ // set c's value  
        c = i;  
    }  
    int getc () { // get c's value  
        return c;  
    }  
}  
  
class AccessTest {  
    public static void main(String args[] ){  
        Test ob = new Test();
```

```
// These are OK/ a and b may be accessed directly
ob.a = 10;
ob.b = 20;

// This is not OK and will cause an error
// ob.c = 100; // Error!

// You must access c through its methods
ob.setc(100); // OK

System.out.println("a/ b/ and c : " + ob.a + " " +
ob.b + " " + ob.getc());
}
}
```

همانطوریکه مشاهده می کنید، داخل کلاس **Test**، **a**، از دسترسی پیش فرض استفاده می کند که در این مثال مطابق مشخص نمودن **public** است **b**. بطور صریح بعنوان **public** مشخص شده است **c**. دارای دسترسی اختصاصی است. بدان معنی که این عضو توسط کدهای خارج از کلاس قابل دسترسی نخواهد بود. همچنین در داخل کلاس **Test Access**، نمیتوان **c** را بصورت مستقیم مورد استفاده قرار داد. این عضو را باید از طریق روش های عمومی اش یعنی **setc()** و **getc()** (مورد دسترسی قرار داد. اگر نشانه صحیح (comment) را ابتدای خط تغییر مکان دهید :

```
+ // ob.c = 100; // Error!
```

آنگاه نمی توانید این برنامه را کامپایل کنید که علت آن هم نقض دسترسی (**access violation**) است. برای اینکه با جنبه های عملی کنترل دسترسی در مثالهای عملی تر آشنا شوید روایت توسعه یافته از کلاس **stack** را در برنامه زیر مشاهده فرمایید :

```
// This class defines an integer stack that can hold 10 values.
class Stack {
/* Now/ both stck and tos are private .This means
that they cannot be accidentally or maliciously
altered in a way that would be harmful to the stack.
*/
private int stck[] = new int[10];
private int tos;

// Initialize top-of-stack
Stack (){
```



```
tos =- 1;
}
// Push an item onto the stack
void push(int item ){
if(tos==9)
System.out.println("Stack is full.");
else
stck[++tos] = item;
}
// Pop an item from the stack
int pop (){
if(tos < 0 ){
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}
```

همانطوریکه مشاهده می کنید ، هم **stck** که پشته را نگهداری می کند و هم **tos** که نمایه بالای پشته است بعنوان **private** مشخص شده اند . این بدان معنی است که آنها قابل دسترسی و جایگزینی جزئی از طریق **push()** و **pop()** نیستند . بعنوان مثال اختصاصی نمودن **tos** سایر بخشهای برنامه اتان را در مقابل قرار دادن غیر عمدی مقداری که فراتر از انتهای آرایه **stck** باشد ، محافظت می کند . برنامه بعدی نشاندهنده کلاس توسعه یافته **stack** است . سعی کنید خطوط غیر توضیحی را حرکت دهید تا بخود اثبات کنید که اعضای **stck** و **tos** در حقیقت غیر قابل دسترسی هستند .

```
class TestStack {
public static void main(String args[] ){
Stack mystack1 = new Stack();
Stack mystack2 = new Stack();

// push some numbers onto the stack
for(int i=0; i<10; i++)mystack1.push(i);
for(int i=10; i<20; i++) mystack2.push(i);

// pop those numbers off the stack
```

```
System.out.println("Stack in mystack1:");
for(int i=0; i<10; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<10; i++)
System.out.println(mystack2.pop());
// these statements are not legal
// mystack1.tos = - 2;
// mystack2.stck[3] = 100;
}
}
```

اگرچه روشها معمولاً دسترسی به داده های تعریف شده توسط یک کلاس را کنترل می کنند، اما همیشه هم اینطور نیست. کاملاً بجاست که هرگاه که دلیل خوبی برای اینکار داشته باشیم، اجازه دهیم تا یک متغیر نمونه **public** باشد. بعنوان مثال اکثر کلاسهای ساده با کمترین توجه نسبت به کنترل دسترسی به متغیرهای نمونه ایجاد شده اند و این بی توجهی فقط بلحاظ حفظ سادگی مثال بوده است.

محافظت دسترسی Access protection

قبلاً می دانستید که دسترسی به یک عضو **private** در یک کلاس فقط به سایر اعضای همان کلاس واگذار شده است. بسته ها بعد دیگری به کنترل دسترسی می افزایند. همانطوریکه خواهید دید، جاوا سطوح چندی از محافظت برای اجازه کنترل خوب طبقه بندی شده روی رویت پذیری متغیرها و روشهای داخل کلاسها، زیر کلاسها و بسته ها فراهم می نماید. کلاسها و بسته ها هر دو وسایلی برای کپسول سازی بوده و دربرگیرنده فضای نام و قلمرو متغیرها و روشها می باشند. بسته ها بعنوان ظروفی برای کلاسها و سایر بسته های تابعه هستند. کلاسها بعنوان ظروفی برای داده ها و کدها می باشند. کلاس کوچکترین واحد مجرد در جاوا است. بلحاظ نقش متقابل بین کلاسها و بسته ها، جاوا چهار طبقه بندی برای رویت پذیری اعضای کلاس مشخص کرده است:

1. زیر کلاسها در همان بسته
2. غیر زیر کلاسها در همان بسته
3. زیر کلاسها در بسته های مختلف
4. کلاسهایی که نه در همان بسته و نه در زیر کلاسها هستند.

سه مشخصگر دسترسی یعنی public ، private و protected و فراهم کننده طیف گوناگونی از شیوه های تولید سطوح چند گانه دسترسی مورد نیاز این طبقه بندیها هستند . جدول زیر این ارتباطات را یکجا نشان داده است .

private Nomodifier protected public |

همان کلاس Yes Yes Yes Yes

همان بسته زیر کلاس Yes Yes Yes No

همان بسته غیر زیر کلاس Yes Yes Yes No

بسته های مختلف زیر کلاس Yes Yes No No

بسته های مختلف غیر زیر کلاس Yes No No No

اگرچه مکانیسم کنترل دسترسی در جاوا ممکن است بنظر پیچیده باشد، اما میتوان آن را بصورت بعدی ساده گویی نمود . هر چیزی که بعنوان public اعلان شود از هر جایی قابل دسترسی است . هر چیزی که بعنوان private اعلان شود خارج از کلاس خودش قابل رویت نیست . وقتی یک عضو فاقد مشخصات دسترسی صریح و روشن باشد ، آن عضو برای زیر کلاسها و سایر کلاسهای موجود در همان بسته قابل رویت است . این دسترسی پیش فرض است . اگر می خواهید یک عضو ، خارج از بسته جاری و فقط به کلاسهای که مستقیما" از کلاس شما بصورت زیر کلاس درآمده اند قابل رویت باشد ، پس آن عضو را بعنوان protected اعلان نمایید . یک کلاس فقط دو سطح دسترسی ممکن دارد : پیش فرض و عمومی . (public) وقتی یک کلاس بعنوان public اعلان می شود ، توسط هر کد دیگری قابل دسترسی است . اگر یک کلاس دسترسی پیش فرض داشته باشد ، فقط توسط سایر کدهای داخل همان بسته قابل دسترسی خواهد بود .

یک مثال از دسترسی

مثال بعدی کلیه ترکیبات مربوط به اصلاحگرهای کنترل دسترسی را نشان می دهد . این مثال دارای دو بسته و پنج کلاس است . بیاد داشته باشید که کلاسهای مربوط به دو بسته متفاوت ، لازم است در دایرکتوریهای که بعداز بسته مربوطه اشان نام برده شده در این مثال p1 و p2 و ذخیره می شوند . منبع اولیه بسته سه کلاس تعریف می کند protection : ، Derived و samepackage و . اولین کلاس چهار متغیر int را در هر یک از حالات مختلف مجاز تعریف می کند. متغیر n با حفاظت پیش فرض اعلان شده است m-pri . بعنوان private ، n-pro ، بعنوان protected و n-pub و بعنوان public می باشند .

هر کلاس بعدی در این مثال سعی می کند به متغیرهایی در یک نمونه از یک کلاس دسترسی پیدا کند. خطوطی که بلحاظ محدودیتهای دسترسی، کامپایل نمی شوند با استفاده از توضیح یک خطی // از توضیح خارج شده اند. قبل از هر یک از این خطوط توضیحی قرار دارد که مکانهایی را که از آنجا این سطح از حفاظت اجازه دسترسی می یابد را فهرست می نماید.

دومین کلاس Derived یک زیر کلاس از protection در همان بسته p1 است. این مثال دسترسی Derived را به متغیری در protection برقرار می سازد بجز n-pri که یک private است.

سومین کلاس Samepackage یک زیر کلاس از protection نیست، اما در همان بسته قرار دارد و بنابراین به کلیه متغیرها بجز n-pri دسترسی خواهد داشت.

```
package p1;

public class Protection {
    int n = 1;
    private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;

    public Protection () {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}

class Derived extends Protection {
    Derived () {
        System.out.println("derived constructor");
        System.out.println("n = " + n);

        // class only
        // System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
    }
}
```

```
System.out.println("n_pub = " + n_pub);
}
}

class SamePackage {
    SamePackage () {
        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + p.n);

        // class only
        // System.out.println("n_pri = " + p.n_pri);

        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

اکنون کد منبع یک بسته دیگر یعنی p2 را مشاهده می کنید. دو کلاس تعریف شده در p2 دو شرایطی را که توسط کنترل دسترسی تحت تاثیر قرار گرفته اند را پوشش داده است. اولین کلاس یعنی protection 2 یک زیر کلاس p1.protection است. این کلاس دسترسی به کلیه متغیرهای مربوط به p1.protection را بدست می آورد غیر از n-pri چون private است) و n_ متغیری که با محافظت پیش فرض اعلان شده است. پیاد داشته باشید که پیش فرض فقط اجازه دسترسی از داخل کلاس یا بسته را می دهد نه از زیر کلاس های بسته های اضافی. در نهایت، کلاس otherpackage فقط به یک متغیر n-pub که بعنوان public اعلان شده بود دسترسی خواهد داشت.

```
package p2;

class Protection2 extends p1.Protection {
    Protection2 () {
        System.out.println("derived other package constructor");

        // class or package only
        System.out.println("n = " + n);

        // class only
        // System.out.println("n_pri = " + n_pri);
    }
}
```

```
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}

class OtherPackage {
    OtherPackage () {
        p1.Protection p = new p1.protection();
        System.out.println("other package contryctor");

        // class or package only
        System.out.println("n = " + p.n);

        // class only
        // System.out.println("n_pri = " + p.n_pri);

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

اگر مایلید تا این دو بسته را آزمایش کنید، در اینجا دو فایل آزمایشی وجود دارد که می توانید از آنها استفاده نمایید. یکی از این فایلها برای

بسته **p1** را در زیر نشان داده ایم :

```
// Demo package p1.
package p1;

// Instantiate the various classes in p1.
public class Demo {
    public static void main(String args[] ){
        Protection ob1 = new Protection();
        Derived ob2 = new Drived();
        SamePackage ob3 = new SamePackage();
    }
}
```

فایل آزمایشی برای **p2** بقرار زیر می باشد :

```
+ // Demo package p2.  
+ package p2;  
+ // Instantiate the various classes in p2.  
+ public class Demo {  
+ public static void main(String args[] ){  
+ Protection2 ob1 = new
```

خیلی خسته کننده است که هر بار یک نمونه ایجاد می شود کلیه متغیرهای یک کلاس را مقداردهی اولیه نماییم . حتی هنگامیکه توابع سهل الاستخدامه نظیر `setDim()` را اضافه می کنید ، بسیار ساده تر و دقیق تر آن است که کلیه تنظیمات در زمان ایجاد اولیه شیء انجام شود . چون نیاز به مقداردهی اولیه بسیار رایج است ، جاوا به اشیاء امکان می دهد تا در زمان ایجاد شدن خودشان را مقداردهی اولیه نمایند . این مقداردهی اولیه خودکار با استفاده از سازنده (`constructor`) انجام می گیرد . یک سازنده بمحض ایجاد یک شیء بلافاصله آن را مقداردهی اولیه می نماید . این سازنده نام همان کلاسی را که در آن قرار گرفته اختیار نموده و از نظر صرف و نحو مشابه یک روش است . وقتی یکبار سازنده ای را تعریف نمایید ، بطور خودکار بلافاصله پس از ایجاد یک شیء و قبل از اینکه عملگر `new` تکمیل شود ، فراخوانی خواهد شد . سازندگان کمی بنظر عجیب می آیند زیرا فاقد نوع برگشتی و حتی فاقد `void` هستند . بخاطر اینکه نوع مجازی برگشتی سازنده یک کلاس ، همان نوع خود کلاس می باشد . این وظیفه سازنده است که وضعیت داخلی یک شیء را بگونه ای مقداردهی اولیه نماید که کدی که یک نمونه را ایجاد می کند ، بلافاصله یک شیء کاملاً قابل استفاده و مقداردهی شده بوجود آورد . می توانید مثال مربوط به `Box` را طوری بازنویسی کنید که ابعاد یک `box` وقتی که یک شیء ساخته می شود ، بطور خودکار مقداردهی اولیه شوند . برای انجام این کار ، یک سازنده را جایگزین `setDim()` نمایید . کار را با تعریف یک سازنده ساده شروع می کنیم که بسادگی ابعاد هر `box` را بهمان مقادیر تنظیم میکند . این روایت جدید بقرار زیر است :

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

// This is the constructor for Box.
Box () {

System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}

// compute and return volume
double volume () {
```



```
return width * height * depth;
}
}

class BoxDemo6 {
public static void main(String args[] ){
// declare/ allocate/ and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();

System.out.println("volume is " + vol);

// get volume of second box
vol = mybox2.volume();
System.out.println("volume is " + vol);
}
}
```

پس از اجرای این برنامه ، خروجی آن بصورت زیر خواهد شد :

```
Constructing Box
Constructing Box
Volume is 1000
Volume is 1000
```

همانطوریکه مشاهده می کنید ، mybox1 و mybox2 هنگام ایجاد شدن بوسیله Box() مقداردهی اولیه میشوند. چون سازنده به همه boxها همان ابعاد یعنی 10x10x10 را می دهد ، هم mybox1 و هم mybox2 فضای اشغالی یکسانی خواهند داشت . دستور println() داخل Box() فقط بمنظور توصیف بهتر قرار گرفته است . اکثر توابع سازنده چیزی را نمایش نمی دهند. آنها فقط خیلی ساده ، اشیاء را مقدار دهی اولیه می کنند . قبل از ادامه بحث ، عملگر new را مجدداً بررسی میکنیم . همانطوریکه می دانید هنگامیکه یک شیء را اختصاص می دهید ، شکل عمومی زیر را مشاهده می کنید :

```
class-var = new classname();
```

اکنون می توانید بفهمید چرا پرانتزها بعد از نام کلاس مورد نیازند . آنچه واقعاً اتفاق می افتد این است که سازنده کلاس فراخوانی شده است .

بدین ترتیب در خط

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

newBox() سازنده Box() را فراخوانی می کند اگر یک سازنده برای کلاس تعریف نشود ، آنگاه جاوا یک سازنده پیش فرض برای کلاس ایجاد می کند . بهمین دلیل خط قبلی از کد در روایتهای اولیه Box که هنوز یک سازنده تعریف نشده بود ، کار می کرد. سازنده پیش فرض بطور خودکار کلیه متغیرهای نمونه را با عدد صفر، مقدار دهی اولیه می نماید . سازنده پیش فرض غالباً " برای کلاسهای ساده کفایت می کند اما برای کلاسهای پیچیده تر کفایت نمی کند . هر بار که سازنده خاص خود را تعریف کنید ، دیگر سازنده پیش فرض استفاده نخواهد شد .

سازندگان پارامتردار شده (parameterized)

هنگامیکه سازنده Box() در مثال قبلی یک شیء Box را مقدار دهی اولیه می کند چندان سودمند نیست ، زیرا کلیه box ها ، ابعاد یکسانی دارند . آنچه مورد نیاز است ، روشی برای ساخت اشیاء Box دارای ابعاد گوناگون است . ساده ترین راه افزودن پارامترها به سازنده است . همانطوریکه احتمالاً " حدس می زنید ، این عمل سودمندی سازندگان را افزایش می دهد . بعنوان مثال ، روایت بعدی Box یک سازنده پارامتردار شده را تعریف می کند که ابعاد یک box را همانطوریکه آن پارامترها مشخص شده اند ، تنظیم می کند. دقت کافی به چگونگی ایجاد اشیاء Box داشته باشید .

```
/* Here/ Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box {
double width;
double height;
double depth;

// This is the constructor for Box.
Box(double w/ double h/ double d ){
width = w;
height = h;
depth = d;
}

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

```
}  
  
class BoxDemo7 {  
public static void main(String args[] ){  
    // declare/ allocate/ and initialize Box objects  
    Box mybox1 = new Box(10/ 20/ 15);  
    Box mybox2 = new Box(3/ 6/ 9);  
  
    double vol;  
  
    // get volume of first box  
    vol = mybox1.volume();  
    System.out.println("volume is " + vol);  
  
    // get volume of second box  
    vol = mybox2.volume();  
    System.out.println("volume is " + vol);  
}  
}
```

خروجی این برنامه بقرار زیر است :

```
Volume is 3000  
Volume is 162
```

همانطوریکه می بینید ، هر شیء همانطوریکه در پارامترهای سازنده اش مشخص شده مقدار دهی اولیه خواهد شد .

```
Box mybox1 = new Box(10, 20, 15);
```

هنگامیکه **new** شیء را ایجاد می کند ، مقادیر 10 ، 20 ، 15 به سازنده Box () گذر می کنند

وراثت inheritance

وراثت را یکی از سنگ بناهای برنامه نویسی شی ئ گراست ، زیرا امکان ایجاد طبقه بندیهای سلسله مراتبی را بوجود می آورد . با استفاده از وراثت ، می توانید یک کلاس عمومی بسازید که ویژگیهای مشترک یک مجموعه اقلام بهم مرتبط را تعریف نماید . این کلاس بعداً " ممکن است توسط سایر کلاسها بارث برده شده و هر کلاس ارث برنده چیزهایی را که منحصر بفرد خودش باشد به آن اضافه نماید . در روش شناسی جاوا ، کلاسی که بارث برده می شود را کلاس بالا (superclass) می نامند . کلاسی که عمل ارث بری را انجام داده و ارث برده است را زیر کلاس (subclass) می نامند . بنابراین ، یک " زیر کلاس " روایت تخصصی تر و مشخص تر از یک " کلاس بالا " است .

زیر کلاس ، کلیه متغیرهای نمونه و روشهای توصیف شده توسط کلاس بالا را بارث برده و منحصر بفرد خود را نیز اضافه می کند .

مبانی وراثت

برای ارث بردن از یک کلاس ، خیلی ساده کافست تعریف یک کلاس را با استفاده از واژه کلیدی extends در کلاس دیگری قرار دهید . برای فهم کامل این مطلب ، مثال ساده ای را نشان می دهیم . برنامه بعدی یک کلاس بالا تحت نام A و یک زیر کلاس موسوم به B ایجاد می کند . دقت کنید که چگونه از واژه کلیدی extends استفاده شده تا یک زیر کلاس از A ایجاد شود .

```
// A simple example of inheritance.

// Create a superclass.
class A {
    int i, j;

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

// Create a subclass by extending class A.
class B extends A {
    int k;
    void showk (){
```

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

class SimpleInheritance {
public static void main(String args[] ){
A superOb = new A();
B subOb = new B();
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb :");
superOb.showij();
System.out.println();

/* The subclass has access to all public members of
its superclass .*/
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb :");
subOb.showj();
subOb.showk();
System.out.println();

System.out.println("Sum of i/ j and k in subOb:");
subOb.sum();
}
}
```

خروجی این برنامه ، بقرار زیر می باشد :

Contents of superOb:

i and j :10 20

Contents of subOb:

i and j :7 8

k :9

Sum of i/ j and k in subOb:

i+j+k :24

همانطوریکه می بینید ، زیر کلاس B دربرگیرنده کلیه اعضای کلاس بالای مربوطه یعنی A است . بهمین دلیل است که subob می تواند به `showij()` را فراخوانی نماید . همچنین داخل `sum()` می توان بطور مستقیم `i` و `j` و همانگونه که قبلاً بخشی از B بودند ، ارجاع نمود . اگرچه A کلاس بالای B می باشد ، اما همچنان یک کلاس کاملاً مستقل و متکی بخود است . کلاس بالا بودن برای یک زیر کلاس بدان معنی نیست که نمی توان خود آن کلاس بالا را بتهایی مورد استفاده قرار داد . بعلاوه ، یک زیر کلاس می تواند کلاس بالای یک زیر کلاس دیگر باشد . شکل عمومی اعلان یک `class` که از یک کلاس بالا ارث می برد ، بصورت زیر است :

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

برای هر زیر کلاسی که ایجاد می کنید ، فقط یک کلاس بالا می توانید تعریف کنید . جاوا از انتقال وراثت چندین کلاس بالا به یک کلاس منفرد پشتیبانی نمی کند . (از این نظر جاوا با C++ متفاوت است که در آن وراثت چند کلاسه امکان پذیر است) . قبلاً گفتیم که می توانید یک سلسله مراتب از وراثت ایجاد کنید که در آن یک زیر کلاس ، کلاس بالای یک زیر کلاس دیگر باشد . اما ، هیچ کلاسی نمی تواند کلاس بالای خودش باشد .

دسترسی به اعضای و وراثت

اگرچه یک زیر کلاس دربرگیرنده کلیه اعضا کلاس بالای خود می باشد ، اما نمیتواند به اعضای از کلاس بالا که بعنوان `private` اعلان شده اند ، دسترسی داشته باشد . بعنوان مثال ، سلسله مراتب ساده کلاس زیر را در نظر بگیرید :

```
/* In a class hierarchy/ private members remain  
private to their class.  
This program contains an error and will not  
compile.  
*/  
  
// Create a superclass.  
class A {  
int i; // public by default
```

```
private int j; // private to A
void setij(int x/ int y ){
    i = x;
    j = y;
}
}

// A's j is not accessible here.
class B extends A {
    int total;
    void sum (){
        total = i + j; // ERROR/ j is not accessible here
    }
}

class Access {
    public static void main(String args[] ){
        B subOb = new B();
        subOb.setij(10, 12);
        subOb.sum();
        System.out.println("Total is " + subOb.total);
    }
}
```

این برنامه کامپایل نخواهد شد زیرا ارجاع به `j` داخل روش `sum()` در `B` ر سبب نقض دسترسی خواهد شد. از آنجاییکه `j` بعنوان `private` اعلان شده، فقط توسط سایر اعضای کلاس خودش قابل دسترسی است و زیر کلاسها هیچگونه دسترسی به آن ندارند. یادآوری: یک عضو کلاس که بعنوان `private` اعلان شده برای کلاس خودش اختصاصی خواهد بود. این عضو برای کدهای خارج از کلاسش از جمله زیر کلاسها، قابل دسترسی نخواهد بود.

.

یک مثال عملی تر

اجازه دهید به یک مثال عملی تر پردازیم که قدرت واقعی وراثت را نشان خواهد داد. در اینجا، روایت نهایی کلاس `Box` بنحوی گسترش یافته تا یک عنصر چهارم تحت نام `weight` را دربرگیرد. بدین ترتیب، کلاس جدید شامل `width`، `height`، `depth` و `weight` و یک `box` خواهد بود.

```
// This program uses inheritance to extend Box.
class Box {
double width;
double height;
double depth;

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

// constructor used when all dimensions specified
Box(double w/ double h/ double d ){
width = w;
height = h;
depth = d;
}

// constructor used when all dimensions specified
Box (){
width =- 1; // use- 1 to indicate
height =- 1; // an uninitialized
depth =- 1; // box
}

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

// Here/ Box is extended to include weight.
class BoxWeight extends Box {
double weight; // weight of box

// constructor for BoxWeight
```



```
BoxWeight(double w/ double h/ double d/ double m ){
width = w;
height = h;
depth = d;
weight = m;
}

class DemoBoxWeight {
public static void main(String args[] ){
Boxweight mybox1 = new BoxWeight(10/ 20/ 15/ 34.3);
Boxweight mybox2 = new BoxWeight(2/ 3/ 4/ 0.076);
double vol;

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

vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
}
}
```

خروجی این برنامه بصورت زیر می باشد :

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

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

کلیه مشخصات **Box** را بارث برده و به آنها عنصر **weight** را اضافه می کند . برای **Boxweight** ضرورتی ندارد که کلیه جوانب موجود در **Box** را مجدداً "ایجاد نماید" . بلکه می تواند بسادگی **Box** را طوری گسترش دهد تا اهداف خاص خودش را تامین نماید . یک مزیت عمده وراثت این است که کافست فقط یکبار یک کلاس بالا ایجاد کنید که خصلتهای مشترک یک مجموعه از اشیاء را تعریف

نماید، آنگاه می توان از آن برای ایجاد هر تعداد از زیر کلاسهای مشخص تر استفاده نمود. هر زیر کلاس می تواند "با طبقه بندی خودش تطبیق یابد. بعنوان مثال، کلاس بعدی، از Box بارث برده و یک خصلت رنگ (color) نیز در آن اضافه شده است.

```
// Here/ Box is extended to include color.
class ColorBox extends Box {
int color; // color of box

ColorBox(double w/ double h/ double d/ double c ){
width = w;
height = h;
depth = d;
color = c;
}
}
```

بیاد آورید که هرگاه یک کلاس بالا ایجاد نمایید که وجوه عمومی یک شیء را تعریف کند، می توان از آن کلاس بالا برای تشکیل کلاسهای تخصصی تر ارث برد. هر زیر کلاس خیلی ساده فقط خصلتهای منحصر بفرد خودش را اضافه می کند. این مفهوم کلی وراثت است. یک متغیر کلاس بالا می تواند به یک شیء زیر کلاس ارجاع نماید یک متغیر ارجاع مربوط به یک کلاس بالا را می توان به ارجاعی، به هر یک از زیر کلاسهای مشتق شده از آن کلاس بالا، متناسب نمود. در بسیاری از شرایط، این جنبه از وراثت کاملاً مفید و سودمند است. بعنوان مثال، مورد زیر را در نظر بگیرید:

```
class RefDemo {
public static void main(String args[] ){
Boxweight weightbox = new BoxWeight(3/ 5/ 7/ 8.37);
Box plainbox = new Box();
double vol;

vol = weightbox.volume();
System.out.println("Volume of weightbox is " + vol);
System.out.println("Weight of weightbox is " +
weightbox.weight);
System.out.println();

// assign BoxWeight reference to Box reference
plainbox = weightbox;

vol = plainbox.volume(); // OK/ volume ()defined in Box
System.out.println("Volume of plainbox is " + vol);
}
```

```
/* The following statement is invalid because plainbox  
dose not define a weight member .*/  
  
// System.out.println("Weight of plainbox is " + plainbox.weight  
}  
}
```

در اینجا `weightbox` یک ارجاع به اشیای `Boxweight` است و `plainbox` یک ارجاع به اشیای `Box` است. از آنجاییکه `Boxweight` یک زیر کلاس از `Box` است، می توان `plainbox` را بعنوان یک ارجاع به شیء `weightbox` منتسب نمود. نکته مهم این است که نوع متغیر ارجاع و نه نوع شیئی که به آن ارجاع شده است که تعیین می کند کدام اعضائی قابل دسترسی هستند. یعنی هنگامیکه یک ارجاع مربوط به یک شیء زیر کلاس، به یک متغیر ارجاع کلاس بالا منتسب می شود، شما فقط به آن بخشهایی از شیء دسترسی دارید که توسط کلاس بالا تعریف شده باشند. بهمین دلیل است که `plainbox` نمی تواند به `weight` دسترسی داشته باشد حتی وقتی که به یک شیء `Boxweight` ارجاع می کند. اگر به آن فکر کنید، آن را احساس می کنید زیرا یک کلاس بالا آگاهی و احاطه ای نسبت به موارد اضافه شده به زیر کلاس مربوطه اش نخواهد داشت. بهمین دلیل است که آخرین خط از کد موجود در قطعه قبلی از توضیح رج شده است. برای یک ارجاع `Box` امکان ندارد تا به فیلد `weight` دسترسی داشته

ایجاد یک سلسله مراتب چند سطحی (Multilevel)

می توانید سلسله مراتبی بسازید که شامل چندین لایه وراثت بدلخواه شما باشند . کاملاً "موجه است که از یک زیر کلاس بعنوان کلاس بالای یک کلاس دیگر استفاده کنیم . بعنوان مثال اگر سه کلاس A ، B و C داشته باشیم آنگاه C می تواند یک زیر کلاس از B و B و یک زیر کلاس از A باشد . وقتی چنین شرایطی اتفاق می افتد ، هر زیر کلاس کلیه خصصتهای موجود در کلیه کلاس بالاهای خود را بارث می برد . در این شرایط ، C کلیه جنبه های B و A و را بارث می برد . در برنامه بعدی ، زیر کلاس Boxweight بعنوان یک کلاس بالا استفاده شده تا زیر کلاس تحت عنوان shipment را ایجاد نماید shipment . کلیه خصصتهای Boxweight و Box را به ارث برده و یک فیلد بنام cost به آن اضافه شده که هزینه کشتیرانی یک محموله را نگهداری می کند .

```
// Extend BoxWeight to include shipping costs.

// Start with Box.
class Box {
private double width;
private double height;
private double depth;

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

// constructor used when all dimensions specified
Box(double w, double h, double d ){
width = w;
height = h;
depth = d;
+ }
+
+ // constructor used when no dimensions specified
+ Box (){
+ width =- 1; // use- 1 to indicate
+ height =- 1; // an uninitialized
```

```
depth =- 1; // box
}

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

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

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

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

// constructor used when all parameters are specified
BoxWeight(double w/ double h/ double d/ double m ){
super(w, h, d); // call superclass constructor
weight = m;
}

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

// constructor used when cube is created
BoxWeight(double len/ double m ){
```

```
super(len);
weight = m;
}
}

// Add shipping costs
class Shipment extends BoxWeight {
double cost;

// construct clone of an object
Shipment(Shipment ob ){ // pass object to constructor
super(ob);
cost = ob.cost;
}

// constructor used when all parameters are specified
BoxWeight(double w, double h, double d,
double m, double c ){
super(w, h, d); // call superclass constructor
cost = c;
}

// default constructor
Shipment (){
super();
cost = -1;
}

// constructor used when cube is created
BoxWeight(double len, double m, double c ){
super(len, m);
cost = c;
}
}

class DemoShipment {
public static void main(String args[] ){
Shipment shipment1 = new Shipment(10, 20, 15, 10, 3.41);
Shipment shipment2 = new Shipment(2, 3, 4, 0.76, 1.28);
```

```
double vol;

vol = shipment1.volume();
System.out.println("Volume of shipment1 is " + vol);
System.out.println("Weight of shipment1 is " + shipment1.weight);
System.out.println("Shipping cost :$" + shipment1.cost);
System.out.println();

vol = shipment2.volume();
System.out.println("Volume of shipment2 is " + vol);
System.out.println("Weight of shipment2 is " + shipment2.weight);
System.out.println("Shipping cost :$" + shipment2.cost);
}
}
```

خروجی این برنامه بصورت زیر می باشد :

```
Volume of shipment1 is 3000
Weight of shipment1 is 10
Shipping cost :$3.41

Volume of shipment2 is 24
Weight of shipment2 is 0.76
Shipping cost :$1.28
```

بدلیل وراثت ، shipment می تواند از کلاسهای تعریف شده قبلی Box و Boxweight و استفاده نماید و فقط اطلاعات اضافی که برای کاربرد خاص خودش نیاز دارد ، اضافه نماید . این بخشی از ارزش وراثت است . وراثت امکان استفاده مجدد از کدهای قبلی را بخوبی بوجود آورده است . این مثال یک نکته مهم دیگر را نشان می دهد (super) : همواره به سازنده موجود در نزدیکترین کلاس بالا ارجاع می کند (super) . در shipment ر سازنده Boxweight را فراخوانی میکند (super) . در Boxweight ر سازنده موجود در Box را فراخوانی میکند . در یک سلسله مراتب کلاس ، اگر یک سازنده کلاس بالا نیازمند پارامترها باشد ، آنگاه کلیه زیر کلاسها باید آن پارامترها را بالای خط (up the line) بگذرانند. این امر چه یک زیر کلاس پارامترهای خودش را نیاز داشته باشد چه نیاز نداشته باشد ، صحت خواهد داشت .

نکته : در مثال قبلی ، کل سلسله مراتب کلاس ، شامل Box ، Boxweight ، و shipment و همگی در یک فایل نشان داده می شوند . این حالت فقط برای راحتی شما است . اما در جاوا ، هر یک از سه کلاس باید در فایلهای خاص خودش قرار گرفته و جداگانه کامپایل شوند . در حقیقت ، استفاده از فایلهای جداگانه یک می و نه یک استثنای در ایجاد سلسله مراتب کلاسهاست

وقتی که سازندگان فراخوانی می شوند constructors

وقتی یک سلسله مراتب کلاس ایجاد می شود ، سازندگان کلاسها که سلسله مراتب را تشکیل می دهند به چه ترتیبی فراخوانی می شوند ؟

بعنوان مثال ، با یک زیر کلاس تحت نام B و یک کلاس بالا تحت نام A ، آیا سازنده A قبل از سازنده B فراخوانی میشود، یا بالعکس ؟
پاسخ این است که در یک سلسله مراتب کلاس ، سازندگان بترتیب مشتق شدنشان از کلاس بالا به زیر کلاس فراخوانی می شوند . بعلاوه چون super() باید اولین دستوری باشد که در یک سازنده زیر کلاس اجرا می شود ، این ترتیب همانطور حفظ می شود ، خواه super() استفاده شود یا نشود . اگر super() استفاده نشود آنگاه سازنده پیش فرض یا سازنده بدون پارامتر هر یک از زیر کلاسها اجرا خواهند شد .
برنامه بعدی نشان می دهد که چه زمانی سازندگان اجرا می شوند :

```
// Demonstrate when constructors are called.

// Create a super class.
class A {
A () {
System.out.println("Inside A's constructor.")
}
}

// Create a subclass by extending class A.
class B extends A {
B () {
System.out.println("Inside B's constructor.")
}
}

// Create another subclass by extending B.
class C extends B {
C () {
System.out.println("Inside C's constructor.")
}
}

class CallingCons {
public static void main(String args[] ){
C c = new C();
}
}
```

خروجی این برنامه بشرح زیر می باشد :

Inside A's constructor

Inside B's constructor

Inside C's constructor

همانطوریکه مشاهده می کنید، سازندگان بترتیب مشتق شدنشان فراخوانی می شوند. اگر درباره آن تفکر کنید، می فهمید که توابع سازنده بترتیب مشتق شدنشان اجرا می شوند. چون یک کلاس بالا نسبت به زیر کلاسهای خود آگاهی ندارد، هر گونه مقدار دهی اولیه که برای اجرا شدن نیاز داشته باشد، جدا از و احتمالاً "پیش نیاز هر گونه مقدار دهی اولیه انجام شده توسط زیر کلاس بوده است."

استفاده از Super

در مثالهای قبلی کلاسهای مشتق شده از Box به کارایی و قدرتمندی که امکان داشت، پیاده سازی نشدند. بعنوان مثال، سازنده Boxweight بطور صریحی فیلدهای width، height، و depth و در Box() را مقدار دهی اولیه می کند. این امر نه تنها کدهای پیدا شده در کلاس بالای آنها را دو برابر می کند که غیر کاراست، بلکه دلالت دارد بر اینکه یک زیر کلاس باید دسترسی به این اعضا داشته باشد. اما شرایطی وجود دارند که می خواهید یک کلاس بالا ایجاد کنید که جزئیات پیاده سازی خودش را خودش نگهداری کند. در این شرایط، راهی برای یک زیر کلاس وجود ندارد تا مستقیماً به این متغیرهای مربوط به خودش دسترسی داشته و یا آنها را مقدار دهی اولیه نماید. از آنجاییکه کپسول سازی یک خصلت اولیه oop است، پس باعث تعجب نیست که جاوا راه حلی برای این مشکل فراهم کرده باشد. هرگاه لازم باشد تا یک زیر کلاس به کلاس بالای قبلی خودش ارجاع نماید، اینکار را با استفاده از واژه کلیدی super انجام می دهیم. super دو شکل عمومی دارد. اولین آن سازنده کلاس بالا را فراخوانی می کند. دومین آن بمنظور دسترسی به یک عضو کلاس بالا که توسط یک عضو زیر کلاس مخفی مانده است، استفاده می شود.

استفاده از super

یک زیر کلاس میتواند روش سازنده تعریف شده توسط کلاس بالای مربوطه را با استفاده از این شکل super فراخوانی نماید:

super(parameter-list);

در اینجا parameter-list مشخص کننده هر پارامتری است که توسط سازنده در کلاس بالا مورد نیاز باشد super(). باید همواره اولین

دستور اجرا شده داخل یک سازنده زیر کلاس باشد. بنگرید که چگونه از super() استفاده شده، و همچنین این روایت توسعه یافته از

کلاس Boxweight() را در نظر بگیرید:

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
double weight; // weight of box

// initialize width/ height/ and depth using super()
BoxWeight(double w/ double h/ double d/ double m ){
super(w, h, d); // call superclass constructor
weight = m;
}
}
```

در اینجا Boxweight() فراخوانی super() را با پارامترهای w، h، و d انجام می دهد. این کار سبب فراخوانده شدن سازنده Box() شده با استفاده از این مقادیر width، height، و depth و را مقدار دهی اولیه می کند. دیگر Boxweight خودش این مقادیر اولیه را مقدار دهی نمی کند. فقط کافی است تا مقدار منحصر بفرد خود weight: را مقدار دهی اولیه نماید. این عمل Box را آزاد می گذارد تا در صورت تمایل این مقادیر را private بسازد.

در مثال قبلی، super() با سه آرگومان فراخوانی شده بود. اما چون سازندگان ممکن است انباشته شوند، می توان super() را با استفاده از هر شکل تعریف شده توسط کلاس بالا فراخوانی نمود. سازنده ای که اجرا می شود، همانی است که با آرگومانها مطابقت داشته باشد. بعنوان مثال، در اینجا یک پیاده سازی کامل از Boxweight وجود دارد که سازندگان را برای طرق گوناگون و ممکن ساخته شدن یک box فراهم می نماید. در هر حالت super() با استفاده از آرگومانهای تقریبی فراخوانی میشود. دقت کنید که width و height و depth داخل Box بصورت اختصاصی درآمده اند.

```
// A complete implementation of BoxWeight.
class Box {
private double width;
private double height;
private double depth;

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

// constructor used when all dimensions specified
Box(double w/ double h/ double d ){
width = w;
height = h;
depth = d;
}

// constructor used when no dimensions specified
Box (){
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
```

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

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

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
double weight; // weight of box

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

// constructor used when all parameters are specified
Box(double w, double h, double d, double m ){
super(w, h, d); // call superclass constructor

weight = m;
}

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

// constructor used when cube is created
BoxWeight(double len, double m ){
super(len);
```

```
weight = m;
}
}

class DemoSuper {
public static void main(String args[] ){
BoxWeight mybox1 = new BoxWeight(10 ,20 ,15 ,34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
BoxWeight mybox3 = new BoxWeight(); // default
BoxWeight mycube = new BoxWeight(3, 2);
BoxWeight myclone = new BoxWeight(mybox1);
double vol;

vol = mybox1.vilume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();

vol = mybox2.vilume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
System.out.println();

vol = mybox3.vilume();
System.out.println("Volume of mybox3 is " + vol);
System.out.println("Weight of mybox3 is " + mybox3.weight);
System.out.println();

vol = myclone.vilume();
System.out.println("Volume of myclone is " + vol);
System.out.println("Weight of myclone is " + myclone.weight);
System.out.println();

vol = mycube.vilume();
System.out.println("Volume of mycube is " + vol);
System.out.println("Weight of mycube is " + mycube.weight);
System.out.println();
}
```

```
}
```

این برنامه خروجی زیر را تولید می کند :

Volume of mybox1 is 3000

Weight of mybox1 is 34.3

Volume of mybox2 is 24

Weight of mybox2 is 0.076

Volume of mybox3 is- 1

Weight of mybox3 is- 1

Volume of myclone is 3000

Weight of myclone is 34.3

Volume of mycube is 27

Weight of mycube is 2

توجه بیشتری نسبت به این سازنده در `Boxweight()` داشته باشید :

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

توجه کنید که `super()` با یک شی ناز نوع `Boxweight` نه از نوع `Box` فراخوانی شده است و نیز سازنده `Box (ob)` را فراخوانی می کند. همانطوریکه قبلاً ذکر شد، یک متغیر کلاس بالا را می توان برای ارجاع به هر شی ئ مشتق شده از آن کلاس مورد استفاده قرار داد. بنابراین، ما قادر بودیم یک شی ئ `Boxweight` را به سازنده `Box` گذر دهیم. البته `Box` فقط نسبت به اعضای خودش آگاهی دارد. اجازه دهید مفاهیم کلیدی مربوط به `super()` را مرور نماییم. وقتی یک زیرکلاس `super()` را فراخوانی می کند، در اصل سازنده کلاس بالای بلافاصله خود را فراخوانی می کند. بنابراین `super()` همواره به کلاس بالای بلافاصله قرار گرفته در بالای کلاس فراخوانده شده، ارجاع میکند. این امر حتی در یک سلسله مراتب چند سطحی هم صادق است. همچنین `super` باید همواره اولین دستوری باشد که داخل یک سازنده زیر کلاس اجرا می شود.

دومین کاربرد `super`

دومین شکل `super` تا حدودی شبیه `this` کار می کند، بجز اینکه `super` همواره به کلاس بالای زیر کلاسی که در آن استفاده می شود ، ارجاع می کند . شکل عمومی این کاربرد بصورت زیر است :

Super .member

در اینجا ، `member` ممکن است یک روش یا یک متغیر نمونه باشد . این دومین شکل `super` برای شرایطی کاربرد دارد که در آن اسامی اعضای یک زیر کلاس ، اعضای با همان اسامی را در کلاس بالا مخفی می سازند . این سلسله مراتب ساده کلاس را در نظر بگیرید :

```
// Using super to overcome name hiding.
class A {
    int i;
}

// Create a subclass by extending class A.
class B extends A {
    int i; // this i hides the in A

    B(int a,int b ){
        super.i = a; // i in A
        i = b; // i in B
    }

    void show (){
        System.out.println("i in superclass :" + super.i);
        System.out.println("i in subclass :" + i);
    }
}

class UseSuper {
    public static void main(String args[] ){
        B subOb = new B(1,2);

        subOb.show();
    }
}
```

این برنامه خروجی زیر را نمایش می دهد :

i in superclass :1

i in subclass :2

اگرچه متغیر نمونه i در B ر متغیر i در A را پنهان می سازد ، اما super امکان دسترسی به i تعریف شده در کلاس بالا بوجود می آورد .

همانطوریکه خواهید دید همچنین میتوان از super برای فراخوانی روشهایی که توسط یک زیر کلاس مخفی شده اند

روش finalize()

گاهی لازم است تا یک شیء هنگامیکه در حال خراب شدن است ، یک عمل خاصی را انجام دهد. بعنوان مثال ، ممکن است یک شیء دربرگیرنده منابع غیر جاوا نظیر یک دستگیره فایل (file handle) یا فونت کاراکتر ویندوز باشد، و می خواهید اطمینان یابید که قبل از اینکه آن شیء خراب شود ، منابع فوق آزاد شوند . برای اداره چنین شرایطی ، جاوا مکانیسمی تحت نام (finalization تمام کننده) فراهم آورده است . با استفاده از این مکانیسم ، می توانید عملیات مشخصی را تعریف نمایید که زمانیکه یک شیء در شرف مرمت شدن توسط جمع آوری زباله است ، اتفاق بیفتند . برای افزودن یک تمام کننده (finalizer) به یک کلاس ، خیلی راحت کافی است تا روش finalize() را تعریف نمایید. سیستم حین اجرای جاوا هرگاه در شرف چرخش مجدد یک شیء از کلاسی باشد ، این روش را فراخوانی می کند . داخل روش finalize() شما عملیاتی را مشخص می کنید که باید درست قبل از خرابی آن شیء انجام گیرند . جمع کننده زباله (garbage collector) بطور متناوب اجرا شده و بدنبال اشیایی میگردد که دیگر مورد ارجاع هیچیک از شرایط اجرایی نبوده و یا غیر مستقیم توسط سایر اشیاء ارجاع شده باشند . درست قبل از اینکه یک دارای رها شود، سیستم حین اجرای جاوا روش finalize() را روی شیء فراخوانی می کند . شکل عمومی روش finalize() بقرار زیر می باشد :

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

در اینجا واژه کلیدی protected توصیفگری است که از دسترسی به روش finalize() توسط کدهای تعریف شده خارج از کلاس جلوگیری می کند . مهم است بدانید که روش finalize() درست مقدم بر جمع آوری زباله فراخوانی می شود . بعنوان مثال وقتی یک شیء از قلمرو خارج می شود ، این روش فراخوانی نخواهد شد. این بدان معنی است که نمیتوانید تشخیص بدهید که چه زمانی finalize() اجرا خواهد شد و یا اصلاً اجرا نخواهد شد . بنابراین ، برنامه شما باید سایر وسائل برای آزاد کردن منابع سیستم مورد استفاده شیء را تدارک ببیند . برنامه نباید برای عملیات برنامه ای عادی ، متکی به روش finalize() باشد . نکته : اگر با ++C آشنایی دارید، پس می دانید که ++C مکان می دهد تا یک خراب کننده (destructor) برای یک کلاس تعریف نمایید ، که هرگاه یک شیء خارج از قلمرو قرار گیرد ، فراخوانی خواهد شد . جاوا چنین کاری نکرده و از این ایده پشتیبانی نمی کند . روش finalize() فقط به یک تابع خراب کننده نزدیک می شود. به مرور که تجربه بیشتری با جاوا کسب می کنید ، می بینید که نیاز به توابع خراب کننده بسیار کم است زیرا زیر سیستم جمع آوری حسن انجام می دهد .

توزیع (dispatch) پویای روش

اگر در لغو روشها چیزی فراتر از یک قرارداد فضای نام وجود نداشت، آنگاه این عمل در بهترین حالت، ارضای نوعی حس کنجکاوی و فاقد ارزش عملی بود. اما این چنین نیست. لغو روش تشکیل دهنده اساس یکی از مفاهیم پر قدرت در جاوا یعنی "توزیع پویای روش" است. این یک مکانیسم است که توسط آن یک فراخوانی به تابع لغو شده در حین اجرا (در عوض زمان کامپایل) از سر گرفته می شود. توزیع پویای روش مهم است چون طریقی است که جاوا با آن چند شکلی را درست حین اجرا پیاده سازی می نماید. توضیح را با تکرار یک اصل مهم شروع میکنیم: یک متغیر ارجاع کلاس بالا میتواند به یک شیء زیر کلاس ارجاع نماید. جاوا از این واقعیت استفاده کرده و فراخوانی به روشهای لغو شده را حین اجرا از سر می گیرد. وقتی یک روش لغو شده از طریق یک ارجاع کلاس بالا فراخوانی می شود، جاوا براساس نوع شیء ارجاع شده در زمانی که فراخوانی اتفاق می افتد، تعیین می کند که کدام روایت از روش باید اجرا شود. بنابراین، عمل تعیین روایت خاص از یک روش، حین اجرا انجام می گیرد. وقتی به انواع مختلف اشیان ارجاع شده باشد، روایتهای مختلفی از یک روش لغو شده فراخوانی خواهند شد. بعبارت دیگر، این نوع شیء ارجاع شده است (نه نوع متغیر ارجاع) که تعیین می کند کدام روایت از روش لغو شده باید اجرا شود. بنابراین اگر یک کلاس بالا دربرگیرنده یک روش لغو شده توسط یک زیر کلاس باشد، آنگاه زمانی که انواع مختلف اشیائی از طریق یک متغیر ارجاع کلاس بالا مورد ارجاع قرار می گیرند روایتهای مختلف آن روش اجرا خواهند شد. در اینجا مثالی را مشاهده میکنید که توزیع پویای روش را به شما نشان میدهد:

```
// Dynamic Method Dispatch
class A {
void callme () {
System.out.println("Inside A's callme method");
}
}

class B extends A {
// override callme()
void callme () {
System.out.println("Inside B's callme method");
}
}

class C extends A {
// override callme()
void callme () {
System.out.println("Inside C's callme method");
}
}
```

```

}

class Dispatch {
public static void main(String args[] ){
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A

r = a; // r refers to an A object
r.callme(); // calls A's version of callme

r = b; // r refers to a B object
r.callme(); // calls B's version of callme

r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}

```

خروجی این برنامه بقرار زیر می باشد :

Inside A's callme method
 Inside B's callme method
 Inside C's callme method

این برنامه یک کلاس بالای نام A و دو زیر کلاس آن تحت نام B و C و را ایجاد می کند. زیر کلاسهای B و C و سبب لغو
 callme() اعلان شده در A می گردند. درون روش main() اشیائی از نوع A و B و C و اعلان شده اند . همچنین یک ارجاع از نوع A
 بنام r اعلان شده است . سپس برنامه یک ارجاع به هر یک از انواع اشیائی به r را نسبت داده و از آن ارجاع برای فراخوانی callme()
 استفاده می کند . همانطوریکه حاصل این برنامه نشان می دهد ، روایتی از callme() که باید اجرا شود توسط نوع شیئی که در زمان
 فراخوانی مورد ارجاع قرار گرفته ، تعیین می شود . اگر این تعیین توسط نوع متغیر ارجاع یعنی r انجام میگرفت شما با سه فراخوانی به روش
 callme()
 مربوط به A مواجه می شدید . نکته : کسانی که با C++ آشنا هستند تشخیص می دهند که روشهای لغو شده در جاوا مشابه توابع مجازی
 (virtual functions) در C++ هستند

چرا روشهای لغو شده ؟

قبلا" هم گفتیم که روشهای لغو شده به جاوا اجازه پشتیبانی از چند شکلی حین اجرا را می دهند. چند شکلی به یک دلیل برای برنامه نویسی شیئی گرا لازم است: این حالت به یک کلاس عمومی اجازه می دهد تا روشهایی را مشخص نماید که برای کلیه مشتقات آن کلاس مشترک باشند، و به زیر کلاس ها اجازه می دهد تا پیاده سازیهای مشخص برخی یا کلیه روشها را تعریف نمایند. روشهای لغو شده راه دیگری برای جاوا است تا " یک ابط و چندین روش " را بعنوان یکی از وجوه چند شکلی پیاده سازی نماید .

بخشی از کلید کاربرد موفقیت آمیز چند شکلی، درک این نکته است که کلاس بالاها و زیر کلاسها یک سلسله مراتب تشکیل میدهند که از مشخصات کوچکتر به بزرگتر حرکت می کنند. اگر کلاس بالا بدرستی استفاده شود، کلیه اجزائی که یک زیر کلاس می تواند بطور مستقیم استفاده نماید، تعریف می کند. این امر به زیر کلاس قابلیت انعطاف تعریف روشهای خودش را می دهد، که همچنان یک رابط منسجم را بوجود می آورد .

بنابراین، بوسیله ترکیب وراثت با روشهای لغو شده، یک کلاس بالا می تواند شکل عمومی روشهایی را که توسط کلیه زیر کلاسهای مربوطه استفاده خواهند شد را تعریف نماید .

چند شکلی پویا و حین اجرا یکی از قدرتمندترین مکانیسمهایی است که طراحی شیئی گرایی را مجهز به استفاده مجدد و تنومندی کدها نموده است . این ابزار افزایش دهنده قدرت کتابخانه های کدهای موجود برای فراخوانی روشهای روی نمونه های کلاسهای جدید بدون نیاز به کامپایل مجدد می باشد در حالیکه یک رابط مجرد و زیبا را نیز حفظ می کنیم .

بکار بردن لغو روش

اجازه دهید تا به یک مثال عملی تر که از لغو روش استفاده می کند، نگاه کنیم. برنامه بعدی یک کلاس بالا تحت نام Figure را ایجاد می کند که ابعاد اشیائی مختلف دو بعدی را ذخیره می کند. این برنامه همچنین یک روش با نام area() را تعریف می کند که مساحت یک شیئی را محاسبه می کند. برنامه، دو زیر کلاس از Figure مشتق می کند. اولین آن Rectangle و دومین آن Triangle است. هر یک از این زیر کلاسها area() را طوری لغو میکنند که بترتیب مساحت یک مستطیل و مثلث را برگردان کنند .

```
// Using run-time polymorphism.
```

```
class Figure {  
    double dim1;  
    double dim2;
```

```
Figure(double a/ double b ){
dim1 = a;
dim2 = b;
}

double area (){
System.out.println("Area for Figure is undefined.");
return 0;
}
}

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

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

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

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

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

```
Rectangle r = new Rectangle(9/ 5);
Triangle t = new Triangle(10/ 8);

Figure figref;

figref = r;
System.out.println("Area is " + figref.area());

figref = t;
System.out.println("Area is " + figref.area());

figref = f;
System.out.println("Area is " + figref.area());
}
}
```

حاصل این برنامه بقرار زیر است :

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

از طریق مکانیسم دوگانه وراثت و چند شکلی حین اجرا، امکان تعریف یک رابط منسجم که برای چندین نوع اشیاء مختلف، اما بهم مرتبط، استفاده می شود، وجود دارد. در این حالت، اگر از **Figure** یک شیء مشتق شود، پس با فراخوانی **area()** می توان مساحت آن شیء را بدست آورد. رابط مربوط به این عملیات صرفنظر از نوع می باشد.

واژه کلیدی This

گاهی لازم است یک روش به شیء که آن را فراخوانی نموده ، ارجاع نماید . برای این منظور، جاوا واژه کلیدی **this** را تعریف کرده است . **this** . را می توان داخل هر روشی برای ارجاع به شیء جاری (current) استفاده نمود . یعنی **this** همواره ارجاعی است به شیء که روش روی آن فراخوانی شده است . می توانید از **this** هر جایی که ارجاع به یک شیء از نوع کلاس جاری مجاز باشد ، استفاده نمایید . برای اینکه بفهمید **this** به چه چیزی ارجاع می کند ، روایت بعدی **Box()** را در نظر بگیرید :

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

این روایت **Box()** دقیقاً مثل روایت اولیه کار می کند . استفاده از **this** به ندرت انجام گرفته اما کاملاً "صحت دارد" . داخل **Box()** همواره **this** به شیء فراخوانده شده ارجاع می کند . اگرچه در این مثالها بندرت پیش آمده می کند . اگرچه در این مثالها بندرت پیش آمده ، اما **this** در سایر متون بسیار مفید است .

مخفی نمودن متغیر نمونه

حتماً می دانید که اعلان دو متغیر محلی با یک نام داخل یک قلمرو یا قلمروهای بسته شده ، غیر مجاز است . بطرز خیلی جالبی ، می توانید متغیرهای محلی شامل پارامترهای رسمی برای روشها ، داشته باشید که با اسامی متغیرهای نمونه کلاسها مشترک باشند . اما ، هنگامیکه یک متغیر محلی همان اسم متغیر نمونه را داشته باشد، متغیر محلی ، متغیر نمونه را مخفی می سازد . بهمین دلیل بود که از **width height** و **depth** بعنوان اسامی پارامترهای سازنده **Box()** داخل کلاس **Box** استفاده نکردیم . اگر از آنها بهمین روش استفاده می کردیم ، آنگاه **width** به پارامتر رسمی ارجاع می کرد و متغیر نمونه **width** را مخفی می ساخت . اگرچه آسان تر است که از اسامی متفاوت استفاده کنیم ، اما راه حل دیگری برای چنین شرایطی وجود دارد . چون **this** امکان ارجاع مستقیم به شیء را به شما می دهد ، می توانید با استفاده از آن هر نوع تصادف و یکی شدن اسامی بین

متغیرهای نمونه و متغیرهای

محلی را رفع نمایید. بعنوان مثال، در اینجا روایت دیگری از Box() وجود دارد که از width، height، و depth و بعنوان اسامی

پارامترها استفاده نموده و آنگاه از this برای دستیابی به متغیرهای نمونه با همین اسامی استفاده کرده است:

```
// Use this to resolve name-space collisions.  
Box(double width/ double height/ double depth ){  
this.width = width;  
this.height = height;  
this.depth = depth;  
}
```

احتیاط. استفاده از this در چنین متنی ممکن است گاهی گیج کننده بشود و برخی از برنامه نویسان مراقب هستند تا از اسامی متغیرهای محلی و پارامترهای رسمی که متغیرهای نمونه را مخفی می سازند، استفاده نکنند. البته، سایر برنامه نویسان طور دیگری فکر میکنند. یعنی معتقدند استفاده از اسامی مشترک برای وضوح، ایده خوبی است و از this برای غلبه بر مخفی سازی متغیر نمونه بهره میگیرند. انتخاب یکی از دو روش با سلیقه شما ارتباط دارد. اگرچه this در مثالهایی که تاکنون نشان داده ایم ارزش زیادی نداشته، اما در واحد بود.

استفاده از کلاسهای مجرد (abstract)

شرایطی وجود دارد که میخواهید یک کلاس بالا تعریف نمایید که ساختار یک انتزاع معین را بدون یک پیاده سازی کامل از هر روشی، اعلان نماید. یعنی گاهی می خواهید یک کلاس بالا ایجاد کنید که فقط یک شکل عمومی شده را تعریف کند که توسط کلیه زیر کلاسهایش با اشتراک گذاشته خواهد شد و پر کردن جزئیات این شکل عمومی بعهده هر یک از زیر کلاس ها واگذار می شود. یک چنین کلاسی طبیعت روشهایی که زیر کلاسها باید پیاده سازی نمایند را تعریف می کند. یک شیوه برای وقوع این شرایط زمانی است که یک کلاس بالا توانایی ایجاد یک پیاده سازی با معنی برای یک روش را نداشته باشد. تعریف area() خیلی ساده یک نگهدارنده مکان (place holder) است. این روش مساحت انواع شیء را محاسبه نکرده و نمایش نمی دهد. هنگام ایجاد کتابخانه های خاص کلاس خود، خواهید دید که غیر معمول نیست اگر یک روش هیچ تعریف بامعنی در متن (context) کلاس بالای خود نداشته باشد. این شرایط را بدو طریق می توانید اداره نمایید. یک طریق این است که یک پیام هشدار (warning) گزارش نمایید. اگرچه این روش در برخی شرایط خاص مثل اشکال زدایی (debugging) فید است، اما روش دائمی نیست. ممکن است روشهایی داشته باشید که باید توسط زیر کلاس لغو شوند تا اینکه آن زیر کلاس معنادار بشود. کلاس Triangle را در نظر بگیرید. اگر area() تعریف نشود، این کلاس هیچ معنایی ندارد. در این حالت، شما بدنبال راهی هستید تا مطمئن شوید که یک زیر کلاس در حقیقت کلیه روشهای ضروری را لغو می کند. راه حل جاوا برای این مشکل روش مجرد (method) (abstract) است. می توانید توسط زیر کلاسها و با مشخص نمودن اصلاح کننده نوع abstract، روشهای معینی را لغو نمایید. به این روشها گاهی subclasser responsibility اطلاق میشود زیرا آنها هیچ پیاده سازی مشخص شده ای در کلاس بالا ندارند. بنابراین یک زیر کلاس باید آنها را لغو نماید چون نمی تواند بسادگی روایت تعریف شده در کلاس بالا را استفاده نماید. برای اعلان یک روش مجرد، از شکل عمومی زیر استفاده نمایید.

abstract type name(parameter-list);

همانطوریکه مشاهده می کنید در اینجا بدنه روش معرفی نشده است. هر کلاسی که دربرگیرنده یک یا چند روش مجرد باشد، باید بعنوان مجرد اعلان گردد. برای اعلان یک کلاس بعنوان مجرد، بسادگی از واژه کلیدی abstract در جلوی واژه کلیدی class در ابتدای اعلان کلاس استفاده می نمایید. برای یک کلاس مجرد هیچ شیئی نمی توان ایجاد نمود. یعنی یک کلاس مجرد نباید بطور مستقیم با عملکرد new نمونه سازی شود. چنان اشیائی بدون استفاده هستند، زیرا یک کلاس مجرد بطور کامل تعریف نشده است. همچنین نمی توانید سازندگان مجرد یا روشهای ایستای مجرد اعلان نمایید. هر زیر کلاس از یک کلاس مجرد باید یا کلیه روشهای مجرد موجود در کلاس بالا را پیاده سازی نماید، و یا خودش بعنوان یک abstract اعلان شود. در اینجا مثال ساده ای از یک کلاس با یک روش مجرد مشاهده می کنید که بعد از آن یک کلاس قرار گرفته که آن روش را پیاده سازی می کند:

```
// A Simple demonstration of abstract.
```

```
abstract class A {  
    abstract void callme();
```

```
// concrete methods are still allowed in abstract classes
void callmetoo (){
System.out.println("This is a concrete method.");
}
}

class B extends A {
void callme (){
System.out.println("B's implementetion of callme.");
}
}

class AbstractDemo {
public static void main(String args[] ){
B b = new B();

b.callme();

b.callmetoo();
}
}
```

توجه کنید که هیچ شیئی از کلاس A در برنامه اعلان نشده است. همانطوریکه ذکر شد، امکان نمونه سازی یک کلاس مجرد وجود ندارد. یک نکته دیگر: کلاس A یک روش واقعی با نام callmetoo() را پیاده سازی می کند. این امر کاملاً مقبول است. کلاسهای مجرد می توانند مادامیکه تناسب را حفظ نمایند، دربرگیرنده پیاده سازیها باشند.

اگرچه نمی توان از کلاسهای مجرد برای نمونه سازی اشیاء استفاده نمود، اما از آنها برای ایجاد ارجاعات شیئی می توان استفاده نمود زیرا روش جاوا برای چند شکلی حین اجرا از طریق استفاده از ارجاعات کلاس بالا پیاده سازی خواهد شد. بنابراین، باید امکان ایجاد یک ارجاع به یک کلاس مجرد وجود داشته باشد بطوریکه با استفاده از آن ارجاع به یک شیئی زیر کلاس اشاره نمود. شما استفاده از این جنبه را در مثال بعدی خواهید دید. با استفاده از یک کلاس مجرد، می توانید کلاس Figure را توسعه دهید. چون مفهوم با معنایی برای مساحت یک شکل دو بعدی تعریف نشده وجود ندارد، روایت بعدی این برنامه area() را بعنوان یک مجرد داخل Figure اعلان می کند. این البته بدان معنی است که کلیه کلاسهای مشتق شده از Figure باید area() را لغو نمایند.

```
// Using abstract methods and classes.
abstract class Figure {
    double dim1;
    double dim2;

    Figure(double a/ double b ){
        dim1 = a;
        dim2 = b;
    }

    // area is now an abstract method
    abstract double area();
}

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

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

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

    // override area for right triangle
    double area (){
        System.out.println("Inside Area for Teriangle.");
        return dim1 * dim2 / 2;
    }
}

class AbstractAreas {
```

```
public static void main(String args[] ){  
    // Figure f = new Figure(10/ 10); // illegal now  
    Rectangle r = new Rectanlge(9/ 5);  
    Triangle t = new Triangle(10/ 8);  
  
    Figure figref; // this is OK/ no object is created  
  
    figref = r;  
    System.out.println("Area is " + figref.area());  
  
    figref = t;  
    System.out.println("Area is " + figref.area());  
}  
}
```

همانطوریکه توضیح درون main() نشان می دهد ، دیگر امکان اعلان اشیاء از نوع Figure وجود ندارد چون اکنون بصورت مجرد است کلیه زیر کلاسهای Figure باید area() را لغو نمایند . برای اثبات این امر ، سعی کنید یک زیر کلاس ایجاد نمایید که area() را لغو نمی کند . حتما " یک خطای compile-time دریافت می کنید . اگرچه امکان ایجاد یک شی از نوع Figure وجود ندارد ، اما می توانید یک متغیر ارجاع از نوع Figure ایجاد نمایید . متغیر ??? بعنوان ارجاعی به Figure اعلان شده و بدان معنی است که با استفاده از آن می توان به یک شی از هر کلاس مشتق شده از Figure ، ارجاع نمود . همانطوریکه توضیح دادیم ، از طریق متغیرهای ارجاع

درک مفهوم static

شرایطی وجود دارد که مایلید یک عضو کلاس را طوری تعریف کنید که مستقل از هر شیء آن کلاس مورد استفاده قرار گیرد. بطور معمول یک عضو کلاس باید فقط همراه یک شیء از همان کلاس مورد دسترسی قرار گیرد. اما، این امکان وجود دارد که عضوی را ایجاد کنیم که توسط خودش استفاده شود، بدون اینکه به یک نمونه مشخص ارجاع نماید. برای ایجاد چنین عضوی، قبل از اعلان آن واژه کلیدی **static** را قرار دهید. وقتی یک عضو بعنوان **static** اعلان می شود، می توان قبل از ایجاد هر شیء آن کلاس، و بدون ارجاعی به هیچیک از اشیاء، آن را مورد استفاده قرار داد. می توانید هم روشها و هم متغیرها را بعنوان **static** اعلان نمایید. رایجترین مثال برای یک عضو **static**، همان **main()** است. بعنوان **static** اعلان می شود چون باید قبل از وجود هر نوع شیئی فراخوانی شود. متغیرهای نمونه اعلان شده بعنوان **static** ضرورتاً "متغیرهای سراسری هستند". هنگامیکه اشیاء کلاس آن اعلان می شوند، هیچ کپی از متغیر **static** ساخته نمی شود. در عوض، کلیه نمونه های آن کلاس همان متغیر **static** را به اشتراک می گذارند. روشهای اعلان شده بعنوان **static** دارای چندین محدودیت هستند: و آنها فقط سایر روشهای **static** را فراخوانی می کنند. و آنها فقط به داده های **static** دسترسی دارند. و آنها بهیچ وجه امکان ارجاع به **this** و **super** را ندارند. اگر لازم است محاسباتی انجام دهید تا متغیرهای **static** را مقدار دهی اولیه نمایید، می توانید یک بلوک **static** اعلان نمایید که فقط یکبار آنهم زمانی که کلاس برای اولین مرتبه بارگذاری می شود، اجرا گردد. مثال بعدی کلاسی را نشان میدهد که یک روش **static** برخی متغیرهای **static** و یک بلوک مقداردهی اولیه **static** دارد:

```
// Demonstrate static variables/ methods/ and blocks.
```

```
class UseStatic {
    static int a = 3;
    static int b;

    static void meth(int x){
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }

    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }
}
```

```
public static void main(String args[] ){  
meth(42);  
}  
}
```

بمحض اینکه کلاس `usestatic` بارگذاری شود، کلیه دستورات `static` اجرا میشوند. ابتدا `a` برابر 3 قرار گرفته، سپس بلوک `static` اجرا شده (یک پیام را چاپ می کند) و در نهایت مقدار `a*4` یا 12 در `b` بعنوان مقدار اولیه نهاده می شود. سپس `main()` فراخوانی می شود که `math()` را فراخوانی نموده و مقدار 42 را به `x` می گذراند. سه دستور `println()` به دو متغیر `static` یعنی `a` و `b` و همچنین به متغیر محلی `x` ارجاع می کنند. یادآوری: ارجاع به هر یک از متغیرهای نمونه داخل یک روش `static` غیرمجاز است. خروجی برنامه فوق بشرح زیر می باشد:

Static block initialized.

```
x = 42  
a = 3  
b = 12
```

خارج از کلاسی که تعریف شده اند، روشها و متغیرهای `static` را می توان مستقل از هر نوع شیء مورد استفاده قرار داد. برای انجام اینکار، فقط کافی است نام کلاس را با یک عملگر نقطه ای بعد از آن مشخص نمایید. بعنوان مثال، اگر بخواهید یک روش `static` را از خارج کلاس مربوطه فراخوانی کنید، با استفاده از شکل عمومی زیر اینکار را انجام می دهید:

`classname.method()`

در اینجا `classname` نام کلاسی است که روش `static` در آن اعلان شده است. همان طوری که می توانید ببینید، این شکل بندی مشابه همان است که برای فراخوانی روش های غیر `static` از طریق متغیرهای ارجاع شیء انجام میگرفت. یک متغیر `static` را نیز می توان با همان روش با استفاده از عملگر نقطه ای روی نام کلاس مورد دسترسی قرار داد. این روشی است که جاوا بوسیله آن یک روایت کنترل شده از توابع سراسری و متغیرهای سراسری را پیاده سازی می کند. در اینجا یک مثال وجود دارد. داخل `main()` روش `classname()` و متغیر `b` که `static` هستند در خارج از کلاسهای خود مورد دسترسی قرار می گیرند.

```
+ class StaticDemo {  
+ static int a = 42;  
+ static int b = 99;  
+ static void callme () {  
+ System.out.println("a = " + a);  
+ }  
+ }
```



```
+  
+ class StaticByName {  
+ public static void main(String args[] ){  
+ StaticDemo.callme();  
+ System.out.println("b + " + StaticDemo.b);  
+ }  
+ }
```

خروجی این برنامه بقرار زیر خواهد بود:

استفاده از اشیاء بعنوان پارامترها

تاکنون فقط از انواع ساده بعنوان پارامترهای روشها استفاده کرده ایم . اما هم صحیح و هم معمول است که اشیاء را نیز به روشها گذر دهیم .

بعنوان مثال برنامه ساده بعدی را در نظر بگیرید :

```
// Objects may be passed to methods.
class Test {
int a/ b;
Test(int , int j ){
    a= i;
    b = j;
}

// return true if o is equal to the invoking object
boolean equals(Test o ){
    if(o.a == a && o.b == b )return true;
    else return false;
}
}

class PassOb {
public static void main(String args[] ){
    Test ob1 = new Test(100, 22);

    Test ob2 = new Test(100, 22);
    Test ob3 = new Test(-1, - 1);

    System.out.println("ob1 == ob2 :"+ ob1.equals(ob2));
    System.out.println("ob1 == ob3 :"+ ob1.equals(ob3));
}
}
```

این برنامه ، خروجی بعدی را تولید می کند :

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

همانطوریکه مشاهده میکنید، روش equals() داخل Test دوشی ئ را از نظر کیفیت مقایسه نموده و نتیجه را برمی گرداند . یعنی این

روش ، شی ئ فراخواننده را با شیئی که گذر کرده مقایسه می کند . اگر محتوی آندو یکسان باشد ، آنگاه روش true را برمی گرداند . در

غیر اینصورت **false** را برمی گرداند . توجه داشته باشید که پارامتر **o** در **equals()** مشخص کننده **Test** بعنوان نوع آن می باشد. اگرچه **Test** یک نوع کلاس ایجاد شده توسط برنامه است ،اما بعنوان انواع توکار جاوا و بهمان روش مورد استفاده قرار گرفته است . یکی از رایجترین کاربردهای پارامترهای شیئی مربوط به سازندگان است . غالباً "ممکن است بخواهید یک شیء جدید را بسازید طوری که این شیء در ابتدا نظیر یک شیء موجود باشد. برای انجام اینکار باید یک تابع سازنده تعریف نمایید که یک شیء از کلاس خود را بعنوان یک پارامتر انتخاب می کند . بعنوان مثال ، روایت بعدی از **Box()** به یک شیء امکان داده تا آغازگر دیگری باشد :

```
// Here/ Box allows one object to initialize another.
```

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

```
// construct clone of an object
```

```
Box(Box ob ){ // pass object to constructor  
width = ob.width;  
height = ob.height;  
depth = ob.depth;  
}
```

```
// constructor used when all dimensions specified
```

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

```
// constructor used when no dimensions specified
```

```
Box (){  
width =- 1; // use- 1 to indicate  
height =- 1; // an uninitialized  
depth =- 1; // box  
}
```

```
// constructor used when cube is created
```

```
Box(double len ){  
width = height = depth  
}
```

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

class OverloadCons2 {
public static void main(String args[] ){
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);

Box myclone = new Box(mybox1);

double vol;

// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);

// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);

// get volume of cube
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);

// get volume of clone
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}
```

بعداً" خواهید دید که وقتی شروع به ایجاد کلاسهای خود می نمایید، معمولاً" برای اینکه اجازه دهیم تا اشیای بصورتی موثر و آسان ساخته شوند، لازم است اشکال های سازنده را فراهم آوریم .

استفاده از instanceof

گاهی خوب است که طی زمان اجرا، نوع شیء را بدانیم. بعنوان مثال، ممکن است یک نخ از اجرا داشته باشید که انواع گوناگونی از اشیاء تولید نموده و همچنین نخی که این اشیاء را پردازش می کند. در این شرایط، برای نخ پردازنده مفید است که نوع هر یک از اشیایی را که به آنها می رسد، بداند. شرایط دیگری که در آن دانستن نوع شیء در زمان اجرا مهم است، تبدیل casting می باشد. در جاوا یک تبدیل نامعتبر و غیر مجاز سبب بروز خطای حین اجرا می شود. بسیاری از تبدیلات غیر مجاز را در زمان کامپایل می توان گرفت. اما تبدیل cast که شامل سلسله مراتب کلاس باشد می تواند تبدیلات غیرمجاز تولید کند که فقط در زمان اجرا قابل کشف هستند. بعنوان مثال، یک کلاس بالا موسوم به A می تواند دو زیر کلاس B کنیم، یا یک شیء C را به نوع A. اما مجاز نیستیم یک شیء B را به نوع C یا بالعکس (تبدیل نماییم). از آنجاییکه یک شیء از نوع A می تواند به اشیاء B یا C ارجاع نماید، در زمان اجرا چگونه می توان فهمید که به چه نوع شیئی ارجاع شده است، قبل از اینکه تلاش برای تبدیل به نوع C را انجام دهیم؟ آن شیء ممکن است شیئی از نوع A و B و C و باشد. اگر از نوع B باشد، یک استثنای زمان اجرا پرتاب خواهد شد. جاوا عملگر حین اجرای instanceof را برای پاسخگویی به همین سوال تدارک دیده است شکل عمومی عملگر instanceof بقرار زیر می باشد :

object instanceof type

در اینجا، object یک نمونه از کلاس است و type یک نوع کلاس است. اگر object از نوع مشخصی باشد یا قابل تبدیل به یک نوع مشخص شده باشد، آنگاه عملگر instanceof مقدار true را نشان می دهد. در غیر اینصورت، منجر به false میگردد. بدین ترتیب، instanceof وسیله ای است که توسط آن برنامه اتان می تواند اطلاعات نوع درباره یک شیء در زمان اجرا را بدست آورد. برنامه بعدی نشان دهنده instanceof می باشد :

```
// Demonstrate instanceof operator.
class A {
    int i/ j;
}

class B {
    int i/ j;
}

class C extends A {
    int k;
}
```

```
class D extends A {
int k;
}
class InstanceOf {
public static void main(String args[] ){
A a = new A();
B b = new B();
C c = new C();
D d = new D();

if(a instanceof A)
System.out.println("a is instance of A");
if(b instanceof B)
System.out.println("b is instance of B");
if(c instanceof C)
System.out.println("c is instance of C");
if(c instanceof A)
System.out.println("c is instance of A");
if(a instanceof C)
System.out.println("a is instance of C");

System.out.println();

// compare types of derived types

A ob;

ob = d; // A reference to d
System.out.println("ob new refers to d");
if(ob instanceof D)
System.out.println("ob is instance of D");

System.out.println();

ob = c; // A reference to c
System.out.println("ob new refers to c");
if(ob instanceof D)
System.out.println("ob is instance of D");
```

```
else
System.out.println("ob is instance of D");

if(ob instanceof A)
System.out.println("ob is instance of A");

System.out.println();

// all object can be cast to Object
if(a instanceof object)
System.out.println("a may be cast to Object");
if(b instanceof object)
System.out.println("b may be cast to Object");
if(c instanceof object)
System.out.println("c may be cast to Object");
if(d instanceof object)
System.out.println("d may be cast to Object");
}
}
```

خروجی حاصل از این برنامه بصورت زیر می باشد :

```
a is instance of A
b is instance of B
c is instance of C
c can be cast to A

ob now refers to d
ob is instance of D

ob now refers to c
ob cannot be cast to D
ob can be cast to A

a may be cast to Object
b may be cast to Object
c may be cast to Object
d may be cast to Object
```

عملگر instanceof برای اکثر برنامه ها مورد نیاز نیست ، زیر معمولاً " شما نوع شیئی را که با آن کار می کنید ، می دانید . اما ، وقتی مشغول نوشتن روالهای عمومی شده هستید که با یک شیء از یک سلسله مراتب کلاس پیچیده عمل می کند ، این عملگر بسیار مفید خواهد بود .

کلاس Singleton

گاهی به کلاس هایی بر می خوریم که لزوماً باید یک و فقط یک متغیر از آنها تعریف شود مثلاً یک عامل یا شیئی که به یک منبع غیر قابل اشتراک دسترسی دارد اما هیچ چیزی نمی تواند شی را از تعریف متغیر دیگری از آن باز دارد پس چه می شود کرد الگوی تک برگ پاسخ به این پرسش است الگوی تک برگ با گرفتن وظیفه ایجاد و قطع دسترسی به متغیر در خود شی طرح را محدود می کند چنین کاری تضمین می کند که تنها یک کتغیر ایجاد شود و دسترسی به آن منفرد باشد.

پیاده سازی الگوی تک برگ :

```
/*
a class refrence to the singleton instance
*/
public class Singleton {
    private static Singleton instance;
    protected Singleton(){}
    public static Singleton getInstance(){
        if (instance== null) {
            instance= new Singleton();
        }
        return instance;
    }
}
```

کلاس Singleton یک متغیر Static از نوع Singleton دارد که دسترسی به آن را فقط به روال getInstance() محدود کرده است .

الگوی تک برگ چه موقع استفاده می شود ؟

وقتی بخواهیم در برنامه از یک کلاس خاص تنها یک متغیر داشته باشیم.

کلاس با الگوی شمارشی با نوع محافظت شده (Enum)

برخی زبانها مانند C++ دارای ساختار داده‌ای به نام نوع شمارشی (Enumeration) هستند که از این پس آنها را شمارشی خواهیم خواند. شمارشی‌ها در واقع فهرستی از ثوابت هستند اما این ثوابت محدود به شرایطی هستند مثلاً نمی‌توانند رفتار خاصی را در نظر بگیرند یا افزودن ثوابت جدید به آنها دشوار است. با تمام این اوصاف الگوی شمارشی راهی شی‌گرا برای تعریف ثوابت فراهم کرده است و به جای تعریف ثوابت صحیح (در C++) برای هر نوع ثابت کلاسی تعریف می‌کنیم.

```
public final class Size{
    // statically define valid values of Size
    public static final Size SMALL =new Size('S');
    public static final Size MEDIUM=new Size('M');
    public static final Size BIG      =new Size('B');
    // helps to iterator over enum values
    public static final Size [] SIZE={ SMALL , MEDIUM , BIG};
    // instance variable for holding onto display value
    private final char display;
    // do not allow instantiation by outside objects
    public Size(char value){
        display=value
    }
    public char getValue {
        return display;
    }
    public String toString(){
        return new String (display);
    }
}
```

کلاس Size ساده است این کلاس از نوع final تعریف شده است لذا هیچ کلاسی از آن نمی‌توان مشتق شود و این کلاس گروهی از ثوابت را تعریف می‌کند که ثابت‌ها اختصاصی تعریف شده است که نمی‌توان به طور مستقیم به آنها دسترسی پیدا کرد در عوض دسترسی به ثوابت تعریف شده کلاس ممکن خواهد بود

Size.MEDIUM

منابع :

<http://www.irandevolvers.com/>
<http://docs.sun.com>

نویسنده :

mamouri@ganjafzar.com محمد باقر معموری

ویراستار و نویسنده قسمت های تکمیلی :

zehs_sha@yahoo.com احسان شاه بختی

کتاب :

انتشارات نصی در 21 روز Java
برنامه نویسی شی گرا انتشارات نص