نگاهی دقیق تر به روشها و کلاسها

از این پس موضوعات مربوط به روشها، شامل انباشتن (over loading) ، گذر دادن پارامترها (parameter passing)و خود فراخوانی یا همان برگشت پذیری (recursion) را بررسی می کنیم . و درباره کنترل دسترسی ، استفاده از واژه کلیدی Static و یکی از مهمترین کلاسهای توکار جاوا یعنی string بحث خواهیم نمود.

معرفی روشها

موضوع روشها بسیار گسترده است ، چون جاوا قدرت و انعطاف پذیری زیادی به روشها داده است . شکل عمومی یک روش بقرار زیر است :

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

در اینجا ، typeمشخص کننده نوع داده برگشت شده توسط روش است . این نوع هر گونه نوع معتبر شامل نوع کلاسی که ایجاد کرده اید ، می تواند باشد . اگر روش مقداری را برنمی گرداند ، نوع برگشتی آن باید Void باشد . نام روش توسط name مشخص می شود . این نام می تواند هر نوع شناسه مجاز باشد ، البته غیر از آنهایی که قبلا" برای اقلام داخل قلمرو جاری استفاده شده باشند parameter-list . یک پس آیند نوع و شناسه است که توسط یک کاما جدا می شود . پارامترها ضرورتا "متغیرهایی هستند که مقدار آرگومان (arguments) گذرکرده در روش ، هنگام فراخوانی را جذب می کنند . اگر روش فاقد پارامتر باشد ، آنگاه فهرست پارامتر تهی خواهد بود . روشهایی که بجای void یک نوع برگشتی (return) دارند ، مقداری را به روال فراخواننده با استفاده از شکل بعدی دستور returnباز می گردانند .

```
return value;
```

در اینجا value همان مقدار برگشتی است .

افزودن یک روش به کلاس Box

اگرچه خیلی خوب است که کلاسی ایجاد نماییم که فقط شامل داده باشد، اما بندرت چنین حالتی پیش می آید. در اغلب اوقات از روشها برای دسترسی به متغیرهای نمونه تعریف شده توسط کلاس ، استفاده می کنیم . در حقیقت ، روشها توصیف گر رابط به اکثر کلاسها هستند . این امر به پیاده سازان کلاس امکان می دهد تا صفحه بندی مشخصی از ساختارهای داده داخلی را در پشت تجریدهای (abstractions)

روشهای زیباتر پنهان سازند . علاوه بر تعریف روشهایی که دسترسی به داده ها را بوجود می آورند ، می توانید روشهایی را تعریف کنید که بصورت داخلی و توسط خود کلاس مورد استفاده قرار می گیرند . اجازه دهید با اضافه کردن یک روش به کلاس Box شروع کنیم . ممکن است درمثالهای قبلی شما هم احساس کرده باشید که محاسبه فضای اشغالی (Volume) یک box چیزی است که توسط کلاس Box در مقایسه با کلاس BoxDemo بسیار راحت تر مدیریت می شود . بعلاوه ، چون فضای اشغالی یک box بستگی به اندازه آن دارد، بنظر می رسد که بهتر است کلاس Box این محاسبه را انجام دهد . برای انجام اینکار، باید بصورت زیر یک روش را به Box اضافه نمایید :

```
// This program includes a method inside the box class.

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

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

class BoxDemo3 {
public static void main(String args[] ){
Box mybox1 = new Box ();
Box mybox2 = new Box ();

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

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

// display volume of first box
```

```
mybox1.volume ();

// display volume of second box
mybox2.volume ();
}
}
```

این برنامه خروجی زیر را تولید می کند که مطابق خروجی روایت قبلی همین برنامه است

```
Volume is 3000

Volume is 162
```

با دقت به دو خط بعدی کدها نگاه کنید :

```
mybox1.volume ();
mybox2.volume ();
```

خط اول در اینجا ، روش volume() را روی mybox1 قرار می دهد. یعنی که این خط volume() را با استفاده از نام شی ئ که بدنبال آن عملگر نقطه ای قرار گرفته است نسبت به شی ئ mybox1 فراخوانی می کند . بدین ترتیب ، فراخوانی mybox1.volume()فضای اشغالیbox توصیف شده بوسیلهmybox1 را نمایش داده وفراخوانی mybox2.volume()فضای اشغالی box توصیف شده بوسیله mybox2 را نمایش می دهد. هر بار که volume() خواسته شود ، فضای اشغالی یک box مشخص را نمایش خواهد داد . اگر با مفهوم فراخوانی یک روش نا آشنا هستید ، توصیف بعدی موضوع را تا حد زیادی روشن می کند . هنگامیکه mybox1.volume()اجرا شود، سیستم حسن اجرای جاوا کنترل را به کدی که داخل volume() تعریف شده منتقل می کند . پس از اینکه دستور داخل volume() اجرا شود ، کنترل به روال فراخواننده برگشته و اجرای خط کد بعد از فراخوانی از سر گرفته خواهد شد . از یک نقطه نظر عمومی ، یک روش (method) شیوه جاوا برای پیاده سازی زیر روالهاست . (subroutions) یک نکته مهم درون روش volume() وجود دارد : ارجاع به متغیرهای نمونهwidth height وdepth و بصورت مستقیم بدون یک نام شی ئ یا یک عملگر نقطه ای پیش آیند) قبلی) انجام می گیرد. وقتی یک روش از متغیر نمونه ای که توسط کلاس خود تعریف شده استفاده می کند ، اینکار را بطور مستقیم و بدون ارجاع صریح به یک شی ئ و بدون استفاده از عملگر نقطه ای انجام می دهد . اگر درباره آن بیندیشید ، بخوبی درک می کنید . یک روش همواره نسبت به شی ئ از کلاس خود قرار داده می شود . وقتی این تعبیه اتفاق می افتد ، شی ئ شناخته شده است . بدین ترتیب ، داخل یک روش نیازی به مشخص کردن یک شی ئ برای بار دوم وجود ندارد . این بدان معنی است که width، height،وdepth و داخل volume() بطور صریحی به کپیهای متغیرهای پیدا شده در آن شی ئ که volume() را فعال می کند ، ارجاع می کنند . اجازه دهید مجددا" مرور کنیم : وقتی یک متغیر نمونه بوسیله کدی که بخشی از کلاسی که آن متغیر نمونه در آن

تعریف شده نیست مورد دسترسی قرار می گیرد اینکار باید از طریق یک شی ئ با استفاده از عملگر نقطه ای انجام شود. اما ، وقتی که متغیر نمونه بوسیله کدی که بخشی از همان کلاس مربوط به آن متغیر نمونه باشد مورد دسترسی قرار گیرد ، به آن متغیر می توان بطور مستقیم ارجاع نمود . همین مورد برای بررسی روشها نیز پیاده سازی می شود .

برگرداندن یک مقدار

در حالیکه پیاده سازی volume() محاسبه فضای اشغالی یک box داخل کلاس مربوطهBox را حرکت می دهد ، اما بهترین راه نیست . بعنوان مثال ، اگر بخش دیگری از برنامه شما بخواهد فضای اشغالی یک box را بداند ، اما مقدار آن را نمایش ندهد چکار باید کرد ? یک راه بهتر برای پیاده سازی volume() این است که آن را وادار کنیم تا فضای اشغالی box را محاسبه نموده و نتیجه را به فراخواننده (caller)برگرداند . مثال بعدی که روایت پیشرفته تر برنامه قبلی است ، همین کار را انجام می دهد :

```
// Now/ volume  ()returns the volume of a box.

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

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

class BoxDemo4 {
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;

// 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);
}
}
```

همانطوریکه می بینید ، وقتی volume() فراخوانی می شود ، در سمت راست یک دستور انتساب قرار می گیرد.در سمت چپ یک متغیر است که در این حالت vol میباشد که مقدار برگشتی توسط volume() را دریافت می کند . بدین ترتیب ، بعد از اجرای

```
vol = mybox.volume ();
```

مقدار mybox1.volume() برابر 3000 شده و سپس این مقدار در vol ذخیره خواهد شد . دو نکته مهم درباره برگرداندن مقادیر وجود دارد : ۞ نوع داده برگشت شده توسط یک روش باید با نوع برگشتی مشخص شده توسط همان روش سازگار باشد. بعنوان مثال ، اگر نوع برگشتی برخی روشها boolean باشد، نباید یک عدد صحیح را برگردان نمایید . ۞ متغیری که مقدار برگشتی توسط یک روش را دریافت می کند (نظیر vol در این مثال) باید همچنین با نوع برگشتی مشخص شده برای آن روش سازگاری داشته باشد . یک نکته دیگر : برنامه قبلی را می توان کمی کاراتر نیز نوشت زیرا هیچ نیاز واقعی برای متغیر vol وجود ندارد . فراخوانی volume() را می شد بطور مستقیم) بصورت زیر) در دستور println() استفاده نمود :

```
System.out.println("Volume is " + mybox1.volume90);
```

در این حالت ، هنگامیکه println() اجرا می شود ، mybox1.volume()بصورت خودکار فراخوانی شده و مقدار آن به println() گذر می کند .

افزودن روشی که پارامترها را می گیرد

اگرچه برخی روشها نیازی به پارامترها ندارند ، اما اکثر روشها این نیاز را دارند . پارامترها به یک روش ، امکان عمومی شدن را می دهند . یعنی که یک روش پارامتردار (parameterized) می تواند روی طیف گوناگونی از داده ها عمل کرده و یا در شرایط نسبتا" گوناگونی مورد استفاده قرار گیرد . برای مشاهده این نکته از یک برنامه خیلی ساده استفاده می کنیم . در اینجا روشی را مشاهده می کنید که مربع عدد 10 را برمی گرداند :

```
int square ()
{
return 10 * 10;
}
```

در حالیکه این روش در حقیقت مقدار مربع عدد 10 را برمی گرداند ، اما کاربرد آن بسیار محدود است . اما اگر روش را بگونه ای تغییر دهید که پارامتری را بگیرد همانطوریکه خواهید دید ، می توانید square() را مفیدتر بسازید .

```
int square(int i)
{
return i * i;
}
```

اکنون square() مربع هر مقداری را که فراخوانی شود ، برمی گرداند . یعنی که اکنون square() یک روش با هدف عمومی است که می تواند مربع هر مقدار عدد صحیح) بجای فقط عدد 10) را محاسبه می کند . مثالی را در زیر مشاهده می کنید :

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

در اولین فراخوانی square() مقدار 5 به پاارمتر i گذر می کند . در دومین فراخوانی ، iمقدار 9 را دریافت می کند . سومین فراخوانی مقدار y که در این مثال 2 است را می گذراند . همانطوریکه این مثالها نشان می دهند، square()قادر است مربع هر داده ای را که از آن می گذرد ، برگرداند . مهم است که دو اصطلاح پارامتر (parameter) و آرگومان (argument) را بوضوح درک نماییم . پارامتر ، متغیری است که توسط یک روش تعریف شده و هنگام فراخوانی روش ، مقداری را دریافت می کند .بعنوان مثال ، درsquare() ، iیک پارامتر است . آرگومان مقداری است که هنگام فراخوانی یک روش به آن گذر می کند . بعنوان مثال) ´100 square(مقدار100 را بعنوان یک آرگومان گذر میدهد. داخلsquare() ، پارامترi مقداری را دریافت می کند . می توانید یک روش پارامتردار را برای توسعه

کلاس Box مورداستفاده قرار دهید .در مثالهای قبلی ، ابعاد هر یک box باید بصورت جداگانه بـا اسـتفاده از یـک سلسـله دسـتورات نظیـر مورد زیر ، تعیین می شدند :

```
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

اگرچه این کد کاری می کند، اما بدو دلیل دارای اشکال است . اول اینکه ، این کد بد ترکیب و مستعد خطا است . بعنوان مثال ، خیلی ساده امکان دارد تعیین یک بعد فراموش شود . دوم اینکه در برنامه های خوب طراحی شده جاوا ، متغیرهای نمونه فقط از طریق روشهای تعریف شده توسط کلاس خودشان قابل دسترسی هستند . در آینده قادر خواهید بود تا رفتار یک روش را تغییر دهید ، اما نمی توانید رفتار یک متغیر نمونه بی حفاظ (افشا شده) را تغییر دهید . بنابراین ، یک راه بهتر برای تعیین ابعاد یک box این است که یک روش ایجاد نماییم که ابعاد boxرا در پارامترهای خود نگهداشته و هر متغیر نمونه را بصورت مناسبی تعیین نماید . این برنامه توسط برنامه زیر ، پیاده سازی خواهد شد :

```
// This program uses a parameterized method.

class Box {
double width;
double height;
double depth;
// compute and return volume
double volume  (){
return width * height * depth;
}
// sets dimensions of box
void setDim(double w, double h, double d ){
width = w;
height = h;
depth = d;
}
}

class BoxDemo5 {
public static void main(String args[] ){
Box mybox1 = new Box ();
Box mybox2 = new Box ();
```

```
double vol;

// initialize each box
mybox.setDim(10, 20, 15);
mybox.setDim(3, 6, 9);

// 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);
}
}
```

همانطوریکه می بینید ، روش setDim() برای تعیین ابعاد هر box مورد استفاده قرار می گیرد . بعنوان مثال وقتی

```
mybox1.setDim(10, 20, 15);
```

اجرا می شود ، عدد 10 به پارامتر w ، 20به h و 15به d کپی می شود . آنگاهheight وdepth و نسبت داده

نگاهی دقیق تر به گذر دادن آرگومانها

در کل ، دو راه وجود دارد تا یک زبان کامپیوتری یک آرگومان را به یک زیر روال گذر دهد. اولین راه " فراخوانی بوسیله مقدار-(call-by-value()است . این روش مقدار یک آرگومان را در پارامتر رسمی زیر روال کپی می کند . بنابراین تغییراتی که روی پارامتر زیر روال اعمال می شود ، تاثیری بر آرگومانی که برای فراخوانی آن استفاده شده نخواهد داشت . دومین راهی که یک آرگومان می تواند گذر کند " فراخوانی بوسیله ارجاع (call-by-reference()" است . در این روش ، ارجاع به یک آرگومان (نه مقدار آن آرگومان) به پارامتر گذر داده می شود . داخل زیر روال از این ارجاع برای دسترسی به آرگومان واقعی مشخص شده در فراخوانی استفاده می شود . این بدان معنی است که تغییرات اعمال شده روی پارامتر ، روی آرگومانی که برای فراخوانی زیر روال استفاده شده ، تاثیر خواهد داشت .خواهید دید که جاوا از هر دو روش برحسب اینکه چه چیزی گذر کرده باشد ، استفاده می کند . در جاوا ، وقتی یک نوع ساده را به یک روش گذر می دهید ، این نوع بوسیله مقدارش گذر میکند . بنابراین ، آنچه برای پارامتری که آرگومان را دریافت میکند اتفاق بیفتد هیچ تاثیری در خارج از روش نخواهد داشت . بعنوان مثال ، برنامه بعدی را در نظر بگیرید :

```
// Simple types are passed by value.
class Test {
void meth(int i, int j ){
i *= 2;
j /= 2;
}
}

class CallByValue {
public static void main(String args[] ){
Test ob = new Test ();
int a = 15, b = 20;
System.out.println("a and b before call :"  + a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call :" ++ a + " " + b);
}
}
```

خروجی برنامه فوق بقرار زیر می باشد :

```
a and b before call :15 20
a and b after call :15 20
```

بخوبی مشاهده می کنید که عملیات اتفاق افتاده داخل meth() هیچ تاثیری روی مقادیر a و b و در فراخوانی استفاده شده اند، نخواهد داشت . در اینجا مقادیر آنها به 30 و 10 تغییر نمی یابد . وقتی یک شی ئ را به یک روش گذر می دهید ، شرایط بطور مهیجی تغییر می کند زیرا اشیائ بوسیله ارجاعشان گذر داده می شوند . بیاد آورید که وقتی یک متغیر از یک نوع کلاس ایجاد می کنید ، شما فقط یک ارجاع به شی ئ خلق می کنید . بدین ترتیب ، وقتی این ارجاع را به یک روش گذر می دهید ، پارامتری که آن را دریافت می کند . بهمان شی ئ ارجاع می کند که توسط آرگومان به آن ارجاع شده بود . این بدان معنی است که اشیائ با استفاده از طریقه " فراخوانی بوسیله ارجاع " به روشها گذر داده می شوند. تغییرات اشیائ داخل روش سبب تغییر شیئی است که بعنوان یک آرگومان استفاده شده است . بعنوان مثال ، برنامه بعدی را در نظر بگیرید :

```
// Objects are passed by reference.

class Test {
int a, b;

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

// pass an object
void meth(Test o ){
o.a *= 2;
o.b /= 2;
}
}

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

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

برنامه فوق ، خروجی زیر را تولید می کند :

```
ob.a and ob.b before call :15 20
ob.a and ob.b after call :30 10
```

همانطوریکه می بینید ، در این حالت ، اعمال داخل meth() ، شیئی را که بعنوان یک آرگومان استفاده شده تحت تاثیر قرار داده است . یک نکته جالب توجه اینکه وقتی یک ارجاع شی ئ به یک روش گذر داده می شود، خود ارجاع از طریق " فراخوانی بوسیله مقدار " گذر داده می شود . اما چون مقداری که باید گذر داده شود خودش به یک شی ئ ارجاع می کند ، کپی آن مقدار همچنان به همان شی ئ ارجاع می کند که آرگومان مربوطه ارجاع می کند . یاد آوری : وقتی یک نوع ساده به یک روش گذر داده میشود اینکار توسط " فراخوانی بوسیله مقدار " انجام میگیرد .اشیائ توسط " فراخوانی بوسیله ارجاع " گذر داده می شوند

برگرداندن اشیائ

یک روش قادر است هر نوع داده شامل انواع کلاسی که ایجاد میکنید را برگرداند .بعنوان مثال ، در برنامه بعدی روش incrByTen() یک شی ئ را برمی گرداند که در آن مقدار a ده واحد بزرگتر از مقدار آن در شی ئ فراخواننده است .

```
// Returning an object.
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);
}
```

```
}
```

خروجی برنامه فوق بقرار زیر می باشد :

```
ob1.a :2
ob2.a :12
ob2.a after second increase :22
```

همانطوریکه مشاهده می کنید ، هر بار که incrByTen() فراخوانده می شود ، یک شی ئ جدید تولید شده و یک ارجاع به آن شی ئ جدید به روال فراخواننده برگردان می شود . مثال قبلی یک نکته مهم دیگر را نشان میدهد : از آنجاییکه کلیه اشیائ بصورت پویا و با استفاده از new تخصیص می یابند ، نگرانی راجع به شیئی که خارج از قلمرو برود نخواهید داشت ، زیرا در این صورت روشی که شی ئ درآن ایجاد شده پایان خواهد گرفت . یک شی ئ مادامیکه از جایی در برنامه شما ارجاعی به آن وجود داشته باشد ، زنده خواهد ماند . وقتی که ارجاعی به آن شی ئ وجود نداشته باشد ، دفعه مرمت خواهد شد.

خود فراخوانی یا برگشت پذیری Recursion

جاوا از خود فراخوانی پشتیبانی می کند . خود فراخوانی پردازشی است که در آن چیزی بر حسب خودش تعریف شود . در ارتباط با برنامه نویسی جاوا ، خود فراخوانی خصلتی است که به یک روش امکان فراخوانی خودش را می دهد . روشی که خودش را فراخوانی می کند موسوم به " خود فراخوانده " یا برگشت پذیر (recursive) است . مثال کلاسیک برای خود فراخوانی محاسبه فاکتوریل یک رقم است . فاکتوریل یک عدد N عبارت است از حاصلضرب کلیه اعداد از 1تاN ا . بعنوان مثال فاکتوریل 3 معادل 1 x2x3یا عدد 6 است . در زیر نشان داده ایم چگونه می توان با استفاده از یک روش خود فراخوان ، فاکتوریل را محاسبه نمود :

```
// A simple example of recursion.
 class Factorial {
 // this is a recursive function
 int fact(int n ){
 int result;

 if(n==1 )return 1;
 result = face(n-1 )* n;
 return result;
 }
 }

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

 System.out.println("Factorial of 3 is " + f.face(3));
 System.out.println("Factorial of 4 is " + f.face(4));
 System.out.println("Factorial of 5 is " + f.face(5));
 }
 }
```

خروجی این برنامه را در زیر نشان داده ایم :

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

اگر با روشهای خود فراخوان ناآشنا باشید ، آنگاه عملیات fact() ممکن است تا حدی بنظرتان گیج کننده باشد . طرز کار آن را توضیح داده ایم . وقتی fact() با یک آرگومان 1 فراخوانی می شود ، تابع مقدار 1 را برگردان می کند ، در غیر این صورت این تابع حاصلضرب

)n* fact(n-1)را برگردان میکند. برای ارزیابی این غبارت fact() را با n-1 فراخوانی می کنیم . این پردازش آنقدر تکرار می شود تا n مساوی 1باشد و فراخوانی های روش ، شروع به برگردان نمایند .

برای درک بهتر نحوه کار روش fact() اجازه دهید یک مثال کوتاه بیاوریم . وقتی فاکتوریل 3 را محاسبه می کنید، اولین فراخوانی fact()سبب دومین فراخوانی با آرگومان 2 می شود . این فراخوانی سبب می شود تا fact() برای سومین بار با آرگومان 1 فراخوانی شود . این فراخوانی مقدار 1 را برگردان می کند که بعدا" در)2 مقدار n در فراخوانی دوم) ضرب می شود. این نتیجه (یعنی عدد 2) آنگاه به فراخوانی اصلی fact() برگردان شده و در 3 (مقدار اصلی (n ضرب می شود . جواب کل عدد 6 است . اگر دستورات println() را در fact() جایگذاری نمایید بسیار جالب خواهد شد چون نشان خواهد داد که هر فراخوانی در چه سطحی است و جوابهای میانی چه مقادیری هستند . وقتی یک روش خودش را فراخوانی می کند ، حافظه پشته ها به متغیرهای محلی و پارامترهای جدید تخصیص داده می شود و کد روش نیز از همان اول با همین متغیرهای جدید اجرا می شود . یک خود فراخوانی ، کپی جدیدی از روش ایجاد نمی کند ، بلکه فقط آرگومانها جدید هستند . همچنانکه هر خود فراخوانی مقداری را برمی گرداند متغیرهای محلی و پارامترهای قدیمی از روی پشته برداشته می شوند ، و اجرا در نقطه ای از فراخوانی داخل روش از سر گرفته خواهد شد . روشهای خود فراخوان را می توان تلسکوپی نامید که باز و بسته می شوند . روایتهای خود فراخوانی بسیاری از روالها ، ممکن است کمی کندتر از روایتهای تکراری اجرا شوند و این بخاطر افزوده شدن انباشت فراخوانهای تابع اضافی است . بسیاری از خود فراخوانی ها به یک روش ممکن است سبب سر ریز شدن یک پشته شوند . از آنجاییکه ذخیره سازی پارامترها و متغیرهای محلی روی پشته انجام می گیرد و هر فراخوانی جدید یک کپی جدید از این متغیرها بوجود می آورد ، این امکان وجود دارد که پشته خراب شود . اگر چنین اتفاقی بیفتد ، سیستم حین اجرای جاوا یک استثنائ را بوجود می آورد. اما احتمالا" نگرانی درباره این مسائل نخواهید داشت مگر آنکه یک روال خود فراخوانی از حالت عادی خارج شود .

مهمترین مزیت روشهای خود فراخوان این است که از آنها برای ایجاد روایتهای ساده تر و روشنتر از الگوریتمهایی که می توان با رابطه های تکراری هم ایجاد نمود استفاده می شود . بعنوان مثال ، الگوریتم دسته بندی Quicksort در روش تکراری (iterative) بسیار بسختی پیاده سازی می شود . برخی مشکلات ، بخصوص مشکلات مربوط به Ai بنظر می رسد که نیازمند راه حلهای خود فراخوانی هستند. در نهایت ، اینکه بسیاری از مردم شیوه های خود فراخوانی را بهتر از شیوه های تکراری درک می کنند . هنگام نوشتن روشهای خود فراخوان ، باید یک دستور if داشته باشید که روش را مجبور کند تا بدون اجرای فراخوان خود فراخوان ، برگردان نماید . اگر اینکار را انجام ندهید ، هر بار که روش را فراخوانی کنید ، هرگز برگردان نخواهد کرد .

هنگام کار با خود فراخوانی ،این یکی از خطاهای رایج است .از دستورات println() هنگام توسعه برنامه بطور آزادانه استفاده کنید تا به شما نشان دهد چه چیزی در حال اتفاق افتادن است و اگر اشتباهی پیش آمده ، بتوانید اجرا را متوقف سازید .

دراینجا مثالی از خودفراخوانی را مشاهده کنید. روش خودفراخوانی printArray() اولین عنصر i در آرایه values را چاپ می کند :

```
// Another example that uses recursion.

class RecTest {
int values[];

RecTest(int i ){
values = new int[i];
}

// display array -- recursivaly
void printArray(int i ){
if(i==0 )return;
else printArray(i-1);
System.out.println("[" +( i-1 )+ "] " + values[i-1]);
}
}

class Recursion2 {
public static void mane(String args[] ){
RecTest ob = new RecTest(10);
int i;

for(i=0; i<10; i++ ) ob.values[i] = i;
ob.printArray(10);
}
}
```

این برنامه خروجی زیر را تولید می کند :

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
```

انباشتن روشها

در جاوا این امکان وجود دارد که دو یا چند روش را داخل یک کلاس که همان نام را دارد تعریف نمود ، البته مادامیکه اعلان پارامترهای آن روشها متفاوت باشد . در چنین شرایطی ، روشها را می گویند " انباشته شده " و این نوع پردازش را" انباشتن روش " می نامند . انباشتن روش یکی از راههایی است که جاوا بوسیله آن" چند شکلی " را پیاده سازی می کند . اگر تا بحال از زبانی که امکان انباشتن روشها را دارد استفاده نکرده اید ، این مفهوم در وهله اول بسیار عجیب بنظر می رسد . اما خواهید دید که انباشتن روش یکی از جنبه های هیجان انگیز و سودمند جاوا است . وقتی یک روش انباشته شده فراخوانی گردد ، جاوا از نوع و یا شماره آرگومانها بعنوان راهنمای تعیین روایت (Version) روش انباشته شده ای که واقعا" فراخوانی می شود ، استفاده می کند . بدین ترتیب ، روشهای انباشته شده باید در نوع و یا شماره پارامترهایشان متفاوت باشند . در حالیکه روشهای انباشته شده ممکن است انواع برگشتی متفاوتی داشته باشند ، اما نوع برگشتی بتنهایی برای تشخیص دو روایت از یک روش کافی نخواهد بود. وقتی جاوا با یک فراخوانی به یک روش انباشته شده مواجه می شود ، خیلی ساده روایتی از روش را اجرا می کند که پارامترهای آن با آرگومانهای استفاده شده در فراخوانی مطابقت داشته باشند . در اینجا یک مثال ساده وجود دارد که نشان دهنده انباشتن روش می باشد :

```
// Demonstrate method overloading.
class OverloadDemo {
void test  (){
System.out.println("No parameters");
}

// Overload test for one integer parameter.
void test(int a ){
System.out.println("a :" + a);
}

// Overload test for two integer parameters.
void test(int a, int b ){
System.out.println("a and b :" + a + " " + b);
}

// Overload test for a double parameter.
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;

// call all versions of test ()
ob.test ();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.2);
System.out.println("Result of ob.test(123.2 :)" + result);
}
}
```

این برنامه خروجی زیر را تولید می کند :

```
No parameters
a :10
a and b :10 20
double a :123.2
Result of ob.test(123.2 :)15178.2
```

همانطوریکه می بینید ، test()چهار بار انباشته شده است . اولین روایت پارامتری نمی گیرد ، دومین روایت یک پارامتر عدد صحیح می گیرد ، سومین روایت دو پارامتر عدد صحیح می گیرد و چهارمین روایت یک پارامتر double می گیرد . این حقیقت که چهارمین روایت test()همچنین مقداری را برمی گرداند، هرگز نتیجه حاصل از عمل انباشتن نیست ، چون انواع برگشتی نقشی در تجزیه و تحلیل انباشت ندارند .وقتی یک روش انباشته شده فراخوانی میشود،جاوا بدنبال تطبیقی بین آرگومانهای استفاده شده برای فراخوانی روش و پارامترهای آن روش می گردد . اما ، این تطابق نباید لزوما "همیشه صحیح باشد. در برخی شرایط تبدیل انواع خودکار جاوا میتواند نقشی در تجزیه و تحلیل انباشت داشته باشد . بعنوان مثال ، برنامه بعدی را در نظر بگیرید :

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
void test  (){
System.out.println("No parameters");
}

// Overload test for two integer parameters.
void test(int a, int b ){
```

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

// Overload test for a double parameter.
double test(double a ){
System.out.println("Inside test(double )a :" + a);
}
}

class Overload {
public static void main(String args[] ){
OverloadDemo ob = new OverloadDemo ();
int i = 88;

ob.test ();
ob.test(10, 20);

ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
}
}
```

این برنامه خروجی زیر را تولید می کندNo parameters :

```
a and b :10 20
Inside test(double )a :88
Inside test(double )a :123.2
```

همانطوریکه مشاهده میکنید، این روایت از overloadDemoتعریف کننده test(int) نمی باشد . بنابراین هنگامیکه test() همراه با یک آرگومان عدد صحیح داخلoverload فراخوانی می شود ، هیچ روش تطبیق دهنده پیدا نخواهد شد . اما جاوا می تواند بطور خودکار یک عدد صحیح را به یک double تبدیل نماید و این تبدیل برای رفع فراخوانی مورداستفاده قرار میگیرد .بنابراین ، بعد از آنکه test(int)پیدا نمی شود ، جاوا i را به double ارتقائ داده و آنگاه test(double) را فراخوانی می کند . البته اگر test(int) تعریف شده بود ، فراخوانی می شد . جاوا فقط در صورتی که هیچ تطبیق دقیقی پیدا نکند ، از تبدیل خودکار انواع استفاده می کند . انباشتن روش از چند شکلی هم پشتیبانی می کند زیرا یکی از شیوه هایی است که جاوا توسط آن الگوی " یک رابط و چندین روش " را پیاده سازی می کند . برای درک این مطلب ، مورد بعدی را درنظر بگیرید.در زبانهایی که از انباشتن روش پشتیبانی

نمی کنند ، هر روش باید یک اسم منحصر بفرد داشته باشد . اما غالبا" می خواهید یک روش را برای چندین نوع داده مختلف پیاده سازی نمایید . مثلا" تابع قدر مطلق را در نظر بگیرید . در زبانهایی که از انباشتن روش پشتیبانی نمی کنند معمولا "سه یا چند روایت مختلف از این تابع وجود دارد ، که هر یک اسم متفاوتی اختیار می کند . بعنوان نمونه ، در زبان C تابع abs() قدر مطلق یک عدد صحیح را برمی گرداند ، labs()قدر مطلق یک عدد صحیح long را برمی گرداند ، fabs()قدر مطلق یک عدد اعشاری را برمی گرداند . از آنجاییکه زبان C از انباشتن روش پشتیبانی نمی کند ، هر تابع باید اسم خاص خودش را داشته باشد ، حتی اگر هر سه تابع یک وظیفه واحد را انجام دهند. از نظر ذهنی این حالت ، شرایط پیچیده تری را نسبت به آنچه واقعا" وجود دارد ، ایجاب می کند . اگرچه مفهوم اصلی این توابع یکسان است ، اما همچنان مجبورید سه اسم را بخاطر بسپارید . این شرایط در جاوا اتفاق نمی افتد ، زیرا تمامی روش های مربوط به قدر مطلق می توانند از یک اسم واحد استفاده نمایند . در حقیقت کتابخانه کلاس استاندارد جاوا Java's standard) (class library شامل یک روش قدر مطلق موسوم به abs() می باشد . این روش توسط کلاس Math در جاوا انباشته شده تا کلیه انواع رقمی را مدیریت نماید . جاوا بر اساس نوع آرگومان ، تصمیم می گیرد که کدام روایت از abs() را فراخوانی نماید . ارزش انباشتن روشها در این است که می توان با استفاده از یک اسم مشترک به کلیه روشهای مرتبط با هم دسترسی پیدا کرد. بدین ترتیب ،اسم abs معرف عمل عمومی است که اجرا خواهد شد . تعیین روایت مخصوص برای هر یک از شرایط خاص بر عهده کامپایلر می باشد . برنامه نویس فقط کافی است تا اعمال عمومی که باید انجام شوند را بخاطر بسپارد . بدین ترتیب با استفاده از مفهوم چند شکلی ، چندین اسم به یک اسم خلاصه شده اند. اگرچه این مثال بسیارساده بود، امااگر مفهوم زیربنایی آن را گسترش دهید ، می فهمید که انباشتن روشها تا چه حد در مدیریت پیچیدگی در برنامه ها سودمند و کارساز است .

وقتی یک روش را انباشته می کنید ، هر یک از روایتهای آن روش قادرند هر نوع عمل مورد نظر شما را انجام دهند. هیچ قانونی مبنی بر اینکه روشهای انباشته شده باید با یکدیگر مرتبط باشند، وجود ندارد. اما از نقطه نظر روش شناسی ، انباشتن روشها مستلزم یک نوع ارتباط است . بدین ترتیب ، اگرچه می توانید از یک اسم مشترک برای انباشتن روشهای غیر مرتبط با هم استفاده نمایید ، ولی بهتر است این کار را انجام ندهید . بعنوان مثال ، می توانید از اسم sqr برای ایجاد روشهایی که مربع یک عدد صحیح و ریشه دوم عدد اعشاری را برمی گردانند ، استفاده نمایید . اما این دو عمل کاملا" با یکدیگر متفاوتند. بکارگیری انباشتن روش درچنین مواقعی سبب از دست رفتن هدف اصلی این کار خواهد شد. در عمل ، فقط عملیات کاملا" نزدیک بهم را انباشته می کنید .

انباشتن سازندگان Overloading constructors

علاوه بر انباشتن روشهای معمولی ، می توان روشهای سازنده را نیز انباشته نمود. در حقیقت برای اکثر کلاسهایی که در دنیای واقعی ایجاد می کنید ، سازندگان انباشته شده بجای استثنائ یک عادت هستند . در زیر آخرین روایت Box را مشاهده می کنید :

```
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;
}
}
```

همانطوریکه می بینید ، سازنده Box() نیازمند سه پارامتر است . یعنی کلیه اعلانات اشیائ Box باید سه آرگومان به سازنده Box() بگذرانند . بعنوان مثال دستور بعدی فعلا" نامعتبر است;() Box ob = new Box :

ازآنجاییکه Box() نیازمند سه آرگومان است ، فراخوانی آن بدون آرگومانها خطا است . این مورد، سوالات مهمی را پیش رو قرار می دهد. اگر فقط یک box را بخواهید و اهمیتی نمی دهید و یا نمی دانید که ابعاد اولیه آن چه بوده اند ، چه پیش می آید ? همچنین ممکن است بخواهید یک مکعب را با مشخص کردن یک مقدار که برای کلیه ابعاد آن استفاده می شوند، مقدار دهی اولیه نمایید ? آنگونه که قبلا" کلاس Boxنوشته شده ، این گزینه ها در دسترس شما نخواهد بود . خوشبختانه راه حل این مشکلات کاملا" ساده است : خیلی ساده تابع سازنده Boxرا انباشته کنید بگونه ای که شرایط توصیف شده را اداره نماید . برنامه بعدی شامل یک روایت توسعه یافته Box است که اینکار را انجام می دهد :

```
/* Here/ Box defines three constructors to initialize
the dimensions of a box various ways.
*/
class Box {
double width;
double height;
double 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 = deoth = len;
}

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

class OverloadDemo {
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);

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 mycube is " + vol);
}
}
```

خروجی این برنامه بقرار زیر می باشد :

```
Volume of mybox1 is 3000
Volume of mybox2 is- 1
Volume of mycube is 343
```

neاجرا می شود ، پارامترها را مشخص

لغو (یا جلوگیری از پیشروی) روش

در یک سلسله مراتب کلاس ، وقتی یک روش در یک زیر کلاس همان نام و نوع یک روش موجود در کلاس بالای خود را داشته باشد، آنگاه میگویند آن روش در زیر کلاس ، روش موجود در کلاس بالا را لغو نموده) یا از پیشروی آن جلوگیری می نماید) . وقتی یک روش لغو شده از داخل یک زیر کلاس فراخوانی می شود ، همواره به روایتی از آن روش که توسط زیر کلاس تعریف شده ، ارجاع خواهد نمود و روایتی که کلاس بالا از همان روش تعریف نموده ، پنهان خواهد شد . مورد زیر را در نظر بگیرید :

```
// Method overriding.
 class A {
 int i, j;

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

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

 class B extends A {
 int k;

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

 // display k -- this overrides show  ()in A
 void show  (){
 System.out.println("k :" + k);
 }
```

```
}

class Override {
public static void main(String args[] ){
B subOb = new B(1, 2, 3);

subOb.show (); // this calls show  ()in B
}
}
```

حاصل تولید شده توسط این برنامه بقرار زیر می باشد :

```
k:3
```

وقتی show() روی یک شی ئ از نوع B فراخوانی می شود ، روایتی از show که داخل B تعریف شده مورد استفاده قرار میگیرد. یعنی که ، روایت show() داخل B ، روایت اعلان شده در A را لغو می کند . اگر می خواهید به روایت کلاس بالای یک تابع لغو شده دسترسی داشته باشید ، این کار را با استفاده از super انجام دهید . بعنوان مثال ، در این روایت از B روایت کلاس بالای show() داخل روایت مربوط به زیر کلاس فراخوانی خواهد شد . این امر به کلیه متغیرهای نمونه اجازه می دهد تا بنمایش درآیند .

```
class B extends A {
int k;

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

void show  (){
super.show (); // this calls A's show ()
System.out.println("k :" + k);
}
}
```

اگر این روایت از A را در برنامه قبلی جایگزین نمایید، خروجی زیر را مشاهده می کنید :

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

در اینجا ، super.show()روایت کلاس بالای show() را فراخوانی می کند . لغو روش فقط زمانی اتفاق می افتد که اسامی و نوع دو روش یکسان باشند . اگر چنین نباشد ، آنگاه دو روش خیلی ساده انباشته (overioaded) خواهند شد . بعنوان مثال ، این روایت اصلاح شده مثال قبلی را در نظر بگیرید :

```
// Methods with differing type signatures are overloaded -- not
// overridden.
class A {
int i, j;

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

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

// Create a subclass by extending class A.
class B extends A {
int k;

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

// overload show ()
void show(String msg ){
System.out.println(msg + k);
}
}

class Override {
public static void main(String args[] ){
B subOb = new B(1, 2, 3);
```

```
subOb.show("This is k :"); // this calls show  ()in B
subOb.show (); // this calls show  ()in A
}
}
```

حاصل تولید شده توسط این برنامه بقرار زیر می باشد :

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

روایتshow() درB ر یک پارامتر رشته ای می گیرد . این عمل سبب متفاوت شدن تاییدیه نوع آن از نـوع موجـود در A شـده ، کـه هـیچ پارامتری را نمی گیرد. بنابراین نباشتگی (یا مخفی شدن اسم (اتفاق نمی افتد .

روشهای Native

اگر چه بندرت اتفاق می افتد، اما گهگاه شرایطی پیش می آید که می خواهید یک زیر روال (subroutine) نوشته شده توسط سایر زبانهای غیر از جاوا را فراخوانی نمایید . معمولا" ، چنین زیر روالی بصورت کد قابل اجرا برای cpu و محیط خاصی که در آن کار می کنید عرضه می شود یعنی بصورت کد بومی . (Native) بعنوان مثال ممکن است بخواهید یک زیر روال کد بومی را فراخوانی کنید تا به زمان اجرای سریعتری برسید. یا ممکن است بخواهید از یک کتابخانه تخصصی شده متفرقه نظیر یک بسته آماری استفاده نمایید . اما از آنجاییکه برنامه های جاوا به کد بایتی کامپایل می شوند و سپس توسط سیستم حین اجرای جاوا تفسیر خواهند شد ، بنابراین بنظر می رسد فراخوانی یک زیر روال کد بومی از داخل برنامه جاوا غیر ممکن باشد .خوشبختانه ، این نتیجه گیری غلط است . جاوا واژه کلیدی native را تدارک دیده که برای اعلان روشهای کدهای بومی استفاده می شود. هربار که این روشها اعلان شوند می توانید آنها را از درون برنامه جاوای خود فراخوانی نمایید . درست مثل فراخوانی هر روش دیگری در جاوا . برای اعلان یک روش بومی ، اصلاحگر nativeرا قبل از روش قرار دهید. اما بدنه ای برای روش تعریف نکنید ، بعنوان مثال :

```
public native int meth ();
```

هر بار که یک روش بومی را اعلان نمودید ، مجبورید روش بومی نوشته و یکسری مراحل پیچیده تر را تعقیب کنید تا آن را به کد جاوای خود پیوند دهید . نکته : مراحل دقیقی که لازم است طی نمایید ممکن است برای محیط ها و روایتهای گوناگون جاوا متفاوت باشند . همچنین زبانی که برای پیاده سازی روش بومی استفاده می کنید ، موثر خواهد بود . بحث بعدی از JDK ، (version 1.02)و ابزارهای آن استفاده میکند. این یک محیط windows 95/NT را فرض میکند .زبانی که برای پیاده سازی روش بومی استفاده شده ، زبان C می باشد . آسانترین شیوه برای درک این پردازش ، انجام یک کار عملی است . برای شروع برنامه کوتاه زیر را که از یک روش native موسوم به test() استفاده می کند وارد نمایید :

```
// A simple example that uses a native method.
public class NativeDemo {
int i;
int j;

public static void main(String args[] ){
NativeDemo ob = new NativeDemo ();

ob.i = 10;
ob.j = ob.test (); // call a native method
System.out.println("This is ob.j :" + ob.j);
}
```

```
// declare native method
public native int test ();

// load DLL that contains static method
static {
System.loadLibrary("NativeDemo");
}
}
```

دقت کنید که روش test() بعنوان native اعلان شده و بدنه ای ندارد . این روشی است که ما در C باختصار پیاده سازی میکنیم . همچنین به بلوک static دقت نمایید .همانطوریکه قبلا" گفتیم : یک بلوک static فقط یکبار اجرا می شود ، و آنهم زمانی است که برنامه شما شروع باجرا می نماید (یا دقیق تر بگویم ، وقتی که کلاس آن برای اولین بار بارگذاری می شود) . در این حالت ، از این بلوک استفاده شده تا" کتابخانه پیوند پویا (dynamic Link Library)" را که دربرگیرنده پیاده سازی بومی test() است ، بارگذاری نماید . کتابخانه فوق توسط روش LoadLibrary() بارگذاری مسشود که بخشی از کلاس System است . شکل عمومی آن بقرار زیر است :

Static void LoadLibrary(string filename)

در اینجا ، filenameرشته ای است که نام فایلی که کتابخانه را نگهداری میکند توصیف می کند . برای محیط ویندوز NT/95 فرض شده که این فایل پسوند DLL ، داشته باشد . بعد از اینکه برنامه را وارد کردید، آن را کامپایل کنید تا NativeDemo.classتولید شود. سپس ، باید ازjava.exeاستفاده نموده تا دو فایل تولید نمایید:

که NativeDemo.Cو NativeDemo.1 و است . شما NativeDemo.h را در پیاده سازی test() می گنجانید . فایل NativeDemo.Cیک فایل stub است که دربرگیرنده میزان کوچکی از کد است که رابط بین روش بومی شما و سیستم حین اجرای جاوا را تدارک می بیند .برای تولید NativeDemo.h ، از دستور بعدی استفاده نماییدjavahNativeDemo : این دستور یک فایل سرآیند موسوم به NativeDemo.h را ایجاد می کند. این فایل باید در فایل C که test() را پیاده سازی می کند ، گنجانده شود. حاصل تولید شده توسط این فرمان بقرار زیر می باشد :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include
/* Header for class NativeDemo */

#ifndef_ Included_NativeDemo
#define_ Included_NativeDemo
```

```
typedef struct ClassNativeDemo {
long j;
long j;
} ClassNativeDemo;
HandleTo(NativeDemo);

#ifdef __ cplusplus
extern "C" {
#endif
extern long NativeDemo_test(struct HNativeDemo *);
#ifdef __ cplusplus
}
#endif
#endif
```

به چند چیز مهم درباره این فایل دقت نمایید. اول اینکه ساختار clssNativeDemoدو عضو را دربرمی گیرد j: وj. این ، متغیرهای نمونه تعریف شده توسط NativeDemo در فایل جاوا را معین می کند . دوم اینکه ، ماکرو HandleTo() ، نوع ساختار HNativeDemoرا ایجاد می کند ، که برای ارجاع به شیئی که test() را فراخوانی می کند مورد استفاده قرار می گیرد . به خط بعدی ، توجه ویژه ای مبذل دارید، که الگویی برای تابع test() که ایجاد کرده اید را تعریف می کند :

```
extern long NativeDemo_test(struct HNativeDemo *);
```

دقت نمایید که نام تابع ، NativeDemo-testاست . باید این را بعنوان نام تابع بومی که پیاده سازی می کنید ، بکار برید . یعنی که بجای ایجاد یک تابع C موسوم به test() ، یک تابع موسوم به NativeDemo-test() را ایجاد می کنید . پیشوند NativeDemo اضافه شده است چون مشخص می کند که روش test() بخشی از کلاسNativeDemoمی باشد . بیاد آورید ، کلاس دیگری ممکن است روش test() بومی خود را تعریف نماید که کاملا" با آنکه توسط NativeDemo اعلان شده ، متفاوت است . پیشوند گذاری نام روش بومی با نام کلاس فراهم کننده ، شیوه ای است برای تمایز بین روایتهای مختلف . بعنوان یک قانون عمومی ، توابع بومی ، یک نام کلاس که در آن اعلان شده اند را بعنوان یک پیشوند قبول می کنند . برگشت NativeDemo-test() از نوع long است ، اگرچه داخل برنامه جاوا که آن را فراخوانی می کند بعنوان یک int توصیف شده است . دلیل این امر بسیار ساده است . در جاوا ، اعداد صحیح مقادیر 32 بیتی هستند . در C ، یک عدد صحیح long لازم است حداقل 32 بیت برای یک نوع عدد صحیح تضمین نماید . یک چیز دیگر درباره الگوی NativeDemo-test() شایان توجه است . این الگو یک پارامتر از نوع HNativeDemo* تعریف می کند. کلیه روشهای بومی حداقل یک پارامتر دارند که اشاره گری است به شیئی که روش

بومی را فراخوانی نموده است . این پارامتر ضرورتا" شبیه This است . می توانید از این پارامتر استفاده کرده تا دسترسی به متغیرهای نمونه شیئی که روش را فراخوانی می کند ، داشته باشید

.برای تولید) NativeDemoC فایل (stub ، از این دستور استفاده کنیدjava-stubs NativeDemo :

فایل تولید شده توسط این دستور بصورت زیر می باشد :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include

/* Stubs for class NativeDemo */
/* SYMBOL :"NativeDemo/test ()I"/ Java_Native_test_stub */
__ declspec(dllexport )stack_item
*Java_NativeDemo_test_stub(stack_item *_P_/struct execenv *_ EE )_{
extern long NativeDemo_test(vide *);
_ P_[0].i = NativeDemo_test_(P_[0].p);
return_ P _+ 1;
}
```

شما این فایل را کامپایل نموده و با فایل پیاده سازی خود پیوند می دهید . پس از تولید فایلهای سرآیند و stub ضروری ، می توانید پیاده سازی خود از test() را بنویسید . از روایتی که در زیر نشان داده ایم ، استفاده نمایید :

```
/* This file contains the C version of the
test  ()method.
*/
#include
#include "NativeDemo.h"
#include
long NativeDemo_test(struct HNativeDemo *this)
{
printf("This is inside the native method.\n");
printf("this->i :%ld\n"/ unhand(this.-)i);
return( unhand(this-)>i);
}
```

دقت نمایید که این فایل دربرگیرنده stubpreamble.h است که شامل اطلاعات رابط سازی (interfacing) است . این فایل توسط کامپایلر جاوا برای شما فراهم می شود . فایل سرآیند NativeDemo قبلا" توسط java ایجاد شده بود . پس از ایجاد test.c باید آن را و NativeDemo.C را کامپایل نمایید . بعد این دو فایل را بعنوان یک) DLL کتابخانه پیوند پویا) با یکدیگر پیوند دهید . برای

انجام اینکار با کامپایلر میکروسافت C++/C ، از خط فرمان بعدی استفاده نماییدCL/LD NativeDemo.ctest.c : این فرمان یک فایل تحت عنوان NativeDemo.dll تولید می کند . هر بار که این کار انجام شود ، می توانید برنامه جاوا را اجرا کنید. انجام اینکار خروجی بعدی را تولید می کند :

```
This is inside the native method.
this->i :10
This is ob.j :20
```

داخل test.c به استفاده از unhand() توجه نمایید . این ماکرو توسط جاوا تعریف شده و یک اشاره گر را به نمونه ساختار class NativeDemoکه در NativeDemo.hتعریف شده و همراهی شیئی است که test() را فراخوانی می کند ، برمی گرداند . در کل هرگاه نیاز به دسترسی به اعضائ یک کلاس جاوا داشته باشید ، از unhand() استفاده می کنید .

یادآوری : توضیحاتی که استفاده از nativeرا محصور کرده اند، بستگی به پیاده سازی و محیط دارند . علاوه بر این ، رفتار مشخصی که در آن به کد جاوا رابط می سازید قابل تغییر است . باید مستندات موجود در سیستم توسعه جاوا خود را مطالعه نمایید تا جزئیات مربوط به روشهای بومی را درک کنید .

مشکلات مربوط به روشهای بومی

روشهای بومی بنظر می رسد تعهد بزرگی را پیشنهاد می کنند زیرا آنها به شما اجازه می دهند تا دسترسی به پایه موجود روالهای کتابخانه ای اتان را بدست آورده و امکان اجرای حین اجرای سریعتر را به شما عرضه می کنند . اما آنها همچنین دو مشکل اصلی را نشان می دهند: فرار امنیت بالقوه و کاهش قابلیت حمل . اجازه دهید باختصار این دو مورد را بررسی نماییم .

یک روش بومی ، یک ریسک امنیتی را مطرح می کند . از آنجاییکه یک روش بومی کد واقعی ماشین را اجرا می کند ، می تواند به هر بخش از رایانه میزبان دسترسی داشته باشد. یعنی که ، کد بومی منحصر به محیط اجرایی جاوا نیست . این کد امکان حمله ویروسی را هم می دهد. بهمین دلیل ریز برنامه ها نمی توانند از روشهای بومی استفاده نمایند. همچنین ، بارگذاری DLL ها می تواند محدود شود و بارگذاری آنها منوط به تصدیق مدیر امنیتی باشد .

دومین مشکل ، قابلیت حمل است . چون کد بومی داخل یک DLL گنجانده شده ، باید روی ماشینی که در حال اجرای برنامه جاوا است ، حاضر باشد. بعلاوه ، چون هر روش بومی بستگی به cpu و سیستم عامل دارد، هر DLL بطور وراثتی غیر قابل حمل میشود .بدین ترتیب ، یک برنامه جاوا که از روشهای بومی استفاده می کند، فقط قادر است وی یک ماشین که برای آن یک DLL سازگار نصب شده باشد ، اجرا شود.

منابع :

http://www.irandevelopers.com/
http://docs.sun.com

نویسنده :
محمد باقر معموری mamouri@ganjafzar.com
ویراستار و نویسنده قسمت های تکمیلی :
احسان شاه بختی zehs_sha@yahoo.com

کتاب :

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