More Classes

Overloading methods

- In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different
- When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading.
- Method overloading is one of the ways that Java supports polymorphism.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call
- Thus, overloaded methods must differ in the type and/or number of their parameters.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.
- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact.
- In some cases, Java's automatic type conversions can play a role in overload resolution. For example, consider the following program:

```java
// 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 and return type void 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) } }
```

Java

- As you can see, this version of OverloadDemo does not define test(int). Therefore, when test( ) is called with an integer argument inside Overload, no matching method is found. However, Java can automatically convert an integer into a double, and this conversion can be used to resolve the call.

- However, Java can automatically convert an integer into a double, and this conversion can be used to resolve the call. Therefore, after test(int) is not found, Java elevates i to double and then calls test(double). Of course, if test(int) had been defined, it would have been called instead. Java will employ its automatic type conversions only if no exact match is found.
- Method overloading supports polymorphism because it is one way that Java implements the “one interface, multiple methods” paradigm. To understand how, consider the following. In languages that do not support method overloading, each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data.
- For instance, in C, the function abs( ) returns the absolute value of an integer, labs( ) returns the absolute value of a long integer, and fabs( ) returns the absolute value of a floating-point value. Since C does not support overloading, each function has its own name, even though all three functions do essentially the same thing.

Overloading Constructors

- In addition to overloading normal methods, you can also overload constructor methods
- In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception.

```
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; } }
```

Java

- As you can see, the Box( ) constructor requires three parameters. This means that all declarations of Box objects must pass three arguments to the Box( ) constructor. For example, the following statement is currently invalid:

```java
Box ob = new Box();
```

- What if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions? Simply overload the box constructor

```java
/* 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 = depth = len; } // compute and return volume double
volume() { return width * height * depth; } } class OverloadCons {
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); } }
```

```
Volume of mybox1 is 3000.0 Volume of mybox2 is -1.0 Volume of mycube
is 343.0
```

Using Objects as Parameters

- So far, we have only been using simple types as parameters to methods.
- However, it is both correct and common to pass objects to methods. For example, consider the following short program:

```java
// Objects may be passed to methods. class Test { int a, b; Test(int i, int j) { a = i; b = j; } // return true if o is equal to the invoking object boolean equalTo(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.equalTo(ob2)); System.out.println("ob1 == ob3: " + ob1.equalTo(ob3)); } }
```

Java

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

Java

- As you can see, the equalTo( ) method inside Test compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed.
- Notice that the parameter o in equalTo( ) specifies Test as its type. Although Test is a class type created by the program, it is used in just the same way as Java's built-in types.
- One of the most common uses of object parameters involves constructors. Frequently, you will want to construct a new object so that it is initially the same as some existing object.
- To do this, you must define a constructor that takes an object of its class as a parameter. For example, the following version of Box allows one object to initialize another:

```java
// 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 = len; } //
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); } }
```

Java

A Closer Look at Argument Passing

- In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is call-by-value
- This approach copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.
- The second way an argument can be passed is call-by-reference. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter.
- Inside the subroutine, this reference is used to access the actual argument specified in the call.
- This means that changes made to the parameter will affect the argument used to call the subroutine.
- When you pass a primitive type to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method.

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

Java

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

Java

- When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference
- Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects act as if they are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument.

```java
// Objects are passed through their references. 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 PassObjRef { 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); } }
```

Java

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

Java

Returning Objects

- A method can return any type of data, including class types that you create. For example, in the following program, the incrByTen( ) method returns an object in which the value of a is ten greater than it is in the invoking object.

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

Java

- As you can see, each time incrByTen( ) is invoked, a new object is created, and a reference to it is returned to the calling routine.
- The preceding program makes another important point: Since all objects are dynamically allocated using new, you don't need to worry about an object going out-of- scope because the method in which it was created terminates. The object will continue to exist as long as there is a reference to it somewhere in your program
- When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

Introducing Access Control

- Allowing access to data only through a well- defined set of methods, you can prevent the misuse of that data
- Thus, when correctly implemented, a class creates a "black box" which may be used, but the inner workings of which are not open to tampering.
- How a member can be accessed is determined by the access modifier attached to its declaration. Java supplies a rich set of access modifiers.
- Some aspects of access control are related mostly to inheritance or packages. (A package is, essentially, a grouping of classes.) These parts of Java's access control mechanism will be discussed later
- Here, let's begin by examining access control as it applies to a single class. Once you understand the fundamentals of access control, the rest will be easy.

- Java's access modifiers are **public, private, and protected**. Java also defines a default access level. protected applies only when inheritance is involved. The other access modifiers are described next.

- When a member of a class is modified by public, then that member can be accessed by any other code
- When a member of a class is specified as private, then that member can only be accessed by other members of its class.
- Now you can understand why main( ) has always been preceded by the public modifier. It is called by code that is outside the program—that is, by the Java run-time system.
- When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. (Packages are discussed in the following chapter.)
- In the classes developed so far, all members of a class have used the default access mode. However, this is not what you will typically want to be the case. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class.

```java
/* 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()); } }
```

Java

Understanding static

- There will be times when you will want to define a class member that will be used independently of any object of that class.

- Normally, a class member must be accessed only
  in conjunction with an object of its class. However, it is possible to create a member that can
  be used by itself, without reference to a specific instance

- To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object
- You can declare both methods and variables to be static. The most common example of a static member is main( )
- main( ) is declared as static because it must be called before any objects exist.
- When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.
- Methods declared as static have several restrictions:
  - They can only directly call other static methods.
  - They can only directly access static data.
  - They cannot refer to this or super in any way. (The keyword super relates to inheritance and is described in the next chapter.)
- If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a static method, some static variables, and a static initialization block

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

Java

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

Java

- As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes, which prints a message and then initializes b to a*4 or 12. Then main( ) is called, which calls meth( ), passing 42 to x. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

- Outside of the class in which they are defined, static methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a static method from outside its class, you can do so using the following general form:

```java
classname.method( )
```

```java
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); } }
```

```
a = 42 b = 99
```

Introducing final

- A field can be declared as final. Doing so prevents its contents from being modified, making it, essentially, a constant.
- This means that you must initialize a **final** field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is the most common. Here is an example:

```java
final int FILE_NEW = 1; final int FILE_OPEN = 2; final int FILE_SAVE
= 3; final int FILE_SAVEAS = 4; final int FILE_QUIT = 5;
```

Java

- In addition to fields, both method parameters and local variables can be declared final. Declaring a parameter final prevents it from being changed within the method. Declaring a local variable final prevents it from being assigned a value more than once.
- The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This additional usage of final is described in the next chapter, when inheritance is described.

Introducing Nested and Inner Classes

- It is possible to define a class within another class; such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class.
- Thus, if class B is defined within class A, then B does not exist independently of A.
- A nested class has access to the members, including private members, of the class in which it is nested
- However, the enclosing class does not have access to the members of the nested class.
- A nested class that is declared directly within its enclosing class scope is a member of its enclosing class
- There are two types of nested classes: static and non-static.
- A static nested class is one that has the static modifier applied but are rarely used
- The most important type of nested class is the inner class. An inner class is a non-static nested class.
- It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do

```java
// Demonstrate an inner class. class Outer { int outer_x = 100; void test() { Inner inner = new Inner(); inner.display(); } // this is an innner class class Inner { void display() { System.out.println("display: outer_x = " + outer_x); } } } class InnerClassDemo { public static void main(String args[]) { Outer outer = new Outer(); outer.test(); } }
```

Java

```
display: outer_x = 100
```

Java

- In the program, an inner class named Inner is defined within the scope of class Outer. Therefore, any code in class Inner can directly access the variable outer_x.