

Inheritance

- Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items.
- This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a superclass.
- The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a superclass. It inherits all of the members defined by the superclass and adds its own, unique elements.

Inheritance Basics

- To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword. Consider the following example.

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

Java ▾

```
Output : 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
```

Java ▾

- You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass.
- You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

Member Access and Inheritance

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private.
- For example, consider the following simple class hierarchy:

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

Java ▾

- This program will not compile because the use of j inside the sum() method of B causes an access violation. Since j is declared as private, it is only accessible by other members of its own class. Subclasses have no access to i.
- A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

A More Practical Example

- Let's look at a more practical example that will help illustrate the power of inheritance. Here, the final version of the Box class developed in the preceding chapter will be extended to include a fourth component called weight. Thus, the new class will contain a box's width, height, depth, and weight.

```

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; } } // 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); } }

```

Java ▾

```

Output : Volume of mybox1 is 3000.0 Weight of mybox1 is 34.3 Volume
of mybox2 is 24.0 Weight of mybox2 is 0.076

```

Java ▾

- BoxWeight inherits all of the characteristics of Box and adds to them the weight component. It is not necessary for BoxWeight to re-create all of the features found in Box. It can simply extend Box to meet its own purposes.
- A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification.

A Superclass Variable Can Reference a Subclass Object

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. You will find this aspect of inheritance quite useful in a variety of situations.

```
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 does not define a weight
member. */ // System.out.println("Weight of plainbox is " +
plainbox.weight); } }
```

Java ▼

- Here, weightbox is a reference to BoxWeight objects, and plainbox is a reference to Box objects. Since BoxWeight is a subclass of Box, it is permissible to assign plainbox a reference to the weightbox object.
- It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.

Using super

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.
- **super** has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass.

```
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; } }
```

Java ▼

- Here, BoxWeight() calls super() with the arguments w, h, and d. This causes the Box constructor to be called, which initializes width, height, and depth using these values.

- BoxWeight no longer initializes these values itself. It only needs to initialize the value unique to it:

```
// 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 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; } } 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.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);
System.out.println(); vol = mybox3.volume();
System.out.println("Volume of mybox3 is " + vol);
System.out.println("Weight of mybox3 is " + mybox3.weight);
System.out.println(); vol = myclone.volume();
System.out.println("Volume of myclone is " + vol);
System.out.println("Weight of myclone is " + myclone.weight);
System.out.println(); vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
System.out.println("Weight of mycube is " + mycube.weight);
System.out.println(); } }
```

```
Output : Volume of mybox1 is 3000.0 Weight of mybox1 is 34.3 Volume
of mybox2 is 24.0 Weight of mybox2 is 0.076 Volume of mybox3 is -1.0
Weight of mybox3 is -1.0 Volume of myclone is 3000.0 Weight of
myclone is 34.3 Volume of mycube is 27.0 Weight of mycube is 2.0
```

Java ▾

Notice that `super()` is passed an object of type `BoxWeight`—not of type `Box`. This still

invokes the constructor `Box(Box ob)`. As mentioned earlier, a superclass variable can be

used to reference any object derived from that class. Thus, we are able to pass a `BoxWeight`

object to the `Box` constructor.

A Second Use for `super`

- The second form of `super` acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:
super.member
- Here, `member` can be either a method or an instance variable. This second form of `super` is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
class A { int i; } // Create a subclass by extending class A. class B
extends A { int i; // this i hides the i 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(); } }
```

Java ▾

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

Java ▾

- Although the instance variable `i` in `B` hides the `i` in `A`, `super` allows access to the `i` defined in the superclass.

Creating a Multilevel Hierarchy

- You can build hierarchies that contain as many layers of inheritance as you like. It is perfectly acceptable to use a subclass as a superclass of another.
- To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass BoxWeight is used as a superclass to create the subclass called Shipment.
- Shipment inherits all of the traits of BoxWeight and Box, and adds a field called cost, which holds the cost of shipping such a parcel.

```
// 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 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 when all parameters are specified
Shipment(double w, double h, double d, double m, double c) { super(w,
h, d, m); // call superclass constructor cost = c; } // default
constructor Shipment() { super(); cost = -1; } // constructor used
when cube is created Shipment(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); } }
```

Java ▾

```
Output : Volume of shipment1 is 3000.0 Weight of shipment1 is 10.0
Shipping cost: $3.41 Volume of shipment2 is 24.0 Weight of shipment2
is 0.76 Shipping cost: $1.28
```

Java ▾

- Because of inheritance, Shipment can make use of the previously defined classes of Box and BoxWeight, adding only the extra information it needs for its own, specific application.
This is part of the value of inheritance; it allows the reuse of code.

When Constructors Are Executed

- In a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass. Further, since `super()` must be the first statement executed in a subclass' constructor, this order is the same whether or not `super()` is used.
- If `super()` is not used, then the default or parameterless constructor of each superclass will be executed.

```
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(); } }
```

Java ▾

Output : Inside A's constructor Inside B's constructor Inside C's
constructor

Java ▾

- If you think about it, it makes sense that constructors complete their execution in order of derivation.
- Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its execution first.

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.

- When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

```
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 } }
```

Java ▾

Output : k: 3

Java ▾

- When show() is invoked on an object of type B, the version of show() defined within B is used. That is, the version of show() inside B overrides the version declared in A.
- If you wish to access the superclass version of an overridden method, you can do so by using super.
- For example, in this version of B, the superclass version of show() is invoked within the subclass' version. This allows all instance variables to be displayed.

```
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); } }
```

Java ▾

- Here, super.show() calls the superclass version of show().
- Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.
- For example, consider this modified version of the preceding example:

```
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 } }
```

Java ▾

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

Java ▾

- The version of show() in B takes a string parameter. This makes its type signature different from the one in A, which takes no parameters.
- Therefore, no overriding (or name hiding) takes place. Instead, the version of show() in B simply overloads the version of show() in A.

Dynamic Method Dispatch

- Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
- Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

```
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 } }
```

Java ▾

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

Java ▾

- This program creates one superclass called A and two subclasses of it, called B and C. Subclasses B and C override callme() declared in A. Inside the main() method, objects of type A, B, and C are declared.
- Also, a reference of type A, called r, is declared. The program then in turn assigns a reference to each type of object to r and uses that reference to invoke callme().
- As the output shows, the version of callme() executed is determined by the type of object being referred to at the time of the call.

Why Overridden Methods?

- As stated earlier, overridden methods allow Java to support run-time polymorphism.

- Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.
- By combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.
- Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness.
- The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Applying Method Overriding

- The following program creates a superclass called Figure that stores the dimensions of a two-dimensional object.
- It also defines a method called `area()` that computes the area of an object. The program derives two subclasses from Figure. The first is Rectangle and the second is Triangle.
- Each of these subclasses overrides `area()` so that it returns the area of a rectangle and a triangle, respectively.

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

Java ▾

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

Java ▾

- Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects.

Using Abstract Classes

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.

- To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator.
- Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared abstract itself.

```
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 implementation of
callme."); } } class AbstractDemo { public static void main(String
args[]) { B b = new B(); b.callme(); b.callmetoo(); } }
```

Java ▾

- Notice that no objects of class A are declared in the program. As mentioned, it is not possible to instantiate an abstract class.
- One other point: class A implements a concrete method called callmetoo(). This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

Using final with Inheritance

- The keyword final has three uses. First, it can be used to create the equivalent of a named constant.
- The other two uses of final apply to inheritance.

Using final to Prevent Overriding

- While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring.
- To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden.

```
class A { final void meth() { System.out.println("This is a final method."); } } class B extends A { void meth() { // ERROR! Can't override. System.out.println("Illegal!"); } }
```

Java ▾

- Because meth() is declared as final, it cannot be overridden in B. If you attempt to do so, a compile-time error will result.
- Methods declared as final can sometimes provide a performance enhancement: The compiler is free to inline calls to them because it “knows” they will not be overridden by a subclass.
- When a small final method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call.
- Inlining is an option only with final methods. Normally, Java resolves calls to methods dynamically, at run time.
- This is called late binding. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

Using final to Prevent Inheritance

- Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final.
- Declaring a class as final implicitly declares all of its methods as final, too. As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

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

Java ▾

- As the comments imply, it is illegal for B to inherit A since A is declared as final.

The Object Class

- There is one special class, Object, defined by Java. All other classes are subclasses of Object.
That is, Object is a superclass of all other classes.
- This means that a reference variable of type Object can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type Object can also refer to any array.
- Object defines the following methods, which means that they are available in every object.

| Method | Purpose |
|--|---|
| Object clone() | Creates a new object that is the same as the object being cloned. |
| boolean equals(Object <i>object</i>) | Determines whether one object is equal to another. |
| void finalize() | Called before an unused object is recycled. |
| Class<?> getClass() | Obtains the class of an object at run time. |
| int hashCode() | Returns the hash code associated with the invoking object. |
| void notify() | Resumes execution of a thread waiting on the invoking object. |
| void notifyAll() | Resumes execution of all threads waiting on the invoking object. |
| String toString() | Returns a string that describes the object. |
| void wait() void wait(long <i>milliseconds</i>) void wait(long <i>milliseconds</i> , int <i>nanoseconds</i>) | Waits on another thread of execution. |

- The methods getClass(), notify(), notifyAll(), and wait() are declared as final. You may override the others.
- The equals() method compares two objects. It returns true if the objects are equal, and false otherwise. The precise definition of equality can vary, depending on the type of objects being compared.
- The toString() method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using println().
- Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.

