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.

To inherit a class, you simply incorporate the definition of one class into another by using the **extends keyword.**

```
// Create a superclass.
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + " " + j);
}
}

// Create a subclass by extending class A.
class B extends A {
int k;
void showk() {
System.out.println("k: " + k);
}
void sum() {
System.out.println("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();
}
}
```

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

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.**

```
// 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);
```

```
// This program uses inheritance to extend Box.
class Box {
double width;
double height;
double depth;
// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}

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

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

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

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
```

```
// Here, Box is extended to include color.
class ColorBox extends Box {
int color; // color of box
ColorBox(double w, double h, double d, int c) {
width = w;
height = h;
depth = d;
color = c;
}
}
```

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.

```
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);
}
}
```

Using super

In the preceding examples, classes derived from Box were not implemented as efficiently or as robustly as they could have been. For example, the constructor for BoxWeight explicitly initializes the width, height, and depth fields of Box. Not only does this **duplicate code found in its superclass**, which is inefficient, but it implies that a subclass must be granted access to these members.

However, there will be times when you will want to **create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private**).

In this case, there would be no way for a subclass to directly access or initialize these variables on its own.

Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem.

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.

Using super to Call Superclass Constructors

super(*arg-list);*

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

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

```
// 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
}
```

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

This program generates the following 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
```

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.*

```
// Using super to overcome name hiding.
class A {
int i;
}
// Create a subclass by extending class A.
class B extends A {
int i; // this i hides the 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();
}
}
```

Creating a Multilevel Hierarchy

```
// 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
```

```
// 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
```

```
// 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
```

```
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);
}
}
```

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

When Constructors Are Executed

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy executed? For example, given a subclass called B and a superclass called A, is A's constructor executed before B's, or vice versa? The answer is that 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. The following program illustrates when constructors are executed:

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

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

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

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

Output :

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

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.**

```
// 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);
```

```
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

output:

```
k: 3
```

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);
}
}
```

If you substitute this version of A into the previous program, you will see the following

output:
i and j: 1 2
k: 3

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:

```
// 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
}
}
```

The output produced by this program is shown here:

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

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.

Let's begin by restating an important principle**: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time.**

Here is how. When an overridden method is called through a superclass reference, 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**.

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

```
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

output :

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

Why Overridden Methods?

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.

Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own.

This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, 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 objectoriented 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

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

output:

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

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.

This is the case with the class Figure used in the preceding example. The definition of area( ) is simply a placeholder. It will not compute and display the area of any type of object.

As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways.
One way, as shown in the previous example, is to simply have it **report a warning message**. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate.
You may have methods that must be **overridden by the subclass in order for the subclass to have any meaning.** Consider the class Triangle. It has no meaning if area( ) is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. **Java's solution to this problem is the *abstract method*.**

To declare an abstract method, use this general form:

abstract *type name(parameter-list);*

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

abstract void callme();
// concrete methods are still allowed in abstract classes

void callmetoo() {
System.out.println("This is a concrete method.");
}
}

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

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

```
// Using abstract methods and classes.
abstract class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
// area is now an abstract method
abstract double area();
}
class Rectangle extends Figure {
Rectangle(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 AbstractAreas {
public static void main(String args[]) {
// Figure f = new Figure(10, 10); // illegal now
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref; // this is OK, no object is created
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
}
}
```

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!");
}
}
```

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.

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
//...
}
```

Local Variable Type Inference and Inheritance

two important aspects of variables.

First, all variables in Java must be declared prior to their use.

Second, a variable can be initialized with a value when it is declared.

when a variable is initialized, the type of the initializer must be the same as (or convertible to) the declared type of the variable. Thus, in principle, it would not be necessary to specify an explicit type for an initialized variable because it could be inferred by the type of its initializer. Of course, in the past, such inference was not supported, and all variables required an explicitly declared type, whether they were initialized or not. Today, that situation has changed.

Beginning with JDK 10, it is now possible to let the compiler infer the type of a local variable based on the type of its initializer, thus avoiding the need to explicitly specify the type.

Local variable type inference offers a number of advantages. For example, it can streamline code by eliminating the need to redundantly specify a variable's type when it can be inferred from its initializer. It can simplify declarations in cases in which the type name is quite lengthy, such as can be the case with some class names. It can also be helpful when a type is difficult to discern or cannot be denoted.

Furthermore, local variable type inference has become a common part of the contemporary programming environment.

Its inclusion in Java helps keep Java up-to-date with evolving trends in language design. To support local variable type inference, the context-sensitive identifier **var** was added to Java as a *reserved type name.*
To use local variable type inference, the variable must be declared with var as the type name and it must include an initializer.

For example, in the past you would declare a local double variable called avg that is initialized with the value 10.0, as shown here:

double avg = 10.0;
Using type inference, this declaration can now also be written like this:
var avg = 10.0; avg will be of type double.
In the first case, its type is explicitly specified.
In the second, its type is inferred as double because the initializer 10.0 is of type double.

As mentioned, var was added as a context-sensitive identifier. When it is used as the type name in the context of a local variable declaration, it tells the compiler to use type inference to determine the type of the variable being declared based on the type of the initializer. Thus, in a local variable declaration, var is a placeholder for the actual, inferred type. However, when used in most other places, var is simply a user-defined identifier with no special meaning.

For example, the following declaration is still valid:

```
int  var = 1; // In this case, var is simply a user-defined identifier.
```

In this case, the type is explicitly specified as int and var is the name of the variable being declared. Even though it is a context-sensitive identifier, there are a few places in which the use of var is illegal.

```
// A simple demonstration of local variable type inference.
class VarDemo {
  public static void main(String args[]) {

    // Use type inference to determine the type of the
    // variable named avg. In this case, double is inferred.
    var avg = 10.0;
    System.out.println("Value of avg: " + avg);

    // In the following context, var is not a predefined identifier.
    // It is simply a user-defined variable name.
    int var = 1;
    System.out.println("Value of var: " + var);

    // Interestingly, in the following sequence, var is used
    // as both the type of the declaration and as a variable name
    // in the initializer.
    var k = -var;
    System.out.println("Value of k: " + k);
  }
}
```

output:
```
Value of avg: 10.0
Value of var: 1
Value of k: -1
```

It is important to have a clear understanding of how type inference works within an inheritance hierarchy.
Recall that a superclass reference can refer to a derived class object, and this feature is part of Java's support for polymorphism.
However, it is critical to remember that, when using local variable type inference, the inferred type of a variable is based on the declared type of its initializer. Therefore, if the initializer is of the superclass type, that will be the inferred type of the variable. It does not matter if the actual object being referred to by the initializer is an instance of a derived class. For example, consider this program:

```
// When working with inheritance, the inferred type is the declared
// type of the initializer, which may not be the most derived type of
// the object being referred to by the initializer.

class MyClass {
  // ...
}

class FirstDerivedClass extends MyClass {
  int x;
  // ...
}

class SecondDerivedClass extends FirstDerivedClass {
  int y;
  // ...
}
```

```
class TypeInferenceAndInheritance {
  // Return some type of MyClass object.
  static MyClass getObj(int which) {
    switch(which) {
      case 0: return new MyClass();
      case 1: return new FirstDerivedClass();
      default: return new SecondDerivedClass();
    }
  }
```

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

    // Even though getObj() returns different types of
    // objects within the MyClass inheritance hierarchy,
    // its declared return type is MyClass. As a result,
    // in all three cases shown here, the type of the
    // variables is inferred to be MyClass, even though
    // different derived types of objects are obtained.

    // Here, getObj() returns a MyClass object.
    var mc = getObj(0);

    // In this case, a FirstDerivedClass object is returned.
    var mc2 = getObj(1);

    // Here, a SecondDerivedClass object is returned.
    var mc3 = getObj(2);

    // Because the types of both mc2 and mc3 are inferred
    // as MyClass (because the return type of getObj() is
    // MyClass), neither mc2 nor mc3 can access the fields
    // declared by FirstDerivedClass or SecondDerivedClass.
//    mc2.x = 10; // Wrong! MyClass does not have an x field.
//    mc3.y = 10; // Wrong! MyClass does not have a y field.
  }
}
```

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 *object*)	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 *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

The methods getClass( ), notify( ), notifyAll( ), and wait( ) are declared as final.

You may override the others. However, notice two methods now: equals( ) and toString( ).

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.