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, 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 instance variables and methods defined by the superclass and add its own, unique elements.

The general form of a **class** declaration that inherits a superclass is shown here:

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

Example:

```
public class Base{
        int i=10;
        public void show (){
                System.out.println (" In the show() ");
                   }
        }
public class Derived extends Base{
                   int j=34;
                public void display(){
        System.out.println (" In the display() ");

        public static void main(String args[ ]){
                Derived d1=new Derived();
                d1.show();
                d1.display();
        }
        }
```

Note:

1. Whenever we create an object of the sub class, with the help of constructor, internally java will call the constructor of the super class & holds the reference object of the super class in "**super**" keyword.
2. Whenever we call a method or access a variable using of subclass, Java will check whether this method or variable is defined in the sub class or not. In case, if it is not available, Java will check for them in the super class. If the super class is also not having them, it will try to find them in it's super class & so on.

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.

The key benefit of overriding is the ability to define behavior that's specific to a particular subclass type. When an overridden method is called from within a 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. Consider the following:

```
// 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
}}
```

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the versio n of **show( )** inside **B** overrides the version declared in **A**. If you wish to access the superclass version of an overridden function, you can do so by using **super**.

The rules for overriding a method are as follows:

1. The argument list must exactly match that of the overridden method. If they don't match, you can end up with an overloaded method you didn't intend.
2. The return type must be the same as, or a subtype of, the return type declared in the original overridden method in the superclass. (More on this in a few pages when we discuss covariant returns.).
3. The access level can't be more restrictive than the overridden method's. The access level CAN be less restrictive than that of the overridden method.
4. Instance methods can be overridden only if they are inherited by the subclass.
5. A subclass within the same package as the instance's superclass can override any superclass method that is not marked `private` or `final`. A subclass in a different package can override only those non-`final` methods marked `public` or `protected`.
6. Method overriding occurs *only* when the names and the type signatures of the two methods are identical.
7. The overriding method CAN throw any unchecked (runtime) exception, regardless of whether the overridden method declares the exception.
8. The overriding method must NOT throw checked exceptions that are new or broader than those declared by the overridden method. For example, a method that declares a FileNotFoundException cannot be overridden by a method that declares a SQLException, Exception, or any other non-runtime exception unless it's a subclass of FileNotFoundException.

super Keyword

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.

1. The first calls the superclass' constructor.
2. The second is used to access a member of the superclass that has been hidden by a member of a subclass. Each use is examined here.

1. Using super to Call Superclass Constructors

A subclass can call a constructor method defined by its superclass by use of the following form of **super**:

super(*parameter-list*);

Here, *parameter-list* specifies any parameters needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass' constructor. To see how **super( )** is used, consider this improved version of the **BoxWeight( )** class:

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
        double weight;
        BoxWeight(double w, double h, double d, double m) {
                super(w, h, d); // call superclass constructor
                weight = m;
                }
        }
```

Here, **BoxWeight( )** calls **super( )** with the parameters **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: **weight**. This leaves **Box** free to make these values **private** if desired.

In the preceding example, **super( )** was called with three arguments. Since constructors can be overloaded, **super( )** can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments.

2. 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.

// 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();
}
}
```

This program displays the following:

i in superclass: 1

i in subclass: 2

Note: The overriding method cannot have a more restrictive access modifier than the method being overridden (for example, you can't override a method marked public and make it protected).

Dynamic Method Dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

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

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. Here is an example that illustrates dynamic method dispatch:

// 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
}}
```

The output from the program is shown here:

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

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 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. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**'s **callme( )** method.

Abstract class

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.

You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

abstract ***type name(parameter-list)*****;**

1. Where, no method body is present.
2. Any class that contains one or more abstract methods must also be declared abstract.
3. To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration.
4. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator.
5. Abstract class can have constructor, but it can called only through the non-abstract child class instace.

Example:

```
abstract  class  AbsDemo{
abstract  int square(int);
}
Class Abs  extends AbsDemo{
Int square(int i)
{return i*I;}
Public static void main(String args[])
{
Abs ob=new Abs();
System.out.println("Square of 5="+square(5));
}
}
```

Using final Keyword

The keyword **final** has three uses.

1. It can be used to create the equivalent of a named constant.
2. To **Prevent Overriding** i.e.disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden.

The following fragment illustrates **final**:

```
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!");
        }}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

3. While 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.
Here is an example of a **final** class:

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

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

Packages

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage.

Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package.

Defining a Package:

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name.
This is the general form of the **package** statement:

package *pkg*;

Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**.

package MyPackage;

NOTE:

- Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly.
- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:
  package *pkg1*[.*pkg2*[.*pkg3*]];
- A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

package java.awt.image;

- Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Understanding CLASSPATH

- While packages solve many problems from an access control and name-space-collision perspective, they cause some curious difficulties when you compile and run programs. This is because the specific location that the Java compiler will consider as the root of any package hierarchy is controlled by **CLASSPATH**.
- Until now, you have been storing all of your classes in the same, unnamed default package. Doing so allowed you to simply compile the source code and run the Java interpreter on the result by naming the class on the command line. This worked because the default current working directory (**.**) is usually in the **CLASSPATH** environmental variable defined for the Java run-time system, by default.
- Assume that you create a class called **PackTest** in a package called **test**. We create a directory called **test** and put **PackTest.java** inside that directory. You then make **test** the current directory and compile **PackTest.java**. This results in **PackTest.class** being stored in the **test** directory, as it should be.
- When you try to run **PackTest**, though, the Java interpreter reports an error message similar to "can't find class PackTest." This is because the class is now stored in a package called **test**. You can no longer refer to it simply as **PackTest**. You must refer to the class by enumerating its package hierarchy, separating the packages with dots. This class must now be called **test.PackTest**.
- However, if you try to use **test.PackTest**, you will still receive an error message similar to "can't find class test/PackTest.".
- The reason you still receive an error message is hidden in your **CLASSPATH** variable. Remember, **CLASSPATH** sets the top of the class hierarchy. The problem is that there's no **test** directory in the current working directory, because *you are in* the **test** directory, itself.
- You have two choices at this point: change directories up one level and try **java test.PackTest**, or add the top of your development class hierarchy to the **CLASSPATH** environmental variable. Then you will be able to use **java test.PackTest** from any directory, and Java will find the right **.class** file. For example, if you are working on your source code in a directory called **C:\\myjava**, then set your CLASSPATH to .;C:\\myjava;C:\\java\\classes

- Call this file **AccountBalance.java**, and put it in a directory called **MyPack**. Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then try executing the **AccountBalance** class, using the following command line:
  java MyPack.AccountBalance

- Remember, you will need to be in the directory above **MyPack** when you execute this command, or to have your **CLASSPATH** environmental variable set appropriately.
- As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:
  java AccountBalance

- **AccountBalance** must be qualified with its package name.

Access Protection:

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages,

Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories.

Table 9-1. Class Member Access

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package Subclass	No	Yes	Yes	Yes
Same package Non subclass	No	Yes	Yes	Yes
Different package Sub class	No	No	Yes	Yes
Different package non- subclass	No	No	No	Yes

Importing Packages

Java includes the **import** statement to bring certain classes, or entire packages, into visibility. In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

import *pkg1*[*.pkg2*].(*classname*|*);

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (**.**). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system.

you specify either an explicit *classname* or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

Caution: The star form may increase compilation time—especially if you import several large packages.

Example:

```
package MyPack;
public class Balance {
String name;
double bal;

public Balance(String n, double b) {
name = n;
bal = b;
}
```

```
public void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
import MyPack.*;
class TestBalance {
public static void main(String args[]) {
/* Because Balance is public, you may use
Balance
class and call its constructor. */
Balance test = new Balance("J. J. Jaspers", 99.88);
test.show(); // you may also call show()
}
}
```

Interfaces

- ✓ Using the keyword **interface**, you can fully abstract a class' interface from its implementation.
- ✓ Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- ✓ Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces.
- ✓ To implement an interface, a class must create the complete set of methods defined by the interface.
- ✓ By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.
- ✓ Interfaces are designed to support dynamic method resolution at run time.

Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

```
access interface_name {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
        type final-varname1 = value;
        type final-varname2 = value;
        // ...
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
        }
```

- ✓ *access* is either **public** or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared.
- ✓ When it is declared as **public**, the interface can be used by any other code.
- ✓ *name* is the name of the interface, and can be any valid identifier.
- ✓ Notice that the methods which are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface.
- ✓ Each class that includes an interface must implement all of the methods.

Example:
```
interface Callback {
void callback(int param);
}
```

Implementing Interfaces:

To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the **implements** clause looks like this:

***access* class *classname* [extends *superclass*] [implements *interface* [,*interface...*]]**
{

// **class-body**
}

- *access* is either **public** or not used.
- If a class implements more than one interface, the interfaces are separated with a omma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared **public**. .

Here is a small example class that implements the **Callback** interface shown earlier.

```
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}}
```

Notice that **callback ( )** is declared using the **public** access specifier.

Variables in Interfaces:

- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values.
- When you include that interface in a class (that is, when you "implement" the interface), all of those variable names will be in scope as constants. This is similar tousing a header file in C/C++ to create a large number of **#defined** constants or **const** declarations.
- If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything.
- It is as if that class were importing the constant variables into the class name space as **final** variables.

Interfaces Can Be Extended

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
void meth1();
void meth2();
}
// B now includes meth1() and meth2() -- it adds
meth3().
interface B extends A {
void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
}}
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}}
```

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 is known to A, but not outside 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.

There are two types of nested classes: *static* and *non-static*.

1. **Static:** A static nested class is one which has the **static** modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.
2. **Non-static:** it is categorized in three types
    1. **Inner class**
    2. **Local Inner class**
    3. **Anonymous**
3. 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. Thus, an inner class is fully within the scope of its enclosing class.

The following program illustrates how to define and use an **inner class**. The class named **Outer** has one instance variable named **outer_x**, one instance method named **test( )**, and defines one inner class called **Inner**.

```
// Demonstrate an inner class.
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner 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();
}}
```

Output from this application is shown here:

```
display: outer_x = 100
```

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**.
- ✓ An instance method named **display( )** is defined inside **Inner**. This method displays **outer_x** on the standard output stream.
- ✓ The **main( )** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test( )** method. That method creates an instance of class **Inner** and the **display( )** method is called.

It is important to realize that class **Inner** is known only within the scope of class **Outer**. The Java compiler generates an error message if any code outside of class **Outer** attempts to instantiate class **Inner**.

Generalizing, a nested class is no different than any other program element: it is known only within its enclosing scope.

As explained, an inner class has access to all of the members of its enclosing class, but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class. For example,

```
// This program will not compile.
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
int y = 10; // y is local to Inner
void display() {
System.out.println("display: outer_x = " +
outer_x);
}}
void showy() {
System.out.println(y); // error, y not known here!
}}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}}
```

Here, **y** is declared as an instance variable of **Inner**. Thus it is not known outside of that class and it cannot be used by **showy( )**. Although we have been focusing on nested classes declared within an outer class scope, it is possible to define inner classes within any block scope.
For example, you can define a nested class within the block defined by a method or even within the body of a **for** loop, as this next program shows.

```
// Define an inner class within a for loop.
class Outer {
int outer_x = 100;
void test() {
for(int i=0; i<10; i++) {
class Inner {
void display() {
System.out.println("display: outer_x = " +
outer_x);
}
}
Inner inner = new Inner();
inner.display();
}}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}
}
```

anonymous inner classes, which are inner classes that don't have a name.
One final point: Nested classes were not allowed by the original 1.0 specification for Java. They were added by Java 1.1.

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. 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.
- Instance variables declared as **static** are, essentially, global variables.
- 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:

1. They can only call other **static** methods.
2. They must only access **static** data.
3. They cannot refer to **this** or **super** in any way.

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block which 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:

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

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 (printing a message), and finally, **b** is initialized 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**.

Note It is illegal to refer to any instance variables inside of a **static** method.
Here is the output of the program:

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

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:

classname.method( );

Here,

1. *classname* is the name of the class in which the **static** method is declared.
2. This format is similar to that used to call non-**static** methods through object- reference variables.
3. A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class.
4. This is how Java implements a controlled version of global functions and global variables.