
UNIT 3 PACKAGES AND INTERFACES

Structure	Page Nos.
3.0 Introduction	45
3.1 Objectives	45
3.2 Package	46
3.2.1 Defining Package	
3.2.2 CLASSPATH	
3.2.3 Package naming	
3.3 Accessibility of Packages	49
3.4 Using Package Members	49
3.5 Interfaces	51
3.6 Implementing Interfaces	53
3.7 Interface and Abstract Classes	56
3.8 Extends and Implements Together	56
3.9 Summary	57
3.10 Solutions/Answers	57

3.0 INTRODUCTION

Till now you have learned how to create classes and use objects in problem solving. If you have several classes to solve a problem, you have to think about a container where you can keep these classes and use them. In Java you can do this by defining packages. Also, Java provides a very rich set of package covering a variety of areas of applications including basic I/O, Networking, and Databases. You will use some of these packages in this course. In this unit you will learn to create your own packages, and use your own created package in programming.

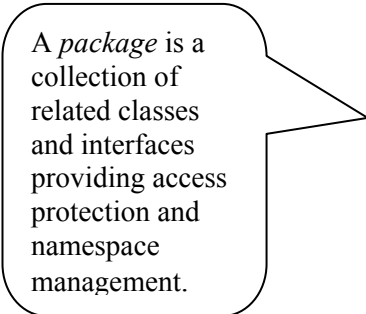
In this unit we will also discuss *interface*, one of Java's useful features. If we take Object Orientation into consideration an *interface in Java* is a subset of the public methods of a class. The *implementation* of a class is the code that makes up those methods. In Java the meaning of interface is different from the meaning in Object Orientation. An *interface* is just a specification of a set of *abstract methods*. If a class implements the interface, then it is essential for the class to provide an implementation for all of the abstract methods in the interface. As we have discussed in Unit 2 of this block that Java does not provide multiple inheritance. You can use interfaces to overcome the limitation of non-availability of multiple inheritance, because in Java a class can implement many interfaces. In this unit you will learn to create interfaces and implement them in classes.

3.1 OBJECTIVES

After going through this unit you will be able to:

- explain what is a package in Java;
- set CLASSPATH variable;
- using user created packages in programs;
- define Interface;
- explain the need of Java interfaces in programming;
- implement interfaces in classes, and
- use interfaces to store constant variables of programs.

3.2 PACKAGE



A *package* is a collection of related classes and interfaces providing access protection and namespace management.

Java programmer creates packages to partition classes. Partitioning of classes helps in managing the program. The package statement is used to define space to store classes. In Java you can write your own package. Writing packages is just like you write any other Java program. You just have to take care of some points during writing packages, which are given below.

There must be not more than one public class per file.

All files in the package must be named `name_of_class.java` where `name_of_class` is the name of the single public class in the file.

The very first statement in each file in the package, before any import statements or anything put the statement `package myPackageName;`

Now let us see how to define a package.

3.2.1 Defining Package

You can define package for your own program
`package PackageName;`

This statement will create a package of the name **PackageName**. You have always to do one thing that is **.class** files created for the classes in package—let us say in package **PackageName**— must be stored in a directory named **PackageName**. In other words you can say that the directory name must exactly match the package name.

You can also create a hierarchy of packages. To do this you have to create packages by separating them using period(.). Generally a multilevel package statement look likes:

```
package MyPak1[.MyPak2[.MyPak3]];
```

For example, if a package is declared as:
`package Java.applet;`

Then it will be stored in directory `Java\applet` in Windows. If the same has to be stored in Unix then it will be stored in directory `Java/applet`.

There is one system environment variable named as CLASS PATH. This variable must be set before any package component takes part in programs. Now we will discuss CLASSPATH.

3.2.2 CLASSPATH

CLASSPATH is an environment variable of system. The setting of this variable is used to provide the root of any package hierarchy to Java compiler.

Suppose you create a package named **MyPak**. It will be stored in **MyPak** directory. Now let us say class named **MyClass** is in **MyPak**. You will store **MyClass.java** in **MyPak** directory. To compile **MyClass.java** you have to make **MyPak** as current directory and **MyClass.class** will be stored in **MyPak**.

Can you run **MyClass.class** file using Java interpreter from any directory? No it is not so, because **MyClass.class** is in package **MyPak**, so during execution you will refer to package hierarchy. For this purpose you have to set CLASSPATH variable for setting the top of the hierarchy. Once you set CLASSPATH for **MyPak**, it can be used from any directory.

For example, if /home/MyDir/Classes is in your CLASSPATH and your package is called **MyPak1**, then you would make a directory called **MyPak** in /home/MyDir/Classes and then put all the **.class** files in the package in /home/MyDir/Classes/MyPak1.

Now let us see an example program to create a package.

```
//program
package MyPack;
class Student
{
    String Name;
    int Age;
    String Course;
    Student( String n, int a, String c)
    {
        Name = n;
        Age = a;
        Course = c;
    }
    void Student_Information()
    {
        System.out.println("Name of the Student :"+ Name);
        System.out.println("Age of the Student :"+Age);
        System.out.println("Enrolled in Course :"+Course);
    }
}
class PackTest
{
    public static void main( String args[])
    {
        Student Std1 = new Student("Rajeev",19, "MCA");
        Std1.Student_Information();
    }
}
```

Output:

```
Name of the Student :Rajeev
Age of the Student :19
Enrolled in Course: "MCA"
```

3.2.3 Packages Naming

Space conflicts will arise if two pieces of code declare the same name. The java runtime system internally keeps track of what belongs to each package. For example, suppose someone decides to put a Test class in **myJava.io**, it won't conflict with a Test class defined in **myJava.net** package. Since they are in different packages Java can tell them apart. Just as you tell Mohan Singh apart from Mohan Sharma by their last names, similarly Java can tell two Test classes apart by seeing their package names.

But this scheme doesn't work if two different classes share the same package name as well as class name. It is not unthinkable that two different people might write packages called **myJava.io** with a class called Test. You can ensure that package names do not conflict with each other by prefixing all your packages in the hierarchy. More about conflict resolution we will discuss in Section 3.4.1 of this Unit.

Java provides various classes and interfaces that are members of various packages that bundle classes by function: fundamental classes are in **Java.lang** classes for reading and writing (input and output) are in **Java.io**, for applet programming **Java.applet** and so on.

Now you are aware of the need of putting your classes and interfaces in packages. Here we again look at a set of classes and try to find why we need to put them in a package.

Suppose that you write a group of classes that represent a collection of some graphic objects, such as circles, rectangles, triangles, lines, and points. You also write an interface, MoveIt is implemented to check if graphics objects can be moved from one location to another with the help of mouse or not..

We are using interface here which is discussed in Section 3.5 of this unit.

Now let us see the code snippet given below:

```
//code snippet
public abstract class Graphic
{
    ...
}
//for Circle, MoveIt is an interface
public class Circle extends Graphic implements MoveIt
{
    ...
}
//for Triangle, MoveIt is an interface
public class Triangle extends Graphic implements MoveIt
{
    ...
}
//interface MoveIt
public interface MoveIt
{
    // method to check movement of graphic objects.
}
```

If you closely observe this basic structure of the code snippet given above, you will find four major reasons to have these classes and the interface in a package. These reasons are:

- i. Any programmers can easily determine that these classes and interfaces are related.
- ii. Anybody can know where to find classes and interfaces that provide graphics-related functions.
- iii. The names of your classes won't conflict with class names in other packages, because the package creates a new namespace.
- iv. Your classes within the package can have unrestricted access to one another, yet still restrict access for classes outside the package.

Now let us create a package named graphics for the above program structure.

This code will appear in the source file Circle.Java and puts the Circle class in the graphics package.

```
package graphics;
public class Circle extends Graphic implements MoveIt
{
    ...
}
```

The Circle class is a public member of the graphics package. Similarly include the statement in Rectangle.Java, triangle.Java, and so on.

```
package graphics;
```

```
public class Rectangle extends Graphic implements MoveIt
{
    ...
}
```

Check Your Progress 1

- 1) What is import? Explain the need of importing a package.

.....

.....

.....

- 2) Write a program to create a package named AccountPack with class BankAccount in it with the method to show the account information of a customer.

.....

.....

Now we will discuss the accessibility of the classes and interfaces defined inside the packages.

3.2 ACCESSIBILITY OF PACKAGES

The scope of the *package* statement is the *entire* source file. So all classes and interfaces defined in Circle.Java and Rectangle.Java are also members of the graphics package. You must take care of one thing, that is, if you put multiple classes in a single source file, only one class *may be public*, and it must share the name of the source files. Remember that only public package members are accessible from outside the package.

If you do not use a package statement, your class or interface are stored in a *default package*, which is a package that has no name. Generally, the default package is only for small or temporary applications or when you are just beginning the development of small applications. For projects developments, you should put classes and interfaces in some packages. Till now all the example programs you have done are created in default package.

One of the major issues is how the classes and interfaces of packages can be used in programs. Now we will discuss using members of packages.

3.4 USING PACKAGE MEMBERS

Only public package members are accessible outside the package in which they are defined. To use a public package member from outside its package, one or more of the following things to be done:

1. Refer to the member by its long (qualified) name.
2. Import the package member.

3. Import an entire package.

Each of this is appropriate for different situations, as explained below:

1) Referring to a Package Member by Name

So far, in the examples in this course we have referred to classes by their simple names. You can use a package member's simple name if the code you are writing is **in the same package as that member** or if the member's package has been **imported**. However, if you are trying to use a member from a different package and that package has not been imported, you must use the member's qualified name, which includes the package name.

For example, this is the qualified name for the Rectangle class declared in the graphics package in the previous example:

```
graphics.Rectangle
```

You can use this long name to create an instance of graphics. Rectangle like this:
`graphics. Rectangle myRect = new graphics.Rectangle();`

It is okay to use long names if you have to use a name once. But if you have to write `graphics. Rectangle`, several times, you would not definitely like it. Also, your code would get very messy and difficult to read. In such cases, you can just import the member.

2) Importing a Package Member

To import a specific member into the current file, you have to put an import statement at the beginning of the file before defining any class or interface definitions. Here is how you would import the **Circle** class from the **graphics** package created in the previous section:

```
import graphics.Circle;
```

Now you can refer to the Circle class by its simple name:
`Circle myCircle = new Circle(); // creating object of Circle class`

This approach is good for situations in which you use just a few members from the graphics package. But if you have to use many classes and interfaces from a package, you should import the entire package.

3) Importing an Entire Package

To import all the classes and interfaces contained in a particular package, use the import statement with the asterisk (*). For example to import a whole graphics package write statement:

```
import graphics.*;
```

Now you can refer to any class or interface in the graphics package by its short name as:

```
Circle myCircle = new Circle();  
Rectangle myRectangle = new Rectangle();
```

The asterisk in the import statement can be used only to specify all the classes within a package, as shown here. It cannot be used to match only a subset of the classes in a

package. With the import statement, you can import only a single package member or an entire package.

For your convenience, the Java runtime system automatically imports two entire packages: the **Java.lang package** and the **current package** by default.

Resolving Name Conflicts

There may be situations in which a member in one package shares the same name with a member in another package and both packages are imported. In these situations you must refer to each member by its *qualified name*. For example, the previous example defined a class named Rectangle in the **graphics** package. The Java.awt package also contains a Rectangle class. If both **graphics** and **Java.awt** have been imported, in this case the following is ambiguous:

Rectangle rect;

In such a situation, you have to be more specific and use the member's qualified name to indicate exactly which Rectangle class you want, for example, if you want Rectangle of graphics package then write:

graphics. Rectangle rect; //rect is object of graphics. Rectangle class

And if you Rectangle class of Java.awt package then write:

Java.awt. Rectangle rect; //rect is object of Java.awt. Rectangle class

Check Your Progress 2

- 1) Write two advantages of packages.

.....

.....

.....

- 2) Write a program to show how a package member is used by name.

.....

.....

.....

- 3) When does name conflict arise in package use? How to resolve it?

.....

.....

.....

Java does not support multiple inheritance. But if you have some implementation which needs solutions similar to multiple inheritance implementation. This can be done in Java using interfaces. Now let us see how to define and implement interfaces.

3.5 INTERFACES

Interfaces in Java look similar to classes but they don't have instance variables and provides methods that too without implementation. It means that every method in the

interface is strictly a declaration.. All methods and fields of an interface must be public.

Interfaces are used to provide methods to be implemented by class(es). A class implements an interface using ***implements*** clause. If a class is implementing an interface it has to define all the methods given in that interface. More than one interface can be implemented in a single class. Basically an interface is used to define a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. As Java does not support multiple inheritance, interfaces are seen as the alternative of multiple inheritance, because of the feature that multiple interfaces can be implemented by a class.

An interface is like a class with nothing but abstract methods and final, static fields. All methods and fields of an interface must be public.

Interfaces are useful because they:

- Provide similarities among unrelated classes without artificially forcing a class relationship.
- Declare methods that one or more classes are expected to implement.
- Allow objects from many different classes which can have the same type. This allows us to write methods that can work on objects from many different classes, which can even *be in different inheritance hierarchies*.

Now let us see how interfaces are defined.

Defining an Interface

Though an interface is similar to a class, there are several restrictions to be followed:

- An interface does not have instance variable.
- Every method of an interface is abstract.
- All the methods of an interface are automatically public.

Figure 1 shows that an interface definition has two components:

- i. The interface declaration
- ii. The interface body.

The interface declaration declares various attributes of the interface such as its name and whether it extends another interface. The interface body contains the constant and method declarations within that interface.

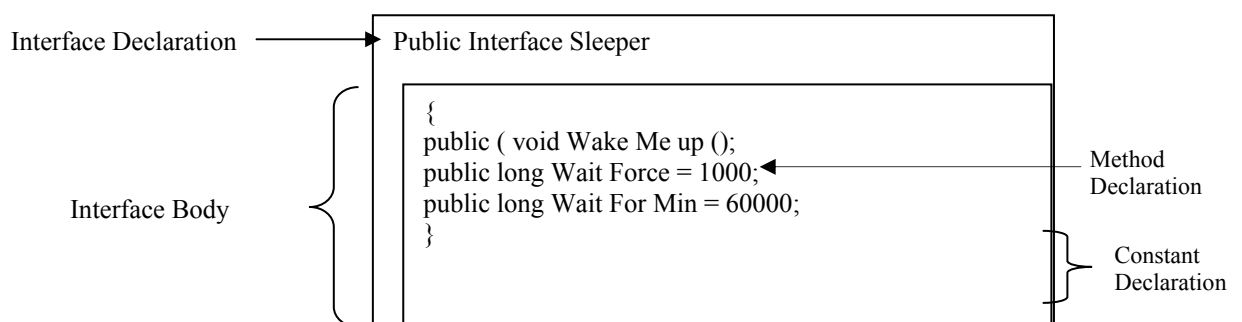


Figure 1: Interface Declaration

Interface defining is similar to defining a class. You can define an interface as given below:


```

access_specifier interface Name_of_Interface
{
return_type method1(patameters);
return_type method2(patameters);
return_type method3(patameters);
.....
return_type methoN(patameters);
type variable_Name1;
type variable_Name2;
}

```

In this declaration `access_specifier` is either `public` or not given. If no specifier is given then interface is available only to the members of the package in which it is declared. If specifier is given as `public`, then it is available to all. As you can see, variables also can be declared within inside of interface but they are *final* and *static* by default. Classes implementing this interface cannot change these variables. All the methods and variables are by default *public*, if interface is *public* otherwise they are visible to the package in which interface is declared.

Unlike a class, a interface can be added to a class that is already a subclass of another class. A single interface can apply to members of many different classes. For example you can define a Calculate interface with the single method `calculateInterest()`.

```

public interface Calculate
{
public double calculateInterest();
}

```

Now you can use this interface on many different classes if you need to calculate interest. It would be inconvenient to make all these objects derive from a single class. Furthermore, each different type of system is likely to have a different means of calculating the interest. It is better to define a Calculate interface and declare that each class implements Calculate.

Many different methods can be declared in a single interface. The method of an interface may be overloaded. Also in an interface may extend other interfaces just as a class extend or subclass another class.

3.6 IMPLEMENTING INTERFACES

As discussed earlier an interface can be implemented by a class using `implements` clause. The class implementing an interface looks like

```

access_specifier class className implements interfaceName
{
//class body consisting of methods definitions of interface
}

```

To declare a class that implements an interface, include an `implements` clause in the class declaration. Your class can implement more than one interface (the Java platform supports multiple interface inheritance), so the `implements` keyword is followed by a comma-separated list of the interfaces implemented by the class. Remember that when a class implements an interface, it is essentially signing a contract with the interface that the class must provide method implementations for all of the methods declared in the interface and its superinterfaces. If a class does not implement all the methods declared in the interface the class must be declared `abstract`. The method signature (the name and the number and type of arguments) for the method in the class must match the method signature as it appears in the interface.

The interface :

```
public interface Sleeper
{
    public void wakeUpMe();
    public long WaitForSec= 1000;
    public long WaitForMin=60000;
}
```

declares two constants that are useful arguments to letMeSleep. All constant values defined in an interface are implicitly public, static, and final. The use of these modifiers on a constant declaration in an interface is discouraged as a matter of style. Any class can use an interface's constants from the name of the interface, like this: Sleeper.WaitForSec

Classes that implement an interface can treat the constants as they were inherited. This is why ControlClock can use WaitForsec directly when calling letMeSleep:

```
public class ControlClock extends Applet implements Sleeper
{
    ...
    public void wakeUpMe()
    {
        repaint();
        clock.letMeSleepr(this, WaitForSec);
    }
}
```

Note: Member declarations in an interface does not allow for some declaration modifiers; you may not use transient, volatile, or synchronized in a member declaration in an interface. Also, you may not use the private and protected specifiers when declaring members of an interface.

Again let us take example of interface Calculate and implement it in a class:

```
public interface Calculate
{
    public double calculateInterest();
}
```

In the following program interface Calculate is implemented by SavingAccount class.

```
//program
interface Calculate
{
    public double calculateInterest();
}
class SavingAccount implements Calculate
{
    int Ac_No;
    int Amount;
    double Rate_of_Int;
    int time;
    SavingAccount( int a , double b, int c, int d)
    {
        Ac_No = a;
        Rate_of_Int = b;
        time = c;
        Amount = d;
    }
    void DisplayInt()
```

```

{
System.out.println("Interest for Account No. "+Ac_No+" is Rs "+calculateInterest());
}
public double calculateInterest( )
{
return( Amount*Rate_of_Int*time/100);
}
}
public class TestInterface
{
public static void main( String args[])
{
SavingAccount S = new SavingAccount( 1010,4.5,5,5000);
S.DisplayInt();
}
}

```

Output:

Interest for Account No. 1010 is Rs 1125.0

Interface and Inheritance

One interface can inherit another interface using **extends** keyword. This is syntactically the same as inheriting classes. Let us say one interface A is inhering interface B. At the same time a class C is implementing interface A. Then class C will implement methods of both the interfaces A and B.

See the program given below to show inheritance of interfaces.

```

//program
interface One
{
void MethodOne( );
}
interface Two extends One
{
public void MethodTwo();
}
class interfaceInherit implements Two
{
public void MethodOne()
{
System.out.println("Method of interface One is implemented");
}
public void MethodTwo()
{
System.out.println("Method of interface Two is implemented");
}
}
class Test
{
public static void main( String args[])
{
System.out.println("Interface Inheritance Demonstration");
interfaceInherit object = new interfaceInherit() ;
object.MethodOne();
object.MethodTwo();
}
}

```

Output:

Interface Inheritance Demonstration
Method of interface One is implemented
Method of interface Two is implemented.

3.7 INTERFACE AND ABSTRACT CLASSES

Abstract classes and interfaces carry similarity between them that methods of both should be implemented but there are many differences between them. These are:

- A class can implement more than one interface, but an abstract class can only subclass one class.
- An abstract class can have non-abstract methods. All methods of an interface are implicitly (or explicitly) abstract.
- An abstract class can declare instance variables; an interface cannot.
- An abstract class can have a user-defined constructor; an interface has no constructors.
- Every method of an interface is implicitly (or explicitly) public.
- An abstract class can have non-public methods.

3.8 EXTENDS AND IMPLEMENTS TOGETHER

By convention the implements clause follows the extends clause, if it exists. You have seen that the Control Clock class implements the Sleeper interface, which contains a single method declaration for the wakeUpMe method. Here Applet class is extended first then Sleeper interface is implemented.

```
public class Control Clock extends Applet implements Sleeper
{
    ...
    public void wakeUpMe()
    {
        // update the display
    }
}
```

Again I remind you that when a class implements an interface, it is essentially signing a contract. The class must provide method implementations for all of the methods declared in the interface and its superinterfaces. The method signature (the name and the number and type of arguments) for the method in the class must match the method signature as it appears in the interface.

Check Your Progress 3

- 1) What is an interface?

.....
.....

- 2) Write a program to show how a class implements two interfaces.

.....
.....
.....

- 3) Show through a program that fields in an interface are implicitly static and final and methods are automatically public.

.....

.....

.....

3.9 SUMMARY

In this unit the concepts of packages and interfaces are discussed. This unit explains the need of packages, how to create package, and the need of setting CLASSPATH variable is discussed. It is also discussed how members of packages are imported in a program and how to remove name conflicts if any. In this unit we have also discussed the need of Interfaces, how interfaces are declared, how an interface is implemented by a class. In the last two sections of this unit we have discussed differences between abstract classes and interfaces and explained how a class may inherit from a class and implement interfaces.

3.10 SOLUTIONS/ANSWERS

Check your Progress 1

- 1) A package is a collection of related classes and interfaces, providing access protection and naming space management.

To define a package keyword package is used. The code statements:

```
package MyOwnPackage;
// class and interfaces
public class A
{
// code
}
public interface I
{
//methods declaration
}
define a package named MyOwnPackage.
```

- 2) CLASSPATH is a system variable used to provide the root of any package hierarchy to Java compiler.

To set CLASSPATH see Java programming section of MCSL–025 course .

3)

```
//program
package AccountPaack;
class Account
{
String Name;
int AccountNo;
String Address;
Account( String n, int a, String c)
{
```

```
Name = n;
AccountNo = a;
Address = c;
}
void Account_Information()
{
System.out.println("Name of the Account holder :"+ Name);
System.out.println("Account Number:"+AccountNo);
System.out.println("Address of Account holder :"+Address);
}
}
class AccountTest
{
public static void main( String args[])
{
Account Ach1 = new Account("Rajeev",110200123, "28-K Saket, New Delhi");
Account Ach2 = new Account("Naveen",110200113, "D-251,Sector-55, Noida");
Ach1.Account_Information();
Ach2.Account_Information();
}
}
```

Output:

```
Name of the Account holder :Rajeev
Account Number:110200123
Address of Account holder :28-K Saket, New Delhi
Name of the Account holder :Naveen
Account Number:110200113
Address of Account holder :D-251,Sector-55, Noida
```

Check your Progress 2

- 1) import is a Java keyword used to include a particular package or to include specific classes or interfaces of any package.
import keyword is used to:
 - either include whole package by statement
import PackageName.*;
 - or to import specific class or interface by statement
import PackageName.ClassName(or InterfaceName).*;
- 2) Naming conflict arise when one package share the same name with a member another package and both packages are imported.
Naming conflict can be resolved using qualified name of the members to be imported.

Check your Progress 3

- 1) Interfaces can be seen as a group of methods declaration that provides basic functionality to classes that share common behavior. Java allows a class to implement multiple interfaces and by this Java try to fill the gap of not supporting multiple inheritance. In Java a class implementing an interface must have to all the methods of in that interface otherwise this class has to be abstract class.
- 2)

```
//program to implement two interfaces in a single class
interface First
```

```

{
void MethodOne( );
int MyValue1 = 100;
}
interface Second
{
public void MethodTwo();
int MyValue2 = 200;
}
class interfaceIn implements First,Second
{
public void MethodOne()
{
System.out.println("Method of interface First is implemented with
value:"+MyValue1);
}
public void MethodTwo()
{
System.out.println("Method of interface Second is implemented with
value:"+MyValue2);
}
}
class TwoInterfTest
{
public static void main( String args[])
{
interfaceIn object = new interfaceIn() ;
object.MethodOne();
object.MethodTwo();
}
}
Output:
Method of interface First is implemented with value:100
Method of interface Second is implemented with value:200

```

```

3)
//program
interface MyInterface
{
void MyMethod( );
int MyValue1 = 500;
}
class MyinterfaceImp implements MyInterface
{
public void MyMethod()
{
System.out.println("Method of interface MyInterface is implemented with
value:"+MyValue1);
//MyValue1 += 50;
}
}
class InterfaceTest
{
public static void main( String args[])
{
MyinterfaceImp object = new MyinterfaceImp() ;
object.MyMethod();//myMethod is by default public.
}
}

```

Output:

Method of interface MyInterface is implemented with value:500

If you remove comment (//) from statement “//MyValue1 += 50;” and compile this program you will get :

----- Compile -----

InterfaceTest.java:12: cannot assign a value to final variable MyValue1

MyValue1 += 50;

^

1 error

This error is because of attempt to modify final variable MyValue1.