Packages

Packages

- ✓ Provides a mechanism for grouping a variety of classes and / or interfaces together.
- ✓ Grouping is based on functionality.

Benefits:

- ✓ The classes contained in the packages of other programs can be reused.
- ✓ In packages, classes can be unique compared with classes in other packages.
- ✓ Packages provides a way to hide classes.

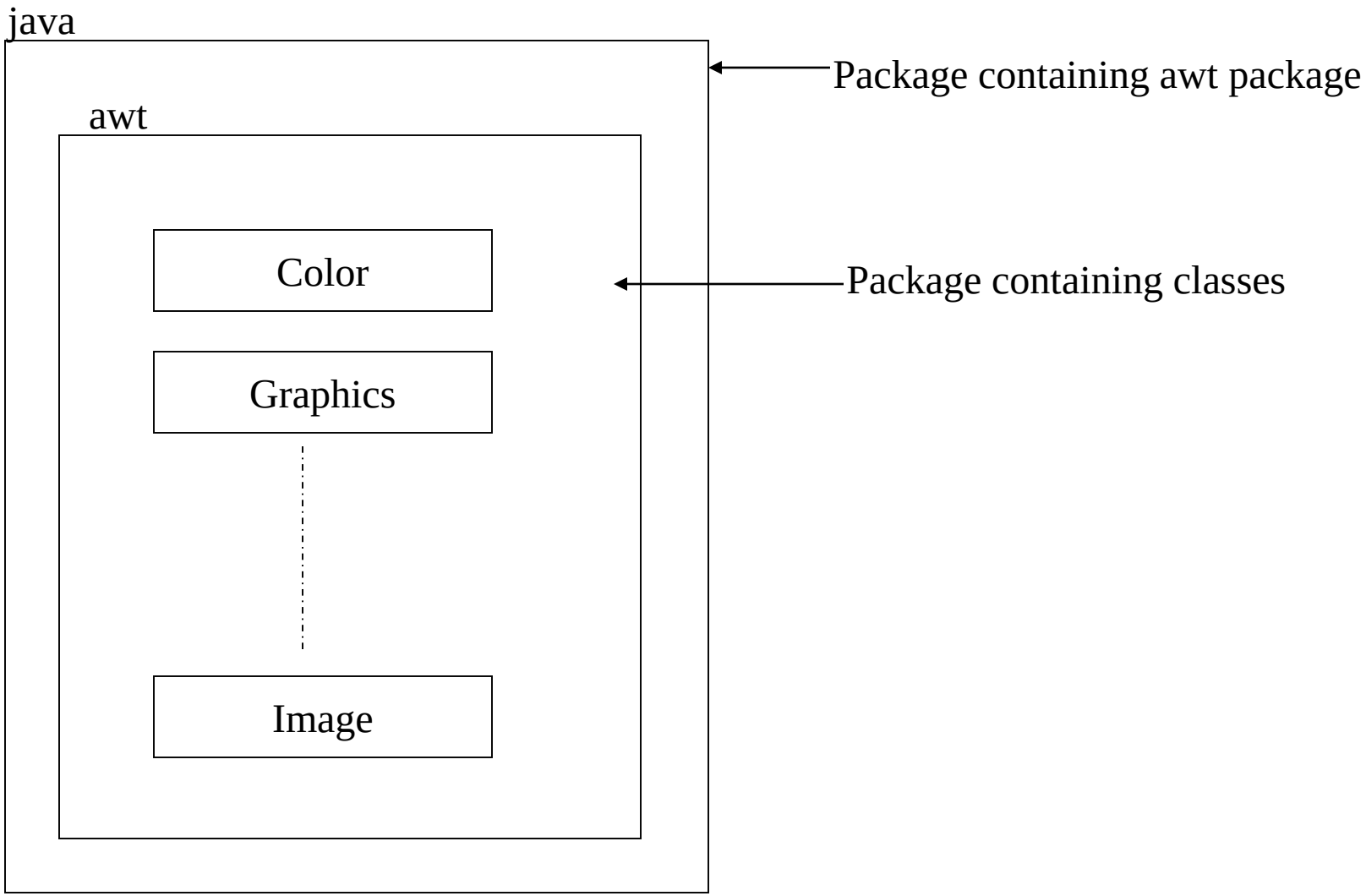
Packages

- ✓ Two types of packages:
  1. Java API packages
  2. User defined packages

<u>Java API Packages:</u>

- ✓ A large number of classes grouped into different packages based on functionality. Examples:
  1. java.lang
  2. java.util
  3. java.io
  4. java.awt
  5. java.net
  6. java. applet etc.

Package


java
Package containing awt package
awt
Color
Package containing classes
Graphics
Image

Accessing Classes in a Package

1. Fully Qualified class name:

   Example:java.awt.Color

2. **import** packagename.classname;

   Example: import java.awt.Color;

   or

   import packagename.*;

   Example: import java.awt.*;

✓ Import statement must appear at the top of the file, before any class declaration.

Creating Your Own Package

1. Declare the package at the beginning of a file using the form

   package *packagename;*

2. Define the class that is to be put in the package and declare it **public**.
3. Create a subdirectory under the directory where the main source files are stored.
4. Store the listing as classname.java in the subdirectory created.
5. Compile the file. This creates .class file in the subdirectory.

Example:

```
package firstPackage;

public class FirstClass
{
    //Body of the class
}
```

Example1-Package

```
package p1;
public class ClassA
{
    public void displayA( )
    {
    System.out.println("Class A");
    }
}
```

Source file – ClassA.java

Subdirectory-p1

ClassA.Java and ClassA.class->p1

```
import p1.*;

Class testclass
{
    public static void main(String str[])
    {
        ClassA obA=new ClassA();
        obA.displayA();
    }
}
```

Source file-testclass.java

testclass.java and testclass.class->in a directory of which *p1* is subdirectory.

Example2-Package

```
package p2;
public class ClassB
{
    protected int m =10;
    public void displayB()
    {
    System.out.println("Class B");
    System.out.println("m= "+m);
    }
}
```

```
import p1.*;
import p2.*;

class PackageTest2
{
    public static void main(String str[])
    {
        ClassA obA=new ClassA();
        Classb obB=new ClassB();
        obA.displayA();
        obB.displayB();
    }
}
```

Example 3- Package

```
import p2.ClassB;
class ClassC extends ClassB
{
    int n=20;
    void displayC()
    {
    System.out.println("Class C");
    System.out.println("m= "+m);
    System.out.println("n= "+n);
    }
}
```

```
class PackageTest3
{
    public static void main(String args[])
    {
        ClassC obC = new ClassC();
        obC.displayB();
        obC.displayC();
    }
}
```

Default Package

- ✓ If a source file does not begin with the *package* statement, the classes contained in the source file reside in the *default package*
- ✓ The java compiler automatically looks in the default package to find classes.

Finding Packages

- ✓ Two ways:

  1.By default, java runtime system uses current directory as starting point and search all the subdirectories for the package.

  2.Specify a directory path using CLASSPATH environmental variable.

CLASSPATH Environment Variable

- ✓ The compiler and runtime interpreter know how to find standard packages such as *java.lang* and *java.util*
- ✓ The CLASSPATH environment variable is used to direct the compiler and interpreter to where programmer defined imported packages can be found
- ✓ The CLASSPATH environment variable is an ordered list of directories and files

CLASSPATH Environment Variable

- ✓ To set the CLASSPATH variable we use the following command:

  set CLASSPATH=c:\

- ✓ Java compiler and interpreter searches the user defined packages from the above directory.
- ✓ To clear the previous setting we use:

  set CLASSPATH=

Packages :-Access Protection or Visibility Control

	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 subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Thank You

Example1-Package[Using CLASSPATH]

```
package p1;
public class ClassA
{
    public void displayA( )
    {
    System.out.println("Class A");
    }
}
```

```
import p1.ClassA;

Class PackageTest1
{
    public static void main(String str[])
    {
        ClassA obA=new ClassA();
        obA.displayA();
}}
```

Source file – c:\p1\ClassA.java

Compile-javac c:\p1\ClassA.java

Class file in –

Source file- c:\java\jdk1.6.0_06\bin\PackageTest1.java

Compile-javac PackageTest1.java

Copy –PackageTest1.class -> c:

Example2-Package[Using CLASSPATH]

```
package p2;
public class ClassB
{
    protected int m =10;
    public void displayB()
    {
    System.out.println("Class B");
    System.out.println("m= "+m);
    }
}
```

Source file – c:\p2\ClassB.java

Compile-c:\p2\ClassB.java

Class file in – c:\p2\ClassB.class

```
import p1.*;
import p2.*;
class PackageTest2
{
    public static void main(String str[])
    {
        ClassA obA=new ClassA();
        Classb obB=new ClassB();
        obA.displayA();
        obB.displayB();}  }
```

Source file- c:\java\jdk1.6.0_06\bin\PackageTest2.java

Compile-javac PackageTest2.java

Copy –PackageTest2.class -> c:

Example 3- Package[Using CLASSPATH]

```
import p2.ClassB;
class ClassC extends ClassB
{
    int n=20;
    void displayC()
    {
    System.out.println("Class C");
    System.out.println("m= "+m);
    System.out.println("n= "+n);
    }
}
```

Source file – c:\ClassC.java

Compile-c:\ClassC.java

Class file in –c:\ClassC.class

```
class PackageTest3
{
    public static void main(String args[])
    {
        ClassC obC = new ClassC();
        obC.displayB();
        obC.displayC();
    }
}
```

Source file- c:\java\jdk1.6.0_06\bin\PackageTest3.java

Compile-javac PackageTest3.java

Copy –PackageTest3.class -> c:

Adding a Class to a Package

- ✓ Every java source file can contain only class declared as **public.**
- ✓ The name of the source file should be same as the name of the public class with **.java** extension.

package p1;

public ClassA{
……}

Source file : ClassA.java

Subdirectory:p1

package p1;

public ClassB{……}

Source file: ClassB.java

Subdirectory:p1

Adding a Class to a Package

1. Decide the name of the package.
2. Create the subdirectory with this name under the directory where the main source file is located.
3. Create classes to be placed in the package in separate source files and declare the package statement

package packagename;

4. Compile each source file. When completed the package will contain .class files of the source files.

public/package/private scope

- ✓ Scope is concerned with the visibility of program elements such as classes and members
- ✓ Class members (methods or instance fields) can be defined with public, package (default), private or protected scope
- ✓ A class has two levels of visibility:

  -**public** scope means it is visible outside its containing package

  - default scope means it is visible only inside the package. (package scope/ friendly scope)

public/package/private scope

- ✓ A class member with **public** scope means it is visible anywhere its class is visible
- ✓ A class member with **private** scope means it is visible only within its encapsulating class
- ✓ A class/class member with **package** scope means it is visible only inside its containing package
- ✓ A class member with **protected** scope means it is visible every where except the non-subclasses in other package.

Example 1

```
package my_package;

class A        // package scope
{
    // A's public & private members
}

public class B      // public scope
{
    // B's public and private members
}
```

Example-2

```
package my_package;

class D
{
    // D's public & private members

    // Class D 'knows' about classes A and B

    private B b;        // OK – class B has public scope
    private A a;    // OK – class A has package scope

}
```

Example-3

```
package another_package;
import my_package.*;

class C
{
    // C's public & private members

    // class C 'knows' about class B

    private B b;        // OK – class B has public scope

}
```

Example 4

```
package my_package;
class A
{
    int get() { return data; }      // package scope
    public A(int d) { data=d;}      // public scope
    private int data;               // private scope
}

class B
{
    void f()
    {
        A a=new A(d);       // OK A has package scope
        int d=a.get();      // OK – get() has package scope
        int d1=a.data;      // Error! – data is private
    }
}
```

Levels of Access Control

	public	protected	friendly (default)	private
same class	Yes	Yes	Yes	Yes
Subclass in the same package	Yes	Yes	Yes	No
Other class in the same package	Yes	Yes	Yes	No
Subclass in other packages	Yes	Yes	No	No
Non-subclass in other package	Yes	No	No	No

Interface

- ✓ Similar to a class.
- ✓ Consists of only abstract methods and final variables.
- ✓ Any number of classes can implement an interface.
- ✓ One class can implement any number of interfaces.
- ✓ To implement an interface a class must define each of the method declared in the interface. Each class can also add new features.
- ✓ Interface disconnect the definition of a method or set of methods from the inheritance hierarchy.

Defining an Interface

- General form of an interface:

  access interface *name* {

  ret-type *method1*(parameter list);

  ret-type *method2*(parameter list);

  ⋮

  type final *var1* = value;

  type final static *val2* = value;

}

<u>Example:</u>

```
interface callback{
    void callback (int param);
}
```

Defining an Interface

- ✓ *Access* is either **public** or **default**.
- ✓ Variables declared inside an interface are implicitly **final** and **static.**
- ✓ Variables must be initialized with a constant value.
- ✓ All methods and variables are implicitly **public** if the interface, itself, is declared as **public**.

Implementing Interfaces

- ✓ The General Form:

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

  }
- ✓ The methods that implement an interface must be declared *public*.
- ✓ The type signature of the implementing method must match exactly the type signature specified in the interface definition.

Accessing Implementations through Interface Reference

- ✓ Interface reference is required to access the implementation.
- ✓ Any instance of the class that implements the interface can be referred to by such a variable.
- ✓ When a method is called through one of the reference, the correct version will be called based on the actual instance of the interface being referred to.
- ✓ The method to be executed is looked up dynamically at run time.

Example-1

```
interface call
{
	void callback(int param);
}

class client implements call
{
	public void callback(int p)
	{
		System.out.println("callback called with "+p);
	}
}

public class testIface
{
	public static void main(String args[])
	{
		call c = new client();
		c.callback(423);
	}
}
```

Example-2

```
interface call
{
	void callback(int param);
}

class client implements call
{
	public void callback(int p)
	{
System.out.println("callback is called with "+p);
	}
}
class anotherclient implements call
{
	public void callback(int p)
	{
	  System.out.println("p squred is "+(p*p));
	}
}
```

```
public class testIface
{
public static void main(String args[])
	{
		call c = new client();
		c.callback(42);
		c=new anotherclient();
		c.callback(10);
	}
}
```

Partial Implementation

- ✓ If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract**.
- ✓ Example:

```
abstract class temp implements call{
int a, b;
void show()
{
        //body of the method
}
}
```

- ✓ Any class that inherits *temp* must implement callback() or declared abstract itself.

Extending Interfaces

- ✓ One interface can inherit another by using the keyword **extends**.
- ✓ The new sub interface will inherit all the member of the super interface.
- ✓ Any class that will implement the interface that inherits another interface, it must provide implementations of all methods defined within the interface inheritance chain.
- ✓ General form:

```
interface name2 extends name1
{
        //body of name2
}
```

- ✓ Example:

```
interface ItemConstant
{
        int code =1001;
        String name =“Pen”;
}
```

```
interface Item extends ItemConstant
{
        void display();
}
```

- ✓ **An interface cannot extends a class.**

Multiple Inheritance Using Interface

- ✓ Java supports multiple inheritance through the use of interface.
- ✓ Care should be taken to avoid some conflicts.

Example-3

```
interface test1
{
    int val=10;
    void display();
}
interface test2
{
    int val=20;
    void display();
}
```

```
class test3 implements test1, test2
{
    public void display()
    {
    System.out.println("In test3");
    System.out.println(test1.val);
    System.out.println(test2.val);
    }
}
```

Example-4

```
interface test1
{
    int val=10;
    void display();
}
interface test2
{
    int val=20;
    void display();
}
interface test3 extends test1, test2
{
    int val=50;
    void display();
}
class test4 implements test3
{
    int val=57;
    public void display()
    {
    System.out.println(test1.val);
    System.out.println(test2.val);
    System.out.println(test3.val);
    System.out.println(val);

    }
}
public class Iface_test
{
public static void main(String args[])
{
    test4 ob = new test4();
    ob.display();
}
}
```

Example-5

```
interface test1
{
    int val=33;
    void display();
}
class test2 implements test1
{
    static int val=34;
    void display()
    {
    System.out.println(test1.val);
    System.out.println(val);
    }
}
```

```
class test3 extends test2
{
    int val=35;
    void show()
    {
    System.out.println(test1.val);
    System.out.println(test2.val);
    System.out.println(val);
    }
}
class test4
{
    public static void main(String args[])
    {
    test3 ob = new test3();
    ob.show();
    }
}
```