OBJECT ORIENTED PROGRAMMING WITH JAVA

Packages in Java - I

Debasis Samanta
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Concept of Package in Java

Two unique features in Java

Java's two most innovative features are:

- Packages
- Interfaces

What is a package?

A package is a container for the classes that are used to keep the class name space compartmentalized

Example:
You can contain all classes related to all sorting programs in your own package.

Why packages?

It allows flexibility to give same name but to many classes, that is to avoid name space collision.

The packages in Java provides mechanism for partitioning the class name space into more manageable chunks.

In fact, package is both a naming and a visibility control mechanism.

It supports reusability and maintainability.

Advantages of Packages

Packages provide code reusability, because a package contains group of classes.

It helps in resolving naming collision when multiple packages have classes with the same name.

Package also provides the hiding of class facility. Thus, other programs cannot use the classes from hidden package.

Access limitation can be applied with the help of packages.

Nesting of a package, that is, one package can be defined in another package in a hierarchy fashion.

Built-in Packages in Java

Built-in packages in Java

Packages are nothing more than the way we organize files into different directories according to their functionality, usability as well as category they should belong to . A Java package is a Java programming language mechanism for organizing classes into name spaces.

In Java, already many predefined packages are available, those are to help programmers to develop their software in an easy way.

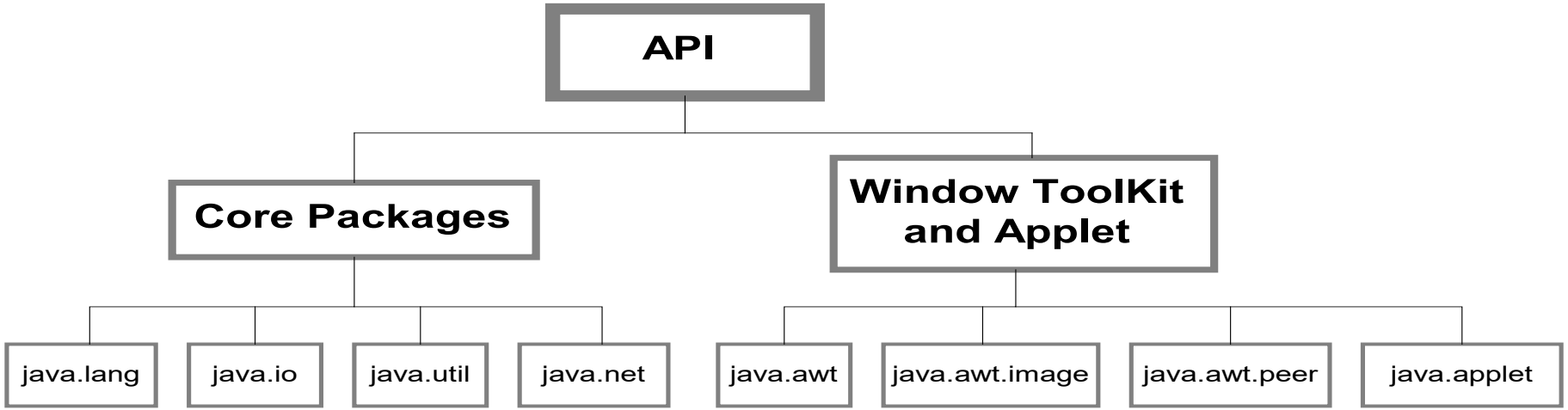
Example:

`javax.swing` is a package in Java providing all basic supports in developing GUI programs.

Packages in Java

- Code reusability is the main philosophy of Object-Oriented Programming.
- To power this advantage, Java has a number of packages called API bundled with the JDK.
- Packages are collection of classes and interfaces to facilitate a number of ready made solutions.
- A great knowledge of packages helps a Java developer to master in Java solution.
- In addition to the API, user can maintain their own packages.

API – The Built-in Java packages



`javax.swing` is the latest addition in Java version 5 and above.

Using API packages

- A package is a collection of classes and each class is a collection of members and methods.
- Any class as well as any member and method in a package are accessible from a Java program
- This can be achieved in Java by **`import`** statement.

With fully quantified class name

- When it is required to access a particular class
- Example: java.lang.String

With default (.*) quantification

- When it is required to access a number of classes
  - Example: java.lang.*

Using API packages

However, instead of importing whole package or a class it is possible to refer a class in order to instantiate an object.

Example:

```
java.util.Date toDay = new java.util.Date ( );
System.out.println(toDay);
```

The same thing but with **`import` statement** can be done as follows

```
import java.util.Date;
Date today = new Date( );
```

Defining Packages in Java

User defined packages

- User can maintain their own package.
- Java uses file system directories to store packages.
- The **`package`** statement can be used for the purpose with the following syntax

```
package myPackage;
public Class myClass {
. . .
. . .
}
```

Here, `myPackage` is the name of the package and it contains the class `myClass.`

Package naming conventions

- Packages are usually defined using a hierarchical naming pattern, with levels in the hierarchy separated by periods (.) .

- Although packages lower in the naming hierarchy are often referred to a "sub packages" of the corresponding packages higher in the hierarchy, there is no semantic relationship between packages.

Organizational package naming conventions

- Package names should be all lowercase characters whenever possible.
- Frequently, a package name begins with the top level domain name of the organization and then the organization's domain and then any subdomains listed in the reverse order.
- The organization can then choose a specific name for their package.

User defined packages

In addition to this, following steps must be taken into consideration

- Use package statement at the beginning of the package file.
- Define the class that is to be put in the package and declare it as public.
- Create a subdirectory under the working directory with the same name as the package name.
- Store the file with the same name as the className.java in the subdirectory created.
- Store the compiled version (i.e., .class) file into the same sub-directory.

Package design guidelines

Design guideline package cohesion

- Only closely related classes should belong to the same package.
- Classes that change together should belong to the same package.
- Classes that are not reused together should not belong to the same package.

Package declaration

Package declaration is file based

- All classes in the same source file belong to the same package.
- Each source file may contain an optional package declaration in the following form.

```
package <PackageName>;
```

➢ Let us consider the source file `ElevatorFrame.java`, for example.

```
package elevator;
public class ElevatorFrame {
    public double x;
    //……..
}
```

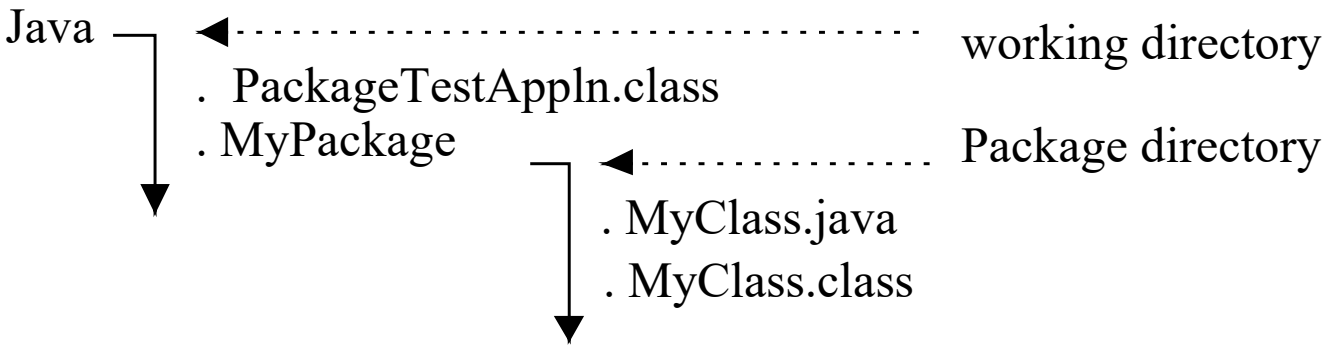
Package declaration

- The package declaration at the top of the source file declares that the `ElevatorFrame` class belongs to the package named `elevator`.
- When the package declaration is absent from a file, all the classes contained in the file belong to unnamed package.
- A class in a named package can be referred in two ways.

User defined packages: Example

```
// User defined package in a subdirectory, say MyPackage
package MyPackage;
public class MyClass {
     public void test ( ) {
          System.out.println ( " Welcome to My Class !");
     }
}
/* Import the package with the following code. From another distant file/
program */
     import MyPackage.MyClass;
     class PackageTestAppln {
          public static void main ( String args [ ] ) {
               MyClass theClass = new MyClass ( );
                theClass.test ( );
          }
     }
```

User defined packages: Example



More on user defined packages

Note:

We cannot put two or more public classes together in a .java file; otherwise there will be an ambiguity in naming the .java file

? How a package then contains multiple classes?

Solution:

```
package P;
public class A {
   . . .
}
```

```
package P;
public class B {
   . . .
}
```

More on user defined packages

```
package P;
public class A {
   . . .
}
```

```
package P;
public class B {
   . . .
}
```

Steps

1. Create a directory named **P**.
2. Store the `class A` in the file `A.java` in it
3. Compile `A.java` and place it in the directory **P**.
4. Store the `class B` in the file `B.java` in it
5. Compile `B.java` and place it in the directory **P**.
6. `import P.*;`
   will import all classes in the package **P**.

Question to think…

- How a package can be accessed in any program?
- Is it possible that two classes having the same name but in two different packages are to be used in another class outside the packages?

Thank You

OBJECT ORIENTED PROGRAMMING WITH JAVA

Packages in Java - II

Debasis Samanta
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Accessing a Package

Finding packages and CLASSPATH

How does the Java run-time system know where to look for packages that you create?

The answer has three parts

- First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a sub-directory of the current directory, it will be found.
- Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable.
- Third, you can use the -classpath option with **`java`** and **`javac`** to specify the path to your classes.

Finding packages and CLASSPATH: Example

Consider the following package specification:

```
package MyPack;
```

For a program to find *MyPack*, one of three things must be true.

- The program can be executed from a directory immediately above MyPack
- The CLASSPATH must be set to include the path to MyPack
- The -classpath option must specify the path to MyPack when the program is run via java.

When the second two options are used, the class path must not include *MyPack*, itself. It must simply specify the path to *MyPack*.

Example: In a Windows environment, if the path to *MyPack* is

```
C:\MyPrograms\Java\MyPack
```

then the class path to MyPack is

```
C:\MyPrograms\Java
```

Finding packages and CLASSPATH: Example

```
package MyPack;
class Balance {
    String name;
    double bal;
    Balance(String n, double b) {
        name = n;
        bal = b;
   }
   void show() {
       if(bal<0)
        System.out.print("Account is dead");
       else
        System.out.print(name+": $ "+bal);
   }
}
```

```
class AccountBalance {
    public static void main(String args[]) {
    Balance current[] = new Balance[3];
    current[0] = new Balance("K. J. Fielding", 123.23);
    current[1] = new Balance("Will Tell",      157.02);
    current[2] = new Balance("Tom Jackson",    -12.33);
    for(int i=0; i<3; i++)
        current[i].show();
    }
}
```

Finding packages and CLASSPATH: Example

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.

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
```

Importing a Package

Importing a package

This is the general form of the import statement:

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

Here, **pkg1** is the name of a top-level package

pkg2 is the name of a sub-ordinate 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.

Finally, you specify either an explicit class name or a star (*), which indicates that the Java compiler should import the entire package.

Example:

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

Using package

Class in a named package can be referred to in two different ways

Import a class or all the classes in the designated package using

```
import PackageName.<ClassName>;
import Packagename.*;
```

Example

The `ElevatorPanel` class in package elevator can simply be referred to as elevator when either of the following import clauses occurs at the top of source file

```
import elevator.ElevatorPanel;
import elevator.*;
```

Access Protection for Packages

Access protection

Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

- Sub classes in the same package.
- Non-sub classes in the same package.
- Sub classes in different packages.
- Classes that are neither in the same package nor sub classes.

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

Access protection

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	No	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Access protection: An example

```
//This program shows all combinations of the access control modifiers
//In this program, we define two packages and five classes.

  MyPackage1:
        class X
        class Y  extends X
        class A

  MyPackage2:
        class Z extends X
        class B
```

Access protection: An example

```
// Defining package MyPackage1
package MyPackage1;

public class X {
      int n = 1;
      private int p = 2;
      protected int q = 3;
      public int r = 4;
      // A constructor of the class protection
      public X() {
                  system.out.println("I am constructor from class X:");
                  system.out.println("n="+n);
                  system.out.println("p="+p);
                  system.out.println("q="+q);
                  system.out.println("r="+r);
      }
}
//Save this as X.java in Mypackage1 directory
```

Access protection: An example

```
package MyPackage1
class Y extends X {
     Y() {
                system.out.println("I am constructor from class Y:");
                system.out.println("n="+n);
                system.out.println("p="+p);
                // Error p is a private member of X. Not accessible outside X.
                system.out.println("q="+q);   // Protected is accessible
                system.out.println("r="+r);   // public is accessible
     }
}
//Save this as Y.java in Mypackage1 directory
```

Access protection: An example

```
// Defining package MyPackage1
package MyPackage1
public class X {
        int n = 1;
        private int p = 2;
        protected int q = 3;
        public int r = 4;
        // A constructor of the class protection
        }
//Save this as X.java in
```

```
//Save this as A.java in Mypackage1 directory
class A {                   // class with default protection
     A() {                  // default constructor with default access
          X x = new X()  // create an object of class X
          System.out.println("Same package constructor ....");
          System.out.println("n from A"+x.n);
          // Default variable is accessible in the same package
          System.out.println("p from A"+x.p); // Error
          System.out.println("q from A"+x.q); // Error protection
          System.out.println("r from A"+x.r); // OK: public
     }
}
```

Access protection: An example

```java
// Defining package MyPackage1
package MyPackage1
public class X {
        int n = 1;
        private int p = 2;
        protected int q = 3;
        public int r = 4;
        // A constructor of the class protection
        }
//Save this as X.java in Mypackage1 directory
```

```java
//Save this as Z.java in Mypackage2 directory
 package MyPackage1;
 class Z extends MyPackage1.X {
    Z() {
          System.out.println("I am constructor from class Z:");
          System.out.println("n from Z"+n); // Error:
          // Default is not accessible outside its package.
          System.out.println("p from Z"+p); // Error : private of X
          System.out.println("q from Z"+q);
                    // Protected member is accessible by inheritance
          System.out.println("r from Z"+r);
                    // public is accessible
          }
   }
```

Access protection: An example

```
//Save this as B.java in Mypackage2 directory
class B {                              // class with default protection
     B() {     // default constructor with default access
          MyPackage1.X x = new MyPackage1.X()
          // create an object of class X
          System.out.println("I am constructor from class B of MyPackage2");
          System.out.println("n from B of myPackage2"+x.n);
          // default variable but is not accessible in this package
          System.out.println("p from B of myPackage2"+x.p);
          // Error
          System.out.println("q from B of myPackage2"+x.q);
          // Error protection
          System.out.println("r from A of myPackage2"+x.r);
     }
}
```

Access protection: An example

```
//This is the demo of MyPackage1
package MyPackage1;
public class Demo1{
     public static void main(String args[])
       X x1 = new X();
       Y y1 = new Y();
       A a1 = new A();
}
```

```
//This is the demo of MyPackage2
package MyPackage2;
public class Demo2{
     public static void main(String args[])
       Z z2 = new Z();
       B a2 = new B();
}
```

Question to think…

- How multiple inheritance is possible in Java?
- How polymorphism is implementable in Java?

Thank You

OBJECT ORIENTED PROGRAMMING WITH JAVA

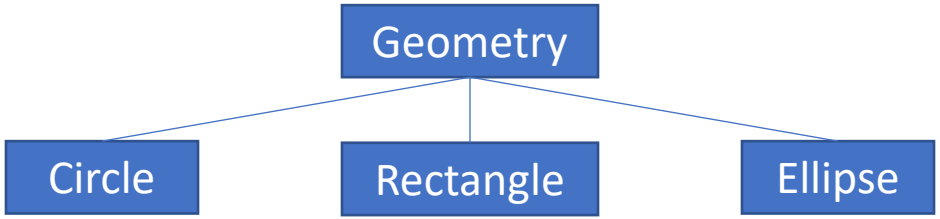
Interfaces in Java – I

Debasis Samanta
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Revisit the Abstract Class

Abstract class concept in Java

- We have discussed a number of programs with class `Circle`.
- Suppose, we want to have a number of other shapes namely `Ellipse`, `Rectangle`, `Triangle`, etc.
- All these shapes can be placed in a package, say, `myShapes`.
- Note that all these shape classes have basic operations, namely `area()` and `circumference()`.
- Lets see, how the above can be realized better using abstract class concept in Java.



Abstract class concept in Java

```
//An abstract class  and its sub class
package myShape;

public abstract class Geometry {
  static final double PI = 3.14159265358979323846;
  public abstract double area ();
  public abstract double circumference();
}
```

```
// Extending Geometry for Circle
public class Circle extends Geometry {
    public double r;
    public Circle (){
          r=1.0
    }
    public Circle (double r){
          this.r=r;
    }
    public double area() {
          return PI*r*r;
    }
    public double circumference() {
          return 2*PI*r;
    }
    public double getRadius() {
          return r;
    }
}
```

Abstract class concept in Java

```
public class Rectangle extends Geometry {
    protected double l,w;
    public Rectangle() {
          l = 0.0;
          w = 0.0;
    }

    public Rectangle(double l,    double w) {
          this.l = l;
          this.w = w;
    }
```

```
    public double area() {
          return l*w;
    }
    public double circumference() {
          return 2*(l+w);
    }

    public double getwidth() {
          return w;
    }

    public double getlength() {
          return l;
    }
}
```

Abstract class concept in Java

```
public class Ellipse extends Geometry {
    protected double a,b;
    public Ellipse() {
          a = 0.0;
          b = 0.0;
    }

    public Ellipse(double a,      double b) {
          this.a = a;
          this.b = b;
    }
```

```
    public double area() {
          return PI * a*b;
    }
    public double circumference() {
          return PI*(a+b);
    }

    public double getMinorAxis() {
          return a;
    }

    public double getMajorAxis() {
          return b;
    }
}
```

Abstract class concept in Java

```
import myShapes.*;

public class GeoDemo {
   public static void main(String args[]) {

        // use the above class definition
        Geometry [] geoObjects = new Geometry[3]
        // create an array to hold Geometry objects
        geoObjects[0] = new Circle (2.0);
        geoObjects[1] = new Rectangle (1.0,3.0);
        geoObjects[2] = new Ellipse (4.0,2.0);
        double totalArea  = 0;
        for (int i = 0; i < 3; i++) {
          totalArea = totalArea + geoObjects[i].area();
        }
        System.out.println("Total area = " + totalArea);
   }
```

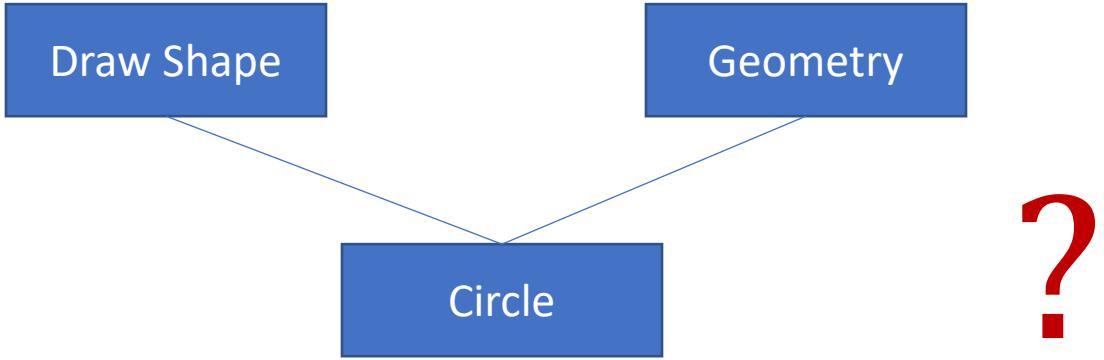
Note:

- Sub class of Geometry can be assigned to elements of an array of Geometry. No cast is required.
- One can invoke the area() and any method for the Geometry objects, even though shape does not define a body for these method, because Geometry declared them abstract .
- If Geometry did not declare them at all, the code would cause compilation error.

Few Important facts about Abstract class

- Any class with an abstract method is automatically abstract itself, and must be declared such.
- A class may be declared abstract even if it has no abstract method. This prevents it from being instantiated.
- A sub class of an abstract class can be instantiated if it overrides each of the abstract methods of its super class and provide an implementation (i.e., a method body of all of them).
- If sub class of an abstract class does not implements all of the abstract methods, it inherits, that sub class is itself abstract.

Multiple inheritance in Java



- However, this is not possible, as Java does not support multiple inheritance.
- Java's solution to this problem is called **`interface`**.

Interfaces in Java

Multiple inheritance and interface

- Java does not support **multiple inheritance.**
- Java supports an alternative approach to this OOP feature known as interface.
- What is an interface?
  - An interface is basically a kind of class. Like classes, an interface contains members and methods; unlike classes, in interface, all members are **`final`** and all methods are **`abstract`**.

Interface concept

An interface defines a protocol of behavior that can be implemented by any class anywhere in the class hierarchy.

An interface defines a set of methods but does not implement them.

A class that implements the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behavior.

An interface is a named collection of method definitions (without implementations).

Interface reserve behaviors for classes that implement them.

Methods declared in an interface are always **`public`** and **`abstract`**, therefore Java compiler will not complain if you omit both keywords. Static methods cannot be declared in the interfaces – these methods are never abstract and do not express behavior of objects.

Basic concept of inheritance

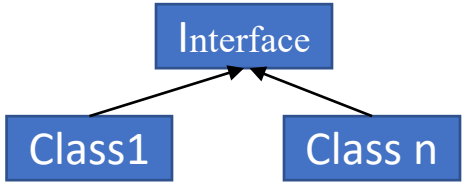
- Using the keyword `interface`, one can define an abstract class.
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are defined without any body.

Example:

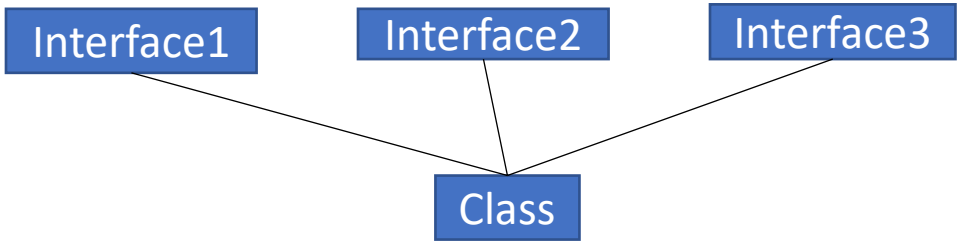
```
interface callMe {
        void call (int p);
}
```

Multiple inheritance in Java

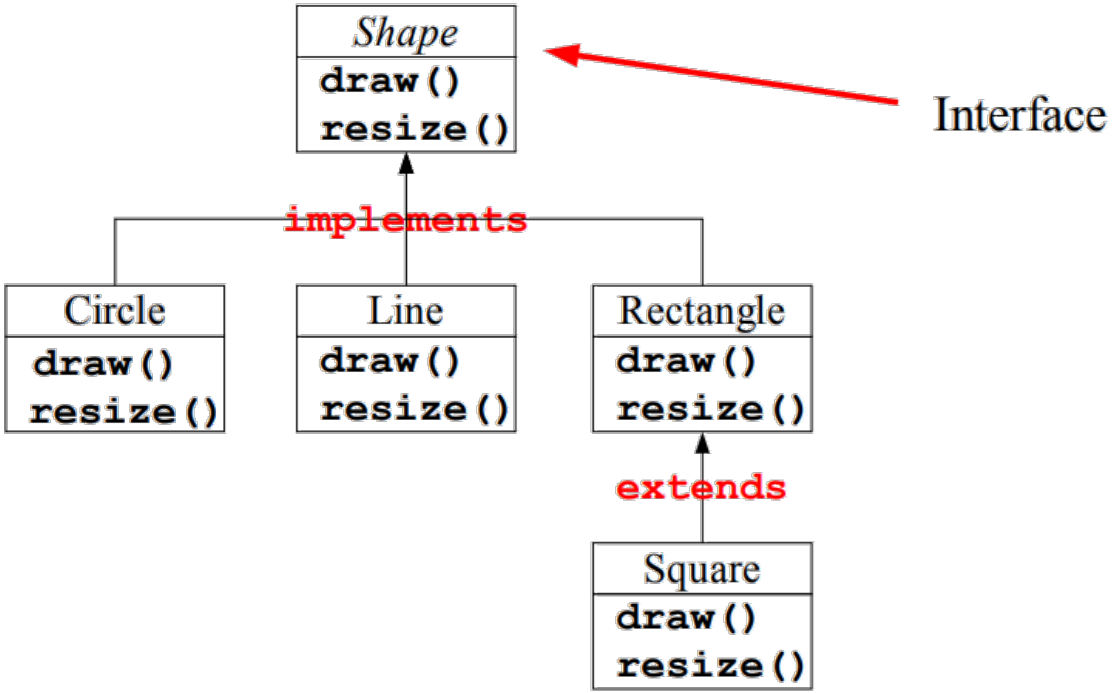
- Once an interface is defined, any number of classes can implement an interface.



- Also, one can implement any number of interfaces using a class.



Interface : An example



Properties of `interface`

Interface must be declared with the keyword **`interface`.**

All interface methods are implicitly **public** and **abstract**. In other words, you do not need to actually type the public or abstract modifiers in the method declaration, but method is still always public and abstract.

All variables defined in an interface is **public, static** and **final.**

In other words, interfaces can declare only constants, no instance variables.

Interface methods must not be **static**.

Because interface methods are abstract, they cannot be marked **final.**

An interfaces can extend one or more other interfaces.

An interface cannot implement another interface or class.

interface types can be used polymorphically.

Syntax for defining `interface`

Following is the syntax to define an interface

```
interface InterfaceName [extends name1, ..... ]
{
      [ Variable(s) declaration;]
      [ Method(s) declaration;]
}
```

Variable in an interface is declared as:

```
static final type varName = value;
```

Method in interface is declared as:

```
Return-type methodName(parameter list);
```

Defining an `interface`: Example

Example:

```
interface anItem
{
      static final int code = 101;
      static final String itemName = "Computer";
      void recordEntry( );
}
```

Defining an Interface: Examples

Example 1

```
interface Template
{
     static final int code = 101;
     static final String itemName = "Computer";
     void recordEntry( );
}
```

Example 2

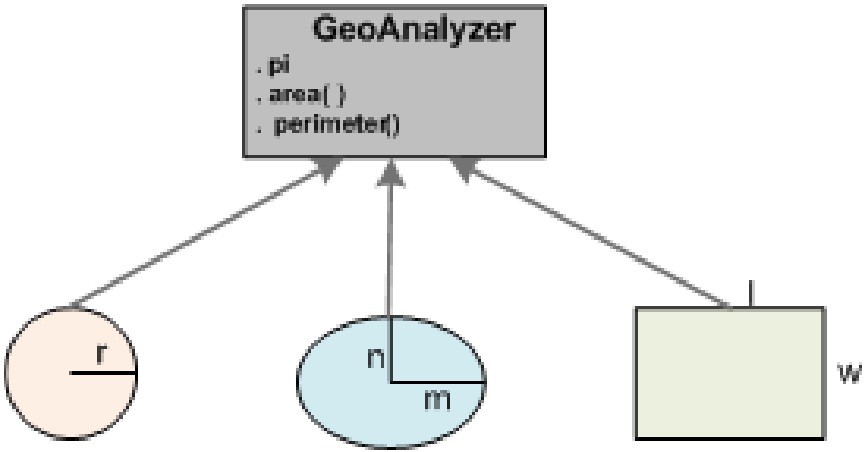
```
interface Curves extends Circle, Ellipse
{
     static final float pi = 3.142F;
     float area(int a, int b);
}
```

Implementation of classes with `interface`

Syntax

```
class className [extends superClassName]
{
    [implements interfaceName1, interfaceName2, ...]
    {
            Class body
    }
}
```

Implementation of classes with `interface`



Implementation of classes with interface: Example

```
interface GeoAnalyzer    {
    final static float pi = 3.142F;
    float area( );
    float perimeter( );
}
class Circle implements GeoAnalyzer {
       float radius;
       Circle(float r) {
              radius = r;
       }
       public float area( ) {
              return (pi*radius*radius);
       }
         public float perimeter( ) {
               return (2*pi*radius);
       }
}
```

Implementation of classes with interface: Example

```
class Ellipse implements GeoAnalyzer {
        float major;
        float minor;
        Ellipse(float m, flaot n) {
               major = m;
               minor = n;
        }
        public float area( ) {
             return(pi*major*minor);
        }
        public float perimeter( ) {
              return(pi*(major+minor);
        }
 }
```

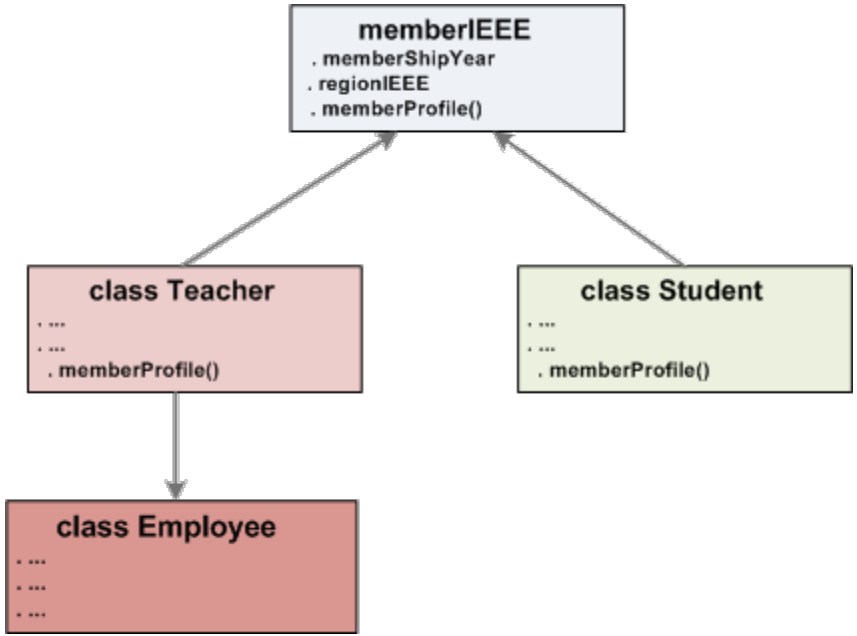
```
class Rectangle implements GeoAnalyzer
 {
        float length;
        float width;
        Rectangle(float l, float w) {
             length = l;
             width = w;
        }
        public float area() {
             return(length*width);
        }
        public float perimeter( ) {
             return(2*(length+width));
        }

}
```

Implementation of classes with interface: Example

```
class Geometry
 {
        static void display(float x, float y) {
                System.out.println("Area = " + x + "Perimeter = " + y);
        }
        public static void main(String args[ ]) {
                Circle c = new Circle(5.2);
                Ellipse e = new Ellipse(4.5, 3.6);
                Rectangle r = new Rectangle(6.5, 4.3);
                GeoAnalyzer geoItem;
                geoItem = c;
                display(geoItem.area(), geoItem.perimeter());
                geoItem = e;
                display(geoItem.area(), geoItem.perimeter());
                geoItem = r;
                display(geoItem.area(), geoItem.perimeter());
        }
 }
```

Implementation of classes with interface: Example



Inheritance with Interface

Extending interface

- Interface can inherit from other interface.
- Interface can also multiply inherits.

```
interface Constants {
    double velOfLight = 3.0e+10;
    String unitVelOfLight = "m/s";
    ....  ....  ....  ....
}
interface Physics {
    void quantumLaw();
    ... ... ... ...

}
```

```
interface Chemistry extends
Constants
{
    ..... ..... ..... .....
    ..... ..... ..... .....
}
interface lawOfPysics extends
Constants, Physics
{
    ..... ..... ..... .....
    ..... ..... ..... .....
}
```


Thank You

OBJECT ORIENTED PROGRAMMING WITH JAVA

Interfaces in Java – II

Debasis Samanta
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Various Types of Interface

Some of Java's Most used interfaces

Iterator	• To run through a collection of objects without knowing how the objects are stored, for example, in array, list, bag, or set.
Cloneable	• To make a copy of an existing object via the `clone()` method on the class `Object`.
Serializable	• Pack a web of objects such that it can be send over a network or stored to disk. A naturally later be restored as a web of objects `Comparable`.
Comparable	• To make a total order of objects, for example, 3, 56, 67, 879, 3422, 34234

Iterator interface

The Iterator interface in the package **java.util** is a basic iterator that works on collections.

```
package java.util.*;
public interface Iterator {
     public abstract boolean hasNext();  // Check, if the list has more
     Object next();   // Return the next element
     void remove(); // optional throws exception
 }
     // use an iterator
      myShapes = getSomeCollectionOfShapes(); // Has set of objects
      Iterator iter = myShapes.iterator();
          while (iter.hasNext()) {
            Shape s = (Shape)iter.next(); // downcast
            s.draw();
      }
```

Cloneable interface

- A class `X` that implements the `Cloneable` interface tells that the objects of class `X` can be cloned.
- The interface is empty, that is, it has no method.
- Returns an identical copy of an object.
  - A shallow copy, by default.
  - A deep copy is often preferable.
- Prevention of cloning
  - Necessary, if unique attribute, for example, database lock or open file reference.
  - Not sufficient to omit to implement `Cloneable`.
    - Sub classes might implement it.
  - Clone method should throw an exception:
    - `CloneNotSupportedException`

Cloneable Interface: Example

```
public class Car implements Cloneable{
      private String make;
      private String model;
      private double price;
      public Car() {// default constructor
            this("", "", 0.0);
      }
 // give reasonable values to instance variables
      public Car(String make, String model, double price){
            this.make = make;
            this.model = model;
            this.price = price;
     }
     public Object clone(){// the Cloneable interface
            return new Car(this.make, this.model, this.price);
    }
}
```

Serializable interface

```
public class Car implements Serializable {
       // rest of class unaltered

}
      // write to and read from disk
import java.io.*;
public class SerializeDemo{
       Car myToyota, anotherToyota;
       myToyota = new Car("Toyota", "Carina", 42312);
       ObjectOutputStream out = getOutput();
       out.writeObject(myToyota);
       ObjectInputStream in = getInput();
       anotherToyota = (Car)in.readObject();
}
```

- ✓ A class X that implements the Serializable interface tells clients that X objects can be stored on a file or other persistent media.
- ✓ The interface is empty, that is, has no methods.

Comparable Interface

In the package *java.lang*.

Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

```
package java.lang.*;
public interface Comparable {
      int compareTo(Object o);
}
```

Comparable interface: Example

```
public class IPAddress implements Comparable{
     private int[] n; // here IP stored, e.g., 125.255.231.123
     /** The Comparable interface */
     public int compareTo(Object o){
          IPAddress other = (IPAddress) o; // downcast
          int result = 0;
          for(int i = 0; i < n.length; i++){
               if (this.getNum(i) < other.getNum(i)){
                    result = -1;
                    break;
               }
               if (this.getNum(i) > other.getNum(i)){
                    result = 1;
                    break;
               }
          }
          return result;
     }
}
```

Some Salient Points

Defining an interface

- Defining an interface is similar to creating a new class.
- An interface definition has two components: the interface declaration and the interface body.

```
interfaceDeclaration
   {
            interfaceBody
   }
```

The `interfaceDeclaration` declares various attributes about the interface such as its name and whether it extends another interface, etc.

➢ The `interfaceBody` contains the constant and method declarations within the interface.

Defining an interface

```
public interface StockWatcher
{
     final String sunTicker = "SUNW";
     final String oracleTicker = "ORCL";
     final String ciscoTicker = "CSCO";
     void valueChanged (String tickerSymbol, double newValue);
}
```

If you do not specify that your interface is public, your interface will be accessible only to classes that are defined in the same package as the interface.

Implementing an interfaces

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

```
class classname [extends superclass] [implements interface [,interface...]]
 {
      // class-body
 }
```

- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then that method will be used by the clients of either interface.
- The methods that implement an interface must be declared public.

Implementing interfaces: An example

Example: A class that implements, say Callback interface:

```
class Client implements Callback {
     // Implement Callback's interface

      public void callback(int p) {
      System.out.println("callback called with " + p);
    }
}
```

When you implement an interface method, it must be declared as `public`.

Implementing interfaces: An example

- It is both permissible and common for classes that implement interfaces to define additional members of their own.

Example: The following version of `Client` implements `callback( )` and adds the method `nonIfaceMeth( )`

```
class Client implements Callback {
 // Implement Callback's interface
    public void callback(int p) { System.out.println("callback called
    with"+ p);
     }
    void nonIfaceMeth() {
             System.out.println("Classes that implement interfaces " + "may
             also define other members, too.");
    }
}
```

Partial implementations

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

Example

```
abstract class Incomplete implements Callback {
     int a, b;
     void show() {
        System.out.println(a + " " + b);
     }
}
```

Here, the class `Incomplete` does not implement `callback( )` and must be declared as `abstract`. Any class that inherits *`Incomplete`* must implement `callback( )` or be declared `abstract` itself.

Nested interfaces

An interface can be declared a member of a class or another interface. Such an interface is called a nested interface.

A nested interface can be declared as public, private, or protected.

When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.

Nested interfaces: Example

```
// This class contains a nested interface.
class A {
  public interface NestedIF {
      boolean isNotNegative(int x);
} }
// B implements the nested interface.
class B implements A.NestedIF {
       public boolean isNotNegative(int x) {
          return x < 0 ? false: true;
} }
 class NestedIFDemo {
      public static void main(String args[]) {
       // use a nested interface reference
       A.NestedIF nif = new B();
       if(nif.isNotNegative(10))
                 System.out.println("10 is not negative");
                  if(nif.isNotNegative(-12))
                            System.out.println("this won't
                 be displayed");
 } }
```

A defines a member interface called *NestedIF* and that it is declared public.

B implements the nested interface by specifying implements *A.NestedIF*

Inside the *main*( ) method, an *A.NestedIF* reference called *nif* is created, and it is assigned a reference to a *B* object. Because B implements *A.NestedIF*.

Variables in interfaces

- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.

```
import java.util.Random;
    interface SharedConstants {
        int NO = 0;
        int YES = 1;
        int MAYBE = 2;
        int LATER = 3;
        int SOON = 4;
        int NEVER = 5;
}
```

This program makes use of one of Java's standard classes: Random, which provides pseudorandom numbers.

Variables in interfaces

```
class Question implements SharedConstants {
      Random rand = new Random();
       int ask() {
           int prob = (int) (100 * rand.nextDouble());
          if (prob < 30)
                    return NO; // 30%
          else if (prob < 60)
                    return YES; // 30%
          else if (prob < 75)
                    return LATER; // 15%
          else if (prob < 98)
                    return SOON; // 13%
          else
                    return NEVER; // 2%
       }
}
```

In this example, the method **nextDouble( )** is used. It returns random numbers in the range 0.0 to 1.0.

In this sample program, the classes, **Question** implements the SharedConstants interface where NO, YES, MAYBE, SOON, LATER, and NEVER are defined. Inside the class, the code refers to these constants as if each class had defined or inherited them directly.

Variables in interfaces

```
class AskMe implements SharedConstants {
    static void answer(int result) {
        switch(result) {
          case NO:
              System.out.println("No");
              break;
           case YES:
              System.out.println("Yes");
              break;
          case MAYBE:
              System.out.println("Maybe");
              break;
          case LATER:
              System.out.println("Later");
              break;
          case SOON:
              System.out.println("Soon");
              break;
          case NEVER:
              System.out.println("Never");
              break;
          }
      }
```

```
    public static void main(String args[]) {
       Question q = new Question();
       answer(q.ask());
       answer(q.ask());
       answer(q.ask());
       answer(q.ask());
     }
   }
```

Interfaces can be extended

- An interface can inherit another using the keyword **`extends`**. The syntax is the same as for inheriting classes.

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

When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

Interfaces can be extended

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

As an experiment, if you try removing the implementation for **meth1( )** in MyClass, it will cause a compile-time error.

Multiple Inheritance Issue

Multiple inheritance issues

- Java does not support the multiple inheritance of classes. There is a key difference between a class and an interface: a class can maintain state information (especially through the use of instance variables), but an interface cannot.

- For example, assume that two interfaces called **Alpha** and **Beta** are implemented by a class called **MyClass**. What happens if both Alpha and Beta provide a method called **reset( )** for which both declare a default implementation? Is the version by Alpha or the version by Beta used by MyClass? Or, consider a situation in which Beta extends Alpha. Which version of the default method is used? Or, what if MyClass provides its own implementation of the method?

- To handle these and other similar types of situations, Java defines a set of rules that resolves such conflicts.

Multiple inheritance issues

Rules.

- First, in all cases, a class implementation takes priority over an interface default implementation.
  - Thus, if `MyClass` provides an override of the `reset( )` method, `MyClass`' version is used.
  - This is the case even if `MyClass` implements, say both `Alpha` and `Beta`. In this case, both defaults are overridden by `MyClass`' implementation.
- Second, in cases in which a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will occurs. Continuing with the example, if `MyClass` implements both `Alpha` and `Beta`, but does not override `reset( )`, then an error will occur.

Multiple inheritance issues

Rules.

- In cases, one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence. Therefore, continuing the example, if `Beta` extends `Alpha`, then `Beta`'s version of `reset( )` will be used.

- It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of `super`. Its general form is shown here:

Multiple Inheritance Issues

- It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of `super`. Its general form is shown here:

```
InterfaceName.super.methodName( )
```

- For example, if `Beta` wants to refer to `Alpha`'s default for `reset( )`, it can use this statement:

```
Alpha.super.reset();
```

Questions to think…

- How a robust program can be developed in Java ?
- How Java manages different types of errors in programs so that it can avoid abnormal termination of programs?

Thank You