Concept of Package in Java

IIT KHARAGPUR

NPTEL ONLINE CERTIFICATION COURSES

DEBASIS SAMANTA

CSE

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.

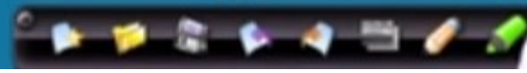
Built-in Packages in Java

IIT KHARAGPUR

NPTEL ONLINE CERTIFICATION COURSES

DEBASIS SAMANTA

CSE



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.

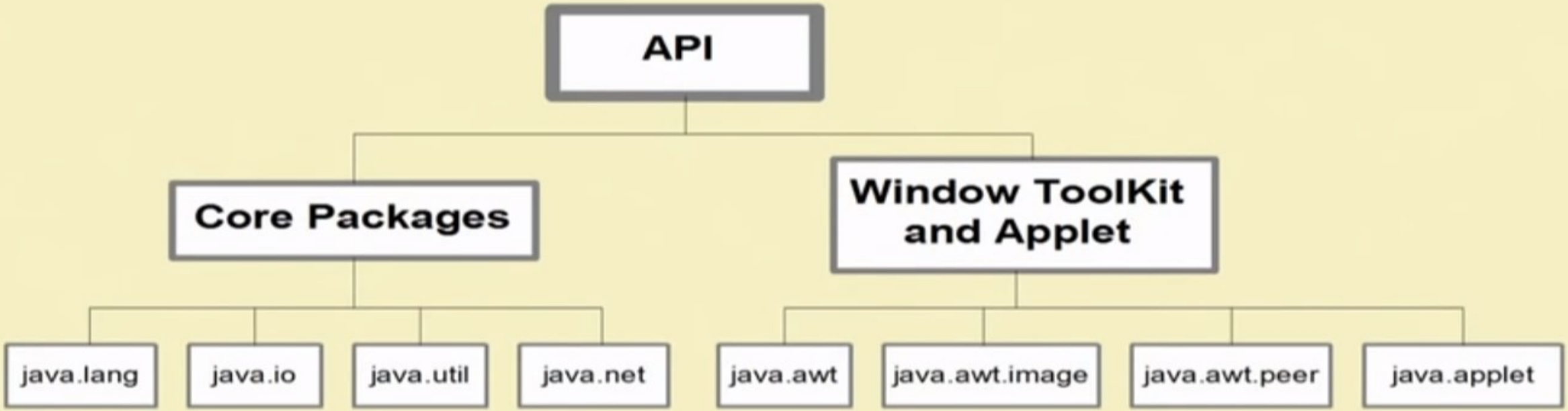
Example:

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

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.

There are two ways of using `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: ng.*

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

IIT KHARAGPUR

NPTEL ONLINE CERTIFICATION COURSES

DEBASIS SAMANTA

CSE

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 sub directory 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;
    //……..
}
```

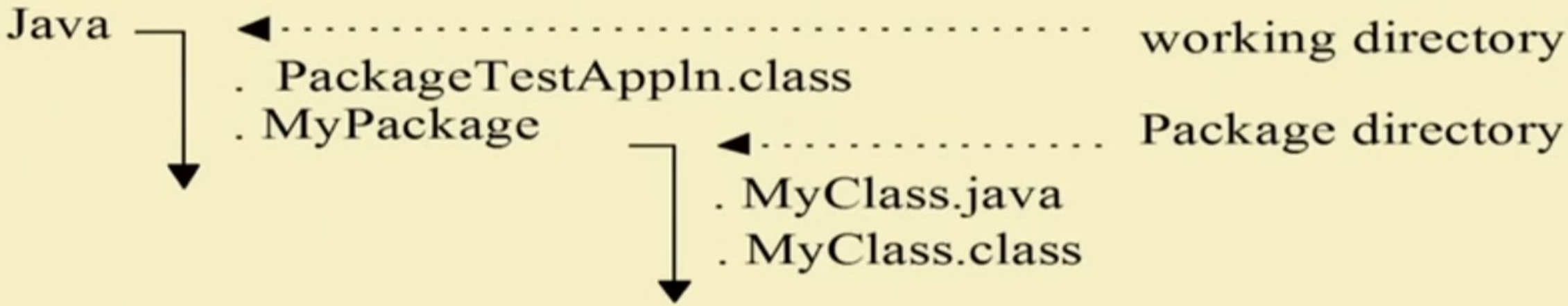
Package declaration

- 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 packa

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?

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

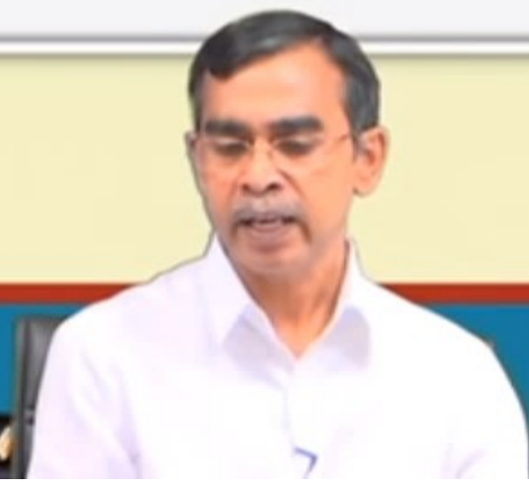
```
package MyPack;
class Balance {
    String name;
    double bal;
    Balance(String n, double b) {
        name = n;
        bal = b;
    }
    void show() {
        if(bal<0)
            System.out.print("Account is dead");
        else
            System.out.print(name+": $ "+bal);
    }
}
```

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);
    }
}
```

import MyPack;

```
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

```
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);
    }
}
```

import a.y.z.MyPack.Balance

```
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
```

IIT KHARAGPUR

NPTEL ONLINE CERTIFICATION COURSES

OBJECT ORIENTED PROGRAMMING WITH JAVA

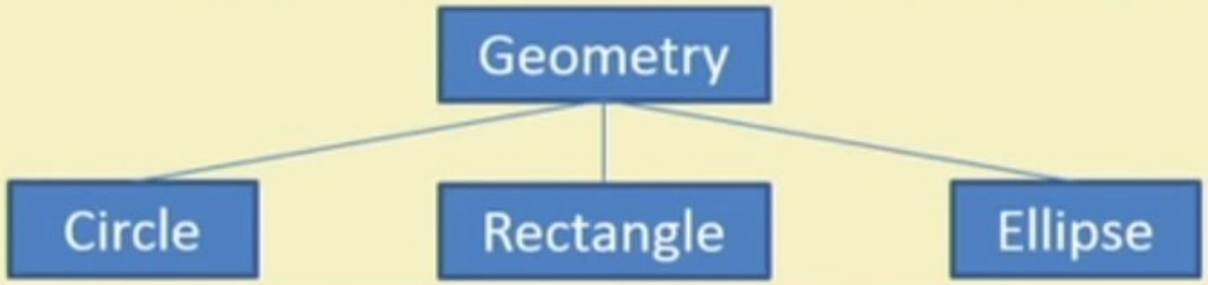
Interfaces in Java – I

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

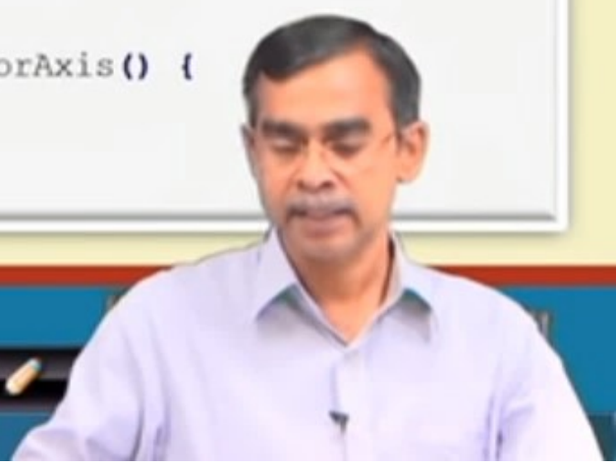
```
public class Ellipse extends Geometry {
    protected double a,b;
    public Ellipse() {
          a = 0.0;
          b = 0.0;
    }

    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);
   }
```

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);
   }
```

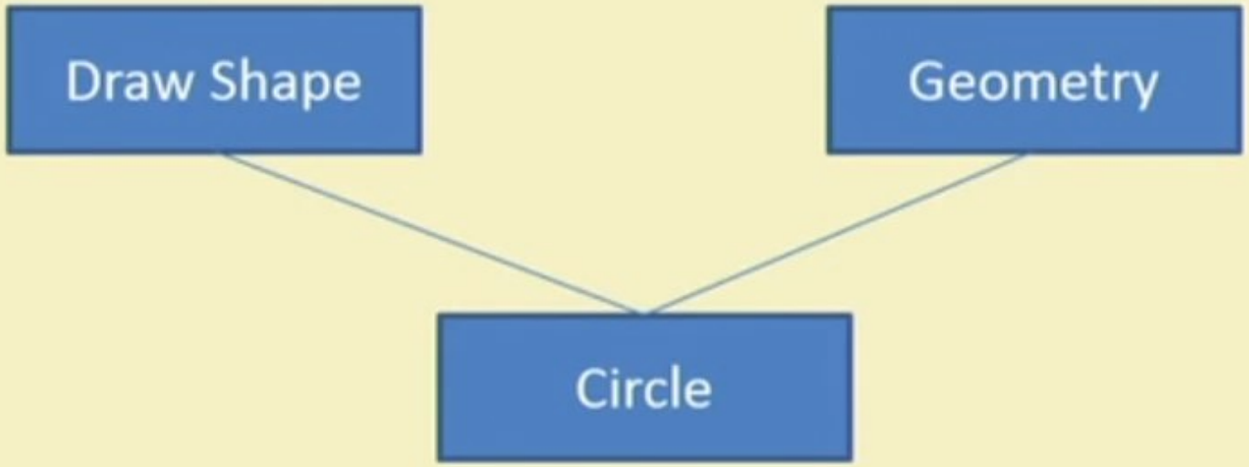
Note:

- 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 t ll, the code would cause compilation e

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`**, the Java compiler will not complain if you omit both keywords. Static methods cannot ared in the interfaces – these methods are never abstract and do not express behavior o jects.

Basic concept of inheritance

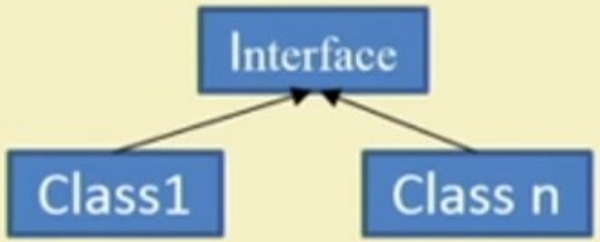
- Using the keyword `interface`, one can define an abstract class.
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are defined without any body.

Example:

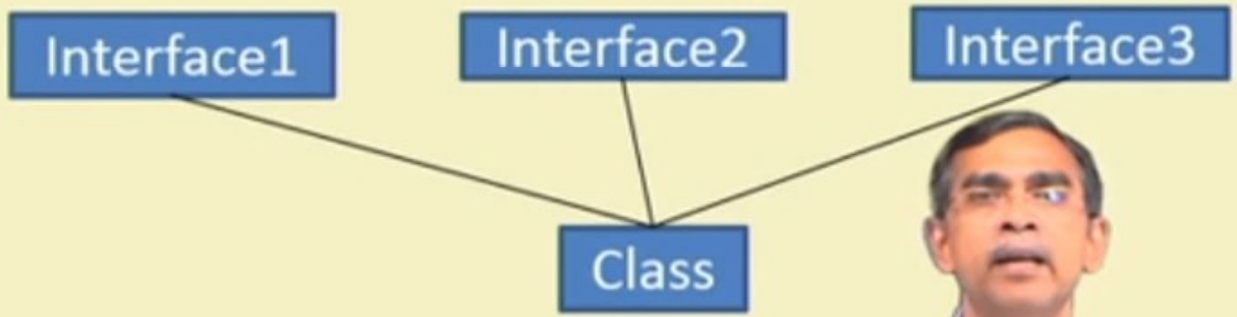
```
interface callMe {
        void call (int p);
}
```

Multiple inheritance in Java

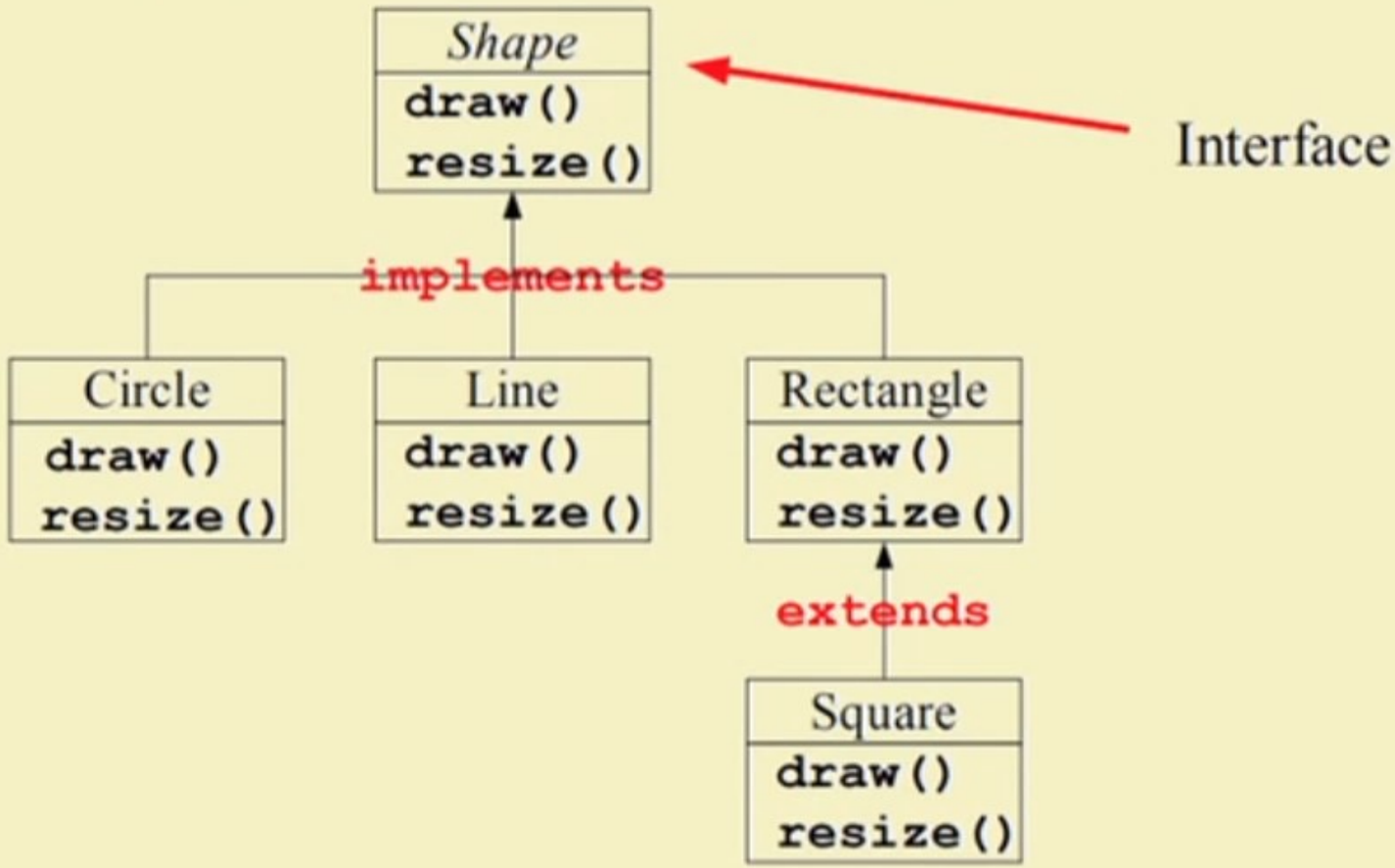
- Once an interface is defined, any number of classes can implement an interface.



- 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

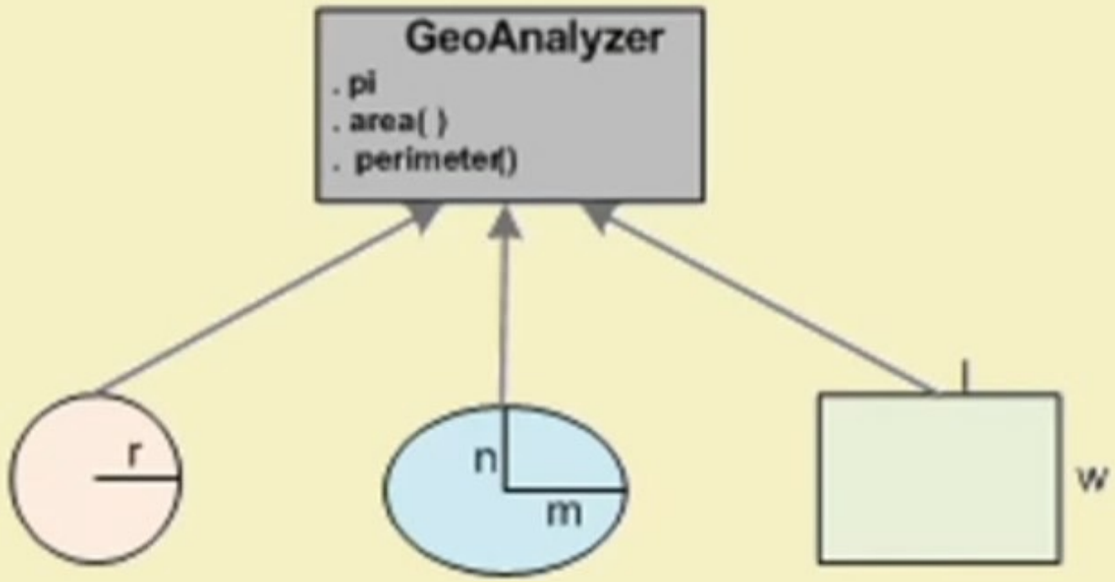
```
interface Curves extends Circle, Ellipse
{
    static final float pi = 3.142F;
    float area(int a, int b);
}
```

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));
        }

}
```

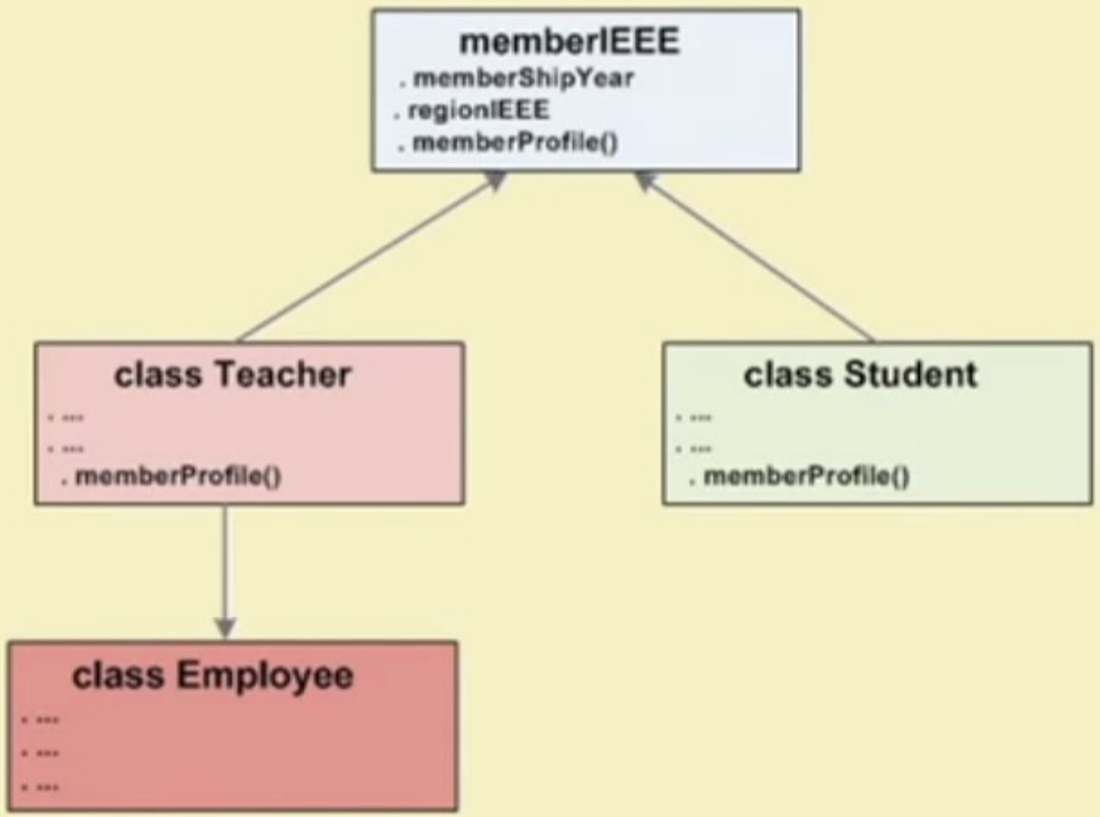
Implementation of classes with interface: Example

```
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

```
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
{
    ..... ..... ..... ..
    ..... ..... ..... ..
}
```