Inheritance

Raju Pal
Assistant Professor
Department of Computer Science and Engineering

Inheritance

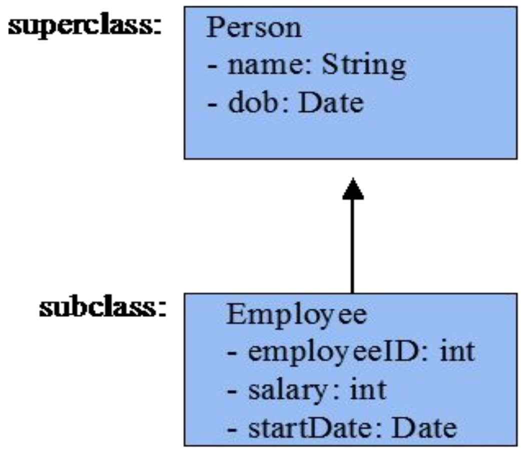
- Inheritance is a fundamental Object Oriented concept
    - On the surface, inheritance is a code re-use issue.
        - we can extend code that is already written in a manageable manner.
    - Inheritance is more
        - it supports polymorphism at the language level

Inheritance Cont…

- Take an existing object type (collection of fields and methods) and extend it.
  - create a special version of the code without re-writing any of the existing code (or even explicitly calling it!).
  - End result is a more specific object type, called the sub-class / derived class / child class.
  - The original code is called the superclass / parent class / base class.

Inheritance Cont…

- Inheritance uses subclasses:
  - A class can be defined as a "subclass" of another class.
  - The subclass inherits all data attributes of its superclass
  - The subclass inherits all methods of its superclass
  - The subclass inherits all associations of its superclass
- The subclass can:
  - Add new functionality
  - Use inherited functionality
  - Override inherited functionality



By: RAJU PAL

What really happens?

- When an object is created using new, the system must allocate enough memory to hold all its instance variables.
    - *This includes any inherited instance variables*
- In this example, we can say that an Employee "is a kind of" Person.
    - *An Employee object inherits all of the attributes, methods and associations of Person*

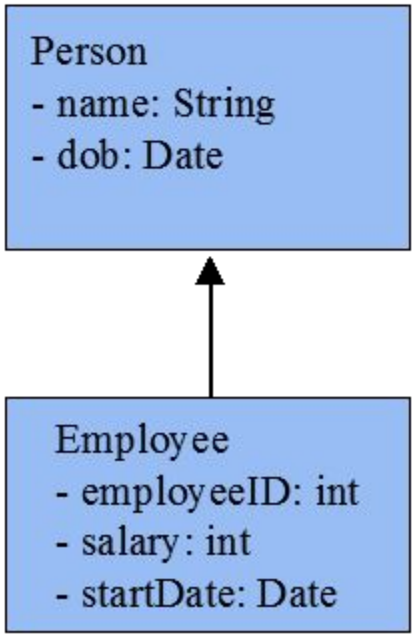
Inheritance in Java

- Inheritance is declared using the "***extends***" keyword
  - *If inheritance is not defined, the class extends a class called Object*

```
public class Person
{
  private String name;
  private Date dob;
  [...]
```

```
public class Employee extends Person
{
  private int employeeID;
  private int salary;
  private Date startDate;
  [...]
```

```
Employee anEmployee = new Employee();
```



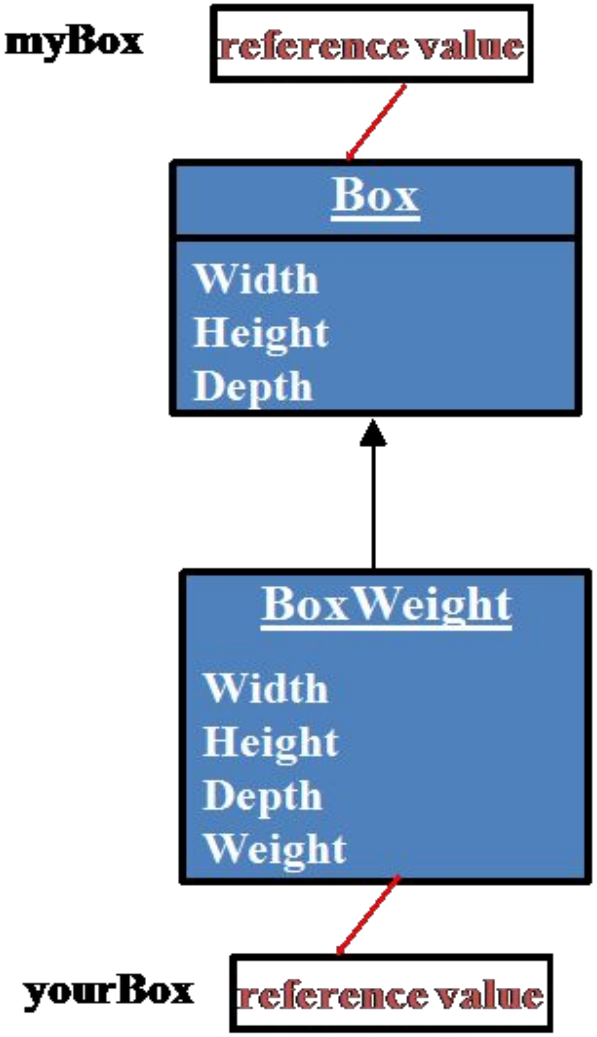
By: RAJU PAL

Inheritance in Java

```
class Box
{
  double width;
  double height;
  double depth;
}
```

```
class BoxWeight extends Box
{
  double weight;
}
```

```
Box myBox = new Box();
BoxWeight yourBox = BoxWeight();
```



By: RAJU PAL

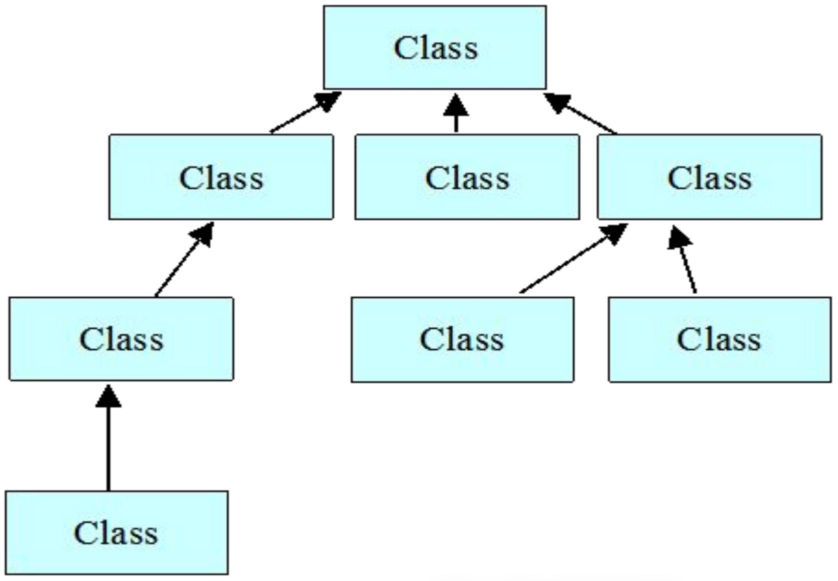
Inheritance and Visibility Rules

- **Private** variables and methods are not visible to subclasses or clients
- **Public** variables and methods are visible to all subclasses and clients
- **Protected** variables and methods can only be referenced by subclasses of the class and no other classes
- **Default** variables and methods are only visible to subclasses and clients defined in the same package as the class

Inheritance Hierarchy

- Each Java class has one (and only one) superclass.
    - *C++ allows for multiple inheritance*
- Inheritance creates a class hierarchy
    - *Classes higher in the hierarchy are more general and more abstract*
    - *Classes lower in the hierarchy are more specific and concrete*

- There is no limit to the number of subclasses a class can have
- There is no limit to the depth of the class tree.



By: RAJU PAL

Types of inheritance

- Single inheritance
  – only one super class
- Multiple inheritance
  – several super classes
- Hierarchical Inheritance
  – one super class, many sub classes
- Multilevel Inheritance
  – Derived from a derived class

$$\begin{tikzcd} A \arrow[d] \\ B \end{tikzcd}$$

a) Single Inheritance

$$\begin{tikzcd} & A \arrow[dl] \arrow[d] \arrow[dr] & \\ B & C & D \end{tikzcd}$$

b) Hierarchical Inheritance

$$\begin{tikzcd} A \arrow[d] \\ B \arrow[d] \\ C \end{tikzcd}$$

c) Multilevel Inheritance

$$\begin{tikzcd} A \arrow[dr] & & B \arrow[dl] \\ & C & \end{tikzcd}$$

d) Multiple Inheritance

By: RAJU PAL

Constructors and Initialization using *super*

- Classes use constructors to initialize instance variables
  - *When a subclass object is created, its constructor is called.*
  - *It also invoke the appropriate superclass constructors so that the instance variables defined in the superclass are properly initialized.*
- Superclass constructors can be called using the "super" keyword.
  - *It must be the first line of code in the constructor*
- If a call to super is not made, the system will automatically attempt to invoke the no-argument constructor (**default**) of the superclass.

Initialization using *super*

```
public class BankAccount
{
  private String ownersName;
  private int accountNumber;
  private float balance;

  public BankAccount(int anAccountNumber, String aName)
  {
        accountNumber = anAccountNumber;
        ownersName = aName;
  }
  [...]
}

public class OverdraftAccount extends BankAccount
{
  private float overdraftLimit;

  public OverdraftAccount(int anAccountNumber, String aName, float aLimit)
  {
        super(anAccountNumber, aName);
        overdraftLimit = aLimit;
  }
}
```

By: RAJU PAL

super as a member accessor

- ***super*** is also used to access a member of the superclass.
- Sometimes the member of superclass is hidden by the member of subclass.
- The general form is:

  super.*member*

  (member = method or variable)

```
class A
{
   int i;
}
```

```
class B extends A
{
   int i;
   B(int a, int b)
   {
           super.i = a;
           i = b;
   }
}
```

Method Overriding

- Subclasses inherit all methods from their superclass
  - *Sometimes, the implementation of the method in the superclass does not provide the functionality required by the subclass.*
  - *In these cases, the method must be overridden.*

- To override a method, provide an implementation in the subclass.
  - *The method in the subclass MUST have the exact same signature as the method it is overriding.*
  - *If they are not, then the two methods are simply overloaded.*

Method Overriding Example

```
class  Sbhape
{
  double dim1;
  double dim2;
  Shape(double a, double b)
  {
       dim1 = a;
       dim2 = b;
  }
  double area()
  {
       System.out.println("Area is Undefined");
       return 0;
  }
}
```

```
class FindArea
{
  public static void mmain(String args[])
  {
       Shape  s = new Shape(10,10);
       Rectangle r = new Rectangle(9,5);
       System.out.println("Area = "+s.area());
       System.out.println("Arae = "+r.area());
  }
}
```

```
class Rectangle extends Shape
{
  Rectangle(double a, double b)
  {
       super(a,b);
  }
  double area()
  {
       System.out.println(Area of rectangle:");
       return dim1*dim2;
  }
}
```

Output:

```
Area is Undefined
Area = 0
Area of Rectangle:
Area = 45
```

By: RAJU PAL

Overriding vs Overloading

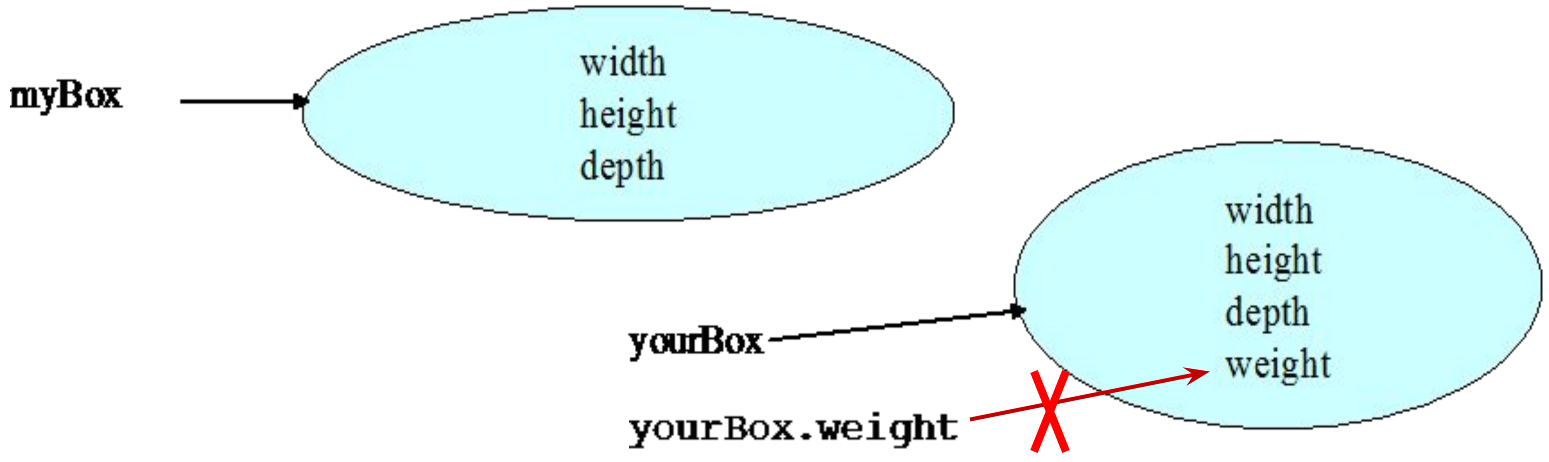
- A method is *overloaded* if it has multiple definitions that are distinguished from one another by having different numbers or types of arguments
- A method is *overridden* when a subclass gives a different definition of the method with the same number and types of arguments
- Selection of *overloaded* methods is done at compile time.
- Selection of *overridden* methods is done at run time.

Object References and Inheritance

- You can assign a subclass object reference to a variable of a superclass type.
- Superclass object variable will only permit access to those items that are members of its own class.

```
Box myBox = new Box();

Box your Box = new weightBox();
```



By: RAJU PAL

Dynamic Method dispatch
(Runtime Polymorphism)

- Mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- A superclass reference variable can refer to a subclass object.

- The version of the overridden method is selected based upon the type of the object being referred to superclass reference variable at the time the call occurs.

Dynamic Method dispatch Example

```
class Sbhape
{
  double dim1;
  double dim2;
  Shape(double a, double b)
  {
       dim1 = a;
       dim2 = b;
  }
  double area()
  {
       System.out.println("Area is Undefined");
       return 0;
  }
}
```

```
class Rectangle extends Shape
{
  Rectangle(double a, double b)
  {
       super(a,b);
  }
  double area()
  {
       System.out.println(Area of rectangle:");
       return dim1*dim2;
  }
}
```

```
class FindArea
{
  public static void main(String args[])
  {
       Shape s = new Shape(10,10);
       Shape s1 = new Rectangle(9,5);
       System.out.println("Area is "+s.area());
       System.out.println("Area is "+s1.area());
  }
}
```

Output:

```
Area is Undefined
Area = 0
Area of Rectangle:
Area = 45
```

By: RAJU PAL

Abstract Classes

- Abstract classes must contain one or more abstract methods
- Abstract classes can not be instantiated
- Classes are declared as abstract classes only if they will never be instantiated
- Cannot declare abstract constructors, or abstract static methods
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.
- Abstract classes can be used to create object references that can be used to point to a subclass object.
- Example:

```
abstract class Shape
{
   //-----------
}
```

Abstract Methods

- Abstract methods have no body at all and just have their headers declared
- The only way to use an abstract class is to create a subclass that implements each abstract method
- *Concrete classes* are classes that implement each abstract method in their superclasses
- Example:

```
abstract double area()
```

Abstract Class Example

```
abstract class Shape
{
   double dim1;
   double dim2;
   Shape(double a, double b)
   {
        dim1 = a;
        dim2 = b;
   }
   abstract double area();
}
```

```
class FindArea
{
  public static void main(String args[])
  {
        Shape s = new Shape(10,10);
        Rectangle r = new Rectangle(9,5);
        Shape s = new Rectangle(10,5);
        System.out.println("Area is "+r.area());
        System.out.println("Area is "+s.area());
  }
}
```

```
class Rectangle extends Shape
{
  Rectangle(double a, double b)
  {
       super(a,b);
  }
  double area()
  {
       System.out.println(Area of rectangle:");
       return dim1*dim2;
  }
}
```

Output:

```
Area of Rectangle:
Area = 45
Area of Rectangle:
Area = 50
```

By: RAJU PAL

Final Methods and Final Classes

- Methods can be qualified with the final modifier
  - *Final methods cannot be overridden.*
  - *This can be useful for security purposes.*
  - *Compiler free to inline calls- improve performance*

```
final double area()
{
   [...]
}
```

- Classes can be qualified with the final modifier
  - *The class cannot be extended*
  - *This can be used to improve performance. Because there will be no subclasses, there will be no polymorphic overhead at runtime.*

```
public final class A
{
   [...]
}
```

By: RAJU PAL

Method Overriding

- Subclasses inherit all methods from their superclass
  - *Sometimes, the implementation of the method in the superclass does not provide the functionality required by the subclass.*
  - *In these cases, the method must be overridden.*

- To override a method, provide an implementation in the subclass.
  - *The method in the subclass MUST have the exact same signature as the method it is overriding.*
  - *If they are not, then the two methods are simply overloaded.*

Interface - Introduction

- Using the keyword **interface**, you can fully abstract a class' interface from its implementation.
- That is, using **interface**, you can specify what a class must do, but not how it does it.
- Methods are declared without any body.
- Any number of classes can implement an interface.
- One class can implement any number of interfaces. (***Multiple Inheritance***)
- To implement an interface, a class must create the complete set of methods defined by the interface.
- Interfaces are designed to support dynamic method resolution at run-time.

Defining an Interface

- Basically a kind of class
- Contains methods (only abstract methods) and variables (only final fields)
- The general form of an interface:

interface *interfaceName*

{

  type final *var_name = value //variable declaration;*

  return-type *method_name(parameter_list) //methods declaration;*

}

```
public interface Shape
{
    static final double PI = 3.142;
    double area();
}
```

Defining an Interface

- Methods declared have no bodies and end with a semicolon after the parameter list
- Each class that includes an interface must implement all of the methods.
- All variables are declared as constants.
- They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class.

Implementing an Interface

- Interfaces are used as “superclasses” whose properties are inherited by classes.
- Necessary to create a class that inherits the given interface
- To implement an interface, include the **implements** clause in a class definition
- Creates the methods defined by the interface

```
class className implements interfaceName1 [,interfaceName2..]
{
                    //body of the class
}
```

- The methods that implement an interface must be declared **public**.

Implementing an Interface

```
public interface Shape
{
   static final double PI = 3.142;
   double area();
}
```

```
class Circle implements Shape
{
    double area()
    {
        return PI*r*r;
    }
}
```

- classes that implement interfaces can define additional members of their own.

By: RAJU PAL

Accessing using Interface Reference

- Can declare object reference variable of an interface.
- Any instance of any class that implements the declared interface can be referred to by such an variable.
- These variable can access only those members that are declared in the interface.
- Provides run-time polymorphism- correct version will be called based on actual instance.
- The method to be executed is looked up dynamically.

Interface Reference- Example

```
public interface Shape
{
  static final double PI = 3.142;
  double area();
}
```

```
class Circle implements Shape
{
   double radius;
   Circle(double r)
   {
     radius = r;
   }
   double area()
   {
      Syste.out.println("Area of circle: ");
      return PI*radius*radius;
   }
}
```

By: RAJU PAL

Interface Reference- Example

```
class Rectangle implements Shape
{
   double l, b;
   Rectangle(double a, double b)
   {
     l = a;
     this.b = b;
   }
   double area()
   {
      Syste.out.println("Area of Rectangle: ");
      return l*b;
   }
}
```

```
class Area
{
   public static void main(String args[])
   {
     Shape sc = new Circle(10);
     Shape sr = new Rectangle(5,15);
     System.out.println(sc.area());
     System.out.println(sr.area());
   }
}
```

By: RAJU PAL

Partial Implementation

- It is possible that a class implementing an interface does not fully implement the methods declared by that interface
- The class must be declared as abstract.

```
public interface Shape
{
  static final double PI = 3.142;
  double area();
  double volume();
}
```

```
abstract class Circle implements Shape
{
   double radius;
   Circle(double r)
   {
     radius = r;
   }
   double area()
   {
      return PI*radius*radius;
   }
}
```

Accessing Interface Variables

- You can use interfaces to import shared constants into multiple classes
- These variables are declared by the interface.
- They should initialized by some desired value.
- It is as if that class were importing the constant variables into the class name space as final variables.

Example:

variable PI is used in previous code

Extending the Interfaces

- Interfaces can be extended by some other interfaces.
- Interfaces are extended using **extends** keyword same as classes.

```
public interface A
{
    void meth1();
    void meth2();
}
```

```
public interface B extends A
{
    void meth3();
}
```

Packages-Introduction

- The main feature of OOP is its ability to support the reuse of code:
  - *Extending the classes (via inheritance)*
  - *Extending interfaces*
- The features in basic form limited to reusing the classes within a program.
- What if we need to use classes from other programs without physically copying them into the program under development ?
- In Java, this is achieved by using what is known as “packages”, a concept similar to “class libraries” in other languages.

Packages

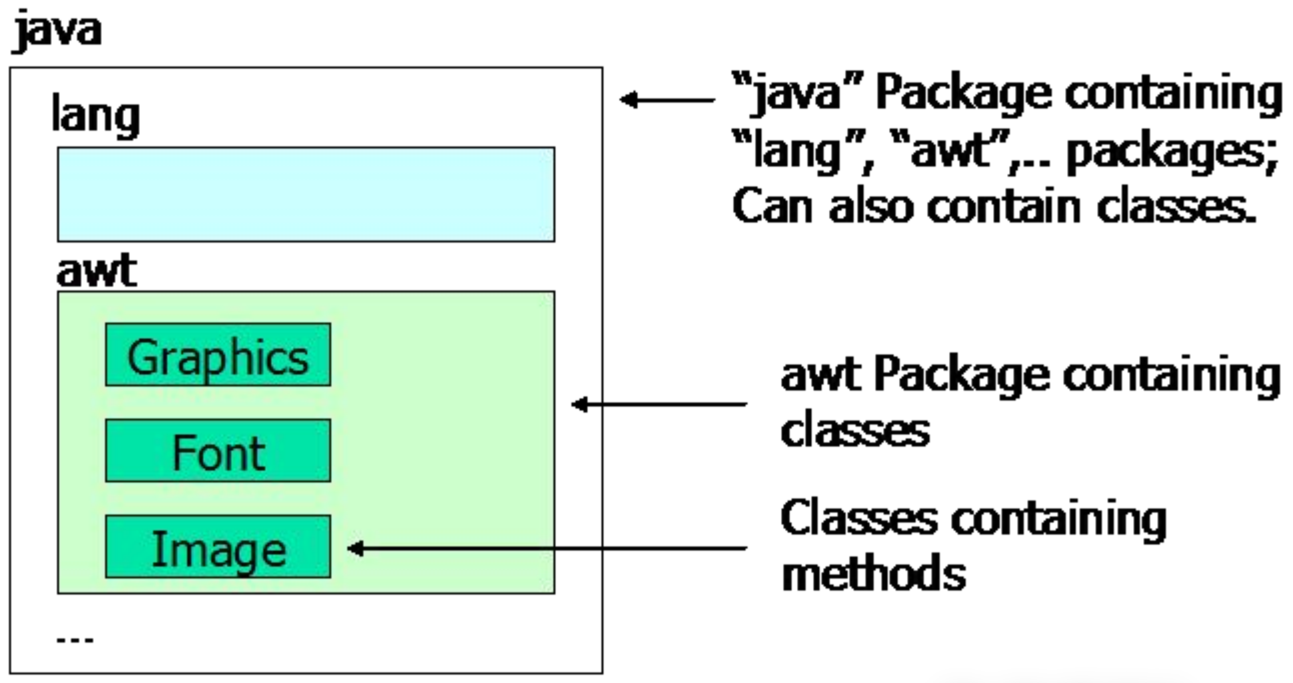
- Packages act as "containers" for classes.
- The mechanism provided by Java for partitioning the class namespace into more manageable chunks. It is both a naming and a visibility control mechanism.
- **<u>The benefits of organising classes into packages are:</u>**
    - The classes contained in the packages of other programs/applications can be reused.
    - In packages classes can be unique compared with classes in other packages. That two classes in two different packages can have the same name. If there is a naming clash, then classes can be accessed with their fully qualified name.
    - Classes in packages can be hidden if we don't want other packages to access them.

Java Foundation Packages

- Java provides a large number of classes grouped into different packages based on their functionality.
  - The six foundation Java packages are:
    - **java.lang**
      - Contains classes for primitive types, strings, math functions, threads, and exception
    - **java.util**
      - Contains classes such as vectors, hash tables, date etc.
    - **java.io**
      - Stream classes for I/O
    - **java.awt**
      - Classes for implementing GUI – windows, buttons, menus etc.
    - **java.net**
      - Classes for networking
    - **java.applet**
      - Classes for creating and implementing applets

Using System Packages

- The packages are organised in a hierarchical structure. For example, a package named “java” contains the package *“awt”*, which in turn contains various classes required for implementing GUI (graphical user interface).


java
lang
awt
Graphics
Font
Image
...
"java" Package containing "lang", "awt",.. packages; Can also contain classes.
awt Package containing classes
Classes containing methods

Defining a Packages

- Java supports a keyword called "**package**" for creating user-defined packages. The package statement must be the first statement in a Java source file (except comments and white spaces) followed by one or more classes.
- The package statement defines a name space in which classes are stored.

General form – ***package*** *packageName*;

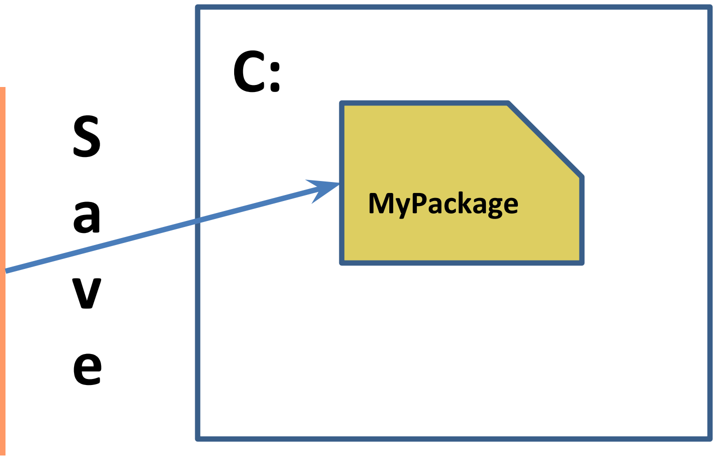
Example:

```
Package MyPackage;
Public class Example
{
    //body of the class
}
```

Storage of Packages

- Java uses file system directories to store packages.
- Directory (folder) name in which the file is saved must match with package name.
- For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory (folder) called **MyPackage.**
- You can not rename a package without renaming the directory in which the classes are stored.

```
Package MyPackage;
Public class Example
{
   //body of the class
}
```



By: RAJU PAL

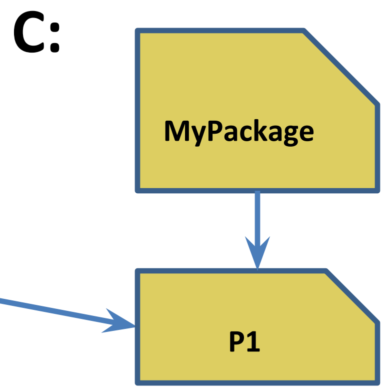
Creating Sub Packages

- Packages in Java are organised hierarchically, hence sub-packages can be created.
- You can create a hierarchy of packages by separating them with a period.
- To create sub package write the statement

  General Form: ***package*** *pkg1.pkg2 .pkg3;*

  Example:

```
Package MyPackage.P1;
Public class Example
{
   //body of the class
}
```



□*Store this file in a subdirectory named* ***MyPackage\P1***.

By: RAJU PAL

Compiling a Package class

- **Can compile in two following ways:**

1. Put source file into the same directory as the package name is declared and compile the program (resulting **.class** file is also in same directory).

```
C:\MyPackage>javac  Example.java
```

Need to create the package directory manually.

1. If you want to put the sourcefile and **.class** file at different location then use the following command (suppose source file is stored in a directory named javaPrograms. Creates package directory automatically.

```
C:\>javac  -d ./ ./javaPrograms/Example.java
```

By: RAJU PAL

Finding Packages and CLASSPATH

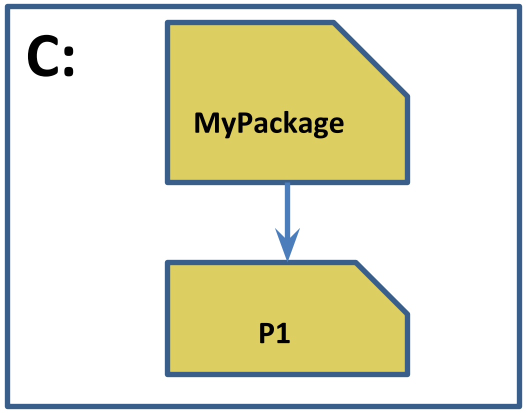
- How does the java run-time system know where to look for packages that you create?
- You can specify using two methods:
- **First**, by default the Java run-time uses the current working directory as its starting point. Thus , if package is in a subdirectory of the current directory, it will be found.
- **Second**, you can specify a directory path or paths by setting the CLASSPATH environmental variable. CLASSPATH must point to the package's parent directory.

```
C:\> set CLASSPATH = C:\;
```

If you want to point the current directory then use .(dot).
More than one location can be added into the CLASSPATH

```
C:\abc> set CLASSPATH = .; C:\;
```

Java will find our class file not only from C: directory but from the current directory as well i. e. ***abc***



By: RAJU PAL

Using Packages

- **There are three ways of using the resources in a package:**

1. *Inline Member Declarations*: declare the package member with its fully qualified package name.

1. *Importing a Single Package Member*: Easily achieved using an **import** statement in which the import keyword is followed by the fully qualified name of the member you wish to use.
2. *Importing an Entire Package*: It may be that you use a number of members from a package and end up with a large number of import statements. . Put asterisk (*) in the place of member Name

Using Packages

1. *Inline Member Declarations*: declare the package member with its fully qualified package name.

```
Public class Test
{
   MyPackage.Example example = new
MyPackage.Example();
}
```

1. *Importing a Single Package Member*: Easily achieved using an **import** statement in which the import keyword is followed by the fully qualified name of the member you wish to use.

```
import MyPackage.Example;
Public class Test
{
     Example example = new Example();
}
```

Using Packages

3. *Importing an Entire Package*: It may be that you use a number of members from a package and end up with a large number of import statements. . Put asterisk (*) in the place of member Name in import statement.

```
import MyPackage.*;
Public class Test
{
     Example1 example1 = new Example1();
     Example2 example2 = new Example2();
}
```

□*import statements occur immediately following the package statement (if it exists) and before any class definitions*

Using Package

- All of the standard Java classes included with Java are stored in a package called **java**.
- The basic language functions are stored in a package inside of the java package called **java.lang.** Since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs.
- If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes.
- You will get a compile time error and have to explicitly name the class specifying its package.

Protection and Packages

- All classes (or interfaces) accessible to all others in the same package.
- Class declared public in one package is accessible within another. Non-public class is not
- Members of a class are accessible from a difference class, as long as they are not private
- protected members of a class in a package are accessible to subclasses in a different class

Visibility - Revisited

- ***Public*** keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.
- ***Private*** fields or methods for a class only visible within that class. Private members are not visible within subclasses, and are not inherited.
- ***Protected*** members of a class are visible within the class, subclasses and also within all classes that are in the same package as that class.

Visibility Modifiers

Accessible to:	public	protected	Package (default)	private
Same Class	Yes	Yes	Yes	Yes
Class in package	Yes	Yes	Yes	No
Subclass in different package	Yes	Yes	No	No
Non-subclass different package	Yes	No	No	No

By: RAJU PAL

Package Example

```
package P1;
public abstract class Shape
{
    public double PI = 3.14;
    public double area();
}
```

```
package P1;
public class Circle extends
Shape
{
   public double radius;
   public Circle(double r)
   {

   }
   public double area()
   {
      return PI*radius*radius;
   }
}
```

```
package P2;
import P1.*;
public class Rectangle extends Shape
{
   public double l,b;
   public Rectangle(double l, double
b)
   {
    this.l = l;
    this.b = b;
   }
   public double area()
   {
       return l*b;
   }
}
```

By: RAJU PAL

Package Example

- ***Save Shape and Circle class in P1 directory.***
- ***Save Rectangle in P2 directory.***

```
import P1.*;
import P2.*;
public class Area
{
   public static void main(String args[])
   {
    Circle c = new Circle(10);
    Rectangle r = new Rectangle(15,10);
    System.out.println("Area : "+c.area());
    System.out.println("Area : "+r.area());
  }
}
```

Thank you !!

By: RAJU PAL