Mobile Application Development (14OE706/IT01)

By
K. Bhaskara Rao
Asst. Prof.
IT Dept.,
BEC

Mobile Application Development
Unit I - Syllabus

UNIT – 1 Syllabus

UNIT – I

- **Classes and Objects** : Concepts, methods, constructors, usage of static, access control, this key word, overloading, parameter passing mechanisms, nested classes and inner classes.
- **Inheritance:** Basic concepts, access specifiers, usage of super key word, method overriding, final methods and classes, abstract classes, Object class.
- **Packages & Interfaces:** Creating a Package, setting CLASSPATH, Access control protection, importing packages, defining an interface, implementing interface, variables in interface and extending interfaces.
- **Strings & Threads:** Exploring the String class, Creating Threads in Java.
- **I/O Streams & Collections:**, Streams, Byte streams, Character streams, File class, File streams, Collections: ArrayList, Hashtable, Dictionary, List.
- **Event Handling:** Events, Event sources, Event classes, Event Listeners, Delegation event model, handling events, Adapter Classes, Anonymous Inner Classes.

Mobile Application Development (UNIT -1)
Java Basics

By
K. Bhaskara Rao
Asst. Prof.
IT Dept.,
BEC

Classes and Objects

Classes and Objects :

- Concepts
- Methods
- Constructors
- Usage of static
- Access Control
- this key word
- Method Overloading
- Parameter passing mechanisms
- Nested classes and Inner classes.

OOP basics

- OOPs principles
  - ✓ Abstraction
  - ✓ Encapsulation
  - ✓ Inheritance
  - ✓ Polymorphism
- Abstraction: *Abstraction is a process where you show only "relevant" data and "hide" unnecessary details of an object from the user.*
  - *We achieve in Java using abstract classes and interfaces.*
- Encapsulation → *Encapsulation simply means binding object state(fields) and behaviour(methods) together. If you are creating class, you are doing encapsulation.*
  - *We can avoid misuse of data and methods.*
- Polymorphism → *Polymorphism (from Greek, meaning "many forms") is a feature that allows one interface to* be used for a general class of actions.
- The popular object-oriented languages are Java, C#, PHP, Python, C++, etc.

Abstraction

- Extract the relevant object properties while ignoring inessentials
  - Defines a view of the object
- Example - car
  - Car dealer views a car from selling features standpoint
    - Price, length of warranty, color, …
  - Mechanic views a car from systems maintenance standpoint
    - Size of the oil filter, type of spark plugs, …

Abstraction

Encapsulation

Encapsulation → *Encapsulation simply means binding object state(fields) and behaviour(methods) together. If you are creating class, you are doing encapsulation.*

- *It makes easy to understands the code*
- *We can avoid misuse of data and methods.*
- *Control the way day is accessed and modified*
- *Increases Reusability of code*
- *Increases flexibility in design*

Inheritance

- Inheritance is the process by which one object acquires the properties of another object.
- By use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent.

Advantages :

- ✓ Allows creating new class from existing class
- ✓ Software reusability

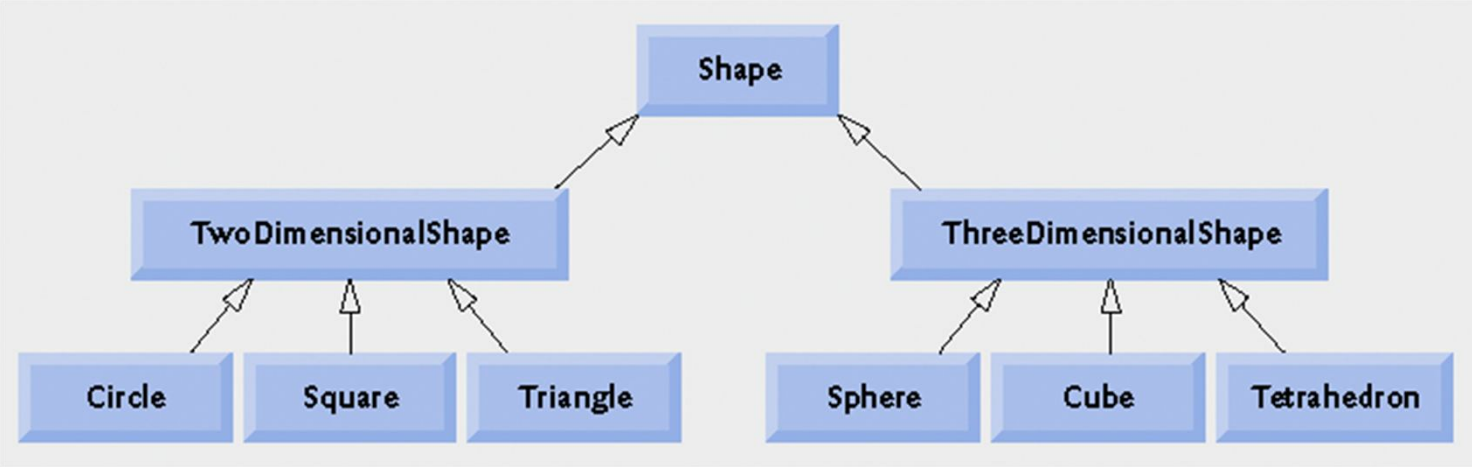
super class and subclass :

- A class that is inherited is called a *super class. The class that* does the inheriting is called a *subclass.*

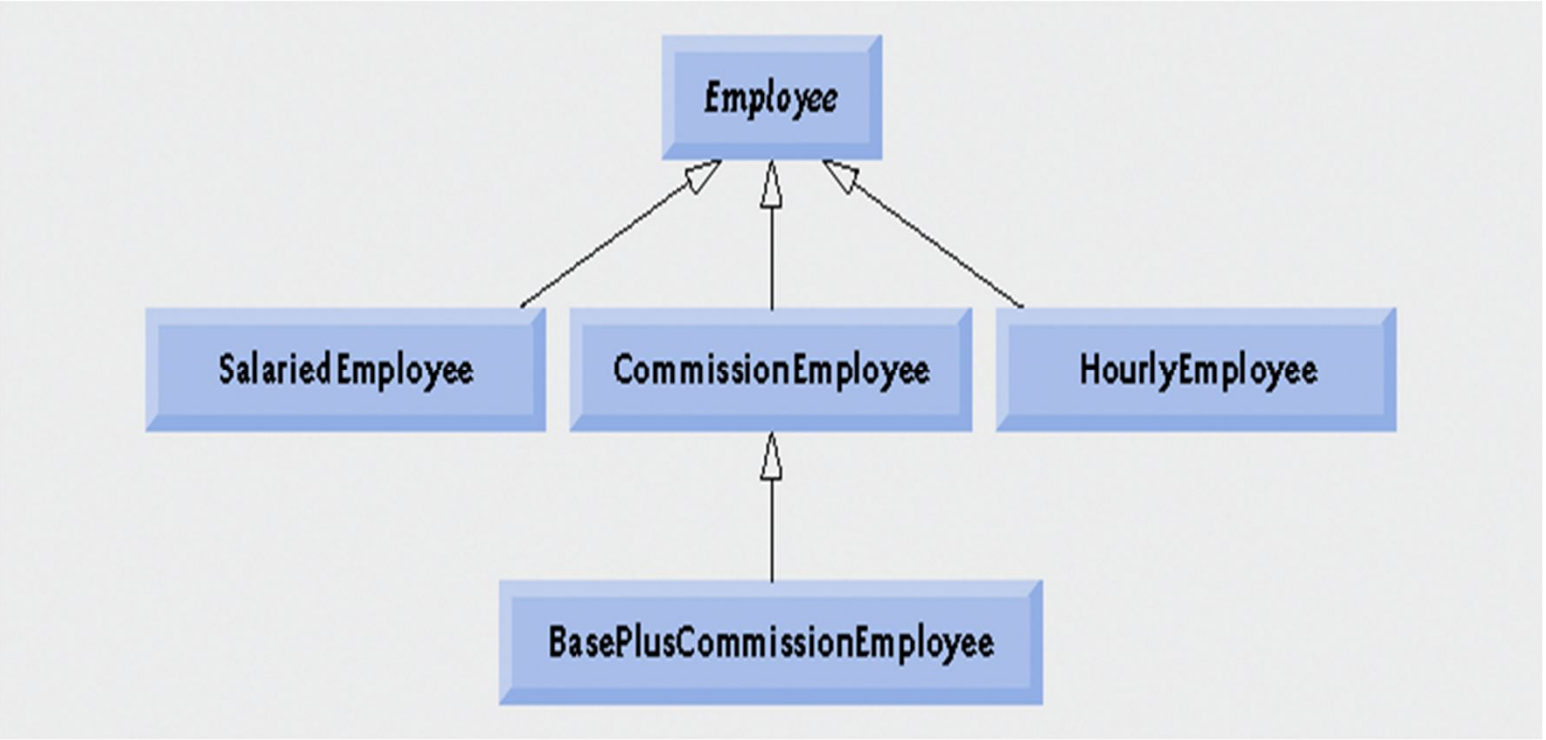
Inheritance

Inheritance examples :

Loan	**CarLoan, HomeImprovementLoan, MortgageLoan**
Employee	**Faculty, Staff**
BankAccount	**CheckingAccount, SavingsAccount**

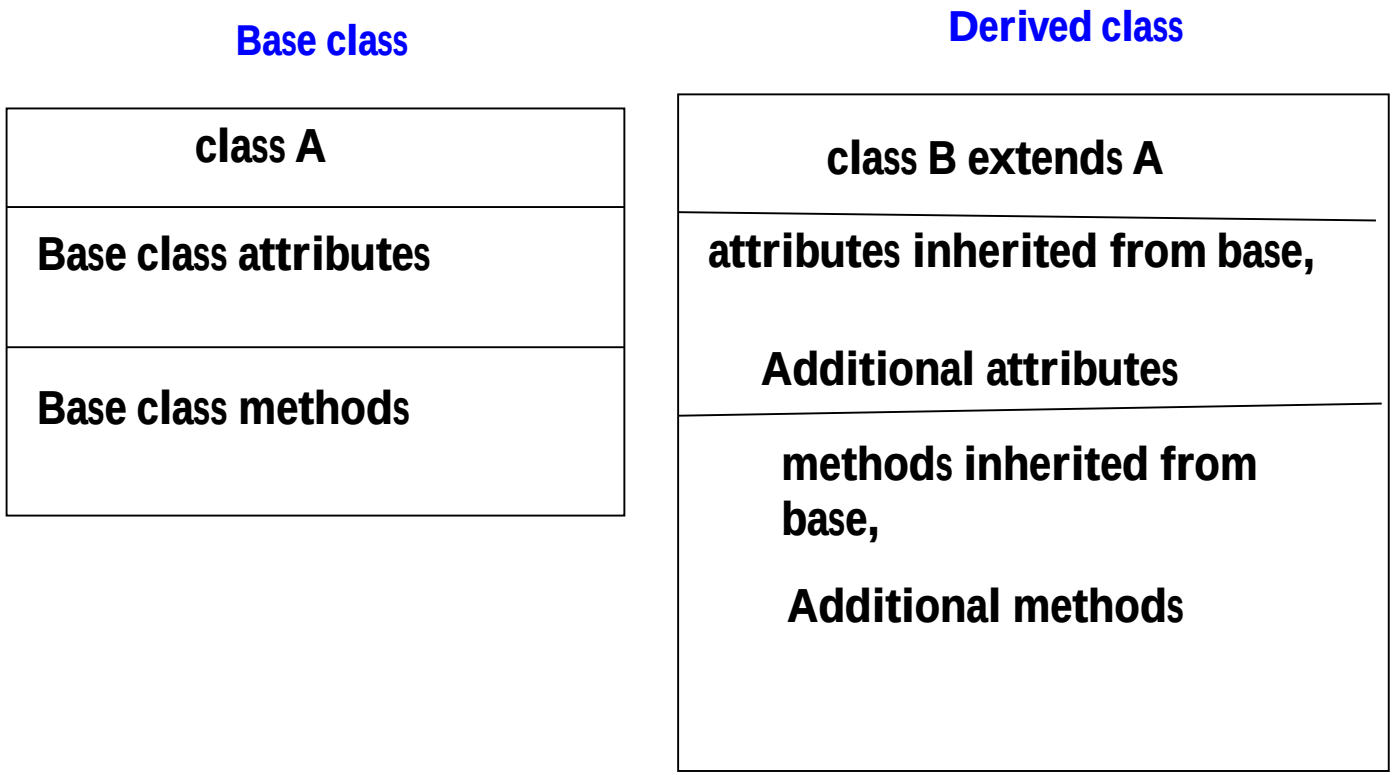


Inheritance

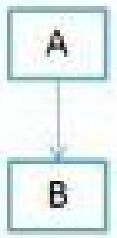


Inheritance

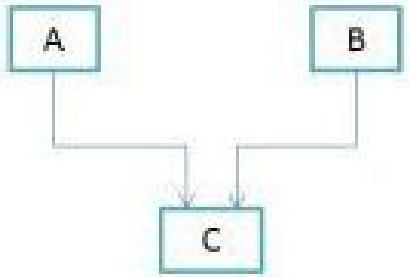
A derived class *extends* a base class. It inherits all of its methods (behaviors) and attributes (data) and it may have additional behaviors and attributes of its own.



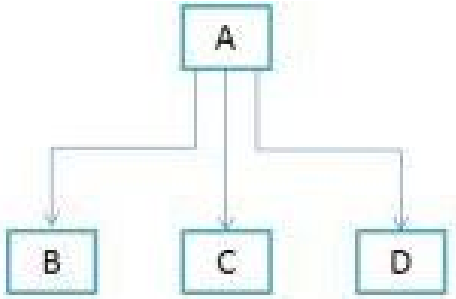
Inheritance types



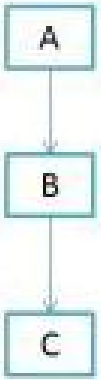
(a) Single Inheritance



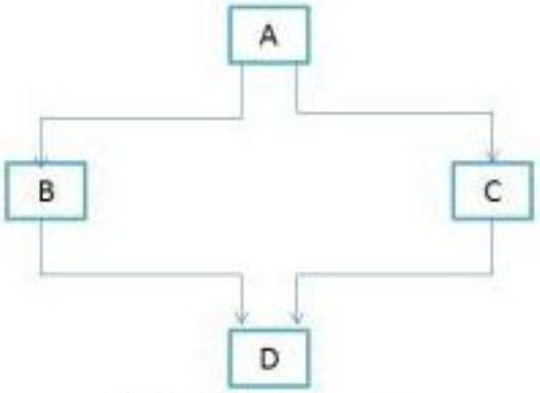
(b) Multiple Inheritance



(c) Hierarchical Inheritance



(d) Multilevel Inheritance



(e) Hybrid Inheritance

Polymorphism

- Polymorphism means *many* (poly) *shapes* (morph)
- In Java, polymorphism refers to the fact that you can have multiple methods with the same name in the same class
- There are two kinds of polymorphism:
  - Overloading
    - Two or more methods with different signatures
  - Overriding
    - Replacing an inherited method with another having the same signature

Reasons for using Polymorphism :

You may want to do “the same thing” with different kinds of data

Note :You can “overload” constructors as well as methods

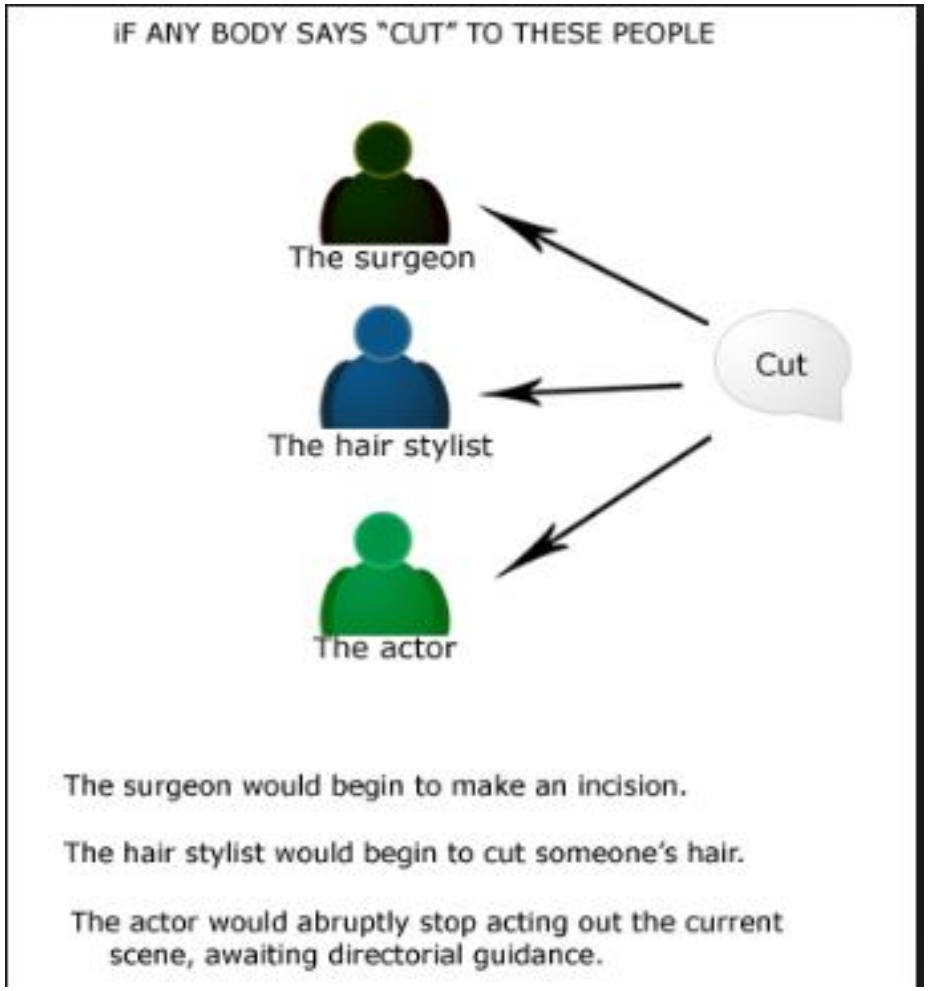
Polymorphism

- Polymorphism
  - Enables “programming in the general”
  - The same invocation can produce “many forms” of results
  - When a program invokes a method through a superclass variable, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable
  - The same method name and signature can cause different actions to occur, depending on the type of object on which the method is invoked
  - Facilitates adding new classes to a system with minimal modifications to the system’s code

Polymorphism

- ✓ The word polymorphism is used in various contexts and describes situations in which something occurs in several different forms. In computer science, it describes the concept that objects of different types can be accessed through the same interface. Each type can provide its own, independent implementation of this interface.
- ✓ The ability to take on different forms.
- ✓ Use of same thing for different purposes
- ✓ ***Polymorphism*** is the concept that different objects have different implementations of the same characteristic. For example, consider two objects, one representing a benz car and the other a Toyota car. They are both cars; that is, they both derive from the Car class, and they both have a drive method, but the implementations of the methods could be drastically different.
- You can create objects that respond to the same message in their own unique implementations. For example, you could send a print message to a printer object that would print the text on a printer, and you could send the same message to a screen object that would print the text to a window on your computer screen.

Polymorphism - examples



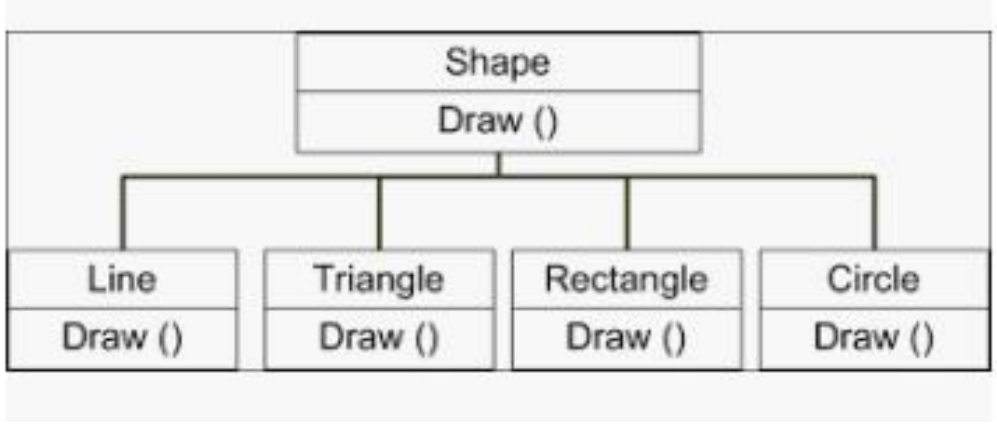
Polymorphism - examples

In Shopping malls behave like Customer

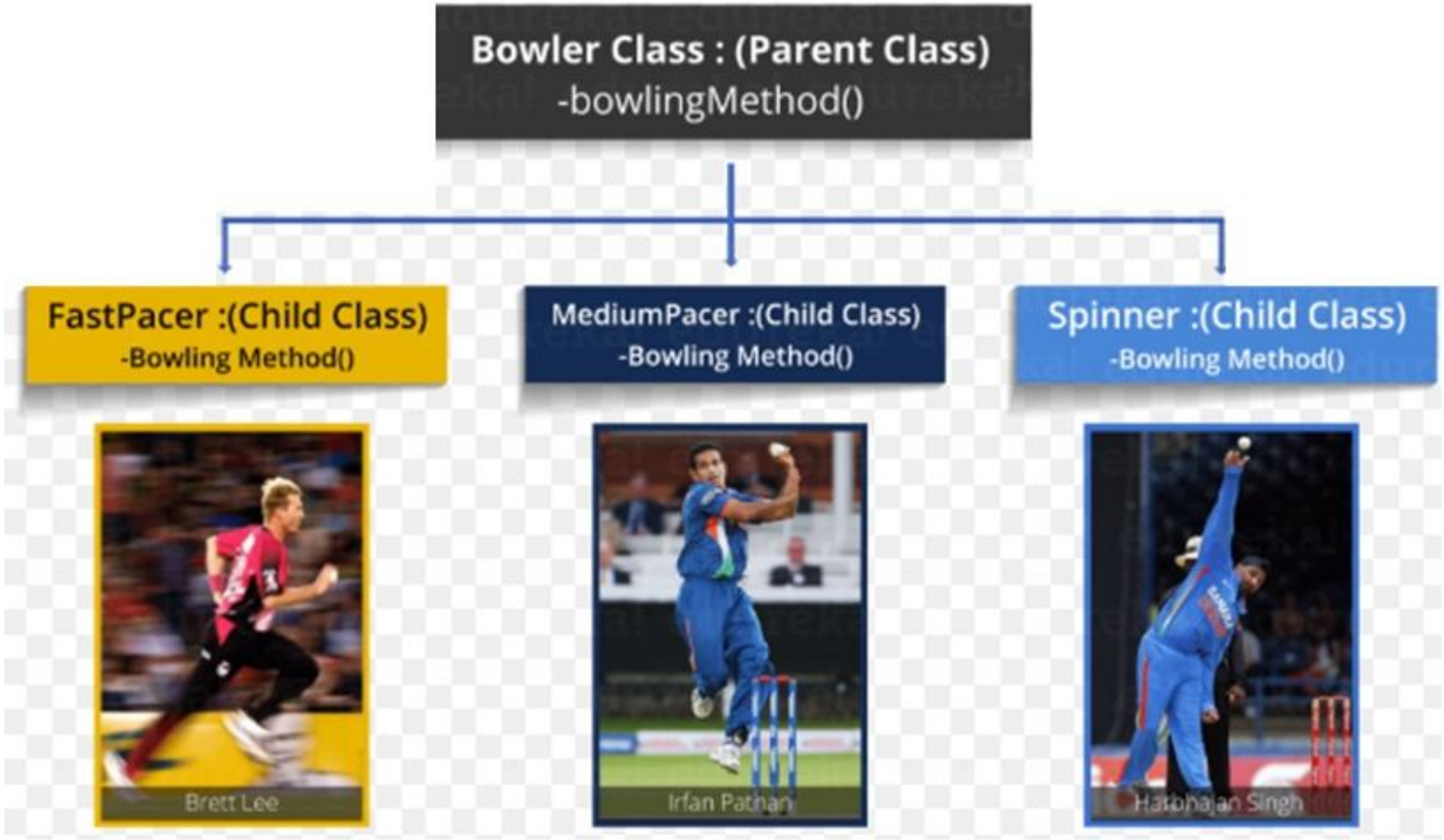
In Bus behave like Passenger

In School behave like Student

At Home behave like Son



Polymorphism - examples



Polymorphic Behaviour

- A superclass reference can be aimed at a subclass object
  - This is possible because a subclass object *is a* superclass object as well
  - When invoking a method from that reference, the type of the actual referenced object, not the type of the reference, determines which method is called

Method Overloading

```
class Test {
    public static void main(String args[]) {
        myPrint(5);
        myPrint(5.0);
    }

    static void myPrint(int i) {
        System.out.println("int i = " + i);
    }

    static void myPrint(double d) { // same name, different parameters
        System.out.println("double d = " + d);
    }
}
```

```
int i = 5
double d = 5.0
```

Overriding

```
class Animal {
    public static void main(String args[]) {
        Animal animal = new Animal();
        Dog dog = new Dog();
        animal. print();
        dog.print();
    }
    void print() {
        System.out.println("Superclass Animal");
    }
}

public class Dog extends Animal {
    void print() {
        System.out.println("Subclass Dog");
    }
}
```

Superclass Animal
Subclass Dog

- This is called overriding a method
- Method print in Dog overrides method print in Animal
- A subclass variable can *shadow* a superclass variable, but a subclass method can *override* a superclass method

Classes & Objects

- A class can be considered as a blueprint using which you can create as many objects as you like
- A Class declares exact form(state) and nature(behavior) of an object.
- A *class defines* the structure and behavior (data and code) that will be shared by a set of objects.
- A class defines a new data type.
- A class is a template for an object.
- Class is the basic unit of encapsulation
- Members of a class – instance variables, methods

Object: *instance of a class.*

Class declaration :

```
class classname {
    type instance-variable1;
    type instance-variable2;
    // ...
    type instance-variableN;
    type methodname1(parameter-list) {
    // body of method
    }
    // ...
    type methodnameN(parameter-list) {
    // body of method
    }
}
```

classes

Class Example :

```
class House {
   String address;
   String color;
   double Area;
   void openDoor() {
      //Write code here
   }
   void closeDoor() {
      //Write code here
   }
 ...
 ...
}
```

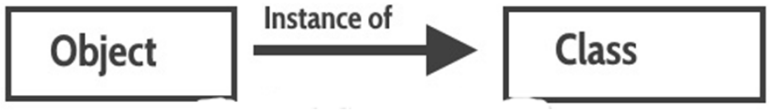
Here,
State → instance variables (address, color and rooms
Behaviour → methods (openDoor, closeDoor

Object

- An Object represents a real world entity that can be distinctly identified.

  Ex: student, circle, button, table

- Is an instance of Class.



- is a bundle of data(instance variables) and its behaviour(often known as methods).
- An object has a unique identity, state and behaviour
- State of an object → set of data fields (properties) with their current values
- Behaviour of an object → defines a set of methods

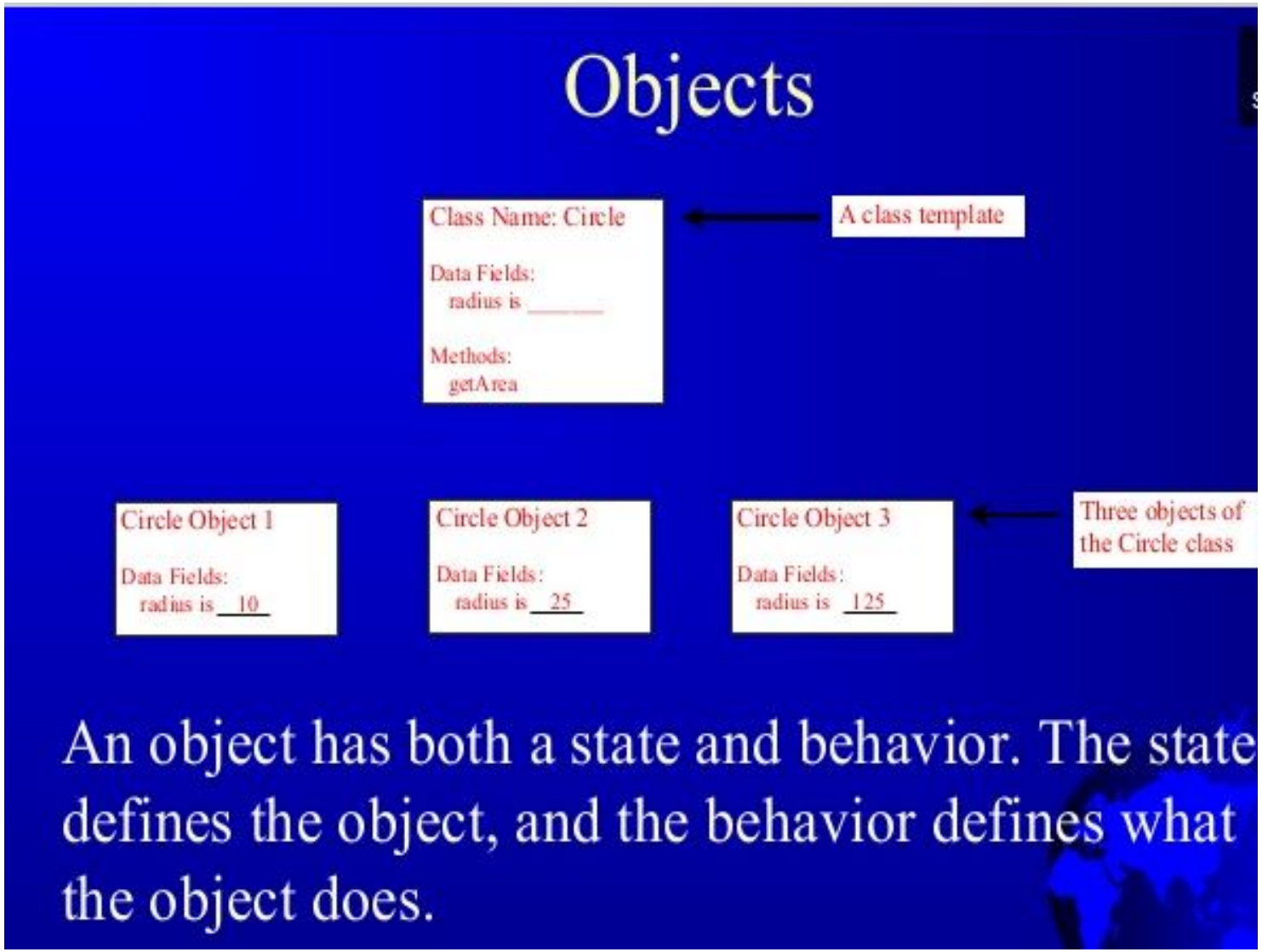
Ex:

Object: House
State: Address, Color, Area
Behavior: Open door, close door

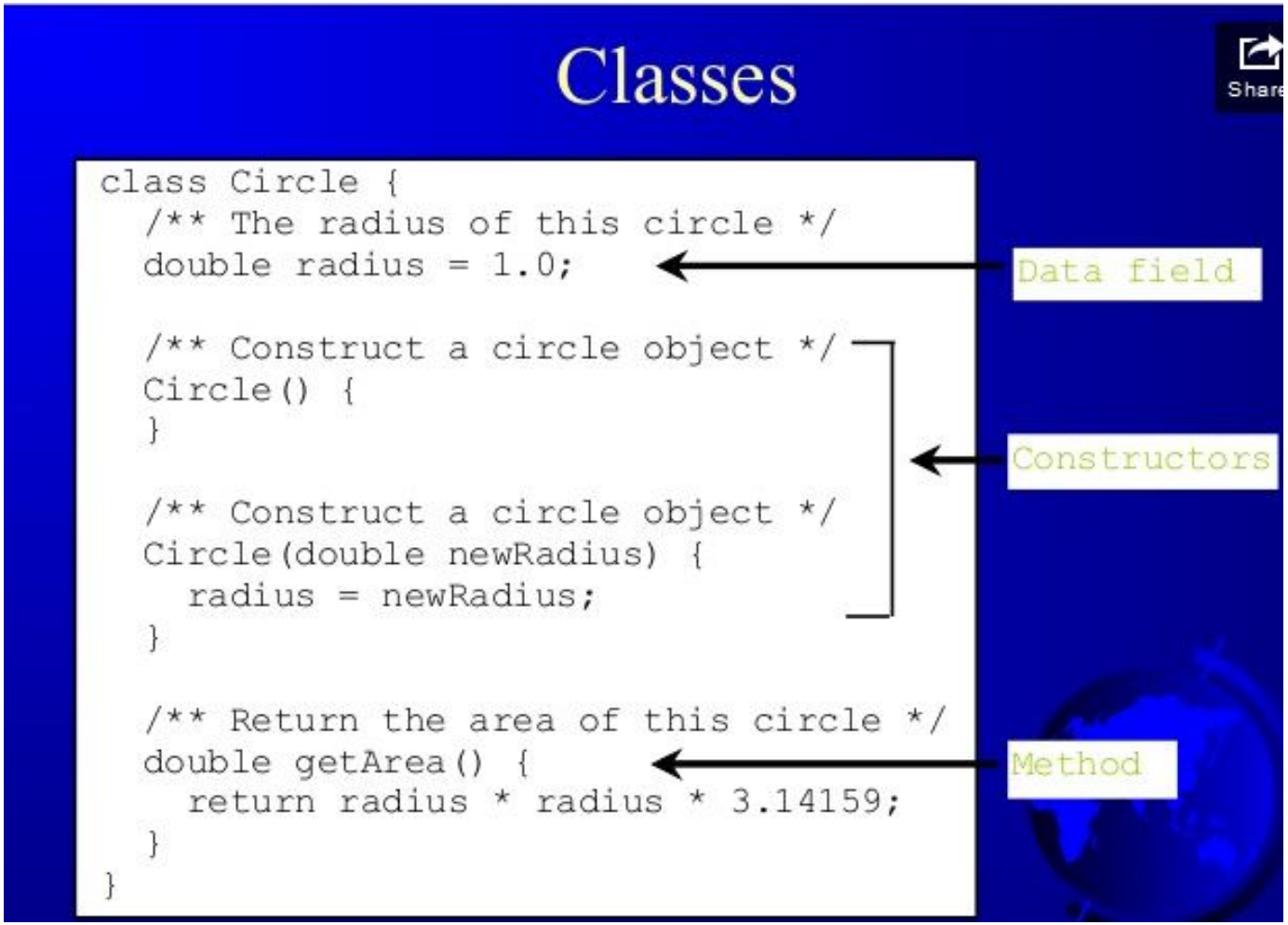
Objects

Characteristics:

- Abstraction
- Encapsulation
- Message Passing


Objects
Class Name: Circle
Data Fields:
radius is _____
Methods:
getArea
A class template
Circle Object 1
Data Fields:
radius is 10
Circle Object 2
Data Fields:
radius is 25
Circle Object 3
Data Fields:
radius is 125
Three objects of the Circle class
An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.

Class Example



Declaring objects

Syntax:

ClassName varName;

varName=new ClassName();

OR

ClassName varName=new ClassName();

Ex: Box mybox;

mybox=new Box();

OR

Box mybox=new Box();

Topics

Date: 19/08/2020

Topics:

- Methods
- Examples on Classes, methods

methods

Syntax:

```
type name(parameter-list)
 {
        //body of method
}
```

type → specifies the type of the data returned by the method

if the method doesn't return a value, its return type should be 'void'

Parameter-list → is a sequence of type and identifier pairs seperated by commas.

parameters receive the actual argument values passed to method.

Ex:

```
class Box{
        double width;
        double height;
        double depth;
      void volume() {
            System.out.println(width*height*depth);  }
```

Method returning value

Method returns a value to the caller using return stmt.

return value;

Example:

```
class Box {
    double width;
    double height;
    double depth;
    // compute and return volume
    double volume() {
    return width * height * depth;
    } }
```

Note: 1. An instance variable can be referred by method directly without referring object and . operator

2. The type of data returned by a method must be compatible with the return type specified by the method.
3. The variable receiving the value returned by a method must also be compatible with the return type specified for the method.

Adding a method that takes parameters

- A well designed java code should access instance vars through instance methods only (to maintain data abstraction)

Ex:

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

    // compute and return volume
    double volume() {
    return width * height * depth;
    }

    //  sets dimensions of box
    void setDim(double w, double h, double d) {
    width = w;
    sheight = h;
    depth = d;
    }
}
```

Topics

Date: 20 / 08 / 2020

Topics:

- Constructors
- this keyword
- Access control
- Parameter passing
- overloading – method, constructor
- usage of static keyword

Constructors

- A Constructor initializes the object immediately upon creation
- Is automatically called after object is created.
- Contains same name as that of the class and does not contain any return type including ‘void’.

Ex:

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box() {
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

- Java creates a default constructor if you don’t define the constructor.

Parameterized Constructors

- A Parameterized constructor allows you to initialize each object with different sets of initial values.

Ex:

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

this keyword

- 'this' keyword refers the current object.
- The **this keyword—which can be used only inside a method—**produces the reference to the object the method has been called for.
- In a method, we can refer the current object using 'this' keyword.
- When a local variable hides the instance variable(both are having same name), 'this' keyword is used to refer the instance variable.

Ex 1:

```
class Box {
   double width;
   double height;
   double depth;
   // This is the constructor for Box.
   Box(double width, double height, double depth) {
   System.out.println("Constructing Box");
   this.width = width;
   this.height = height;
   this.depth = depth;
   }
}
```

this keyword

Ex 2:

```
class Box {
   double width;
   double height;
   double depth;
   // This is the constructor for Box.
   Box(double width, double height, double depth) {
   System.out.println("Constructing Box");
   this.width = width;
   this.height = height;
   this.depth = depth;
   }
}
```

Parameter passing mechanisms

- Call by value

  This approach copies the *value of an argument into the formal* parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.

- Call by reference

  In this, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

Java uses call by value → for value types (int, float, double, char etc..)

call by reference → for objects (reference types)

caller ,actual args ---> calle , formal parameters

Access Control

- ✓ Encapsulation provides controlled access to the data and code.
- ✓ By encapsulation, you can control what parts of a program can access the members of a class.

Access specifiers:

- public
- private
- protected
- default access level

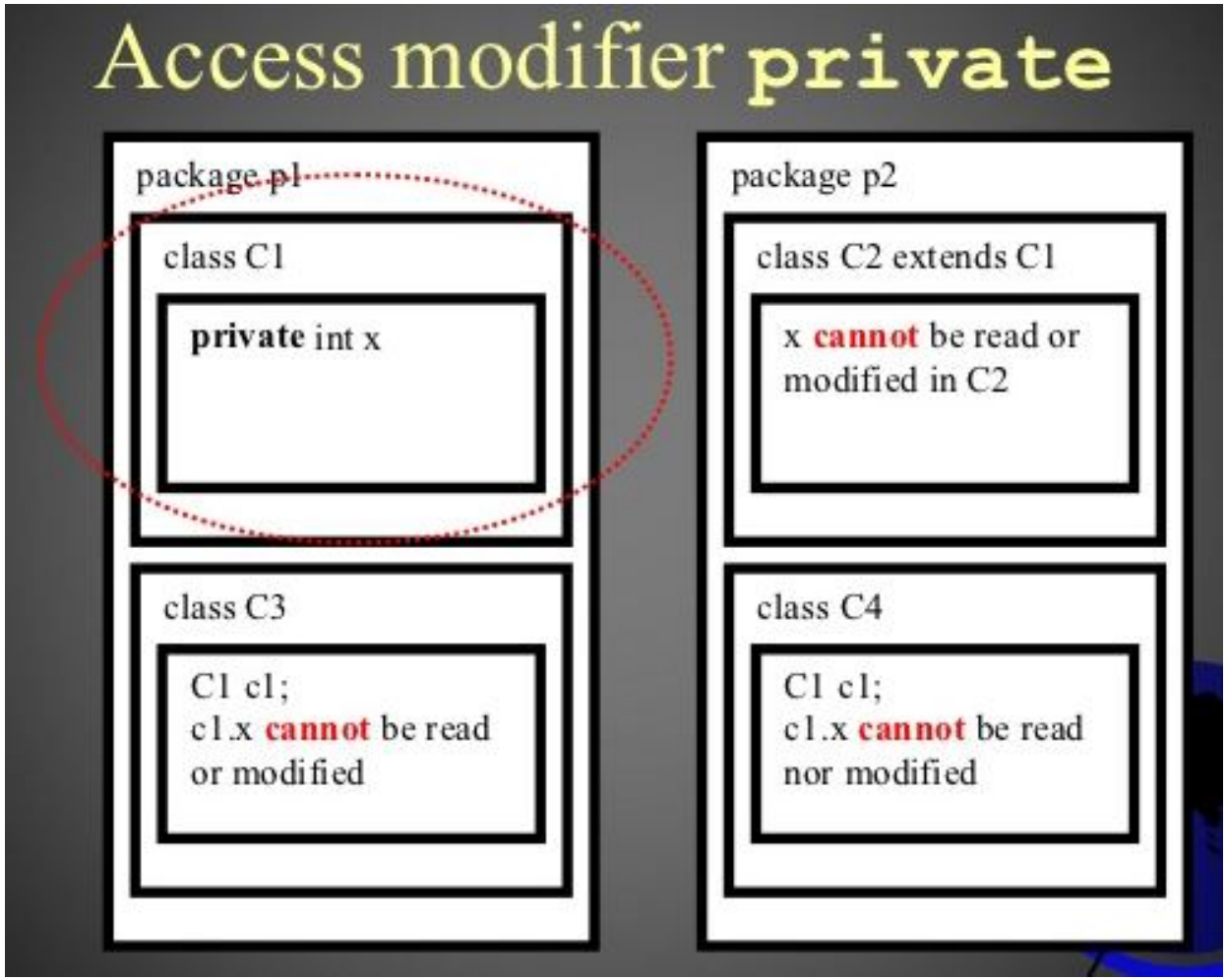
If class member is 'private' → member is accessible only by other members of the class.

If class member is 'public' → member is accessible by any other code

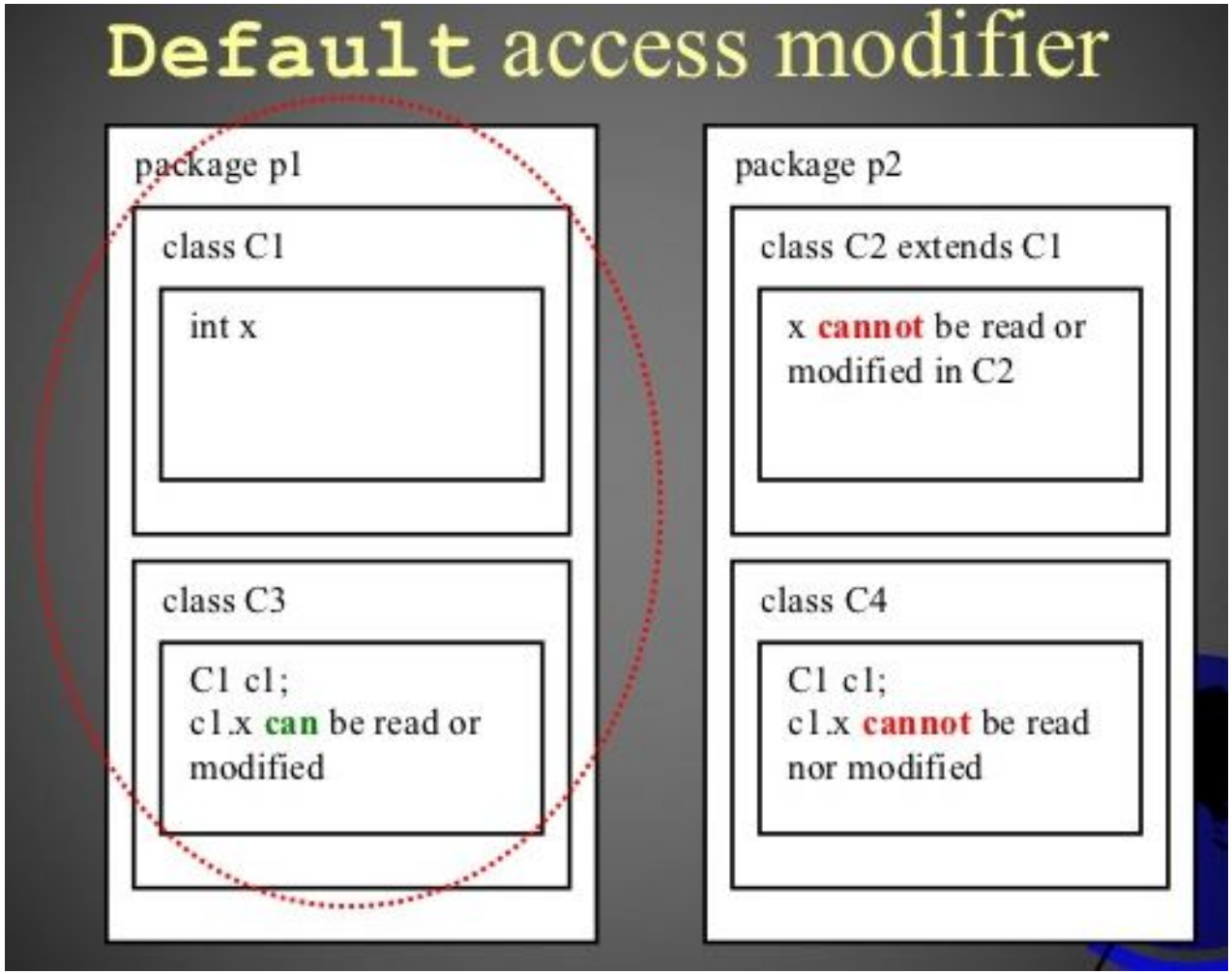
default level (no keyword) → up to the package

protected → accessible to the immediate derived class.(1 level down)

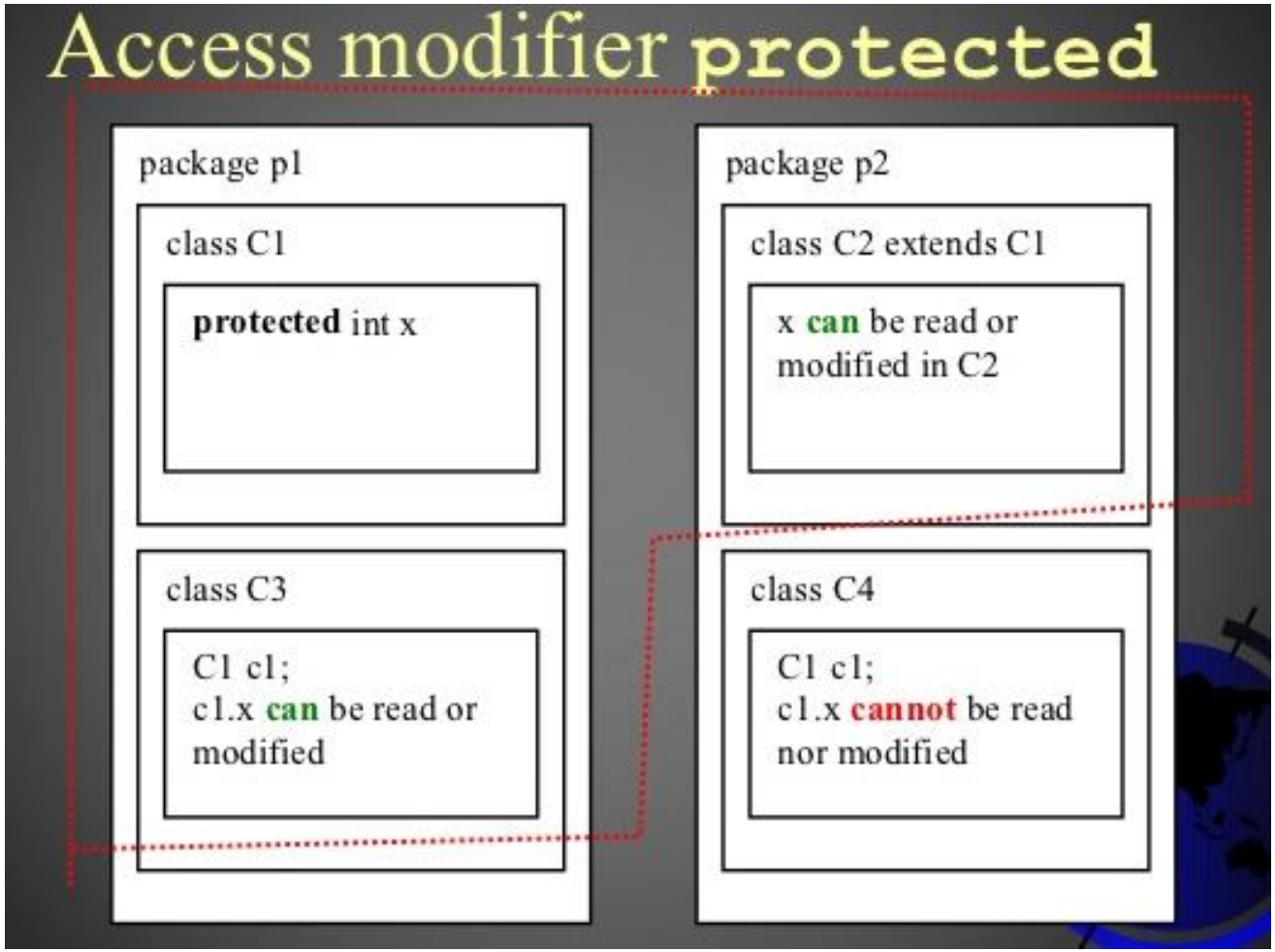
private access specifier



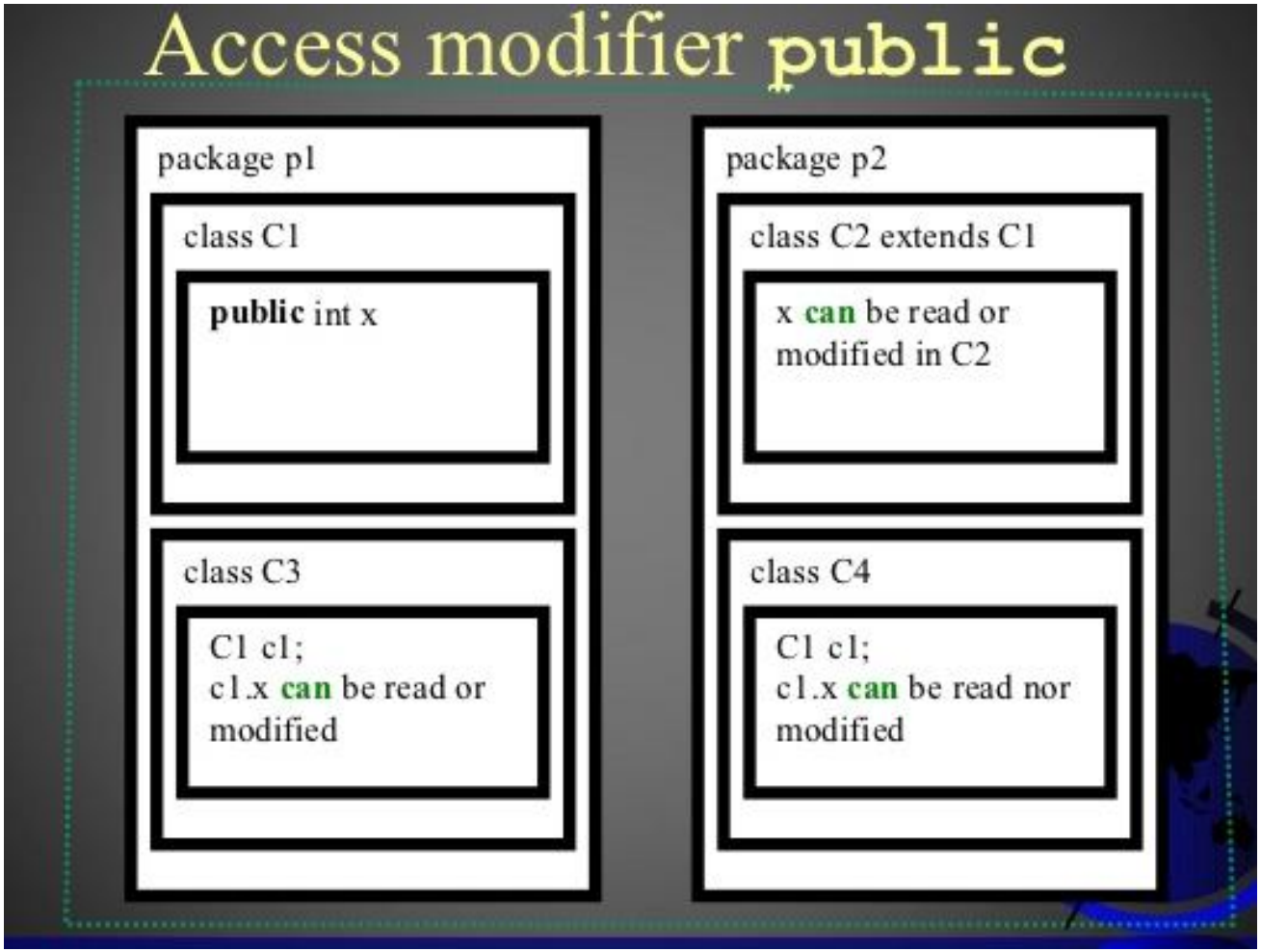
default access specifier



protected access specifier



public access specifier



Topics

Date: 24/08/2020

Topics:

- Access control -- example
- overloading – method, constructor
- usage of static keyword
- Inheritance – basic concepts, super keyword, final classes, final methods, access specifiers, abstract classes, method overriding, Object class.

Method Overloading

- In Java, defining more than one method with the same name and with different parameter declarations is called method overloading.
- Method overloading is one way of implementing polymorphism.
- When a overloaded method is called, java uses type and / or number of arguments as its guide to determine which version of the overloaded method to actually call.
- When there is no exact match, java uses automatic type conversion rules to determine the method to be called.

Method Overloading

```
class Test {
    public static void main(String args[]) {
        myPrint(5);
        myPrint(5.0);
    }

    static void myPrint(int i) {
        System.out.println("int i = " + i);
    }

    static void myPrint(double d) { // same name, different parameters
        System.out.println("double d = " + d);
    }
}
```

```
int i = 5
double d = 5.0
```

Constructor Overloading

Ex:

```
class Box {
    double width;
    double height;
    double depth;
    Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
    }
    Box() {
    width = -1; // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1; // box
    }
    Box(double len) {
    width = height = depth = len;
    }
    double volume() {
    return width * height * depth;
    }
}
```

static keyword

- If you want to use class member which is to be used without reference to a specific instance, member can be declared as static.
- main method is declared as static because it is called by runtime system before any objects exist.
- All objects of a class share the common static variables.
- A static member is accessed by referring it using class name.

  ex : classname.membername;

  classname.methodname();
- static variables
- static methods
- static blocks

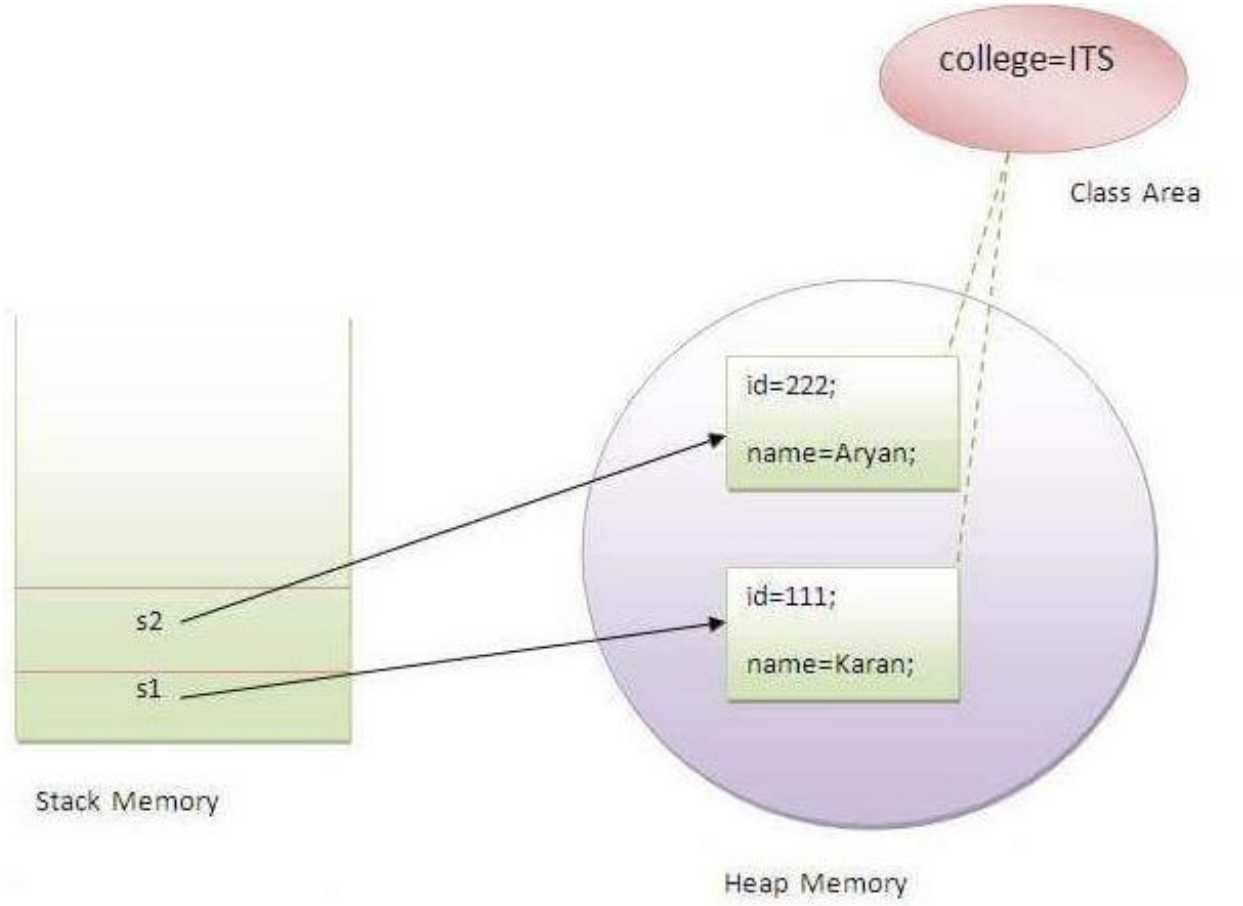
static variable

- If you declare any variable as static, it is known as a static variable.
- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- **Adv** → It makes your program memory efficient (i.e., it saves memory).
- It is a variable which belongs to the class and not to object.
- Static variables are **initialized only once** , when class is loaded. These variables will be initialized first, before the initialization of any instance variables.
- A **single copy** to be shared by all instances of the class
- A static variable can be **accessed directly** by the **class name** and doesn't need any object
- Default values: primitive integers(long, short etc): 0
  primitive floating points(float, double): 0.0
  boolean: false
  object references: null
- Syntax : <***class-name***>.<***variable-name***>

  Example :

```
class Temp{
        static int a=10;
}
….
System.out.println(Temp.a);
```

static variable



static method

- If you apply static keyword with any method, it is known as static method.
- It is a method which belongs to the class and not to object.
- A static method **can access only static data**. It can not access non-static data (instance variables)
- A static method **can call only** other **static methods** and can not call a non-static method from it.
- A static method can be **accessed directly** by the **class name** and doesn't need any object
- Syntax : ***<class-name>.<method-name>();***
- A static method cannot refer to "this" or "super" keywords in anyway
- To access static methods, we don't need any objects

static block : The static block, is a block of statement inside a Java class that will be executed when a class is first loaded in to the JVM.

- A class can have more than one static block. JVM combines them into one in runtime.

Ex:

```
class Test{
    static int a;
Int b;
    static { a=10;}
}
```

static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.
- Unlike C++, Java supports a special block, called static block (also called static clause) which can be used for static initializations of a class.
- The code inside static block is executed only once: the first time you make an object of that class or the first time you access a static member of that class (even if you never make an object of that class).

Inheritance

Inheritance

Inheritance Topics:

- Basic concepts
- access specifiers
- usage of super key word
- method overriding
- final methods and classes
- abstract classes
- Object class.

Inheritance

- Inheritance is the process by which one object acquires the state and behavior (data and methods) of another object.
- By use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent.

Advantages :

- ✓ Allows creating new class from existing class
- ✓ Software reusability

super class and subclass :

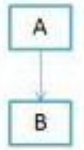
- A class that is inherited is called a ***super class (base class).*** *The class that* does the inheriting is called a ***subclass (derived class).***

General form of single level or simple inheritance:

class *subclass-name extends superclass-name {*

// body of class

}



(a) Single Inheritance

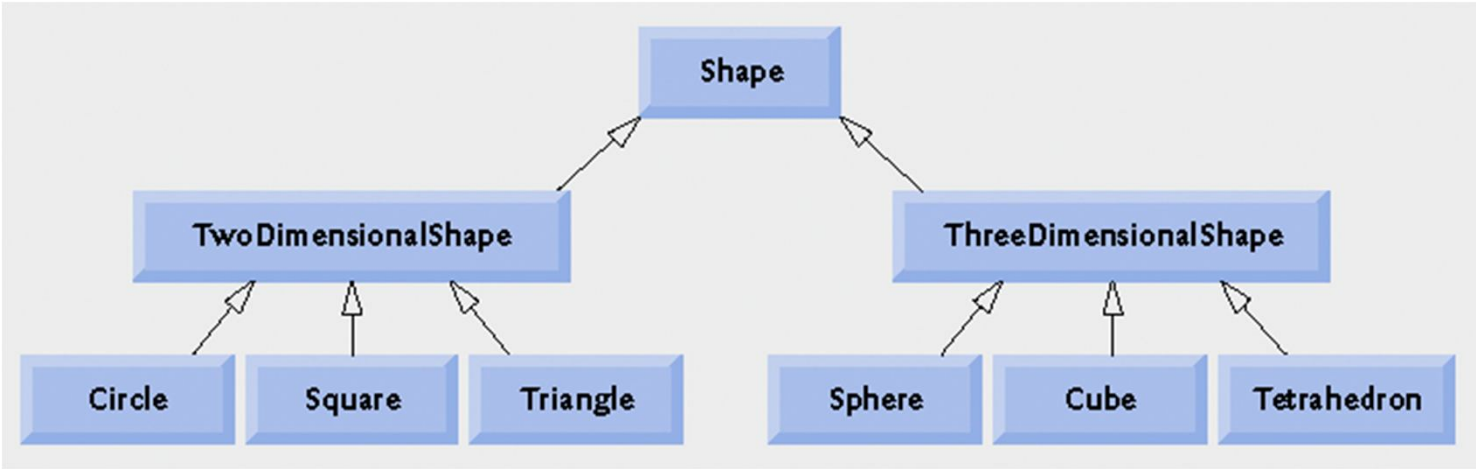
----> in the adjacent diagram, A is the Super class (base class)

B is the sub class (derived class)

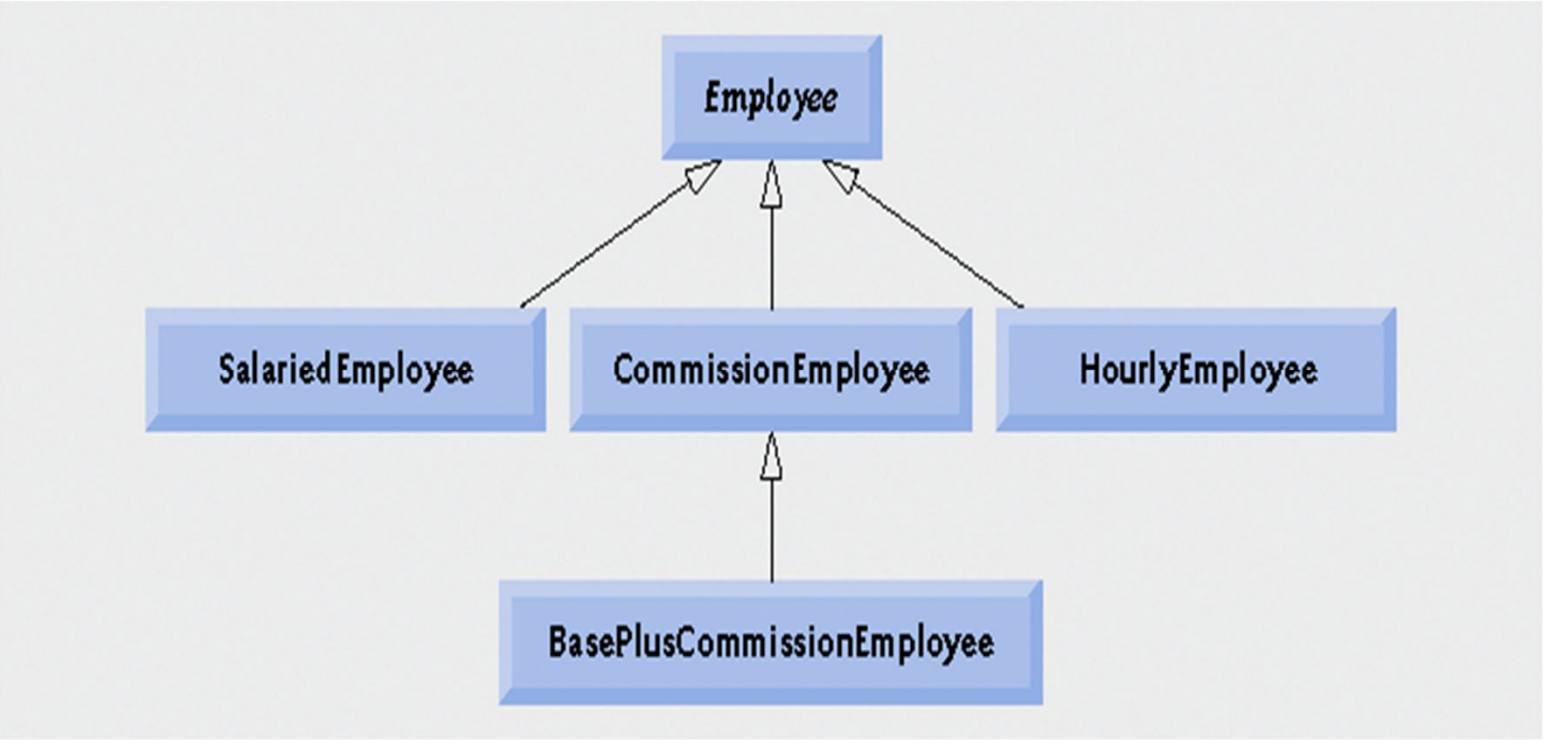
Inheritance

Inheritance examples :

Loan	**CarLoan, HomeImprovementLoan, MortgageLoan**
Employee	**Faculty, Staff**
BankAccount	**CheckingAccount, SavingsAccount**

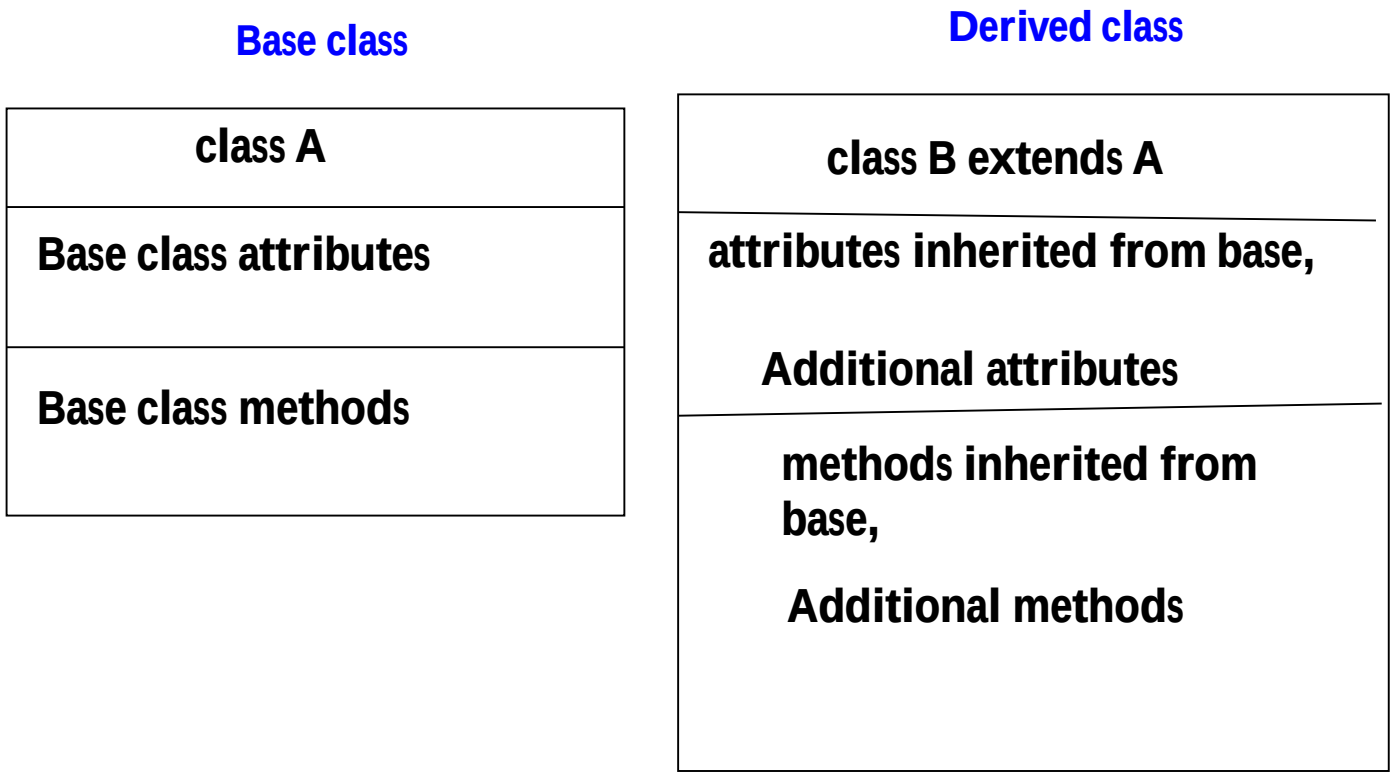


Inheritance



Inheritance

A derived class *extends* a base class. It inherits all of its methods (behaviors) and attributes (data) and it may have additional behaviors and attributes of its own.



Inheritance

Base class

Derived classes have their own constructors

Derived classes may add their own unique behaviors

```
class Animal
protected String name, sound;
protected double xpos, ypos;
public Animal(String n, String s) {..}
public void speak( ) {..}
public void moveTo(double x,double y){..}
public double getX( ) {..}
public double getY( ) {..}
```

class name

attributes (data)

methods (behavior)

Derived classes

```
class Dog  extends Animal
// no additional attributes
public Dog(String n) {..}
public void chaseCats(Cat c){.}
```

```
class Cat  extends Animal
// no additional attributes
public Cat(String n) {..}
public void runAway( ) {..}
```

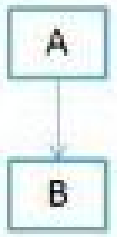
Inheritance types

- Single level inheritance
- Multi level inheritance
- Multiple inheritance (java doesn't support)
- Hierarchical Inheritance
- Hybrid inheritance

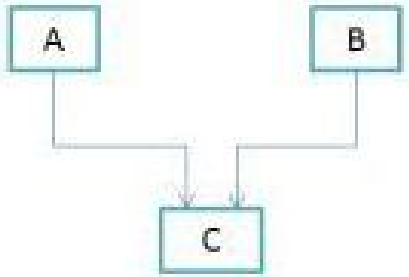
General form of single level or simple inheritance:

```
class subclass-name extends superclass-name {
// body of class
}
```

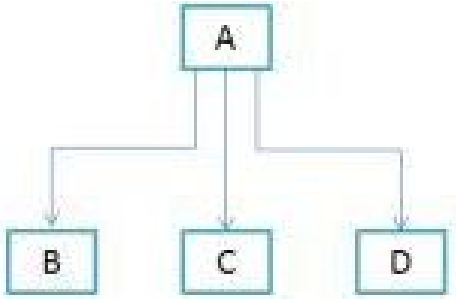
Inheritance types



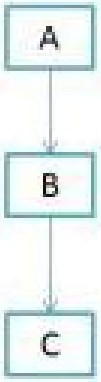
(a) Single Inheritance



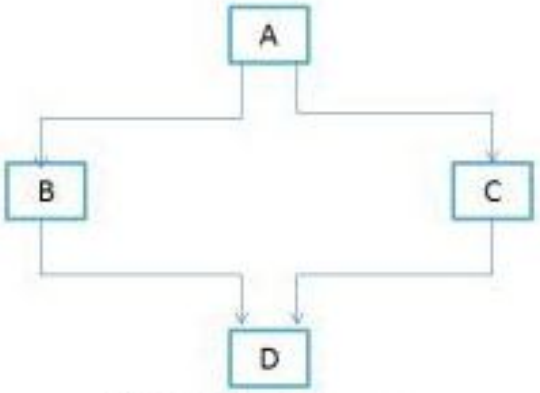
(b) Multiple Inheritance



(c) Hierarchical Inheritance



(d) Multilevel Inheritance



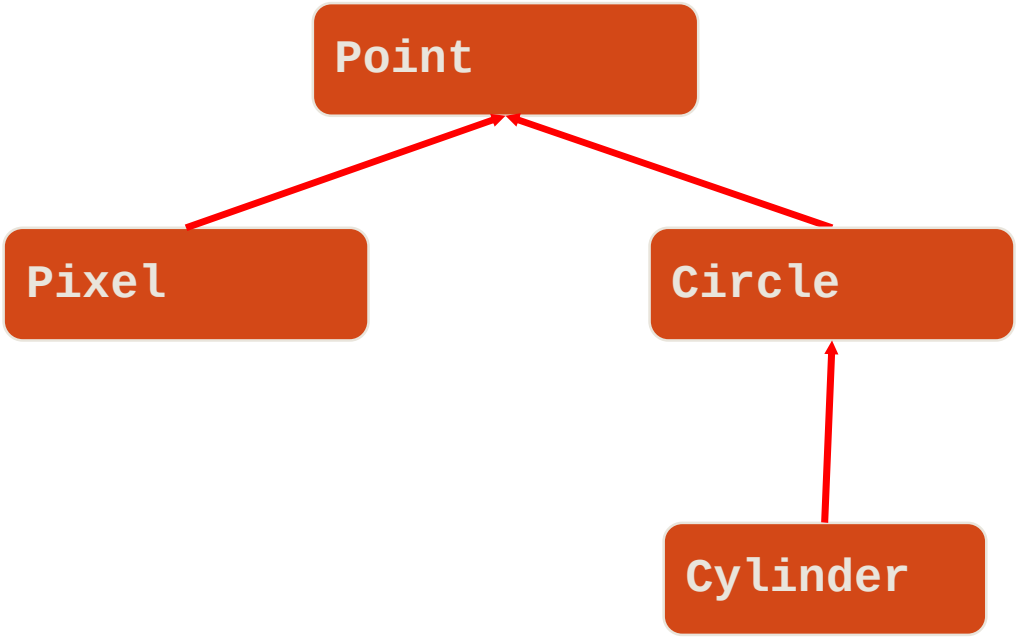
(e) Hybrid Inheritance

Single Inheritance

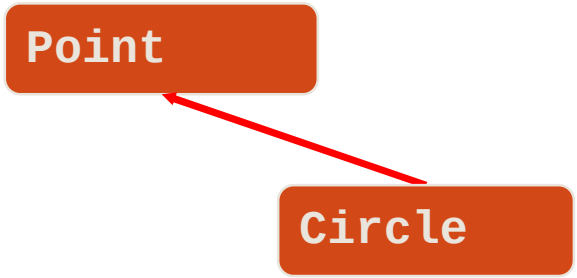
- Example

```
// Create a superclass.
class A {
    int i, j;
    void showij() {
    System.out.println("i and j: " + i + " " + j);
    }
}
// Create a subclass by extending class A.
class B extends A {
    int k;
    void showk() {
    System.out.println("k: " + k);
    }
    void sum() {
    System.out.println("i+j+k: " + (i+j+k));
    }
}
```

The class hierarchy



Single Inheritance example



super keyword

Ex :

```
   class A{
      int i;
   A() {  }
}
Class B extends A{
   int i;
  B(){
     super.i=10;i=20;  ….. }
}
```

super keyword

- Whenever a subclass needs to refer to its immediate super class, it can do so by use of the keyword ‘super’.

2 purposes of using super keyword :

1) To call the super class constructor in the derived class constructor.
2) To access super class member overridden by subclass member in the sub class.

Ex :

```
  class A{
     ;;;
  A() {  }
}
Class B extends A{
  ;;
  B(){
    super();  ..... }
}
```

Multi level inheritance

- inheriting from a subclass creates a class which inherits attributes and operations from more than one level.

Ex :

```
class A{
    …
  A() {  }
}
Class B extends A{
  …
  B(){
   … }
}
Class C extends B{
   …
 C(){ … }
}
```

Method Overriding

- When a method in a subclass has the same name and type signature as a method in its super class, then the method in the subclass is said to override the method in the super class.

Ex :

```
class A{
   ;;;
  A() { … }
  void m1(){ … }
}
Class B extends A{
   …
  B(){ … }
  void m1(){…} //m1 method is overriden
}
```

How to override a method

- Create a method in a subclass having the same *signature* as a method in a superclass
- That is, create a method in a subclass having the same name and the same number and types of parameters
  - Parameter *names* don't matter, just their *types*

Restrictions:

- The return type must be the same
- The overriding method cannot be *more private* than the method it overrides

Overload & Override

- You should *overload* a method when you want to do essentially the same thing, but with different type of data.
- You should *override* an inherited method parameters if you want to do something slightly different than in the superclass
  - It's almost always a good idea to override public void toString() -- it's handy for debugging, and for many other reasons
- You should never intentionally *shadow* a variable

final Methods and Classes

- final methods
  - Cannot be overridden in a subclass
  - private and static methods are implicitly final
  - final methods are resolved at compile time, this is known as static binding
    - Compilers can optimize by inlining the code
- final classes
  - Cannot be extended by a subclass
  - All methods in a final class are implicitly final

Abstract class

- To declare a class abstract, you simply use the **abstract keyword in front of the class keyword,** at the beginning of the class declaration.
- Abstract class contains atleast one abstract method.
- Abstract method declaration:

  abstract *type name(parameter-list);*
- an abstract class cannot be directly instantiated with the **new operator**
- you 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**

- **Example:**

```
abstract class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
// area is now an abstract method
abstract double area();
}
```

Object class

- Object class is the super class for all classes in java
- To an Object class reference we can assign any type of Object.

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object)*	*Determines whether one object is equal to* another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( )	Waits on another thread of execution
void wait(long *milliseconds)*	
void wait(long *milliseconds*, int *nanoseconds)*	

Packages & Interfaces

Packages

- Packages are containers for classes that are used to keep the class namespace compartmentalized.
- Packages avoids class name space collisions that may occur in integration of java projects.
- A package is both a naming and a visibility control mechanism.
- With packages, Java programmers can freely define classes without worrying about the class names of other programmers of the project.

Defining a package: package pkgname; ---- as first stmt in the program

package statement specifies to which package the classes defined in a file belong.

- Java stores packages in file system directories.
- Package name should be same as that of the name of the directory holding it.
- Packages can be created hierarchically.

  Syntax: package pkg1[.[pkg2].[pkg3]];

  Ex: package java.awt.image;

Finding packages and CLASSPATH

- JVM uses current working directory by default to search for the class files. If they are found in a sub dir, JVM loades them. Otherwise it searches in a directories specified in the CLASSPATH environment variables.

  Method 1: Place ur package in the current working dir.

  Method 2: Set CLASSPATH to include path to the package. (excluding packagename in path)

  Note: In this case, you can run your program from any location in file system.

  Method 3: using –classpath compiler option in javac command to specify path to the classes.

  For a package 'MyPack' in packagaes directory, use the following syntax:

  ```
  E:\>java -classpath d:\java2014exs\packages MyPack.AccountBalance
  ```

import statement

- import statement allows you to bring certain classes, or entire packages into visibility.

Syntax:

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

Ex:

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

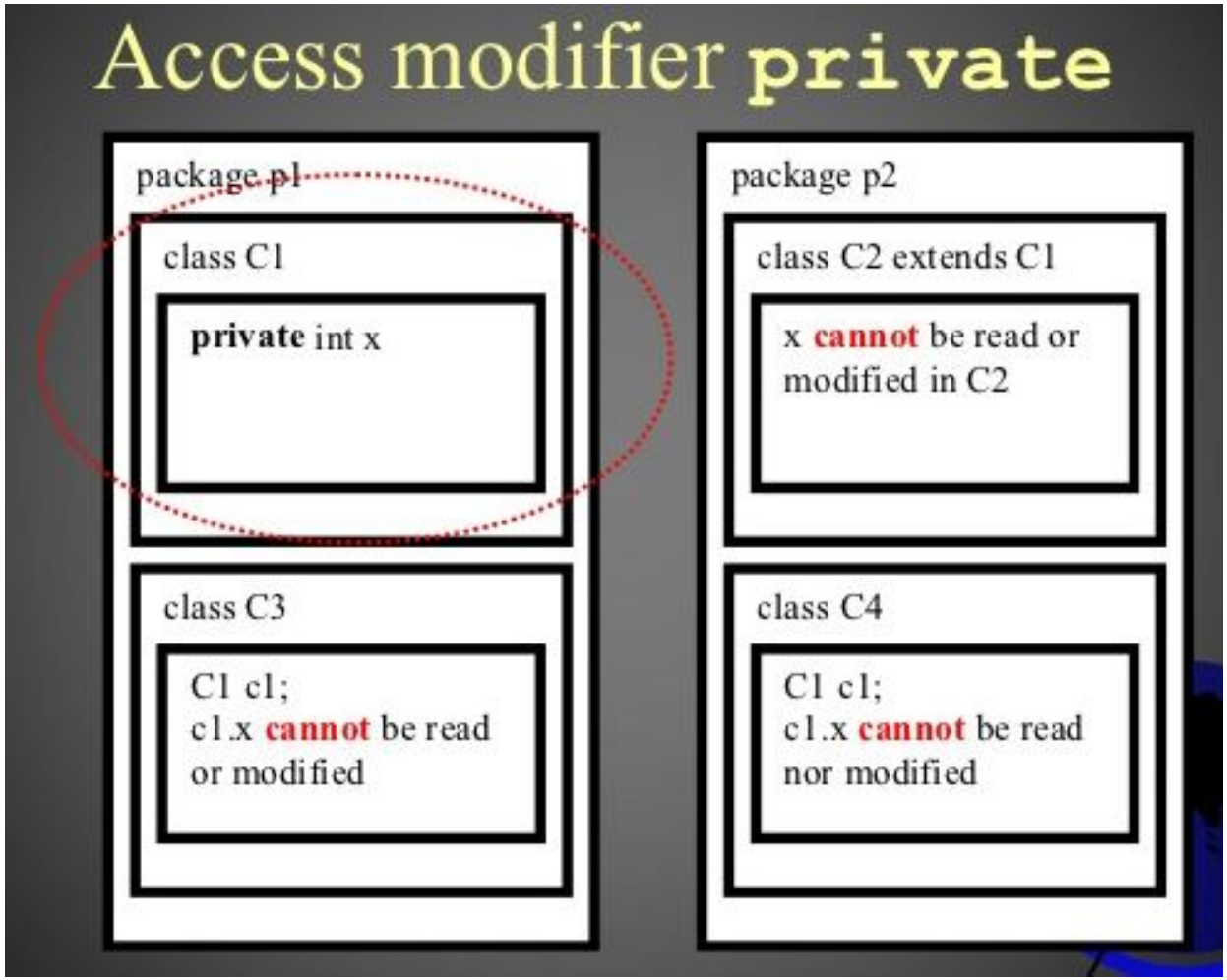
- Java Compiler automatically imports java.lang package

Packages

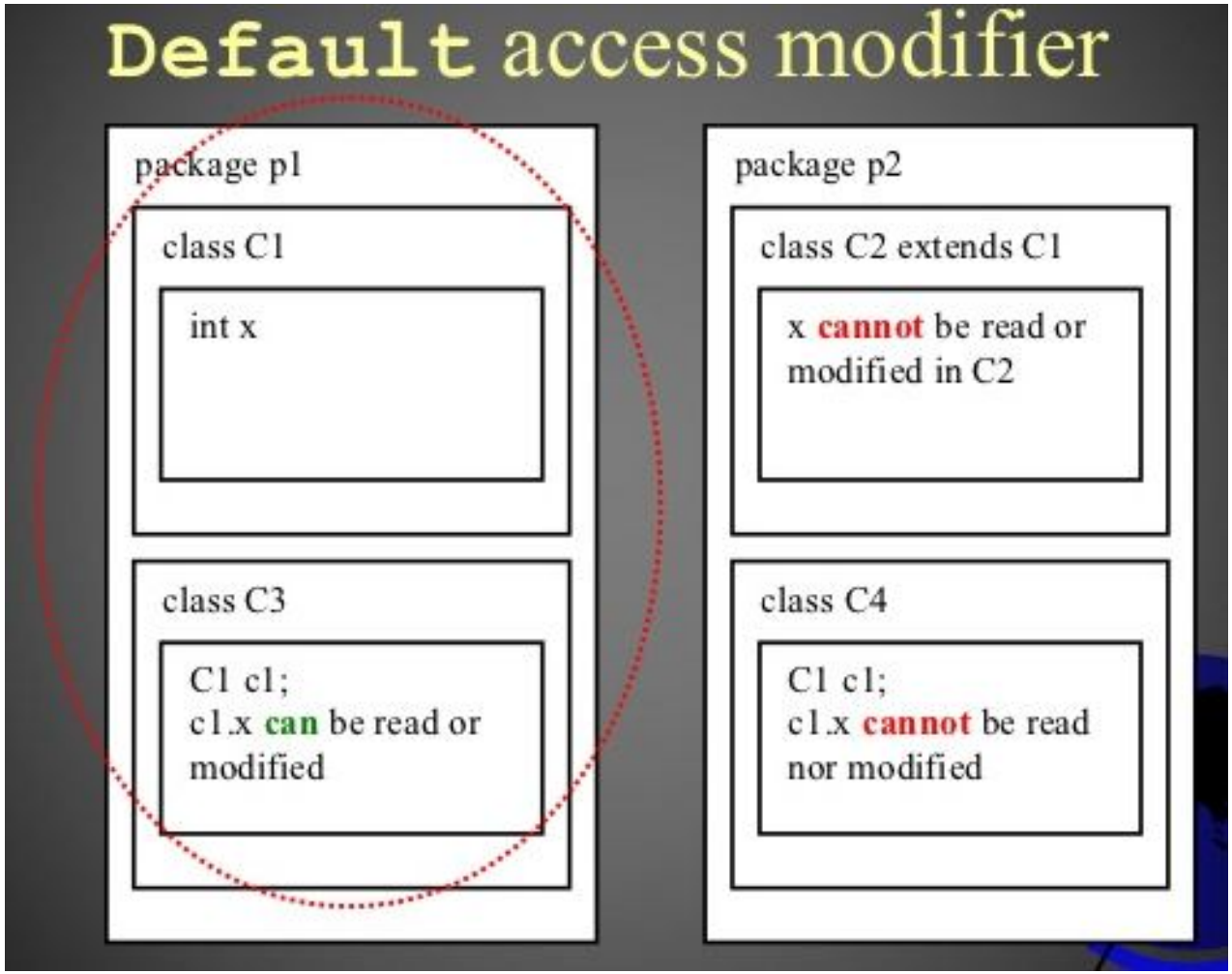
	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Class Member Access

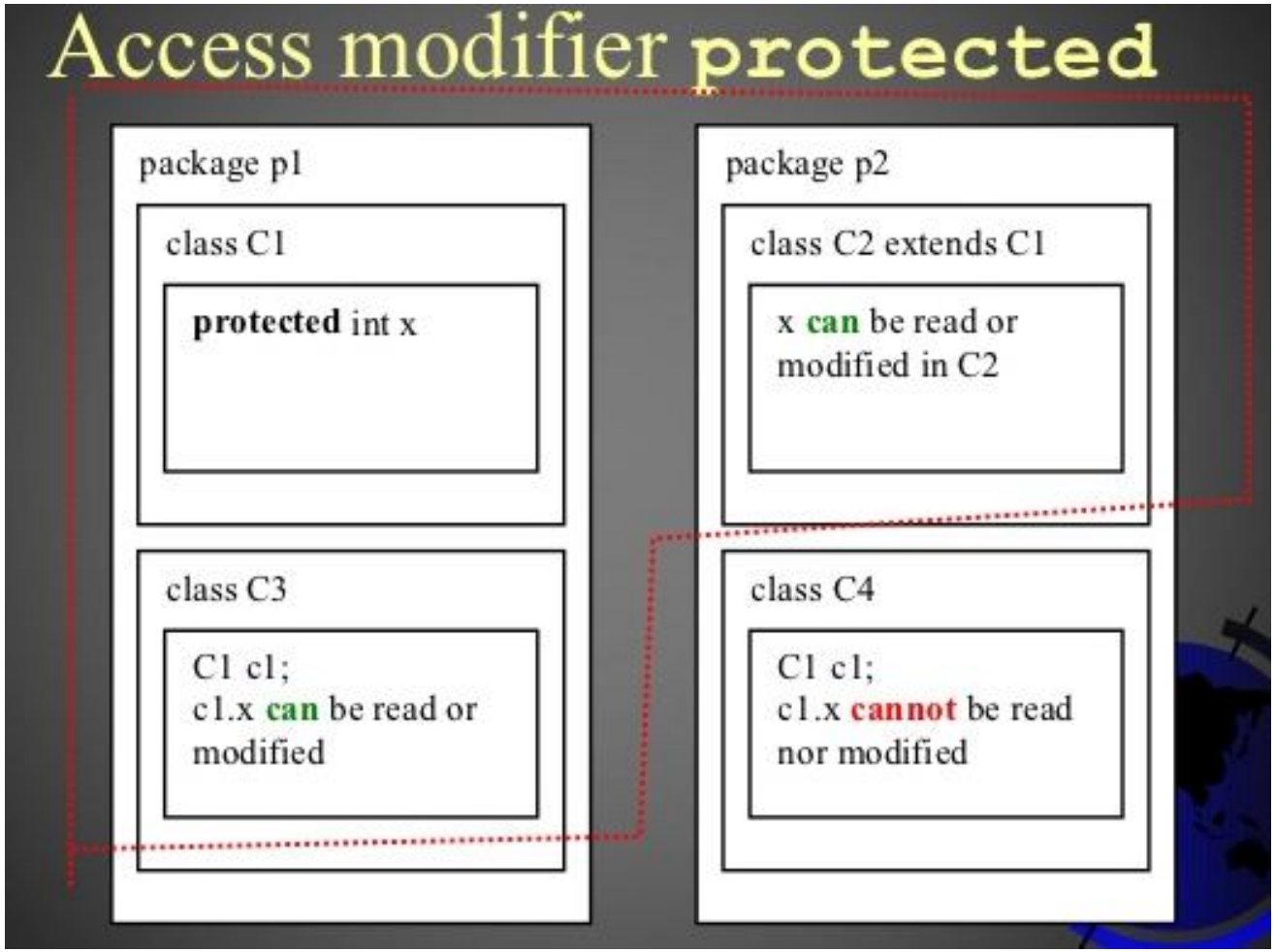
private access specifier



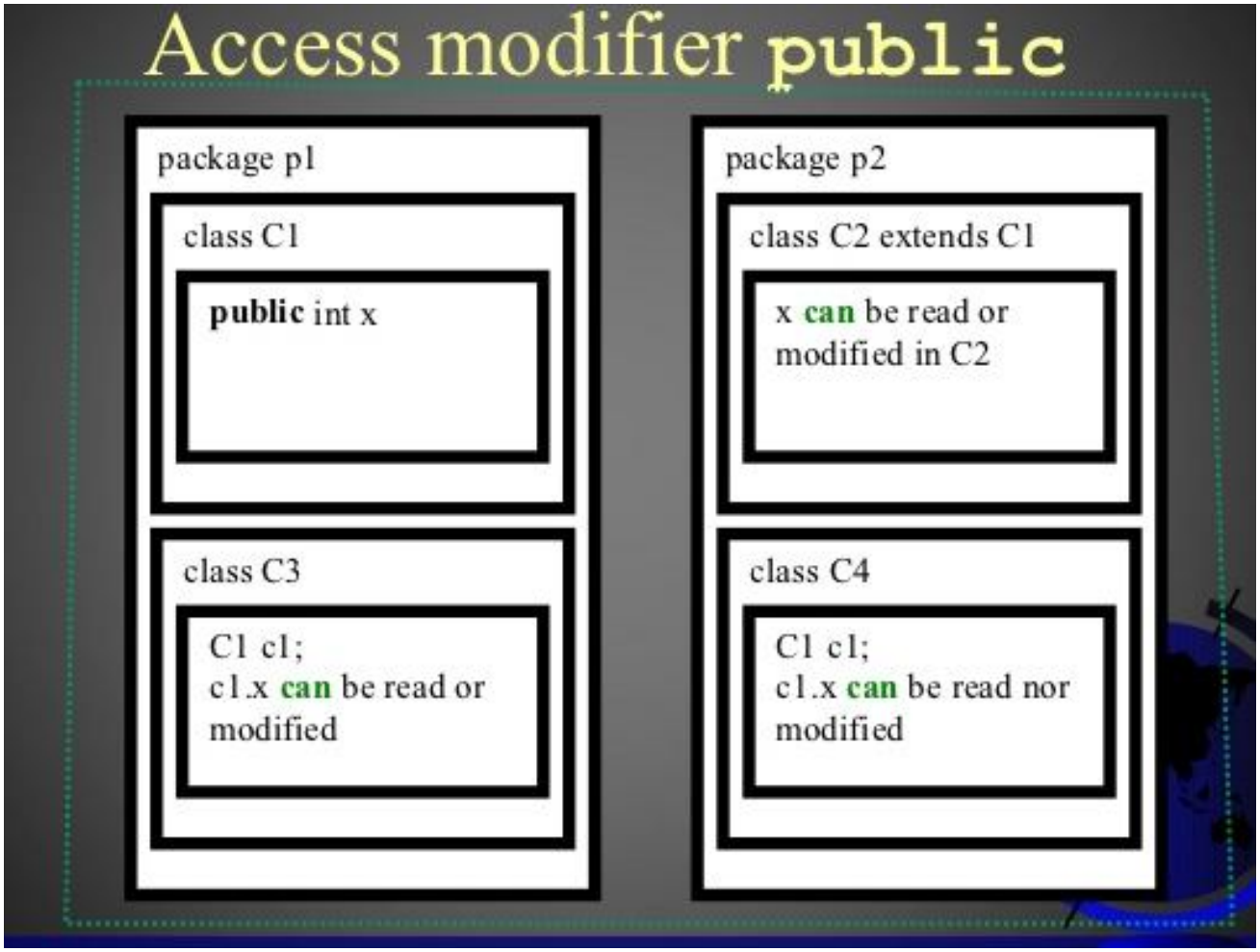
default access specifier



protected access specifier



public access specifier



Interfaces

- Interface is a way of describing what classes should do, without specifying how they should do it.
- With interface, you can fully abstract a class's interface from its implementation.
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- Any number of classes can implement an interface
- One class can implement any number of interfaces
- When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface.
- Using interfaces is one way of achieving run time polymorphism.
- Interfaces is an alternative way to implement multiple inheritance.
- By disconnecting the definition of a method or a set of methods from the inheritance hierarchy, the functionality is not exposed to more sub classes. The hierarchy of interfaces is different than that of the classes.

  Different classes which are unrelated in terms of class hierarchy can implement the same interface.

Interfaces

Definition :

```
access interface name {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
        type final-varname1 = value;
        type final-varname2 = value;
        // ...
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
}
```

- ✓ access is either public or default (if not specified it assumes public w.r.t package)
- ✓ *Members in an interface are by default public*
- ✓ *Variables in interface are by default ' public static '*

Ex :

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

Interfaces

Implementing interfaces :

access class classname [extends superclass]

[implements *interface [,interface...]]*

{

/ / class-body

}

- ✓ access is either public or not used
- ✓ The methods that implement an interface must be declared **public.**

Ex :

```
class Client implements Callback {
    / / Implement Callback's interface
    public void callback(int p) {
    System.out.println("callback called with " + p);
    }
}
```

Interfaces

Accessing Implementations Through Interface References :

- ✓ Interface reference can refer objects of a class that implements the interface.
- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.

Points :

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

Variables in interfaces :

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

Ex :

```
interface SharedConstants {
int NO = 0;
intYES = 1;
}
```

Interfaces

- We can think of a class using an interface as being like an electric wall outlet , and think of implementation as the plug. The outlet doesn't care what is behind the plug, as long as it fits in the outlet.s

class interface

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Extending Interfaces

- An interface can extend another interface, similarly to the way that a class can extend another class.
- The **extends** keyword is used to extend an interface

Ex:

```
public interface Sports
{
  public void setHomeTeam(String name);
  public void setVisitingTeam(String name);
}
interface Football extends Sports
{
  public void homeTeamScored(int points);
  public void visitingTeamScored(int points);
  public void endOfQuarter(int quarter);
}
interface Hockey extends Sports
{
  public void homeGoalScored();
  public void visitingGoalScored();
  public void endOfPeriod(int period);
  public void overtimePeriod(int ot);
}
```

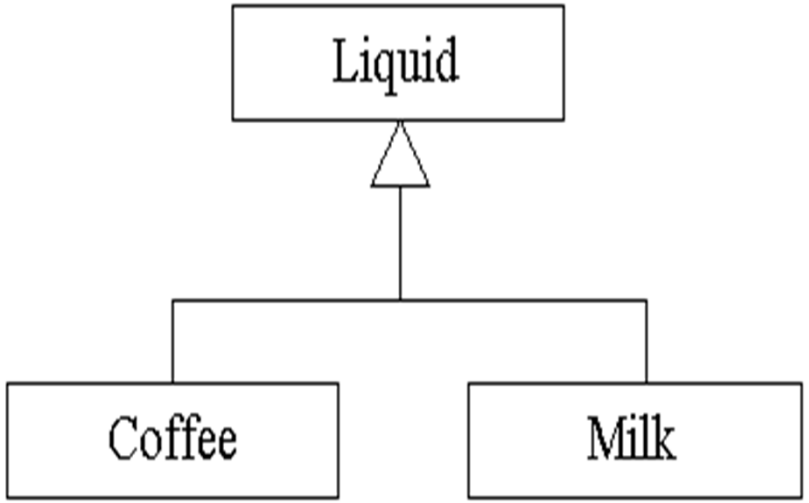
Extending Multiple Interfaces

- A Java class can only extend one parent class. Multiple inheritance is not allowed.
- An interface can extend more than one parent interface.

Ex :

```
public interface Hockey extends Sports, Event
{
    ....
}
```

Polymorphism & Interfaces



Example

```
Class Liquid {
  void Swirl(boolean clockwise) {    }
}
Class Coffee extends Liquid{
void Swirl(boolean clockwise) {    }
}
class Milk extends Liquid {
     void swirl(boolean clockwise) {   }
}
```

String class

String class constructors

Constructor	Description
String()	This initializes a newly created String object so that it represents an empty character sequence.
String(byte[] bytes)	This constructs a new String by decoding the specified array of bytes using the platform's default charset.
String(byte[] bytes, Charset charset)	This constructs a new String by decoding the specified array of bytes using the specified charset.
String(byte[] bytes, int offset, int length)	This constructs a new String by decoding the specified subarray of bytes using the platform's default charset
String(byte[] bytes, int offset, int length, Charset charset)	This constructs a new String by decoding the specified subarray of bytes using the specified charset.
String(byte[] bytes, String charsetName)	This constructs a new String by decoding the specified array of bytes using the specified charset.

String class constructors

Constructor	Description
String(char[] value)	This allocates a new String so that it represents the sequence of characters currently contained in the character array argument.
String(char[] value, int offset, int count)	This allocates a new String that contains characters from a subarray of the character array argument.
String(String original)	This initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.
String(StringBuffer buffer)	This allocates a new string that contains the sequence of characters currently contained in the string buffer argument.
String(StringBuilder builder)	This allocates a new string that contains the sequence of characters currently contained in the string builder argument.

String class methods

Method	Description
char charAt(int index)	This method returns the char value at the specified index.
int compareTo(String anotherString)	This method compares two strings lexicographically.
int compareToIgnoreCase(String str)	This method compares two strings lexicographically, ignoring case differences.
String concat(String str)	This method concatenates the specified string to the end of this string.
boolean contains(CharSequence s)	This method ceturns true if and only if this string contains the specified sequence of char values.
boolean contentEquals(StringBuffer sb)	This method compares this string to the specified StringBuffer.

String class methods

Method	Description
boolean endsWith(String suffix)	This method tests if this string ends with the specified suffix.
boolean equals(Object anObject)	This method compares this string to the specified object.
boolean equalsIgnoreCase(String anotherString)	This method compares this String to another String, ignoring case considerations.
static String format(String format, Object... args)	This method returns a formatted string using the specified format string and arguments.
byte[] getBytes()	This method encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
byte[] getBytes(Charset charset)	This method encodes this String into a sequence of bytes using the given charset, storing the result into a new byte array.
byte[] getBytes(String charsetName)	This method encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.

String class methods

Method	Description
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	This method copies characters from this string into the destination character array.
int hashCode()	This method returns a hash code for this string.
int indexOf(int ch)	This method returns the index within this string of the first occurrence of the specified character.
int indexOf(int ch, int fromIndex)	This method returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
int indexOf(String str)	This method returns the index within this string of the first occurrence of the specified substring.
int indexOf(String str, int fromIndex)	This method returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
boolean isEmpty()	This method returns true if, and only if, length() is 0.

String class methods

Method	Description
int lastIndexOf(int ch)	This method returns the index within this string of the last occurrence of the specified character.
int lastIndexOf(int ch, int fromIndex)	This method returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
int lastIndexOf(String str)	This method returns the index within this string of the rightmost occurrence of the specified substring.
int lastIndexOf(String str, int fromIndex)	This method returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
int length()	This method returns the length of this string.
boolean matches(String regex)	This method tells whether or not this string matches the given regular expression.
String replace(char oldChar, char newChar)	This method returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
boolean startsWith(String prefix)	This method tests if this string starts with the specified prefix.

String class methods

Method	Description
boolean startsWith(String prefix, int toffset)	This method tests if the substring of this string beginning at the specified index starts with the specified prefix.
String substring(int beginIndex)	This method returns a new string that is a substring of this string.
String substring(int beginIndex, int endIndex)	This method returns a new string that is a substring of this string. (beginIndex to endIndex-1)
char[] toCharArray()	This method converts this string to a new character array.
String toLowerCase()	This method converts all of the characters in this String to lower case using the rules of the default locale.
String toString()	This method returns the string itself.
String toUpperCase()	This method converts all of the characters in this String to upper case using the rules of the default locale
String trim()	This method returns a copy of the string, with leading and trailing whitespace omitted

String class methods

Method	Description
static String valueOf(boolean b)	This method returns the string representation of the boolean argument.
static String valueOf(char c)	This method returns the string representation of the char argument.
static String valueOf(char[] data)	This method returns the string representation of the char array argument.
static String valueOf(char[] data, int offset, int count)	This method Returns the string representation of a specific subarray of the char array argument.
static String valueOf(double d)	This method returns the string representation of the double argument.
static String valueOf(float f)	This method returns the string representation of the float argument.
static String valueOf(int i)	This method returns the string representation of the int argument.
static String valueOf(long l)	This method returns the string representation of the long argument.
static String valueOf(Object obj)	This method returns the string representation of the Object argument.

String class

String class constructors :

String s = new String();

1) String(char *chars[ ])*

Ex : char chars[] = { 'a', 'b', 'c' };

String s = new String(chars);

2) String(char *chars[ ], int startIndex, int numChars)*

Ex : char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

String s = new String(chars, 2, 3);

3) String(String *strObj)*

```
char c[] = {'J', 'a', 'v', 'a'};
String s1 = new String(c);
String s2 = new String(s1);

int length( )
```

String class

```
// Construct one String from another.
class MakeString {
    public static void main(String args[]) {
        char c[] = {'J', 'a', 'v', 'a'};
        String s1 = new String(c);
        String s2 = new String(s1);
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

String class

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

String class methods :

1) char charAt(int *where)*

Ex : char ch;

```
ch = "abc".charAt(1);
```

2) void getChars(int *sourceStart, int sourceEnd, char target[ ], int targetStart)*

3) byte[ ] getBytes( )

4) char[ ] toCharArray( )

5) boolean equals(Object *str)*

boolean equalsIgnoreCase(String *str)*

String class methods

startsWith( ) and endsWith( )

boolean startsWith(String *str)*

boolean endsWith(String *str)*

```
class EqualsNotEqualTo {
        public static void main(String args[]) {
        String s1 = "Hello";
        String s2 = new String(s1);
        System.out.println(s1 + " equals " + s2 + " -> " +
        s1.equals(s2));
        System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
        }
}
```

String class methods

- int compareTo(String *str)*
- int indexOf(int *ch)*
- int indexOf(int *ch, int startIndex)*
- int indexOf(String *str)*
- int indexOf(String *str, int startIndex)*
- int lastIndexOf(int *ch)*
- int lastIndexOf(int *ch, int startIndex)*
- int lastIndexOf(String *str)*
- int lastIndexOf(String *str, int startIndex)*

String class methods

- String substring(int *startIndex)*
- String substring(int *startIndex, int endIndex)*
  - *Extracts chars from startIndex to endIndex-1 positions.*
- String concat(String *str)*
- String replace(char *original, char replacement)*
- String trim( )
- String toLowerCase( )
- String toUpperCase( )

I/O Streams

Introduction

- ***Stream***: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)
  - it acts as a buffer between the data source and destination
- ***Input stream***: a stream that provides input to a program
  - `System.in` is an input stream
  - **System.in** is an object of type **InputStream**
- ***Output stream***: a stream that accepts output from a program
  - `System.out` is an output stream
  - **System.out & System.err** are objects of type **PrintStream**
- A stream connects a program to an I/O object
  - `System.out` connects a program to the screen
  - `System.in` connects a program to the keyboard

Byte Stream Classes

Stream Class	Description
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output
RandomAccessFile	Supports random access file I/O

Character Stream IO classes

Stream Class	Description
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file

Character Stream IO classes

Classes	Description
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( ) and println( )**
PushbackReader	Input stream that allow characters to be returned to the input stream
Reader	Abstract class that describes character stream input

Character Stream IO classes

Classes	Description
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

I/O

File class method : String[ ] list( )

Byte Oriented IO streams:

InputStream, OutputStream

Character oriented IO streams:

Reader, Writer

InputStream methods:

int available( )

void close( )

void mark(int *numBytes)*

boolean markSupported( )

int read( )

int read(byte *buffer[ ])*

int read(byte *buffer[ ], int offset,* int *numBytes)*

void reset( )

long skip(long *numBytes)*

OutputStream

OutputStream methods:

void close( )

void flush( )

void write(int *b*)

void write(byte *buffer[ ]*)

void write(byte *buffer[ ]*, *int offset*, int *numBytes*)

Reading and Writing Files

- FileInputStream(String *fileName) throws FileNotFoundException*
- FileOutputStream(String *fileName) throws FileNotFoundException*
- void close( ) throws IOException
- int read( ) throws IOException

FileInputStream

- FileInputStream(String *filepath)*
- FileInputStream(File *fileObj)*
- FileInputStream f0 = new FileInputStream("/autoexec.bat“)
- File f = new File("/autoexec.bat");
- FileInputStream f1 = new FileInputStream(f);

Character Stream IO classes

Classes	Description
• FilterReader	Filtered reader
• FilterWriter	Filtered writer
• InputStreamReader	Input stream that translates bytes to characters
• OutputStreamWriter	Output stream that translates characters to bytes
• PipedReader	Input pipe
• PipedWriter	Output pipe
• PrintWriter	Output stream that contains **print( ) and println( )**
• PushbackReader	Input stream that allow characters to be returned to the input stream
• Reader	Abstract class that describes character stream input

Character Stream IO classes

Classes	Description
• StringReader	Input stream that reads from a string
• StringWriter	Output stream that writes to a string
• Writer	Abstract class that describes character stream output

Character Streams

Reader class :

Methods:

abstract void close( )

void mark(int *numChars)*

boolean markSupported( )

int read( )

int read(char *buffer[ ])*

abstract int read(char *buffer[ ],* int *offset,* int *numChars)*

boolean ready( )

void reset( )

long skip(long *numChars)*

Buffering

- **Not buffered:** each byte is read/written from/to disk as soon as possible
  - "little" delay for each byte
  - A disk operation per byte---higher overhead
- **Buffered:** reading/writing in "chunks"
  - Some delay for some bytes
    - Assume 16-byte buffers
    - Reading: access the first 4 bytes, need to wait for all 16 bytes are read from disk to memory
    - Writing: save the first 4 bytes, need to wait for all 16 bytes before writing from memory to disk
  - A disk operation per a buffer of bytes---lower overhead

File I/O

- *All* data and programs are ultimately just zeros and ones
  - each digit can have one of two values, hence *binary*
  - *bit* is one binary digit
  - *byte* is a group of eight bits
- *Text files*: the bits represent printable characters
  - one byte per character for ASCII, the most common code
  - for example, Java source files are text files
  - so is any file created with a "text editor"
- *Binary files*: the bits represent other types of encoded information, such as executable instructions or numeric data
  - these files are easily read by the computer but not humans
  - they are *not* "printable" files
    - actually, you *can* print them, but they will be unintelligible
    - "printable" means "easily readable by humans when printed"

File class

- Acts like a wrapper class for file names
- A file name like "`numbers.txt`" has only `String` properties
- `File` has some very useful methods
  - `exists`: tests if a file already exists
  - `canRead`: tests if the OS will let you read a file
  - `canWrite`: tests if the OS will let you write to a file
  - `delete`: deletes the file, returns true if successful
  - `length`: returns the number of bytes in the file
  - `getName`: returns file name, excluding the preceding path
  - `getPath`: returns the path name—the full name

```
File numFile = new File(“numbers.txt”);
if (numFile.exists())
  System.out.println(numfile.length());
```

File class

File class:

Constructors:

File(String *directoryPath)*
File(String *directoryPath*, *String filename)*
File(File *dirObj*, *String filename)*
File(URI *uriObj)*

Methods:

boolean canRead()
boolean **canWrite**()
boolean **delete**()
deleteOnExit()
boolean **exists**()
String **getAbsolutePath**()
String **getParent**()

File Class

Methods :

String **getPath**()

boolean **isDirectory**()

boolean **isFile**()

boolean **isHidden**()

long **lastModified**()

long **length**()

boolean **mkdir**()

boolean **renameTo**(File dest)

boolean **setLastModified**(long time)

boolean **setReadOnly**()

Collections

Collections

Java.util package collections:

It is a hierarchy of interfaces & classes that provide functionalities for managing group of objects.

- ArrayList
- LinkedList
- Hashtable
- Vector
- Stack

✓ A List may have duplicate values

✓ A Set will not allow duplicate values

ArrayList :

- Extends AbstractList and implements List.
- Supports dynamic arrays.
- It is a variable length array of object references.

Constructors :

ArrayList()

ArrayList(Collection c)

ArrayList(int capacity)

ArrayList constructors:

Name	Description
ArrayList( )	builds an empty array list.
ArrayList(Collection c)	builds an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	builds an array list that has the specified initial capacity

ArrayList methods

Name	Description
void add(int index, Object element)	Inserts the specified element at the specified position index in this list.
boolean add(Object o)	Appends the specified element to the end of this list
boolean addAll(Collection c)	Appends all of the elements in the specified collection to the end of this list.
boolean addAll(int index, Collection c)	Inserts all of the elements in the specified collection into this list, starting at the specified position.
void clear()	Removes all of the elements from this list.
Object clone()	Returns a shallow copy of this ArrayList.
boolean contains(Object o)	Returns true if this list contains the specified element.
void ensureCapacity(int minCapacity)	Ensures the capacity of the Collection to the minCapacity specified.
Object get(int index)	Returns the element at the specified position in this list

ArrayList methods

Name	Description
int indexOf(Object o)	Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
int lastIndexOf(Object o)	Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object remove(int index)	Removes the element at the specified position in this list.
protected void removeRange(int fromIndex, int toIndex)	Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
Object set(int index, Object element)	Replaces the element at the specified position in this list with the specified element.
int size()	Returns the number of elements in this list.
Object[] toArray()	Returns an array containing all of the elements in this list in the correct order.
void trimToSize()	Trims the capacity of this ArrayList instance to be the list's current size.

ArrayList methods

- void **add**(int index, Object element)
- boolean **add**(Object o)
- boolean **addAll**(Collection c)
- boolean **addAll**(int index, Collection c)
- void **clear**()
- Object **clone**()
- boolean **contains**(Object elem)
- void **ensureCapacity**(int minCapacity)
- Object **get**(int index)
- int **indexOf**(Object elem)
- boolean **isEmpty**()
- int **lastIndexOf**(Object elem)

ArrayList methods

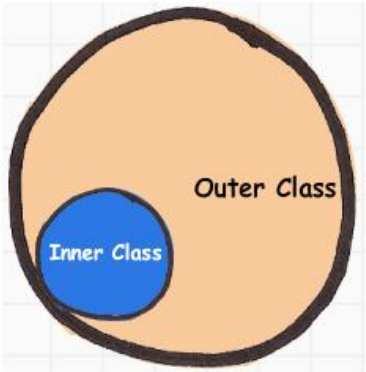
- Object **remove**(int index)
- protected void **removeRange**(int fromIndex, int toIndex)
- Object **set**(int index, Object element)
- int **size**()
- Object[] **toArray**()
- void **trimToSize**()

Nested classes and inner classes

- A nested class is a class defined inside the definition (body) of another enclosing class.
- A nested class can be of two types - static or non-static.
- Nested class improves encapsulation and maintenance.
- If a class can only be useful to one particular class, it makes sense to .keep that inside the class itself.
- A non static nested class is called inner class.

Why Use Nested Classes?

- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation.
- Nested classes can lead to more readable and maintainable code.



Nested & Inner classes

```
class OuterClass
{
    ...
      static class StaticNestedClass
     {
        ....
     }
    class InnerClass
     {
        ....
        void getResult()
        {
           .....
           class LocalInnerClass
           {
              ...
           }
        }
     }
    ...
}
```

Nested classes and inner classes

Static Nested class : A nested class defined with keyword static is known as static Nested class.

- Static nested classes are accessed using the enclosing class name:

  OuterClass.StaticNestedClass

- Static nested classes **do not have access to other members** of the enclosing class.
- to create an object for the static nested class, use this syntax:

OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

Non Static Nested Class : Non static nested class is also known as "**Inner Class"**.

- A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.
- To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

  OuterClass.InnerClass innerObject = outerObject.new InnerClass();

Advantages of inner classes

Logical grouping of classes— If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

Increased encapsulation—Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

More readable, maintainable code—Nesting small classes within top-level classes places the code closer to where it is used.