

Session 1

Programs

Computer *programs*, known as *software*, are instructions to the computer.

You tell a computer what to do through programs. Without programs, a computer is an empty machine. Computers do not understand human languages, so you need to use computer languages to communicate with them.

Programs are written using programming languages.

Programming Languages

Machine Language Assembly Language High-Level Language

Machine language is a set of primitive instructions built into every computer. The instructions are in the form of binary code, so you have to enter binary codes for various instructions. Program with native machine language is a tedious process. Moreover, the programs are highly difficult to read and modify. For example, to add two numbers, you might write an instruction in binary like this:

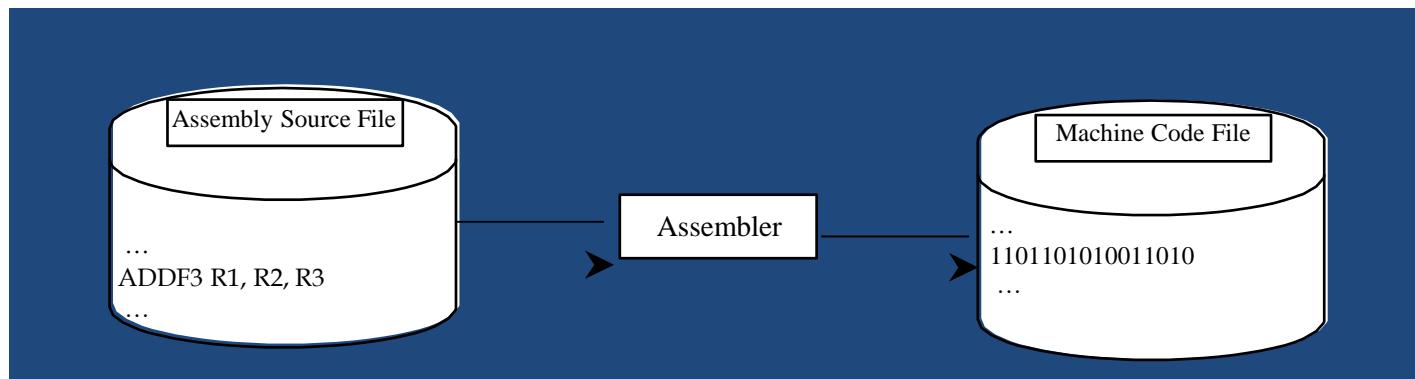
1101101010011010

Programming Languages

Machine Language Assembly Language High-Level Language

Assembly languages were developed to make programming easy. Since the computer cannot understand assembly language, however, a program called assembler is used to convert assembly language programs into machine code. For example, to add two numbers, you might write an instruction in assembly code like this:

ADDF3 R1, R2, R3



Programming Languages

Machine Language Assembly Language High-Level Language

The high-level languages are English-like and easy to learn and program. For example, the following is a high-level language statement that computes the area of a circle with radius 5:

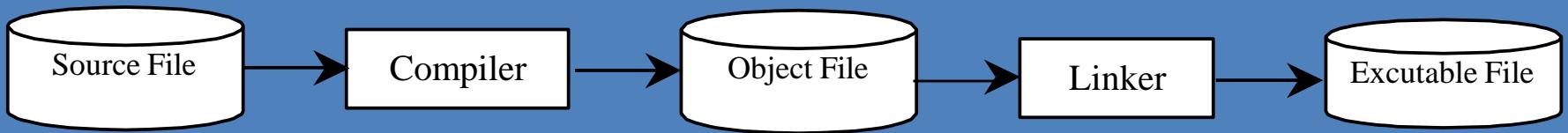
```
area = 5 * 5 * 3.1415;
```

Popular High-Level Languages

- COBOL (COmmon Business Oriented Language)
- FORTRAN (FORmula TRANslatiOn)
- BASIC (Beginner All-purpose Symbolic Instructional Code)
- Pascal (named for Blaise Pascal)
- Ada (named for Ada Lovelace)
- C (whose developer designed B first)
- Visual Basic (Basic-like visual language developed by Microsoft)
- Delphi (Pascal-like visual language developed by Borland)
- C++ (an object-oriented language, based on C)
- Java (We will use it in the course)

Compiling Source Code

A program written in a high-level language is called a *source program*. Since a computer cannot understand a source program. Program called a *compiler* is used to translate the source program into a machine language program called an *object program*. The object program is often then linked with other supporting library code before the object can be executed on the machine.



Popular Programming paradigms

- 1 Structured programming techniques
- 2 Procedural-oriented programming
- 3 Functional programming
- 4 Logic programming
- 5 Event-driven programming
- 6 Object-oriented programming

Structured programming [Cont....]

- Structured programming techniques involve giving the code you write structures; these often involve writing code in blocks such as:
 - Sequence - code executed line by line
 - Selection - branching statements such as if..then..else, or case.
 - Repetition - iterative statements such as for, while, repeat, loop, do, until.

Procedural-oriented programming [Cont....]

- Sharing the same features as structured programming techniques, Procedural-oriented programming implement procedures/subroutines to execute common functionality

Functional programming [Cont....]

- In functional programming programs define mathematical functions. A solution to a problem consists of a series of function calls.
- There are no variables or assignment statements, but instead there are lists and functions that manipulate these lists.
- An example of a functional programming is R language.

Logic programming [Cont....]

- A logic program consists of a set of facts and rules.
- A knowledge base is built up about a specific subject and an inference engine uses the knowledge base to answer queries which are presented in the form of a goal.
- Logic programming is often used for artificial intelligence systems for example Prolog.

Event-driven programming [Cont....]

- Event Driven programming refers to your standard Windows Form idea, the program waits in a loop until an event(e.g. the click of a button, or a keystroke).
- The program then runs the code associated with this event and then returns to its loop, providing that the code did not instruct it to close.
- If more than one event occurs, code is queued by the program and run in the order the events were triggered.

Object-oriented programming [Cont....]

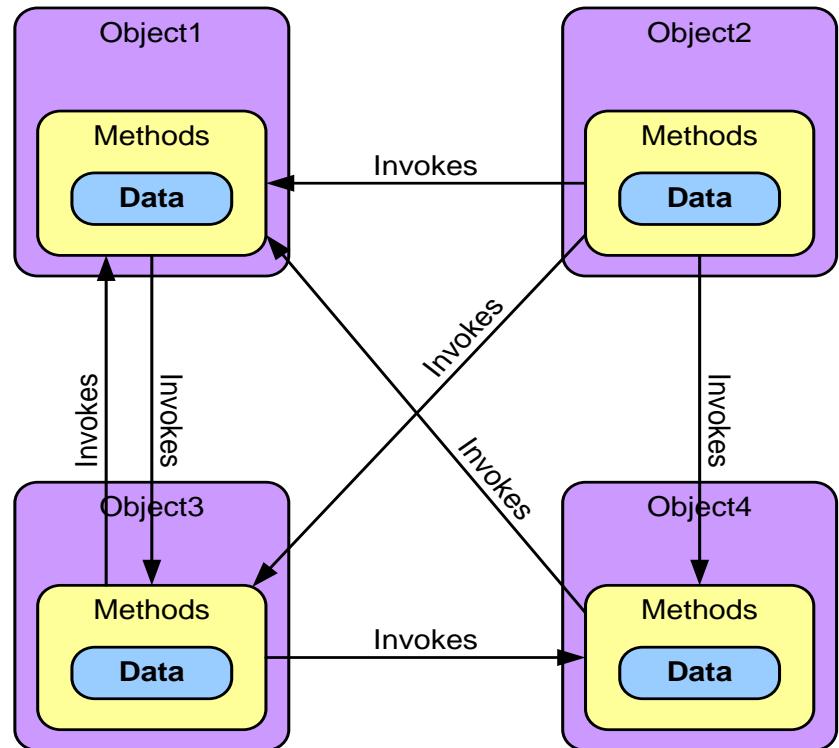
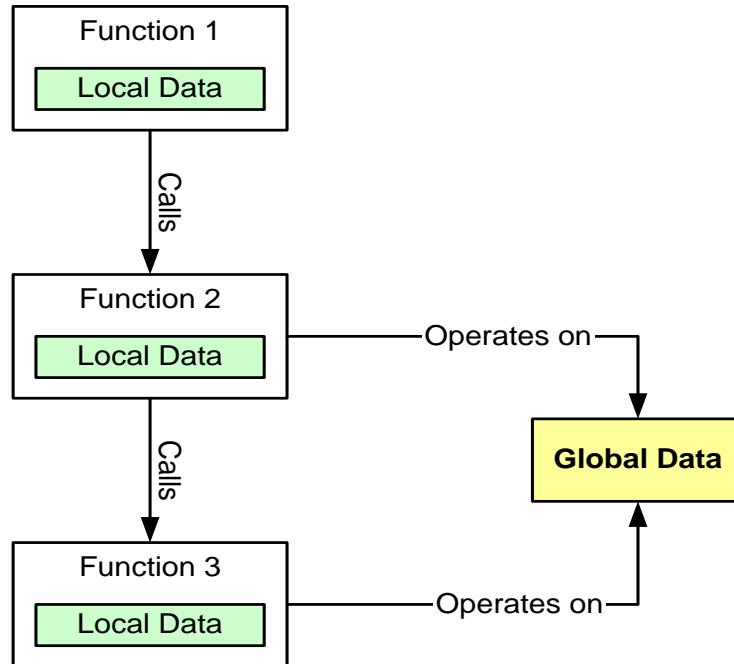
- Object-oriented programming takes the techniques used in structured programming further and combines routines and the data they use into classes.
- The data items stored for a class are called fields and the routines that operate on these fields are called methods.
- To use a class a programmer can declare instances of the class, which are called objects.
- Object-Oriented programming language are C#, C++, Java etc.

The Object-Oriented Approach -I

- Object oriented programming grew in the 70's as a solution to the problems of structured programming
- Models human thought process as closely as possible
- Deals with data and procedures that operate on data as a single 'object'

Procedural versus Object Oriented Programming

- In procedural programming, functions operate based on either local or global data
- In Object oriented programming, Objects exist and objects interact with other objects



Calls in Procedural Language

Message passing between Objects

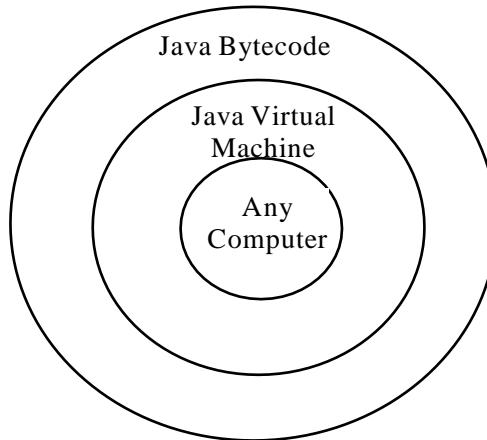
Java

- Java is a general-purpose computer-programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible.
- It is intended to let application developers "write once, run anywhere" (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.
- Java applications are typically compiled to byte code that can run on any Java virtual machine (JVM) regardless of computer architecture.
- Java is one of the most popular programming languages in use, particularly for client-server web applications, with a reported over 9 million developers.
- Java was originally developed by James Gosling at Sun Microsystems (which has since been acquired by Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform.
- The language derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them.

Source: [https://en.wikipedia.org/wiki/Java_\(programming_language\)](https://en.wikipedia.org/wiki/Java_(programming_language))

Compiling Java Source Code

You can port a source program to any machine with appropriate compilers. The source program must be recompiled, however, because the object program can only run on a specific machine. Nowadays computers are networked to work together. Java was designed to run object programs on any platform. With Java, you write the program once, and compile the source program into a special type of object code, known as *bytecode*. The bytecode can then run on any computer with a Java Virtual Machine, as shown in Figure Java Virtual Machine is a software that interprets Java bytecode.



The Object-Oriented Approach - II

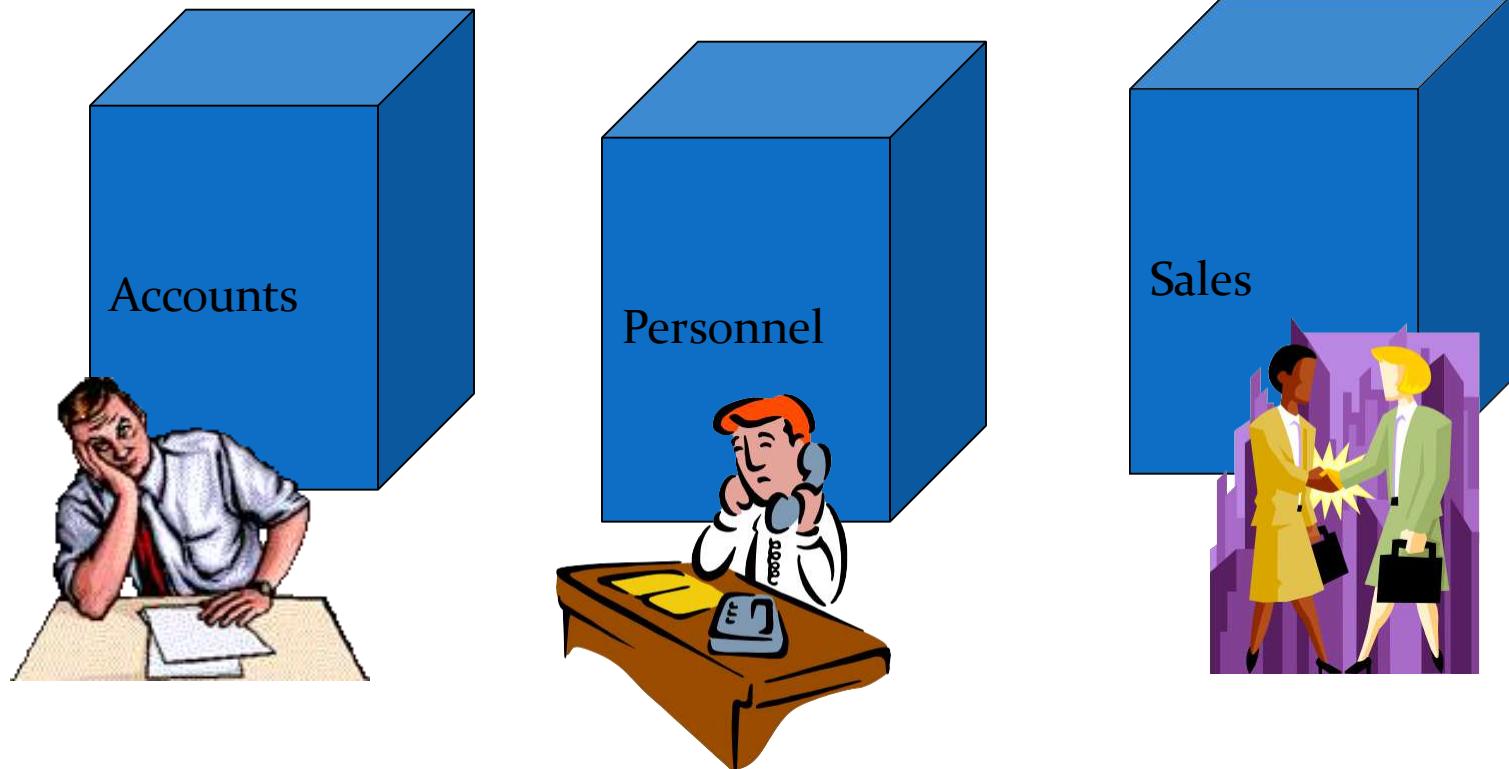
All around us in the realworld
are objects.



Each object has certain
characteristics and exhibits
certain behaviour



The Object-Oriented Approach - III



The real world around us is full of objects. We can consider both living beings as well as things as objects. For example, the different departments in a company are objects.

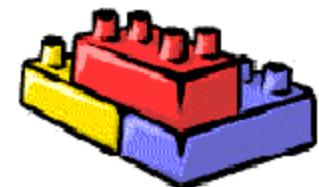
Drawbacks of Traditional Programming

The drawbacks of Traditional Programming are:

- Unmanageable programs
- Problems in modification of data
- Difficulty in implementation

Why do we care about objects?

- Modularity - large software projects can be split up in smaller pieces.
- Reuseability - Programs can be assembled from pre-written software components.
- Extensibility - New software components can be written or developed from existing ones.



Object – Oriented Programming

Here the application has to implement the entities as they are seen in real life and associate actions and attributes with each.

Data

Employee details

Salary statements

Bills

Vouchers

Receipts



Functions

Calculate salary

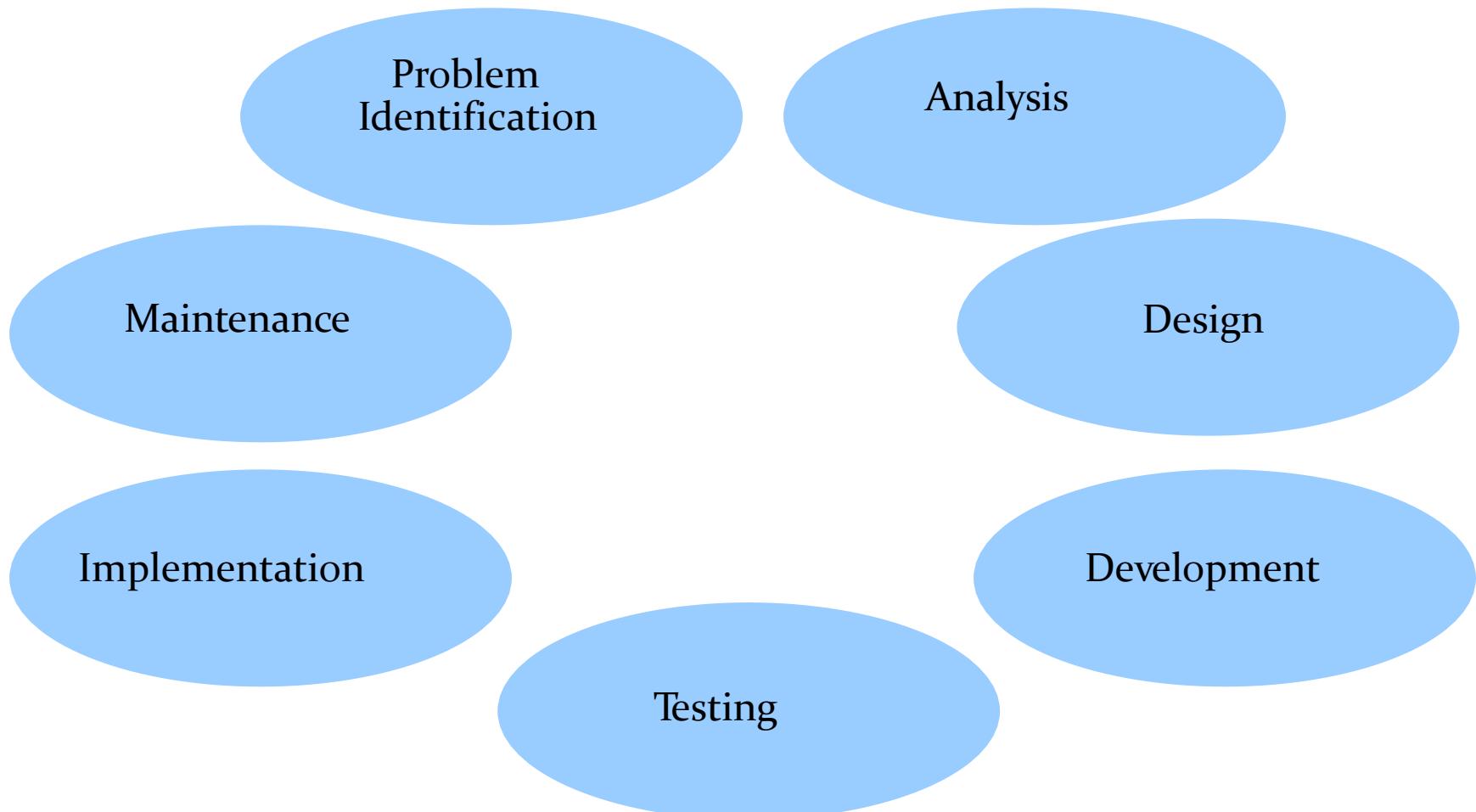
Pay salary

Pay bills

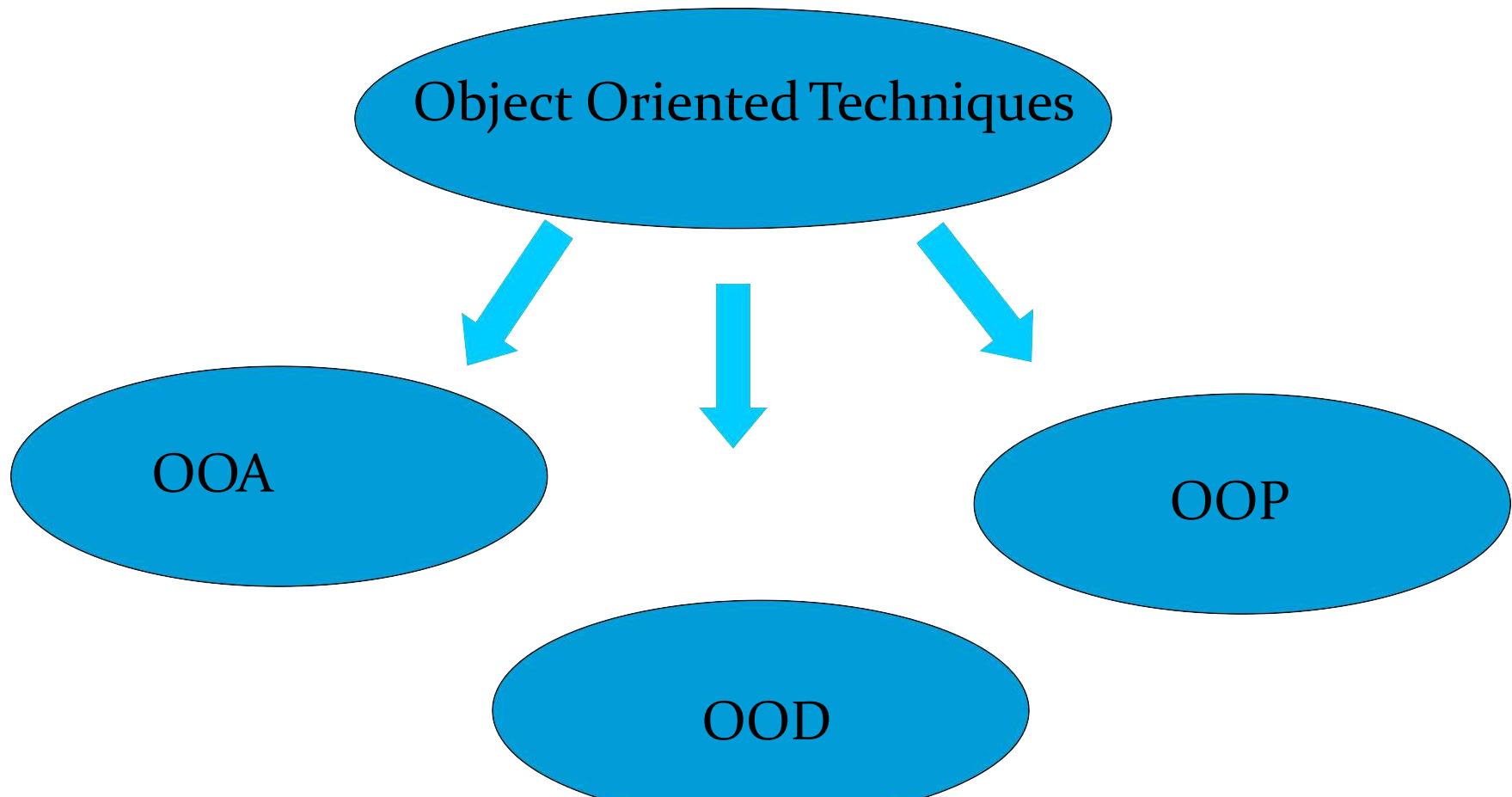
Tally accounts

Transact with banks

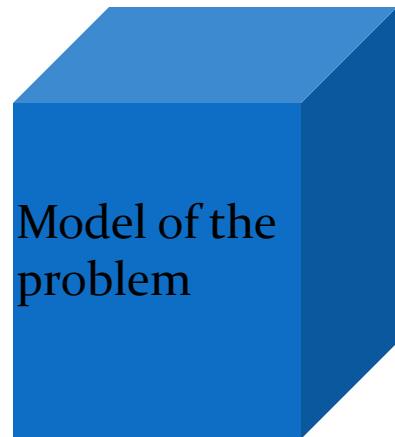
Object Oriented Approach



Object Oriented Techniques

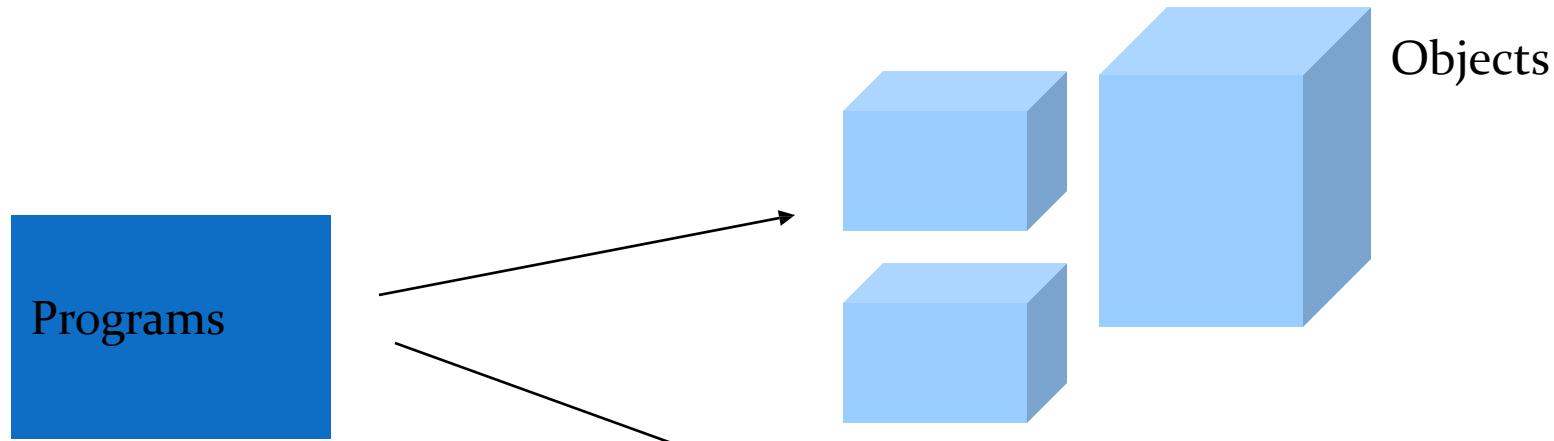


Object Oriented Analysis

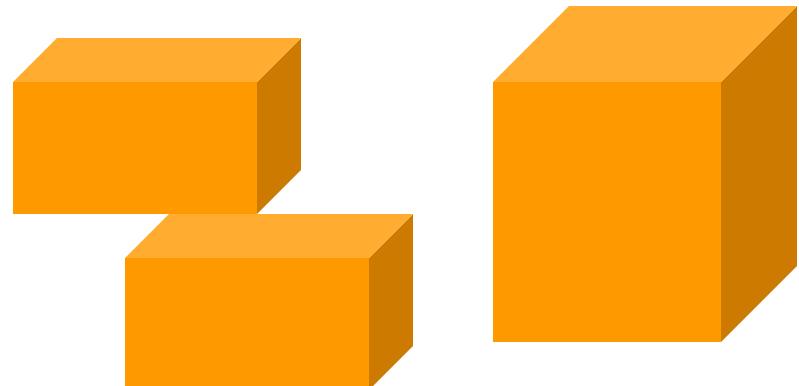


OOA is the phase if any project during which a precise and concise model of the problem in terms of real-world objects and concepts as understood by the user is developed

Object Oriented Design

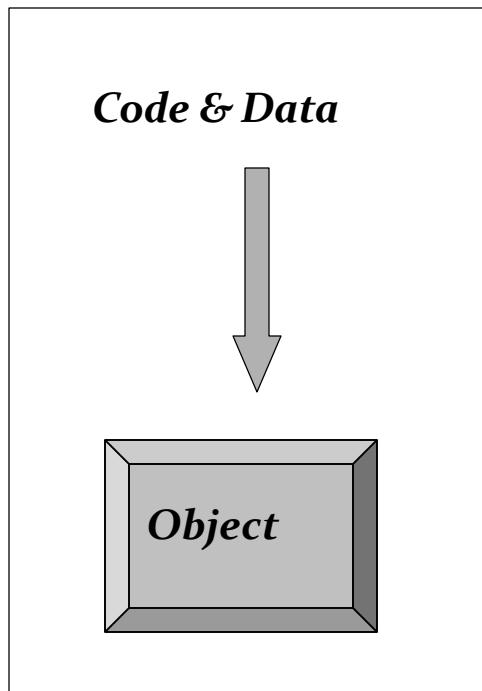


OOD is the phase in which programs are organized as cooperative collection of objects , each of which represents an instance of a class, and whose classes are all members of a hierarchy of classes united via inheritance relationship



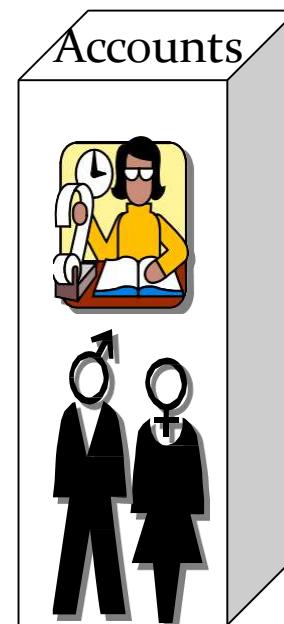
Object Oriented Programming

OOP (Object oriented Programming) is the construction phase of the life cycle that Object-Oriented Techniques follows



Data:

- *No. of employees*
- *Salary statements*
- *Bills*
- *Vouchers*
- *Receipts*
- *Petty cash records*
- *Banking data*



Functions:

- *Calculate salary*
- *Pay salary*
- *Pay bills*
- *Tally accounts*
- *Transact with banks*

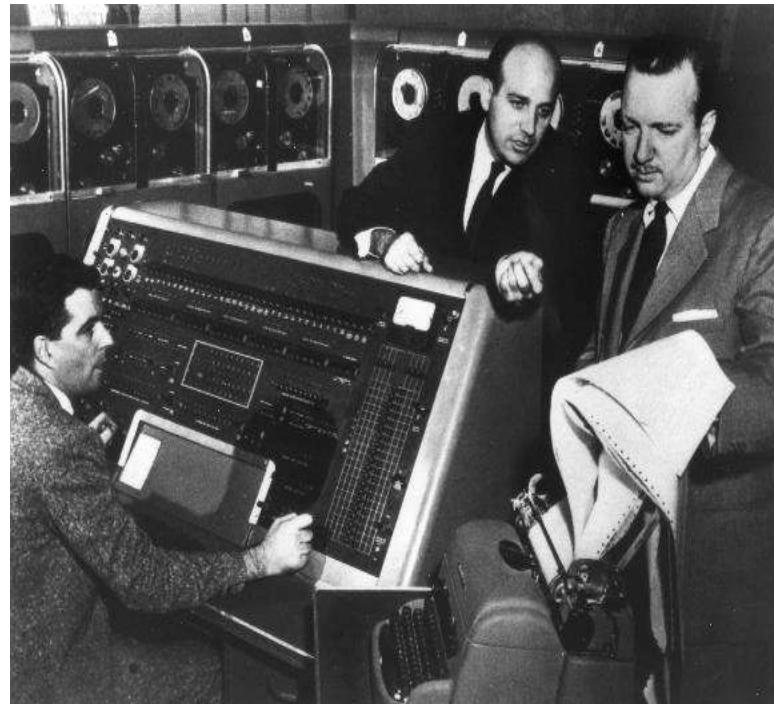
Basic Object-Oriented Concepts

- Object
 - Helps to understand the realworld
 - Provides a practical basis for computer applications
- Class
 - Describes a set of related objects
- Property
 - A characteristic of an object – also called *attribute*
- Method
 - An action performed by an object

Object-Oriented Programming

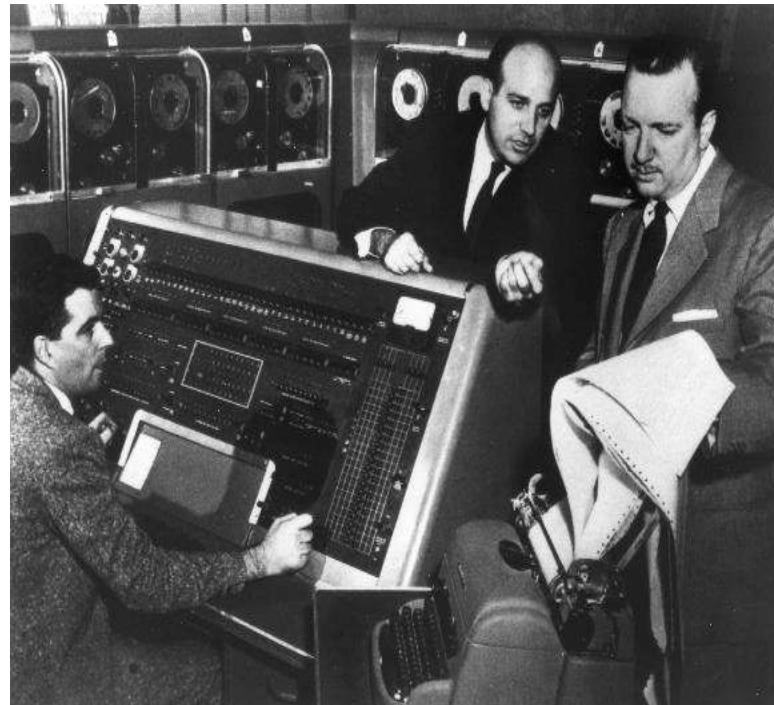
Early computers were far less complex than computers are today.

Their memories were smaller and their programs were much simpler.



Object-Oriented Programming

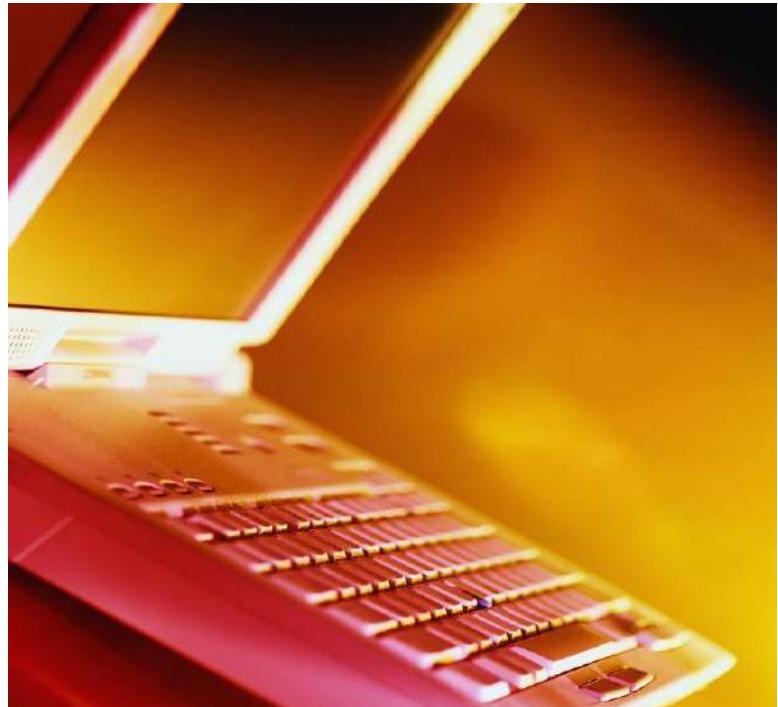
They usually executed only
one program at a time.



Object-Oriented Programming

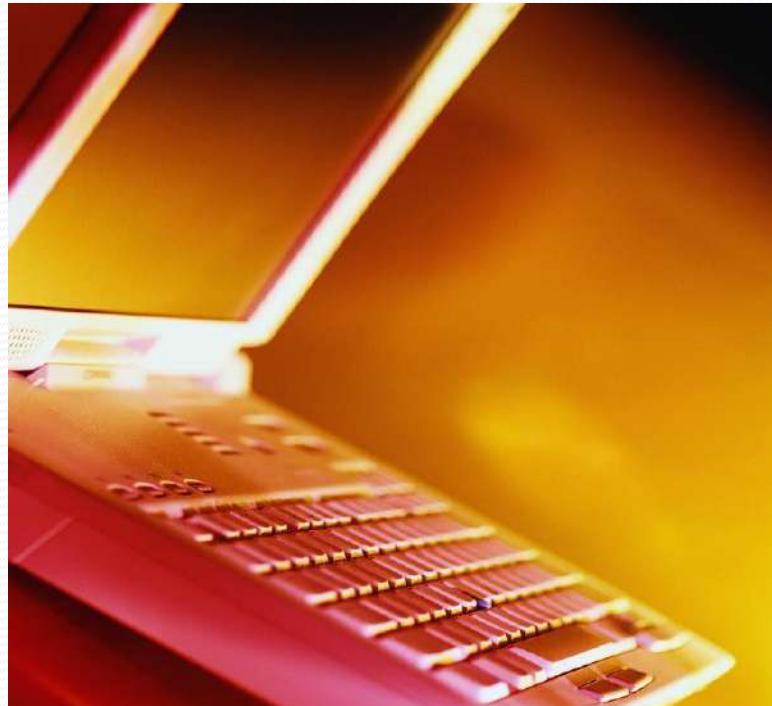
Modern computers are smaller, but far more complex than early computers.

They can execute many programs at the sametime.



Object-Oriented Programming

Computer scientists have introduced the notion of *objects* and *object-oriented programming* to help manage the growing complexity of modern computers.



Object-Oriented Programming

An *object* is anything that can be represented by data in a computer's memory and manipulated by a computer program.

Object-Oriented Programming

An object can be something in the physical world or even just an abstract idea.

A bank transaction is an example of an object that is not physical.

Date	Amount
10/20	\$ 738.97
10/21	526.82
10/22	580.53
10/23	524.21
10/24	362.24
10/27	308.42

Object-Oriented Programming

To a computer, an object is simply something that can be represented by data in the computer's memory and manipulated by computer programs.



Object-Oriented Programming

The data that represent the object are organized into a set of *properties*.

The values stored in an object's properties at any one time form the *state* of an object.

<u>Name:</u>	PA 3794
<u>Owner:</u>	Pakistan International Airline
<u>Location:</u>	39 52' 06" N 75 13' 52" W
<u>Heading:</u>	271°
<u>Altitude:</u>	19 m
<u>AirSpeed:</u>	0
<u>Make:</u>	Boeing
<u>Model:</u>	737
<u>Weight:</u>	32,820 kg

Fields – Declaration

- **Field Declaration**

- a type name followed by the field name, and optionally an initialization clause
- primitive data type vs. Objectreference
 - boolean, char, byte, short, int, long, float, double
- field declarations can be preceded by different modifiers
 - access control modifiers
 - static
 - Final [const in c++/c#]

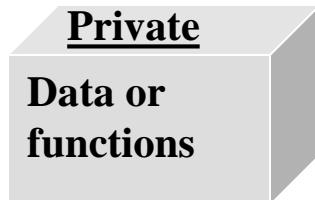
More about field modifiers (1)

- Access control modifiers
 - *private*: private members are accessible only in the class itself
 - *package*: package members are accessible in classes in the same package and the class itself
 - *protected*: protected members are accessible in classes in the same package, in subclasses of the class, and in the class itself
 - *public*: public members are accessible anywhere the class is accessible

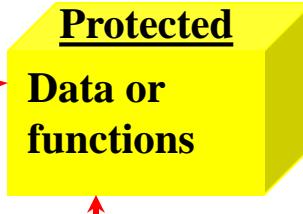
Private, Protected and Public (2)

Class

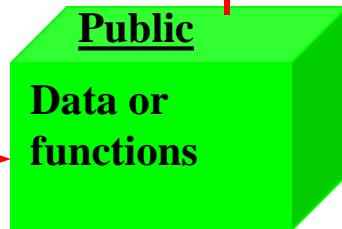
Not accessible from outside class



Accessible from member functions of a derived class.



Accessible from outside class

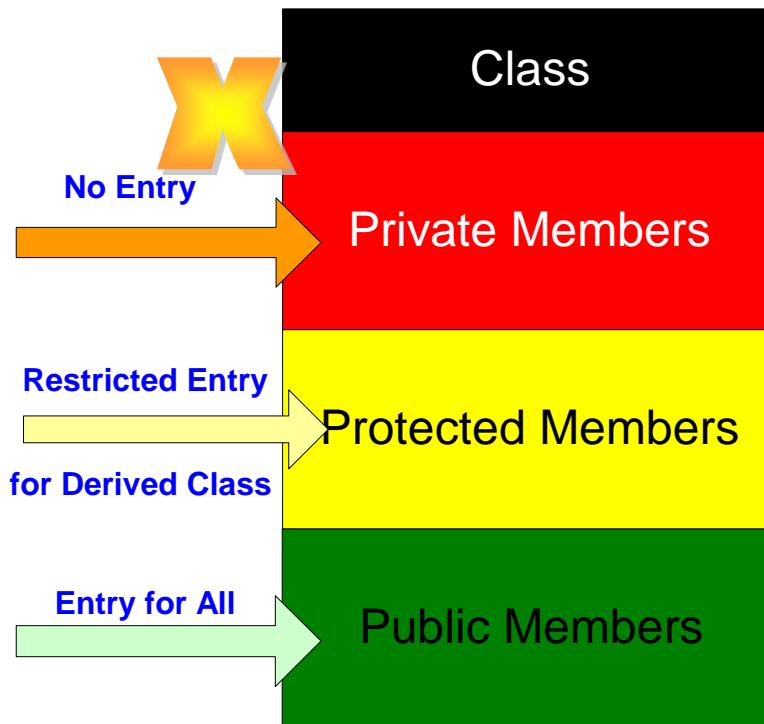


Private data is not available outside the class

Private data of other classes is hidden and not available within the functions of this class.

Class – Implementation of Encapsulation (Access Specifiers)

- Public Members
 - Accessed within and outside the class
- Protected Members
 - Accessed within and only by derived classes
- Private Members
 - Accessed only within the class
- Good design practice is to keep data members under private and methods under public
- Methods invoked ONLY by other methods of the same class are kept private
- Such methods are called **helper** methods



Pencil.java

```
public class Pencil {  
    public String color = "red";  
    public int length;  
    public float diameter;  
    private float price;  
  
    public static long nextID = 0;  
  
    public void setPrice (float newPrice) {  
        price = newPrice;  
    }  
}
```

CreatePencil.java

```
public class CreatePencil {  
    public static void main (String args[]) {  
        Pencil p1 = new Pencil();  
        p1.price = 0.5f;  
    }  
}
```

```
%> javac Pencil.java  
%> javac CreatePencil.java  
CreatePencil.java:4: price has private access in Pencil  
    p1.price = 0.5f;
```

More about field modifiers

- static

- only one copy of the static field exists, shared by all objects of this class
- can be accessed directly in the class itself
- access from outside the class must be preceded by the class name as follows

```
System.out.println(Pencil.nextID);
```

or via an object belonging to the class

- from outside the class, non-static fields must be accessed through an object reference

Class Pencil

```
{
```

```
    static int nextID;
```

```
}
```

```
public class CreatePencil {  
    public static void main (String args[]){  
        Pencil p1 = new Pencil();  
        Pencil.nextID++;  
        System.out.println(p1.nextID);  
        //Result? 1  
  
        Pencil p2 = new Pencil();  
        Pencil.nextID++;  
        System.out.println(p2.nextID);  
        //Result?  
  
        System.out.println(p1.nextID);  
        //Result? 2  
    }  
}  
still 2!
```

Note: this code is only for the purpose of showing the usage of static fields. It has POOR design!

More about field modifiers (3)

- **final**
 - once initialized, the value cannot be changed
 - often be used to define named constants
 - static final fields must be initialized when the class is initialized
 - non-static final fields must be initialized when an object of the class is constructed

Object-Oriented Programming

Computer programs implement algorithms that manipulate the data.

In object-oriented programming, the programs that manipulate the properties of an object are the object's **methods**.

```
class Bicycle {  
  
    int cadence = 0;  
    int speed = 0;  
    int gear = 1;  
  
    void changeCadence(int newValue) {  
        cadence = newValue;  
    }  
  
    void changeGear(int newValue) {  
        gear = newValue;  
    }  
  
    void speedUp(int increment) {  
        speed = speed + increment;  
    }  
  
    void applyBrakes(int decrement) {  
        speed = speed - decrement;  
    }  
}
```

Object-Oriented Programming

We can think of an object as a collection of properties and the methods that are used to manipulate those properties.

Properties

Methods

```
class Bicycle {  
    int cadence = 0;  
    int speed = 0;  
    int gear = 1;  
  
    void changeCadence(int newValue) {  
        cadence = newValue;  
    }  
  
    void changeGear(int newValue) {  
        gear = newValue;  
    }  
  
    void speedUp(int increment) {  
        speed = speed + increment;  
    }  
  
    void applyBrakes(int decrement) {  
        speed = speed - decrement;  
    }  
}
```

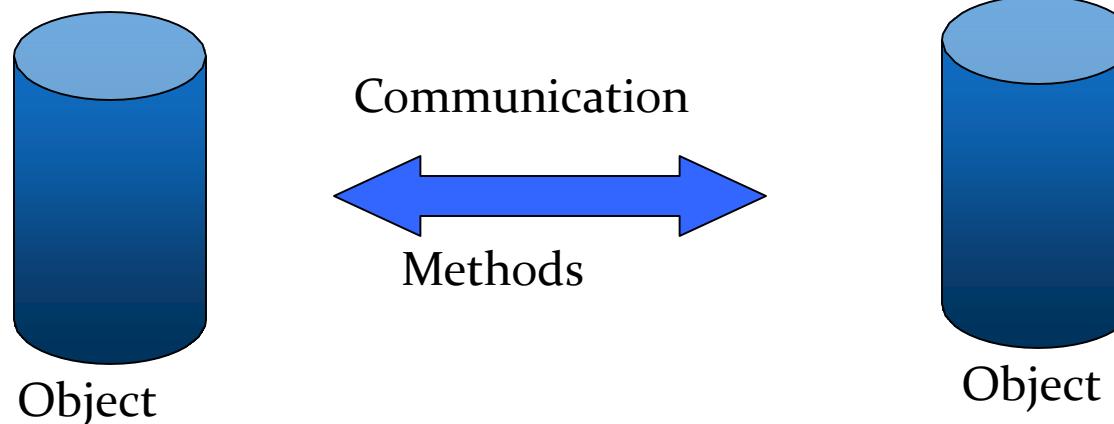
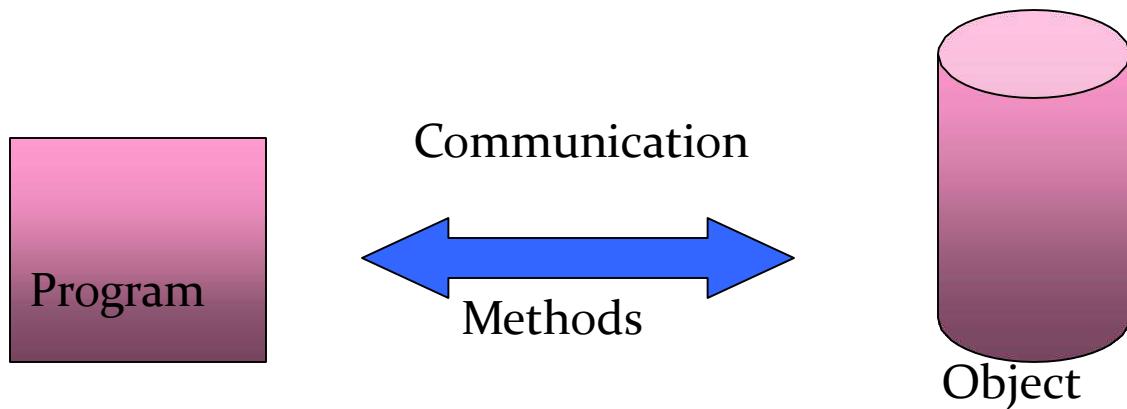
Methods – Declaration

- Method declaration: two parts
 1. method header
 - consists of modifiers (optional), return type, method name, parameter list and a throws clause (optional)
 - types of modifiers
 - *access control modifiers*
 - *abstract*
 - the method body is empty. E.g.

```
abstract void sampleMethod();
```
 - *static*
 - represent the whole class, no a specific object
 - can only access static fields and other static methods of the same class
 - 2. method body

```
abstract class SDate{  
    int month, day, year;  
    void setDate(int m,int d,int y){  
        month=m;  
        day=d;  
        year=y;  
    }  
    abstract void abc();  
}  
class yDate extends SDate{  
    void abc(){  
        System.out.println("My Class");  
    }  
}  
public class MyClass{  
    public static void main(String[] args) {  
        yDate y = new yDate();  
        y.abc();  
    }  
}
```

Calling Methods



Methods – Invocation

- Method invocations
 - invoked as operations on objects/classes using the dot (.) operator
`reference.method(arguments)`
 - static method:
 - Outside of the class: “reference” can either be the class name or an object reference belonging to the class
 - Inside the class: “reference” can be omitted
 - non-static method:
 - “reference” must be an object reference

Calling methods

A method is always called to act on a specific object, not on the class in general

Example:

S1.setdate(27,1,1969)

The general syntax for accessing a member function of a class is

Syntax:

Class.object.method()

Returning object from a method

A return statement in a function is considered to initialize a variable of the returned type

Syntax:

Test testobject = S1.func()

```
test func ()  
{  
    test temp_object;  
    .  
    .  
    return temp_object;  
}
```

Accessor Functions

- Usually, the data member are defined in private part of a class – information hiding
- Accessor functions are functions that are used to access these private datamembers
- Accessor functions also useful in reducing error

Example – Accessing Data Member

```
class Student{  
    ...  
    private int semester;  
    public void setSem(int aSem) {  
        if((aSem<1) || (aSem>8))  
            System.out.println("Invalid Semester");  
        else  
            semester = aSem;  
    }  
    public int getCurrentSem()  
    {  
        return semester;  
    }  
}
```

Object-Oriented Programming

A *class* is a group of objects with the same properties and the same methods.

```
class Bicycle {  
  
    int cadence = 0;  
    int speed = 0;  
    int gear = 1;  
  
    void changeCadence(int newValue) {  
        cadence = newValue;  
    }  
  
    void changeGear(int newValue) {  
        gear = newValue;  
    }  
  
    void speedUp(int increment) {  
        speed = speed + increment;  
    }  
  
    void applyBrakes(int decrement) {  
        speed = speed - decrement;  
    }  
}
```

Object-Oriented Programming

Each copy of an object from a particular class is called an *instance* of the object.



Object-Oriented Programming

The act of creating a new instance of an object is called **instantiation**.



Object-Oriented Programming

A class can be thought of
as a blueprint for
instances of an object.



Object-Oriented Programming

Two different instances of the same class will have the same properties, but different values stored in those properties.



Using the Class

```
class Sdate {  
    int month, day, year;  
    void setDate(int m,int d,int y)  
    {  
        month=m;  
        day=d;  
        year=y;  
    }  
    public static void main(String args[])  
    {  
        Sdate S1,S2;  
        S1=new Sdate();  
        S2=new Sdate();  
        S1.setDate(11,27,1967);  
        S2.setDate(4,3,1969);  
    }  
}
```

Defining Objects

Statement	Effect
Sdate S1;	<p>A rectangular box labeled "Null" above a smaller box labeled "S1". An arrow points from the "S1" box to the "Null" box.</p>
S1 = new Sdate();	<p>A rectangular box labeled "S1" above a larger vertical stack of three boxes. The top box is labeled "Month (11)", the middle box is labeled "Day (27)", and the bottom box is labeled "Year (1967)". An arrow points from the "S1" box to the top box.</p>
Sdate S2 = new Sdate();	<p>A rectangular box labeled "S2" above a larger vertical stack of three boxes. The top box is labeled "Month (4)", the middle box is labeled "Day (3)", and the bottom box is labeled "Year (1969)". An arrow points from the "S2" box to the top box.</p>

Object Reference

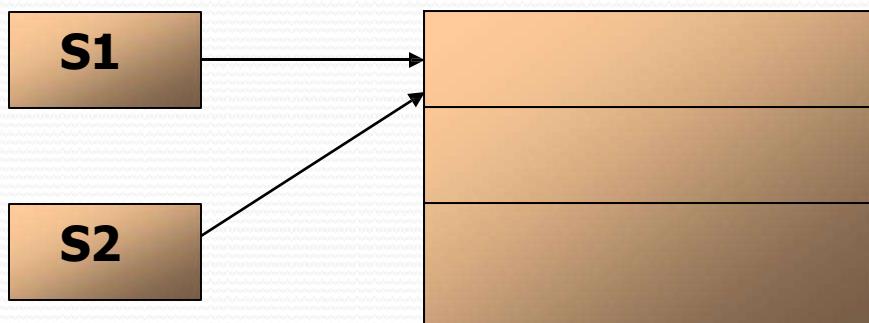
New
operator

create
→



Points to
→

Physical
location of
data



```
class Sdate{  
.....  
.....  
Sdate S1 = new Sdate();  
Sdate S2 = S1;  
.....  
}
```

Class

- A Class defines an entity in terms of common characteristics and actions

Class Customer
Name of the customer
Address of the customer
Model of the car bought
Salesman's name who sold the car
Accept Name
Accept Address
Accept Model of the car purchased
Accept the name of the salesman who sold the car
Generate the bill

Messages

- Objects communicate through messages
- They send messages (stimuli) by invoking appropriate operations on the target object
- The number and kind of messages that can be sent to an object depends upon its interface

```
S1.SetDate(2025,02,03);
```

Examples – Messages

- A Person sends message (stimulus) “stop” to a Car by applying brakes
- A Person sends message “place call” to a Phone by pressing appropriate button

Object

- Attribute
 - Characteristic that describes an object
- Operation
 - Service that can be requested of an object
- Method
 - Specification of how the requested operation is carried out
- Message
 - Request for an operation
- Event
 - Stimulus sent from one object to another

Class vs. Object

- Class defines an entity, while an object is the actual entity
- Class is a conceptual model that defines all the characteristics and actions required of an object, while an object is a real model
- Class is a prototype of an object
- All objects belonging to the same class have the same characteristics and actions

Class and Objects – Example

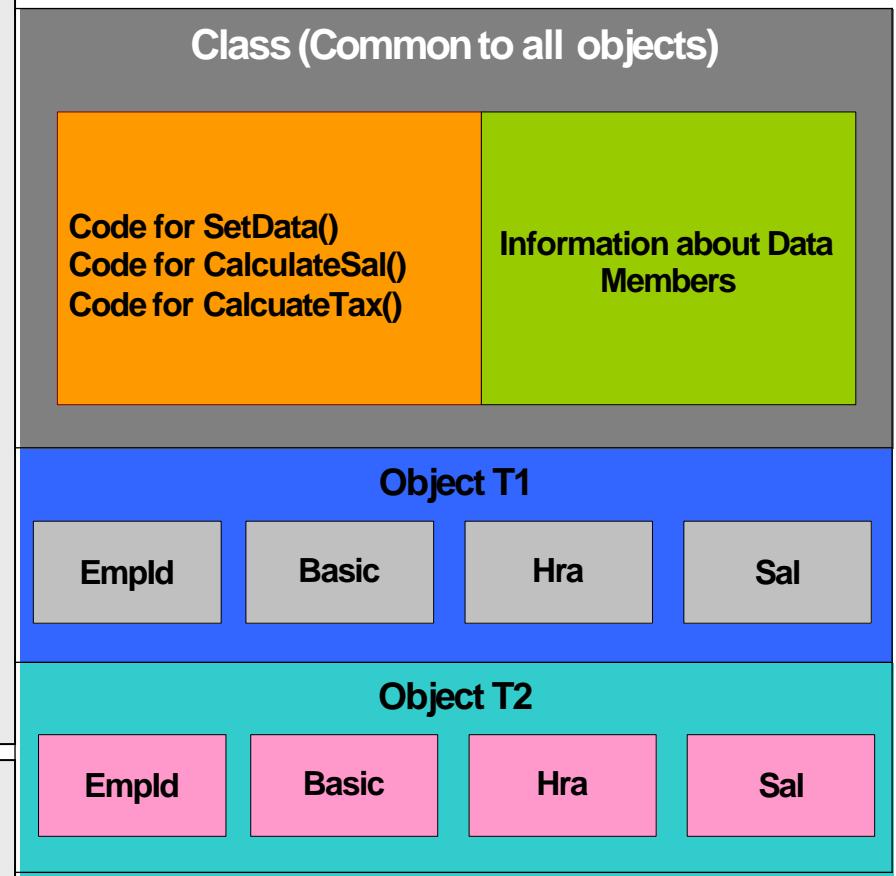
```
class Trainee {  
    private int empId;  
    private String empName;  
    private float basic;  
    private float hra;  
  
    public void SetData(int iEmpId,  
                        String acEmpName, float fBasic,  
                        float fHRA) {  
        ..... }  
  
    public void CalculateSal() {  
        ..... }  
  
    public void CalculateTax() {  
        ..... }  
}  
//class Trainee ends here
```

```
public static void main(String [] args){  
    /* Object Creation */  
    Trainee oT1 = new Trainee();  
  
    /* Invoking SetData (Message)*/  
    oT1.SetData(101,"Hamza",1200,150)  
  
    /* Invoking CalculateSal (Message)*/  
    oT1.CalculateSal();  
  
    /* Invoking CalculateTax */  
    oT1.CalculateTax();  
}
```

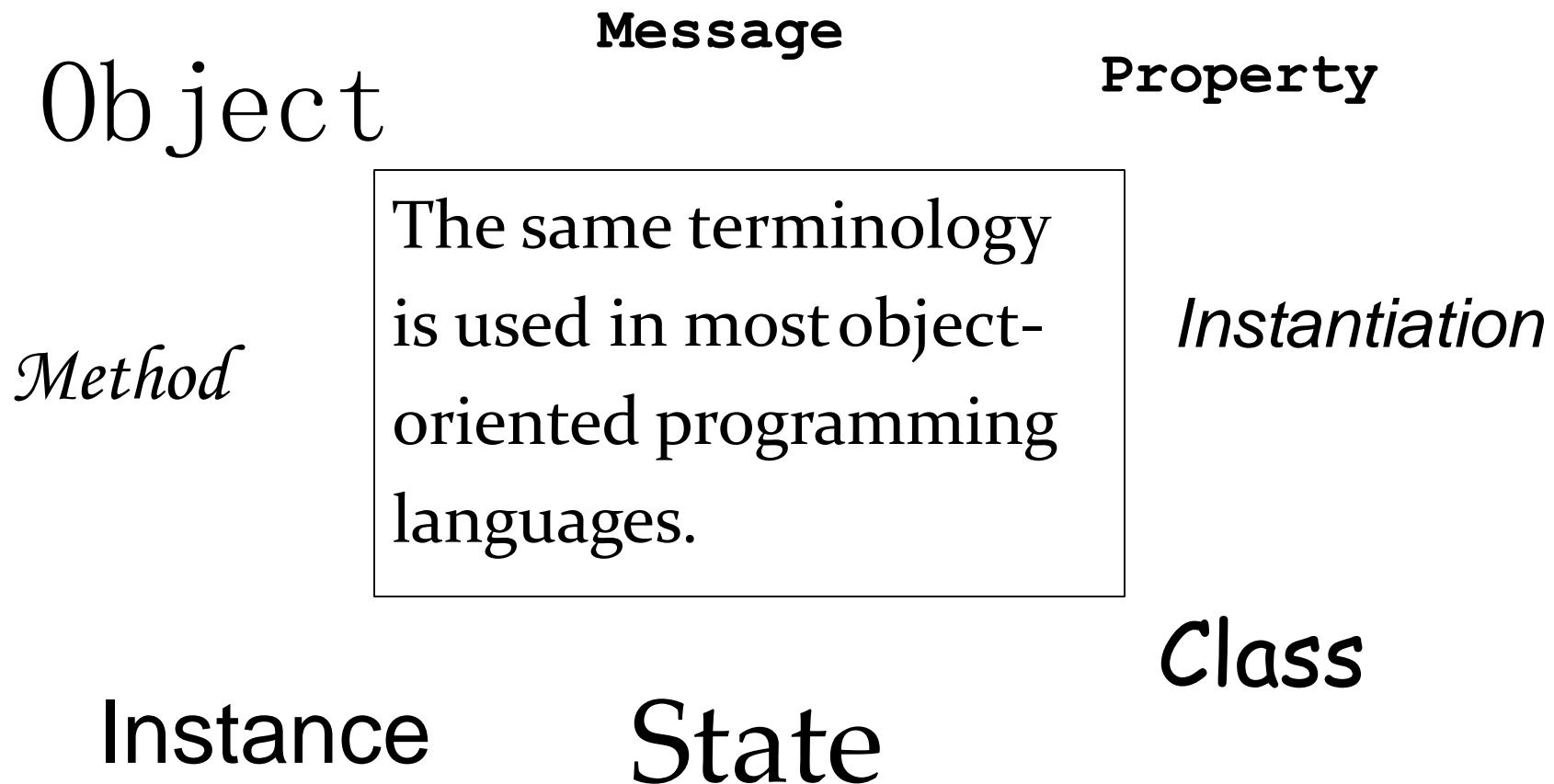
Memory allocation for Classes and Objects

```
class Trainee {  
    private int m_iEmpId;  
    private float m_fBasic;  
    private float m_fHRA;  
    private float m_fSalary;  
  
    public void SetData(int iEmpId, float fBasic,  
                        float fHRA){  
    }  
    public void CalculateSal() {  
        // code goes here  
    }  
    public void CalculateTax() {  
        //code goes here  
    }  
}
```

```
Trainee oT1 = new Trainee();  
Trainee oT2= new Trainee();
```



Object-Oriented Programming



Reading input in JAVA

A Simple Java Program

```
//This program prints Welcome to Java!
public class Welcome {
    public static void main(String[] args) {
        System.out.println("Welcome to Java!");
    }
}
```

Here **System** is the class available in **java.lang** package and **out** is an object of **PrintStream** class, while **println()** method.

READING INPUT FROM THE `input`

- The standard input device is normally the computer input.
- The Java API has an object `System.in` which is associated with the standard input device.

READING INPUT FROM THE `input`

The *Scanner* Class

- We will use the `System.in` object in conjunction with the `Scanner` class to read input data from the input.
- The `Scanner` class has methods that can be used to read input and format it as either a value of a primitive data type or a `String`.

READING INPUT FROM THE `input`

The *Scanner* Class

- To use the *Scanner* class in our program we must put the following statement near the top of our file, **before any class definition:**

```
import java.util.Scanner;
```

This statement tells the compiler where to find the *Scanner* class in the Java API.

READING INPUT FROM THE `input`

The *Scanner* Class

- You must also create a *Scanner* object and connect it to the *System.in* object. You can do this with a statement like the following:

```
Scanner input= new Scanner(System.in);
```

Create this object inside your *main* method before you attempt to read anything from the input.

READING INPUT FROM THE *input*

The *Scanner* Class

- The words *Scanner* *input* declare a variable named *input* of type *Scanner*. This variable will reference an object of the *Scanner* class.

Scanner **input**= new Scanner(System.in);

You could have chosen any name you wanted for the variable, but *input* is a good one since you are going to use it to access the keyboard i.e. an input device.

READING INPUT FROM THE `input`

The *Scanner* Class

`Scanner input = new Scanner(System.in);`

- The `new` key word is used to create an object in memory.
- In the statement above we are creating an object of the *Scanner* class.
- Inside the parentheses, we have `System.in`. Here we are saying that we want the object we are creating to be connected with the `System.in` object, which again is associated with the keyboard.
- We are assigning the address of the object created using the `new` operator to our variable named *input*, so *input* now references the object we have linked with the actual keyboard.

READING INPUT FROM THE `input`

The *Scanner* Class

- Every object created from the *Scanner* class has methods that read a string of characters entered at the input, convert them to a specified type, and return the converted value. This value can be stored in a variable of compatible type.

READING INPUT FROM THE `input`

The *Scanner* Class

For example, the code below could be used to read an integer entered at the input and store it in an integer variable named `age`.

```
int age;
```

```
System.out.print("Enter your age: ");
age = input.nextInt();
```

- The `nextInt()` method formats the characters entered by the user as an `int` and returns the integer value.
- The integer value is assigned to the variable named `age`.

Method	Example and Description
nextByte	<p>Example Usage:</p> <pre>byte x; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a byte value: "); x = keyboard.nextByte();</pre> <p>Description: Returns input as a byte.</p>
nextDouble	<p>Example Usage:</p> <pre>double number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a double value: "); number = keyboard.nextDouble();</pre> <p>Description: Returns input as a double.</p>
nextFloat	<p>Example Usage:</p> <pre>float number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a float value: "); number = keyboard.nextFloat();</pre> <p>Description: Returns input as a float.</p>
nextInt	<p>Example Usage:</p> <pre>int number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter an integer value: "); number = keyboard.nextInt();</pre> <p>Description: Returns input as an int.</p>
nextLine	<p>Example Usage:</p> <pre>String name; Scanner keyboard = new Scanner(System.in); System.out.print("Enter your name: "); name = keyboard.nextLine();</pre> <p>Description: Returns input as a String.</p>
nextLong	<p>Example Usage:</p> <pre>long number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a long value: "); number = keyboard.nextLong();</pre> <p>Description: Returns input as a long.</p>
nextShort	<p>Example Usage:</p> <pre>short number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a short value: "); number = keyboard.nextShort();</pre> <p>Description: Returns input as a short.</p>

READING INPUT FROM THE `input`

The *Scanner* Class

- We can use the `nextLine` method of a `Scanner` object to read a string of characters entered at the keyboard.

Example:

To get the user's first name we could write:

```
String firstName;
```

```
System.out.print("Enter your first name: ");
firstName = input.nextLine( );
```

READING INPUT FROM THE `input`

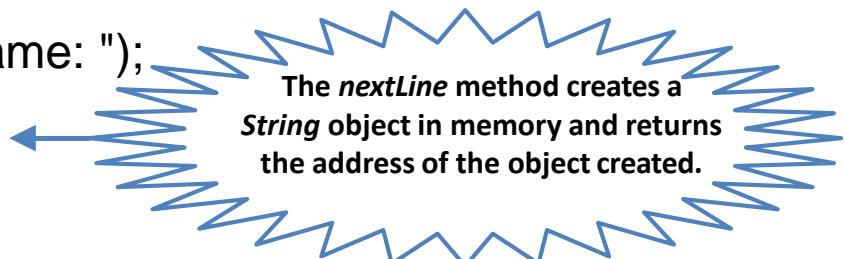
The *Scanner* Class

- The `nextLine` method creates a `String` object in memory that contains the sequence of characters entered at the keyboard before the **Enter** key is pressed and returns the address of this object.

Below we are assigning the address of the object created by the `nextLine` method to the `String` reference variable named `firstName`.

```
String firstName;
```

```
System.out.print("Enter your first name: ");  
firstName = input.nextLine( );
```



READING INPUT FROM THE `input`

The *Scanner* Class

- The *Scanner* class does not have a method for reading a single character.
- In the text, they suggest using the *Scanner* classes *nextLine* method to read the character as a string, and then using the *String* classes *charAt* method to extract the first character from the string. Remember, the first character is at index 0.

READING A SINGLE CHARACTER ENTERED AT THE input

Example:

```
String stringInitial;  
char initial;
```

```
System.out.print("Enter your middle initial " );  
stringInitial = input.nextLine( );  
initial = stringInitial.charAt(0);
```

READING Data from File

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ScannerFileExample {
    public static void main(String[] args) {
        File file = new File("example.txt");

        try {
            Scanner scanner = new Scanner(file);

            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                System.out.println(line);
            }

            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        }
    }
}
```

Java, Web, and Beyond

- Java can be used to develop Web applications.
- Java Applets
- Java Servlets and JavaServer Pages
- Java can also be used to develop applications for hand-held devices such as Palm and cell phones

Characteristics of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Characteristics of Java

- Java Is Simple
 - Java Is Object-Oriented
 - Java Is Distributed
 - Java Is Interpreted
 - Java Is Robust
 - Java Is Secure
 - Java Is Architecture-Neutral
 - Java Is Portable
 - Java's Performance
 - Java Is Multithreaded
 - Java Is Dynamic
- Java is partially modeled on C++, but greatly simplified and improved. Some people refer to Java as "C++--" because it is like C++ but with more functionality and fewer negative aspects.

Characteristics of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Java is inherently object-oriented. Although many object-oriented languages began strictly as procedural languages, Java was designed from the start to be object-oriented. Object-oriented programming (OOP) is a popular programming approach that is replacing traditional procedural programming techniques.

One of the central issues in software development is how to reuse code. Object-oriented programming provides great flexibility, modularity, clarity, and reusability through encapsulation, inheritance, and polymorphism.

Characteristics of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Distributed computing involves several computers working together on a network. Java is designed to make distributed computing easy. Since networking capability is inherently integrated into Java, writing network programs is like sending and receiving data to and from a file.

Characteristics of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

You need an interpreter to run Java programs. The programs are compiled into the Java Virtual Machine code called bytecode. The bytecode is machine-independent and can run on any machine that has a Java interpreter, which is part of the Java Virtual Machine (JVM).

Characteristics of Java

- Java Is Simple
 - Java Is Object-Oriented
 - Java Is Distributed
 - Java Is Interpreted
 - Java Is Robust
 - Java Is Secure
 - Java Is Architecture-Neutral
 - Java Is Portable
 - Java's Performance
 - Java Is Multithreaded
 - Java Is Dynamic
- Java compilers can detect many problems that would first show up at execution time in other languages.
- Java has eliminated certain types of error-prone programming constructs found in other languages.
- Java has a runtime exception-handling feature to provide programming support for robustness.

Characteristics of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
 - Java implements several security mechanisms to protect your system against harm caused by stray programs.
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Characteristics of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral Write once, run anywhere
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

With a Java Virtual Machine (JVM), you can write one program that will run on any platform.

Characteristics of Java

- Java Is Simple
 - Java Is Object-Oriented
 - Java Is Distributed
 - Java Is Interpreted
 - Java Is Robust
 - Java Is Secure
 - Java Is Architecture-Neutral
 - Java Is Portable
 - Java's Performance
 - Java Is Multithreaded
 - Java Is Dynamic
- Because Java is architecture neutral, Java programs are portable. They can be run on any platform without being recompiled.

Characteristics of Java

- Java Is Simple
 - Java Is Object-Oriented
 - Java Is Distributed
 - Java Is Interpreted
 - Java Is Robust
 - Java Is Secure
 - Java Is Architecture-Neutral
 - Java Is Portable
 - Java's Performance
 - Java Is Multithreaded
 - Java Is Dynamic
- Java's performance Because Java is architecture neutral, Java programs are portable. They can be run on any platform without being recompiled.

Characteristics of Java

- Java Is Simple
 - Java Is Object-Oriented
 - Java Is Distributed
 - Java Is Interpreted
 - Java Is Robust
 - Java Is Secure
 - Java Is Architecture-Neutral
 - Java Is Portable
 - Java's Performance
 - Java Is Multithreaded
 - Java Is Dynamic
- Multithread programming is smoothly integrated in Java, whereas in other languages you have to call procedures specific to the operating system to enable multithreading.

Characteristics of Java

- Java Is Simple
 - Java Is Object-Oriented
 - Java Is Distributed
 - Java Is Interpreted
 - Java Is Robust
 - Java Is Secure
 - Java Is Architecture-Neutral
 - Java Is Portable
 - Java's Performance
 - Java Is Multithreaded
 - Java Is Dynamic
- Java was designed to adapt to an evolving environment. New code can be loaded on the fly without recompilation. There is no need for developers to create, and for users to install, major new software versions. New features can be incorporated transparently as needed.

JDK/J2SE Versions

Version	Date
JDK Beta	1995
JDK 1.0	January 23, 1996 [40]
JDK 1.1	February 19, 1997
J2SE 1.2	December 8, 1998
J2SE 1.3	May 8, 2000
J2SE 1.4	February 6, 2002
J2SE 5.0	September 30, 2004
Java SE 6	December 11, 2006
Java SE 7	July 28, 2011
Java SE 8 (LTS)	March 18, 2014
Java SE 9	September 21, 2017
Java SE 10	March 20, 2018
Java SE 11 (LTS)	September 25, 2018 [41]
Java SE 12	March 19, 2019
Java SE 13	September 17, 2019
Java SE 14	March 17, 2020
Java SE 15	September 15, 2020 [42]
Java SE 16	March 16, 2021
Java SE 17 (LTS)	September 14, 2021
Java SE 18	March 2022

JDK Editions

- **Java Standard Edition (J2SE)**
 - J2SE can be used to develop client-side standalone applications or applets.
- **Java Enterprise Edition (J2EE)**
 - J2EE can be used to develop server-side applications such as Java servlets and Java ServerPages.
- **Java Micro Edition (J2ME)**
 - J2ME can be used to develop applications for mobile and embedded devices such as cell phones.

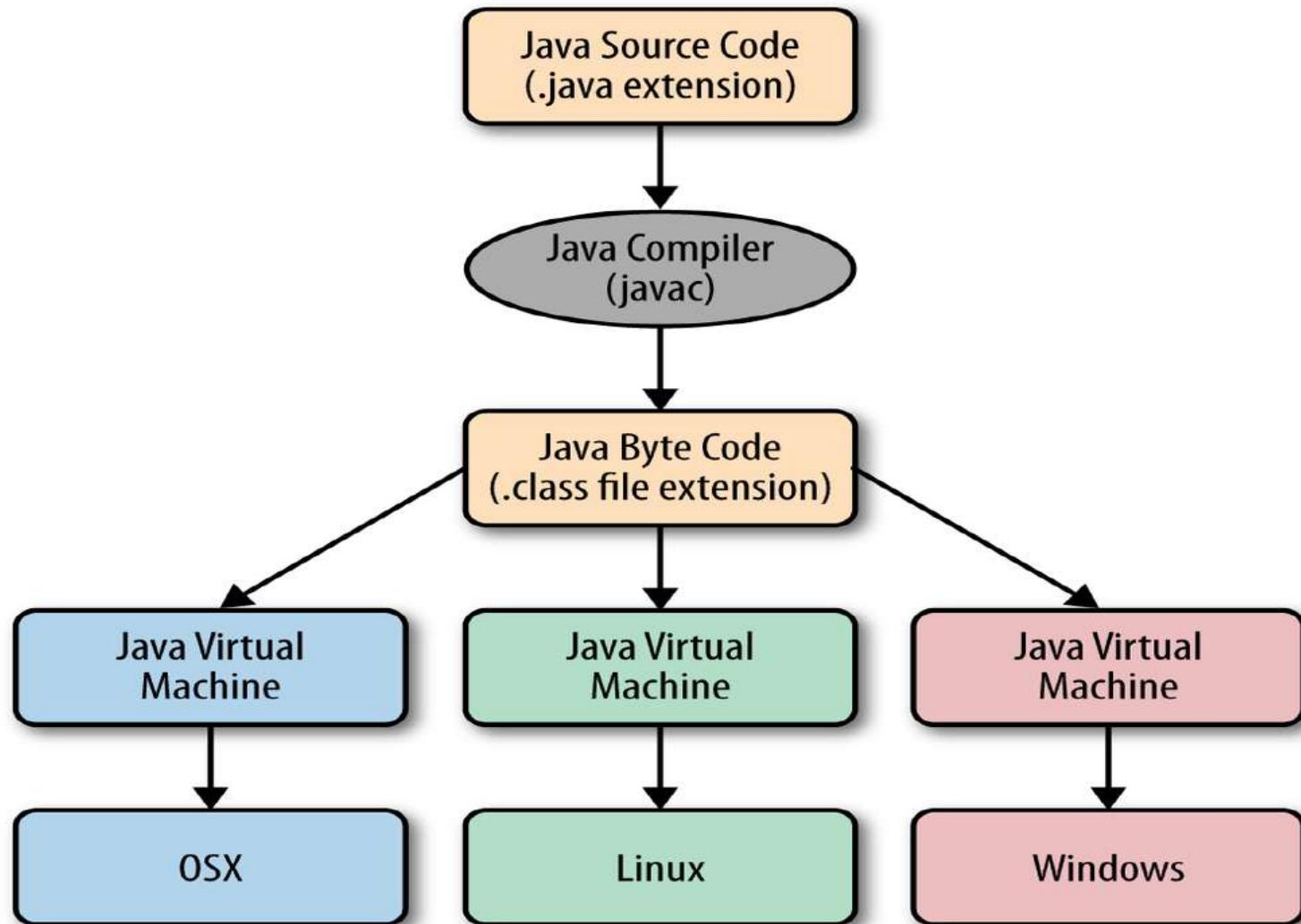
Java IDE Tools

- Borland JBuilder
- NetBeans Open Source by Sun
- Sun ONE Studio by Sun MicroSystems
- Eclipse Open Source by IBM

A Simple Java Program

```
//This program prints Welcome to Java!
public class Welcome {
    public static void main(String[] args) {
        System.out.println("Welcome to Java!");
    }
}
```

Java Compilation



Creating, Compiling, and Running Programs

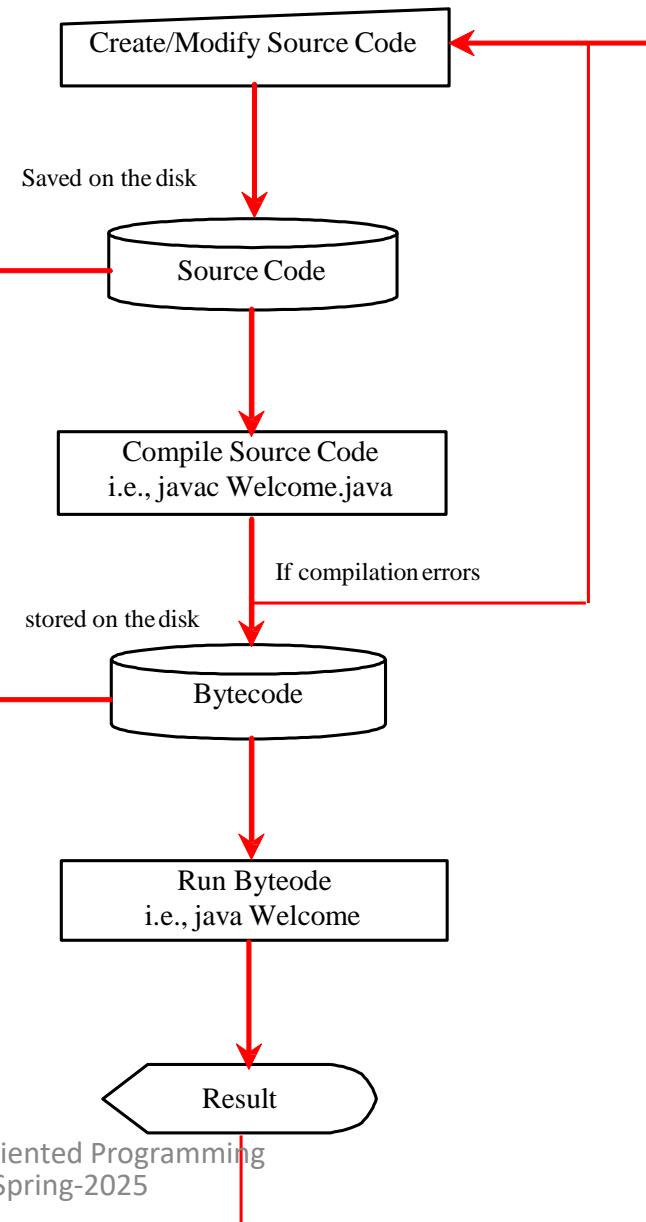
```
public class welcome {  
    public static void main(String[] args) {  
        System.out.println("welcome to Java!");  
    }  
}
```

Source code (developed by the programmer)

```
public class Welcome {  
    public static void main(String[] args) {  
        System.out.println("Welcome to Java!");  
    }  
}
```

Byte code (generated by the compiler for JVM to read and interpret, not for you to understand)

```
...  
Method Welcome()  
0 aload_0  
...  
  
Method void main(java.lang.String[])  
0 getstatic #2 ...  
3 ldc #3 <String "Welcome to  
Java!">  
5 invokevirtual #4 ...
```



Trace a Program Execution

```
//This program prints Welcome to Java!
public class Welcome {
    public static void main(String[] args) {
        System.out.println("Welcome to Java!");
    }
}
```



print a message to the
console

Reading input in JAVA

READING INPUT FROM THE `input`

- Previously, we have seen the standard output device is a console window and the `System.out` object is associated with the standard output device.
- The standard input device is normally the computer input.
- The Java API has an object `System.in` which is associated with the standard input device.

READING INPUT FROM THE `input`

The *Scanner* Class

- We will use the `System.in` object in conjunction with the `Scanner` class to read input data from the input.
- The `Scanner` class has methods that can be used to read input and format it as either a value of a primitive data type or a `String`.

READING INPUT FROM THE `input`

The *Scanner* Class

- To use the *Scanner* class in our program we must put the following statement near the top of our file, **before any class definition:**

```
import java.util.Scanner;
```

This statement tells the compiler where to find the *Scanner* class in the Java API.

READING INPUT FROM THE `input`

The *Scanner* Class

- You must also create a *Scanner* object and connect it to the *System.in* object. You can do this with a statement like the following:

```
Scanner input= new Scanner(System.in);
```

Create this object inside your *main* method before you attempt to read anything from the input.

READING INPUT FROM THE *input*

The *Scanner* Class

- The words *Scanner* *input* declare a variable named *input* of type *Scanner*. This variable will reference an object of the *Scanner* class.

Scanner **input**= new Scanner(System.in);

You could have chosen any name you wanted for the variable, but *input* is a good one since you are going to use it to access the keyboard i.e. an input device.

READING INPUT FROM THE `input`

The *Scanner* Class

`Scanner input = new Scanner(System.in);`

- The `new` key word is used to create an object in memory.
- In the statement above we are creating an object of the *Scanner* class.
- Inside the parentheses, we have `System.in`. Here we are saying that we want the object we are creating to be connected with the `System.in` object, which again is associated with the keyboard.
- We are assigning the address of the object created using the `new` operator to our variable named *input*, so *input* now references the object we have linked with the actual keyboard.

READING INPUT FROM THE `input`

The *Scanner* Class

- Every object created from the *Scanner* class has methods that read a string of characters entered at the input, convert them to a specified type, and return the converted value. This value can be stored in a variable of compatible type.

READING INPUT FROM THE `input`

The *Scanner* Class

For example, the code below could be used to read an integer entered at the input and store it in an integer variable named `age`.

```
int age;
```

```
System.out.print("Enter your age: ");
age = input.nextInt();
```

- The `nextInt()` method formats the characters entered by the user as an `int` and returns the integer value.
- The integer value is assigned to the variable named `age`.

Method	Example and Description
nextByte	<p>Example Usage:</p> <pre>byte x; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a byte value: "); x = keyboard.nextByte();</pre> <p>Description: Returns input as a byte.</p>
nextDouble	<p>Example Usage:</p> <pre>double number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a double value: "); number = keyboard.nextDouble();</pre> <p>Description: Returns input as a double.</p>
nextFloat	<p>Example Usage:</p> <pre>float number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a float value: "); number = keyboard.nextFloat();</pre> <p>Description: Returns input as a float.</p>
nextInt	<p>Example Usage:</p> <pre>int number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter an integer value: "); number = keyboard.nextInt();</pre> <p>Description: Returns input as an int.</p>
nextLine	<p>Example Usage:</p> <pre>String name; Scanner keyboard = new Scanner(System.in); System.out.print("Enter your name: "); name = keyboard.nextLine();</pre> <p>Description: Returns input as a String.</p>
nextLong	<p>Example Usage:</p> <pre>long number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a long value: "); number = keyboard.nextLong();</pre> <p>Description: Returns input as a long.</p>
nextShort	<p>Example Usage:</p> <pre>short number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a short value: "); number = keyboard.nextShort();</pre> <p>Description: Returns input as a short.</p>

READING INPUT FROM THE `input`

The *Scanner* Class

- We can use the *nextLine* method of a *Scanner* object to read a string of characters entered at the keyboard.

Example:

To get the user's first name we could write:

```
String firstName;
```

```
System.out.print("Enter your first name: ");
firstName = input.nextLine( );
```

READING INPUT FROM THE `input`

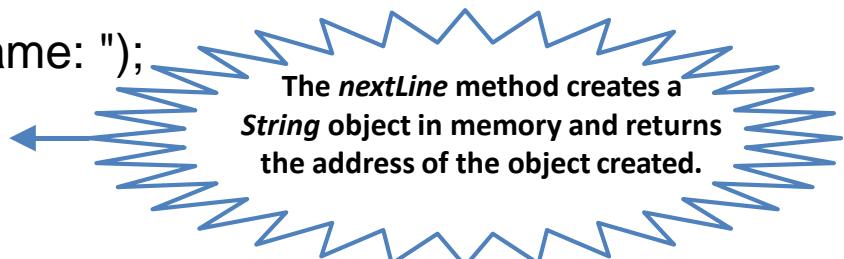
The *Scanner* Class

- The `nextLine` method creates a `String` object in memory that contains the sequence of characters entered at the keyboard before the **Enter** key is pressed and returns the address of this object.

Below we are assigning the address of the object created by the `nextLine` method to the `String` reference variable named `firstName`.

```
String firstName;
```

```
System.out.print("Enter your first name: ");  
firstName = input.nextLine( );
```



READING INPUT FROM THE `input`

The *Scanner* Class

- The *Scanner* class does not have a method for reading a single character.
- In the text, they suggest using the *Scanner* classes *nextLine* method to read the character as a string, and then using the *String* classes *charAt* method to extract the first character from the string. Remember, the first character is at index 0.

READING A SINGLE CHARACTER ENTERED AT THE input

Example:

```
String stringInitial;  
char initial;
```

```
System.out.print("Enter your middle initial " );  
stringInitial = input.nextLine( );  
initial = stringInitial.charAt(0);
```

Java Programming Constructs

Java Identifiers

- Identifiers
 - Used to name local variables
 - Names of attributes
 - Names of classes
- Primitive Data Types Available in Java (size in bytes)
 - byte (1), -128 to 127
 - short (2), -32768 to 32767
 - int (4), -2147483648 to 2147483647
 - long (8), -9223372036854775808 to 9223372036854775807
 - float (4), -3.4E38 to 3.4E38, 7 digit precision
 - double (8), -1.7E308 to 1.7E308, 17 digits precision
 - char (2), unicode characters
 - boolean (true, false), discrete values

Java Identifiers

- Naming Rules
 - Must start with a letter
 - After first letter, can consist of letters, digits (0,1,...,9)
 - The underscore “_” and the dollar sign “\$” are considered letters
- Variables
 - All variables must be declared in Java
 - Can be declared almost anywhere (scope rules apply)
 - Variables have default initialization values
 - Integers: 0
 - Reals: 0.0
 - Boolean: False
 - Variables can be initialized in the declaration

Java Identifiers

- Example Declarations

```
int speed;                      // integer, defaults to 0
int speed = 100;                 // integer, init to 100
long distance = 3000000000L;     // "L" needed for a long
float delta = 25.67f;            // "f" needed for a float
double delta = 25.67;            // Defaults to double
double bigDelta = 67.8E200d;     // "d" is optional here
boolean status;                 // defaults to "false"
boolean status = true;
```

- Potential Problems (for the C/C++ crew)

```
long double delta = 3.67E204;    // No "long double" in
                                 Java
unsigned int = 4025890231;      // No unsigned ints in
                                 Java
```

Java Types

- Arrays

```
int[] numbers = new int[n]
    // Array of integers, size is n
```

- size can be computed at run time, but can't be changed
- allocated on heap (thus enabling run time size allocation)
- invalid array accesses detected at run time (e.g. numbers[6];)
- numbers.length; // read only variable specifying length of array
- reference semantics

```
int[] winning_numbers;
winning_numbers = numbers; // refer to same array
numbers[0] = 13;           // changes both
```

Java Types

- **Strings**

```
String message = "Error " + errnum;
```

- strings are immutable – can't be changed, although variables can be changed (and old string left for garbage collection)
- message = "Next error " + errnum2;
- use **StringBuffer** to edit strings

```
StringBuffer buf = new StringBuffer(greeting);
buf.setCharAt( 4, '?' );
greeting = buf.toString();
```

Java Types

- Strings
 - String comparison

```
if (greeting == "hello" ) ....  
    // error, compares location only  
if ( greeting.equals("hello")) .... // OK  
string1.compareTo(string2)  
    // negative if string1 < string 2;  
    // zero when equal,  
    // positive if string1 > string2  
string1.substring(2, 6);  
    // return substring between position 2  
and 5
```

Java Statements

- Assignments

- General Format: variable = expression ;

Where variable is a previously declared identifier and expression is a valid combo of identifiers, operators, and method (a.k.a. procedure or function) calls

- Shortcuts:

```
var *= expr ; // Equivalent to var = var * (expr);  
var /= expr ; // Equivalent to var = var / (expr);  
var += expr ; // Equivalent to var = var + (expr);  
var -= expr ; // Equivalent to var = var - (expr);  
var %= expr ; // Equivalent to var = var % (expr);  
var++; // Equivalent to var = var + 1;  
var--; // Equivalent to var = var - 1;
```

Java Conditional Constructs

- “if” Statements

- if with code block

```
if (boolean_expr)
{
    statements
}
```

- if with single statement

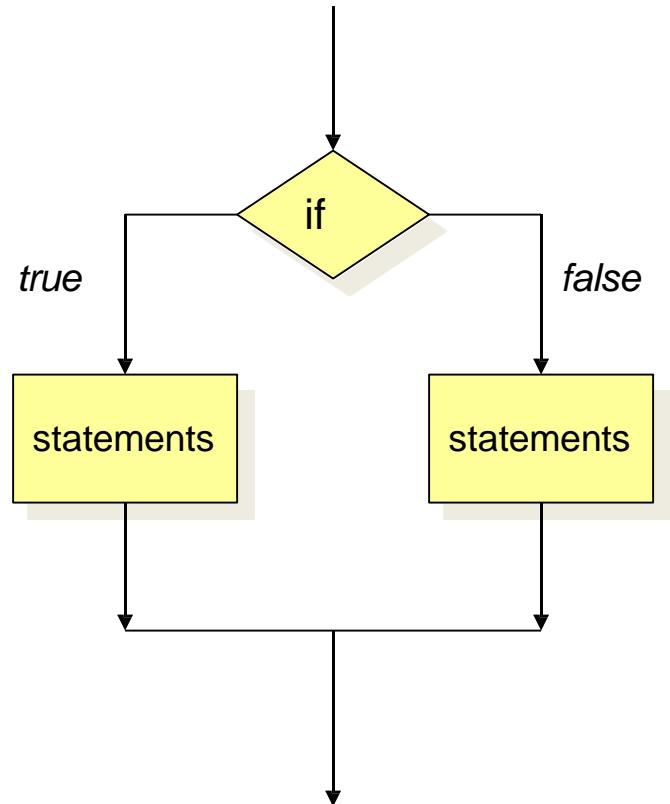
```
if (boolean_expr)
    statement;
```

Java Conditional Constructs

- “if” Statements (Continued)

- if-else

```
if (boolean_expr)
{
    statements for true
}
else
{
    statements for false
}
```



Java Conditional Constructs

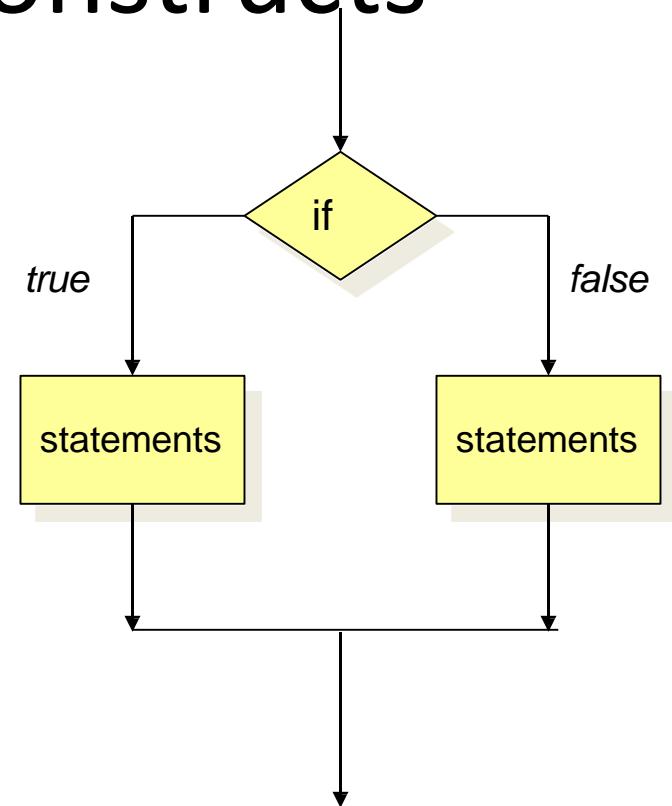
- Boolean Expressions
 - Boolean expressions use conditional operators such that they result in a value of true or false
 - Conditional Operators (Not by order of precedence)

Operator	Operation
== or !=	Equality, not equal
> or <	Greater than, less than
>= or <=	Greater than or equal, less than or equal
!	Unary negation (NOT)
& or &&	Evaluation AND, short circuit AND
 or 	Evaluation OR, short circuit OR

Java Conditional Constructs

- “if-else” Statement Example

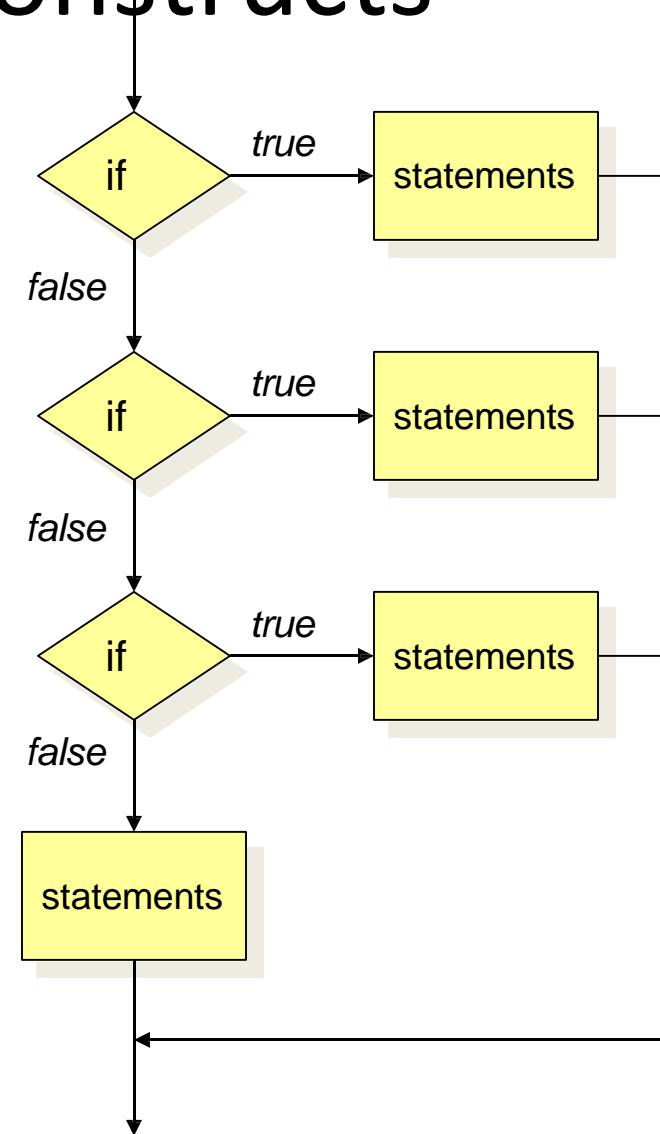
```
class Example
{
    static public void main(String args[])
    {
        // A very contrived example
        int i1 = 1, i2 = 2;
        System.out.print("Result: ");
        if (i1 > i2)
        {
            System.out.println("i1 > i2");
        }
        else
        {
            System.out.println("i2 >= i1");
        }
    }
}
```



Java Conditional Constructs

- The Switch Statement

```
switch (integer_expression)
{
    case int_value_1:
        statements
        break;
    case int_value_2:
        statements
        break;
    ...
    case int_value_n:
        statements
        break;
    default:
        statements
}
```

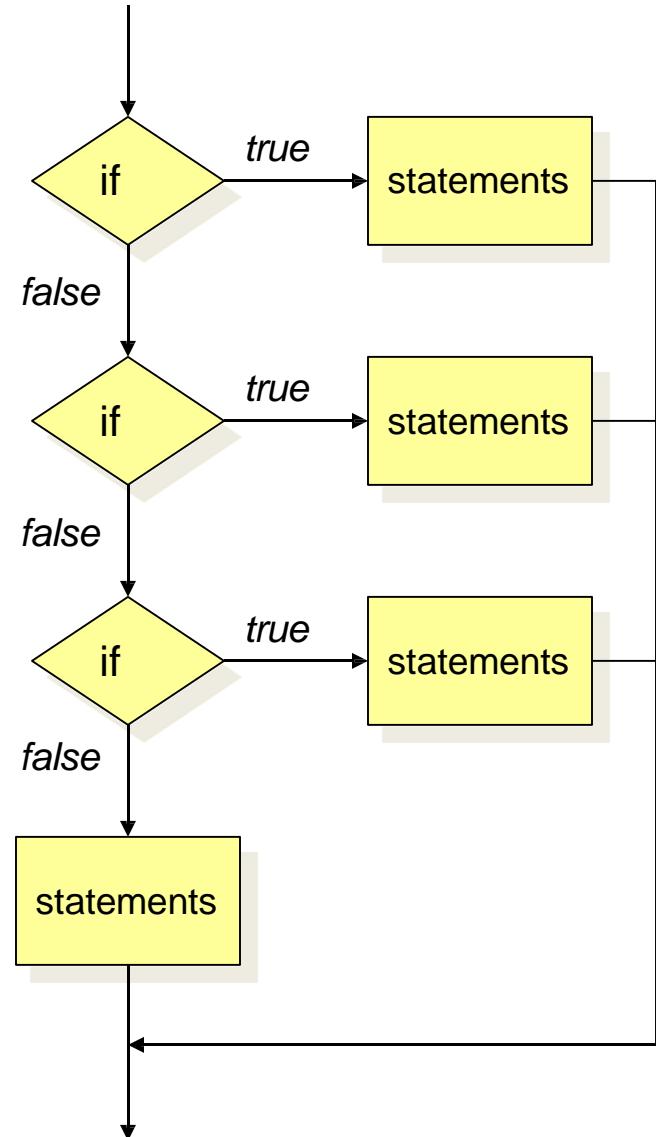


- Don't forget the “break”

```

switch (integer_expression)
{
    case int_value_1:
        statements
        // No break!
    case int_value_2:
        statements
        break;
    ...
    case int_value_n:
        statements
        break;
    default:
        statements
}

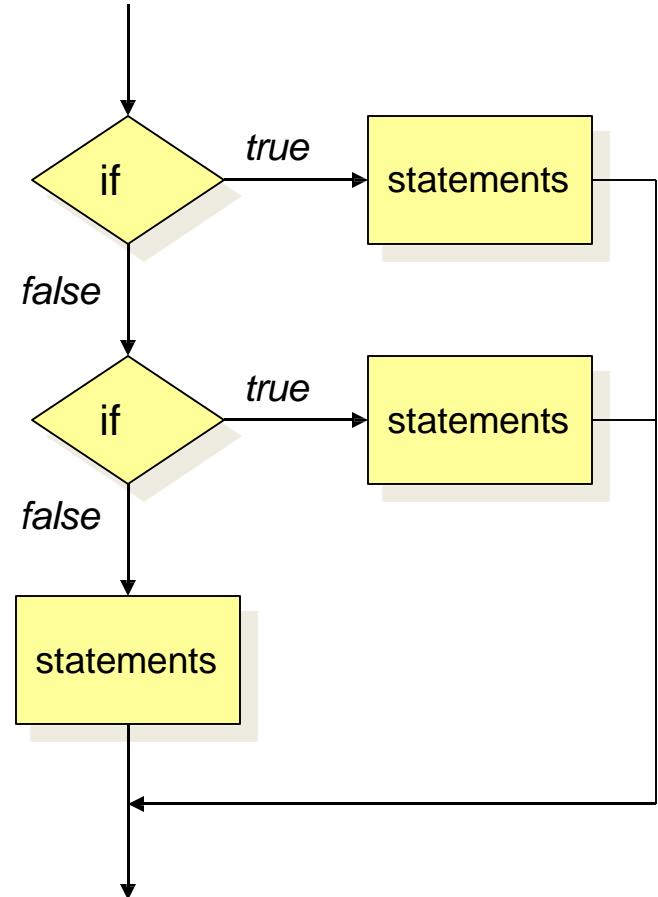
```



Java Conditional Constructs

- Example

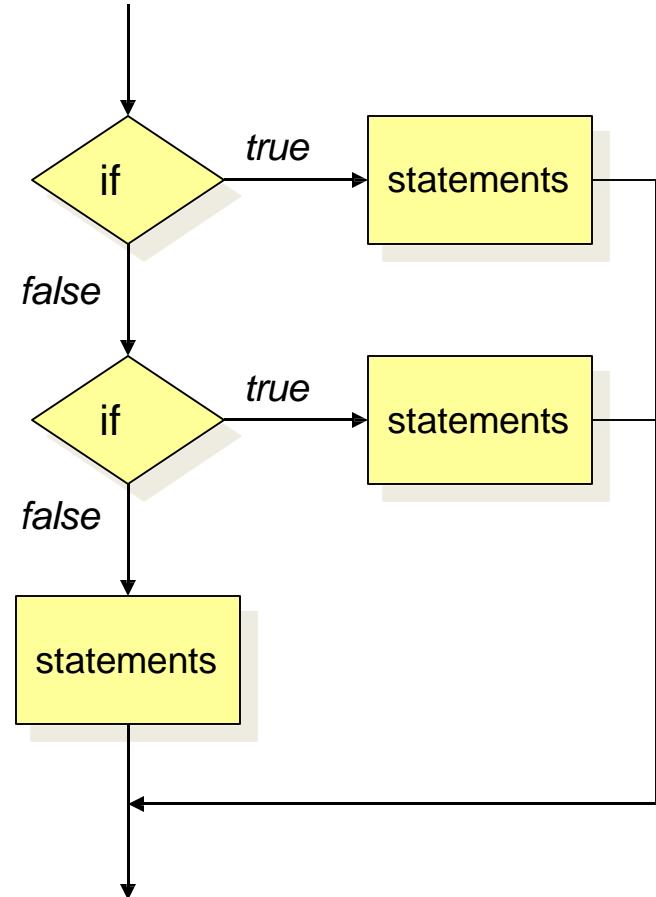
```
int n = 5;  
switch (n)  
{  
    case 1:  
        n = n + 1;  
        break;  
    case 5:  
        n = n + 2;  
        break;  
    default:  
        n = n - 1;  
}
```



Java Conditional Constructs

- Example

```
char c = 'b';
int n = 0;
switch (c)
{
    case 'a':
        n = n + 1;
        break;
    case 'b':
        n = n + 2;
        break;
    default:
        n = n - 1;
}
```



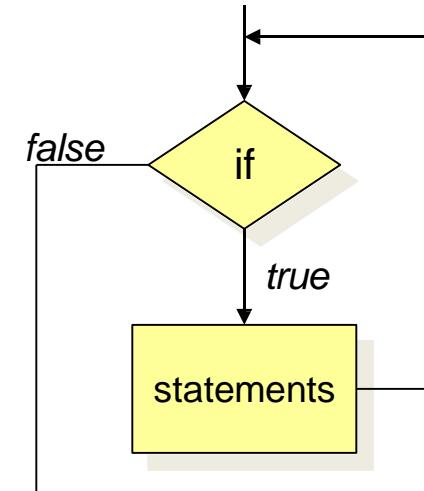
Java Looping Constructs

- while loop
 - Exit condition evaluated at top
- do loop
 - Exit condition evaluated at bottom
- for loop
 - Exit condition evaluated at top
 - Includes a initialization statements
 - Includes a update statements for each iteration

Java Looping Constructs

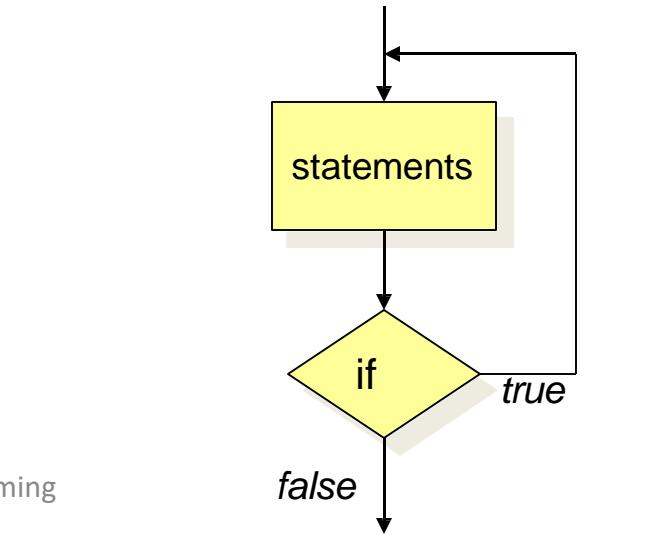
- while loop

```
while (boolean_expr)
{
    statements
}
```



- do loop

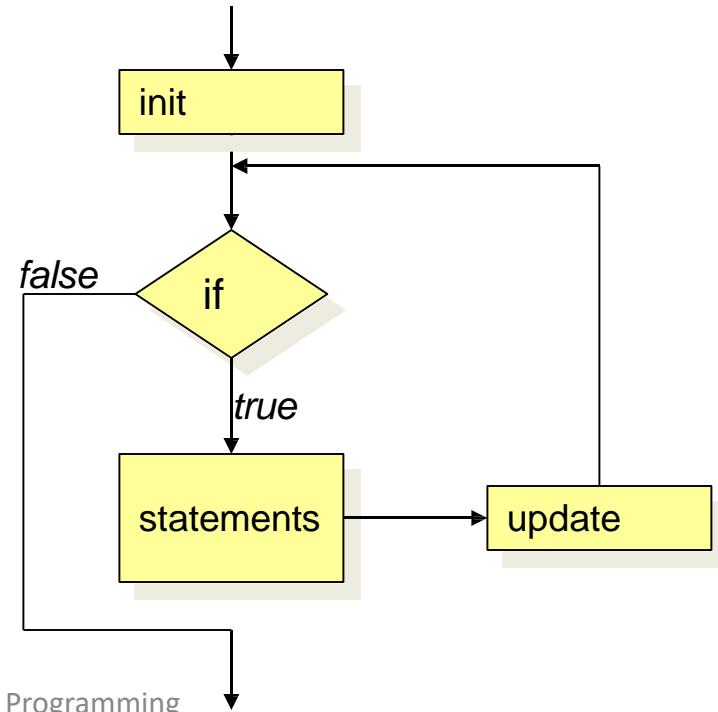
```
do
{
    statements
}
while (boolean_expr)
```



Java Looping Constructs

- for loop

```
for (init_stmnt; bool_expr; update_stmnt)  
{  
    statements  
}
```



```
class Example
{
    static public void main(String args[])
    {
        int i = 0;
        System.out.println("while loop");
        while (i < 10)
        {
            System.out.println(i);
            i++;
        }
        System.out.println("do loop");
        do
        {
            System.out.println(i);
            i--;
        }
        while (i > 0);
        System.out.println("for loop");
        for (i = 0; i < 10; i++)
        {
            System.out.println(i);
        }
    } // End main
} // End Example
```

Session - 2

What is Object-Orientation?

- A technique for system modeling
- OO model consists of several interacting objects

What is a Model?

- A model is an abstraction of something
- Purpose is to understand the product before developing it

Examples – Model

- Highway maps
- Architectural models
- Mechanical models

Object-Orientation - Advantages

- People think in terms of objects
- OO models map to reality
- Therefore, OO models are
 - easy to develop
 - easy to understand

What is an Object?

An object is

- Something tangible (Ali, Car)
- Something that can be apprehended intellectually (Time, Date)

... What is an Object?

An object has

- State (attributes)
- Well-defined behaviour (operations)
- Unique identity

Information Hiding

- Information is stored within the object
- It is hidden from the outside world
- It can only be manipulated by the object itself

Example – Information Hiding

- A phone stores several phone numbers
- We can't read the numbers directly from the SIM card
- Rather phone-set reads this information for us

Information Hiding Advantages

- Simplifies the model by hiding implementation details
- It is a barrier against change propagation

Encapsulation

- Data and behaviour are tightly coupled inside an object
- Both the information structure and implementation details of its operations are hidden from the outer world

Example – Encapsulation

- A Phone stores phone numbers in digital format and knows how to convert it into human-readable characters
- We don't know
 - How the data is stored
 - How it is converted to human-readable characters

Encapsulation – Advantages

- Simplicity and clarity
- Low complexity
- Better understanding

Object has an Interface

- An object encapsulates data and behaviour
- So how objects interact with each other?
- Each object provides an interface (operations)
- Other objects communicate through this interface

Example – Interface of a Car

- Steer Wheels
- Accelerate
- Change Gear
- Apply Brakes
- Turn Lights On/Off

Example – Interface of a Phone

- Input Number
- Place Call
- Disconnect Call
- Add number to address book
- Remove number
- Update number

Implementation

- Provides services offered by the object interface
- This includes
 - Data structures to hold object state
 - Functionality that provides required services

Example – Implementation of Gear Box

- Data Structure
 - Mechanical structure of gear box
- Functionality
 - Mechanism to change gear

Example – Implementation of Address Book in a Phone

- Data Structure
 - SIM card
- Functionality
 - Read/write circuitry

Separation of Interface & Implementation

- Means change in implementation does not affect object interface
- This is achieved via principles of information hiding and encapsulation

Example – Separation of Interface & Implementation

- A driver can drive a car independent of engine type (petrol, diesel)
- Because interface does not change with the implementation

Example – Separation of Interface & Implementation

- A driver can apply brakes independent of brakes type (simple, disk)
- Again, reason is the same interface

Advantages of Separation

- Users need not to worry about a change until the interface is same
- Low Complexity
- Direct access to information structure of an object can produce errors

Messages

- Objects communicate through messages
- They send messages (stimuli) by invoking appropriate operations on the target object
- The number and kind of messages that can be sent to an object depends upon its interface

Examples – Messages

- A Person sends message (stimulus) “stop” to a Car by applying brakes
- A Person sends message “place call” to a Phone by pressing appropriate button

Java Classes and Objects

Session 3



- ◆ Explain the process of creation of classes in Java
- ◆ Explain the instantiation of objects in Java
- ◆ Explain the purpose of instance variables and instance methods
- ◆ Explain constructors in Java
- ◆ Explain the memory management in Java
- ◆ Explain object initializers



◆ Class in Java:

- ❖ Is the prime unit of execution for object-oriented programming in Java.
- ❖ Is a logical structure that defines the shape and nature of an object.
- ❖ Is defined as a new data type that is used to create objects of its type.
- ❖ Defines attributes referred to as fields that represents the state of an object.

Conventions to be followed while naming a class

- Class declaration should begin with the keyword class followed by the name of the class.
- Class name should be a noun.
- Class name can be in mixed case, with the first letter of each internal word capitalized.
- Class name should be simple, descriptive, and meaningful.
- Class name cannot be Java keywords.
- Class name cannot begin with a digit. However, they can begin with a dollar (\$) symbol or an underscore character.

Declaring a Class 1-2



- The syntax to declare a class in Java is as follows:

Syntax

```
class <class_name> {  
    // class body  
}
```

- The body of the class is enclosed between the curly braces { }.
- In the class body, you can declare members, such as fields, methods, and constructors.
- Following figure shows the declaration of a sample class:

```
class Student {  
    String studName;  
    int studAge;  
  
    void initialize()  
    {  
        studName = "James Anderson";  
        studAge = 26;  
    }  
  
    void display()  
    {  
        System.out.println("Student Name: " + studName);  
        System.out.println("Student Age:" + studAge);  
    }  
  
    public static void main(String[] args)  
    {  
        Student objStudent = new Student();  
        objStudent.initialize();  
        objStudent.display();  
    }  
}
```

Fields or Instance Variables

Functions or Instance Methods

Declaring a Class 2-2



- ◆ Following code snippet shows the code for declaring a class **Customer**:

```
class Customer {  
    // body of class  
}
```

- ◆ In the code:
 - ❖ A class is declared that acts as a new data type with the name **Customer**.
 - ❖ It is just a template for creating multiple objects with similar features.
 - ❖ It does not occupy any memory.
- ◆ **Creating Objects:**
 - ❖ Objects are the actual instances of the class.



- ◆ An object is created using the `new` operator.
- ◆ On encountering the `new` operator:
 - ◆ JVM allocates memory for the object.
 - ◆ Returns a reference or memory address of the allocated object.
 - ◆ The reference or memory address is then stored in a variable called as reference variable.
- ◆ The syntax for creating an object is as follows:

Syntax

```
<class_name> <object_name> = new <class_name> () ;
```

where,

`new`: Is an operator that allocates the memory for an object at runtime.

`object_name`: Is the variable that stores the reference of the object.

Declaring and Creating an Object 2-2



- ◆ Following code snippet demonstrates the creation of an object in a Java program:

```
Customer objCustomer = new Customer();
```

- ◆ The expression on the right side, **new Customer()** allocates the memory at runtime.
- ◆ After the memory is allocated for the object, it returns the reference or address of the allocated object, which is stored in the variable, **objCustomer**.



- ◆ Alternatively, an object can be created using two steps that are as follows:
 - ◆ Declaration of an object reference.
 - ◆ Dynamic memory allocation of an object.
- ◆ **Declaration of an object reference:**
 - ◆ The syntax for declaring the object reference is as follows:

Syntax

```
<class_name> <object_name>;
```

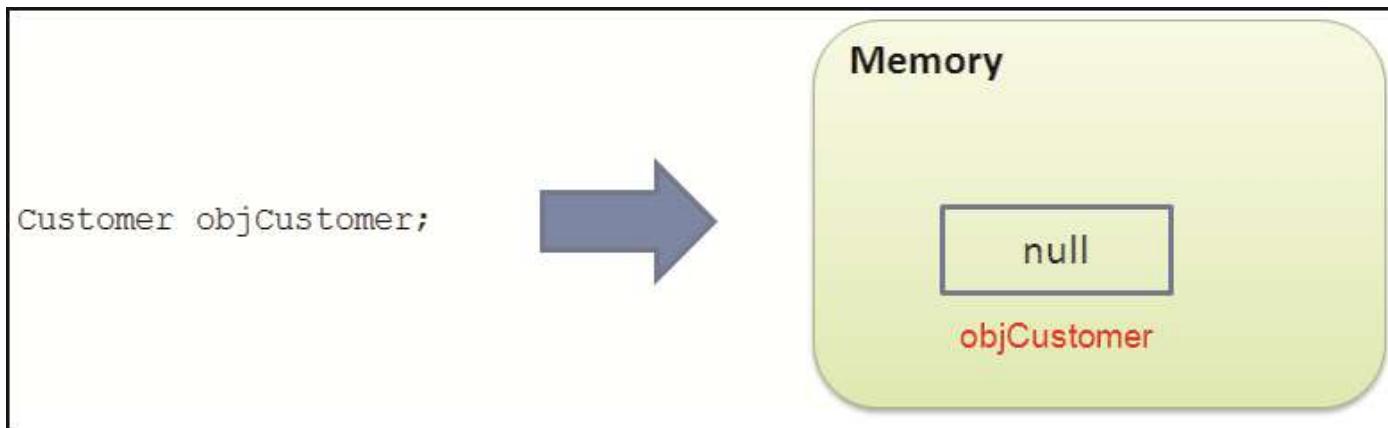
where,

`object_name`: Is just a variable that will not point to any memory location.

Creation of an Object: Two Stage Process 2-3



- Following figure shows the effect of the statement, **Customer objCustomer;** which declares a reference variable:

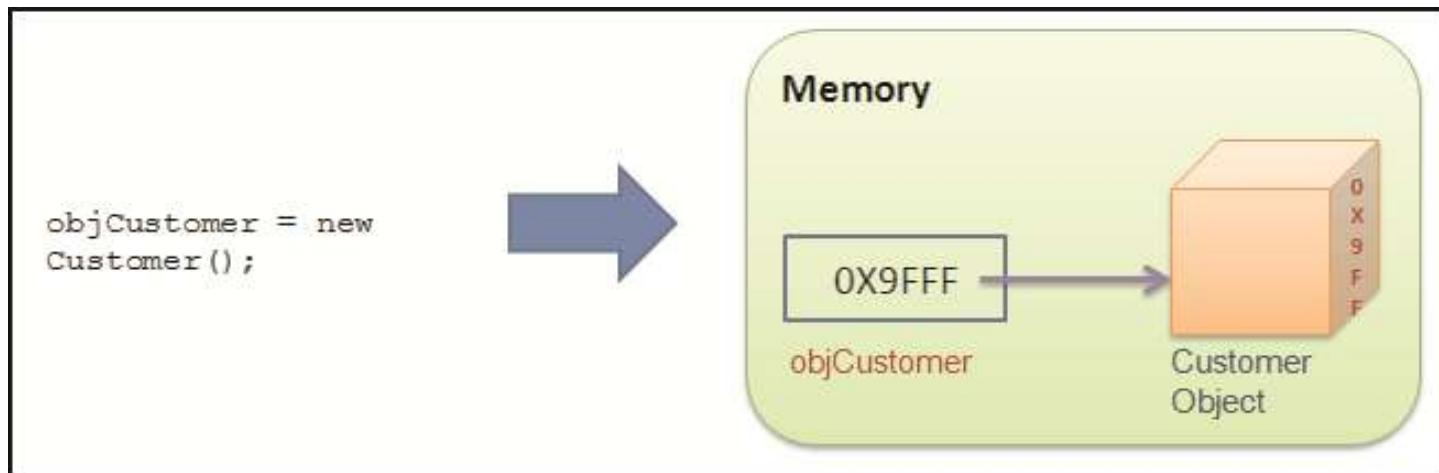


- By default, the value `null` is stored in the object's reference variable which means reference variable does not point to an actual object.
- If **objCustomer** is used at this point of time, without being instantiated, then the program will result in a compile time error.

Creation of an Object: Two Stage Process 3-3



- ◆ **Dynamic memory allocation of an object:**
 - ❖ The object should be initialized using the `new` operator which dynamically allocates memory for an object.
 - ❖ For example, the statement, `objCustomer = new Customer();` allocates memory for the object and memory address of the allocated object is stored in the variable `objCustomer`.
- ◆ Following figure shows the creation of object in the memory and storing of its reference in the variable, `objCustomer`:





- ◆ The members of a class are fields and methods.

Fields

- Define the state of an object created from the class.
- Referred to as instance variables.

Methods

- Implement the behavior of the objects.
- Referred to as instance methods.

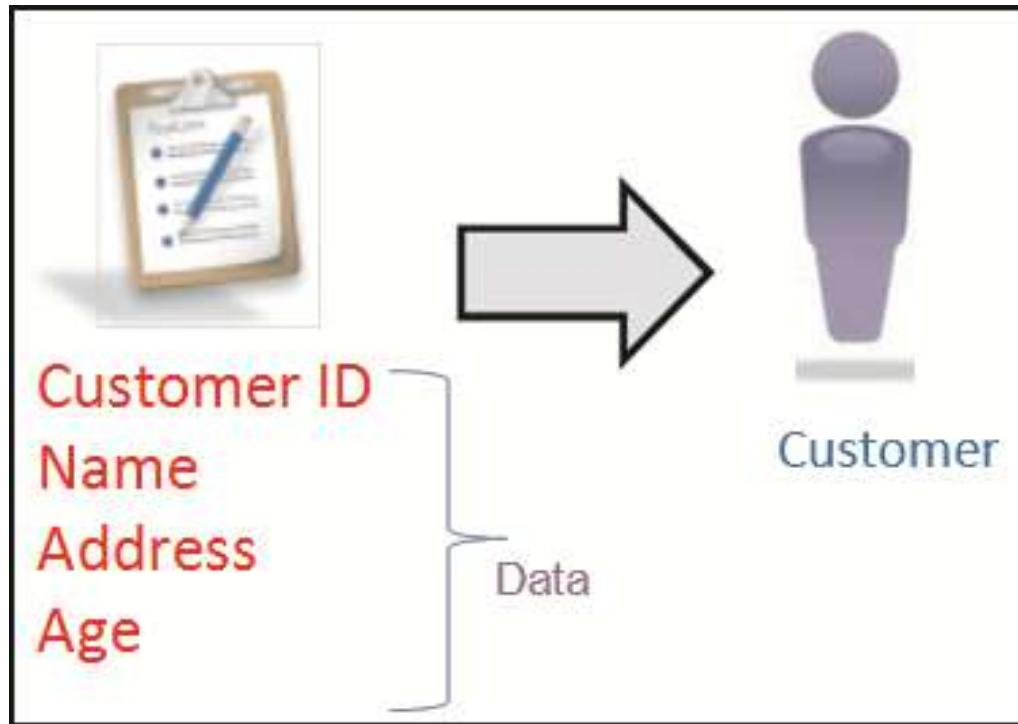


- ◆ They are used to store data in them.
- ◆ They are called instance variables because each instance of the class, that is, object of that class will have its own copy of the **instance variables**.
- ◆ They are declared similar to local variables.
- ◆ They are declared inside a class, but outside any method definitions.
- ◆ For example: Consider a scenario where the **Customer** class represents the details of customers holding accounts in a bank.
 - ❖ A typical question that can be asked is 'What are the different data that are required to identify a customer in a banking domain and represent it as a single object?'.

Instance Variables 2-7



- Following figure shows a **Customer** object with its data requirement:

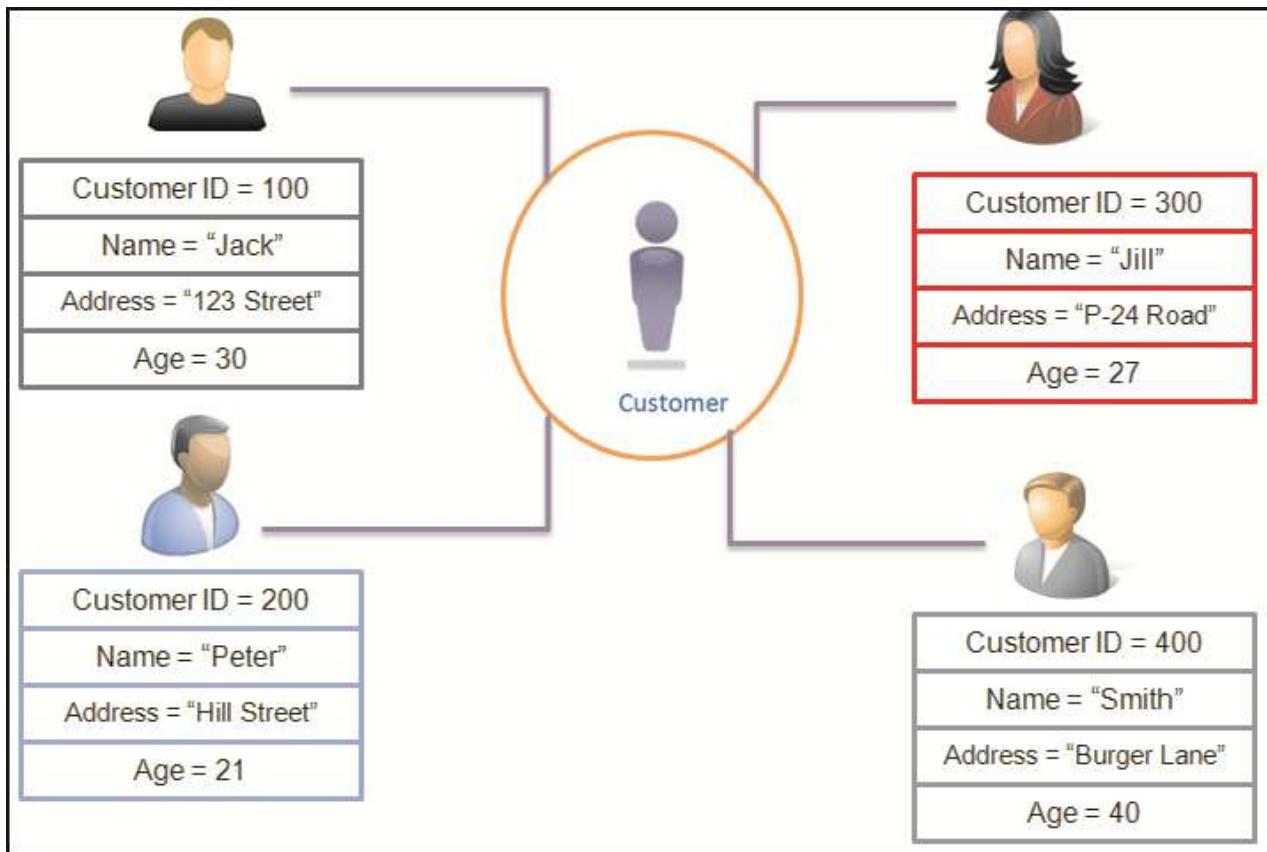


- The identified data requirements for a bank customer includes: Customer ID, Name, Address, and Age.
- To map these data requirements in a **Customer** class, **instance variables** are declared.

Instance Variables 3-7



- ◆ Each instance created from the **Customer** class will have its own copy of the instance variables.
- ◆ Following figure shows various instances of the class with their own copy of instance variables:





- The syntax to declare an instance variable within a class is as follows:

Syntax

```
[access_modifier] data_type instanceVariableName;
```

where,

access_modifier: Is an optional keyword specifying the access level of an instance variable. It could be private, protected, and public.

data_type: Specifies the data type of the variable.

instanceVariableName: Specifies the name of the variable.

- Instance variables are accessed by objects using the dot operator (.).



- Following code snippet demonstrates the declaration of instance variables within a class in the Java program:

```
1: public class Customer {  
2:     // Declare instance variables  
3:     int customerID;  
4:     String customerName;  
5:     String customerAddress;  
6:     int customerAge;  
  
7:     /* As main() method is a member of class, so it can access other  
8:      * members of the class */  
9:     public static void main(String[] args) {  
10:         // Declares and instantiates an object of type Customer  
11:         Customer objCustomer1 = new Customer();
```

- Lines 3 to 6 declares instance variables.
- Line 11 creates an object of type **Customer** and stores its reference in the variable, **objCustomer1**.

Instance Variables 6-7



```
12: // Accesses the instance variables to store values
13: objCustomer1.customerID = 100;
14: objCustomer1.customerName = "John";
15: objCustomer1.customerAddress = "123 Street";
16: objCustomer1.customerAge = 30;

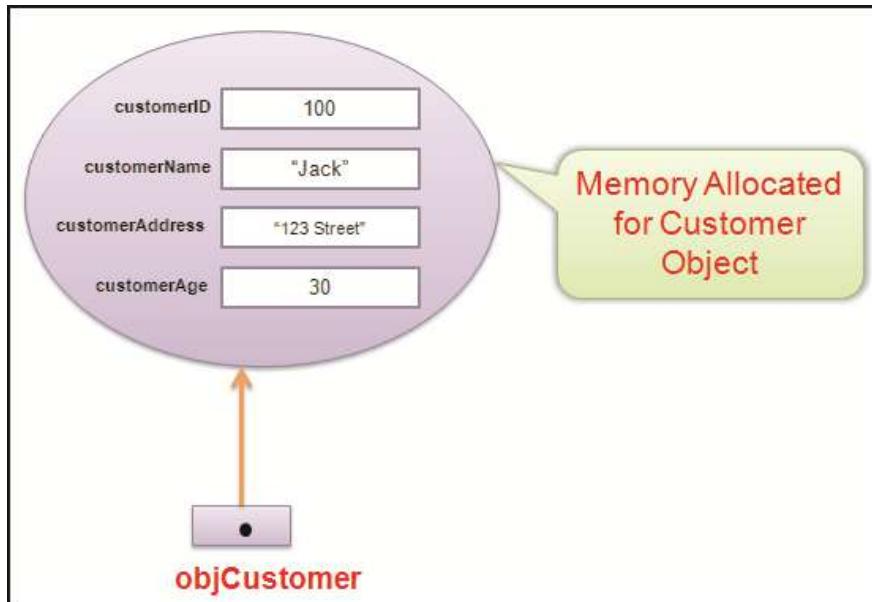
17: // Displays the objCustomer1 object details
18: System.out.println("Customer Identification Number: " +
objCustomer1.customerID);
19: System.out.println("Customer Name: " + objCustomer1.customerName);
20: System.out.println("Customer Address: " + objCustomer1.
customerAddress);
21: System.out.println("Customer Age: " + objCustomer1.customerAge);
}
}
```

- ❖ Lines 13 to 16 accesses the instance variables and assigns them the values.
- ❖ Lines 18 to 21 display the values assigned to the instance variables for the object, **objCustomer1**.

Instance Variables 7-7



- Following figure shows the allocation of **Customer** object in the memory:



- Following figure shows the output of the code:

A screenshot of a Java IDE showing the output of a program. The window title is **Output - Session 6 (run)**. The output pane displays the following text:
run:
Customer Identification Number: 100
Customer Name: John
Customer Address: 123 Street
Customer Age: 30
BUILD SUCCESSFUL (total time: 1 second)



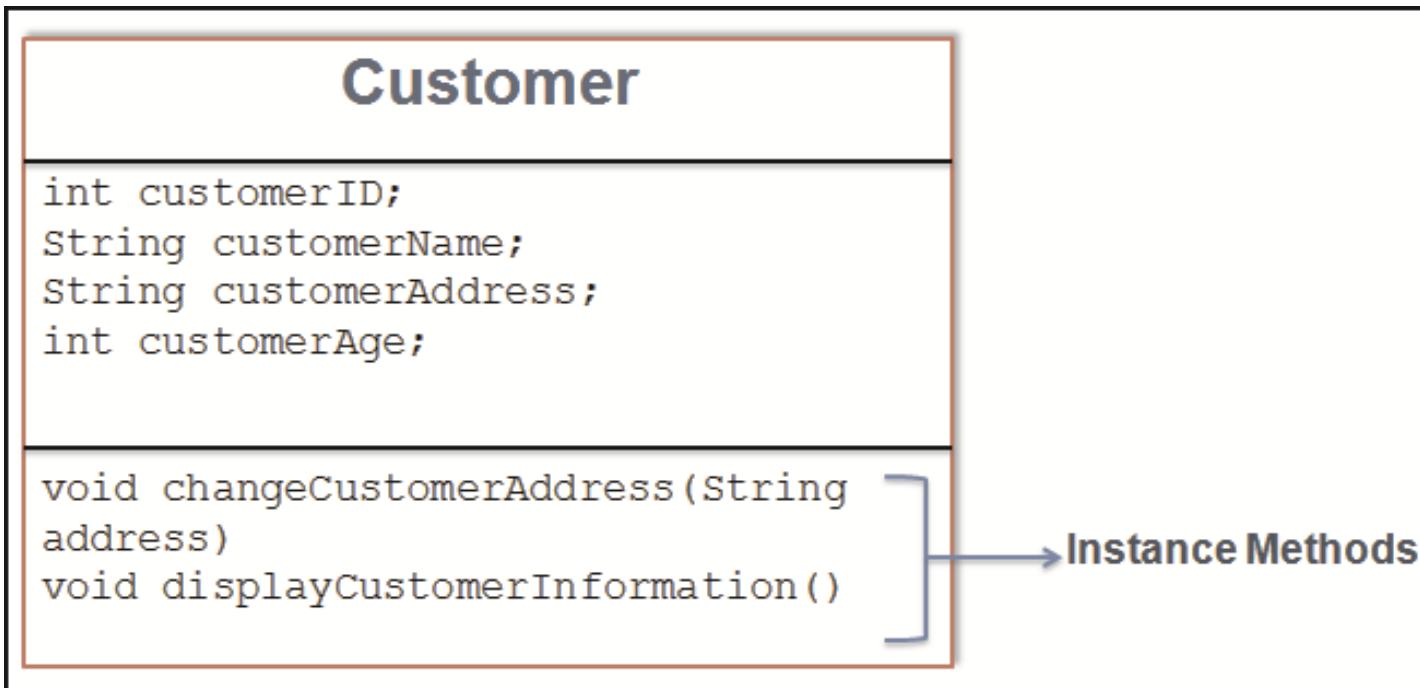
- ◆ They are functions declared in a class.
- ◆ They implement the behavior of an object.
- ◆ They are used to perform operations on the instance variables.
- ◆ They can be accessed by instantiating an object of the class in which it is defined and then, invoking the method.
- ◆ For example: the class **Car** can have a method `Brake()` that represents the 'Apply Brake' action.
 - ◆ To perform the action, the method `Brake()` will have to be invoked by an object of class **Car**.

Conventions to be followed while naming a method are as follows:

- Cannot be a Java keyword.
- Cannot contain spaces.
- Cannot begin with a digit.
- Can begin with a letter, underscore, or a '\$' symbol.
- Should be a verb in lowercase.
- Should be descriptive and meaningful.
- Should be a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, and so forth.



- Following figure shows the instance methods declared in the class, **Customer**:





- ◆ The syntax to declare an instance method in a class is as follows:

Syntax

```
[access_modifier] <return type> <method_name> ([list  
of parameters]) {  
  
    // Body of the method  
  
}
```

where,

access_modifier: Is an optional keyword specifying the access level of an instance method. It could be private, protected, and public.

returntype: Specifies the data type of the value that is returned by the method.

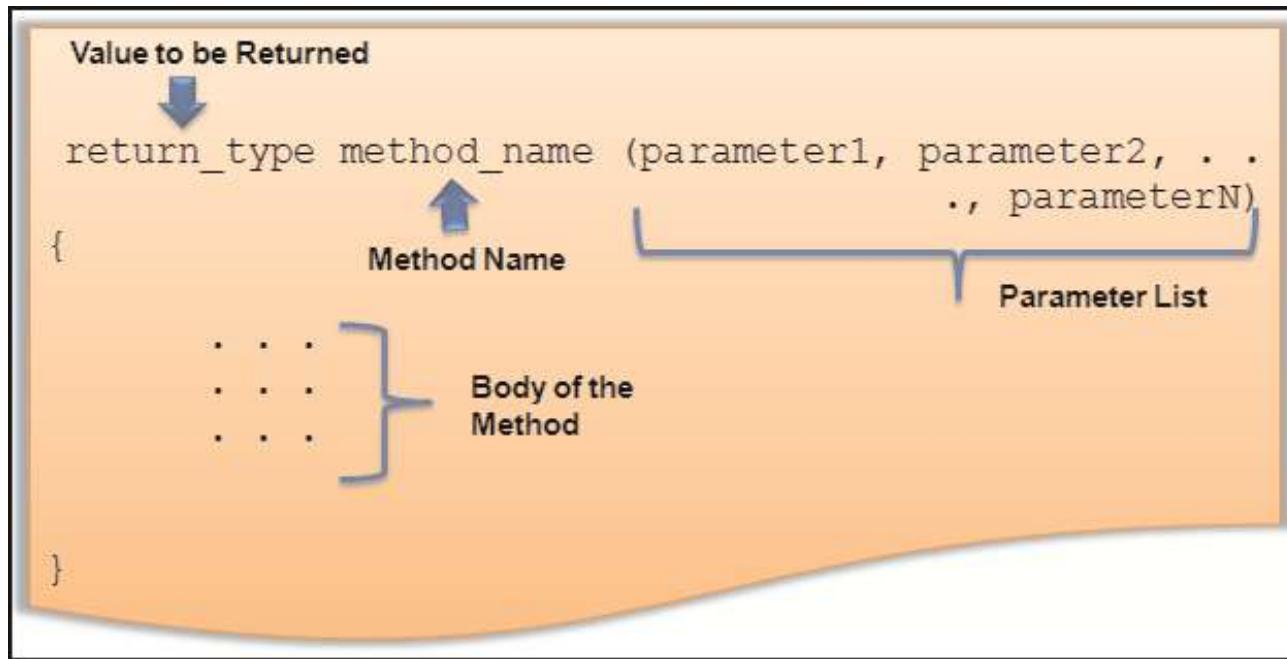
method_name: Is the method name.

list of parameters: Are the values passed to the method.

Instance Methods 4-6



- Following figure shows the declaration of an instance method within the class:



- Each instance of the class has its own instance variables, but the instance methods are shared by all the instances of the class during execution.

Instance Methods 5-6



- Following code snippet demonstrates the code that declares instance methods within the class, **Customer**:

```
public class Customer {  
    // Declare instance variables  
    int customerID;  
    String customerName;  
    String customerAddress;  
    int customerAge;  
  
    /**  
     * Declares an instance method changeCustomerAddress is created to  
     * change the address of the customer object  
     */  
    void changeCustomerAddress(String address) {  
        customerAddress = address;  
    }  
}
```

- The instance methods **changeCustomerAddress ()** method will accept a string value through parameter address.
- It then assigns the value of address variable to the **customerAddress** field.

Instance Methods 6-6



```
/**  
 * Declares an instance method displayCustomerInformation is created  
 * to display the details of the customer object  
 */  
void displayCustomerInformation() {  
    System.out.println("Customer Identification Number: " + customerID);  
    System.out.println("Customer Name: " + customerName);  
    System.out.println("Customer Address: " + customerAddress);  
    System.out.println("Customer Age: " + customerAge);  
}  
}
```

- ◆ The method **displayCustomerInformation ()** displays the details of the customer object.



- ◆ To invoke a method, the object name is followed by the dot operator (.) and the method name.
- ◆ A method is always invoked from another method.
- ◆ The method which invokes a method is referred to as the **calling method**.
- ◆ The invoked method is referred to as the **called method**.
- ◆ After execution of all the statements within the code block of the invoked method, the control returns back to the **calling method**.

Invoking Methods 2-4



- Following code snippet demonstrates a class with main() method which creates the instance of the class **Customer** and invokes the methods defined in the class:

```
public class TestCustomer {  
  
    /**  
     * @param args the command line arguments  
     * The main() method creates the instance of class Customer  
     * and invoke its methods  
     */  
    public static void main(String[] args) {  
        // Creates an object of the class  
        Customer objCustomer = new Customer();  
  
        // Initialize the object  
        objCustomer.customerID = 100;  
        objCustomer.customerName = "Jack";  
        objCustomer.customerAddress = "123 Street";  
        objCustomer.customerAge = 30;  
    }  
}
```

- The code instantiates an object **objCustomer** of type **Customer** class and initializes its instance variables.

Invoking Methods 3-4



```
/*
 * Invokes the instance method to display the details
 * of objCustomer object
 */
    objCustomer.displayCustomerInformation();

/*
 * Invokes the instance method to
 * change the address of the objCustomer object
 */
    objCustomer.changeCustomerAddress("123 Fort, Main Street");

/*
 * Invokes the instance method after changing the address field
 * of objCustomer object
 */
    objCustomer.displayCustomerInformation();
}

}
```

- ◆ The method **displayCustomerInformation()** is invoked using the object **objCustomer** and displays the values of the initialized instance variables on the console.
- ◆ Then, the method **changeCustomerAddress ("123 Fort, Main Street")** is invoked to change the data of the **customerAddress** field.

Invoking Methods 4-4



- Following figure shows the output of the code:

```
Output - Session 6 (run) ✘
run:
Customer Details
=====
Customer Identification Number: 100
Customer Name: Jack
Customer Address: 123 Street
Customer Age: 30

Modified Customer Details
=====
Customer Identification Number: 100
Customer Name: Jack
Customer Address: 123 Fort, Main Street
Customer Age: 30
BUILD SUCCESSFUL (total time: 0 seconds)
```

Constructor 1-3



- ◆ It is a method having the same name as that of the class.
- ◆ It initializes the variables of a class or perform startup operations only once when the object of the class is instantiated.
- ◆ It is automatically executed whenever an instance of a class is created.
- ◆ It can accept parameters and do not have return types.
- ◆ It is of two types:
 - ◆ No-argument constructor
 - ◆ Parameterized constructor
- ◆ Following figure shows the constructor declaration:

```
class <ClassName>
{
    <ClassName>()
    {
        // Initialization code
    }
}
```

The code shows a Java constructor declaration. The constructor is highlighted with a red box and labeled 'Constructor' with a red arrow pointing to it. The code includes a placeholder for the class name (<ClassName>) and contains initialization code within curly braces {}.



- ◆ The syntax for declaring constructor in a class is as follows:

Syntax

```
<classname>() {  
    // Initialization code  
}
```



- ◆ **No-argument Constructor:**

- ◆ Following code snippet demonstrates a class **Rectangle** with a constructor:

```
public class Rectangle {  
    int width;  
    int height;  
  
    /**  
     * Constructor for Rectangle class  
     */  
    Rectangle() {  
        width = 10;  
        height = 10;  
    }  
}
```

- ◆ The code declares a method named **Rectangle()** which is a constructor.
- ◆ This method is invoked by JVM to initialize the two instance variables, **width** and **height**, when the object of type **Rectangle** is constructed.
- ◆ The constructor does not have any parameters; hence, it is called as **no-argument constructor**.

Invoking Constructor 1-3



- ◆ The constructor is invoked immediately during the object creation which means:
 - ❖ Once the new operator is encountered, memory is allocated for the object.
 - ❖ Constructor method is invoked by the JVM to initialize the object.
- ◆ Following figure shows the use of new operator to understand the constructor invocation:

```
<class_name> <object_name> = new <class_name>();
```

The parenthesis after the class name indicates the invocation of the constructor.

Invoking Constructor 2-3



- Following code snippet demonstrates the code to invoke the constructor for the class **Rectangle**:

```
public class TestConstructor {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        // Instantiates an object of the Rectangle class  
        Rectangle objRec = new Rectangle();  
  
        // Accesses the instance variables using the object reference  
        System.out.println("Width: " + objRec.width);  
        System.out.println("Height: " + objRec.height);  
    }  
}
```

- The code does the following:
 - Creates an object, **objRec** of type **Rectangle**.
 - Then, the constructor is invoked.
 - The constructor initializes the instance variables of the newly created object, that is, **width** and **height** to 10.

Invoking Constructor 3-3



- ◆ Following figure shows the output of the code:

The screenshot shows the 'Output - Session 6 (run)' window of an IDE. The window contains the following text:
run:
Constructor Invoked...
Width: 10
Height: 10
BUILD SUCCESSFUL (total time: 0 seconds)



- ◆ Consider a situation, where the constructor method is not defined for a class.
- ◆ In such a scenario, an implicit constructor is invoked by the JVM for initializing the objects.
- ◆ This implicit constructor is also known as default constructor.
- ◆ **Default Constructor:**
 - ◆ Created for the classes where explicit constructors are not defined.
 - ◆ Initializes the instance variables of the newly created object to their default values.
 - ◆ Default constructor is defined in **Object** class. Object class is root class in both C# and Java.

Default Constructor 2-5



- Following table lists the default values assigned to instance variables of the class depending on their data types:

Data Type	Default Value
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0
char	'\u0000'
boolean	False
String (any object)	Null

Default Constructor 3-5



- Following code snippet demonstrates a class **Employee** for which no constructor has been defined:

```
public class Employee {  
    // Declares instance variables  
    String employeeName;  
    int employeeAge;  
    double employeeSalary;  
    boolean maritalStatus;  
    /**  
     * Accesses the instance variables and displays  
     * their values using the println() method  
     */  
    void displayEmployeeDetails() {  
        System.out.println("Employee Details");  
        System.out.println("=====");  
        System.out.println("Employee Name: " + employeeName);  
        System.out.println("Employee Age: " + employeeAge);  
        System.out.println("Employee Salary: " + employeeSalary);  
        System.out.println("Employee MaritalStatus:" + maritalStatus);  
    }  
}
```

- The code declares a class **Employee** with instance variables and an instance method **displayEmployeeDetails()** that prints the value of the instance variables.

Default Constructor 4-5



- Following code snippet demonstrates a class containing the main () method that creates an instance of the class **Employee** and invokes its methods:

```
public class TestEmployee {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        // Instantiates an Employee object and initializes it  
        Employee objEmp = new Employee();  
  
        // Invokes the displayEmployeeDetails() method  
        objEmp.displayEmployeeDetails();  
    }  
}
```

- As the **Employee** class does not have any constructor defined for itself, a **default constructor** is created for the class at runtime.
- When the statement **new Employee()** is executed, the object is allocated in memory and the instance variables are initialized to their default values by the constructor.
- Then, the method **displayEmployeeDetails()** is executed which displays the values of the instance variables referenced by the object, **objEmp**.

Default Constructor 5-5



- Following figure shows the output of the code:

The screenshot shows an IDE's output window titled "Output - Session 6 (run)". The window displays the execution of a Java application. The output text is as follows:

```
run:  
Employee Details  
=====  
Employee Name: null  
Employee Age: 0  
Employee Salary: 0.0  
Employee MaritalStatus:false  
BUILD SUCCESSFUL (total time: 2 seconds)
```

Parameterized Constructor 1-5



- ◆ The parameterized constructor contains a list of parameters that initializes instance variables of an object.
- ◆ The value for the parameters is passed during the object creation.
- ◆ This means each object will be initialized with different set of values.
- ◆ Following code snippet demonstrates a code that declares parameterized constructor for the **Rectangle** class:

```
public class Rectangle {  
    int width;  
    int height;  
  
    /**  
     * A default constructor for Rectangle class  
     */  
  
    Rectangle() {  
        System.out.println("Constructor Invoked...");  
        width = 10;  
        height = 10;  
    }  
}
```

Parameterized Constructor 2-5



```
/**  
 * A parameterized constructor with two parameters  
 * @param wid will store the width of the rectangle  
 * @param heig will store the height of the rectangle  
 */  
Rectangle (int wid, int heig) {  
    System.out.println("Parameterized Constructor");  
    width = wid;  
    height = heig;  
}  
  
/**  
 * This method displays the dimensions of the Rectangle object  
 */  
void displayDimensions () {  
    System.out.println("Width: " + width);  
    System.out.println("Width: " + height);  
}  
}
```

- ❖ The code declares a parameterized constructor, **Rectangle(int wid, int heig)**.
- ❖ During execution, the constructor will accept the values in two parameters and assigns them to **width** and **height** variable respectively.

Parameterized Constructor 3-5



- Following code snippet demonstrates the code with `main()` method that creates objects of type **Rectangle** and initializes them with parameterized constructor:

```
public class RectangleInstances {  
  
    /**  
     * @param args the command line arguments  
     */  
  
    public static void main(String[] args) {  
        // Declare and initialize two objects for Rectangle class  
        Rectangle objRec1 = new Rectangle(10, 20);  
        Rectangle objRec2 = new Rectangle(6, 9);  
  
        // Invokes displayDimensions() method to display values  
        System.out.println("\nRectangle1 Details");  
        System.out.println("=====");  
        objRec1.displayDimensions();  
        System.out.println("\nRectangle2 Details");  
        System.out.println("=====");  
        objRec2.displayDimensions();  
    }  
}
```

Parameterized Constructor 4-5



- ◆ The statement `Rectangle objRect1 = new Rectangle(10, 20);` instantiates an object.
- ◆ During instantiation, the following things happen in a sequence:
 - ◆ Memory allocation is done for the new instance of the class.
 - ◆ Values 10 and 20 are passed to the parameterized constructor, `Rectangle(int wid, int heig)` which initializes the object's instance variables `width` and `height`.
 - ◆ Finally, the reference of the newly created instance is returned and stored in the object, `objRec1`.
- ◆ Following figure shows the output of the code:

The screenshot shows an IDE's output window titled "Output - Session 6 (run)". It displays the following text:

```
run:
Parameterized Constructor Invoked...
Parameterized Constructor Invoked...

Rectangle1 Details
=====
Width: 10
Width: 20

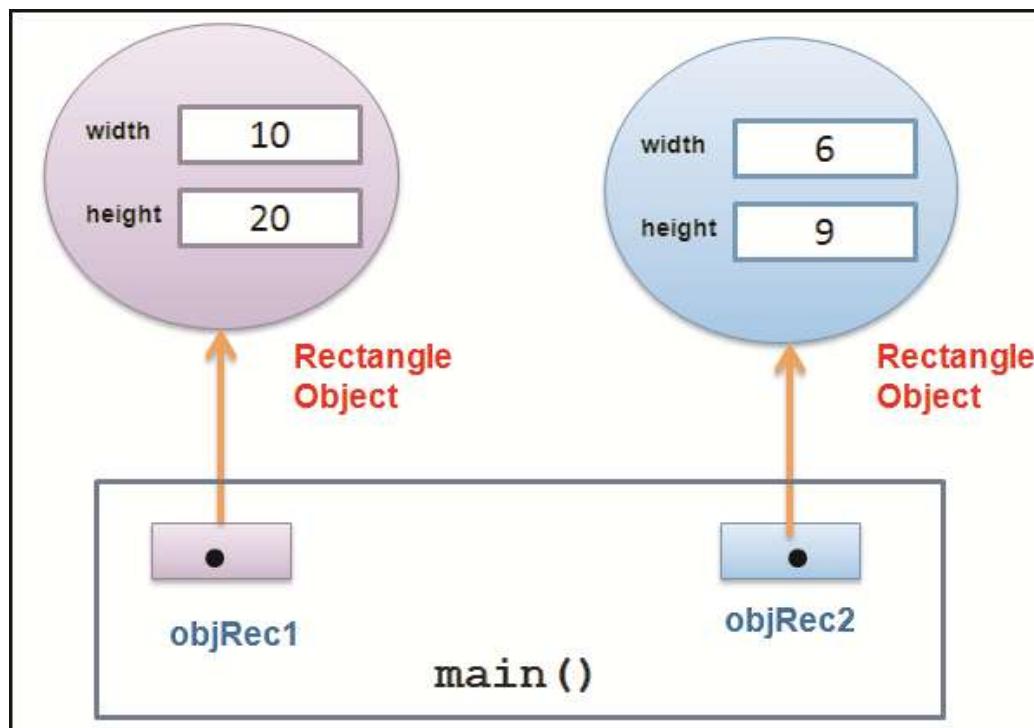
Rectangle2 Details
=====
Width: 6
Width: 9

BUILD SUCCESSFUL (total time: 1 second)
```

Parameterized Constructor 5-5



- Following figure displays both the instance of the class **Rectangle**.
- Each object contains its own copy of instance variables that are initialized through constructor:





- ◆ The memory comprises two components namely, stack and heap.

Stack

- It is an area in the memory which stores object references and method information.
- It stores parameters of a method and local variables.

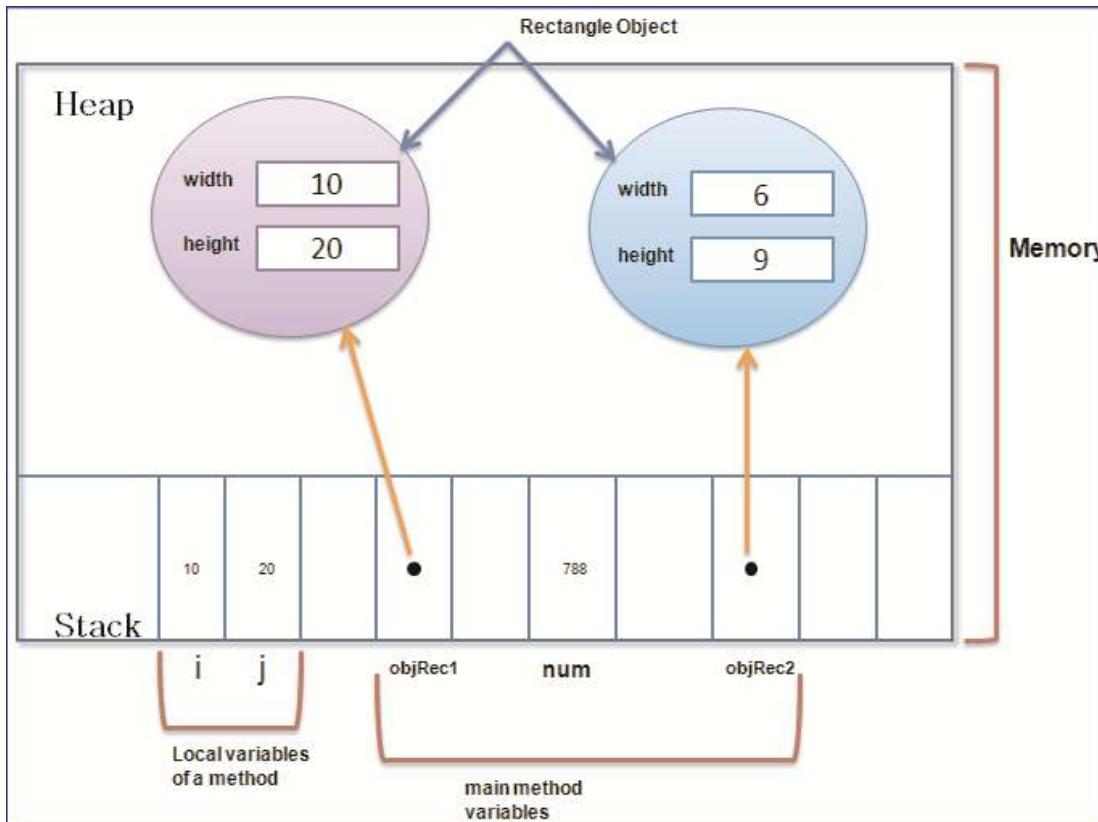
Heap

- It is area of memory deals with dynamic memory allocations.
- In Java, objects are allocated physical memory space on the heap at runtime, that is, whenever JVM executes the new operator.

Memory Management in Java 2-2



- Following figure shows the memory allocation for objects in stack and heap for **Rectangle** object:



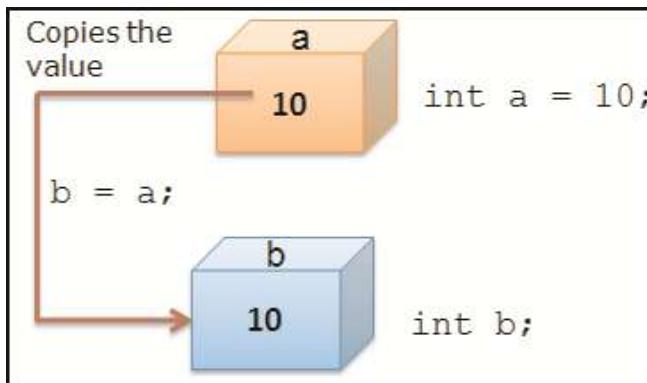
- The heap memory grows as and when the physical allocation is done for objects.
- Hence, JVM provides a garbage collection routine which frees the memory by destroying objects that are no longer required in Java program.

Assigning Object References 1-4



◆ Working with primitive data types:

- ❖ The value of one variable can be assigned to another variable using the assignment operator.
- ❖ For example, `int a = 10; int b = a;` copies the value from variable **a** and stores it in the variable **b**.
- ❖ Following figure shows assigning of a value from one variable to another:



◆ Working with object references:

- ❖ Similar to primitive data types, the value stored in an object reference variable can be copied into another reference variable.
- ❖ Both the reference variables must be of same type, that is, both the references must belong to the same class.

Assigning Object References 2-4



- Following code snippet demonstrates assigning the reference of one object into another object reference variable:

```
public class TestObjectReferences {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
  
        /* Instantiates an object of type Rectangle and stores  
         * its reference in the object reference variable, objRec1  
         */  
        Rectangle objRec1 = new Rectangle(10, 20);  
  
        // Declares a reference variable of type Rectangle  
        Rectangle objRec2;
```

- The **objRec1** points to the object that has been allocated memory and initialized to 10 and 20.
- The **objRec2** is an object reference variable that does not point to any object.

Assigning Object References 3-4



```
// Assigns the value of objRec1 to objRec2
objRec2 = objRec1;
System.out.println("\nRectangle1 Details");
System.out.println("=====");

/* Invokes the method that displays values of the
 * instance variables for object, objRec1
 */
objRec1.displayDimensions();
System.out.println("\nRectangle2 Details");
System.out.println("=====");
objRec2.displayDimensions();
}

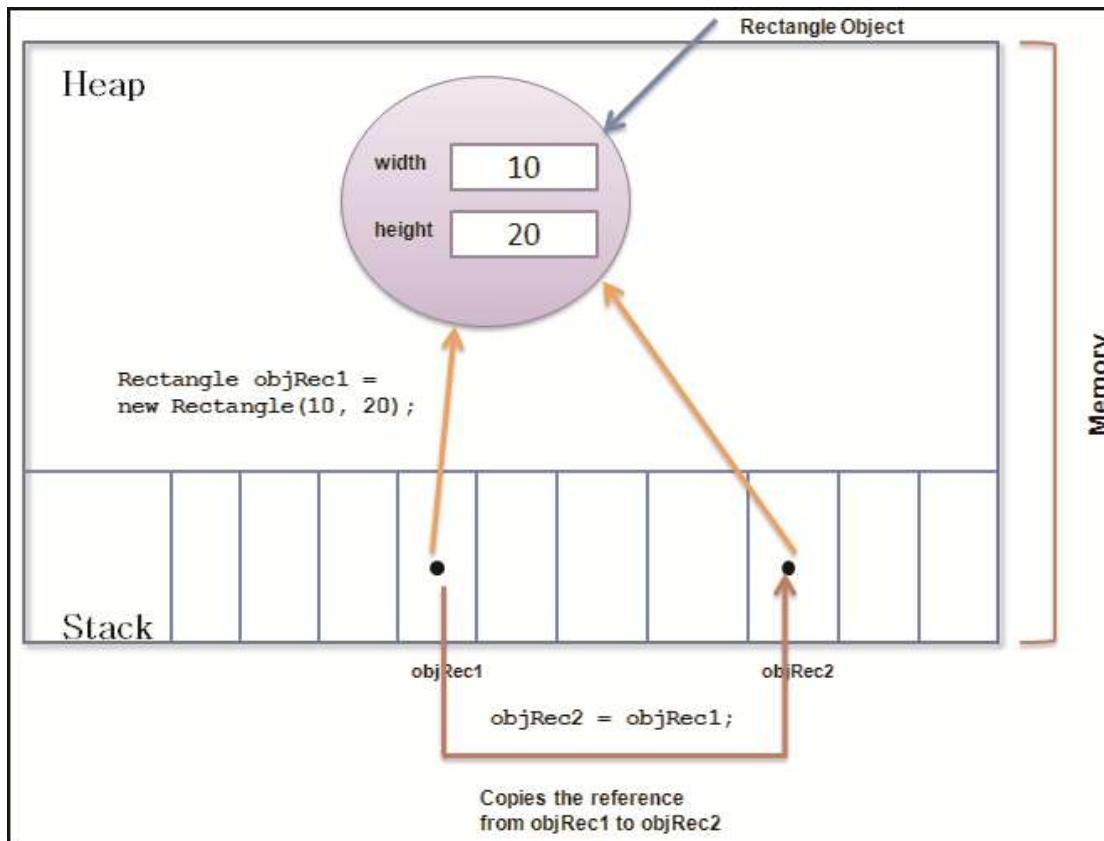
}
```

- ❖ The statement, **objRec2 = objRec1;** copies the address in **objRec1** into **objRec2**.
- ❖ Thus, the references are copied between the variables created on the stack without affecting the actual objects created on the heap.

Assigning Object References 4-4



- Following figure shows the assigning of reference for the statement,
`objRec2 = objRec1;`.





- ◆ In OOP languages, the concept of hiding implementation details of an object is achieved by applying the concept of encapsulation.

Encapsulation

- It is a mechanism which binds code and data together in a class.
- Its main purpose is to achieve data hiding within a class which means:
 - Implementation details of what a class contains need not be visible to other classes and objects that use it.
 - Instead, only specific information can be made visible to the other components of the application and the rest can be hidden.
- By hiding the implementation details about what is required to implement the specific operation in the class, the usage of operation becomes simple.



- ◆ In Java, the data hiding is achieved by using **access modifiers**.
- ◆ **Access Modifiers:**
 - ◆ Determine how members of a class, such as instance variable and methods are accessible from outside the class.
 - ◆ Decide the scope or visibility of the members.
 - ◆ Are of four types:

public

- Members declared as public can be accessed from anywhere in the class as well as from other classes.

private

- Members are accessible only from within the class in which they are declared.

protected

- Members to be accessible from within the class as well as from within the derived classes.

package (default)

- Allows only the public members of a class to be accessible to all the classes present within the same package.
- This is the default access level for all the members of the class.



- ◆ As a general rule in Java, the details and implementation of a class is hidden from the other classes or external objects in the application.
- ◆ This is done by making instance variables as private and instance methods as public.
- ◆ Following code snippet demonstrates the use of the concept of encapsulation in the class **Rectangle**:

```
public class Rectangle {  
    // Declares instance variables  
    private int width;  
    private int height;  
    /*  
     * Declares a no-argument constructor  
     */  
    public Rectangle() {  
        System.out.println("Constructor Invoked...");  
        width = 10;  
        height = 10;  
    }  
}
```

- ◆ The access specifiers of the instance variables, **width** and **height** are changed from default to **private** which means that the class fields are not directly accessible from outside the class.

Access Modifiers 3-3



```
/**  
 * Declares a parameterized constructor with two parameters  
 * @param wid  
 * @param heig  
 */  
  
public Rectangle (int wid, int heig) {  
    System.out.println("Parameterized Constructor Invoked...");  
    width = wid;  
    height = heig;  
}  
  
/**  
 * Displays the dimensions of the Rectangle object  
 */  
public void displayDimensions () {  
    System.out.println("Width: " + width);  
    System.out.println("Width: " + height);  
}  
}
```

- ◆ The access modifiers for the methods are changed to public.
- ◆ Thus, the users can access the class members through its methods without impacting the internal implementation of the class.

Types of Access Specifiers



- ◆ Following table shows the access level for different access specifiers:

Access Specifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
No modifier (default)	Y	Y	N	N
private	Y	N	N	N

- ◆ The first column states whether the class itself has access to its own data members.
- ◆ As can be seen, a class can always access its own members.
- ◆ The second column states whether classes within the same package as the owner class (irrespective of their parentage) can access the member.
- ◆ As can be seen, all members can be accessed except `private` members.
- ◆ The third column states whether the subclasses of a class declared outside this package can access a member.
- ◆ In such cases, `public` and `protected` members can be accessed.
- ◆ The fourth column states whether all classes can access a data member.

Rules for Access Control



- ◆ Java has rules and constraints for usage of access specifiers as follows:

While declaring members, a private access specifier cannot be used with abstract, but it can be used with final or static.

No access specifier can be repeated twice in a single declaration.

A constructor when declared private will be accessible in the class where it was created.

A constructor when declared protected will be accessible within the class where it was created and in the inheriting classes.

private cannot be used with fields and methods of an interface.

The most restrictive access level must be used that is appropriate for a particular member.

Mostly, a private access specifier is used at all times unless there is a valid reason for not using it.

Avoid using public for fields except for constants.



- ◆ They provide a way to create an object and initialize its fields.
 - ◆ They complement the use of constructors to initialize objects.
 - ◆ There are two approaches to initialize the fields or instance variables of the newly created objects:
 - ◆ Using instance variable initializers
 - ◆ Using initialization block
- ◆ **Instance Variable Initializers:**
- ◆ In this approach, you specify the names of the fields and/or properties to be initialized, and give an initial value to each of them.
 - ◆ Following code snippet demonstrates a Java program that declares a class, **Person** and initializes its fields:

```
public class Person {  
    private String name = "John";  
    private int age = 12;  
  
    /**  
     * Displays the details of Person object  
     */
```

Object Initializers 2-6



```
void displayDetails() {  
    System.out.println("Person Details");  
    System.out.println("=====");  
    System.out.println("Person Name: " + name);  
}  
}
```

- ❖ The instance variables **name** and **age** are initialized to values '**John**' and **12** respectively.
- ❖ Following code snippet shows the class with main () method that creates objects of type **Person**:

```
public class TestPerson {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        Person objPerson1 = new Person();  
        objPerson1.displayDetails();  
    }  
}
```

Object Initializers 3-6



- ◆ The code creates an object of type **Person** and invokes the method to display the details.
- ◆ Following figure shows the output of the code:

The screenshot shows the 'Output - Session 6 (run)' window from a Java IDE. It displays the following text:
run:
Person Details

Person Name: John
Person Age: 12
BUILD SUCCESSFUL (total time: 0 seconds)

- ◆ **Initialization Block:**
 - ◆ In this approach, an initialization block is specified within the class.
 - ◆ The initialization block is executed before the execution of constructors during an object initialization.

Object Initializers 4-6



- Following code snippet demonstrates the class **Account** with an initialization block:

```
public class Account {  
    private int accountID;  
    private String holderName;  
    private String accountType;  
  
    /**  
     * Initialization block  
     */  
    {  
        accountID = 100;  
        holderName = "John Anderson";  
        accountType = "Savings";  
    }  
}
```

- In the code, the initialization blocks initializes the instance variables or fields of the class.

Object Initializers 5-6



```
/**
 * Displays the details of Account object
 */
public void displayAccountDetails() {
    System.out.println("Account Details");
    System.out.println("=====");
    System.out.println("Account ID: " + accountID + "\nAccount
Type: " + accountType);
}
```

- Following code snippet shows the code with main () method to initialize the **Account** object through initialization block:

```
public class TestInitializationBlock {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Account objAccount = new Account();
        objAccount.displayAccountDetails();
    }
}
```

Object Initializers 6-6



- Following figure shows the output of the code:

The screenshot shows the 'Output - Session 6 (run)' window from a Java IDE. The window title is 'Output - Session 6 (run)'. On the left, there are four icons: a green arrow for running, a yellow arrow for stopping, a brown square for closing, and a circular icon with a gear for other options. The main pane displays the following text:
run:
Account Details
=====

Account ID: 100
Account Type: Savings
BUILD SUCCESSFUL (total time: 1 second)



- ◆ The class is a logical construct that defines the shape and nature of an object.
- ◆ Objects are the actual instances of the class and are created using the new operator. The new operator instructs JVM to allocate the memory for the object.
- ◆ The members of a class are fields and methods. Fields define the state and are referred to as instance variables, whereas methods are used to implement the behavior of the objects and are referred to as instance methods.
- ◆ Each instance created from the class will have its own copy of the instance variables, whereas methods are common for all instances of the class.
- ◆ Constructors are methods that are used to initialize the fields or perform startup operations only once when the object of the class is instantiated.
- ◆ The heap area of memory deals with the dynamic memory allocations for objects, whereas the stack area holds the object references.
- ◆ Data encapsulation hides the instance variables that represents the state of an object through access modifiers. The only interaction or modification on objects is performed through the methods.

Implementing OOP in Java

Session 4

Data Abstraction

- Data Abstraction is the process of identifying and grouping attributes and actions related to a particular entity as relevant to the application at hand
- Advantages
 - It focuses on the problem
 - It identifies the essential characteristics and actions
 - It helps to eliminate unnecessary detail

Method Overloading

- is defining different versions of a method in class, and the compiler will automatically select the most appropriate one based on the parameters supplied
- Method overloading is achieved by passing:
 - Different types of parameters
 - Different number of parameters
 - Different sequence of parameters

```
void display(); // Display methods
```

```
void display(int one, char two);
```

```
void display(int a, int b, int c);
```

```
void display(char one, int two);
```

Advantages

- Eliminates use of different method names for the same operation
- Helps to understand and debug code easily
- Maintaining code is easier

Overloading using different Data Types

```
int square(int);
float square(bool);
double square(char);
```

The compiler can distinguish between overloaded methods with same number of arguments provided their type is different

Overloading using Different Number/Sequence of Arguments

```
int square(int, char)
```

```
int square(char,int)
```

```
int square(int,int, int)
```

```
int asq=square(3,'a')
```

```
int bsq=square(1,2,3)
```

```
int asq=square('a',3)
```

At compile time, compiler compares the types of actual arguments with the types of formal arguments of all methods called square

Construction

- The process of bringing an object into existence is called Construction
- A Constructor
 - Allocates memory
 - Initializes attributes, if any
 - Enables access to attributes and methods

Destruction

- The process of deleting an object is called Destruction
- A Destructor
 - Freed allocated space
 - Disables access to attributes and methods

Constructors

- They are special types of methods in a class.
- They are called every time an object is created.
- They are generally used for initialization.
- They have the same name as the class.
- They return no value.

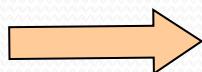
Constructors

A constructor is a special method for automatic initialization of an object

General syntax for constructors is:

```
class username{  
    .  
    username() //constructor  
    {  
        .....  
        .... //Code written  
    }  
};
```

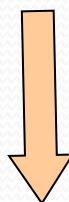
Default Constructor in Java



Defines constructor



**Default
constructor not
used**

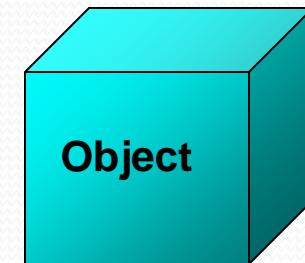


No

If we do not define any constructor then Java invokes default constructor for the class



**Constructor for the
class**



Default Constructor in Java

```
class Sdate {  
    int month;  
    int day;  
    int year;  
    Sdate() //Constructor  
    {  
        month=11;  
        day=27;  
        year=1969;  
    }  
    public static void main(String args[])  
    {  
        Sdate S1,S2;  
        S1=new Sdate();  
        S2=new Sdate();  
    }  
}
```

Parameterized Constructor

Constructors defined with parameters are known as parameterized constructor

```
class Sdate {  
    int month;  
    int day;  
    int year;  
    Sdate(int m,int d,int y){  
        month=m;  
        day=d;  
        year=y;  
    }  
    public static void main(String args[]){  
        Sdate S1,S2;  
        S1=new Sdate(11,27,1969);  
        S2=new Sdate(3,3,1973);  
    }  
}
```

Static Constructors

- They will be called only once before the first object is created.
- A static constructor is used to initialize any static data, or to perform a particular action that needs to be performed only once.
- They can be declared in the same way as a static method is declared.
- They cannot have any parameters.

```
...
static semester()
{
    // Static Constructor Implementation
}
...
...
```

Static Constructors

- In Java, the concept of a **static constructor** does not exist in the same way as in some other programming languages. However, you can achieve similar functionality using **static initialization blocks**.

```
class Example {  
    static int value;  
  
    // Static initialization block  
    static {  
        value = 42; // Initialize static variable  
    }  
}
```

Private Constructors

- **private constructor** is a constructor that cannot be accessed from outside the class in which it is defined.
- Private constructors are used **to prevent creating instances of a class when there are no instance fields or methods**, such as the Math class, or when a method is called to obtain an instance of a class. If all the methods in the class are static, consider making the complete class static.

```
...
class AllStatic
{
private AllStatic()
{
    //Private Constructor Implementation
}
}
```

Copy Constructor

- A **copy constructor** is a special constructor used to create a new object as a copy of an existing object.
- It takes an instance of the same class as a parameter and initializes the new object with the values of the existing object's attributes.

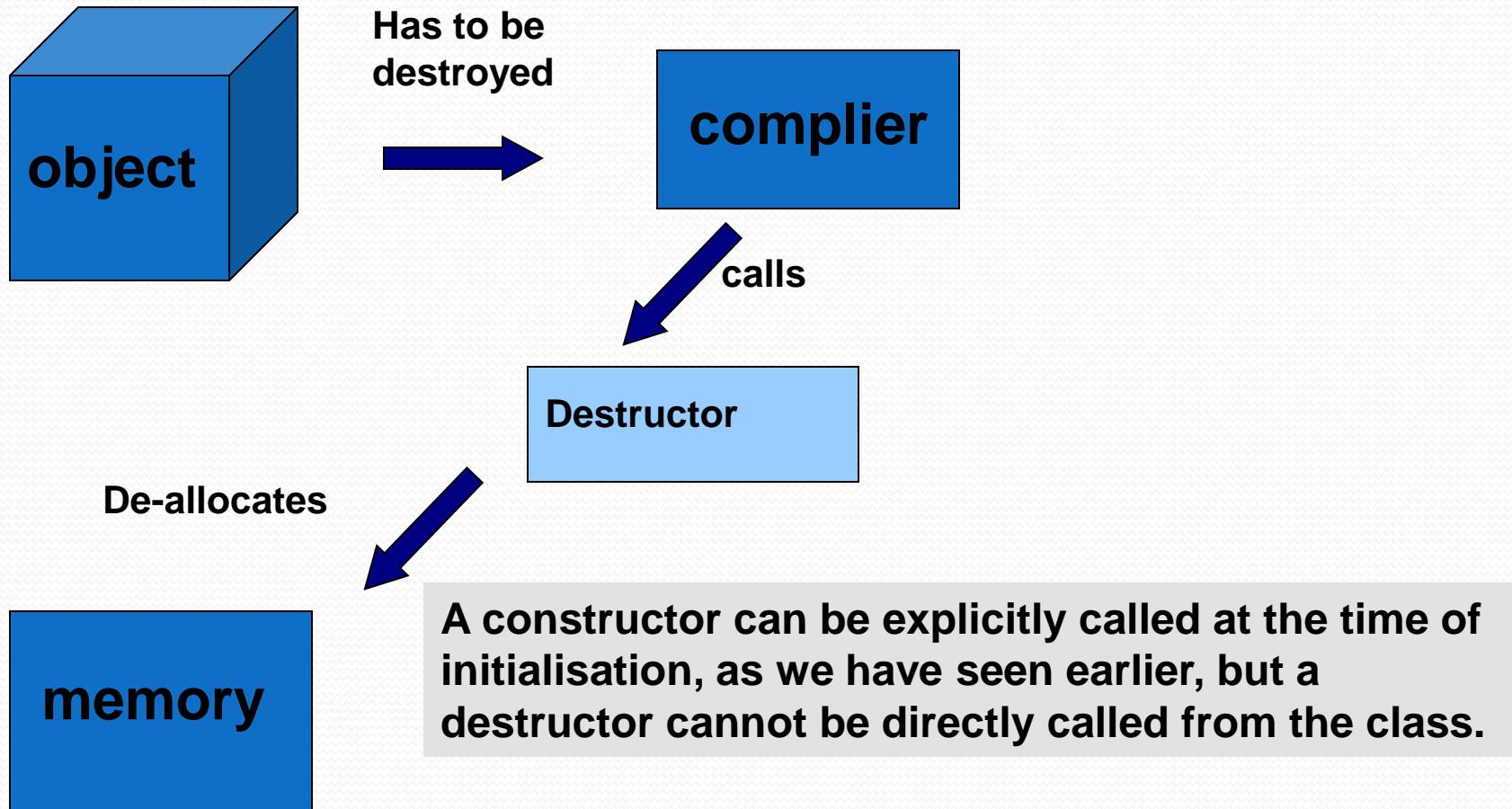
```
class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    // Copy constructor  
    public Person(Person other) {  
        this.name = other.name;  
        this.age = other.age;  
    }  
  
    // Getters for demonstration  
    public String getName() {  
        return name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
  
        Person original = new Person("Alice", 30);  
  
        Person copy = new Person(original);  
  
        System.out.println("\t\t Original");  
        System.out.println("Name: " + original.getName());  
        System.out.println("Age: " + original.getAge());  
  
        System.out.println("\t\t Copy");  
        System.out.println("Name: " + copy.getName());  
        System.out.println("Age: " + copy.getAge());  
    }  
}
```

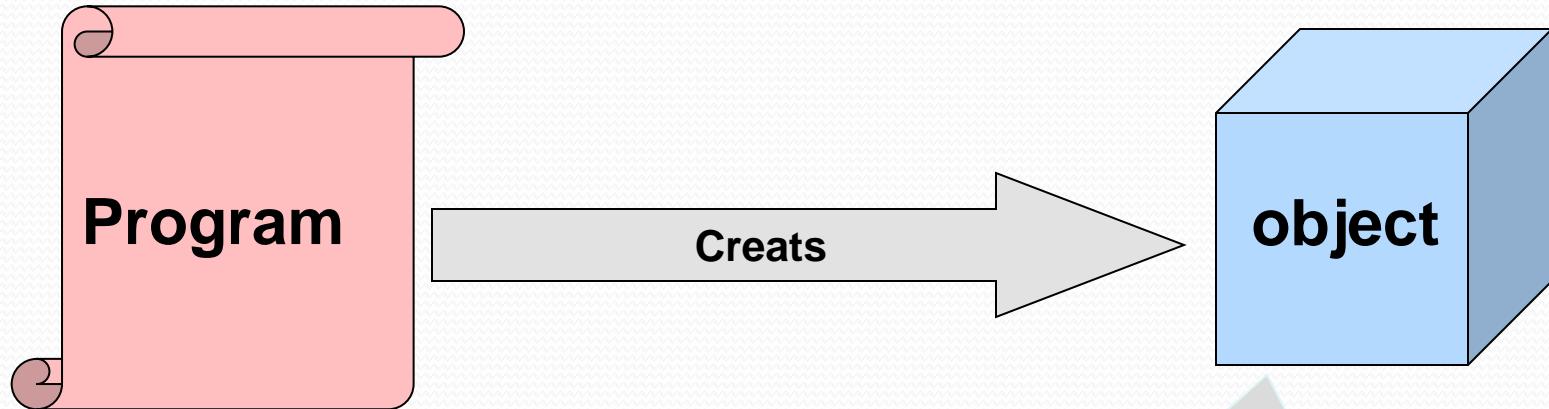
Destructors

- They are called by **Garbage Collector** in Java
- Garbage Collector frees memory by destroying objects that are no longer required/referenced.

Destructors



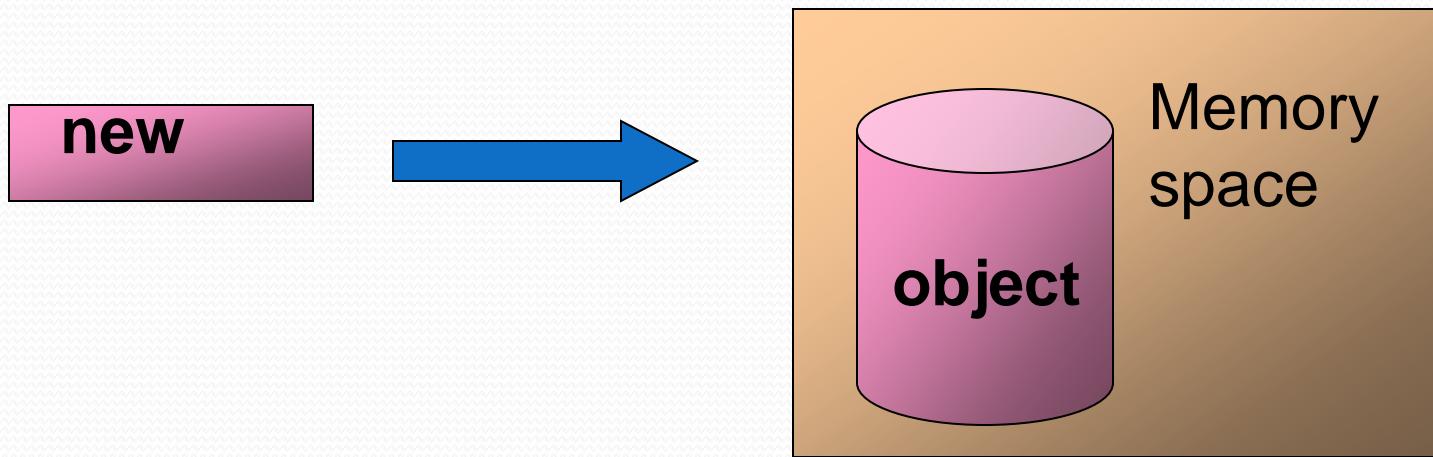
Memory allocation for an Object



It is useful to create a new object that will exist only as long as it is needed, otherwise huge memory will be occupied by all these unused objects.

According to the definition given in the Class.

Allocating Memory



```
int[] p = new int[3];
```

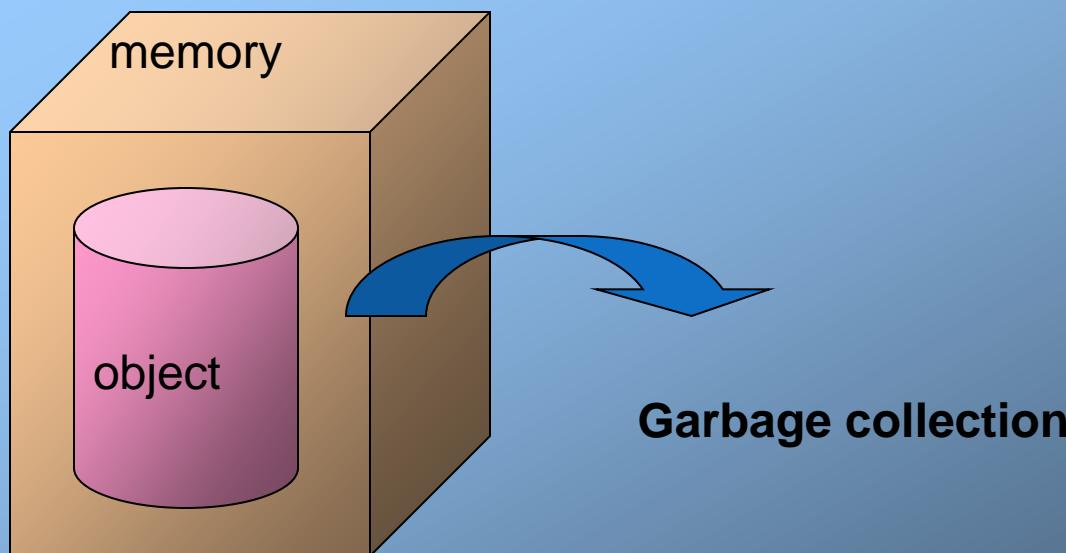
Syntax:

Student stu_ptr=new Student()

Allocating Memory

```
class Sdate {  
    int[] date;  
    Sdate(int m,int d,int y) {  
        date = new int[3];  
        date[0]=m;  
        date[1]=d;  
        date[2]=y;  
    }  
}  
  
class Newdate {  
    public static void main(String args[]) {  
        Sdate S1,S2;  
        S1=new Sdate(11,27,1967);  
        S2=new Sdate(4,3,1973);  
    }  
}
```

De-allocating Memory



Garbage Collector (1)

- The working of garbage collector is as follows:
 - An object with a destructor defined is added to a list of objects that require destruction.
 - Garbage collector starts on its rounds and checks if there are objects that have no references.
 - If an object is found and if the name of the object does not appear in the finalizer list then it is cleared up instantly.
 - If the name of the object appears on the list of objects that require finalization, it is marked as “Ready for Finalization”.

Garbage Collector (2)

- When the garbage collection is complete then the finalizer thread is called, which goes about calling the finalize methods of all objects that have been marked as “Ready for Finalization”.
- After the finalization of an object has occurred, it is removed from the list of objects, which require finalization.
- Since the object is no longer on the finalizer list, it gets cleaned up when the next garbage collection is done.

Garbage Collector (3)

- Objects with finalize method take up more resources as they stay for a longer period in memory even when they are not required.
- Finalization takes place as a separate thread, again eating into the resources.

Finalize method

```
Protected void finalize()  
{  
//finalization code  
}
```

java



Finalization
mechanism

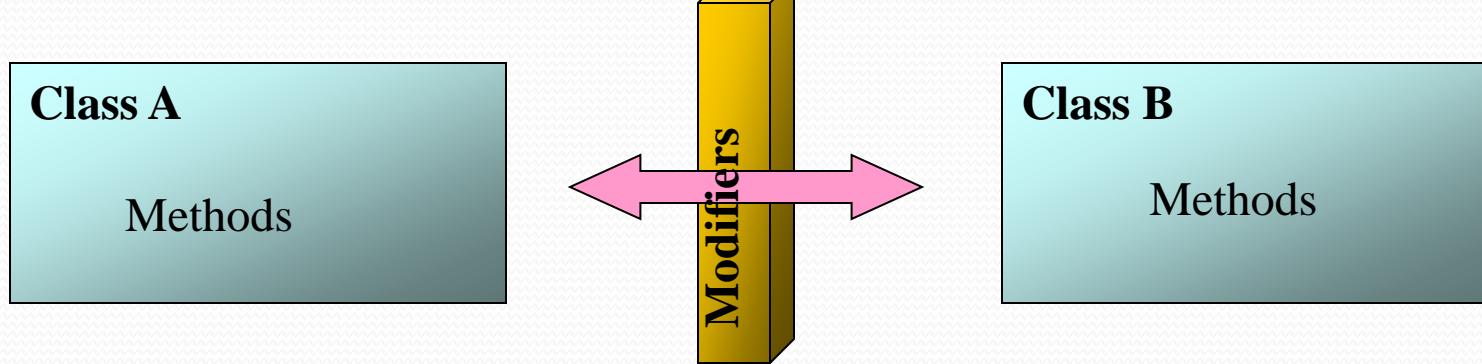


Defines finalize()
method in our class

Modifiers

- Java is a tightly encapsulated language.
- Java provides a set of access specifiers such as **public**, **private**, **protected**, and **default** that help to restrict access to class and class members.
- Java provides additional field and method modifiers to further restrict access to the members of a class to prevent modification by unauthorized code.

Modifiers - I



Modifiers restrict access to variables by other classes

They are keywords that give additional meaning to variables, code, and classes

Modifiers enhances the encapsulation features of OOP

Modifiers - II

```
class Stack
{
    int[] stak = new int[10];
    int size ;
    int top = -1;
    Stack(int s)
    {
        top = -1;
        size=s;
    }
    void push(int it)
    {
        if(top==this.size-1)

            System.out.println("Stack is full ");
    }
}
```

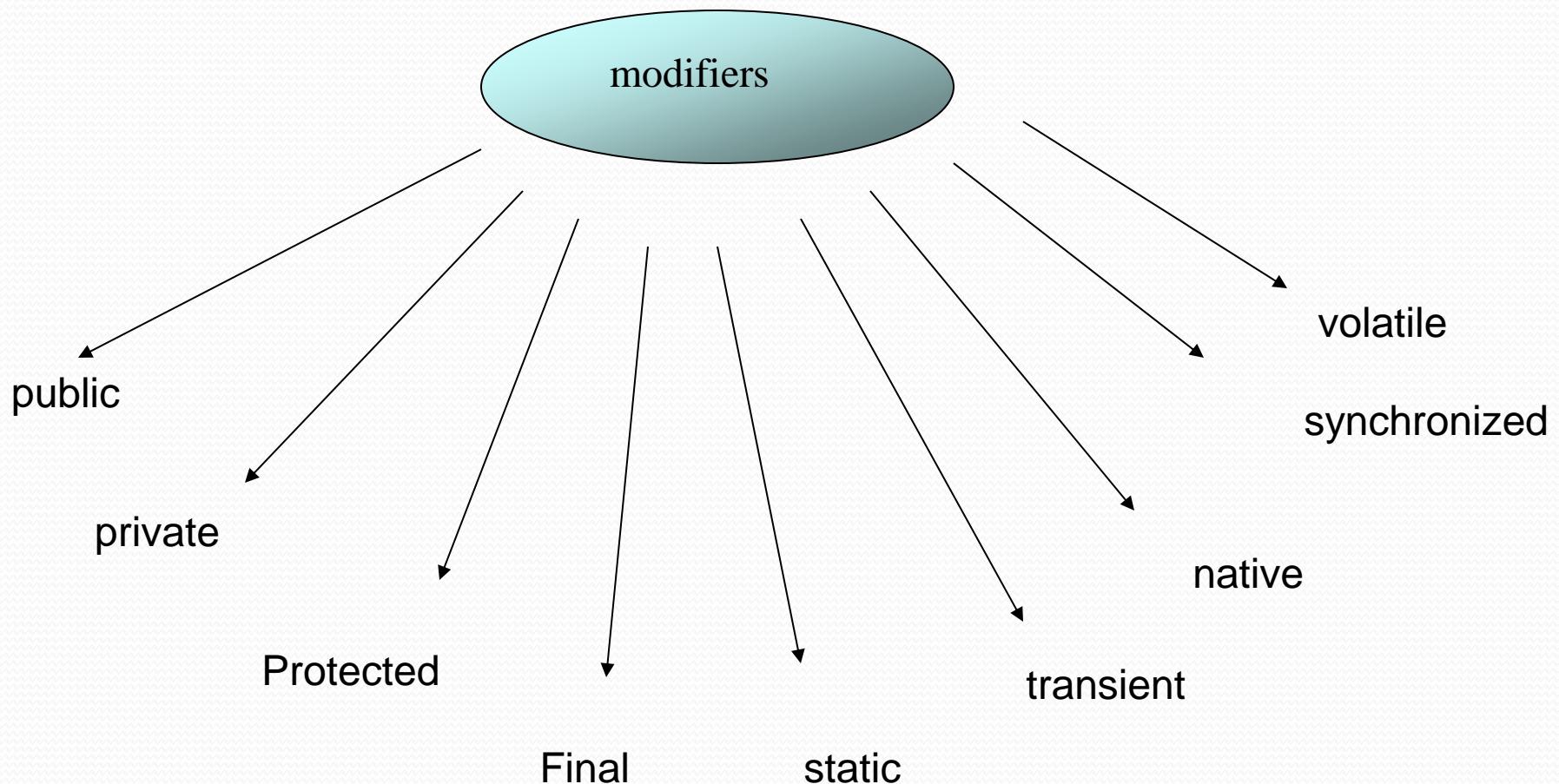
Modifiers - III

```
        else
            stack[++top] = it;
    }
int pop()
{
    if (top < 0)
    {
        System.out.println("Stack underflow   ");
        return 0 ;
    }
    else
        return stack[top--];
}
```

Modifiers - IV

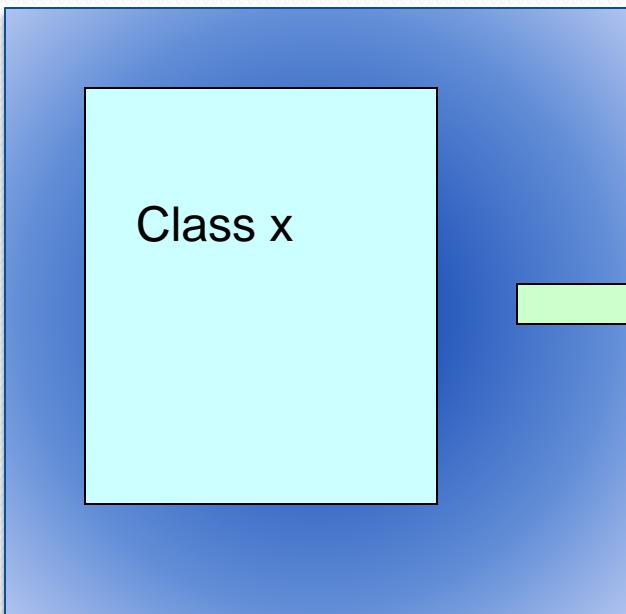
```
class Test {  
    Stack S = new Stack(4);  
    ..... . . .  
    System.out.println("The stack size " +  
        S.size);  
    ..... }
```

Modifiers - V

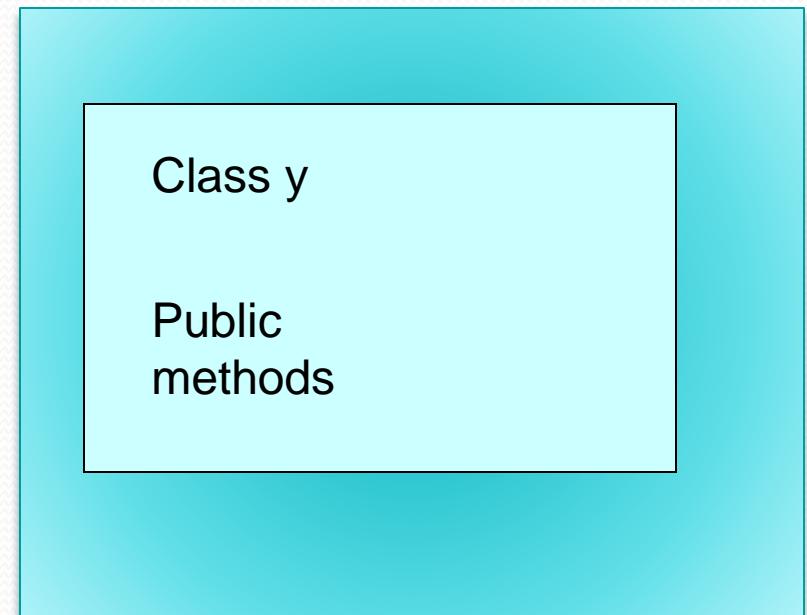


Public

Application A



Application B



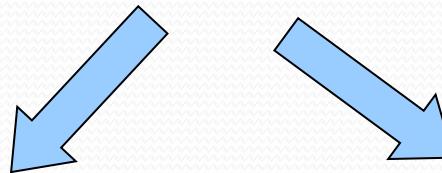
Protected

Class A
Protected methods
and variables

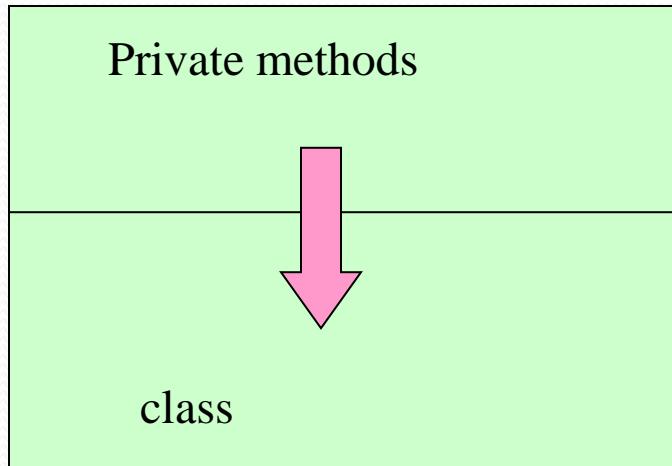
Accessible to sub
classes

Sub-class 1

Sub-class 2



Private



Methods and data members accessible only to the members of the same class

Class Specification	Private	Default	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package- different class	No	Yes	Yes	Yes
Different package- sub class	No	No	Yes	Yes
Different package- different class	No	No	No	Yes

Static Variables

- Lifetime of static variable is throughout the program life
- If static variables are not explicitly initialized then they are initialized to 0 of appropriate type

Static Data Member

Definition

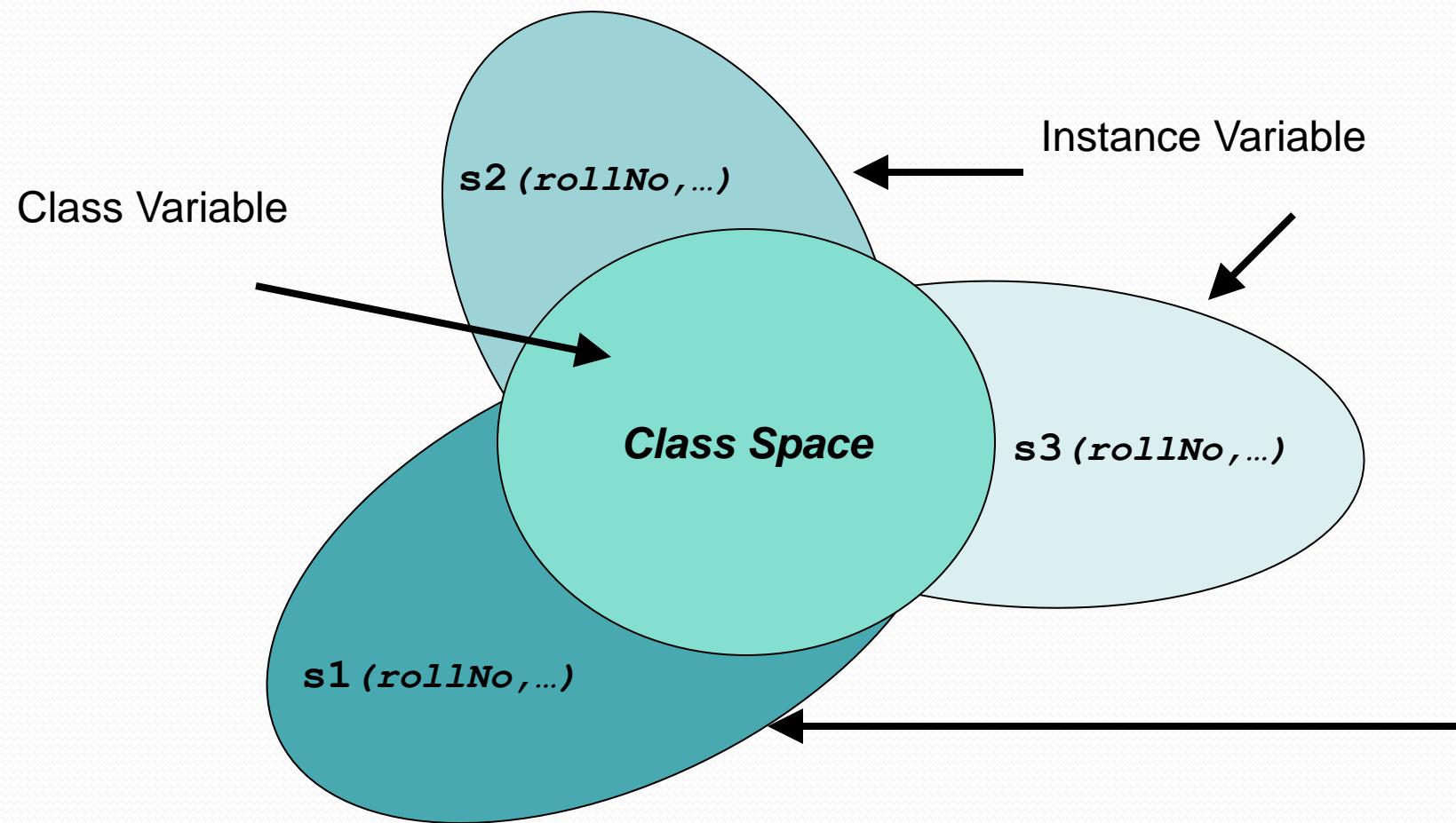
“A variable that is part of a class, yet is not part of an object of that class, is called static data member”

Static Data Member

- They are shared by all instances of the class
- They do not belong to any particular instance of a class

Class vs. Instance Variable

- Student s1, s2, s3;



Static Data Member (Syntax)

- Keyword static is used to make a data member static

```
class ClassName{  
...  
static DataType VariableName;  
}
```

Initializing Static Data Member

- Static data members should be initialized once at file scope
- They are initialized at the time of definition

Life of Static Data Member

- They are created even when there is no object of a class
- They remain in memory even when all objects of a class are destroyed

Static - I

```
class Cvar

{
    static String name="Aladdin";
    //... constructor
    static void showName() {
        ...
        System.out.println("Static name:" + name);
    }
}
```

Static - II

```
class Staticvar
{
    static int s = 20;
    static void print()
    {
        System.out.println("From the class : " + s);
    }
}
class Display
{
    public static void main(String args[])
    {
        Staticvar.print();
        System.out.println("From outside the class : " +
        Staticvar.s);
    }
}
```

Uses

- They can be used to store information that is required by all objects, like global variables

Example

- Develop Student class such that one can know the number of student created in a system

Example

```
class Student{  
...  
    public static int noOfStudents=0;  
    public Student(){  
        noOfStudents++;  
    }  
...  
}
```

Example

```
public static void main(String[] args) {  
  
    System.out.println(Student.noOfStudents);  
    Student s = new Student();  
    System.out.println(Student.noOfStudents);  
    Student s1 = new Student();  
    Student s2= new Student();  
    Student s3 = new Student();  
    Student s4 = new Student();  
  
    System.out.println(Student.noOfStudents);  
}
```

Output:

0
1
5

Problem

- noOfStudents is accessible outside the class
- Bad design as the local data member is kept public

Static Member Function

Definition:

“The function that needs access to the members of a class, yet does not need to be invoked by a particular object, is called static member function”

Static Member Function

- They are used to access static data members
- Access mechanism for static member functions is same as that of static data members
- They cannot access any non-static members

Example

```
class Student{
    private static int noOfStudents;
    int rollNo;
public static int getTotalStudent() {
    return noOfStudents;
}
}

Class Display{
public static void main(String [] args) {
    int i = Student.getTotalStudents();
}
}
```

Accessing non static data members

```
static int getTotalStudents() {  
    return rollNo;  
}  
.....  
public static void main(String [] args) {  
    int i = Student.getTotalStudents();  
    /*Error: */  
}
```

Final

‘Final’ modifier when used with :

Variable

Indicates that once a value is assigned, it cannot be changed

Method

Indicates that method body cannot be overridden

Class

Indicates that this class cannot be inherited

‘final’ Modifier

The **final** modifier is used when modification of a class or data member is to be restricted.

The **final** modifier can be used with a variable, method, and class.

Final Variable

A variable declared as **final** is a constant whose value cannot be modified.

A **final** variable is assigned a value at the time of declaration.

A compile time error is raised if a **final** variable is reassigned a value in a program after its declaration.

- Following code snippet shows the creation of a **final** variable:

```
final float PI = 3.14;
```

- ◆ The variable **PI** is declared **final** so that its value cannot be changed later.

'final' Modifier

Final Method

A method declared `final` cannot be overridden in a Java subclass.

The reason for using a `final` method is to prevent subclasses from changing the meaning of the method.

A `final` method is commonly used to generate a random constant in a mathematical application.

- Following code snippet depicts the creation of a `final` method:

```
final float getCommission(float sales){  
    System.out.println("A final method. . .");  
}
```

- ◆ The method **getCommission()** can be used to calculate commission based on monthly sales.
- ◆ The implementation of the method cannot be modified by other classes as it is declared as final.
- ◆ A final method cannot be declared abstract as it cannot be overridden.

'final' Modifier

Final Class

A class declared `final` cannot be inherited or subclassed.

Such a class becomes a standard and must be used as it is.

The variables and methods of a class declared `final` are also implicitly `final`.

- The reason for declaring a class as `final` is to limit extensibility and to prevent the modification of the class definition.
- Following code snippet shows the creation of a `final` class:

```
public final class Stock {  
    ...  
}
```

- ◆ The class **Stock** is declared `final`.
- ◆ All data members within this class are implicitly `final` and cannot be modified by other classes.

Native

- The ‘*native*’ modifier indicates that a method body has been written in a language other than Java, like C or C++
- To implement a native method, you typically use the Java Native Interface (JNI).
- JNI provides a way for Java code running in the Java Virtual Machine (JVM) to call and be called by native applications and libraries written in other languages like C or C++.
- Native methods are platform-dependent.

```
native void codeSomeWhere ()
{
    // C / C++ Code
}
```

‘native’ Modifier

The `native` modifier is used only with methods.

It indicates that the implementation of the method is in a language other than Java such as C or C++.

Constructors, fields, classes, and interfaces cannot use this modifier.

The methods declared using the `native` modifier are called native methods.

The Java source file typically contains only the declaration of the native method and not its implementation.

The implementation of the method exists in a library outside the JVM.

- Before invoking a native method, the library that contains the method implementation must be loaded by making the following system call:

```
System.loadLibrary("libraryName");
```

- To declare a native method, the method is preceded with the `native` modifier.
- The implementation is not provided for the method. For example,
`public native void nativeMethod();`

‘native’ Modifier

- After declaring a native method, a complex series of steps are used to link it with the Java code.
 - Following code snippet demonstrates an example of loading a library named NativeMethodDefinition containing a native method named nativeMethod ():

```
class NativeModifier {  
    native void nativeMethod(); // declaration of a native method  
  
    // static code block  
    static {  
        System.loadLibrary("NativeMethodDefinition");  
    }  
  
    public static void main(String[] args) {  
        NativeModifier objNative = new NativeModifier(); // line1  
        objNative.nativeMethod(); // line2  
    }  
}
```

'native' Modifier

- Notice that a static code block is used to load the library.
- The static keyword indicates that the library is loaded as soon as the class is loaded.
- This ensures that the library is available when the call to the native method is made. The native method can be used in the same way as a non-native method.
- Native methods allow access to existing library routines created outside the JVM.
- However, the use of native methods introduces two major problems.

Impending security risk

- The native method executes actual machine code, and therefore, it can gain access to any part of the host system.
- That is, the native code is not restricted to the JVM execution environment.
- This may lead to a virus infection on the target system.

Loss of portability

- The native code is bundled in a DLL, so that it can be loaded on the machine on which the Java program is executing.
- Each native method is dependent on the CPU and the OS.
- This makes the DLL inherently non-portable.
- This means, that a Java application using native methods will run only on a machine in which a compatible DLL has been installed.

Threading in Java

- **Threading** allows concurrent execution of two or more threads, enabling a program to perform multiple tasks simultaneously.
- Threads are lightweight sub-processes that share the same memory space, making them efficient for tasks like handling multiple user requests, performing background operations, or improving the responsiveness of applications.
- Threads can have priorities ranging from 1 (lowest) to 10 (highest). Use `setPriority(int priority)` to set a thread's priority.

Thread:

A thread is a single sequential flow of control within a program. Java provides built-in support for threading through the Thread class and the Runnable interface.

Main Thread:

When a Java program starts, the JVM creates the main thread, which executes the main() method.

Multithreading:

The ability of a program to manage multiple threads concurrently.

Thread Lifecycle:

A thread goes through various states during its lifecycle:

New: Thread is created but not started.

Runnable: Thread is ready to run, waiting for CPU time.

Running: Thread is executing.

Blocked/Waiting: Thread is waiting for a resource or another thread.

Terminated: Thread has finished execution.

Example

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Thread is running.");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        MyThread t1 = new MyThread();  
        t1.start(); // Starts the thread  
    }  
}
```

'volatile' Modifier

The `volatile` modifier allows the content of a variable to be synchronized across all running threads.

A thread is an independent path of execution of code within a program.
Many threads can run concurrently within a program.

The `volatile` modifier is applied only to fields.

Constructors, methods, classes, and interfaces cannot use this modifier.

The `volatile` modifier is not frequently used.

While working with a multithreaded program, the `volatile` keyword is used.

'volatile' Modifier

When multiple threads of a program are using the same variable, in general, each thread has its own copy of that variable in the local cache.

In such a case, if the value of the variable is updated, it updates the copy in the local cache and not the main variable present in the memory.

The other thread using the same variable does not get the updated value.

- To avoid this problem, a variable is declared as `volatile` to indicate that it will not be stored in the local cache.
- Whenever a thread updates the values of the variable, it updates the variable present in the main memory.
- This helps other threads to access the updated value.
- For example,

```
private volatile int testValue; // volatile variable
```

Synchronized

- In Java, the synchronized keyword is used to control access to a block of code or a method by multiple threads.
- You can declare an entire method as synchronized or synchronize a specific block of code within a method.

```
public synchronized void synchronizedMethod()  
{ // Critical section code }
```

```
public void someMethod()  
{ synchronized (this)  
    { // Critical section code }  
}
```

'transient' Modifier

When a Java application is executed, the objects are loaded in the Random Access Memory (RAM).

Objects can also be stored in a persistent storage outside the JVM so that it can be used later.

This determines the scope and life span of an object.

The process of storing an object in a persistent storage is called serialization.

For any object to be serialized, the class must implement the Serializable interface.

If transient modifier is used with a variable, it will not be stored and will not become part of the object's persistent state.

The transient modifier is useful to prevent security sensitive data from being copied to a source in which no security mechanism has been implemented.

The transient modifier reduces the amount of data being serialized, improves performance, and reduces costs.

‘transient’ Modifier

- The transient modifier can only be used with instance variables.
- It informs the JVM not to store the variable when the object, in which it is declared, is serialized.
- Thus, when the object is stored in persistent storage, the instance variable declared as transient is not persisted.
- Following code snippet depicts the creation of a transient variable:

```
class Circle {  
  
    transient float PI; // transient variable that will not persist  
    float area; // instance variable that will persist  
  
}
```

Modifiers Snap Shot

Modifier	Method	Variable	Class
Public	Yes	Yes	Yes
Private	Yes	Yes	Yes(Nested Classes)
Protected	Yes	Yes	Yes(Nested Classes)
Abstract	Yes	No	Yes
Final	Yes	Yes	Yes
Native	Yes	No	No

this Pointer

```
class Student{  
    private int rollNo;  
    .....  
    public int getRollNo(){...}  
    public void setRollNo(int aRollNo)  
    {...}  
}
```

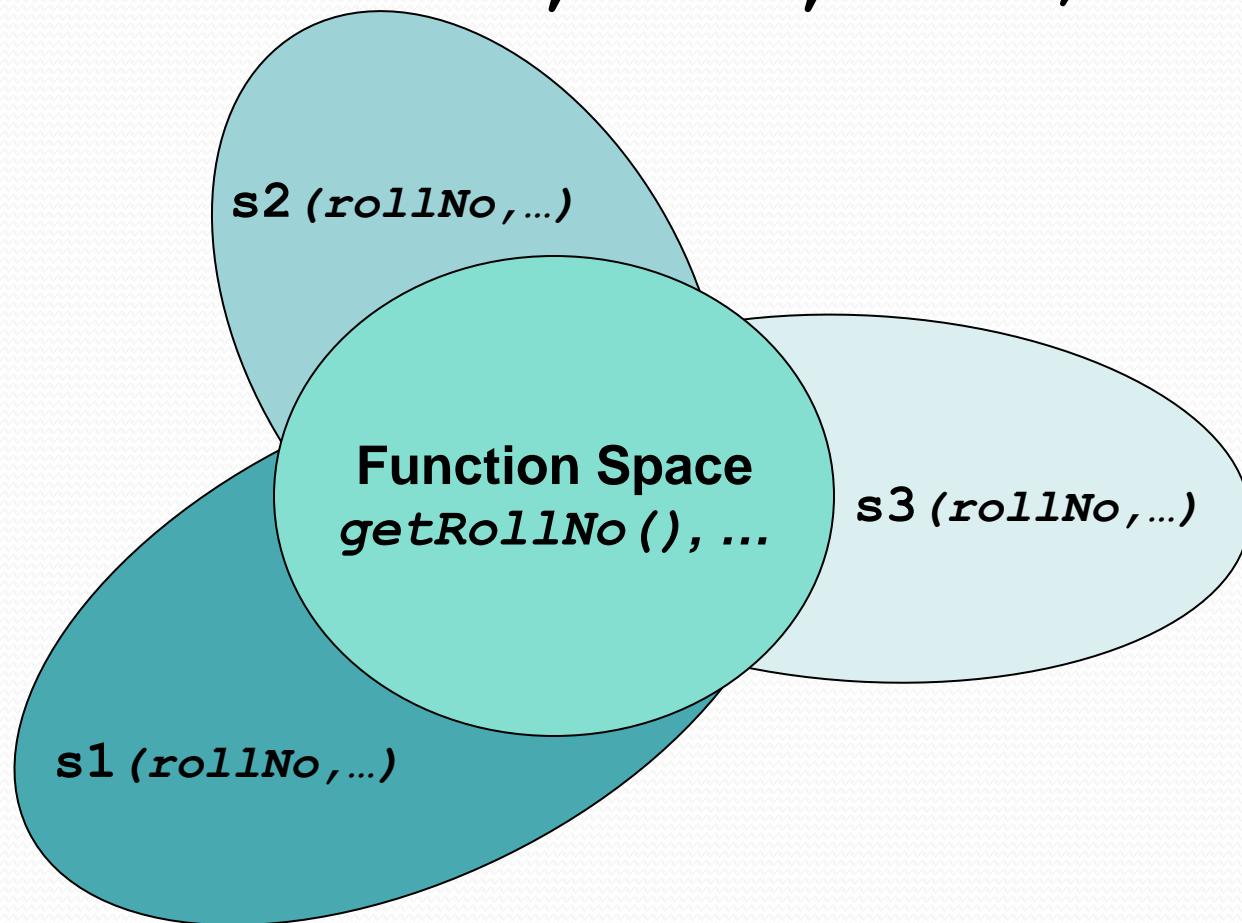
this Pointer

- The compiler reserves space for the functions defined in the class
- Space for data is not allocated (*since no object is yet created*)

Function Space
`getRollNo()`, ...

this Pointer

- Student s1, s2, s3;



this Pointer

- Function space is common for every object
- Whenever a new object is created:
 - Memory is reserved for variables only
 - Previously defined functions are used over and over again

this Pointer

- Memory layout for objects created:



Function Space
getRollNo(), ...

- How does the functions know on which object to act?

this Pointer

- Address of each object is passed to the calling function
- This address is dereferenced by the functions and hence they act on correct objects

s1 rollNo, ... address	s2 rollNo, ... address	s3 rollNo, ... address	s4 rollNo, ... address
-------------------------------------	-------------------------------------	-------------------------------------	-------------------------------------

- The variable containing the “self-address” is called this pointer

Passing *this* Pointer

- Whenever a function is called the *this* pointer is passed as a parameter to that function
- Function with n parameters is actually called with $n+1$ parameters

Example

```
void setName(String a)
```

is internally represented as

```
void setName(String a, const Student * this)
```

Compiler Generated Code

```
Student () {  
    rollNo = 0;  
}
```

```
Student () {  
    this.rollNo = 0;  
}
```

Packages in Java

Session 5

Packages 1-2

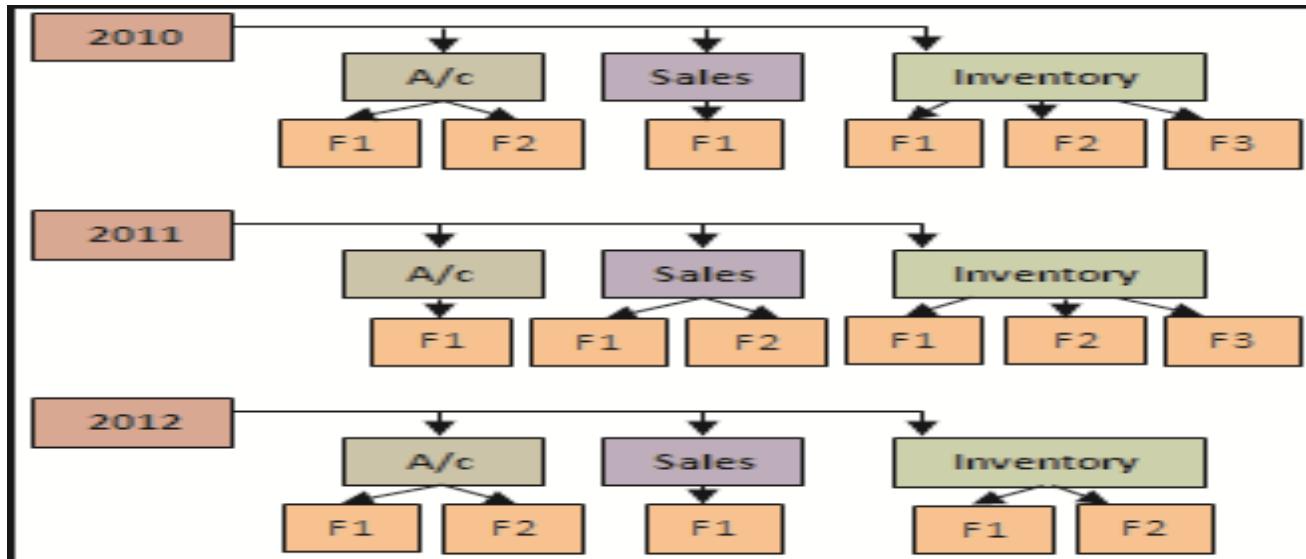
Consider a situation where a user has about fifty files of which some are related to sales, others are related to accounts, and some are related to inventory.

Also, the files belong to different years. All these files are kept in one section of a cupboard.

Now, when a particular file is required, the user has to search the entire cupboard. This is very time consuming and difficult.

For this purpose, the user creates separate folders and divides the files according to the years and further groups them according to the content.

- This is depicted in the following figure:



Packages 2-2

- Similarly, in Java, one can organize the files using packages.

A package is a namespace that groups related classes and interfaces and organizes them as a unit.

- For example, one can keep source files in one folder, images in another, and executables in yet another folder. Packages have the following features:

A package can have sub packages.

A package cannot have two members with the same name.

If a class or interface is bundled inside a package, it must be referenced using its fully qualified name, which is the name of the Java class including its package name.

If multiple classes and interfaces are defined within a package in a single Java source file, then only one of them can be public. Which means we need to add a new class.

Package names are written in lowercase.

Standard packages in the Java language begin with `java` or `javax`.

Advantages of Using Packages

One can easily determine that these classes are related.

One can know where to find the required type that can provide the required functions.

The names of classes of one package would not conflict with the class names in other packages as the package creates a new namespace.

For example, `myPackage1 . Sample` and `myPackage2 . Sample`.

One can allow classes within one package to have unrestricted access to one another while restricting access to classes outside the package.

Packages can also store hidden classes that can be used within the package, but are not visible or accessible outside the package.

Packages can also have classes with data members that are visible to other classes, but not accessible outside the package.

When a program from a package is called for the first time, the entire package gets loaded into the memory.

Due to this, subsequent calls to related subprograms of the same package do not require any further disk Input/Output (I/O).

Type of Packages 1-11

- The Java platform comes with a huge class library which is a set of packages.
- These classes can be used in applications by including the packages in a class.
- This library is known as the Application Programming Interface (API).
- Every Java application or applet has access to the core package in the API, the `java.lang` package.
- For example,



The String class stores the state and behavior related to character strings.



The File class allows the developer to create, delete, compare, inspect, or modify a file on the file system.



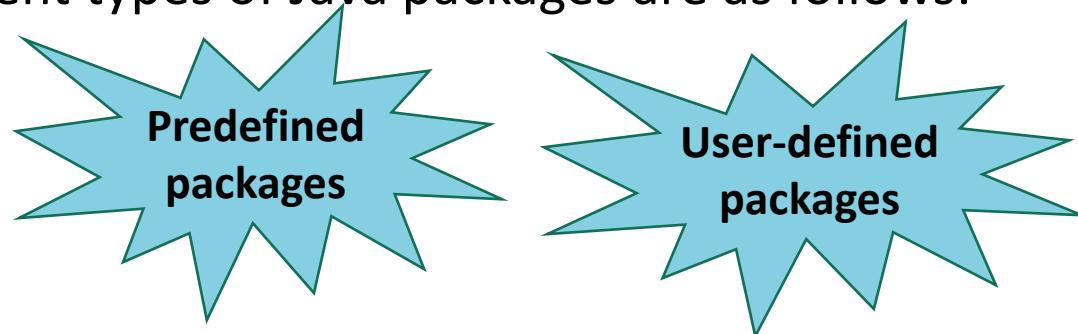
The Socket class allows the developer to create and use network sockets.



The various Graphical User Interface (GUI) control classes such as Button, Checkbox, and so on provide ready to use GUI controls.

Type of Packages 2-11

- The different types of Java packages are as follows:



- ◆ The predefined packages are part of the Java API. Predefined packages that are commonly used are as follows:



Type of Packages 3-11

- To create user-defined packages, perform the following steps:



- Select an appropriate name for the package by using the following naming conventions:
 - Package names are usually written in all lower case to avoid conflict with the names of classes or interfaces.
 - Companies usually attach their reversed Internet domain name as a prefix to their package names.
 - For example, **com.sample.mypkg** for a package named **mypkg** created by a programmer at sample.com.
 - Naming conflicts occurring in the projects of a single company are handled according to the naming conventions specific to that company.
 - This is done usually by including the region name or the project name after the company name.
 - For example, **com.sample.myregion.mypkg**.
 - Package names should not begin with java or javax as they are used for packages that are part of Java API.

Type of Packages 4-11

- In certain cases, the Internet domain name may not be a valid package name.
- For example, if the domain name contains special characters such as hyphen, if the package name consists of a reserved Java keyword such as `char`, or if the package name begins with a digit or some other character that is illegal to use as the beginning of a Java package name.
- In such a case, it is advisable to use an underscore as shown in the following table:

Domain Name	Suggested Package Name
sample-name.sample.org	org.sample.sample_name
sample.int	int_.sample
007name.sample.com	com.sample._007name

2

- Create a folder with the same name as the package.

3

- Place the source files in the folder created for the package.

Type of Packages 5-11

4

- Add the package statement as the first line in all the source files under that package as depicted in the following code snippet:

```
package mysession;
class StaticMembers{
    public static void main(String[] args)
    {}
}
```

5

- Note that there can only be one package statement in a source file.
- Save the source file **StaticMembers.java** in the package **mysession**.

6

- Compile the code as follows:

```
javac StaticMembers.java
```

OR

- Compile the code with -d option as follows:

```
javac -d . StaticMembers.java
```
- where, -d stands for directory and '.' stands for current directory.
- The command will create a sub-folder named **mysession** and store the compiled class file inside it.

Type of Packages 6-11

7

- From the parent folder of the source file, execute it using the fully qualified name as follows:

```
java mysession.StaticMembers
```

- Java allows the user to import the classes from predefined as well as user-defined packages using an import statement.
- However, the access specifiers associated with the class members will determine if the class members can be accessed by a class of another package.
- A member of a public class can be accessed outside the package by doing any of the following:



Referring to the member class by its fully qualified name, that is,
package-name.class-name.



Importing the package member, that is,
import package-name.class-name.



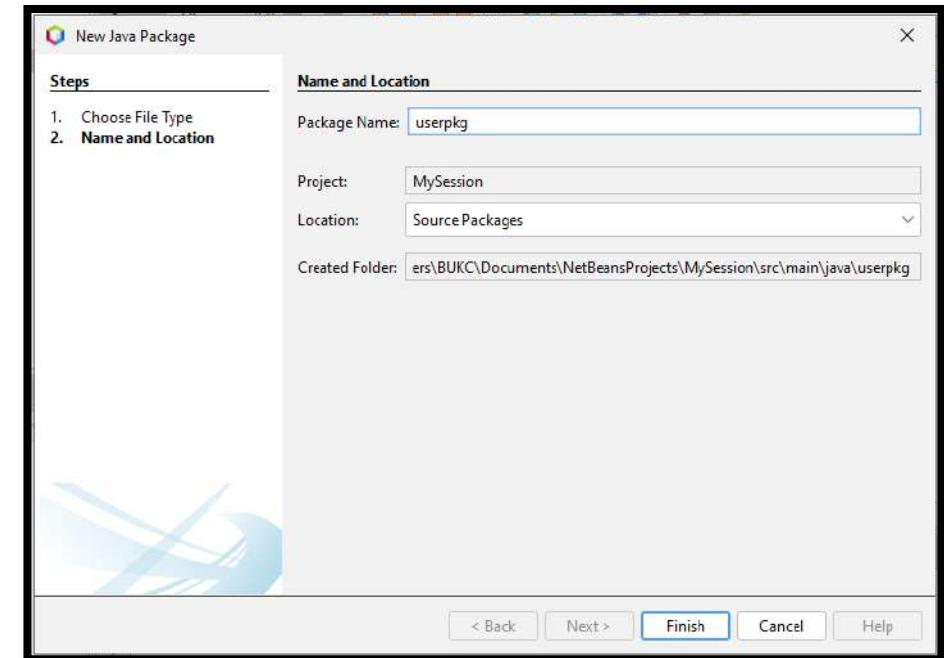
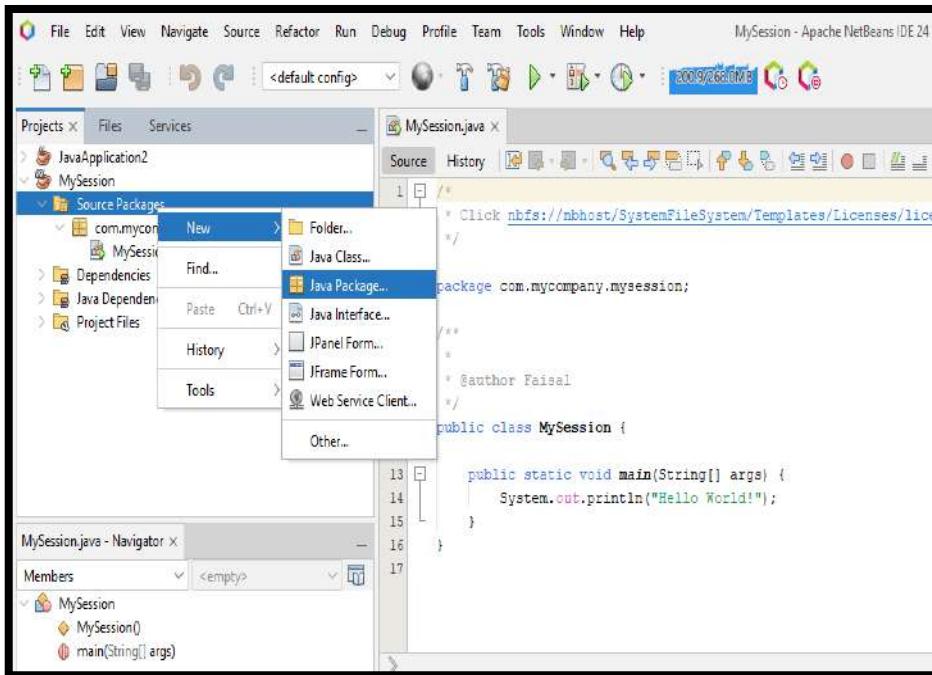
Importing the entire package, that is,
import package-name.*.

Type of Packages 7-11

- To create a new package using NetBeans IDE, perform the following steps:

1
2

- Open the project in which the package is to be created.
 - In this case **MySession** project has been chosen.
- Right-click **Source Packages** → **New** → **Java Package** to display the **New Java Package** dialog box.
 - For example, in the following figure, the project **mySession** is opened and the **Java Package** option is selected:



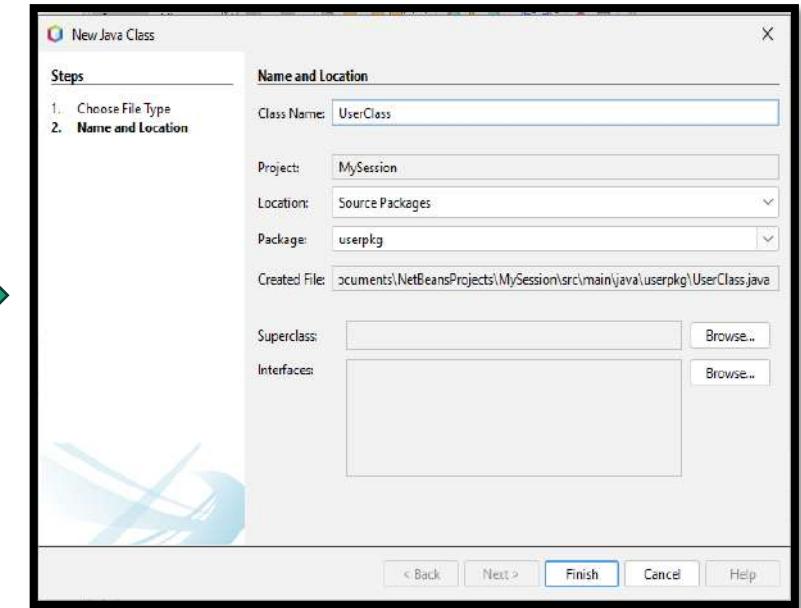
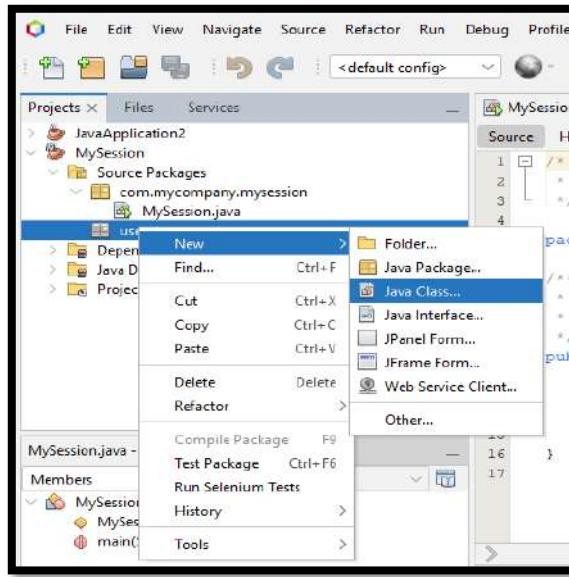
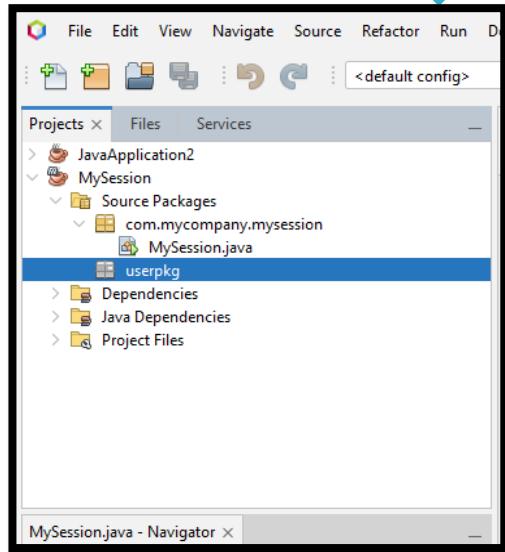
Type of Packages 8-11

3

- Type **userpkg** in the Package Name box of the New Java Package dialog box that is displayed.

4

- Click **Finish**. The **userpkg** package is created as shown in the following figure:



5

- Right-click **userpkg** and select **New → Java Class** to add a new class to the package.

6

- Type **UserClass** as the **Class Name** box of the **New Java Class** dialog box and click **Finish**.

Type of Packages 9-11

7

- Type the code in the class as depicted in the following code snippet:

```
package userpkg;  
// Import the predefined and user-defined packages  
import java.util.ArrayList;  
import mysession.StaticMembers;  
  
public class UserClass {  
  
    // Instantiate ArrayList class of java.util package  
    ArrayList<String> myCart = new ArrayList<>(); // line 1  
  
    /**  
     * Initializes an ArrayList  
     *  
     * @return void  
     */  
    public void createList() {  
  
        // Add values to the list  
        myCart.add("Doll");
```

Type of Packages 10-11

```
myCart.add("Bus");
myCart.add("Teddy");
// Print the list
System.out.println("Cart contents are:"+ myCart);
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {

    // Instantiate the UserClass class
    UserClass objUser = new UserClass();
    objUser.createList(); // Invoke the createList() method

    // Instantiate the StaticMembers class
    StaticMembers objStatic = new StaticMembers();
    objStatic.displayCount(); // Invoke the displayCount() method
}
}
```

Type of Packages 11-11

- The two import statements namely, `java.util.ArrayList` and `mysession.StaticMembers` are used to import the packages `java.util` and `mysession` into the `UserClass` class.
- Following figure shows the output of the program:

```
:Output - Session9 (run)
run:
Cart contents are:[Doll, Bus, Teddy]
I am static block
Static counter is:1
Instance counter is:1
```

- ◆ The output shows the execution of the static block of `StaticMembers` class also.
- ◆ This is because the static block is executed as soon as the class is launched.

Implementing Operator Overloading in C#

Session 6

Operator Overloading [Java]

- In Java, operator overloading is not supported, which means you cannot redefine the behavior of built-in operators (like +, -, *, etc.) for user-defined types (like classes).
- This is a design choice in Java to keep the language simple and to avoid ambiguity in operator behavior.
- You can't overload operators; you can achieve similar functionality through method overloading and well-defined methods.

Operator Overloading

- Overloading an operator means making it behave differently.

```
...  
int result = Int.Add(54, 200);  
  
int result2 = 54 + 200;  
...
```

- We use operators to make equations look simple and easy to understand.
- A list of operators that can be overloaded are as follows:

+	-		~	++	--
*	/	%	&		^
<<	>>	=	<, >	<=	>=

Operator overloading

- Consider the following class:

```
class Complex{  
    private double real, img;  
    public Complex Add(Complex c);  
    public Complex Subtract(Complex cm);  
    public Complex Multiply(Complex cs);  
    ...  
}
```

Operator overloading

- Function implementation:

```
Complex Add (Complex c1, complex c2)
{
    Complex t;
    t.real = real + c1.real;
    t.img  = img   + c1.img;
    return t;
}
```

Operator overloading

- The following statement:
 - **Complex c3 = c1.Add(c2) ;**
 - Adds the contents of **c2** to **c1** and assigns it to **c3**

Operator overloading

- To perform operations in a single mathematical statement
e.g:
 - `c1+c2+c3+c4`
- We have to explicitly write:
 - `c1.Add(c2.Add(c3.Add(c4)))`

Operator overloading

- Alternative way is:
 - `t1 = c3.Add(c4);`
 - `t2 = c2.Add(t1);`
 - `t3 = c1.Add(t2);`

Operator overloading

- If the mathematical expression is big:
 - Converting it to C# code will involve complicated mixture of function calls
 - Less readable
 - Chances of human mistakes are very high
 - Code produced is very hard to maintain

Operator overloading

- C# provides a very elegant solution:
“Operator overloading”
- C# allows you to overload common operators like +, - or * etc...
- Mathematical statements don't have to be explicitly converted into function calls

Operator overloading

- Assume that operator **+** has been overloaded
- Actual C# code becomes:
 - **c1+c2+c3+c4**
- The resultant code is very easy to read, write and maintain

Operator overloading

- C# automatically overloads operators for pre-defined types
- Example of predefined types:
 - **int**
 - **float**
 - **double**
 - **char**
 - **long**

Operator overloading

- Example:
 - `float x;`
 - `int y;`
 - `x = 102.02 + 0.09;`
 - `Y = 50 + 47;`

Operator overloading

- The compiler probably calls the correct overloaded low-level function for addition i.e:
 - **// for integer addition:**
 - **Add(int a, int b)**
- **// for float addition:**
- **Add(float a, float b)**

Operator Overloading

- operators are static methods whose return values represent the result of an operation and whose parameters are the operands.
- When you create an operator for a class you say you have “overloaded” that operator, much as you might overload any member method.

public static Fraction operator+(Fraction lhs, Fraction rhs)

Operator Overloading

```
using System;

public struct Time
{
    public Time(int hours, int minutes)
    {
        this.hours = hours;
        this.minutes = minutes;
    }

    int hours, minutes;
    public static Time operator + (Time first, Time second)
    {
        return new Time(first.hours + second.hours, first.minutes + second.minutes);
    }
    public static void Main()
    {
        Time start = new Time();
        Time duration = new Time();
        Time finish = new Time();

        start.hours = 12;
        start.minutes = 10;

        duration.hours = 1;
        duration.minutes = 30;

        finish = start + duration;

        Console.WriteLine("Finish time would be : {0} hours and {1} minutes.", finish.hours,
        finish.minutes);
    }
}
```

Finish time would be : 13 hours and 40 minutes.

ASSOCIATION

CLASS ASSOCIATION [INHERITANCE]

OBJECT ASSOCIATION [COMPOSITION]

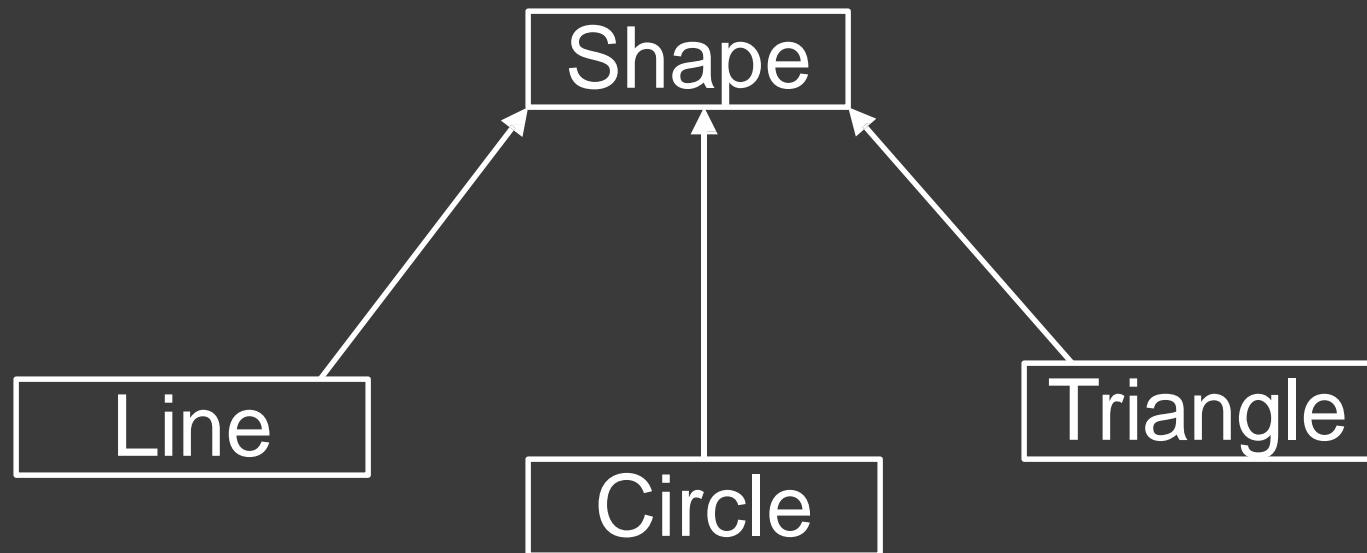
Inheritance

- A child inherits characteristics of its parents
- Besides inherited characteristics, a child may have its own unique characteristics

Inheritance in Classes

- If a class B inherits from class A then it contains all the characteristics (information structure and behaviour) of class A
- The parent class is called *base class* and the child class is called *derived class*
- Besides inherited characteristics, derived class may have its own unique characteristics

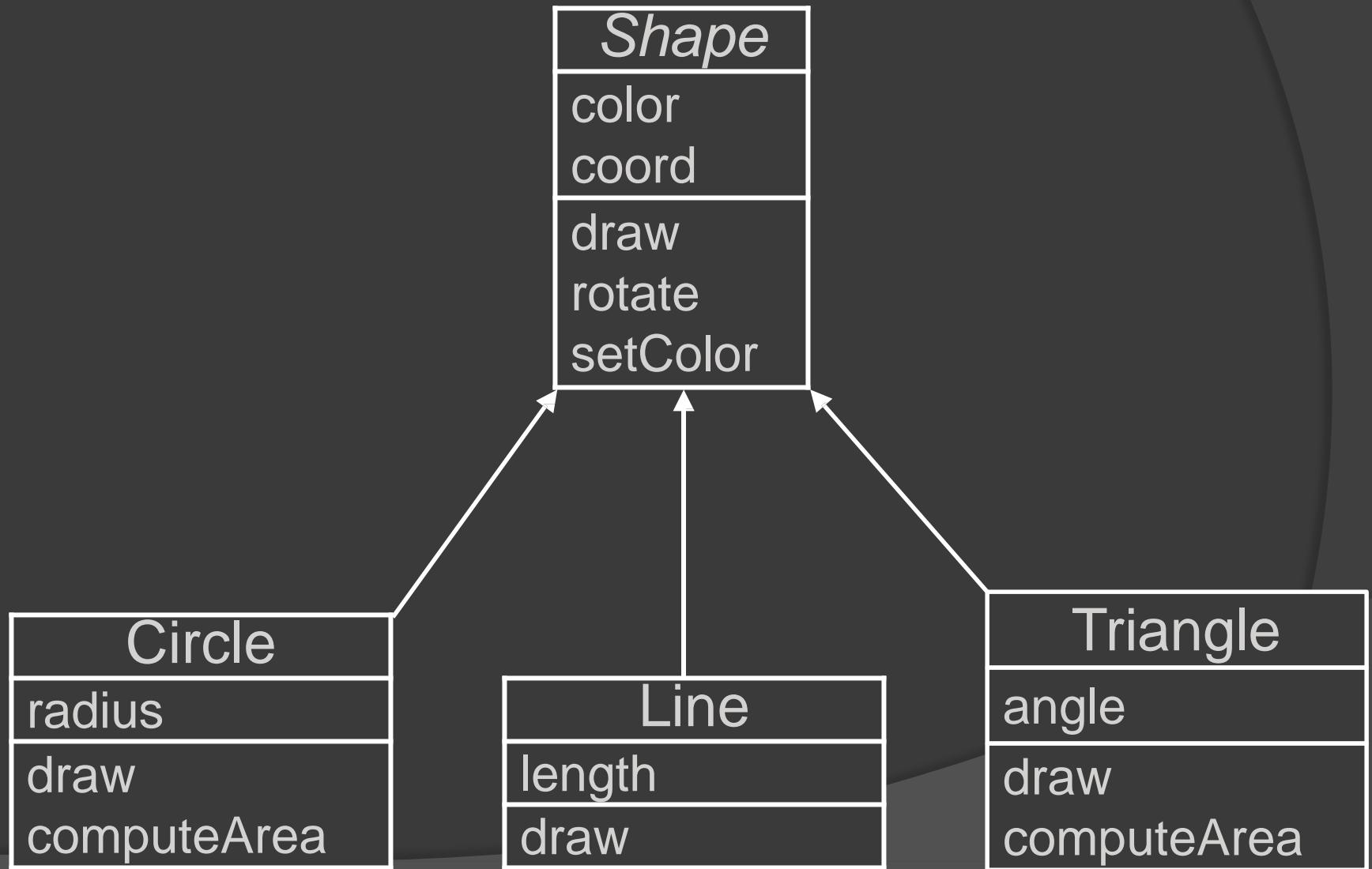
Example - Inheritance



Inheritance – “IS A” or “IS A KIND OF” Relationship

- ➊ Each derived class is a special kind of its base class

Example – “IS A” Relationship



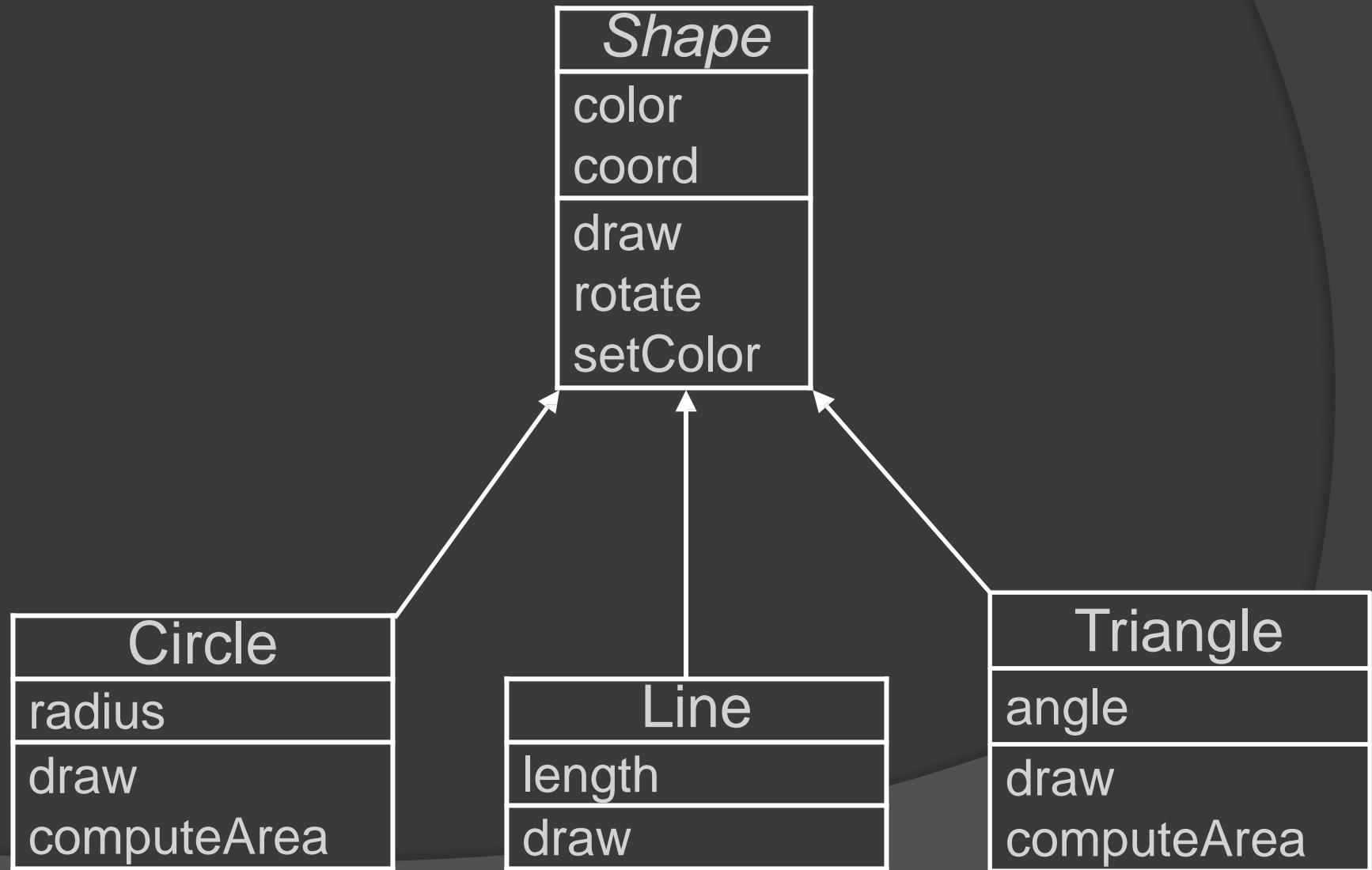
Inheritance – Advantages

- Reuse
- Less redundancy
- Increased maintainability

Reuse with Inheritance

- Main purpose of inheritance is reuse
- We can easily add new classes by inheriting from existing classes
 - Select an existing class closer to the desired functionality
 - Create a new class and inherit it from the selected class
 - Add to and/or modify the inherited functionality

Example Reuse



Recap-Inheritance

- Derived class inherits all the characteristics of the base class
- Besides inherited characteristics, derived class may have its own unique characteristics
- Major benefit of inheritance is reuse

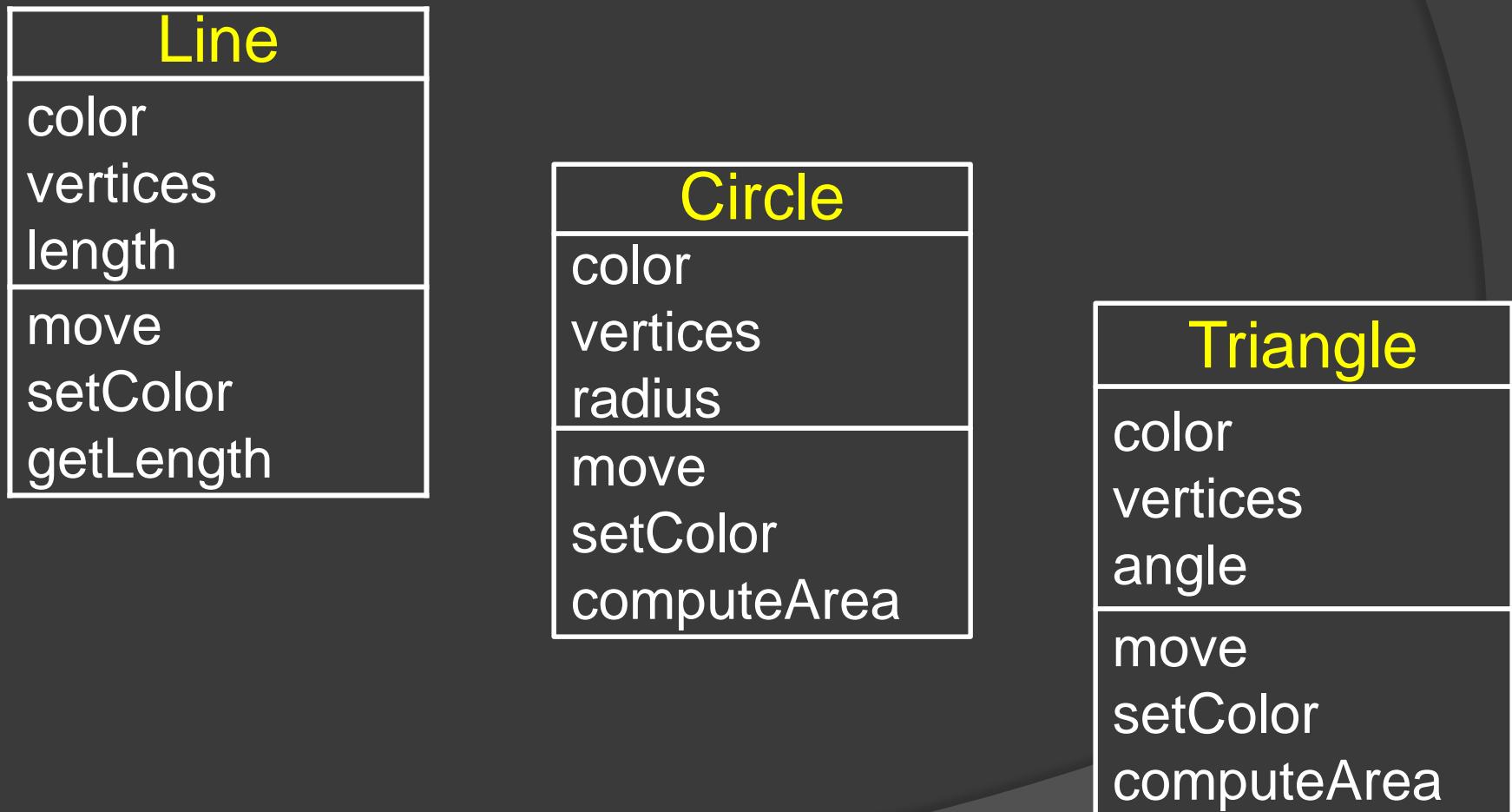
Concepts Related with Inheritance

- Generalization
- Subtyping (extension)
- Specialization (restriction)

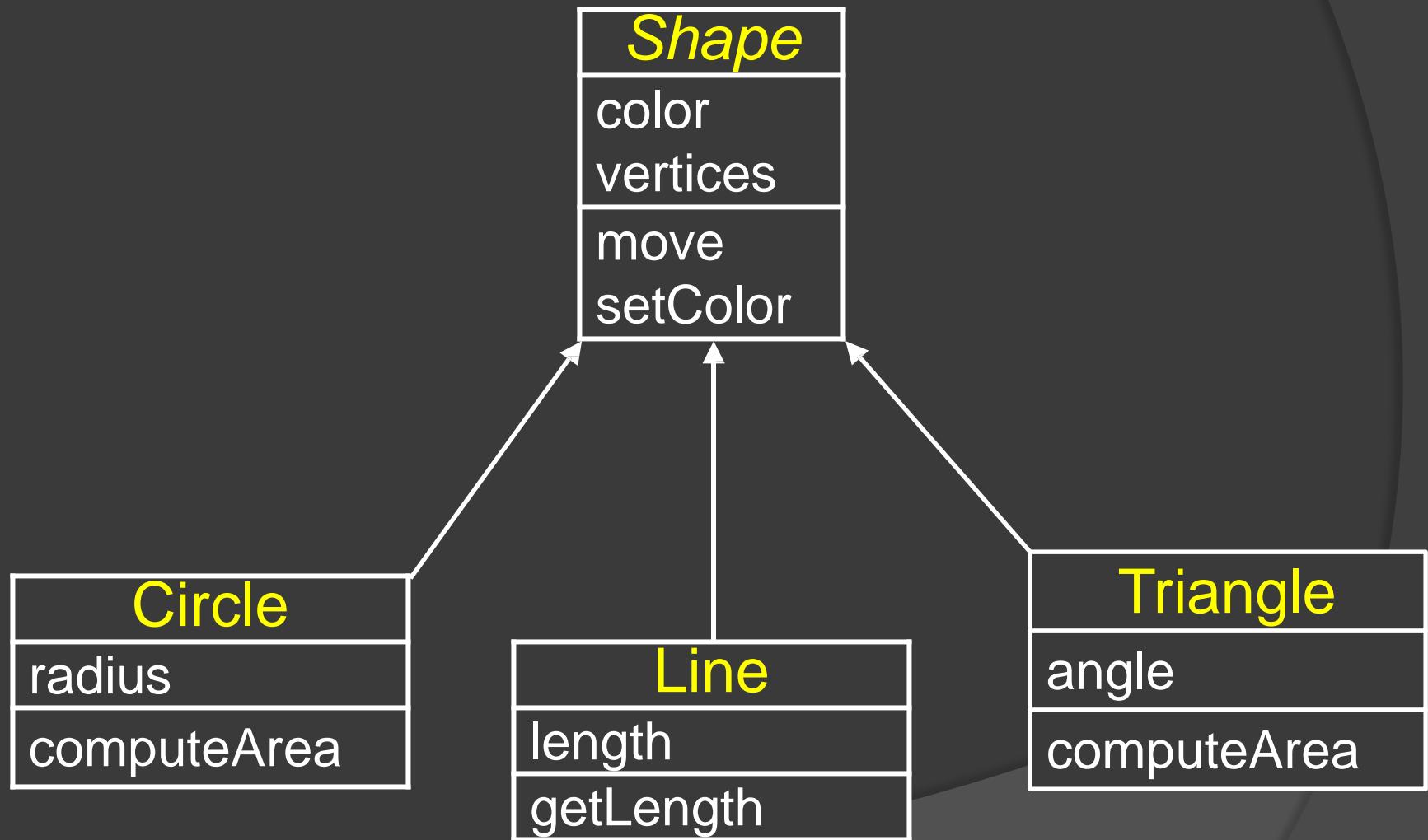
Generalization

- In OO models, some classes may have common characteristics
- We extract these features into a new class and inherit original classes from this new class
- This concept is known as Generalization

Example – Generalization



Example – Generalization



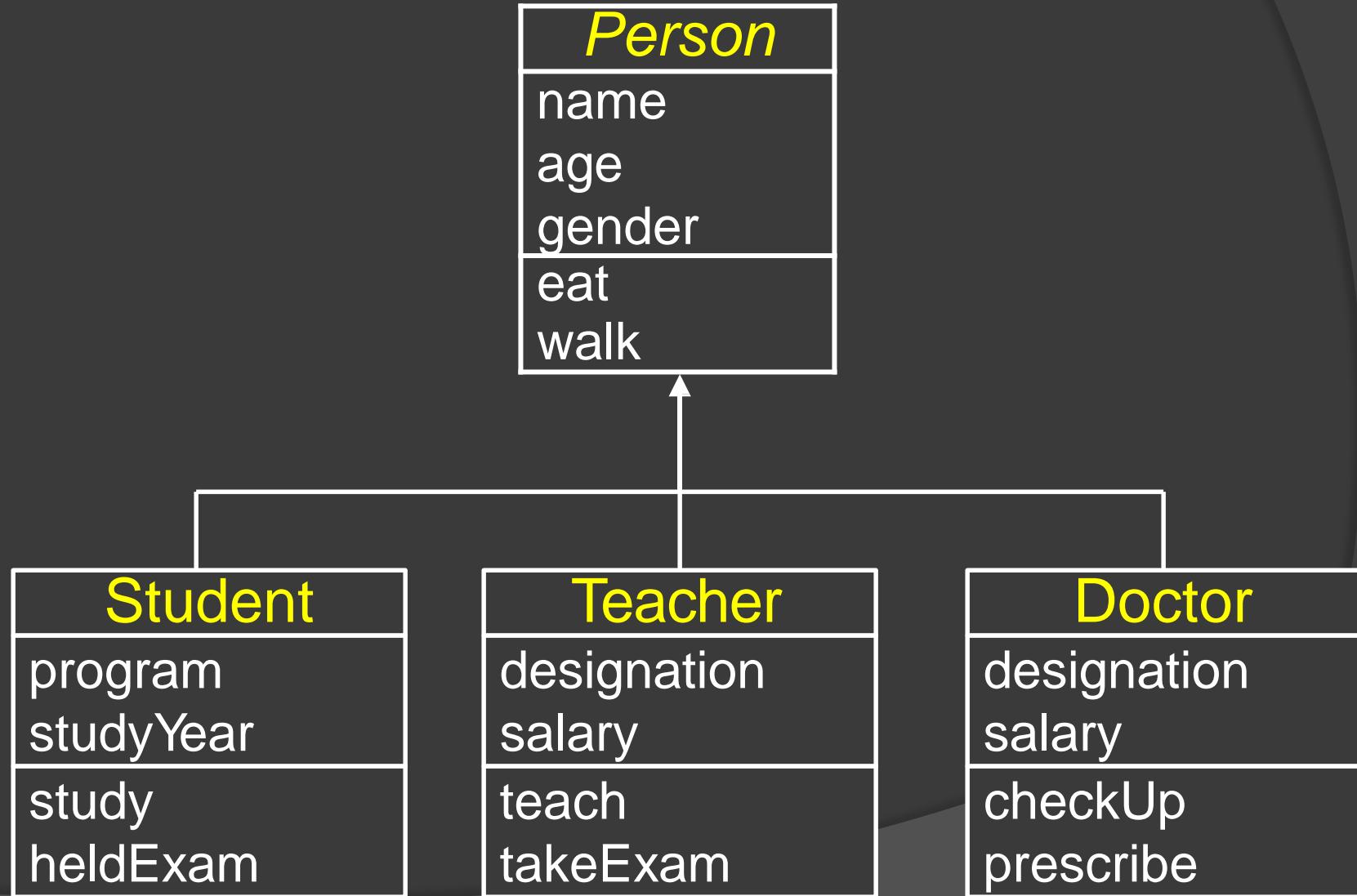
Example – Generalization

Student
name
age
gender
program
studyYear
study
heldExam
eat
walk

Teacher
name
age
gender
designation
salary
teach
takeExam
eat
walk

Doctor
name
age
gender
designation
salary
checkUp
prescribe
eat
walk

Example – Generalization



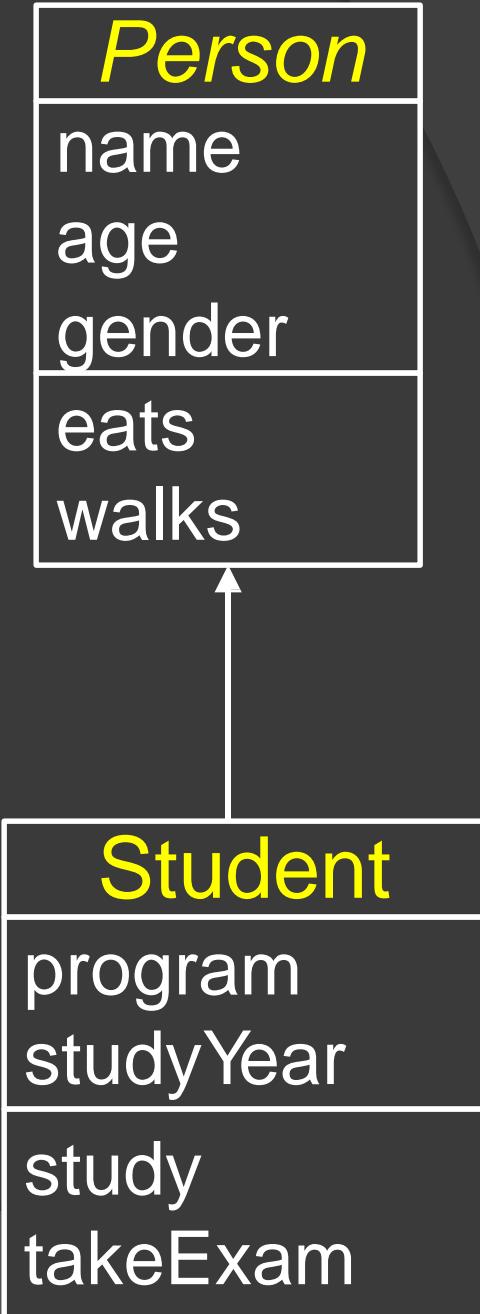
Sub-typing & Specialization

- We want to add a new class to an existing model
- Find an existing class that already implements some of the desired state and behaviour
- Inherit the new class from this class and add unique behaviour to the new class

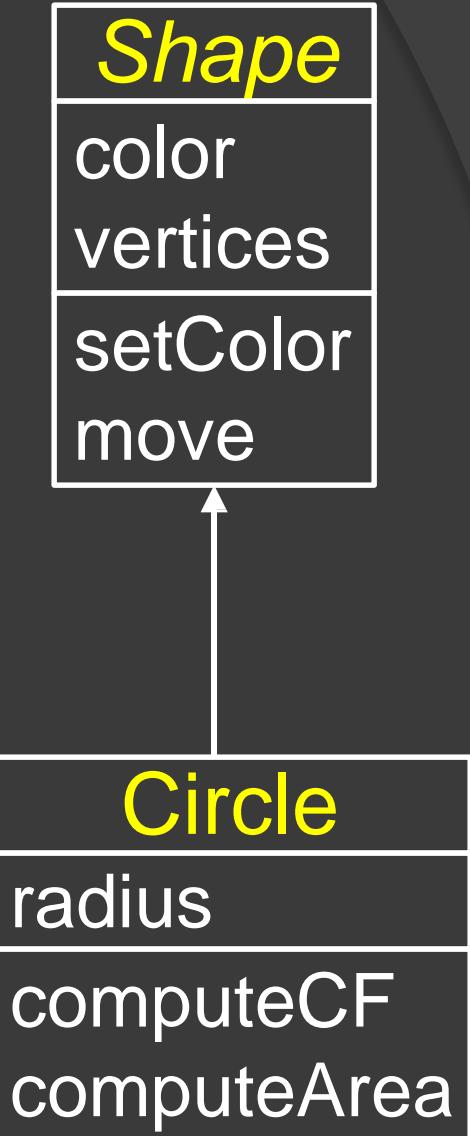
Sub-typing (Extension)

- Sub-typing means that derived class is behaviourally compatible with the base class
- Behaviourally compatible means that base class can be replaced by the derived class

Example – Sub-typing (Extension)



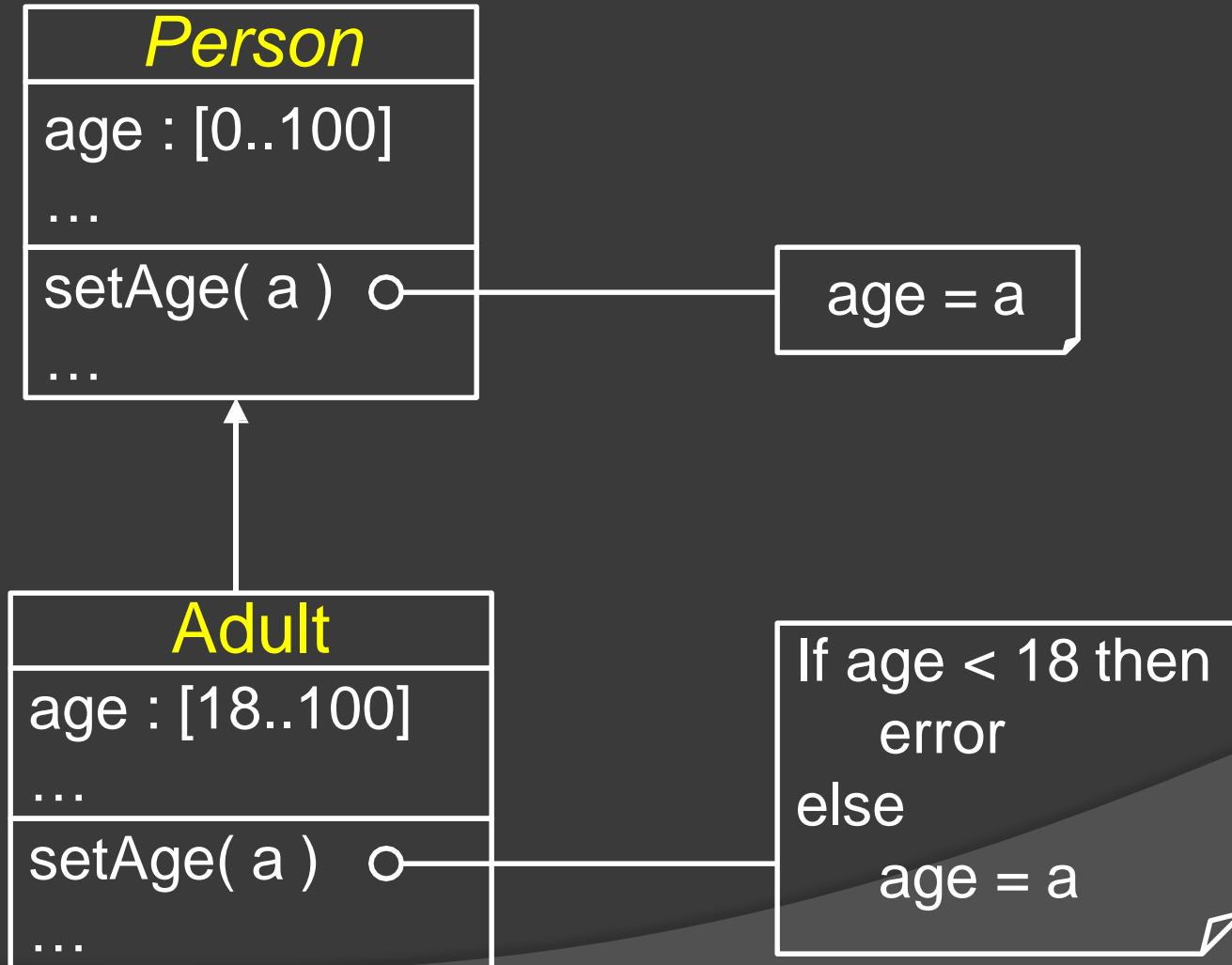
Example – Sub-typing (Extension)



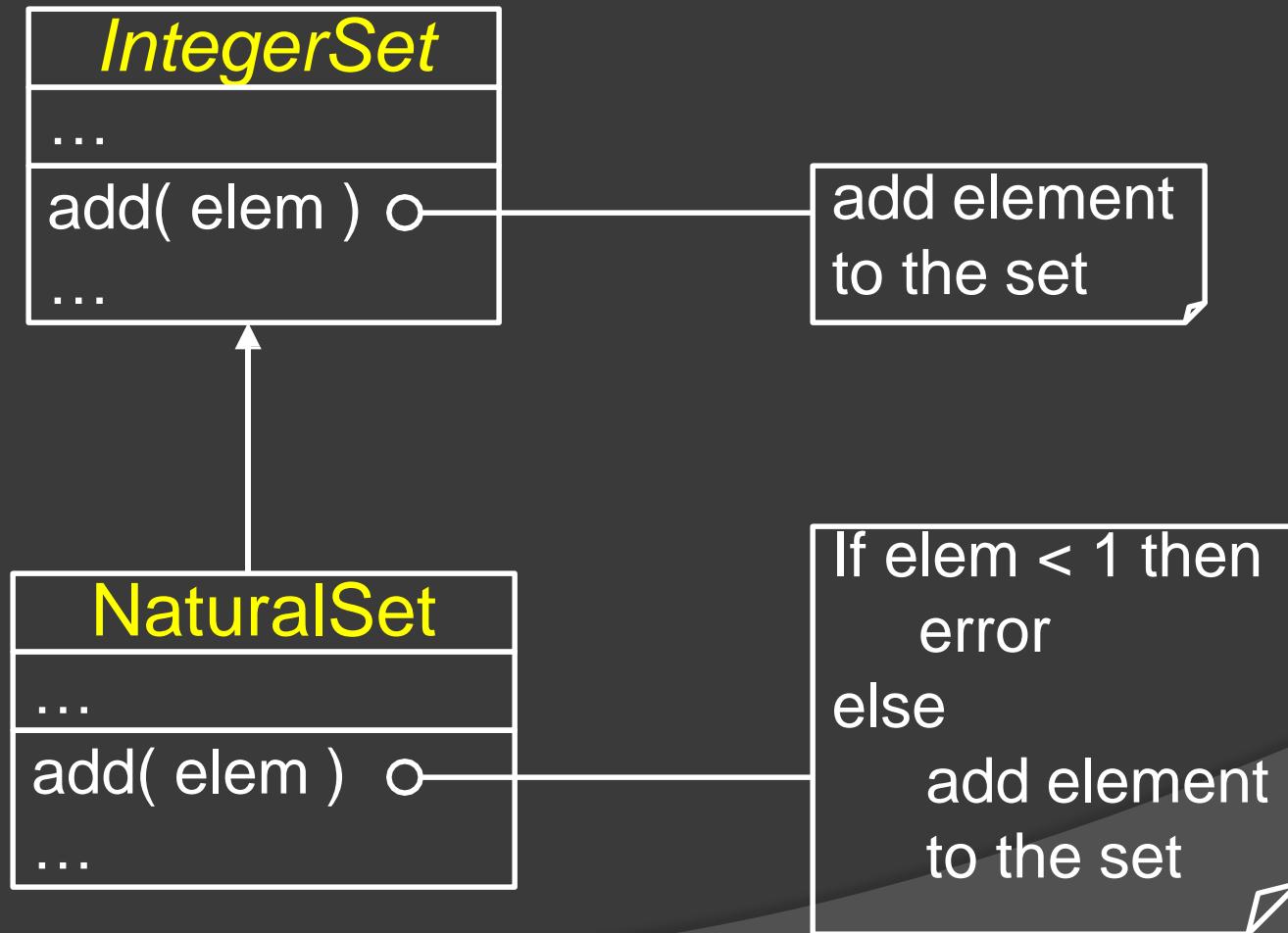
Specialization (Restriction)

- Specialization means that derived class is behaviourally incompatible with the base class
- Behaviourally incompatible means that base class can't always be replaced by the derived class

Example – Specialization (Restriction)



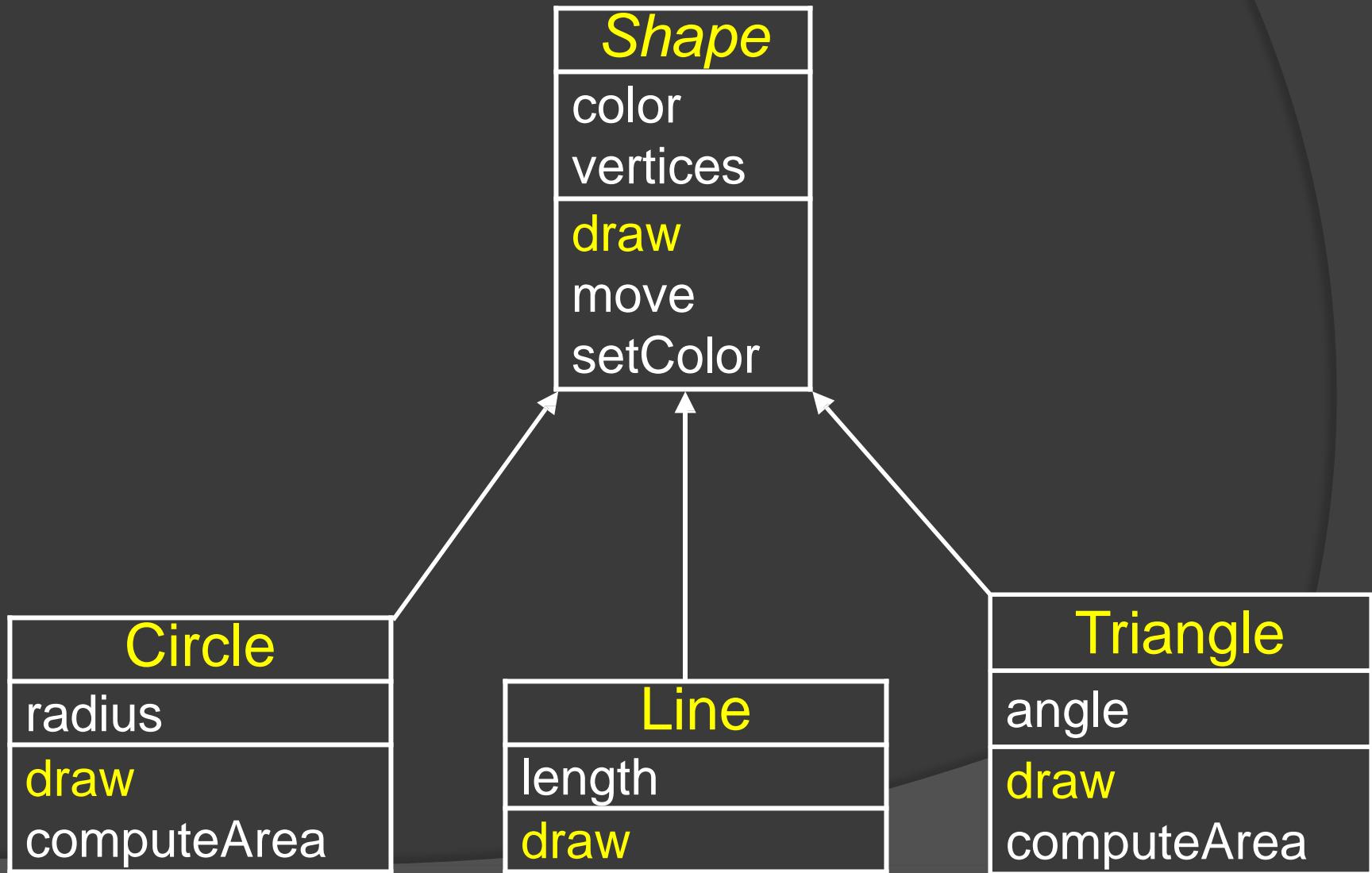
Example – Specialization (Restriction)



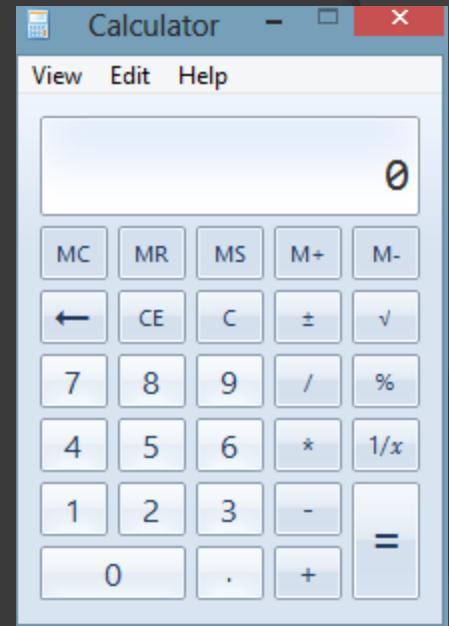
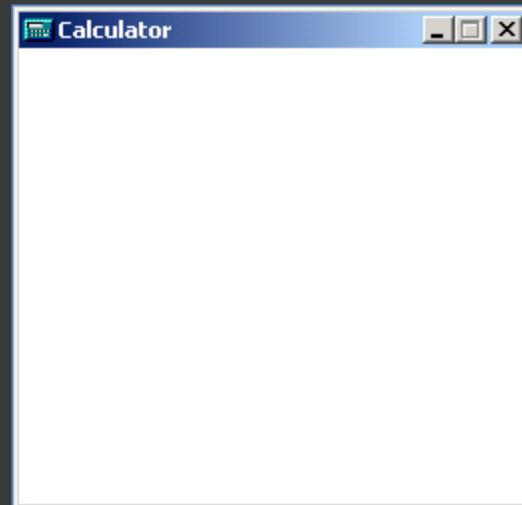
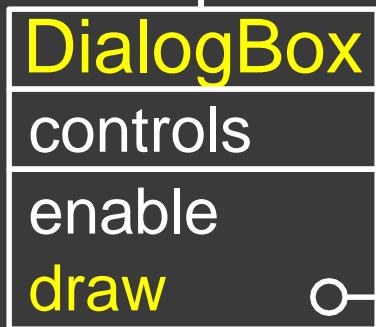
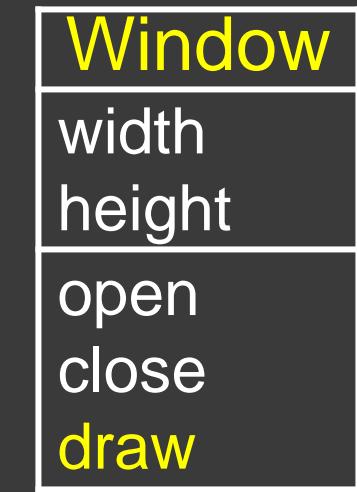
Overriding

- A class may need to override the default behavior provided by its base class
- Reasons for overriding
 - Provide behavior specific to a derived class
 - Extend the default behavior
 - Restrict the default behavior
 - Improve performance

Example – Specific Behaviour

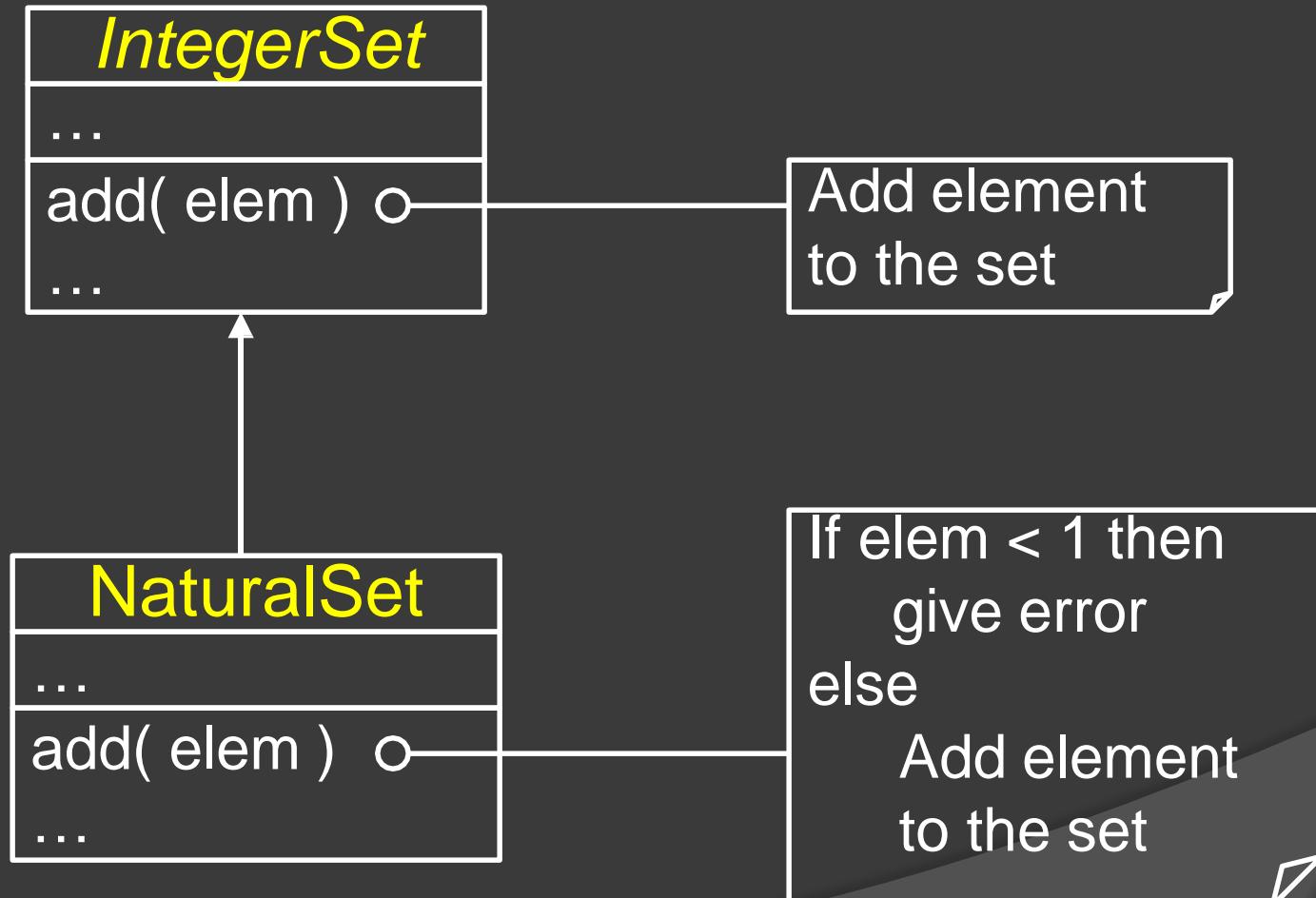


Example - Extension



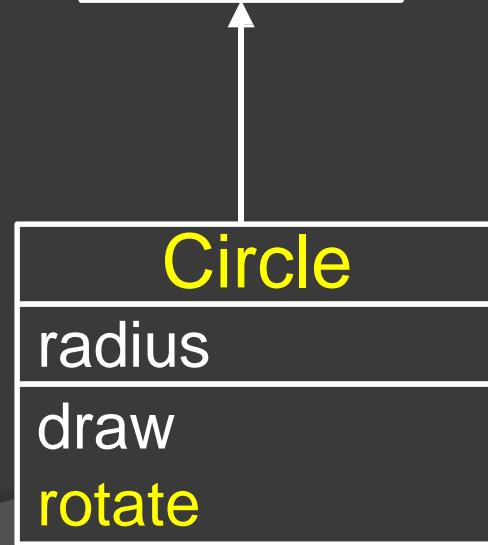
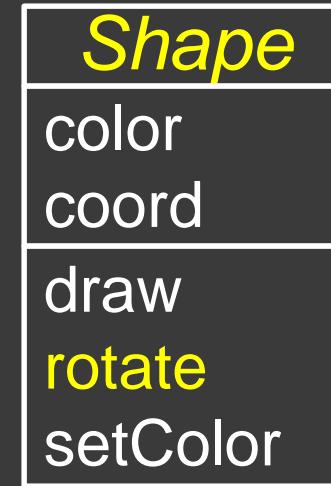
1 Invoke Window's draw
2 draw the dialog box

Example – Restriction



Example – Improve Performance

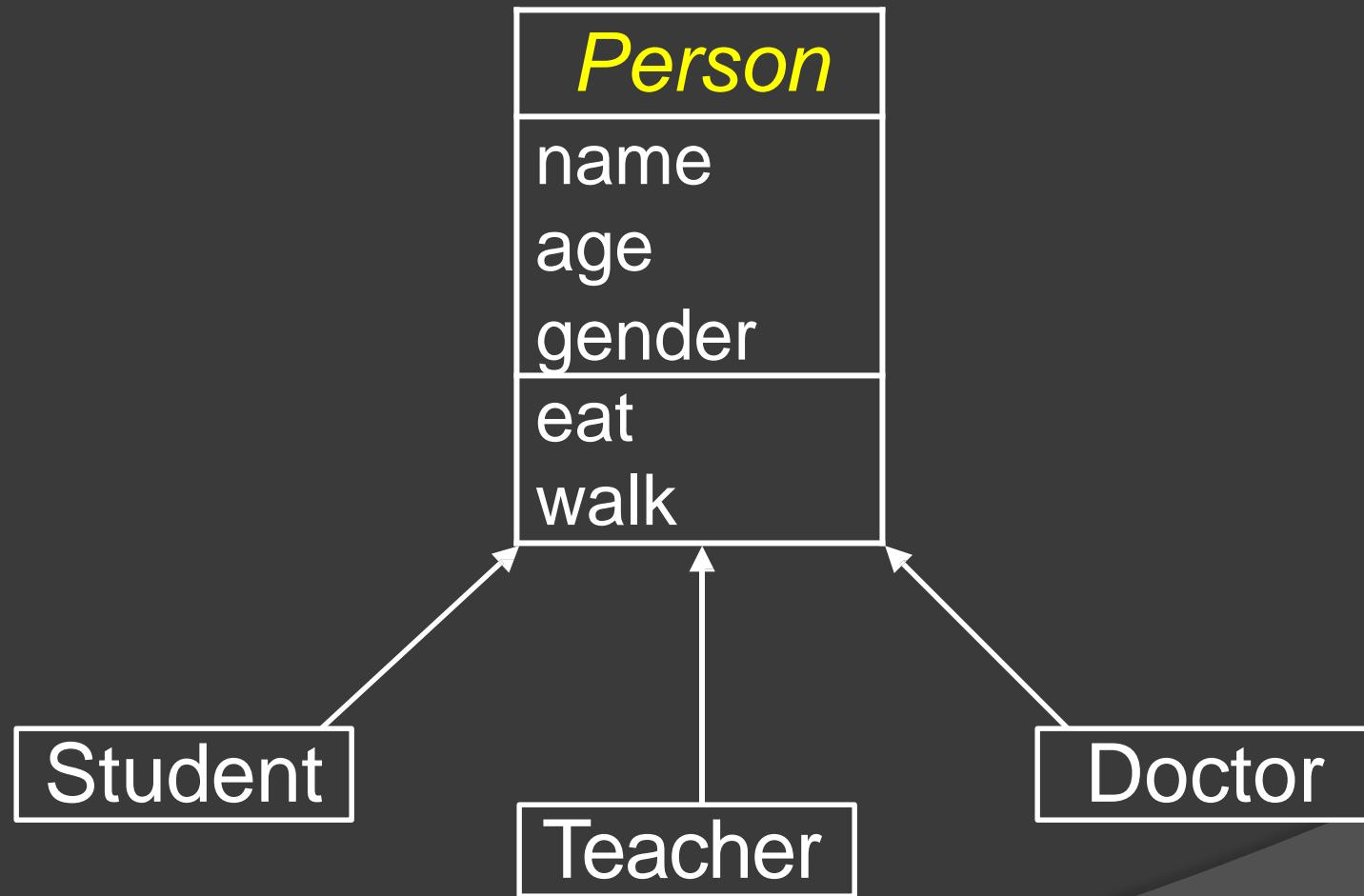
- Class Circle overrides *rotate* operation of class Shape with a Null operation.



Abstract Classes

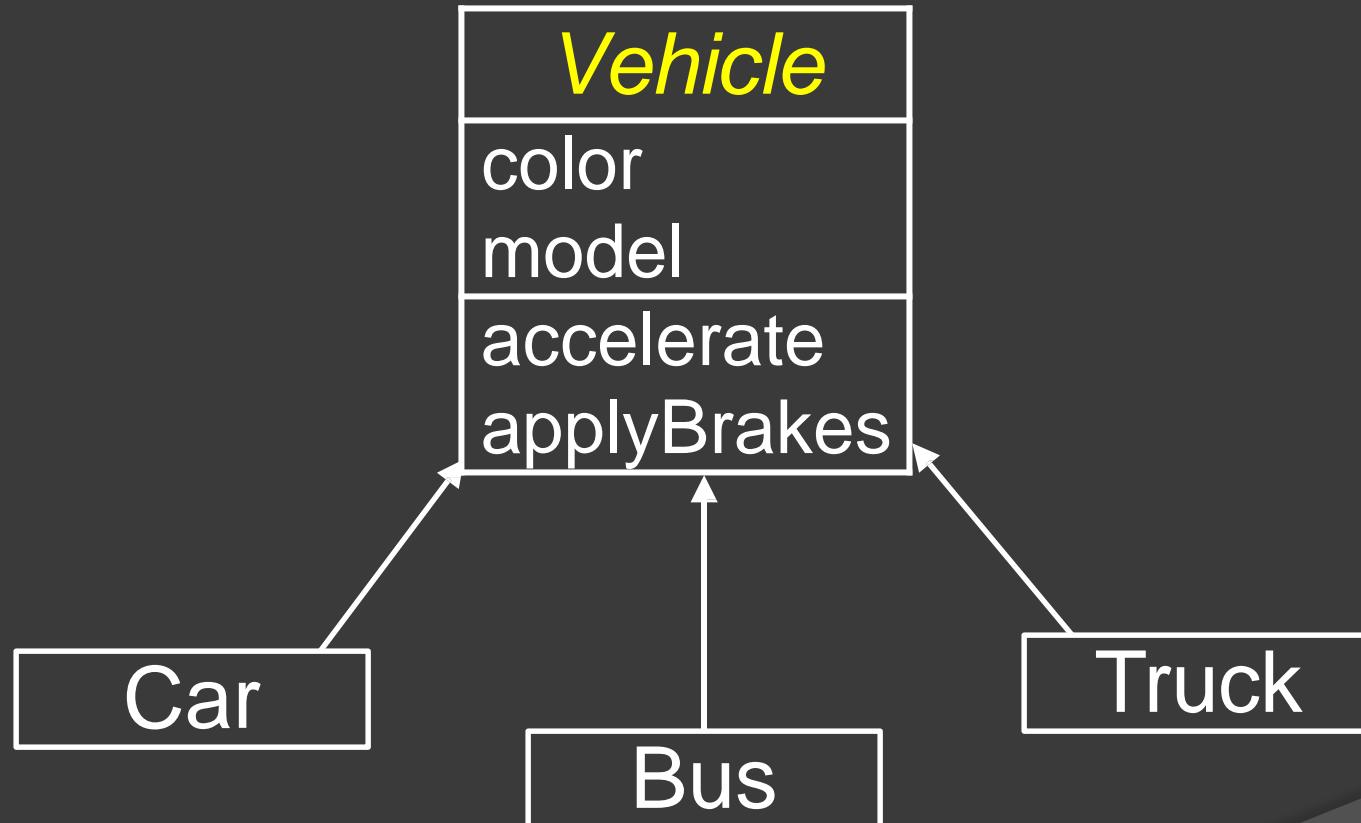
- An abstract class implements an abstract concept
- Main purpose is to be inherited by other classes
- Can't be instantiated
- Promotes reuse

Example – Abstract Classes



- Here, Person is an abstract class

Example – Abstract Classes

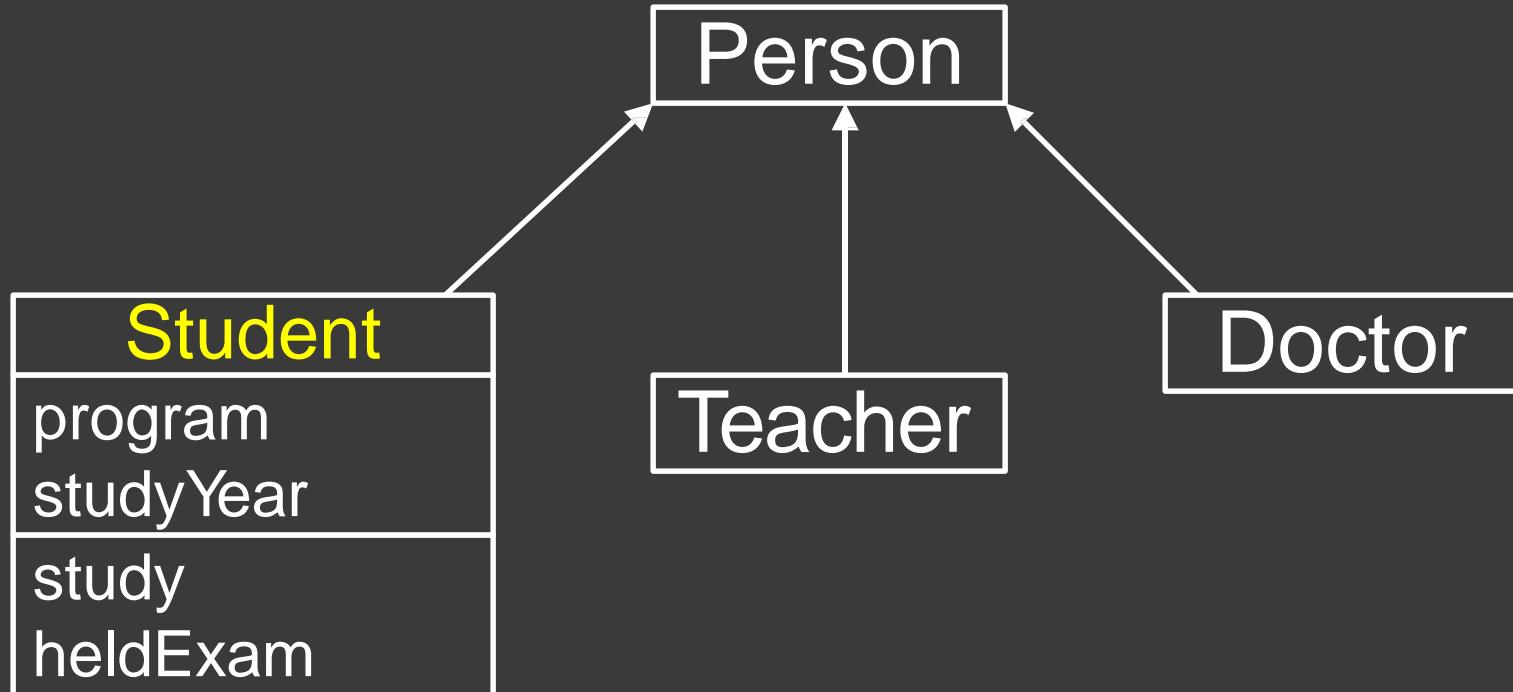


- Here, **Vehicle** is an abstract class

Concrete Classes

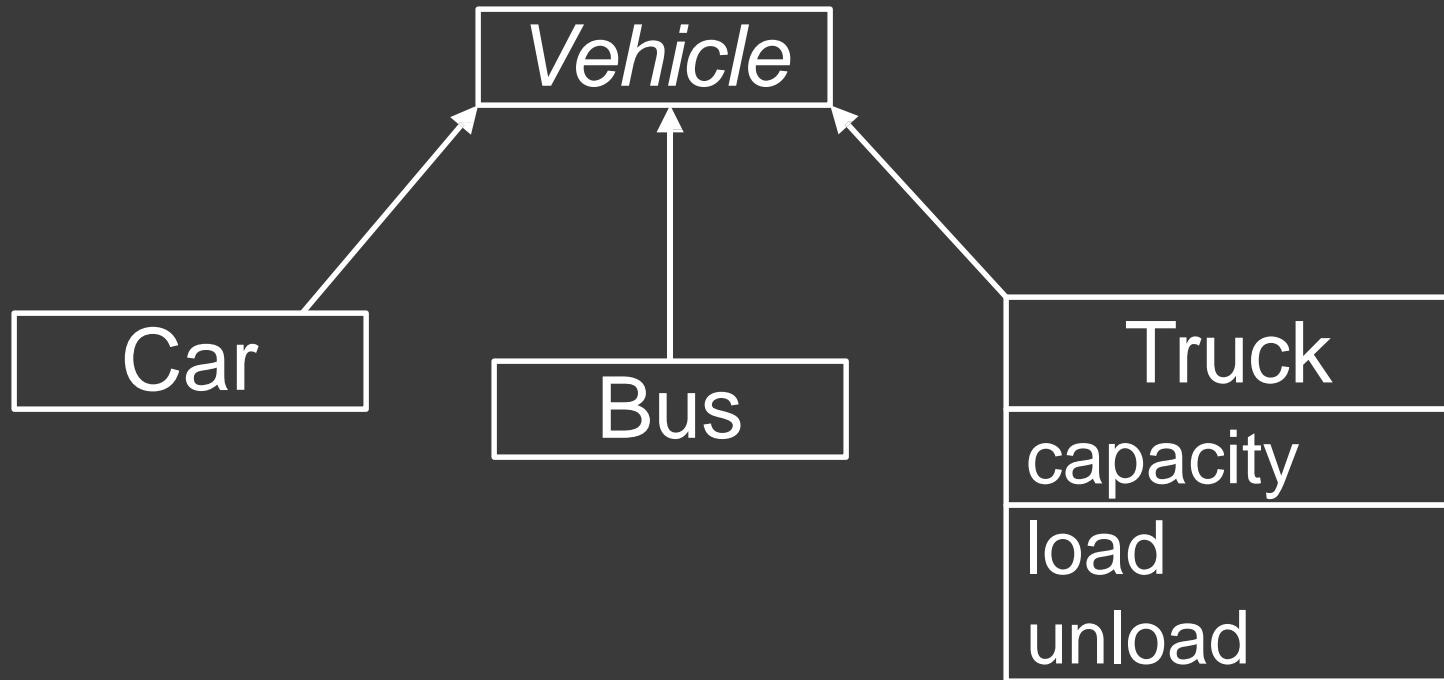
- A concrete class implements a concrete concept
- Main purpose is to be instantiated
- Provides implementation details specific to the domain context

Example – Concrete Classes



- Here, Student, Teacher and Doctor are concrete classes

Example – Concrete Classes



- Here, Car, Bus and Truck are concrete classes

Multiple Inheritance

[Not Supported by Java or C#]

- We may want to reuse characteristics of more than one parent class

Example – Multiple Inheritance



Mermaid

Example – Multiple Inheritance

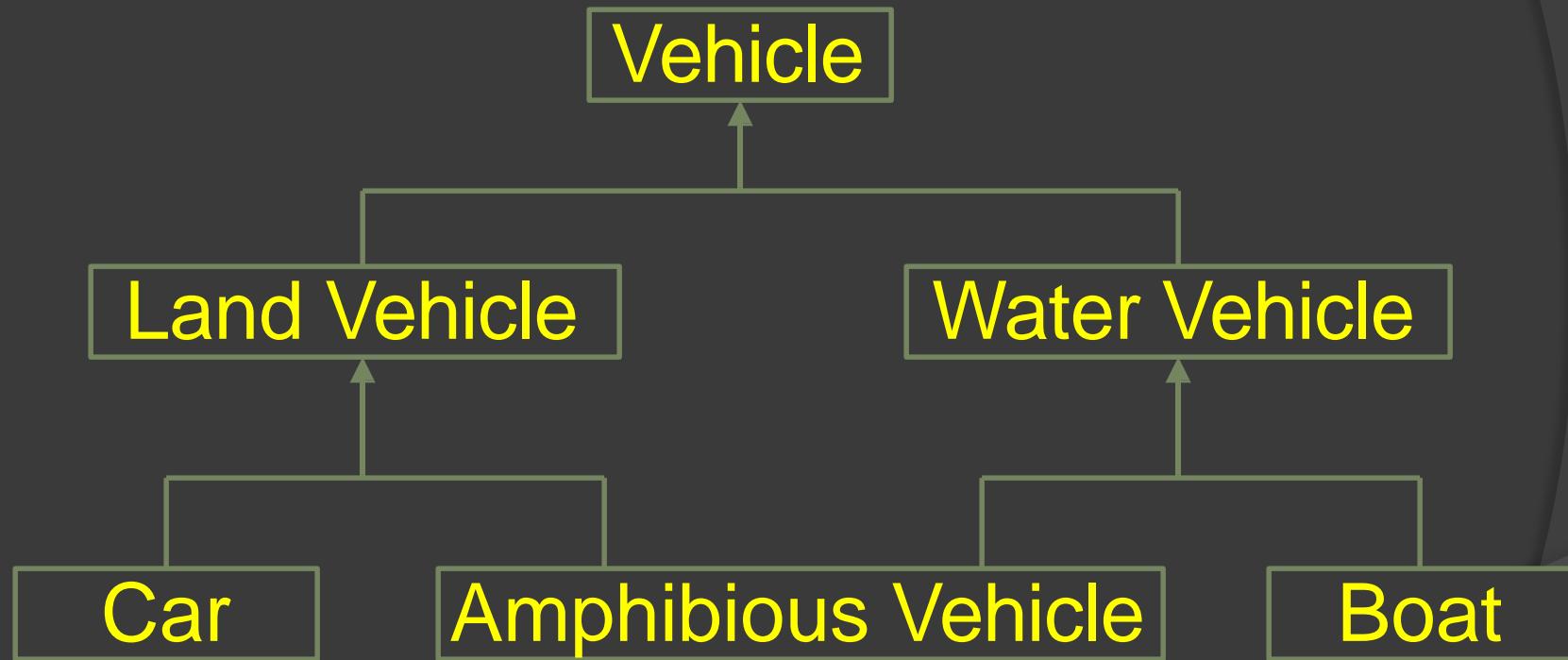


Example – Multiple Inheritance



Amphibious Vehicle

Example – Multiple Inheritance



Problems with Multiple Inheritance

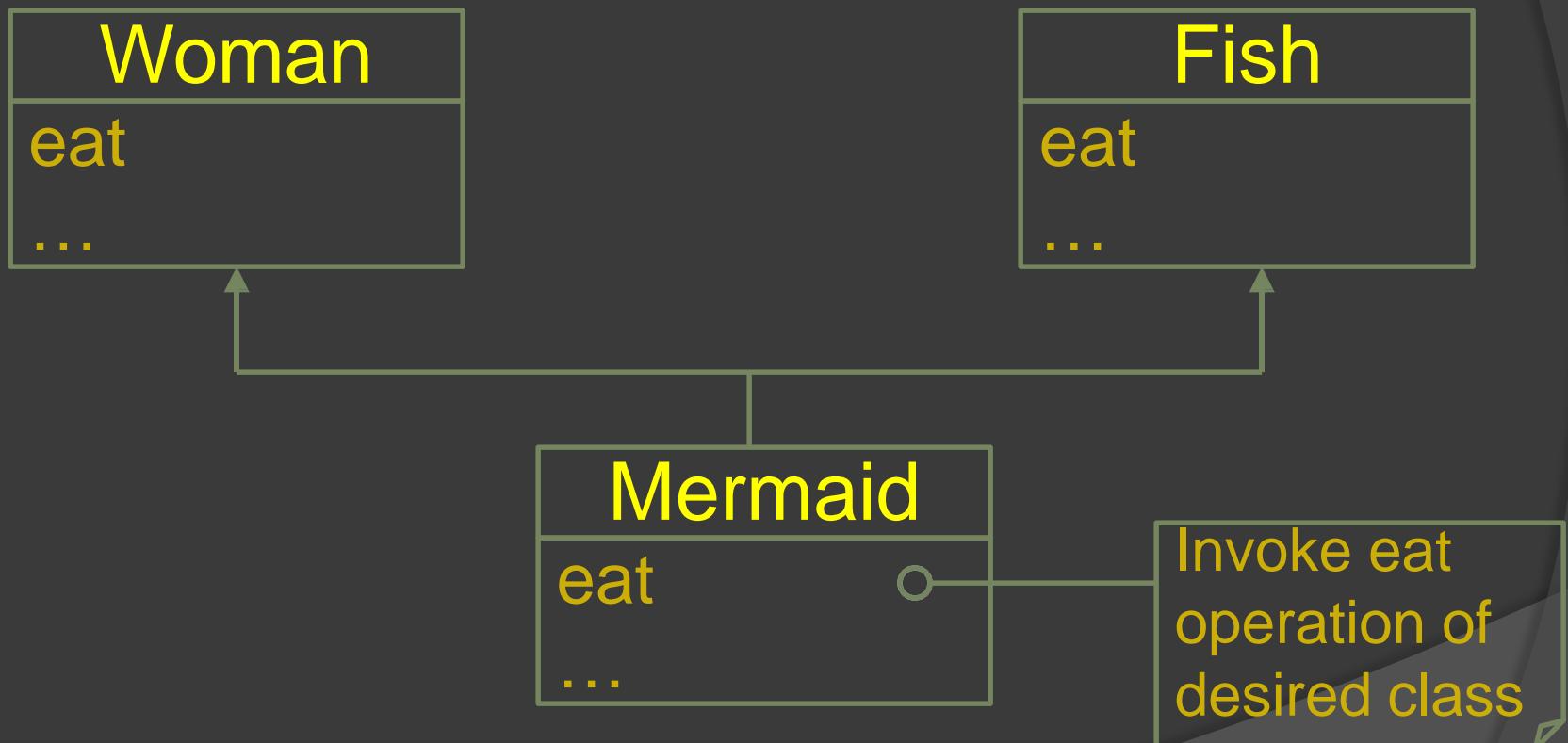
- Increased complexity
- Reduced understanding
- Duplicate features

Problem – Duplicate Features

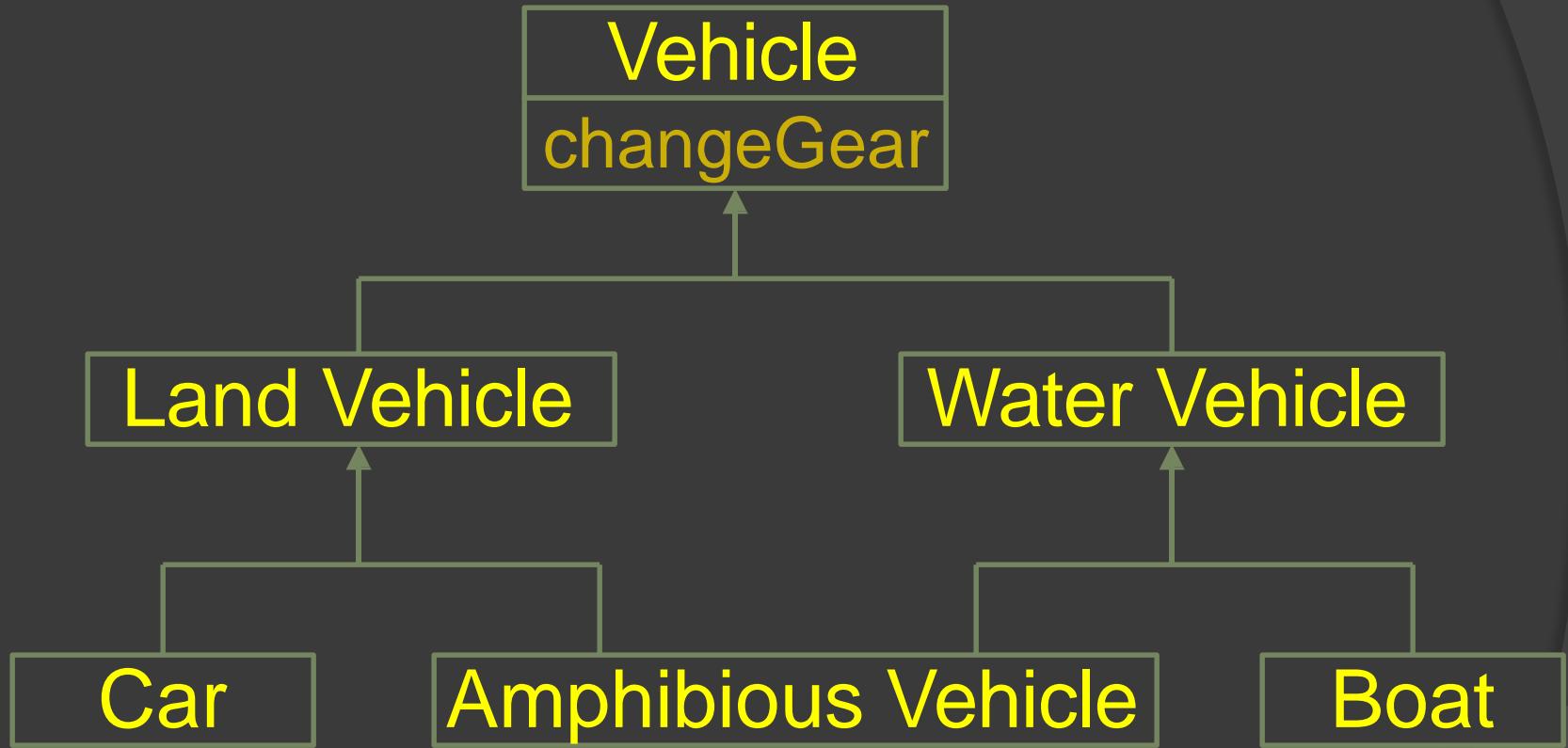


- Which *eat* operation *Mermaid* inherits?

Solution – Override the Common Feature



Problem – Duplicate Features (Diamond Problem)



- Which *changeGear* operation *Amphibious Vehicle* inherits?

Solution to Diamond Problem

- Some languages disallow diamond hierarchy
- Others provide mechanism to ignore characteristics from one side. In C++ we use virtual keyword during inheritance.
- Virtual specify that the base class should only be instantiated once, even if it's inherited multiple times through different paths.

```
#include <iostream>

class Base {
public:
    void display() {
        std::cout << "Base Class" << std::endl;
    }
};

class Derived1 : virtual public Base { // Virtual inheritance
};

class Derived2 : virtual public Base { // Virtual inheritance
};

class FinalDerived : public Derived1, public Derived2 {
};

int main() {
    FinalDerived obj;
    obj.display(); // Calls Base::display() without ambiguity
    return 0;
}
```

Association

- Objects in an object model interact with each other
- Usually, an object provides services to several other objects
- An object keeps associations with other objects to delegate tasks

Kinds of Association

- Class Association
 - Inheritance
- Object Association
 - Simple Association
 - Composition
 - Aggregation

Simple Association

- Is the weakest link between objects
- Is a reference by which one object can interact with some other object
- Is simply called as “association”

Kinds of Simple Association

- w.r.t navigation
 - One-way Association
 - Two-way Association

- w.r.t number of objects
 - Binary Association
 - Ternary Association
 - N-ary Association

One-way Association

- We can navigate along a single direction only
- Denoted by an arrow towards the server object

Example – Association



- Ali lives in a House

Example – Association



- Ali drives his Car

Two-way Association

- We can navigate in both directions
- Denoted by a line between the associated objects

Example – Two-way Association



- Employee works for company
- Company employs employees

Example – Two-way Association



- Yasir is a friend of Ali
- Ali is a friend of Yasir

Binary Association

- Associates objects of exactly two classes
- Denoted by a line, or an arrow between the associated objects

Example – Binary Association



- Association “works-for” associates objects of exactly two classes

Example – Binary Association

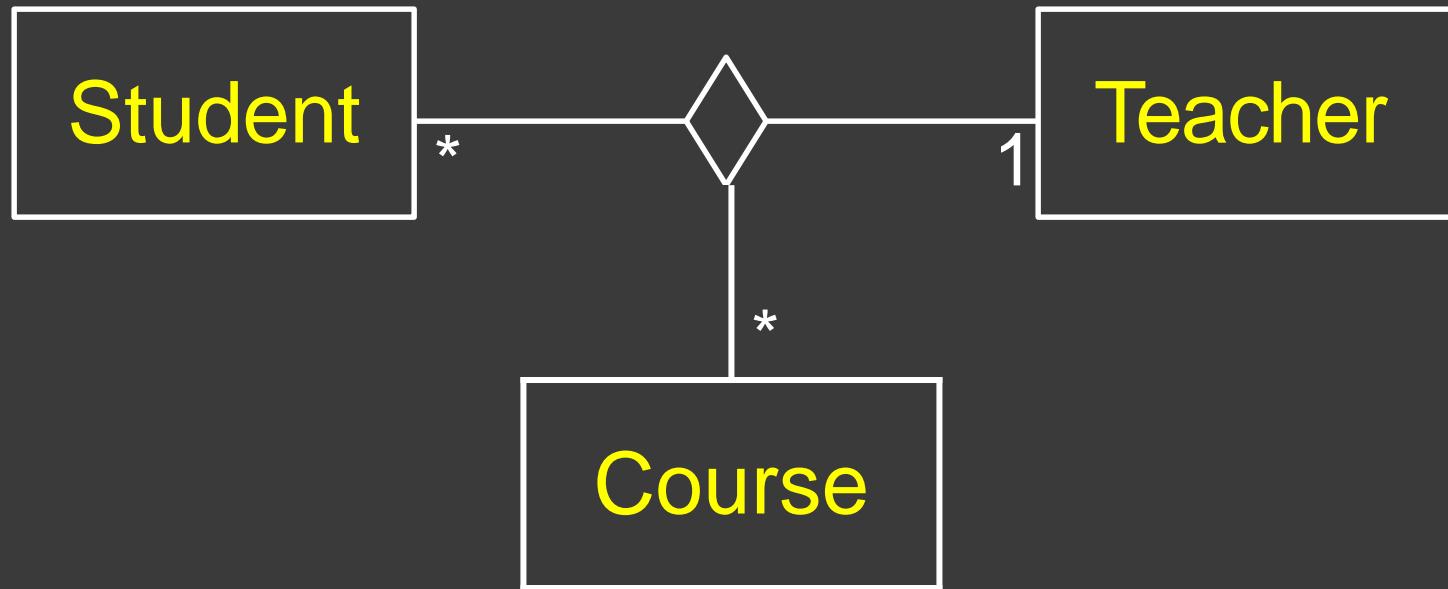


- Association “drives” associates objects of exactly two classes

Ternary Association

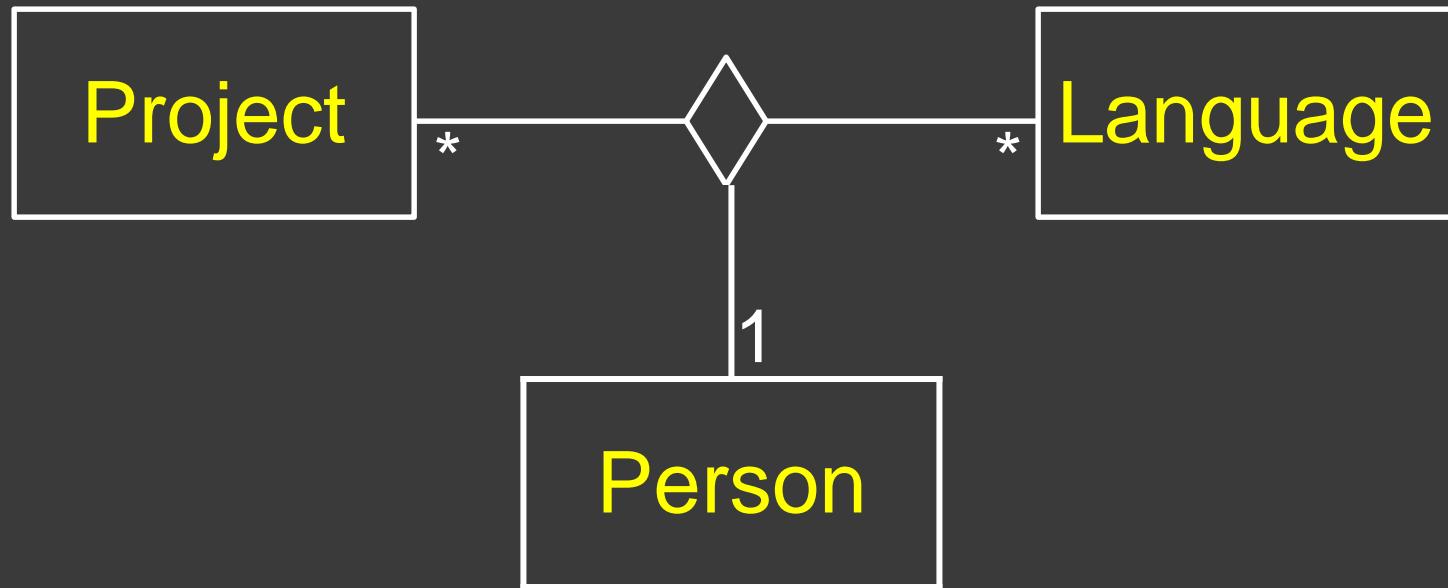
- Associates objects of exactly three classes
- Denoted by a diamond with lines connected to associated objects

Example – Ternary Association



- Objects of exactly three classes are associated

Example – Ternary Association



- Objects of exactly three classes are associated

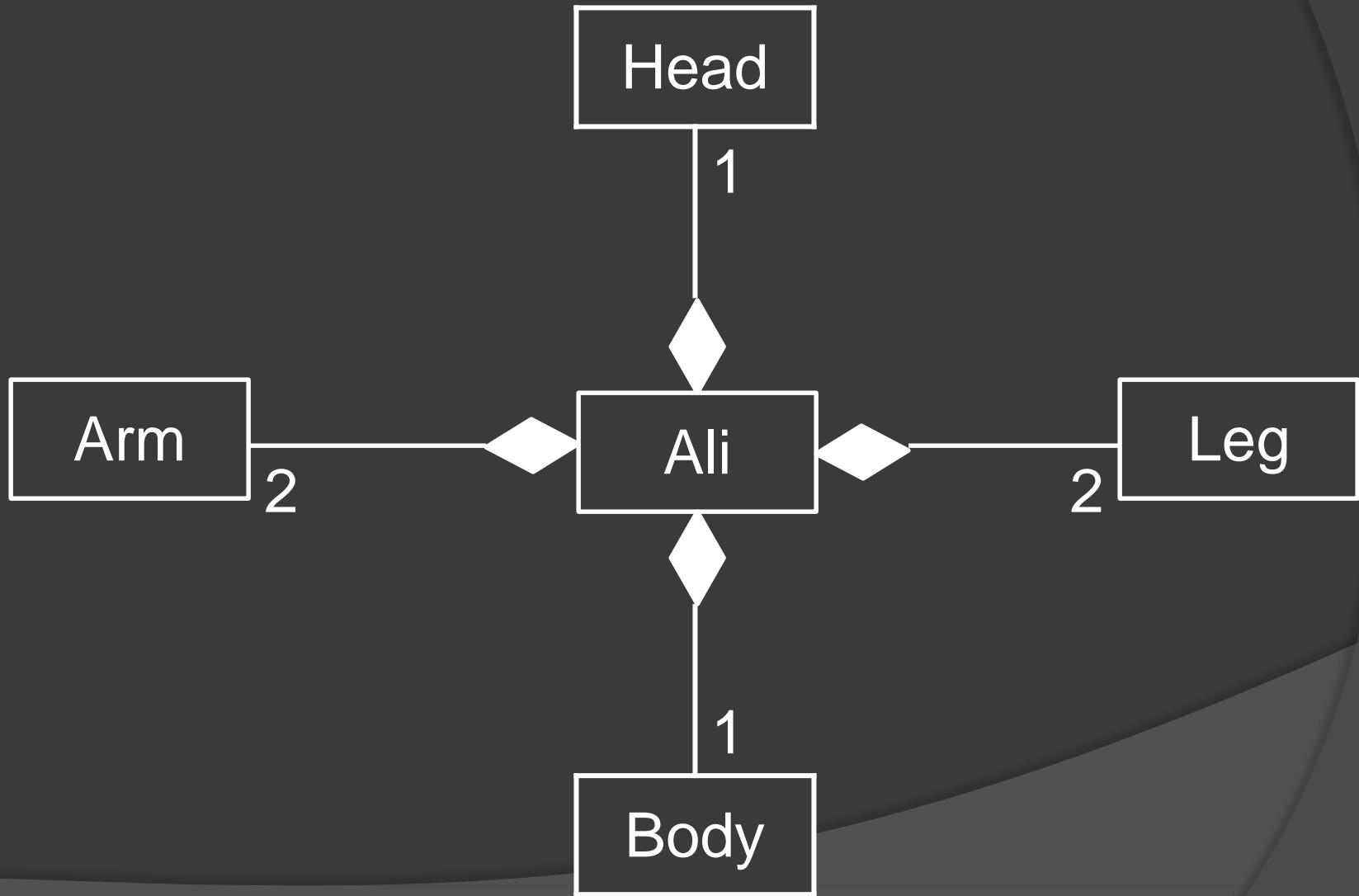
N-ary Association

- An association between 3 or more classes
- Practical examples are very rare

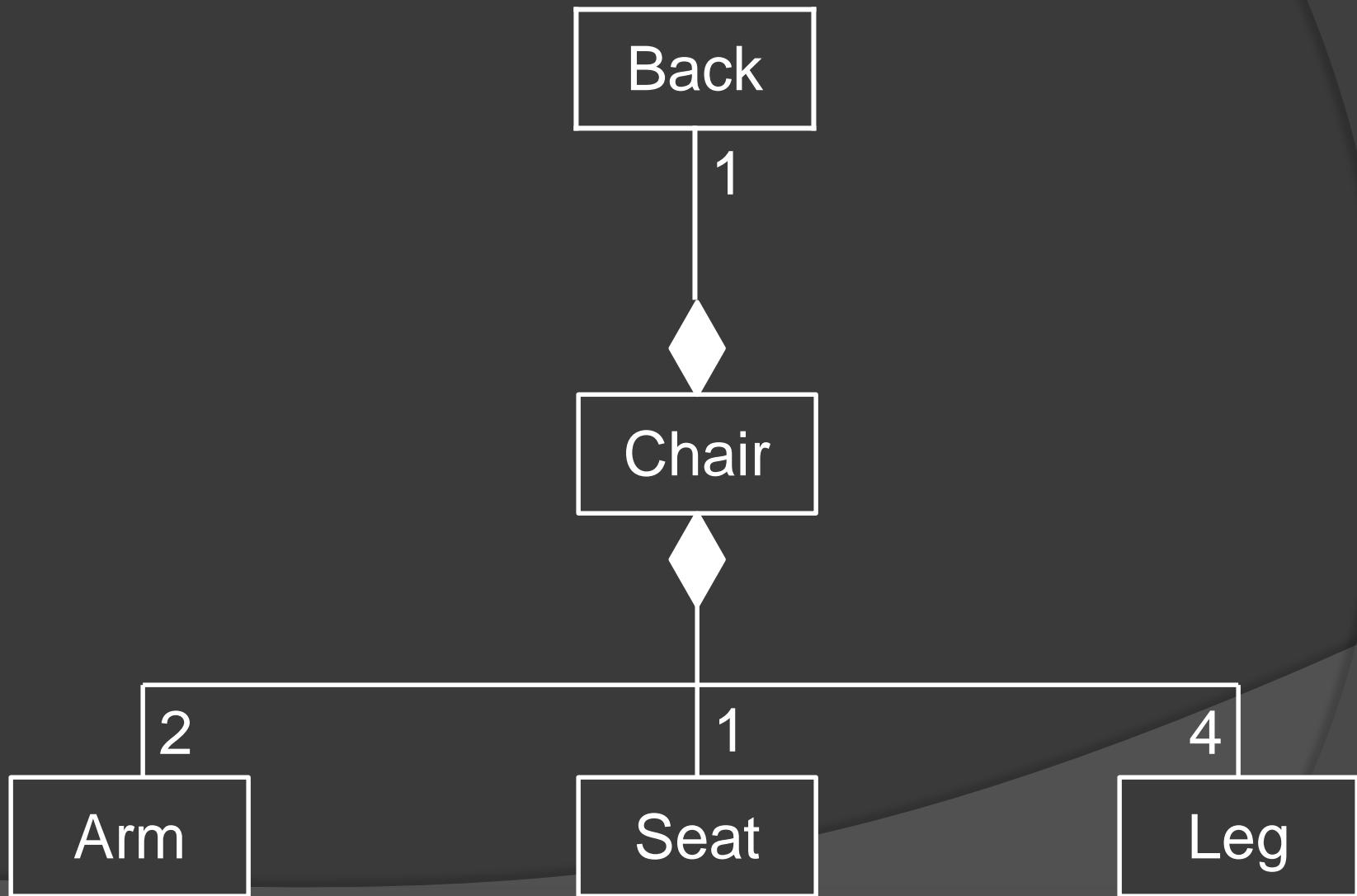
Composition

- An object may be composed of other smaller objects
- The relationship between the “part” objects and the “whole” object is known as Composition
- Composition is represented by a line with a filled-diamond head towards the composer object

Example – Composition of Ali



Example – Composition of Chair



Composition is Stronger

- Composition is a stronger relationship, because
 - Composed object becomes a part of the composer
 - Composed object can't exist independently

Example – Composition is Stronger

- Ali is made up of different body parts
- They can't exist independent of Ali

Example – Composition is Stronger

- Chair's body is made up of different parts
- They can't exist independently

```
class Engine {  
    private String engineType;  
  
    public Engine(String engineType) {  
        this.engineType = engineType;  
    }  
  
    public String getEngineType() {  
        return engineType;  
    }  
}
```

```
// Class representing a Wheel  
class Wheel {  
    private int size;  
  
    public Wheel(int size) {  
        this.size = size;  
    }  
  
    public int getSize() {  
        return size;  
    }  
}
```

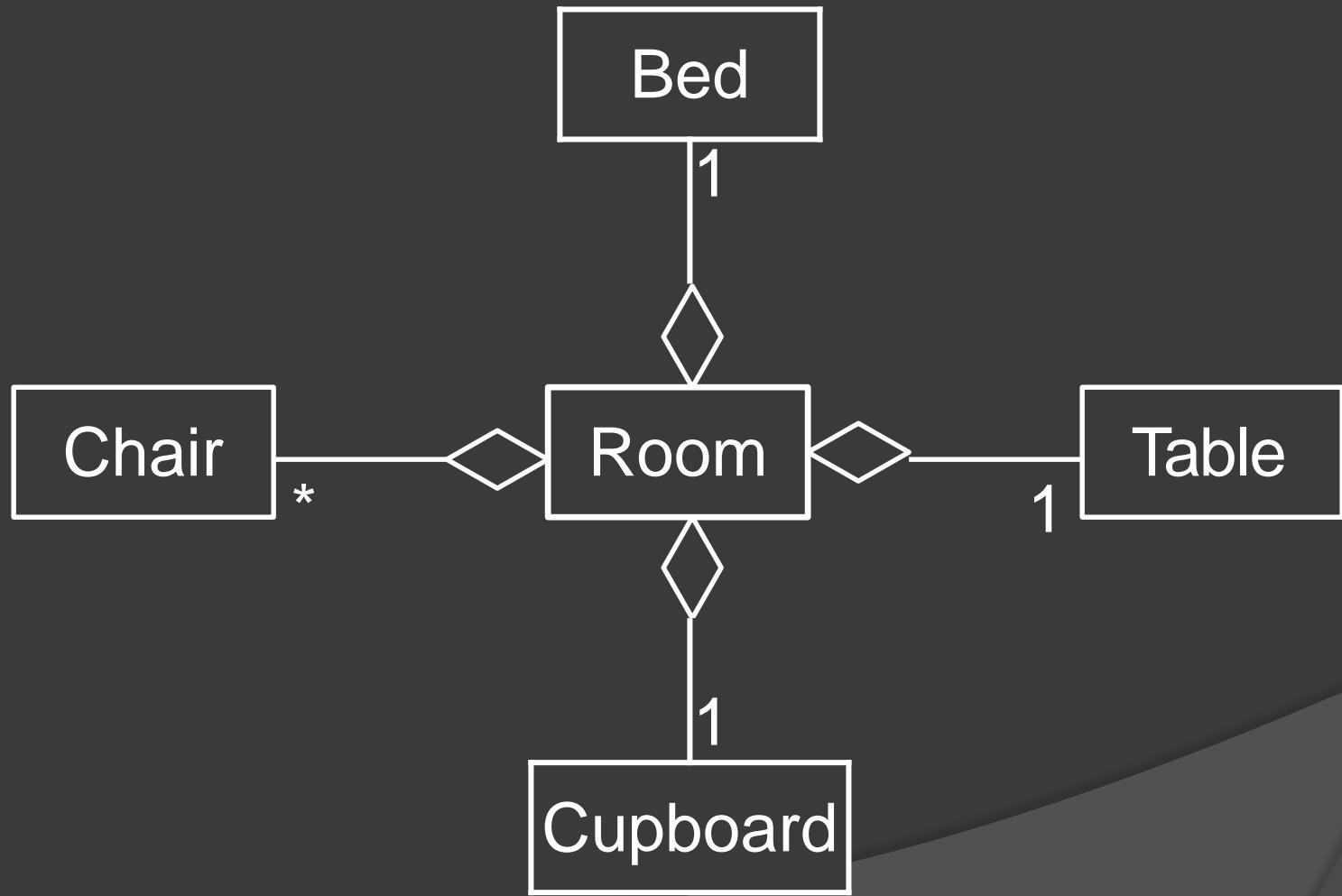
```
class Car {  
    private Engine engine;  
    // Composition: Car "has-a" Engine  
    private Wheel[] wheels;  
    // Composition: Car "has-a" array of Wheels  
  
    public Car(String engineType, int wheelSize) {  
        this.engine = new Engine(engineType);  
        // Creating an Engine instance  
        this.wheels = new Wheel[4];  
        // Array for 4 wheels  
        for (int i = 0; i < wheels.length; i++) {  
            wheels[i] = new Wheel(wheelSize);  
        }  
        // Creating Wheel instances  
    }  
  
    public void displayDetails() {  
        System.out.println("Car with " +  
            engine.getEngineType() + " engine and wheels of  
            size " + wheels[0].getSize());  
    }  
}
```

```
// Main class to demonstrate composition
public class CompositionExample {
    public static void main(String[] args) {
        Car myCar = new Car("V8", 17);
        // Create a Car with a V8 engine and wheels of size 17
        myCar.displayDetails();      // Display car details
    }
}
```

Aggregation

- An object may contain a collection (aggregate) of other objects
- The relationship between the container and the contained object is called aggregation
- Aggregation is represented by a line with unfilled-diamond head towards the container

Example - Aggregation



Example – Aggregation



Aggregation is Weaker

- Aggregation is weaker relationship, because
 - Aggregate object is not a part of the container
 - Aggregate object can exist independently

Example – Aggregation is Weaker

- Furniture is not an intrinsic part of room
- Furniture can be shifted to another room, and so can exist independent of a particular room

Example – Aggregation is Weaker

- A plant is not an intrinsic part of a garden
- It can be planted in some other garden, and so can exist independent of a particular garden

```
import java.util.ArrayList;
import java.util.List;

// Class representing a Book
class Book {
    private String title;
    private String author;

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }
}
```

```
// Class representing a Library
class Library {
    private List<Book> books;
    // Aggregation: Library "has-a" list of Books

    public Library() {
        this.books = new ArrayList<>();
    }

    public void addBook(Book book) {
        books.add(book);
    }
    // Adding a Book to the Library

    public void displayBooks() {
        System.out.println("Books in the Library:");
        for (Book book : books) {
            System.out.println("- " + book.getTitle() +
" by " + book.getAuthor());
        }
    }
}
```

```
// Main class to demonstrate aggregation
public class AggregationExample {
    public static void main(String[] args) {
        // Creating books
        Book book1 = new Book("1984", "George Orwell");
        Book book2 = new Book("To Kill a Mockingbird", "Harper Lee");

        // Creating library and adding books
        Library library = new Library();
        library.addBook(book1);
        library.addBook(book2);

        // Displaying books in the library
        library.displayBooks();

        // The books can exist independently of the library
        System.out.println(book1.getTitle() + " can exist outside of the library.");
    }
}
```

Inheritance

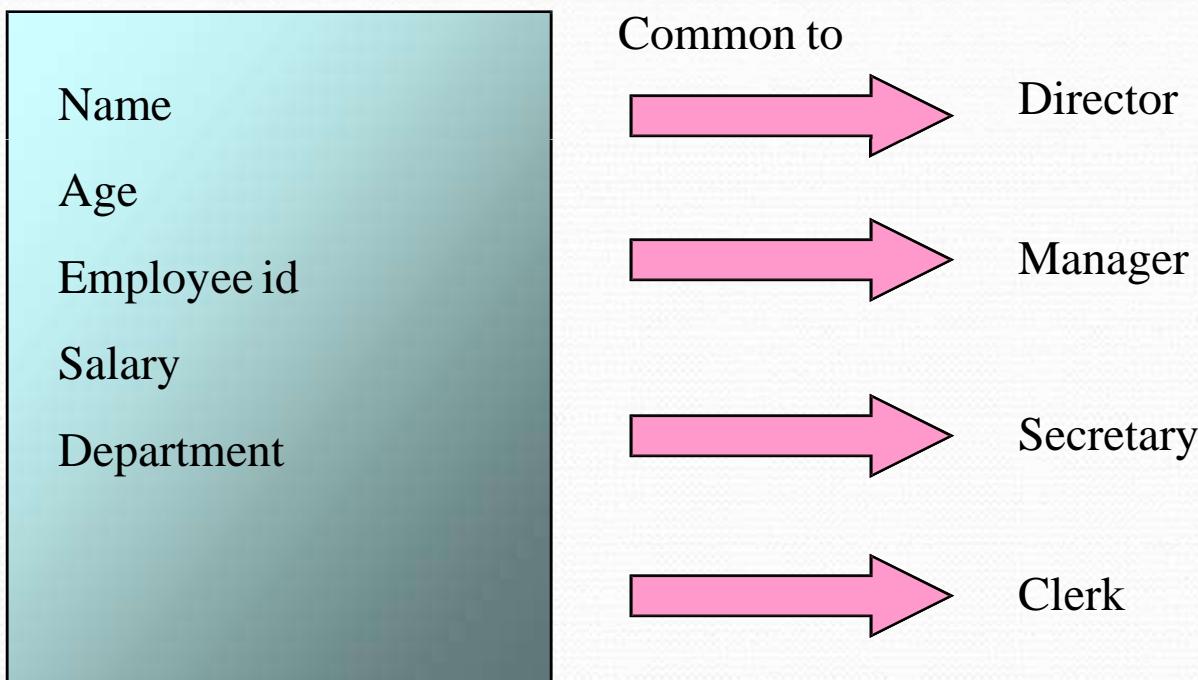
Session 8

Features of an Object

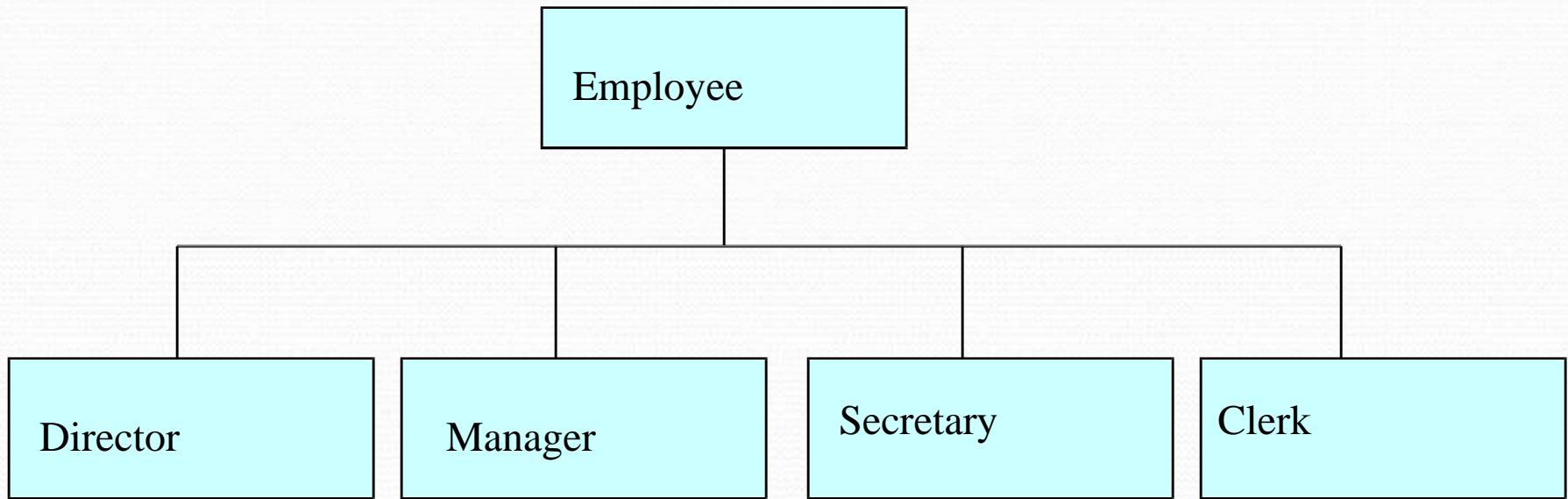
- An object must combine data and behavior
- Objects must focus on essential properties ignoring any accidental properties
- Object must hide the implementation details from the rest of the world. It may expose certain functionalities to other objects
- Object must understand messages from other objects and can pass messages to other objects .
- An object can be reused on any future project

Inheritance

Class Employee



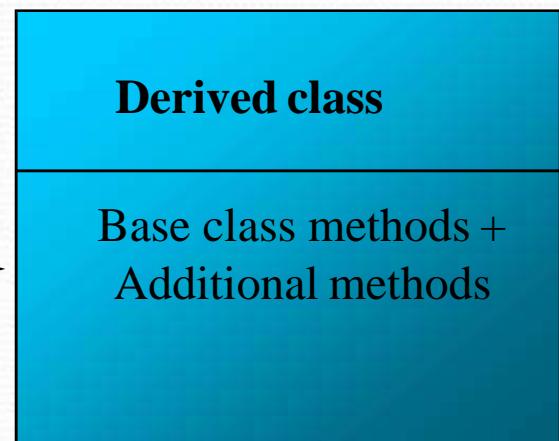
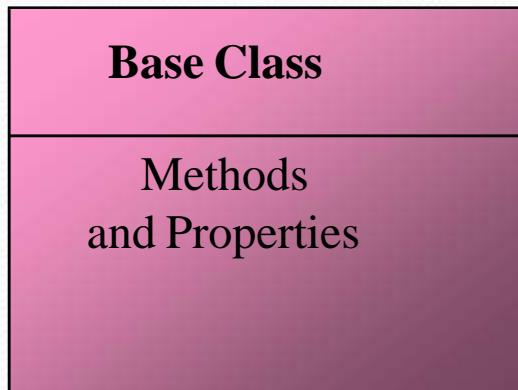
Inheritance



Each of the sub-classes is considered to be derived from the parent class Employee

Inheritance

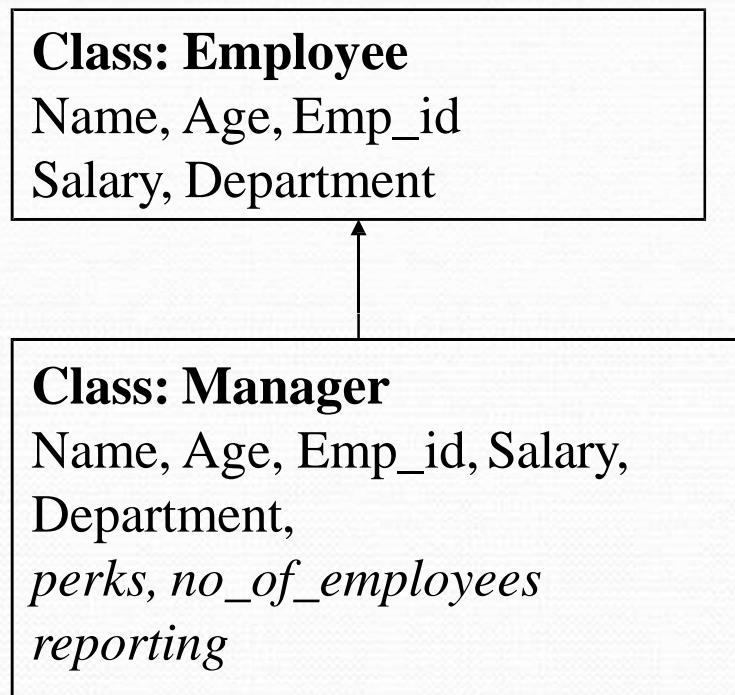
Inheritance is the property that allows the reuse of an existing class to build a new class



Advantages of Inheritance

- Reusability of code
- The base class need not be changed but can be adapted to suit the requirements in different applications
- Saves developers time and effort so that they need not to spend time to know the core technical facts

Generalization and Specialization



Base Class and Derived Class

```
class Employee{ //Base Class
    String Name;
    int Age;
    String emp_id;
    Float Salary;

    public void create()
    { ..... }
    public void resign()
    { ....... }

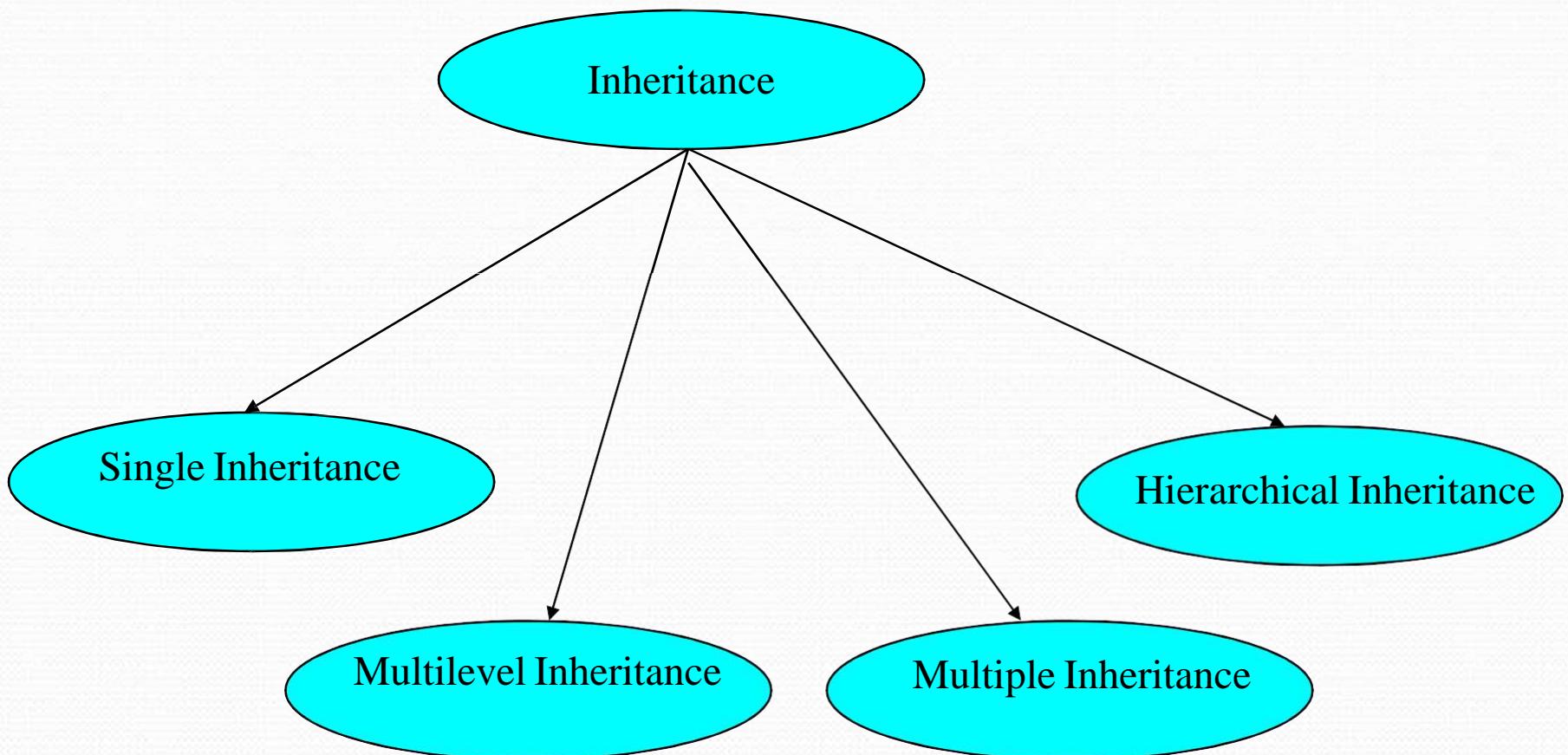
}
class Manger extends Employee{ //Subclass

    float perks;
    int no_of_employees;

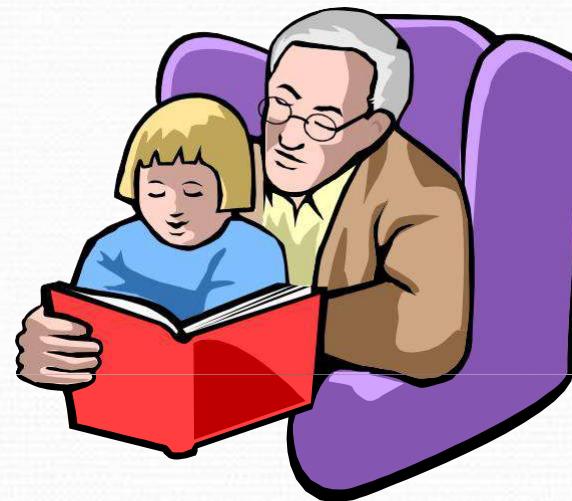
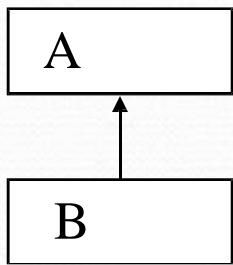
    public void report() { ..... }

}
```

Types of Inheritance



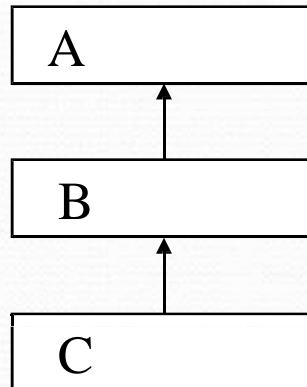
Single Inheritance



The son will have his own features as well as features of the father

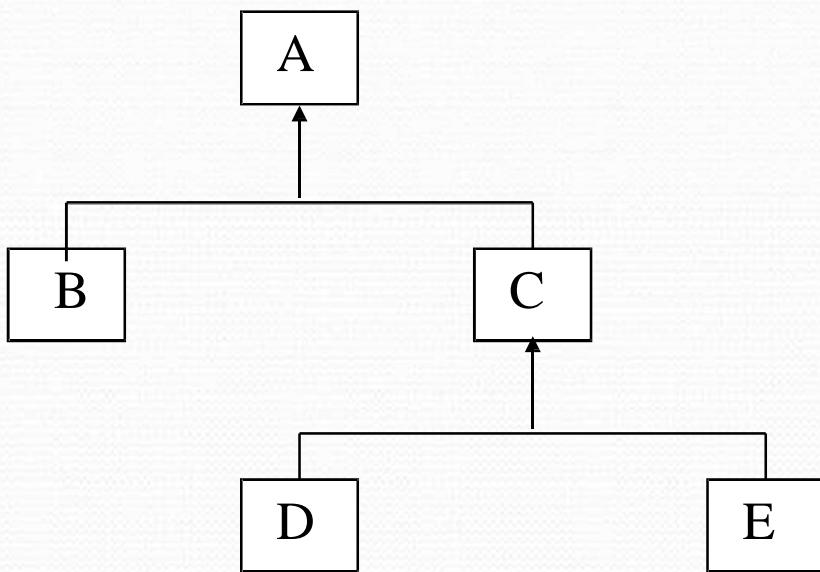
```
class A  
{ ..... }  
class B extends A  
{ .... }
```

Multilevel Inheritance



```
class A  
{ ..... }  
class B extends A  
{ ..... }  
class C extends B{  
..... }
```

Hierarchical Inheritance



```
class A  
{ ..... }  
class B extends A  
{ ..... }  
class C extends A  
{ ..... }  
class D extends C  
{ ..... }  
  
class E extends C  
{ ..... }
```

Single Level Inheritance in Java

```
class Container{
    int width;
    int height;
    int depth;
    Container() //default constructor
    {
        System.out.println("Default object created with zero
dimension");
        width=0;
        height=0;
        depth=0;
    }
    Container(Container C) //Pass object to constructor
```

Single Level Inheritance

```
{  
    width=C.width;  
    height=C.height;  
    depth=C.depth;  
}  
Container(int W, int H, int D) //Pass values to  
constructor  
{  
    width=W;  
    height=H;  
    depth=D;  
}  
Container(int C) //Constructor for Cube  
{  
    // In Cube width=height=depth  
    width=C;  
    height=C;  
    depth=C;  
}
```

Single Level Inheritance

```
long getVolume()
{
    return width*height*depth;
}
class Containerweight extends Container {
    int filledweight;
Containerweight(int W, int H, int D, int WT)
{
    width=W;
    height=H;
    depth=D;
    filledweight=WT;
    System.out.println("This is from the derived
class");
}
}
```

Single Level Inheritance

```
class DemoSingleInh
{
    public static void main(String args[])
    {
        //Creating Base class object created
        Containerweight containerX = new
        Containerweight(10,20,15,35);
        long vol;
        vol = containerX.getVolume();
        System.out.println("Volume of container = " +
        vol);
        System.out.println("Weight of container = " +
        containerX.filledweight);
    }
}
```

Multi-level Inheritance

```
class Container{
    int width;
    int height;
    int depth;
    Container() //default constructor
    {
        System.out.println("Default object created with zero
dimension");
        width=0;
        height=0;
        depth=0;
    }
    Container(Container C) //Pass object to constructor
    {
        width=C.width;
        height=C.height;
        depth=C.depth;
    }
}
```

Multi-level Inheritance

```
Container(int W, int H, int D)//Pass values to constructor
{
    width=W;
    height=H;
    depth=D;
}
Container(int C) //Constructor for Cube
{
    // In Cube all width=height=depth
    width=C;
    height=C;
    depth=C;
}
long getVolume()
{
    return width*height*depth; }
```

Multi-level Inheritance

```
}

class Containerweight extends Container
{
    int filledweight;

    Containerweight(int W, int H, int D, int WT)
    {
        width=W;
        height=H;
        depth=D;
        filledweight=WT;
        System.out.println("This is from the derived class");
    }
}

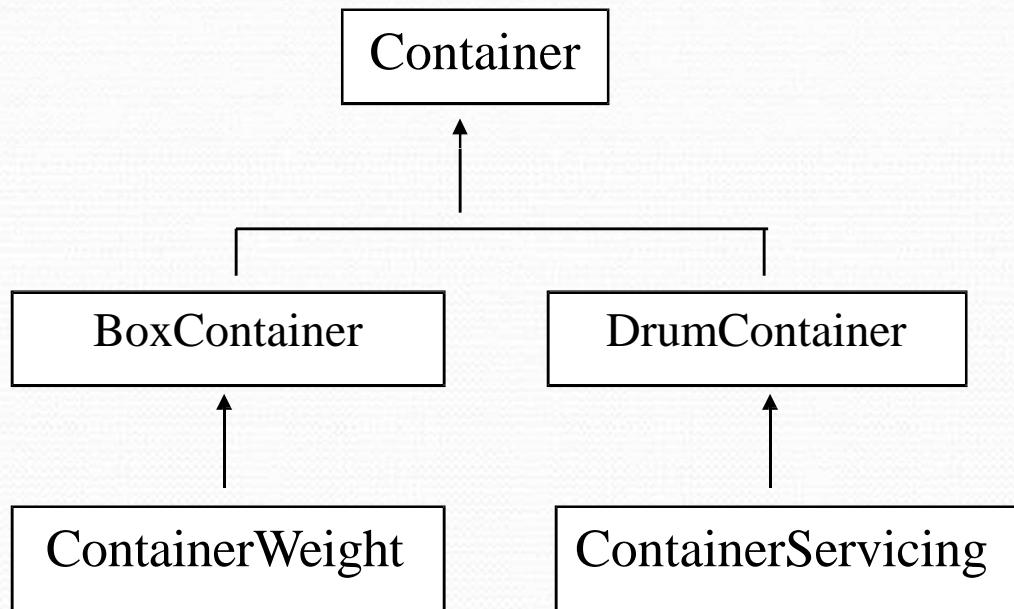
class Shipment extends Containerweight {
    int cost;
    Shipment(int W, int H, int D, int WT,int C) {
```

Multi-level Inheritance

```
        width=W;
        height=H;
        depth=D;
        filledweight=WT;
        cost=C;
        System.out.println("This is from the derived class");
    }
}
class DemoMultilevelInh{
    public static void main(String args[])
    {
        Shipment shipcontainerX = new Shipment(5,10,7,20,4);
        long vol;
        vol = shipcontainerX.getVolume();
        System.out.println("Volume of container = " + vol);
        System.out.println("Weight of container = " +
        shipcontainerX.weight);

    }
}
```

Hierarchical Inheritance



Hierarchical Inheritance

```
class Container
{
    String containerNo;
    Container(String S)
    { ..... }
    SetContainerNo(String S) { ..... }

    }
    getContainerNo()
    {
        return containerNo;
    }
}
```

Hierarchical Inheritance

```
class BoxContainer extends Container{
    int width;
    int height;
    int depth;
    Boxcotainer() { ..... } //default constructor
Boxcontainer(int W, int H, int D, String S)//Pass values to
constructor
{.....}
long getVolume()
{
    ..... .
}
}
```

Hierarchical Inheritance

```
}

class Containerweight extends BoxContainer
{
    int filledweight;
Containerweight(int W, int H, int D, int WT)
{ ..... }
}

class ContainerServicing extends BoxContainer {
    int servicecost;
        Shipment(.....)
{
    ..... .
}
int getServiceCost() {.....}

.....
```

Rules for Access modifiers

The methods of the derived class can access members of the base class if its members are public.

The private keyword makes a member of a class really private. Because the derived class members cannot access the private members of the base class.

The scope of the base class has to be broader than that of derived class. For example, If the base class has private as its access specifier then the derived class cannot have the access specifier as public.

Class members can always be accessed by methods of their own class, whether the members are private or public. But the inherited class's object can access base class members only if the members are public or protected.

Rules for Access modifiers

Protected members of a class can be accessed by objects or functions from its sub-classes. However, the difference between them appears only in derived classes.

Private members of a class cannot be derived, only public and protected members of a class can be derived"

Access modifiers

```
class Employee{                                //base class
    private int privA;
    protected int protA;
    public int pubA;

    public static void main(String args[])
    {
        Employee emp = new Employee();          //Object of base class type
        emp.privA = 1;                          //valid
        emp.protA = 1;                          //valid
        emp.pubA = 1;                           //valid
    }
}
```

Access modifiers

```
Manager mgr = new Manager();      //object of derived class
    mgr.privA = 1;                //error:not accessible
    mgr.protA = 1;                //valid
    mgr.pubA = 1;                //valid

}
class Manager extends Employee{      //derived class
    void fn()
    { int a;
        a = privA;              //error:not accessible
        a = protA;                //valid
        a = pubA;                //valid
    }
}
```

Using Super

In Java, the super keyword is used to refer to the superclass (parent class) of the current object. It can be used to access the superclass's members (fields and methods) from within a subclass (child class).

Using Super we can save extra lines of coding

Using Super

```
class Container{ int width;
    int height;
    int depth;
    Container() //default constructor
    {
        .....
    }
    Container(Container C) {
width=C.width;
height=C.height;
depth=C.depth;
    }
    Container(int W, int H, int D) {
width=W;
height=H;
depth=D;
    }
```

Using Super

```
Container(int C) {  
    width=C;  
    height=C;  
    depth=C;  
}  
long getVolume()  
{ ..... }  
}  
}  
}  
class Containerweight extends Container {  
    int filledweight;  
Containerweight() {  
    super();  
    filledweight=0;  
    System.out.println("Calling default constructor of Base  
class from derived class");  
}
```

Using Super

```
Containerweight(int W, int H, int D, int WT) {  
    super(W,H,D);  
    filledweight=WT;  
    System.out.println("Calling parameterised  
constructor of Base class from derived class");  
}  
  
Containerweight(Containerweight cobj) {  
    super(cobj);  
    filledweight=cobj.filledweight;  
    System.out.println("Calling constructor of Base  
class passing object from derived class");  
}  
}
```

Using Superclass variable

```
class DemoSingleInh {  
    public static void main(String args[]) {  
        Containerweight containerX = new Containerweight(10,20,15,35);  
        Container Cobj=new Container();  
        long vol;  
        vol = containerX.getVolume();  
        System.out.println("Volume of container = " + vol);  
        System.out.println("Weight of container = " +  
        containerX.filledweight);  
        Cobj=containerX; //assign object of subclass to a super class reference.  
        vol = Cobj.getVolume();  
        System.out.println("Volume of container(duplicate)= "+ vol);  
        /*System.out.println("Weight of container(duplicate) =" +  
        Cobj.filledweight); */  
    }  
}
```

Using Super keyword

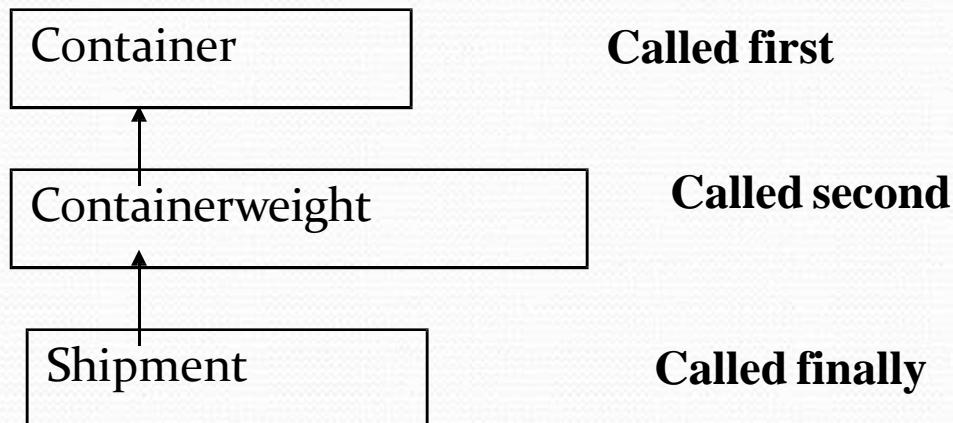
```
public class Room {  
    double height;  
    double length;  
    double width;  
}  
class bedroom extends Room  
{  
    bedRoom() {  
        super.height //height in Room  
        //main method  
    ...  
    }  
}
```

The super can be used in another way – as reference. The super calls members of superclass from a derived class.

super.member

Calling sequence of Constructor

```
class ShowSequence
{
    public static void main(String args[])
    {
        //Creating Inherited class object
        Shipment shipcontainerX =
            new Shipment(5,10,7,20,4);
    }
}
```



CUSTOMER CLASS

```
import java.util.ArrayList;  
  
import java.util.List;  
  
public class Customer {  
  
    private String name;  
  
    private String email;  
  
    private List<Double> purchaseHistory;  
  
    public Customer(String name, String email) {  
  
        this.name = name;  
  
        this.email = email;  
  
        this.purchaseHistory = new ArrayList<>();  
    }  
  
    public void addPurchase(double amount) {  
  
        purchaseHistory.add(amount);  
    }  
}
```

```
    public double calculateTotalExpenditure() {  
  
        double total = 0;  
  
        for (double purchase : purchaseHistory) {  
  
            total += purchase;  
        }  
  
        return total;  
    }  
  
    public String getName() {  
  
        return name;  
    }  
  
    public String getEmail() {  
  
        return email;  
    }  
  
    public List<Double> getPurchaseHistory() {  
  
        return purchaseHistory;  
    }  
}
```

LOYAL CUSTOMER CLASS

```
import java.util.ArrayList;  
  
import java.util.List;  
  
class LoyalCustomer extends Customer {  
  
    private double discountRate;  
  
    public LoyalCustomer(String name, String  
email, double discountRate) {  
  
        super(name, email);  
  
        this.discountRate = discountRate;  
    }  
  
    public double applyDiscount(double amount)  
{  
    return amount - (amount * discountRate / 100);  
}  
  
    public double getDiscountRate()  
{  
    return discountRate;  
}
```

//In Java, the @Override annotation is used to indicate that a method is intended to override a method declared in a superclass.

// Override the addPurchase method to apply the discount before adding the purchase

```
@Override  
  
public void addPurchase(double amount) {  
  
    double discountedAmount =  
applyDiscount(amount);  
  
    super.addPurchase(discountedAmount); // Call  
the superclass method to add the discounted amount  
}  
  
public void setDiscountRate(double discountRate) {  
  
    this.discountRate = discountRate;  
}  
}
```

Main Class

```
import java.util.ArrayList;  
  
import java.util.List;  
  
public class Main {  
  
    public static void main(String[] args) {  
  
        Customer customer1 = new  
Customer("Muhammad Shafan",  
"ms@example.com");  
  
        customer1.addPurchase(200);  
        customer1.addPurchase(300);  
  
System.out.println("Total expenditure for " +  
customer1.getName() + ": " +  
customer1.calculateTotalExpenditure());
```

```
LoyalCustomer loyalCustomer = new  
LoyalCustomer("Fulchard Sofya",  
"fulchard@example.com", 10);  
  
loyalCustomer.addPurchase(200);  
loyalCustomer.addPurchase(300);  
  
System.out.println("Total expenditure for " +  
loyalCustomer.getName() + " after discount: " +  
loyalCustomer.calculateTotalExpenditure());  
}
```

Session - 9

Polymorphism

Polymorphism

- *Polymorphism* comes from Greek meaning “many forms.”
- There are three basic types of polymorphism
 1. Ad hoc polymorphism [Method Overloading]
 2. Parametric polymorphism [Template or Generic Type]
 3. Subtyping [Method Overriding]
- In Java, polymorphism refers to the dynamic binding mechanism that determines which method definition will be used when a method name has been overridden.
- Thus, polymorphism refers to dynamic binding.

Late Binding/Dynamic Binding

- Late binding or dynamic binding (run-time binding):
 - Method to be executed is determined at execution time, not compile time
- Polymorphism: to assign multiple meanings to the same method name
- Implemented using late binding
- Method overloading is resolved by the compiler (*early binding/static binding*)

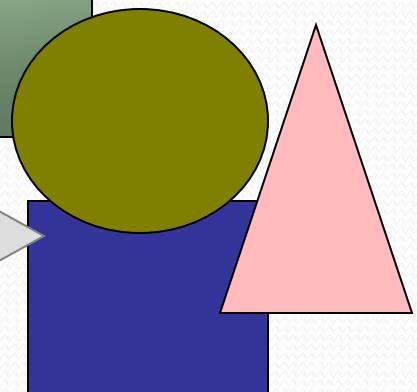
Polymorphic Function

- A **Polymorphic** function is one that has the **same name** for different classes of the same family, but has **different implementations/behaviour** for the various classes.
- In other words, polymorphism means **sending the same message (invoke/call member function)** to different objects of different classes in a hierarchy

Polymorphism and Method Overriding

Polymorphism is nothing but the ability of methods taking more than one form

Measuring Area of all these objects



Method overriding is one of the way to implement Polymorphism in object-oriented technology

Cont....

- There are 3 pre-requisite before we can apply polymorphism:
 1. Having a hierarchy of classes/implementing inheritance
 2. Having functions with same signatures in that hierarchy of classes, but each function in each class is having different implementation (function definition)
 3. Would like to use base-class pointer that points to objects in that hierarchy

Polymorphism [Methods & Fields]

- An object of a given class can have multiple forms: either as its declared class type, or as any subclass of it
- An object of an extended class can be used wherever the original class is used
- **Question:** given the fact that an object's actual class type may be different from its declared type, then when a method accesses an object's member which gets redefined in a subclass, then which member the method refers to (subclass's or super class's)?
 - When you **invoke a method** through an object reference, the *actual class of the object* decides which implementation is used.
 - When you **access a field**, the declared type of the **reference** decides which field to access.

Example

```
class SuperShow {  
    public String str = "SuperStr";  
    public void show( ) {  
        System.out.println("Super.show:" + str);  
    }  
}  
  
class ExtendShow extends SuperShow {  
    public String str = "ExtendedStr";  
    public void show( ) {  
        System.out.println("Extend.show:" + str);  
    }  
  
    public static void main (String[] args) {  
        ExtendShow ext = new ExtendShow( );  
        SuperShow sup = ext;  
        sup.show( );  
        ext.show( );  
        System.out.println("sup.str = " + sup.str);  
        System.out.println("ext.str = " + ext.str);  
    }  
}
```

Output:

Extend.show: ExtendStr
Extend.show: ExtendStr
sup.str = SuperStr
ext.str = ExtendStr

Virtual methods in Java

- In Java, all **non-static** methods are by default "**virtual functions.**"
- Only methods marked with the keyword **final**, which cannot be overridden, along with **private methods**, which are not inherited, are **non-virtual**.

Pointer example in c++

Pointers in classes

C++ Example

```
void main()
{
    Manager mgr;
    Employee* emp = &mgr;
    //valid: every Manager is an
    Employee

    Employee emp1;
    Manager* man = &emp1;
    //error: not every Employee
    is a Manager
}
```

C# & Java Example

```
void main()
{
    Employee emp = new Manager();
    //valid: every Manager is an Employee

    Manager man = new Employee();
    //error: not every Employee is a
    Manager
}
```

Polymorphism in C#

Method Overriding [using new]

- To override an existing method of the base class:
 - Declare a new method in the inherited class of the same name.
 - Prefix it with the **new** keyword.

Method Overriding [using new]

```
using System;

class A
{
    public void Driver()
    {
        Console.WriteLine("This is the Driver method of the class A");
    }
}

class B : A
{
    new public void Driver()
    {
        Console.WriteLine("This is the Overridden Driver method of the class B");
    }
}

class Test
{
    public static void Main()
    {
        B objB = new B();

        objB.Driver();
    }
}
```

This is the Overridden Driver method of the class B

Cont....

```
using System;
class A
{
    public void driver()
    {
        Console.WriteLine("Driver from A");
    }
}
class B:A
{
    new public void driver()
    {
        Console.WriteLine("Driver from B");
    }
}
class Program
{
    static void Main(string[] args)
    {
        A a = new B();
        a.driver();
    }
}
```

Driver from A

Method Overriding [using virtual & override]

- In C# & C++ Polymorphism is achieved using virtual methods.

virtual return_type functionName(argument list);

Method Overriding [using virtual & override]

```
class A
{
    virtual public void driver()
    {
        Console.WriteLine("Driver from A");
    }
}
class B:A
{
    override public void driver()
    {
        Console.WriteLine("Driver from B");
    }
}
class Program
{
    static void Main(string[] args)
    {
        A a = new B();
        a.driver();
    }
}
```

Driver from B

Polymorphism in C#

- In C# & C++ Polymorphism is achieved using virtual methods.

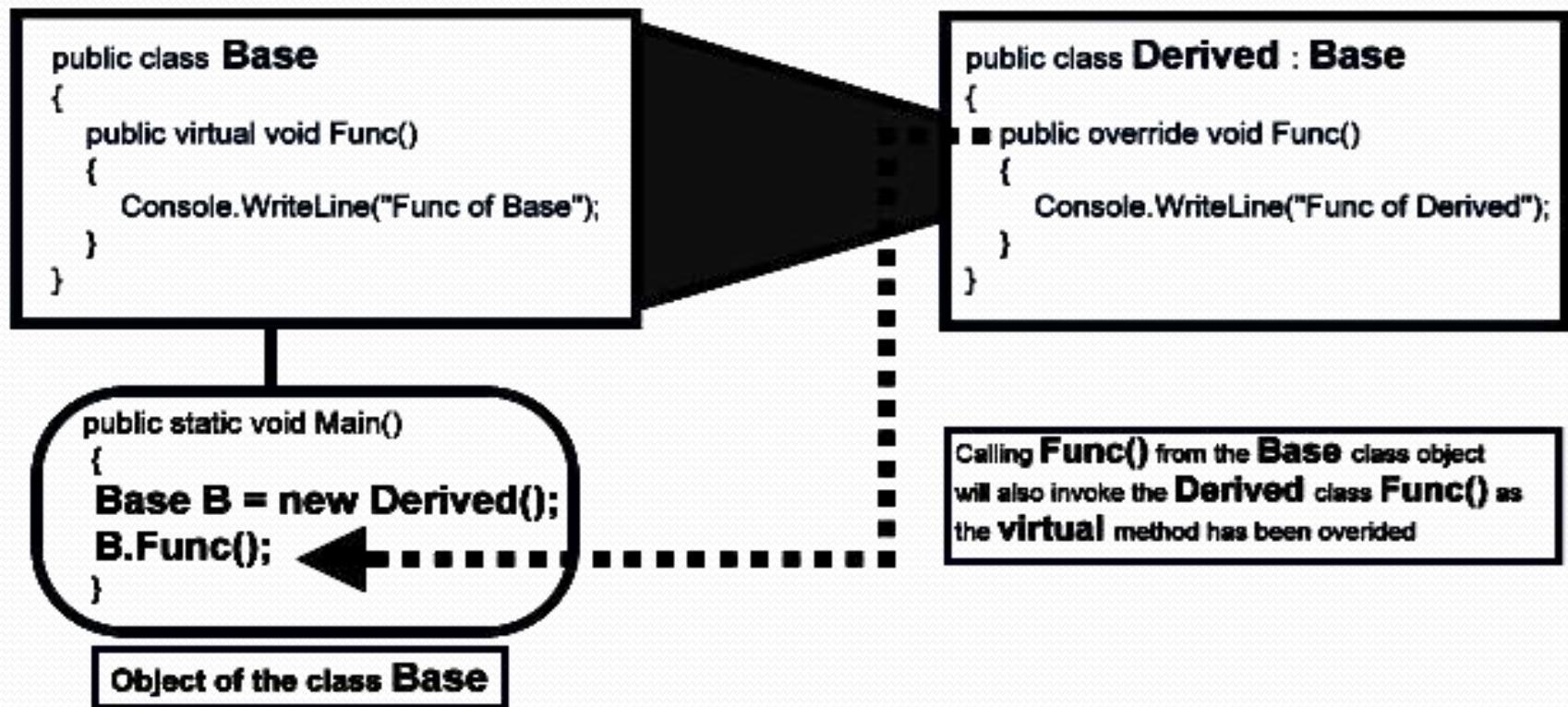
```
class DrawObj
{
    public virtual void Draw()
    {
        System.Console.WriteLine("This is the Virtual Draw
method");
    }
}
```

- Polymorphism allows us to implement the derived class methods during runtime.

- virtual -> override
- non-virtual -> redefine

Polymorphism in C#

- Virtual functions come in handy when we need to call the derived class method from an object of the base class.



Polymorphism in C#

```
public class Line : DrawObj
{
    public override void Draw()
    {
        System.Console.WriteLine("This is the Draw() method of
line");
    }
}

public class Circle : DrawObj
{
    public override void Draw()
    {
        System.Console.WriteLine("This is the Draw() method of
Circle ");
    }
}

public class Square : DrawObj
{
    public override void Draw()
    {
        System.Console.WriteLine("This is the Draw() method of
Square ");
    }
}
```

Polymorphism in C#

```
public class Test
{
    public static void Main()
    {
        DrawObj[] ObjD = new DrawObj[4];

        ObjD[0] = new DrawObj();
        ObjD[1] = new Line();
        ObjD[2] = new Circle();
        ObjD[3] = new Square();

        foreach (DrawObj Iterated in ObjD)
        {
            Iterated.Draw();
        }
    }
}
```

This is the Virtual Draw method
This is the Draw() method of line
This is the Draw() method of Circle
This is the Draw() method of Square

Polymorphism in C#

```
class A
{
    public int MethodA()
    {
        return(MethodB () *MethodC ());
    }

    public virtual int MethodB ()
    {
        return(10);
    }

    public int MethodC ()
    {
        return(20);
    }
}

class B : A
{
    public override int MethodB()
    {
        return(30);
    }
}

class Test
{
    public static void Main()
    {
        B ObjB = new B();
        System.Console.WriteLine(ObjB.MethodA());
    }
}
```

Output

600

Static binding

- *Static binding* means that the legality of a member function invocation is checked at the earliest possible moment: by the compiler at compile time.
- The compiler uses the static type of the pointer to determine whether the member function invocation is legal.

Dynamic binding

- *Dynamic binding* means that the legality of a member function invocation is determined at the last possible moment: based on the dynamic type of the object at run time.
- It is called "dynamic binding" because the binding to the code that actually gets called is accomplished dynamically (at run time).

Example Static / Dynamic

Polymorphic Pointers

- A reference of a parent class is allowed to point to an object of the child class. E.g.

```
class Vehicle {  
    // ...  
}  
class Car : Vehicle {  
    // ...  
}  
// ...  
Vehicle vp = new Car();
```

Overriding Methods

- Methods in the parent class can be redefined in the child class.

```
class Vehicle {  
    void move(int i){....}  
}  
class Car : Vehicle {  
    void move(int i) {....}  
}  
// ...  
Vehicle vp = new Car();  
vp.move(100);
```

Overriding Methods

- Methods in the parent class can be redefined in the child class.

```
class Vehicle {  
    void move(int i){...}  
}  
class Car : Vehicle {  
    void move(int i){...}  
}  
// ...  
Vehicle vp = new Car();  
vp.move(100);
```

BUT:

- Which of these two move() methods will be called?

Overriding Methods

- Methods in the parent class can be redefined in the child class.

```
class Vehicle {  
    void move(int i){.....}  
}  
class Car : Vehicle {  
    void move(int i){.....}  
}  
// ...  
Vehicle vp = new Car();  
vp.move(100);
```

static binding

Overriding Methods

- Methods in the parent class can be redefined in the child class.

```
class Vehicle {  
    void move(int i){.....}  
}  
class Car : Vehicle {  
    void move(int i){.....}  
}  
// ...  
Vehicle vp = new Car();  
vp.move(100);
```

dynamic binding

Overriding Methods

- Methods in the parent class can be redefined in the child class.

```
class Vehicle {  
    void move(int i){.....}  
}  
class Car : Vehicle {  
    new void move(int i){.....}  
}  
// ...  
Vehicle vp = new Car();  
vp.move(100);
```

static binding!

- Without virtual keyword

As vp is of type pointer to a Vehicle, the method of the Vehicle is called.

Overriding Methods -The virtual keyword

- Methods in the parent class can be redefined in the child class.

```
class Vehicle {  
    virtual void move(int i){.....}  
}  
class Car : Vehicle {  
    override void move(int i){.....}  
}  
// ...  
Vehicle vp = new Car();  
vp.move(100);
```

dynamic binding!

The keyword `virtual` allows the use of dynamic binding.

As `vp` points to a `Car` object the method of the `Car` is called

Advantage of Polymorphism [Software Extension]

- Polymorphism promotes extensibility: Software that invokes polymorphic behavior is independent of the object types to which messages are sent.
- New object types that can respond to existing method calls can be incorporated into a system without requiring modification of the base system.
- Only client code that instantiates new objects must be modified to accommodate new types.
- It allows system to evolve over time, meeting the needs of an ever-changing application

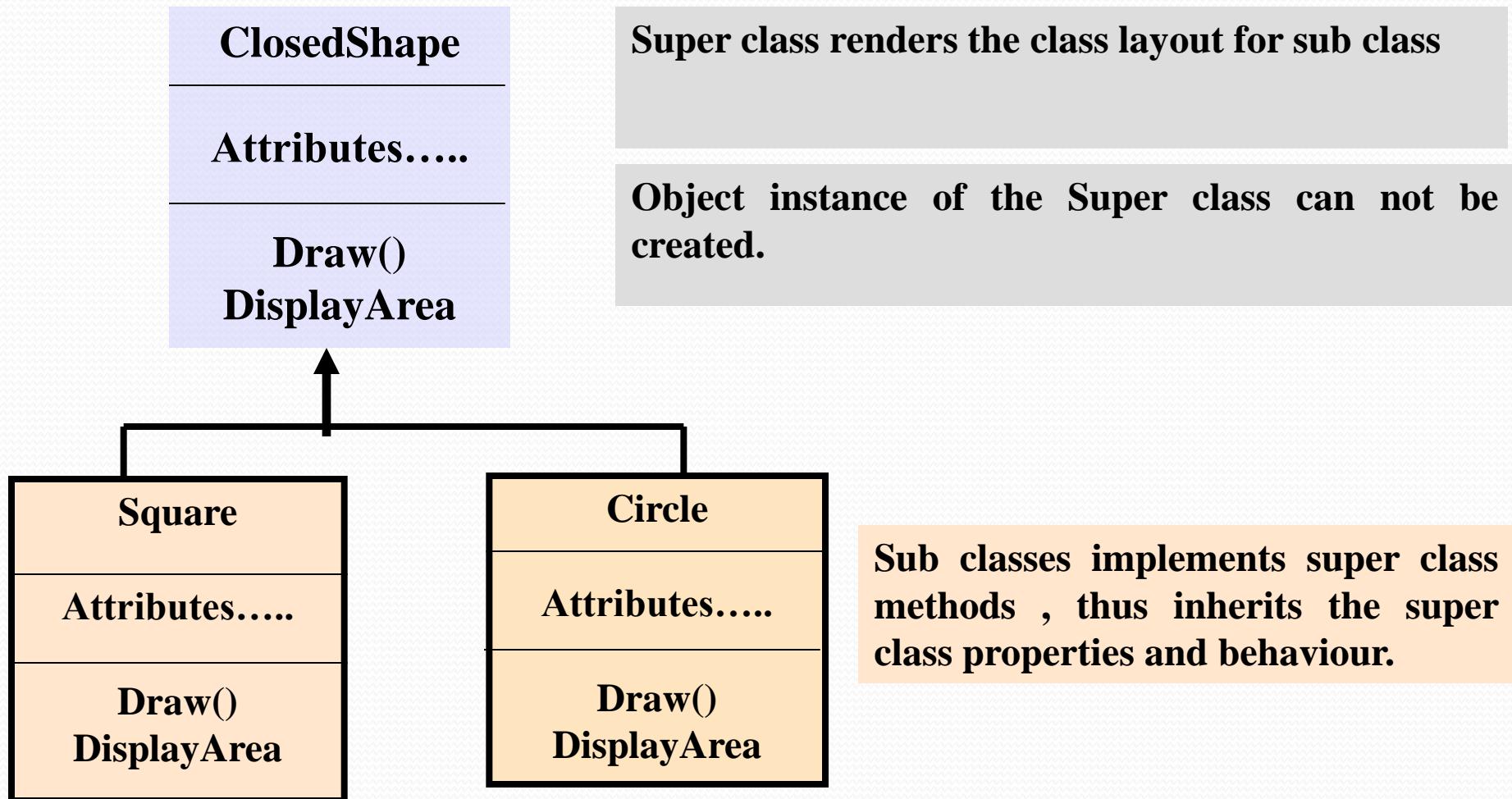
Session - 10

Abstract Classes and Interfaces

Abstract Classes

- If a class contains one or more abstract methods, then the class has to be declared as an abstract class
- Abstract classes provide the basic structure to its sub-classes when it is inherited
- The abstract class cannot be instantiated using new operator
- The constructor of an abstract class cannot be declared abstract

Abstract Class



Abstract Class – An example

```
abstract class ClosedShapes
{
.....// data members
..... //Constructors for circle, rectangle ...

abstract void draw();           //function in the base class
abstract void displayarea();
}

class Circle extends ClosedShapes{
    Circle(int r)    {
        super(r);    // calling super class constructor
    }
    void draw() {
        System.out.println("Draw Circle with radius " +radius);
    }
    void displayarea() {
        System.out.println("Area of Circle = " + 3.14*radius*radius);
    }
}
```

Abstract Classes

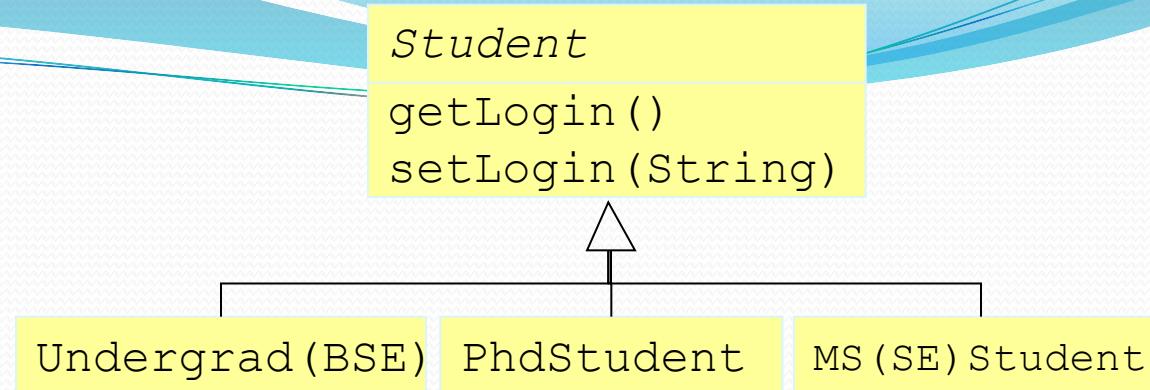
- Like classes, they introduce types.
 - but no objects can have as actual type the type of an abstract class.
- Why use them?
 - Because there is a set of common features and implementation for all derived classes but...
 - We want to prevent users from handling objects that are too generic
 - We cannot give a full implementation for the class

Italics indicates
abstract

AbstractClass

Example 1

- The problem:
 - Students are either undergraduate, PhD or MS (SE) .
 - We want to guarantee that nobody creates a Student object. The application always creates a specific kind of Student.
- The solution:
 - Declare Student as abstract.
- Why have the Student class in the first place?
 - A common implementation of common aspects of all students.
(e.g. setLogin() and getLogin())
 - To handle all students independently of their subclass using type Student and polymorphism.



Abstract Classes in Java

```
public abstract class Student {  
    protected String login, department, name;  
  
    public Student() {  
        login = ""; department = ""; name = "";  
    }  
  
    public void setLogin(String login) {  
        this.login = login;  
    }  
  
    public String getLogin() {  
        return login;  
    }  
}
```

PhdStudent is said
to be a **concrete class**

Student
getLogin()
setLogin(String)



PhdStudent

```
public class PhdStudent extends Student{  
    private String supervisor;  
  
    public void setSupervisor(String login) {  
        ...  
    }  
}
```

Shape

getArea () : double
setColour (int)

Example 2

Rectangle

Triangle



Circle

Hexagon

- The Problem
 - How do we calculate the area of an arbitrary shape?
 - We cannot allow Shape objects, because we cannot provide a reasonable implementation of *getArea()*;
- The Solution
 - So, we declare the Shape to be an **abstract** class.
 - Furthermore, we declare *getArea()* as an **abstract method** because it has no implementation
- Why have the Shape class in the first place?
 - Same reasons as for Student: a common implementation, a placeholder in the hierarchy and polymorphism.
 - Plus, that we want to force all shapes to provide an implementation for *getArea()*;

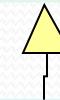
Abstract Methods in Java

```
public abstract class Shape {  
    final static int BLACK = 0;  
    private int colour;  
  
    public Shape() {  
        colour = BLACK;  
    }  
  
    public void setColour(int c) {  
        this.colour = c;  
    }  
  
    public abstract double getArea();  
}
```

Abstract methods
have no body

Shape

getArea(): double
setColour(int)



Circle

```
public class Circle extends Shape {  
    final static double PI = 3.1419;  
    private int radius;  
  
    public Circle(int r) {  
        radius = r;  
    }  
  
    @Override  
    public double getArea() {  
        return (radius^2)*PI;  
    }  
}
```

If Circle did not implement getArea() then
it would have to be declared abstract too!

Abstract Classes

- What are the differences between both examples?
- In Example 1
 - I **choose** to declare Student abstract because I think it is convenient to prevent the existence of plain Students
- In Example 2
 - I **must** declare Shape abstract because it lacks an implementation for getArea();

Using abstract classes

```
// Shape s = new Shape(); // ERROR  
Shape s = new Circle(4); // Ok  
double area = s.getArea(); // Ok - Remember polymorphism?  
Circle c = new Circle(3); // Ok  
c.setColour(GREEN); // Ok  
area = c.getArea(); // Ok
```

- Class *Shape* cannot be instantiated (it provides a partial implementation)
- Abstract methods can be called on an object of apparent type *Shape* (they are provided by *Circle*) (**Polymorphism**)

```
abstract class Account {  
    private String accountHolder;  
    private double balance;  
  
    public Account(String accountHolder, double initialBalance) {  
        this.accountHolder = accountHolder;  
        this.balance = initialBalance;  
    }  
  
    // Abstract methods  
    abstract void deposit(double amount);  
    abstract void withdraw(double amount);  
    abstract void displayAccountInfo();  
  
    // Concrete method to get the balance  
    public double getBalance() {  
        return balance;  
    }  
  
    protected void setBalance(double balance) {  
        this.balance = balance;  
    }  
  
    public String getAccountHolder() {  
        return accountHolder;  
    }  
}
```

```
class SavingsAccount extends Account {  
    private double interestRate;  
  
    public SavingsAccount(String accountHolder, double initialBalance, double interestRate) {  
        super(accountHolder, initialBalance);  
        this.interestRate = interestRate;  
    }  
  
    @Override  
    void deposit(double amount) {  
        setBalance(getBalance() + amount);  
        System.out.println("Deposited: " + amount + " to savings account.");  
    }  
  
    @Override  
    void withdraw(double amount) {  
        if (amount <= getBalance()) {  
            setBalance(getBalance() - amount);  
            System.out.println("Withdrew: " + amount + " from savings account.");  
        } else {  
            System.out.println("Insufficient funds in savings account.");  
        }  
    }  
    @Override  
    void displayAccountInfo() {  
        System.out.println("Savings Account Holder: " + getAccountHolder());  
        System.out.println("Balance: " + getBalance());  
        System.out.println("Interest Rate: " + interestRate + "%");  
    }  
}
```

```
class CheckingAccount extends Account {  
    private double overdraftLimit;  
  
    public CheckingAccount(String accountHolder, double initialBalance, double overdraftLimit) {  
        super(accountHolder, initialBalance);  
        this.overdraftLimit = overdraftLimit;  
    }  
    @Override  
    void deposit(double amount) {  
        setBalance(getBalance() + amount);  
        System.out.println("Deposited: " + amount + " to checking account.");  
    }  
    @Override  
    void withdraw(double amount) {  
        if (amount <= getBalance() + overdraftLimit) {  
            setBalance(getBalance() - amount);  
            System.out.println("Withdrew: " + amount + " from checking account.");  
        } else {  
            System.out.println("Withdrawal exceeds overdraft limit.");  
        }  
    }  
    @Override  
    void displayAccountInfo() {  
        System.out.println("Checking Account Holder: " + getAccountHolder());  
        System.out.println("Balance: " + getBalance());  
        System.out.println("Overdraft Limit: " + overdraftLimit);  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        // Create a SavingsAccount object  
        Account savings = new SavingsAccount("Alice", 1000.0, 2.5);  
        savings.displayAccountInfo();  
        savings.deposit(200);  
        savings.withdraw(50);  
        savings.displayAccountInfo();  
  
        // Create a CheckingAccount object  
        Account checking = new CheckingAccount("Bob", 500.0, 300.0);  
        checking.displayAccountInfo();  
        checking.deposit(100);  
        checking.withdraw(600);  
        checking.withdraw(100);  
        checking.displayAccountInfo();  
    }  
}
```

Abstract Base Classes – C#

- Abstract classes are classes that can be inherited from, but objects of that class cannot be created.
- C# allows creation of Abstract Base classes by an addition of the abstract modifier to the class definition.

Abstract Base Classes (2)

```
using System;

abstract class ABC
{
    public abstract void AFunc();

    public void BFunc()
    {
        Console.WriteLine("This is the BFunc() method!");
    }
}

class Derv : ABC
{
    public override void AFunc()
    {
        Console.WriteLine("This is the AFunc() method!");
    }
}

class Test
{
    static void Main()
    {
        Derv b = new Derv();
        ABC a = b;
        a.AFunc();
        b.BFunc();
    }
}
```

Interfaces

- An interface is a set of methods and constants that is identified with a name.
- They are similar to abstract classes
 - You cannot instantiate interfaces
 - An interface introduces types
 - But they are completely abstract (no implementation)
- Classes and abstract classes realize or implement interfaces.
 - They must have (at least) all the methods and constants of the interface with public visibility

interface
Clock
<i>MIDNIGHT:Time</i>
<i>setTime(Time):void</i>

Interfaces

Abstracting the implementation of a class enriches the data hiding principles.

Class and Object

Calculator

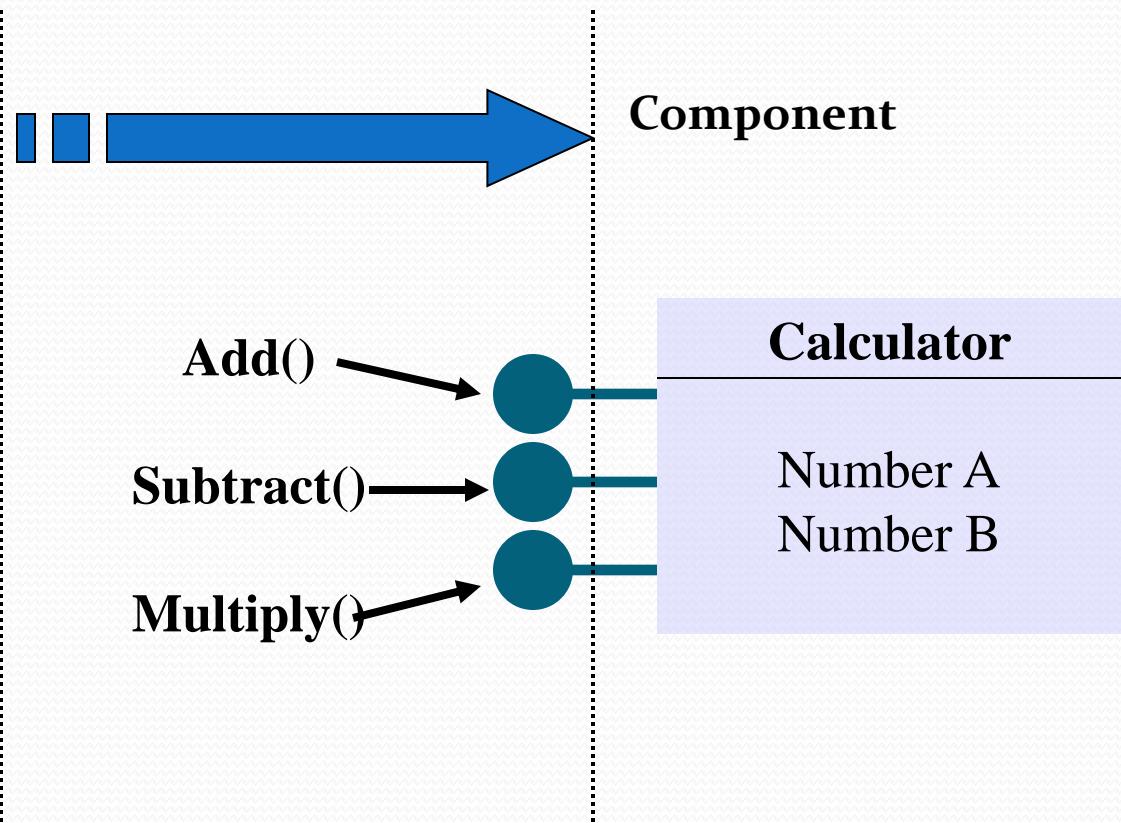
Number A
Number B

Add()
Subtract()
Multiply()

Component

Calculator

Number A
Number B



Features of Interface

- Are similar to a class but they do not contain instance variables and the methods contained in them
- Are similar to abstract classes with all methods declared as abstract
- Are support dynamic method resolution at run time
- Disconnect the definition of a method from the inheritance hierarchy
- Possible for classes

Defining an Interface

```
<access Specifier> interface <name>
{
    final <data type> variable name = value;
    <access specifier> <return type> method name(parameter list)
}
```

Example:

```
interface Student
{
    void Learn(String sub);
}
```

Implementing an Interface

```
interface Area
{
    final double pi=3.14;
    void displayarea();
}

class Circle implements Area
{
    private int radius;
    Circle(int r)
    {
        radius=r;
    }
    public void displayarea()
    {
        System.out.println("Area of Circle = " + pi*radius*radius);
    }
}
```

Implementing an Interface

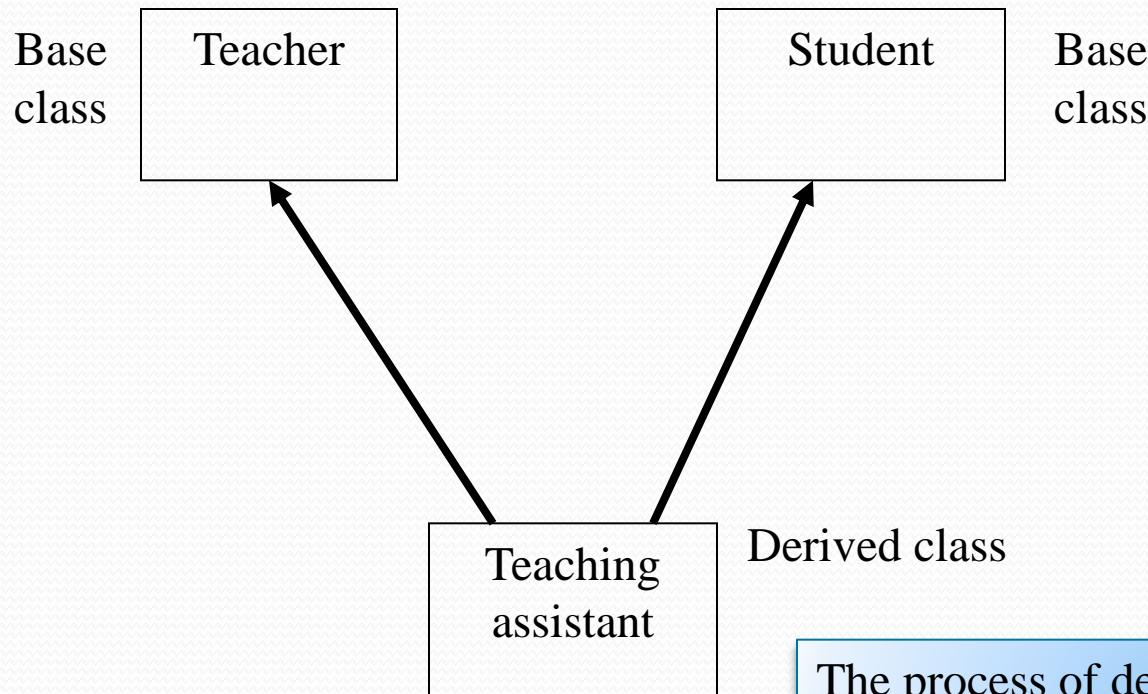
```
class Rectangle implements Area
{
    private int length;
    private int width;
    Rectangle(int l, int w)
    {
        length=l;
        width=w;
    }

    public void displayarea()
    {
        System.out.println("Area of Rectangle=" + length*width);
    }
}
```

Implementing an Interface

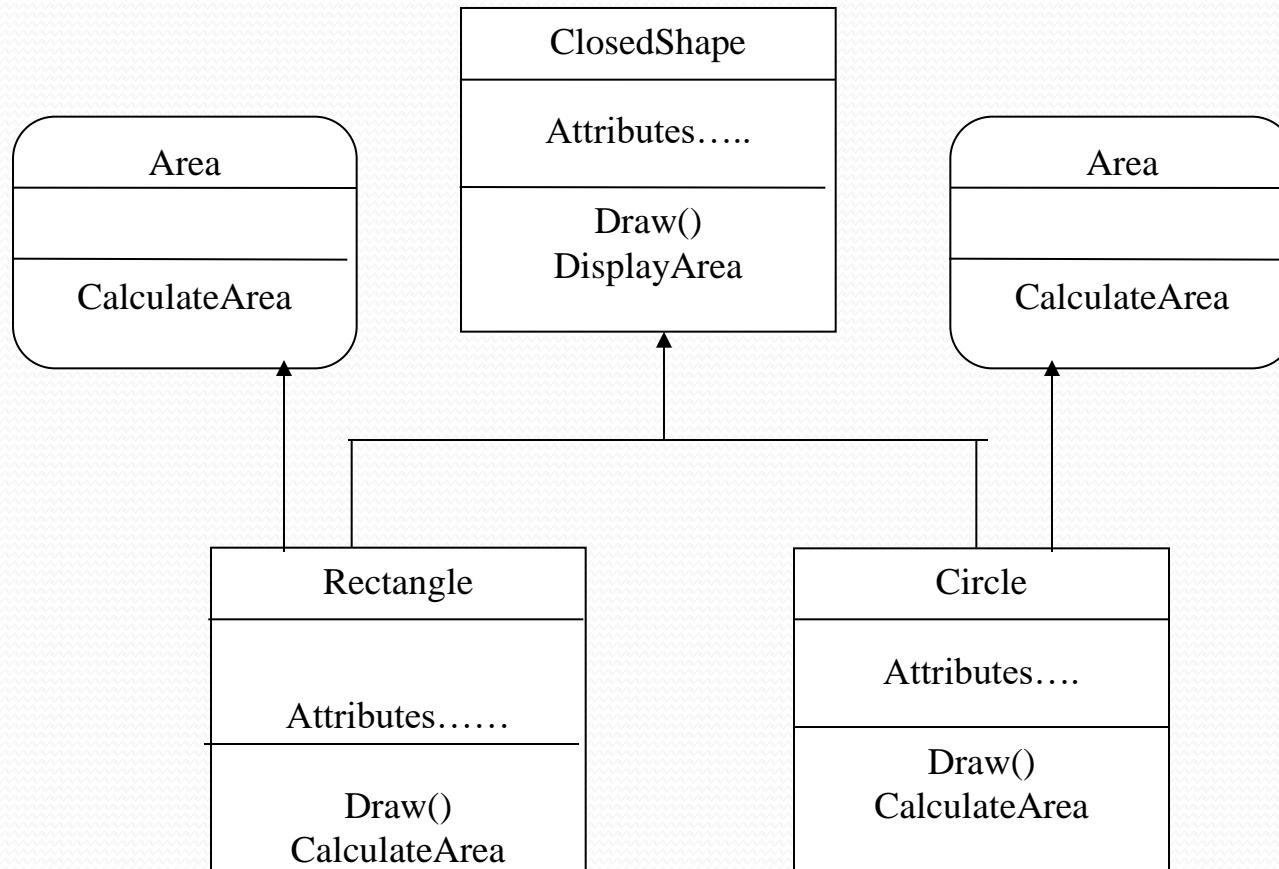
```
class DemoInterface
{
    public static void main(String args[])
    {
        Circle c = new Circle(5);
        Rectangle s = new Rectangle(10,20);
        Area ref;
        ref = c;
        c.displayarea();
        ref = s;
        s.displayarea();
    }
}
```

Multiple Inheritance



The process of deriving from more than one base class is called multiple inheritance

How Multiple Inheritance is implemented in Java



Important Issues on Interfaces

When interface is implemented through a class, the instance of that class can be created and stored in a variable of that interface type.

```
interface Infa {      void print(); }  
class Clsb implements Infa  
{  
    public void print()  
    {   System.out.println("This is implemented in class B");  
    }  
}  
  
class Test {  
public static void main(String args[]) {  
    Infa A=new Clsb();  
    A.print();  
}  
}
```

Important Issues on Interfaces

Interfaces can be extended. Therefore, one interface can be derived from another interface. Interfaces behave same as classes in inheritance hierarchy.

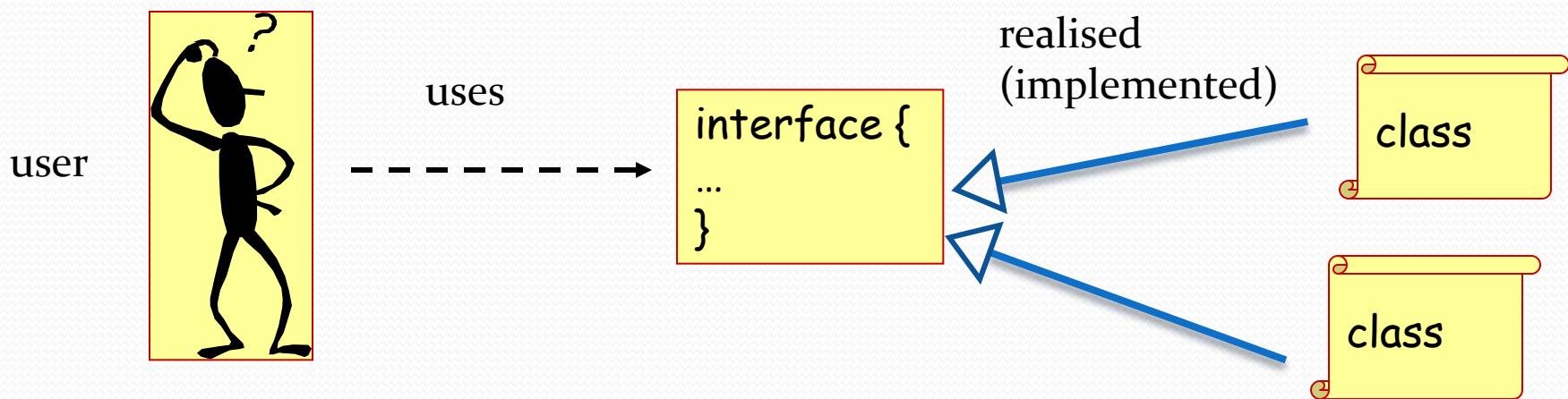
```
interface infA {  
    void printA();  
}  
interface infB extends infA {  
    void printB();  
}  
class ClsTest implements infB  
{  
    public void printA() {  
        System.out.println("This is declared in interface A");  
    }  
}
```

Important Issues on Interfaces

```
public void printB()
{
    System.out.println("This is declared in interface B");
}
class TestExtend
{
    public static void main(String args[])
    {
        ClsTest A=new ClsTest();
        A.printA();
        A.printB();
    }
}
```

Why use Interfaces?

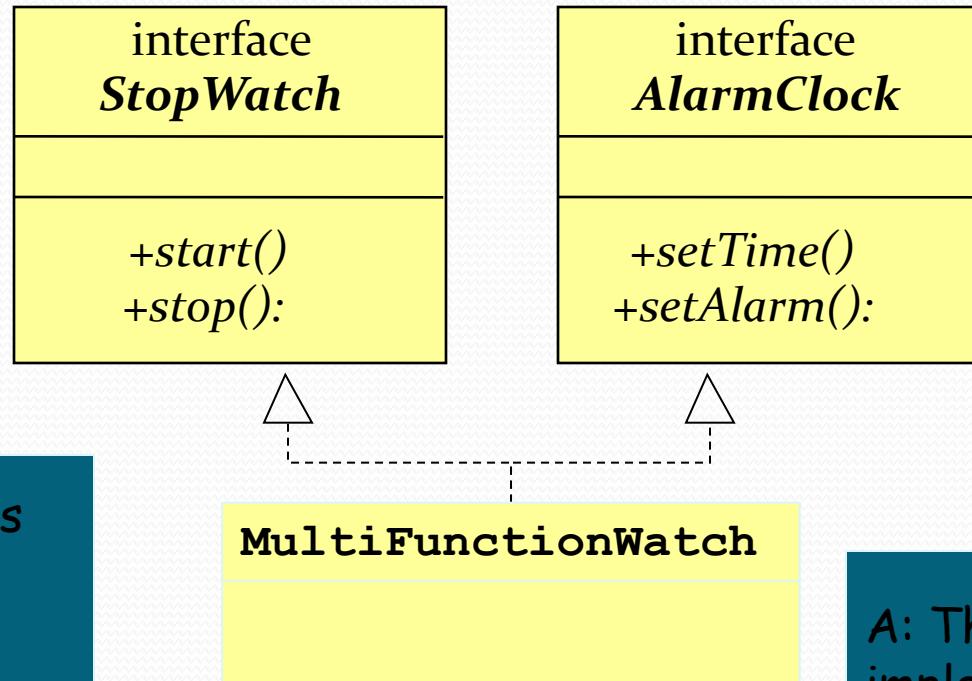
- To *separate (decouple)* the specification available to the user from implementation
 - I can use any class that implements the interface through the interface type (i.e. polymorphism)



- As a partial solution to Java's lack of multiple inheritance

Multiple Interfaces

- Classes are allowed to implement multiple interfaces



Q: Why is this
not the same
as multiple
inheritance?

A: There is no
implementation
to inherit

Review-Interfaces with C#

- An interface is a pure abstract base class.
- It can contain only abstract methods, and no method implementation.
- A class that implements a particular interface must implement the members listed by that interface.

```
public interface IPict
{
    int DeleteImage();
    void DisplayImage();
}
```

Interfaces (2)

```
public class MyImages : IPict
{
    public int DeleteImage()
    {
        System.Console.WriteLine("DeleteImage Implementation!");
        return(0);
    }

    public void DisplayImage()
    {
        System.Console.WriteLine("DisplayImage Implementation!");
    }
}

class Test
{
    static void Main()
    {
        MyImages m = new MyImages();

        m.DisplayImage();
        int t = m.DeleteImage();
    }
}
```

Interfaces

- If we merge the last two codes and compile them, we will get the following output:

```
DisplayImage Implementation!  
DeleteImage Implementation!
```

- Take another example:

```
public class BaseIO  
{  
    public void Open()  
    {  
        System.Console.WriteLine("This is the Open method of  
BaseIO");  
    }  
}
```

Interfaces (4)

- Now, if we need to inherit a class, **MyImages.....**

```
public class MyImages : BaseIO, IPict
{
    public int DeleteImage()
    {
        System.Console.WriteLine("DeleteImage Implementation!");
        return(0);
    }

    public void DisplayImage()
    {
        System.Console.WriteLine("DisplayImage Implementation!");
    }
}

class Test
{
    static void Main()
    {
        MyImages m = new MyImages();

        m.DisplayImage();
        int t = m.DeleteImage();

        m.Open();
    }
}
```

Interfaces (5)

- The output of the example is:

```
DisplayImage Implementation!  
DeleteImage Implementation!  
This is the Open method of BaseIO
```

Multiple Interface Implementation (1)

- C# allows multiple interface implementations.

```
public interface IPictManip
{
    void ApplyAlpha();
}
```

Multiple Interface Implementation (2)

```
public class MyImages : BaseIO, IPict, IPictManip
{
    public int DeleteImage()
    {
        System.Console.WriteLine("DeleteImage Implementation!");
        return(0);
    }

    public void DisplayImage()
    {
        System.Console.WriteLine("DisplayImage Implementation!");
    }

    public void ApplyAlpha()
    {
        System.Console.WriteLine("ApplyAlpha Implementation!");
    }
}

class Test
{
    static void Main()
    {
        MyImages m = new MyImages();

        m.DisplayImage();
        int t = m.DeleteImage();

        m.Open();

        m.ApplyAlpha();
    }
}
```

Output

```
DisplayImage Implementation!
DeleteImage Implementation!
This is the Open method of BaseIO
ApplyAlpha Implementation!
```

Explicit Interface Implementation

- Explicit interface implementation can be used when a method with same name is available in 2 interfaces.

```
public interface IPict
{
    int DeleteImage();
    void DisplayImage();
}

public interface IPictManip
{
    void ApplyBlending();
    void DisplayImage();
}

public class MyImages : BaseIO, IPict, IPictManip
{
    ...
    void IPict.DisplayImage()
    {
        System.Console.WriteLine("IPict Implementation of
DisplayImage");
    }

    void IPictManip.DisplayImage()
    {
        System.Console.WriteLine("IPictManip Implementation of
DisplayImage");
    }
    ...
}
```

Interface Inheritance

- New Interfaces can be created by combining together other interfaces.

```
interface IPictAll : IPict, IPictManip
{
    // More operations can be added if necessary (apart from that of
    IPict & IManip)
}
```

default method

- In Java, a **default method** is a feature introduced in Java 8 that allows you to define a method in an interface with a default implementation. This means that when a class implements the interface, it can either use the default implementation or override it with its own.
- In C#, the concept similar to Java's default methods in interfaces is achieved through **default interface methods**, introduced in C# 8.0. This feature allows you to provide a default implementation for methods in an interface.
- Default methods in java are defined using the **default** keyword in the interface.
- Default methods are defined within an interface using the **default** keyword (not explicitly required in C#), along with a method body.

Default Method [Java]

```
// Define an interface with a default method
public interface Vehicle {
    void start(); // Abstract method

    // Default method
    default void stop() {
        System.out.println("Vehicle has stopped.");
    }
}

// Implementing the interface in a class
public class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car has started.");
    }

    // Optionally, override the default method
    @Override
    public void stop() {
        System.out.println("Car has stopped.");
    }
}
```

```
// Another implementing class
public class Bike implements Vehicle {
    @Override
    public void start() {
        System.out.println("Bike has started.");
    }

    // Use the default implementation of stop
}

// Main class to test the implementation
public class Main {
    public static void main(String[] args) {
        Vehicle car = new Car();
        car.start(); // Output: Car has started.
        car.stop(); // Output: Car has stopped.

        Vehicle bike = new Bike();
        bike.start(); // Output: Bike has started.
        bike.stop(); // Output: Vehicle has stopped.
    }
}
```

Default Method [C#]

```
public interface IShape
{
    double GetArea(); // Abstract method
    double GetPerimeter()
    {
        return 0; // Default implementation (can be overridden)
    }
}

public class Circle : IShape
{
    private double radius;

    public Circle(double radius)
    {
        this.radius = radius;
    }

    public double GetArea()
    {
        return Math.PI * radius * radius;
    }

    public double GetPerimeter()
    {
        return 2 * Math.PI * radius;
    }
}
```

```
public class Rectangle : IShape
{
    private double width;
    private double height;

    public Rectangle(double width, double height)
    {
        this.width = width;
        this.height = height;
    }

    public double GetArea()
    {
        return width * height;
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        IShape circle = new Circle(5);
        Console.WriteLine($"Circle Area: {circle.GetArea()}");
    }
}
```

Abstract classes vs. Interfaces

- Can have data fields
- Methods may have an implementation
- Classes and abstract classes **extend** abstract classes.
- Class cannot extend multiple abstract classes
- Can only have constants
- Methods have **no** implementation
- Classes and abstract classes **implement** interfaces
- Interfaces can **extend multiple** interfaces
- A class can implement multiple interfaces

Session - 11

Serialization in JAVA

Storing Objects/Structures in Files

- Many programs need to save information between program runs
- Alternately, we may want one program to save information for later use by another program

Serialization of Objects

- Java provides a way to save objects directly
- Saving an object with this approach is called *serializing* the object
- **Serialization** in other languages can be *very* difficult, because objects may contain references to other objects. Java makes serialization (almost) easy
- Any object that you plan to serialize must implement the **Serializable interface**

Conditions for serializability

- If an object is to be serialized:
 - The class must be declared as public
 - The class must implement **Serializable**
 - If the object is a sub type of another class, the parent class must have a no-argument constructor
 - All fields of the class must be **serializable**: either primitive types or **serializable** objects

Implementing Serializable

- To “implement” an interface means to define all the methods declared by that interface, but...
- The **Serializable** interface does not define any methods!
 - Question: What possible use is there for an interface that does not declare any methods?
 - Answer: **Serializable is used as flag to tell Java it needs to do extra work with this class**
 - When an object implements **Serializable**, its state is converted to a byte stream to be written to a file so that the byte stream can be converted back into a copy of the object when it is read from the file.

The Serializable Interface

- The **Serializable** interface is a marker interface.
- It has no methods, so you don't need to add additional code in your class except that the class must implement **Serializable**.
- You must also import **java.io** which contains all the streams needed.

The Object Streams

- You need to use the **ObjectOutputStream** class for storing objects and the **ObjectInputStream** class for restoring objects.
- These two classes are built upon several other classes.

A Serializable Version of a Circle Class

```
package ch09.circles;

import java.io.*;

public class SCircle implements Serializable
{
    public int xValue;
    public int yValue;
    public float radius;
    public boolean solid;
}
```

A Program to Save a SCircle Object

```
import java.io.*;
import ch09.circles.*;

public class SaveSCircle
{
    public static void main(String[] args) throws IOException
    {
        SCircle c1 = new SCircle();
        c1.xValue = 5;
        c1.yValue = 3;
        c1.radius = 3.5f;
        c1.solid = true;

        ObjectOutputStream out = new ObjectOutputStream(new
                                                FileOutputStream("objects.dat"));
        out.writeObject(c1);
        out.close();
    }
}
```

Cont....

```
public class FlightRecord implements Serializable
{
    private String flightNumber; // ex. = AA123
    private String origin; // origin airport; ex. = Khi
    private String destination; // destination airport; ex. = Isl
    private int numPassengers; // number of passengers
    private double avgTicketPrice; // average ticket price
```

// Constructor

```
public FlightRecord (String startFlightNumber, String startOrigin, String
startDestination, int startNumPassengers, double startAvgTicketPrice )
{
    flightNumber = startFlightNumber;
    origin = startOrigin;
    destination = startDestination;
    numPassengers = startNumPassengers;
    avgTicketPrice = startAvgTicketPrice;
}
```

Flight Record class

```
public String toString()
{
    return "Flight " + flightNumber
        + ": from " + origin
        + " to " + destination
        + "\n\t" + numPassengers + " passengers";
}
// accessors, mutators, and other methods ...
}
```

```
import java.io;

public class WritingObjects
{
    public static void main( String [] args )
    {
        // instantiate the objects
        FlightRecord fr1 = new FlightRecord( "AA31", "Khi", "Lhr", 200, 13500 );
        FlightRecord fr2 = new FlightRecord( "CO25", "Lhr", "Isl", 225, 11500 );
        FlightRecord fr3 = new FlightRecord( "US57", "Khi", "Isl", 175, 17500 );

        try
        {
            FileOutputStream fos = new FileOutputStream( "objects.dat" );
            ObjectOutputStream oos = new ObjectOutputStream( fos );

            // write the objects to the file
            oos.writeObject( fr1 );
            oos.writeObject( fr2 );
            oos.writeObject( fr3 );

            // release resources associated with the objects file
            oos.close();
        }
    }
}
```

```
catch( FileNotFoundException e )
{
    System.out.println( "Unable to write to objects" );
}
catch( IOException e )
{
    ioe.printStackTrace( );
}
}
```

Saving Hierarchical Objects

- Ensure that each of the objects involved implements the **Serializable** interface

```
import java.io.*;
public class SPoint implements Serializable
{
    public int xValue; // this is for example only
    public int yValue;
}
import java.io.*;
public class SNewCircle implements Serializable
{
    public SPoint location;
    public float radius;
    public boolean soldi;
}
// initialize location's xValue and yValue
```

Reading Objects from a file

- **ObjectInputStream** reads objects from a file. The **readObject()** method reads the next object from the file and returns it.
- Because it returns a generic object, the returned object must be cast to the appropriate class.
- When the end of file is reached, it throws an **EOFException** versus when reading from a text file where a **null String** is returned.

Reading objects from a file

```
ObjectInputStream objectIn = new  
ObjectInputStream( new BufferedInputStream(  
    new FileInputStream(fileName)));
```

```
myObject = (itsType) objectIn.readObject( );  
// some code  
objectIn.close( );
```

```
import java.io.*;

public class GetCircle
{
    public static void main( String [] args )
    {
        SCircle s2 = new SCircle();
        ObjectInputStream in =new ObjectInputStream( new
        BufferedInputStream(new FileInputStream("Objects.dat")));
        try {
            s2 = (SCircle) in.readObject();
        }
        catch (Exception e) { System.out.println (" Error in reading " + e)
        }
        System.out.println( " The value of xvalue is " + s2.xValue;
        System.out.println( " The value of yvalue is " + s2.yValue;
    }
    in.close();
}
```

```
import java.io.ObjectInputStream;  
  
public class ReadingObjects  
{  
    public static void main( String [] args )  
    {  
        try  
        {  
            FileInputStream fis = new FileInputStream( "objects.dat" );  
            ObjectInputStream ois = new ObjectInputStream( fis );  
            try  
            {  
                while ( true )  
                {  
                    // read object, type cast returned object to FlightRecord  
                    FlightRecord temp = ( FlightRecord ) ois.readObject( );  
                    // print the FlightRecord2 object read  
                    System.out.println( temp );  
                }  
            } // end inner try block  
            catch( EOFException eofe )  
            {  
                System.out.println( "End of the file reached" );  
            }  
        }  
    }  
}
```

```
catch( ClassNotFoundException e )
{
    System.out.println( cnfe.getMessage( ) );
}
finally
{
    System.out.println( "Closing file" );
    ois.close( );
}
} // end outer try block
catch( FileNotFoundException e )
{
    System.out.println( "Unable to find objects" );
}
catch( IOException ioe )
{
    ioe.printStackTrace( );
}
}
```

Reading Objects from a file.

- The while loop runs until the end of file is reached and an exception is thrown
- Control goes to the catch block and will always execute in a normal program run.
- The **EOFException** catch block must come before **IOException** as it is subclass of **IOException**. Otherwise, the program will not produce the correct stack trace.

Output from reading objects

----jGRASP exec: java ReadingObjects

Flight AA31: from Khi to Lhr

200 passengers; average ticket price: 13500

Flight CO25: from Lhr to Isl

225 passengers; average ticket price: 11500

Flight US57: from Khi to Isl

175 passengers; average ticket price: 17500

End of the file reached // EOF exception caught
Closing file

Example-Serialization

```
public class Employee implements java.io.Serializable
{
    public String name;
    public String address;
    public transient int SSN;
    public int number;

    public void mailCheck()
    {
        System.out.println("Mailing a check to " + name + " " + address);
    }
}
```

Cont....

```
import java.io.*;
public class SerializeDemo
{
    public static void main(String [] args)
    {
        Employee e = new Employee();
        e.name = "Muhammad Shafan";
        e.address = "DHA, Karachi";
        e.SSN = 11122333;
        e.number = 101;
        try
        {
            FileOutputStream fileOut = new FileOutputStream("/tmp/employee.ser");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(e);
            out.close();
            fileOut.close();
            System.out.println("Serialized data is saved in /tmp/employee.ser");
        }catch(IOException i)
        {
            i.printStackTrace();
        }
    }
}
```

Example-Deserialization

```
import java.io.*;
public class DeserializeDemo
{
    public static void main(String [] args)
    {
        Employee e = null;
        try
        {
            FileInputStream fileIn = new
            FileInputStream("/tmp/employee.ser");
            ObjectInputStream in = new
            ObjectInputStream(fileIn);
            e = (Employee) in.readObject();
            in.close();
            fileIn.close();
        }
    }
}
```

```
        } catch(IOException i)
        {
            i.printStackTrace();
            return;
        }
    catch(ClassNotFoundException c)
    {
        System.out.println("Employee class not found");
        c.printStackTrace();
        return;
    }
    System.out.println("Deserialized Employee...");
    System.out.println("Name: " + e.name);
    System.out.println("Address: " + e.address);
    System.out.println("SSN: " + e.SSN);
    System.out.println("Number: " + e.number);
}
}
```

.NET Serialization

Objectives

- Discuss Object serialization
- Discuss serialization of objects to streams
- Discuss Binary and SOAP formatters
- Discuss XML and JSON formatters

Introduction

- In .NET, serialization is the process of converting an object or a graph of objects into a stream of bytes that can be transmitted or stored.
- The opposite process, deserialization, involves taking a stream of bytes and reconstructing the original object or graph of objects.
- Serialization is commonly used in distributed systems, where objects need to be transmitted across a network, and in persistent storage, where objects need to be stored on disk or in a database. .NET provides several built-in serialization mechanisms that make it easy to serialize and deserialize objects.

Introduction

- Converting an object instance into a format that can either be stored to the disk or transported over the network
- Object can be recreated with its current state at a different location

Object Serialization (1)

- Serializing objects to a stream using binary and SOAP formatters.
- Serializing objects to a stream using XML serialization and saving them as XML files.
- JSON serialization allows you to convert objects to JSON format and vice versa, making it easy to store and transmit data.

Object Serialization (2)

- Objects may be remoted by serializing an object to a stream of bytes
- This stream is then transmitted to another machine that understands the serialization format

How .NET serialization works?

- A .NET formatter class must be used to control the serialization of the object to and from the stream
- The serialized stream carries information about the objects type, including its assembly name, culture & version

Role of formatters

- Determines the serialization format for objects
- All formatters expose an interface called the IFormatter interface

IFormatter Interface

- The 2 formatters that inherit from the IFormatter Interface are –
 - BinaryFormatter
 - XML/SOAPFormatter

Binary Formatter

- To serialize an object, we need –
 1. A formatter which is used to serialize objects
 2. The object that is to be serialized
 3. A stream to hold the serialized object

Serializing an object Binary formatter

- Example 1 -

```
using System;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;

namespace XML2Ex1
{
    class Class1
    {
        static void Main(string[] args)
        {
            Test MyObj = new Test();

            MyObj.Name = "Garfield";
            MyObj.phoneNumber= 5555555;

            Stream MyStream =
File.OpenWrite("C:\\BinSerialization.ex");
            BinaryFormatter formatter = new BinaryFormatter();
            formatter.Serialize(MyStream, MyObj);
            MyStream.Close();
        }
    }
}
```

Serializing an object Binary formatter

- Example 2 -

```
using System;

namespace Serialization
{
    [Serializable]
    public class Test
    {
        public string Name;
        public int phoneNumber;
    }
}
```

Deserializing an object using Binary formatter

- Example 3 -

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;

namespace Serialization
{
    class Class2
    {
        static void Main(string[] args)
        {
            FileStream file = new
FileStream("C:\\BinSerialization.ex", FileMode.Open);

            BinaryFormatter formatter = new BinaryFormatter();

            Test MyObj = formatter.Deserialize(file) as Test;

            Console.WriteLine(MyObj.Name);
            Console.WriteLine(MyObj.phoneNumber);
            Console.ReadLine();
        }
    }
}
```

Deserializing an object using Binary formatter

- Output -



Serializing an object using SOAP formatter

- Example 4 -

```
using System;
using System.Runtime.Serialization.Formatters.Soap;
using System.IO;

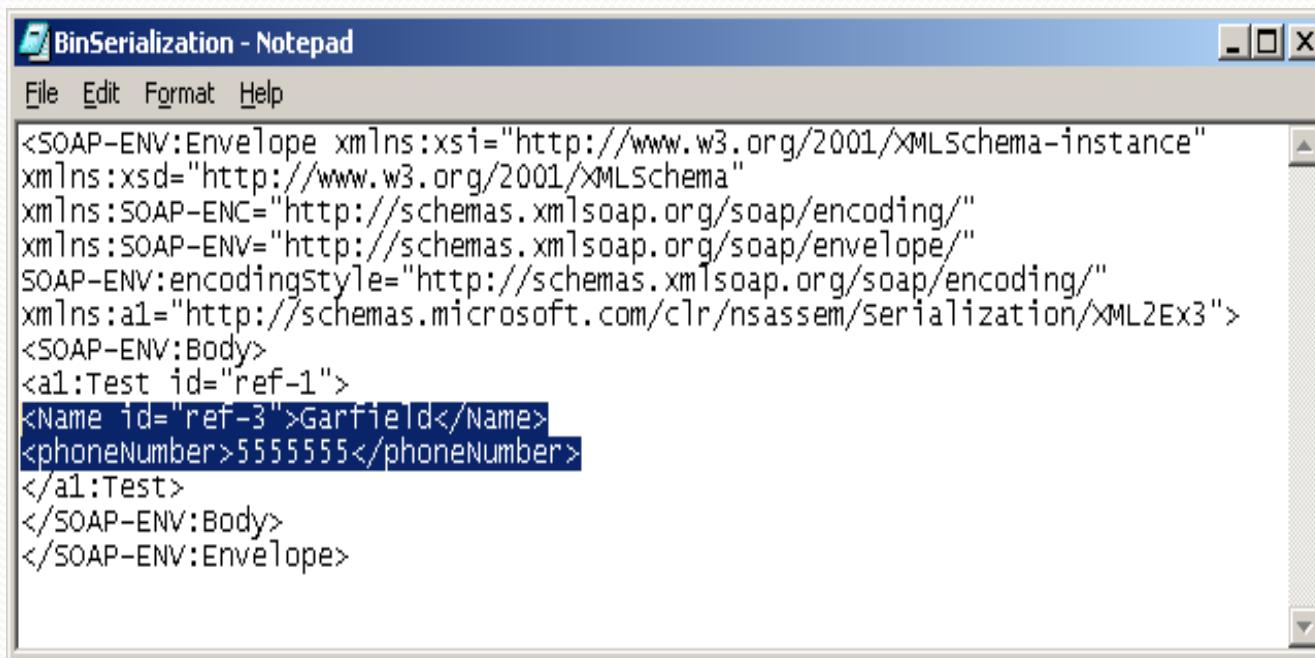
namespace Serialization
{
    class Class1
    {
        static void Main(string[] args)
        {
            Test MyObj = new Test();

            MyObj.Name = "Garfield";
            MyObj.phoneNumber= 5555555;

            Stream MyStream =
File.OpenWrite("C:\\\\BinSerialization.ex");
            SoapFormatter formatter = new SoapFormatter();
            formatter.Serialize(MyStream, MyObj);
            MyStream.Close();
        }
    }
}
```

Serializing an object using SOAP formatter

- Output -



The screenshot shows a Windows Notepad window titled "BinSerialization - Notepad". The window contains XML code representing the serialization of an object. The XML uses namespaces defined in the envelope, such as xsi, xsd, SOAP-ENC, and SOAP-ENV, along with a specific namespace for the test class, a1.

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Serialization/XML2EX3">
    <SOAP-ENV:Body>
        <a1:Test id="ref-1">
            <Name id="ref-3">Garfield</Name>
            <phoneNumber>55555555</phoneNumber>
        </a1:Test>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Deserializing an object using SOAP formatter

- Example 5 -

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;
using System.Runtime.Serialization;

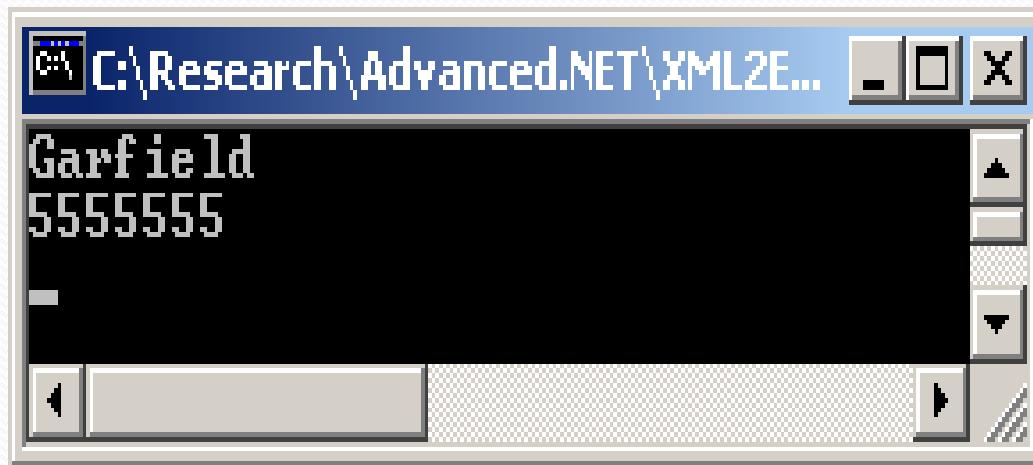
namespace Serialization
{
    class Class2
    {
        static void Main(string[] args)
        {
            FileStream file = new
FileStream("C:\\BinSerialization.ex", FileMode.Open);
            SoapFormatter formatter = new SoapFormatter();

            Test MyObj = (Test) formatter.Deserialize(file);

            Console.WriteLine(MyObj.Name);
            Console.WriteLine(MyObj.phoneNumber);
            Console.ReadLine();
        }
    }
}
```

Deserializing an object using SOAP formatter

- Output -



Selectively serializing the members of an object

- Example 6 -

```
using System;

namespace Serialization
{
    [Serializable]
    public class Test
    {
        public string Name;
        public int phoneNumber;

        [NonSerialized]
        public bool CalledToday;
    }
}
```

XML serialization

- XML serialization allows you to convert objects into XML format and vice versa, making it easy to store and transmit data in a structured way.
- The `XmlSerializer` class is commonly used for this purpose.

XML Serialization

```
using System;  using System.IO;      using System.Xml.Serialization;

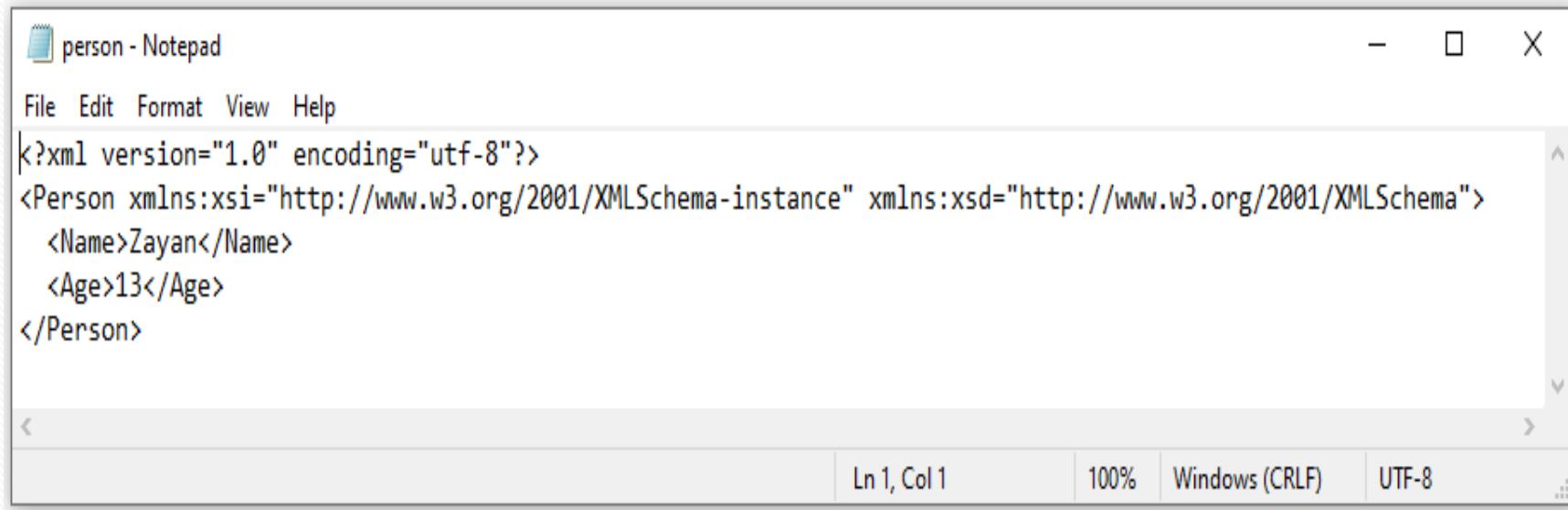
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

class XmlSerializationExample
{
    static void Main()
    {
        Person person = new Person { Name = "Zayan", Age = 13 };

        // Serialize
        XmlSerializer serializer = new XmlSerializer(typeof(Person));
        using (TextWriter writer = new StreamWriter("person.xml"))
        {
            serializer.Serialize(writer, person);
        }

        // Deserialize
        using (TextReader reader = new StreamReader("person.xml"))
        {
            Person serializedPerson = (Person)serializer.Deserialize(reader);
            Console.WriteLine($"{serializedPerson.Name}, {serializedPerson.Age}");
        }  }  }
```

Person.xml



A screenshot of a Windows Notepad window titled "person - Notepad". The window contains XML code defining a person with a name and age. The XML is well-formed with proper indentation and closing tags.

```
<?xml version="1.0" encoding="utf-8"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <Name>Zayan</Name>
    <Age>13</Age>
</Person>
```

The Notepad window includes standard menu options (File, Edit, Format, View, Help) and status bar information (Ln 1, Col 1, 100%, Windows (CRLF), UTF-8).

JSON Serialization

- JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate.
- In C#, JSON serialization and deserialization allow you to convert objects to and from JSON format, making data exchange straightforward.
- The two most common libraries for JSON serialization in C# are
 - 1) System.Text.Json (introduced in .NET Core 3.0)
 - 2) Newtonsoft.Json (third-party library).

JSON Serialization

By using NuGet manager Install-Package System.Memory

```
using System;
using System.IO;
using System.Text.Json;

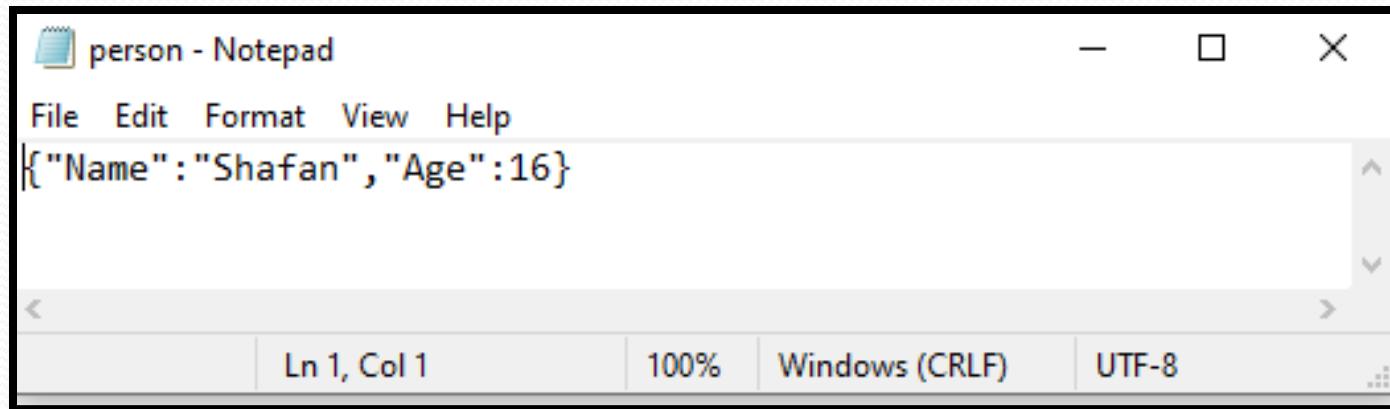
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

class JsonSerializerExample
{
    static void Main()
    {
        Person person = new Person { Name = "Shafan", Age = 16 };

        // Serialize
        string jsonString = JsonSerializer.Serialize(person);
        File.WriteAllText("person.json", jsonString);

        // Deserialize
        jsonString = File.ReadAllText("person.json");
        Person serializedPerson = JsonSerializer.Deserialize<Person>(jsonString);
        Console.WriteLine($"{serializedPerson.Name}, {serializedPerson.Age}");
    }
}
```

Person.json



Exception Handling

Session 13

Introduction

- Rarely does a program runs successfully at its very first attempt.
- It is common to make mistakes while developing as well as typing a program.
- Such mistakes are categorised as:
 1. syntax errors - compilation errors.
 2. semantic errors- leads to programs producing unexpected outputs.
 3. runtime errors - most often lead to abnormal termination of programs or even cause the system to crash.

Introduction to Exception

- Is a special type of error
- It occurs at runtime in a code sequence
- Abnormal conditions that occur while executing the program cause exceptions
- If these conditions are not dealt with, then the execution can be terminated abruptly

Purpose of Exception handling

- Minimize the chances of a system crash, or abrupt program termination
- For example,

In an I/O operation in a file. If the data type conversion is not properly done, an exception occurs, and the program aborts, without closing the file. This may damage the file, and the resources allocated to the file may not return to the system

Handling Exceptions

- When an exception occurs, an object that represents that exception is created
- This object is then passed to the method where the exception has occurred
- The object contains detailed information about the exception. This information can be retrieved and processed
- The 'Throwable' class that Java provides is the superclass of the Exception class, which is, in turn, the superclass of individual exceptions

Throwable

- In Java, `Throwable` is the superclass of all errors and exceptions.
- `Throwable` class has two main subclasses:
 1. **Error**: Represents serious issues that a reasonable application should not catch. For example, `OutOfMemoryError` and `StackOverflowError`.
 2. **Exception**: Represents conditions that a reasonable application might want to catch. This includes **checked** exceptions (e.g., `IOException`, `SQLException`) and **unchecked** exceptions (e.g., `NullPointerException`, `ArrayIndexOutOfBoundsException`).

Checked vs. Unchecked Exceptions

- **Checked Exceptions:** Subclasses of `Exception` that are checked at compile time. The programmer is required to handle these exceptions using try-catch blocks or declare them in the method signature.
- **Unchecked Exceptions:** Subclasses of `RuntimeException` that are not checked at compile time. These can occur due to programming errors, such as accessing an array out of bounds.

Exception handling Model

- Is also known as the ‘catch and throw’ model
- When an error occurs, an ‘exception’ is thrown, and caught in a block
- Keywords to handle exceptions
 - try
 - catch
 - throw
 - throws
 - finally

Structure of the exception handling model

- **Syntax**

try { }

catch(Exception e1) { }

catch(Exception e2) { }

catch(Exception eN) { }

finally { }

Advantages of the ‘Catch and Throw’ Model

- The programmer has to deal with an error condition only where necessary. It need not be dealt with at every level
- An error message can be provided in the exception-handler

‘try’ and ‘catch’ Blocks

- Is used to implement the ‘catch and throw’ model of exception handling
- A ‘try’ block consists of a set of executable statements
- A method, which may throw an exception, can also be included in the ‘try’ block
- One or more ‘catch’ blocks can follow a ‘try’ block
- These ‘catch’ blocks catch exceptions thrown in the ‘try’ block

try' and 'catch' Blocks (Contd...)

- To catch any type of exception, specify the exception type as '**Exception**'

catch(Exception e)

- When the type of exception being thrown is not known, the class '**Exception**' can be used to catch that exception
- The error passes through the 'try catch' block, until it encounters a 'catch' that matches it, or the program terminates

Multiple Catch Blocks

- Multiple ‘catch()’ blocks process various exception types separately
- Example

```
try
{ doFileProcessing();
  displayResults(); }
catch(LookupException e)
{ handleLookupException(e); }
catch(Exception e)
{ System.err.println("Error:"+e.printStackTrace()); }
```

Multiple Catch Blocks (Contd...)

- When nested ‘try’ blocks are used, the inner ‘try’ block is executed first
- Any exception thrown in the inner ‘try’ block is caught in the following ‘catch’ blocks
- If a matching ‘catch’ block is not found, then ‘catch’ blocks of the outer ‘try’ blocks are inspected
- Otherwise, the Java Runtime Environment handles the exception

Without Error Handling – Example 1

```
class NoErrorHandler{  
    public static void main(String[] args){  
        int a, b;  
        a = 7;  
        b = 0;  
  
        System.out.println("Result is " + a/b);  
        System.out.println("Program reached this line");  
    }  
}
```

Program does not reach here

```
Exception in thread "main" java.lang.ArithmetricException: / byzero  
at javaapplication2.JavaApplication2.main(JavaApplication2.java:63)  
Java Result: 1
```

Traditional way of Error Handling - Example 2

```
class WithErrorHandler{  
    public static void main(String[] args){  
        int a, b;  
        a = 7;  b = 0;  
        if (b != 0){  
            System.out.println("Result is " + a/b);  
        }  
        else{  
            System.out.println(" B is zero);  
        }  
        System.out.println("Program is complete");  
    }  
}
```

Program reaches here



Exceptions

- An exception is a condition that is caused by a runtime error in the program.
- Provide a mechanism to signal errors directly without using flags.
- Allow errors to be handled in one central part of the code without cluttering code.

Exceptions and their Handling

- When the JVM encounters an error such as divide by zero, it creates an exception object and throws it – as a notification that an error has occurred.
- If the exception object is not caught and handled properly, the interpreter will display an error and terminate the program.
- If we want the program to continue with execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then take appropriate corrective actions. This task is known as *exception handling*.

Common Java Exceptions

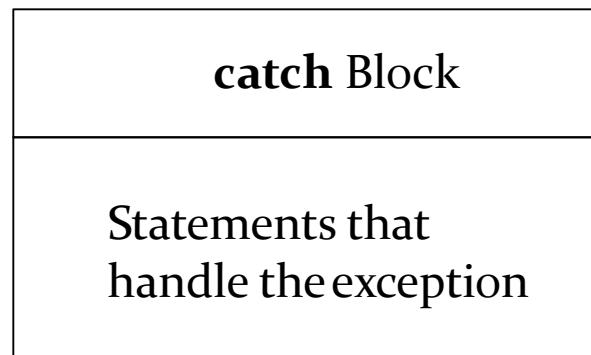
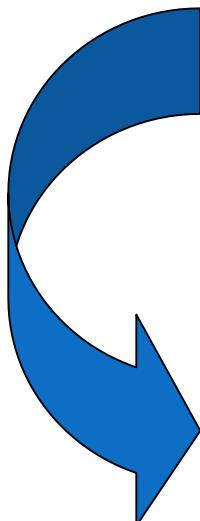
- `ArithmaticException`
- `ArrayIndexOutOfBoundsException`
- `FileNotFoundException`
- `IOException` – general I/O failure
- `NullPointerException` – referencing a null object
- `OutOfMemoryException`
- `SecurityException` – when applet tries to perform an action not allowed by the browser's security setting.
- `StringIndexOutOfBoundsException`

Exceptions in Java

- A method can signal an error condition by throwing an exception – *throws*
- The calling method can transfer control to an exception handler by catching an exception - *try, catch*
- Clean up can be done by - *finally*

Exception Handling Mechanism

Throws
exception
Object



Syntax of Exception Handling Code

```
...
...
try {
    // statements
}
catch( Exception-Type e)
{
    // statements to process exception
}
...
...
```

With Exception Handling - Example 3

```
class WithExceptionHandling{
    public static void main(String[] args){
        int a,b; float r;
        a = 7; b = 0;
        try{
            r = a/b;
            System.out.println("Result is " + r);
        }
        catch(ArithmaticException e){
            System.out.println(" B is zero");
        }
        System.out.println("Program reached this line");
    }
}
```

Program will not Reaches here

Program Reaches here

```
graph TD
    A[Program will not Reaches here] --> B[try{]
    C[Program Reaches here] --> D[System.out.println("Program reached this line")]
    style A fill:#90EE90,stroke:#000,stroke-width:1px
    style C fill:#90EE90,stroke:#000,stroke-width:1px
```

Finding a Sum of Values Passed as Command Line Arguments

```
// ComLineSum.java: adding command line parameters
class ComLineSum
{
    public static void main(String args[])
    {
        int InvalidCount = 0;
        int number, sum = 0;

        for( int i = 0; i < args.length; i++)
        {
            try {
                number = Integer.parseInt(args[i]);
            }
            catch(NumberFormatException e)
            {
                InvalidCount++;
                System.out.println("Invalid Number: "+args[i]);
                continue;//skip the remaining part of loop
            }
            sum += number;
        }
        System.out.println("Number of Invalid Arguments = "+InvalidCount);
        System.out.println("Number of Valid Arguments = "+(args.length-InvalidCount));
        System.out.println("Sum of Valid Arguments = "+sum);
    }
}
```

Sample Runs

C:>java ComLineSum 1 2

Number of Invalid Arguments = 0

Number of Valid Arguments = 2

Sum of Valid Arguments = 3

C:>java ComLineSum 1 2 abc

Invalid Number: abc

Number of Invalid Arguments = 1

Number of Valid Arguments = 2

Sum of Valid Arguments = 3

Multiple Catch Statements

- If a try block is likely to raise more than one type of exceptions, then multiple catch blocks can be defined as follows:

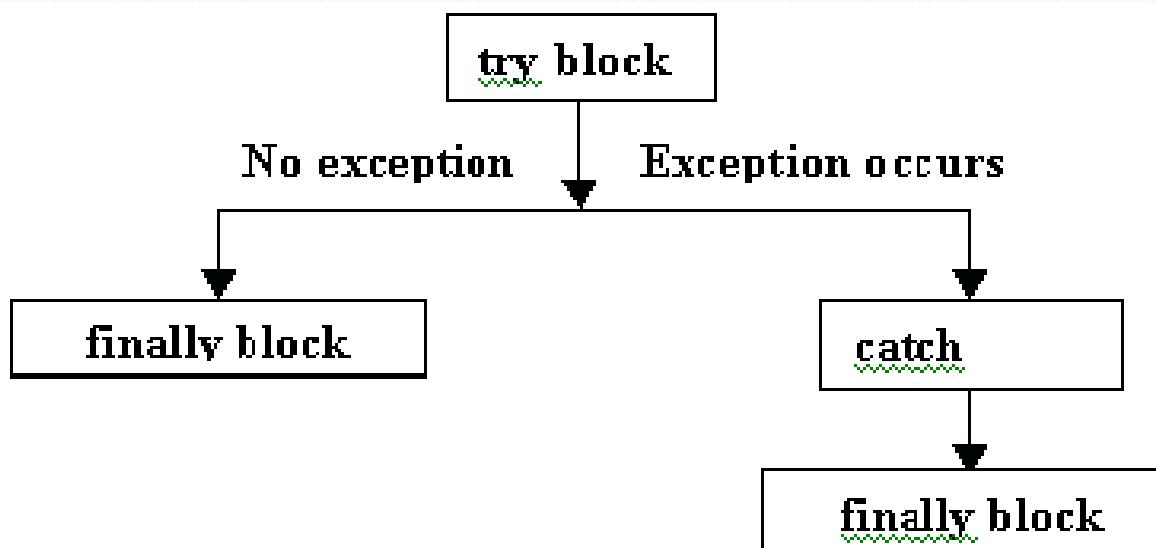
```
...
...
try {
    // statements
}
catch( Exception-Type1 e)
{
    // statements to process exception 1
}
...
...
catch( Exception-TypeN e)
{
    // statements to process exception N
}
...
```

‘finally’ Block

- Takes care of all the cleanup work when an exception occurs
- Can be used in conjunction with a ‘try’ block
- Contains statements that either return resources to the system, or print messages
 - Closing a file
 - Closing a result set (used in Database programming)
 - Closing the connection established with the database

'finally' Block (Contd...)

- Is optional
- Is placed after the last 'catch' block
- The 'finally' block is guaranteed to run, whether or not an exception occurs



Flow of the 'try', 'catch' and 'finally' blocks

finally block

- When a finally is defined, it is executed regardless of whether or not an exception is thrown. Therefore, it is also used to perform certain house keeping operations such as closing files and releasing system resources.

```
...
try {
    // statements
}
catch( Exception-Type e )
{
    // statements to process exception 1
}
...
...
finally {
    ....
}
```

User-defined Exceptions with ‘throw’ and ‘throws’ statements

- Exceptions are thrown with the help of the ‘throw’ keyword
- The ‘throw’ keyword indicates that an exception has occurred
- The operand of throw is an object of a class, which is derived from the class ‘Throwable’
- Example of the ‘throw’ statement

```
try{  
    if (flag < 0)  
    {  
        throw new MyException( ) ; // user-defined  
    }  
}
```

User-defined Exceptions with ‘throw’ and ‘throws’ statements (Contd...)

- A single method may throw more than one exception
- Example of the ‘throws’ keyword to handle multiple exceptions

```
public class Example {  
    public void exceptionExample( ) throws ExException,  
        LookupException {  
        try  
        { // statements }  
        catch(ExException exmp)  
        { .... }  
        catch(LookupException lkpx)  
        { .... } } }
```

User-defined Exceptions with ‘throw’ and ‘throws’ statements (Contd...)

- The ‘Exception’ class implements the ‘Throwable’ interface, and provides some useful features for dealing with exceptions
- Advantage of subclassing the Exception class is that the new exception type can be caught separately from other Throwable types

With Exception Handling - Example 4

```
class WithExceptionCatchThrow{  
    public static void main(String[] args){  
        int a,b; float r; a = 7; b = 0;  
        try{  
            r = a/b;  
            System.out.println("Result is " + r);  
        }  
        catch(ArithmaticException e){  
            System.out.println(" B is zero);  
            throw e;  
        }  
        System.out.println("Program is complete");  
    }  
}
```

Program Does Not
reach here when
exception occurs



With Exception Handling - Example 5

```
class WithExceptionCatchThrowFinally{  
    public static void main(String[] args){  
        int a,b; float r; a = 7; b = 0;  
        try{  
            r = a/b;  
            System.out.println("Result is " + r);  
        }  
        catch(ArithmaticException e){  
            System.out.println(" B is zero);  
            throw e;  
        }  
        finally{  
            System.out.println("Program is complete");  
        }  
    }  
}
```

Program reaches here



User-Defined Exceptions

- Problem Statement :
 - Consider the example of the Circleclass
 - Circle class had the following constructor

```
public Circle(double centreX, double centreY, double radius){  
    x = centreX; y = centreY; r = radius;  
}
```

- How would we ensure that the radius is not zero or negative?

Defining your own exceptions

```
import java.lang.Exception;
class InvalidRadiusException extends Exception {

    private double r;

    public InvalidRadiusException(double radius){
        r = radius;
    }
    public void printError(){
        System.out.println("Radius [" + r + "] is not valid");
    }
}
```

Throwing the exception

```
class Circle {  
    double x, y, r;  
  
    public Circle (double centreX, double centreY, double  
radius ) throws InvalidRadiusException {  
        if (r <= 0 ) {  
            throw new InvalidRadiusException(radius);  
        }  
        else {  
            x = centreX ; y = centreY;  r = radius;  
        }  
    }  
}
```

Catching the exception

```
class CircleTest {  
    public static void main(String[] args){  
        try{  
            Circle c1 = new Circle(10, 10, -1);  
            System.out.println("Circle created");  
        }  
        catch(InvalidRadiusException e)  
        {  
            e.printError();  
        }  
    }  
}
```

User-Defined Exceptions in standard format

```
class MyException extends Exception  
{  
    MyException(String message)  
    {  
        super(message); // pass to superclass if parameter is not handled by used defined exception  
    }  
}  
  
class TestMyException {  
    ...  
    try {  
        ..  
        throw new MyException("This is error message");  
    }  
    catch(MyException e)  
    {  
        System.out.println("Message is: "+e.getMessage());  
    }  
}
```



Get Message is a method defined in a standard Exception class.

Summary

- A good programs does not produce unexpected results.
- It is always a good practice to check for potential problem spots in programs and guard against program failures.
- Exceptions are mainly used to deal with runtime errors.
- Exceptions also aid in debugging programs.
- Exception handling mechanisms can effectively used to locate the type and place of errors.

Summary

- *Try* block, code that could have exceptions / errors
- *Catch* block(s), specify code to handle various types of exceptions. First block to have appropriate type of exception is invoked.
- If no ‘local’ catch found, exception propagates up the method call stack, all the way to `main()`
- Any execution of try, normal completion, or catch then transfers control on to *finally* block

Properties and Events

Session 14

Session Objectives

- Explain Properties
- Implement Indexers
- Implement Delegates
- Define and raise Events

Properties [1]

- C# provides the facility to protect a field (private attributes) in a class by reading and writing to it through a feature called **Properties**.
- With Properties, we just have to define the Set/Get methods and continue to use the data members as fields.
- The runtime takes care of identifying and calling the appropriate Set/Get function.

Properties [2]

```
using System;

public class Employee
{
    public string eName;           //Field

    private string internal_eDept; //Field
    public string eDept           //Property
    {
        get
        {
            return internal_eDept;
        }

        set
        {
            internal_eDept = value;
        }
    }
}
```

Continued..

Properties [3]

Continued..

```
class Test
{
    static void Main()
    {
        Employee one = new Employee();

        one.eName = "Scooby";
        one.eDept = "Design";

        Console.WriteLine("The Name of the Employee is {0} and the Department is {1}",
one.eName, one.eDept);
    }
}
```

Properties [4]

```
private string internal_eDept; //field  
public string eDept //Property
```

```
{  
    get
```

```
    {  
        return internal_eDept;  
    }
```

```
    set
```

```
    {  
        internal_eDept = value;  
    }
```

```
one.eName = "scooby";
```

```
one.eDept = "Design";
```

```
Console.WriteLine("The Name of the Employee is {0} and the  
Department is {1}", one.eName, one.eDept);  
}
```

This line of code will automatically
implicitly call the set method

This line of code will automatically
implicitly call the get method

Types of Properties

Read/Write

Read-Only

Write-Only

Read/Write Property

- Properties of this type provide both read and write access to the data members.

```
private string internal_eDept; //Field
public string eDept          //Property
{
    get
    {
        return internal_eDept;
    }

    set
    {
        internal_eDept = value;
    }
}
```

Read-Only Property

- Properties with only a get accessor are called Read-Only properties.

```
public class Employee
{
    private int sal = 40000; //Field
    public int Sal          //Property
    {
        get
        {
            return sal;
        }
    }
}

class Test
{
    static void Main()
    {
        Employee one = new Employee();

        System.Console.WriteLine("The salary is {0}", one.Sal);
        one.Sal = 55000;
    }
}
```

Write-Only Property

- Properties with only a **set** accessor are called Write-Only properties.
- We can only write values to it but can never retrieve the value.

Properties and Fields

- Properties are logical fields
 - With set and get we can perform validations before assigning the value similarly we can return a calculated value
- Properties are an extension of fields
 - we can declare properties with any access modifier. They can be even static
- Unlike fields, properties do not correspond directly to a storage location
 - Properties refer to the get/set methods while the fields refer directly to the storage location

Properties and Methods

- Properties do not use parenthesis.
- Properties cannot be void.

Properties with Logic

```
public class Student
{
    private int age;

    public int Age
    {
        get { return age; }
        set
        {
            if (value < 0)
                throw new ArgumentOutOfRangeException("Age cannot be negative.");
            age = value;
        }
    }
}
```

Auto-Implemented Properties

- C# provides a shorthand for properties called auto-implemented properties, where you don't need to explicitly define a backing field.

```
public class Student
{
    public string Name { get; set; } // Auto-implemented property
    public int Age { get; set; }
}
```

```
public class Person
{
    public string Name { get; set; } // Read/write property
    public int Age { get; private set; } // Read-only property
}
```

Indexers

- An indexer allows an object to be indexed in the same way as an array. As properties provide us with the field-like access to the data of an object, indexers enable array-like access to our class members

Indexers [1]

```
class IndexerTest
{
    public int[] list=new int[10];

    public int this[int index]
    {
        get
        {
            return list[index];
        }

        set
        {
            list[index] = value;
        }
    }
}
```

Continued..

Indexers [2]

Continued..

```
class Test
{
    static void Main()
    {
        IndexerTest IndexMe = new IndexerTest();

        IndexMe.list[1]=5;

        IndexMe[2]=10;

        System.Console.WriteLine("IndexMe[1] is {0}\nIndexMe[2] is {1}", IndexMe[1],
IndexMe[2]);
    }
}
```

➤ Output:

```
IndexMe[1] is 5
IndexMe[2] is 10
```

Steps

- Mention the access modifier, which decides the visibility of the indexer
- State the return type of the indexer (what the get accessor returns)
- Specify the “this” keyword (this is the name of the indexer, it should always be this)
- Insert the open square brackets
- Specify the data type of the index (indexers unlike arrays, can also be indexed on string, or any other data type).
- State the variable name for the index, followed by the close square brackets
- Insert the open curly braces. The get and set accessors are specified here, just as in the case of properties. Finally, the close braces must be inserted

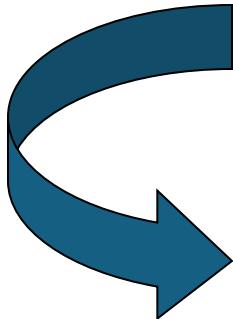
Defining an Indexer [1]

- Syntax:

Access Modifier return type this [data type variable]

Defining an Indexer [2]

➤ Rules:



- At least one Indexer parameter must be specified.
- The parameters should be assigned values.

Indexers vs. Arrays [1]

- Indexers do not point to memory locations.
- Indexers can have non-integer subscripts(indexes).
- Indexers can be overloaded.

Indexers vs. Arrays [2]

```
public int this[int index]
{
    get
    {
        return list[index];
    }

    set
    {
        list[index] = value;
    }
}

IndexMe[2] = 10;

System.Console.WriteLine("IndexMe[1] is {0}\nIndexMe[2] is {1}",
    IndexMe[1], IndexMe[2]);
}
```

Implicitly calls the **get** accessor

Implicitly calls the **set** accessor

Indexers vs. Arrays [3]

```
using System.Collections;

class StrIndex
{
    public Hashtable PhoneList = new Hashtable();

    public string this[string Name]
    {
        get
        {
            return (string) PhoneList[Name];
        }
        set
        {
            PhoneList[Name] = value;
        }
    }
}
```

Continued..

Indexers vs. Arrays [4]

Continued..

```
class Test
{
    static void Main()
    {
        StrIndex PhoneIndex = new StrIndex();

        PhoneIndex["Scooby"] = "7568375";
        PhoneIndex["Scottie"] = "8461373";

        System.Console.WriteLine("Phone number of Scottie is {0}", PhoneIndex["Scottie"]);
    }
}
```

Indexers vs. Arrays [5]

```
using System.Collections;

class IndexerTest
{
    public int[] list=new int[10];

    public int this[int index]
    {
        get
        {
            return list[index];
        }

        set
        {
            list[index] = value;
        }
    }
}
```

Continued..

Indexers vs. Arrays [6]

Continued..

```
public Hashtable PhoneList = new Hashtable();

public string this[string Name]
{
    get
    {
        return (string) PhoneList[Name];
    }
    set
    {
        PhoneList[Name] = value;
    }
}
```

Continued..

Indexers vs. Arrays [7]

Continued..

```
class Test
{
    static void Main()
    {
        IndexerTest IndexMe = new IndexerTest();

        IndexMe.list[1]=5;
        IndexMe[2] = 10;

        IndexMe["Scooby"] = "5555111";
        IndexMe["Scottie"] = "8461373";
    }
}
```

Multiple Parameters in Indexers

- More than one indexer parameter can be specified.

```
...
class TestMultip
{
    public int this[int firstP, int secondP]
    {
        //Get and Set Accessors appear here
    }
}
```

```
class Test
{
void static Main()
{
    TestMultip m = new TestMultip();
    int I = m[1,1];
}
}
```

Indexers with multiple indexer parameters are accessed like a Multi-Dimensional Array

Delegates

- A delegate contains a reference to a method.
- A delegate connects a name with the specification of a method.
- An implementation of some method can be then attached to this name.
- A component can call the method by using this name.

Defining a Delegate

➤ Syntax:

```
Access Modifier delegate void DelegateName()
```

Instantiating Delegates [1]

- Instantiating a delegate means pointing it to some method.
- To instantiate a delegate, we must call the constructor of the delegate and pass the method (along with its object name), to be associated with the delegate, as its parameter.

Instantiating Delegates [2]

```
class TestDelegates
{
    public delegate int DeleMe(int a, int b);

    class Maths
    {
        public int AddMe(int a, int b)
        {
            return a+b;
        }

        public int SubMe(int a, int b)
        {
            return a-b;
        }
    }
}
```

Continued..

Instantiating Delegates [3]

Continued..

```
class Test
{
    static void Main()
    {
        DeleMe DelegateObj;
        Maths m = new Maths();

        DelegateObj = new DeleMe(m.AddMe);
    }
}
```

Using Delegates [1]

- Using a delegate means instantiating a method using delegates.

```
class TestDelegates
{
    public delegate int DeleMe(int a, int b);

    class Maths
    {
        public int AddMe(int a, int b)
        {
            return a+b;
        }

        public int SubMe(int a, int b)
        {
            return a-b;
        }
    }
}
```

Continued..

Using Delegates [1]

Continued..

```
class Test
{
    static void Main()
    {
        DeleMe DelegateObj;
        Maths m = new Maths();

        DelegateObj = new DeleMe(m.AddMe);

        int t=DelegateObj(5,3);
        System.Console.WriteLine("The Result is : {0}",t);
    }
}
```

➤ Output:

The Result is : 8

Events [1]

- Events in C# allow an object to notify other objects about the event, or that a change has occurred.
- The object that notifies others about the event is known as the Publisher.
- An object that registers to an event is known as the Subscriber.

Events [2]

➤ Steps:



Defining Events

➤ The Publisher,

- defines a delegate
- defines an event based on the delegate

```
public delegate void delegateMe();  
private event delegateMe eventMe;
```

Subscribing to an Event

- Subscribing an object to an event depends on whether the event exists.
- If the event exists, then the subscribing object simply adds a delegate that calls a method when the event is raised.

```
eventMe += new delegateMe(objA.Method);
```

```
eventMe += new delegateMe(objB.Method);
```

Notifying Objects

- To notify all the objects that have subscribed to an event, we just need to raise the event:

```
if(condition)
{
    eventMe();
}
```

```
using System;
class ClassA
{
    public void DispMethod()
    {
        Console.WriteLine("Class A has been notified of NotifyEveryone Event!");
    }
}

class ClassB
{
    public void DispMethod()
    {
        Console.WriteLine("Class B has been notified of NotifyEveryone Event!");
    }
}

class Test
{
    public static void Main()
    {
        Dele dA = new Dele();
        ClassA objA = new ClassA();
        ClassB objB = new ClassB();

        dA.NotifyEveryone += new Dele.MeDelegate(objA.DispMethod);
        dA.NotifyEveryone += new Dele.MeDelegate(objB.DispMethod);

        dA.Notify();
    }
}

class Dele
{
    public delegate void MeDelegate();
    public event MeDelegate NotifyEveryone;

    public void Notify()
    {
        if(NotifyEveryone != null)
        {
            Console.WriteLine("Raise Event : ");
            NotifyEveryone();
        }
    }
}
```

Output....

```
E:\>cd faisal  
E:\faisal>event  
Raise Event :  
Class A has been notified of NotifyEveryone Event!  
Class B has been notified of NotifyEveryone Event!  
E:\faisal>
```