Lecture-1

Rizwan Rashid

Outline

- Evolution of Object Oriented Programming (OOP)
- Difference between Object Oriented Approach & Modular/Structural Approach
- Object-Oriented Concepts and Principles

Evolution of Object Oriented Programming (OOP)

- Object-oriented programming, originating from the work on SIMULA by Ole-Johan Dahl and Kristen Nygaard
- The language introduced all elements of an object-oriented language such as
  - Encapsulation, inheritance, late binding, and dynamic object creation.

Evolution of Object Oriented Programming (OOP)

- SIMULA I in (1962-65) and SIMULA 67 in 1967
- Alan Kay integrated philosophy of SIMULA 67 in Smalltalk in 1970
- In 1980, Bjarne Stroustrup introduce the philosophy of SIMULA to C systems developers community.
  - C++ object-oriented language
- In 1990, James Gosling developed JAVA

Structured Languages

- Program consists of list of instruction
  - E.g. Get some input, add these numbers, divide by six, display that output
- NO organizing principle: complexity increases with increase in program size
- Example
  - C, Pascal, FORTRON

Structured Languages (Cont'd)

- Division into Functions
  - Large program divided into small functions
- A function has a clearly defined purpose and a clearly defined interface to the other functions
- Module: is grouping of functions togeather into larger entity called module

Structured Programming

- Dividing a program into functions and modules to improve clarity, quality of a program
- Two related problems with structured programs
  - Unrestricted Access
  - Unrelated data and procedure

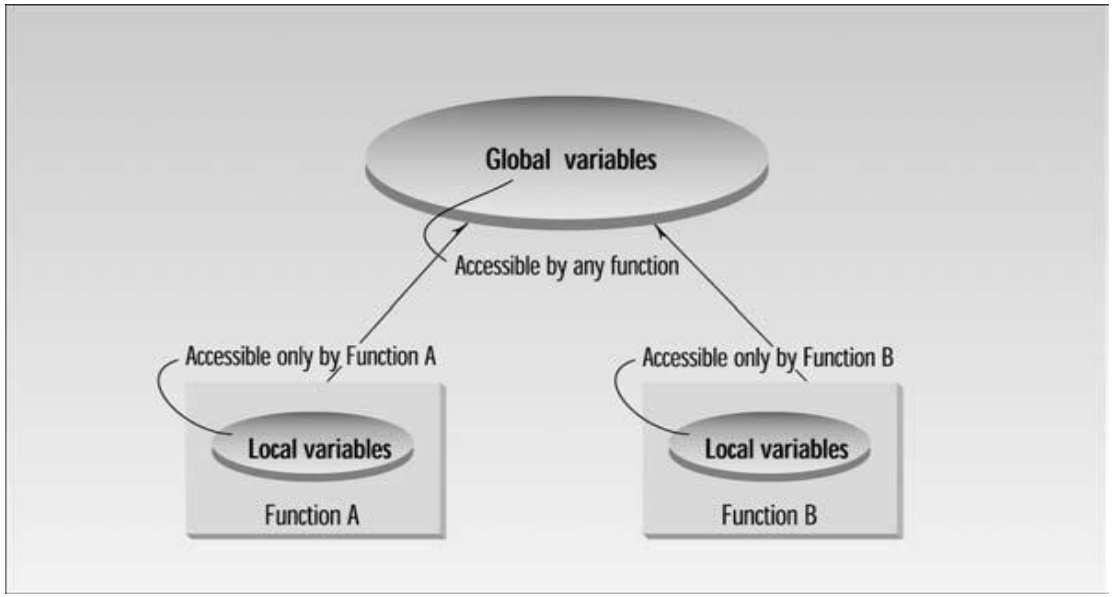
Unrestricted Access

- In structured language, two kinds of data
- Local data
  - Local access to function which is safe from modification by other functions

```
int AddNumber()
{
        int num1;
        int num2;
        int num3;
        num3 = num1 + num2;
        return num3;
}
```

Unrestricted Access (Cont'd)

- Global data
  - When one or more function must access same data
  - Can be accessed by any function in a program
  - Difficult to modify the program



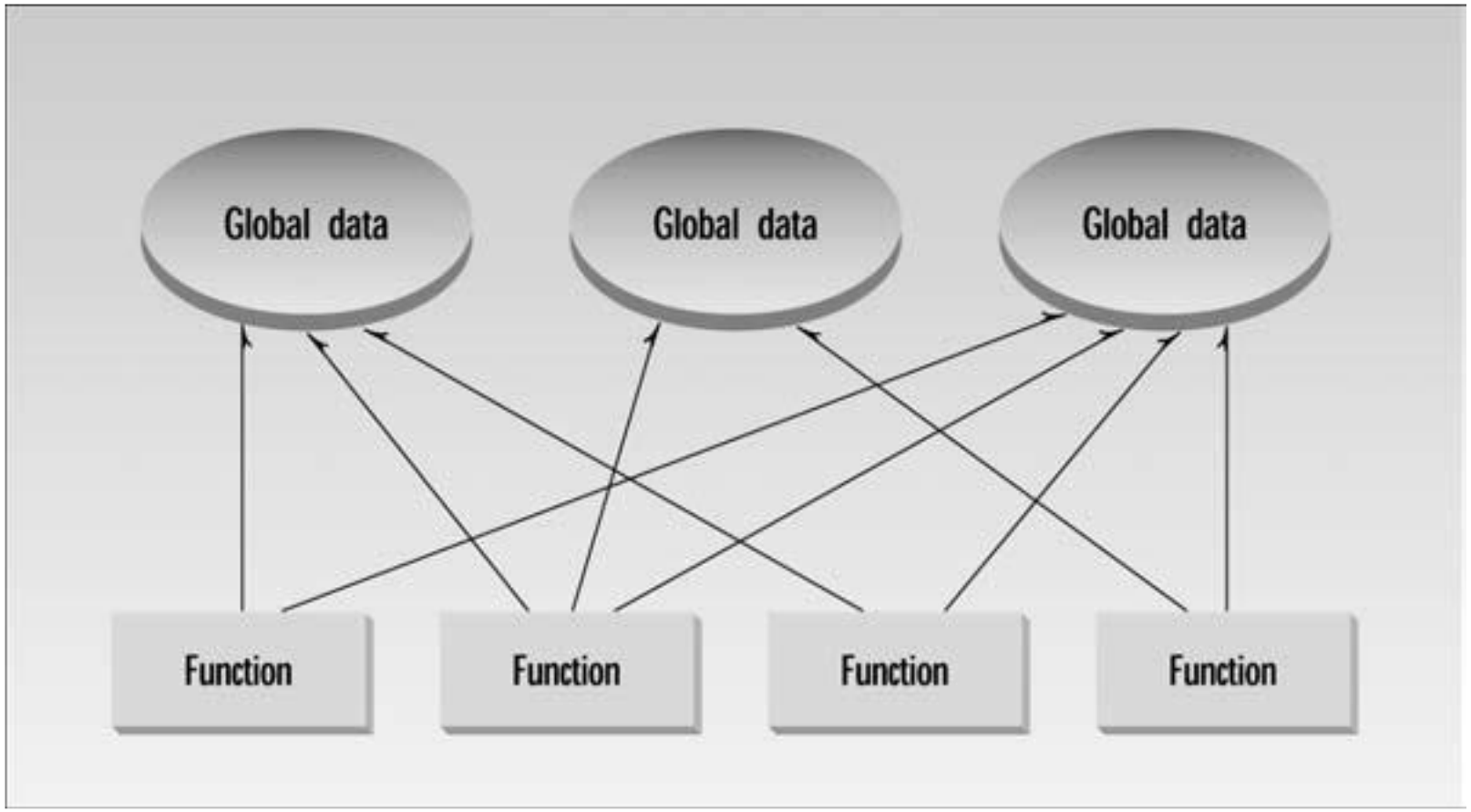
Example

```
int num1=2;
int num2=5;
int  num3;
Void AddNumber()
{
        num3 = num1 +  num2;
}
Void SubNumber()
{
        num3 = num2 – num1;
}
```

Unrelated data and procedure

- In a large program, there are many functions and many global data items
- The problem with the structured/modular paradigm is that this leads to larger number of potential connections between functions and data
  - It makes a program's structure difficult to conceptualize
  - It makes the program difficult to modify

Cont’d

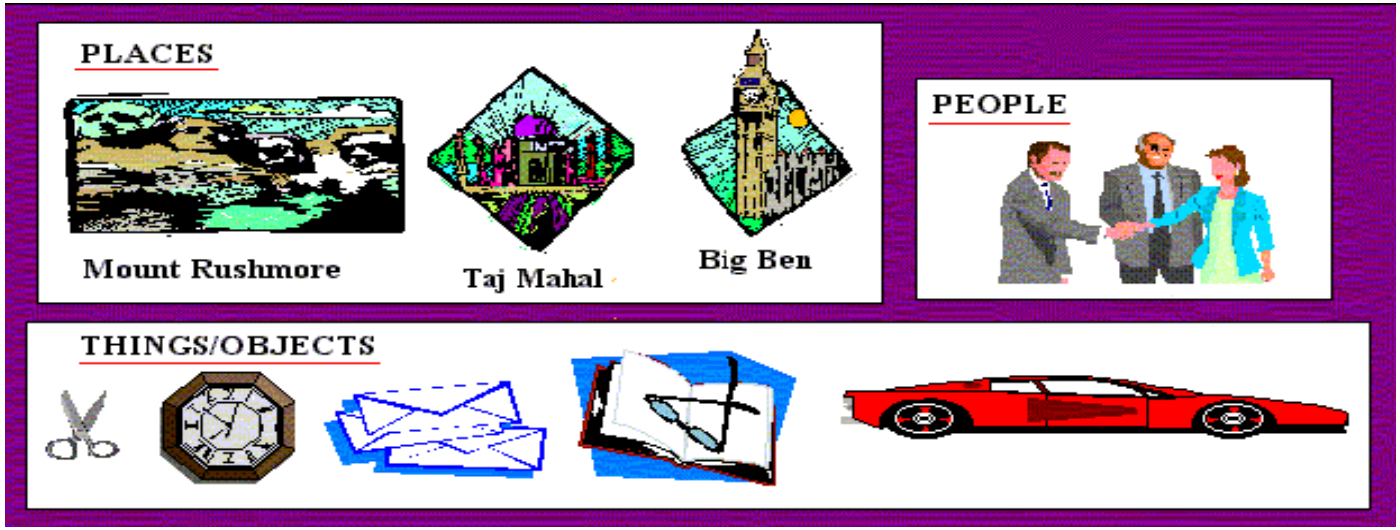

Global data
Global data
Global data
Function
Function
Function
Function

Real – World Modeling

- Physical World entities like car, people
- Focus on real objects to be mapped in computer program
- Objects has
  - Attributes – characteristics
    - For people; eye color, job title
    - For cars; horsepower, number of doors
  - Behavior
    - Is like function: call a function to do something

Object

- ***An Object*** is a computer representation of some real-world thing (i.e. person, place).
- Objects can have both *attributes* and *behaviors*



Object (Cont'd)

- When an object is mapped into software representation, it consists of two parts:
- **PRIVATE data structure**
  characteristics of private data structure are referred to as *ATTRIBUTES e.g. FirstName, LastName, Color etc.*
- **PROCESSES** that may correctly change the data structure
- Processes are referred to as *OPERATIONS* or *METHODS e.g. SetFirstName(), SetColor()*

Objects Examples

Physical objects
- Automobiles in a traffic-flow simulation
- Electrical components in a circuit-design program
- Countries in an economics model
- Aircraft in an air traffic control system

Components in computer games
- Cars in an auto race
- Positions in a board game (chess, checkers)
- Animals in an ecological simulation
- Opponents and friends in adventure games

Elements of the computer-user environment
- Windows
- Menus
- Graphics objects (lines, rectangles, circles)
- The mouse, keyboard, disk drives, printer

Collections of data
- An inventory
- A personnel file
- A dictionary
- A table of the latitudes and longitudes of world cities

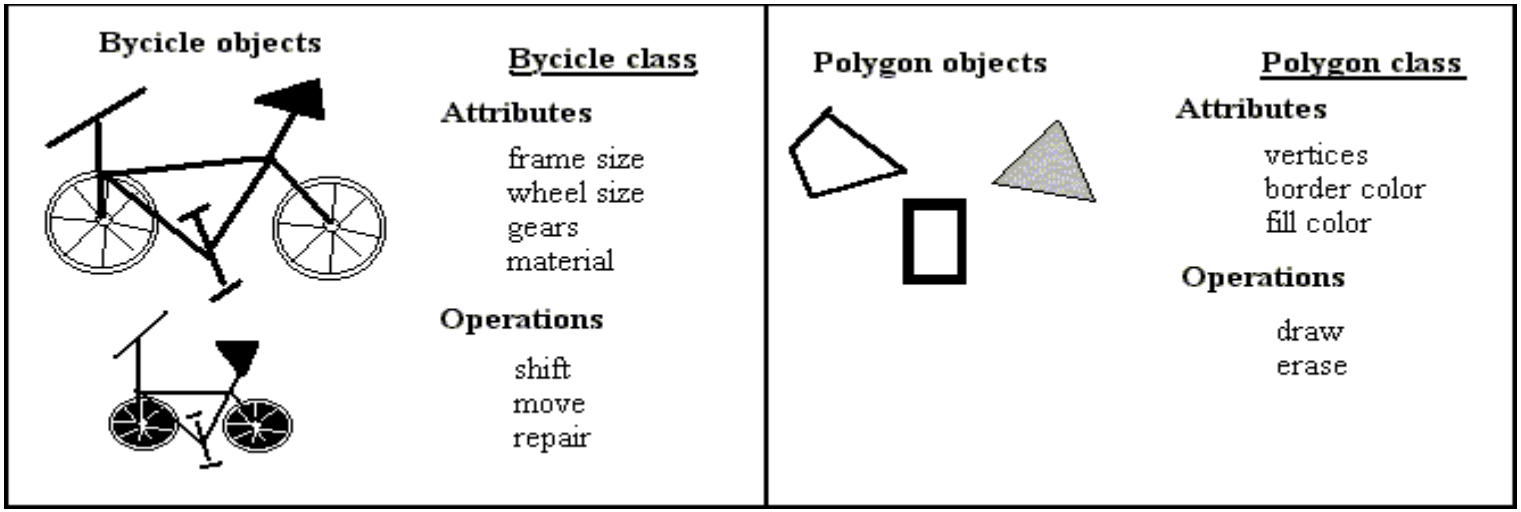
User-defined data types
- Time Angles
- Complex numbers
- Points on the plane

Data-storage constructs
- Customized arrays
- Stacks
- Linked lists
- Binary trees

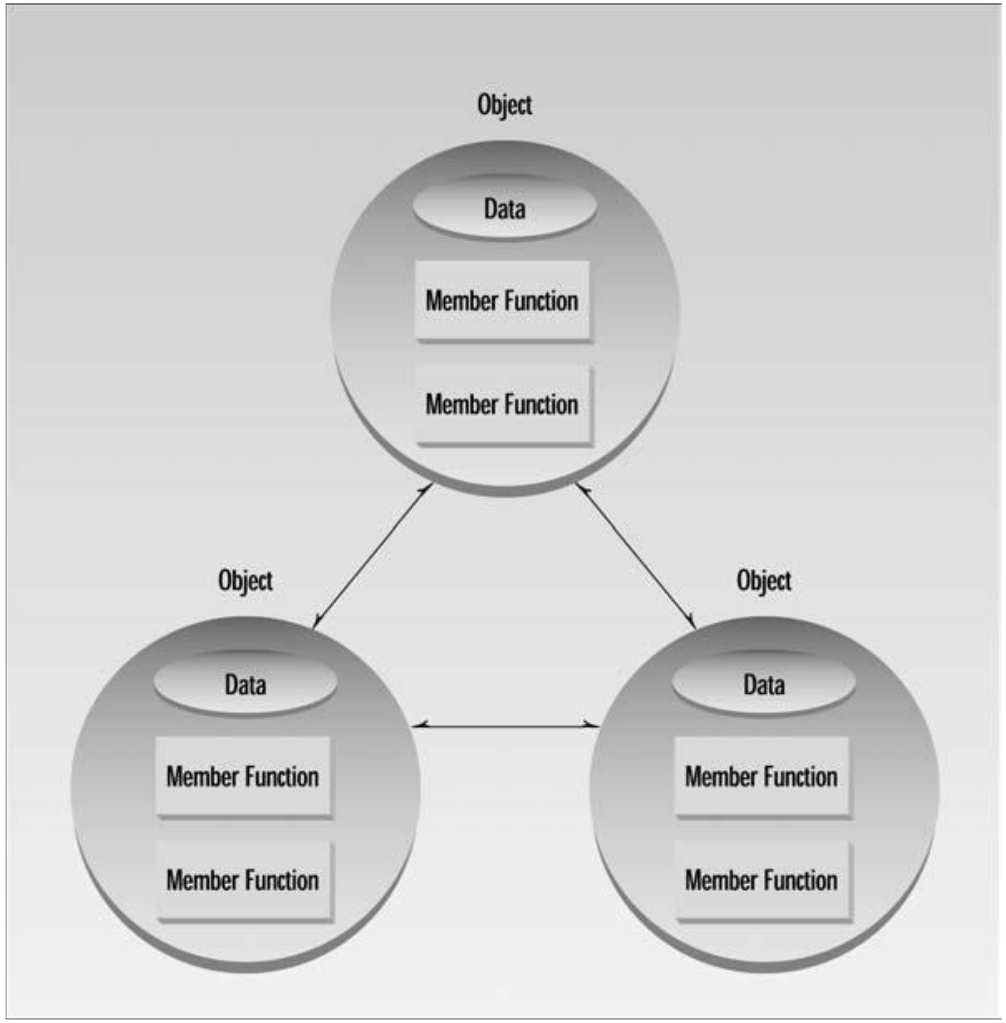
Human entities
- Employees
- Students
- Customers
- Salespeople

Class

- Objects with the same data structure ***(Attributes)*** and behavior ***(Methods or Operations)*** are grouped together called a ***class***
- Multiple objects can be created from the same class

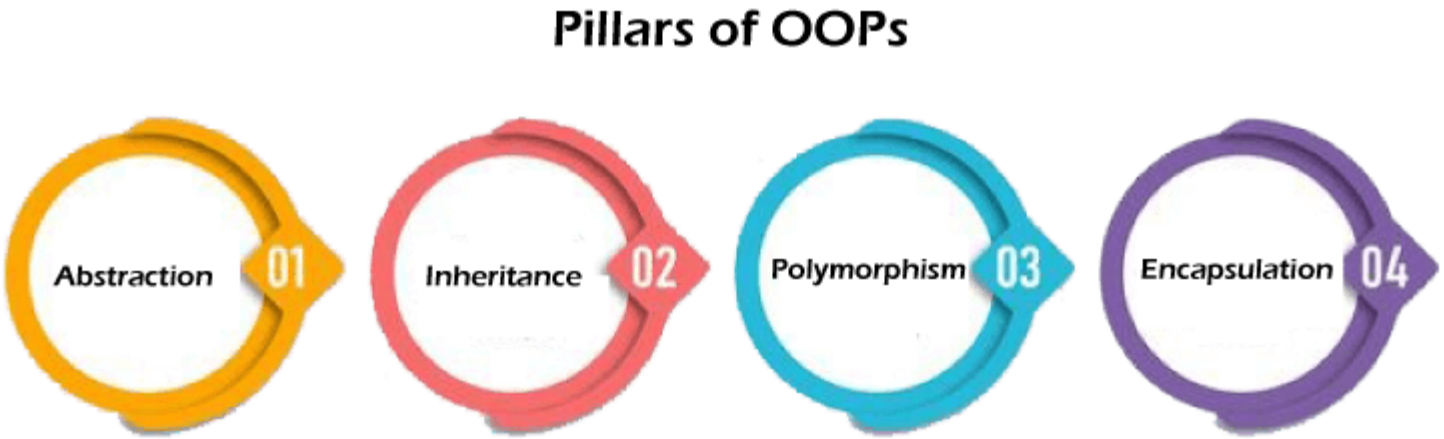


The object-oriented paradigm



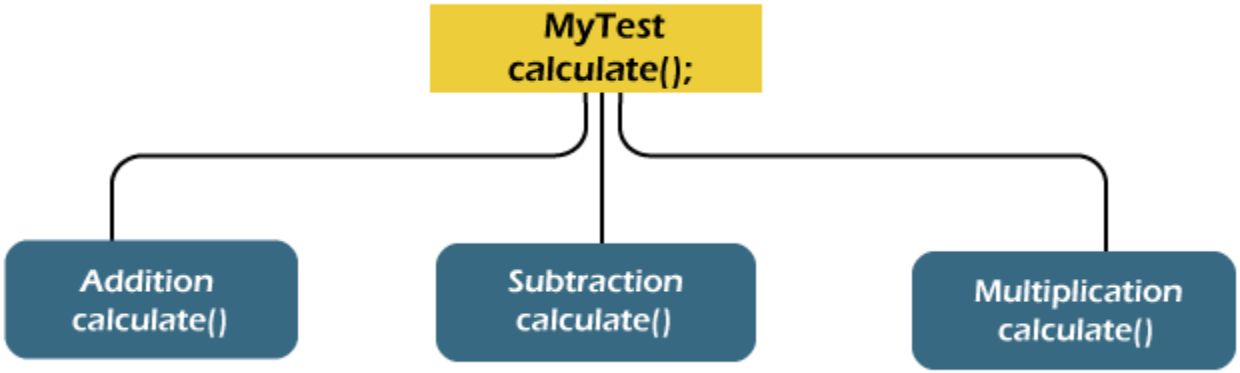
Pillars of OOPs

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



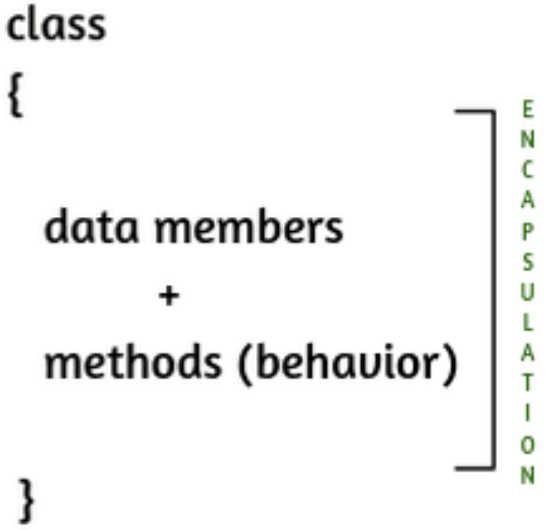
Pillars of OOPs

- Abstraction
  - Hide the implementation from the user but shows only essential information to the user



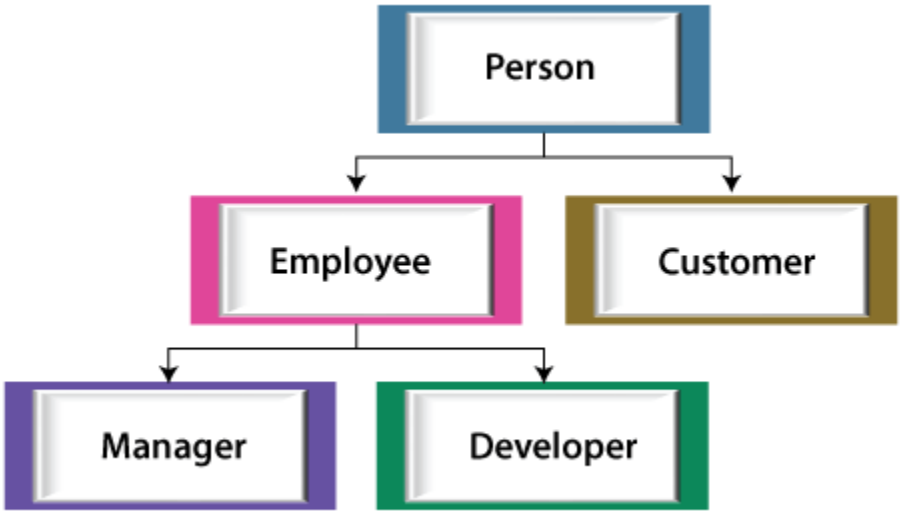
Pillars of OOPs

- Encapsulation
  - Mechanism that allows to bind data and functions of a class into an entity



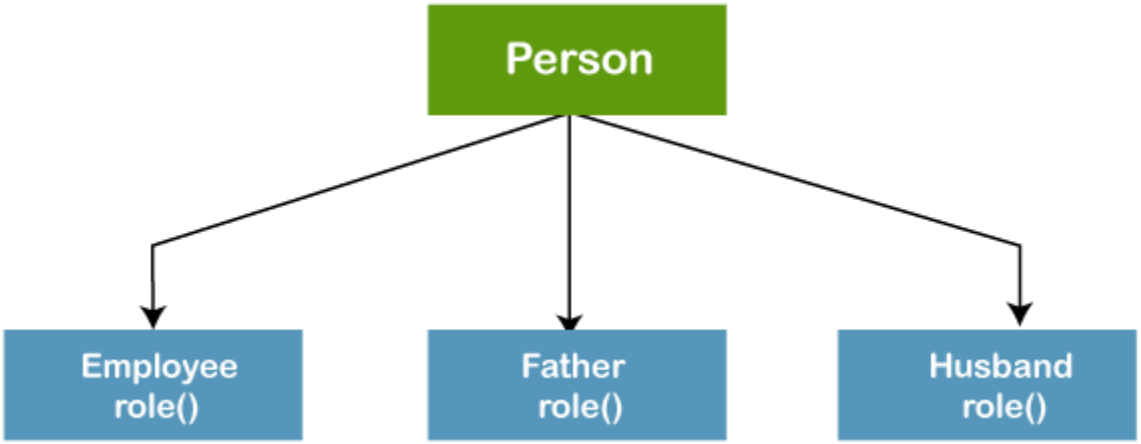
Pillars of OOPs

- Inheritance
  - Inherit or acquire the properties of an existing class (parent class) into a newly created class (child class)



Pillars of OOPs

- Polymorphism
  - Derived from the two words i.e. **ploy** and **morphs**. Poly means many and morphs means forms.
  - Create methods with the same name but different method signatures



Difference between Structured and OOP

Structured Programming

- Divides the code into modules or function
- Focuses on dividing the program into a set of functions in which each function works as a subprogram
- Main method communicates with the functions by calling those functions in the main program

Object Oriented Programming

- based on the concept of objects, contain both data and methods
- Focuses on representing a program using a set of objects which encapsulates data and object
- The objects communicate with each other by passing messages

Difference between Structured and OOP

Structured Programming

- No access specifiers
- **Data** is not secure
- Difficult to reuse code
- No way of data hiding

Object Oriented Programming

- Access specifiers such as private, public and protected
- **Data** is secured
- Easy to reuse code. E.g. inheritence
- Hide data is more secure

Procedural Approach	Object Oriented Approach

Public class **Circle**{
int radius;
Public void setRadius(int r)
{ radius = r;}
Public void showCircumference()
{
double c = 2*3.14*radius;
System.out.println("Circumferenceis"+ c);
}
Public static void main()
{
setRadius(5);
showCircumference();
//output would be 31.4
setRadius(10);
showCircumference();
// output would be 62.8
}
}
	Public class **Circle**{
	Private int radius;
	Public void setRadius(int r)
	{ radius = r;}
	Public void showCircumference()
	{
	double c = 2*3.14* radius;
	System.out.println("Circumference is"+
	c);
	}
	}
	Public class **runner**
	{
	Public static void main()
	{
	Circle c1= new circle();
	c1.setRadius(5);
	c1.showCircumference();
	//output would be 31.4; it belongs to c1
	Circle c2= new circle();
	c2.setRadius(10);
	c2.showCircumference();
	//output would be 62.8; it belongs to c2
	}
	}

Lecture-2

Defining Classes and Objects

Introduction

- Classes are the most important language feature that make *object-oriented programming* (*OOP*) possible
- Programming in Java consists of defining a number of classes
  - Every program is a class
  - All helping software consists of classes
  - All programmer-defined types are classes
- Classes are central to Java

Class Definitions

- You already know how to use classes and the objects created from them, and how to invoke their methods
  - For example, you have already been using the predefined **`String`** and **`Scanner`** classes
- Now you will learn how to define your own classes and their methods, and how to create your own objects from them

A Class Is a Type

- A class is a special kind of programmer-defined type, and variables can be declared of a class type
- A value of a class type is called an object or *an instance of the class*
  - If A is a class, then the phrases "bla is of type A," "bla is an object of the class A," and "bla is an instance of the class A" mean the same thing
- A class determines the types of data that an object can contain, as well as the actions it can perform

Primitive Type Values vs. Class Type Values

- A primitive type value is a single piece of data
- A class type value or object can have multiple pieces of data, as well as actions called *methods*
  - All objects of a class have the same methods
  - All objects of a class have the same pieces of data (i.e., name, type, and number)
  - For a given object, each piece of data can hold a different value

The Contents of a Class Definition

- A class definition specifies the data items and methods that all of its objects will have
- These data items and methods are sometimes called *members* of the object
- Data items are called *fields* or *instance variables*
- Instance variable declarations and method definitions can be placed in any order within the class definition

Class Definition

- Syntax

```
public class Class_Name {
        Instance_Variable_Declaration_1
        Instance_Variable_Declaration_2
                …
        Instance_Variable_Declaration_Last

        Method_Definition_1 Method_Definition_2
                ...
        Method_Definition_Last
}
```

The **`new`** Operator

- An object of a class is named or declared by a variable of the class type:

```
ClassName  classVar;
```

- The **`new`** operator must then be used to create the object and associate it with its variable name:

```
classVar = new ClassName();
```

- These can be combined as follows:

```
ClassName classVar = new ClassName();
```

Instance Variables and Methods

An object's data and methods can be invoked using dot operator(.) known as ***object member access operator***

- Instance variables can be defined as in the following two examples
  - Note the **`public`** modifier (for now):

```
public String  instanceVar1;
public int  instanceVar2;
```

- In order to refer to a particular instance variable, preface it with its object name as follows:

```
objectName.instanceVar1
objectName.instanceVar2
```

Instance Variables and Methods

- An invocation of a method that returns a value can be used as an expression anyplace that a value of the **`typeReturned`** can be used:

```
typeReturned tRVariable;
tRVariable = objectName.methodName();
```

- An invocation of a **`void`** method is simply a statement:

```
objectName.methodName();
```

State of an Object

- The state of an object (also known as its properties or attributes) is represented by data fields with their current values.
- Default state of object after its creation is default values in its attributes

```
class Student {
        String name;
        int age;
        boolean
        char gender;
        double marks;
}
```

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

                Student std1 = new Student();
                std1.name = "Riz";
                System.out.println(std1.age);
                Student std2 = new Student();
                std2.name = "Ali";
                System.out.println(std1.name);
                System.out.println(std2.name);
        }
}
```

Reference Data Fields and the null Value

- Default state of an object when it is created means, default values in data fields.
  - **Reference Type** has default value ***null*** ; when reference type variable is not referencing any object
  - **Numeric Type** has default value 0
  - **Boolean Type** has default value true
  - **Char Type** has \u0000
- Java assigns **no default** value to a local variable inside a method

Example

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

		Student std1 = new Student();
		System.out.println(std1.name);
		System.out.println(std1.age);
		System.out.println(std1.isScienceMajor);
		System.out.println(std1.gender);
		System.out.println(std1.marks);

	}
}
class Student {
	String name; // name has the default value null
	int age; // age has the default value 0
	boolean isScienceMajor; //isScienceMajor default value false
	char gender; // gender has default value '\u0000'
	double marks; // marks has default value 0.0
}
```

File Names and Locations

- Reminder: a Java file must be given the same name as the class it contains with an added `.java` at the end
  - For example, a class named `MyClass` must be in a file named `MyClass.java`
- For now, your program and all the classes it uses should be in the same directory or folder

Lecture – 3

Constructors

Methods

- There are two kinds of methods:
  - Methods that compute and return a value

    ```
    public typeReturned methodName(paramList)
    ```

  - Methods that perform an action – `void` methods

    ```
    public void methodName(paramList)
    ```

Parameters of Methods

- When a method is invoked, the appropriate values must be passed to the method in the form of *arguments*
  - Arguments are also called *actual parameters*
- The number and order of the arguments must exactly match that of the parameter list
- The type of each argument must be compatible with the type of the corresponding parameter
- Parameters can of two types
  - Primitive Types: public void sum(int a, int b)
  - Reference Types: public void showData(String name, Object obj)

Overloading

- *Overloading* is when two or more methods *in the same class* have the same method name
- To be valid, any two definitions of the method name must have different *signatures*
  - A signature consists of the name of a method together with its parameter list
  - Differing signatures must have different numbers and/or types of parameters
  - ***Removing ambiguity by giving different parameters***

Overloading and Automatic Type Conversion

- If Java cannot find a method signature that exactly matches a method invocation, it will try to use automatic type conversion
- In some cases of overloading, because of automatic type conversion, a single method invocation can be resolved in multiple ways
  - Ambiguous method invocations will produce an error in Java

Example

```
class Adder{
    int add(int a,int b){
        return a+b;
    }
    double add(double a,double b){
        return a+b;
    }
}
class ExampleOverloading{
    public static void main(String[] args){
        Adder add = new Adder();
        System.out.println(add.add(11,11));
        System.out.println(add.add(11.0,11));
    }
}
```

Initializing Objects

- Initializing an object mean assigning values to its data members
- Example
  - To create a `Point` object and initialize it:

```
Point p = new Point();
p.x = 3;
p.y = 8;      // tedious
```

- We'd rather pass the fields' initial values as parameters:
  - `Point p = new Point(3, 8);  // better!`
  - We can do this with most types of objects in Java

Constructors

- A *constructor* is a special kind of method that is designed to initialize the instance variables(or state)for an object:

```
public ClassName(anyParameters){
        Code
}
```

  - A constructor must have the same name as the class
  - A constructor has no type returned, not even **`void`**
  - Constructors are typically overloaded
- Two types of constructors in Java:
  - no-arg constructor (Default Constructor)
  - Parameterized constructor

Constructors

- A constructor is called when an object of the class is created using **`new`**

```
ClassName objectName = new ClassName(anyArgs);
```

  - This is the **only** valid way to invoke a constructor
  - A constructor cannot be invoked like an ordinary method
- If a constructor is invoked again (using **`new`**), the first object is discarded and an entirely new object is created

```
Student std1 = new Student(12,"Ali")
    std1 = new Student(13,"Hamza");
```

No-Argument Constructor

```
public class ConsDemo{
    public static void main(String args[]){
        Student std1 = new Student();
        Student std2 = new Student();
        Student std3 = new Student();
        std1.show();
        std2.show();
        std3.show();
    }
}
class Student{
    int id;
    String name;
    public void show(){
        System.out.println("ID: "+id+" and Name: "+name);
    }

}
```

```
ID: 0 and Name: null
ID: 0 and Name: null
ID: 0 and Name: null
```

Parameterized Constructor

```
public class ConsDemo{
    public static void main(String args[]){
        Student std1 = new Student(12,"Ali");
        Student std2 = new Student(13,"Hashir");
        Student std3 = new Student(14,"Ahmad");
        std1.display();
        std2.display();
        std3.display();
    } }
class Student{
    int id;
    String name;
    Student(int stdId, String stdName){
        id = stdId;
        name = stdName;
    }
    public void display(){
        System.out.println("ID: "+id+" and Name: "+name);
    }}
```

```
ID: 12 and Name: Ali
ID: 13 and Name: Hashir
ID: 14 and Name: Ahmad
```

Overloaded Constructor

```
public class ConsDemo{
    public static void main(String args[]){
        Student1 std1 = new Student1(12,"Ali");
        Student1 std2 = new Student1(13,"Hashir",23);
        std1.display();
        std2.display();
    }
}
class Student1{
    int id;
    String name;
    int age;
    Student1(int stdId, String stdName){
        id = stdId;
        name = stdName;
    }
    Student1(int stdId, String stdName, int stdAge){
        id = stdId;
        name = stdName;
        age = stdAge;
    }
    public void display(){
        System.out.println("ID: "+id+" Name: "+name+" Age: "+age);
    }
}
```

```
ID: 12 Name: Ali Age: 0
ID: 13 Name: Hashir Age: 23
```

Points About Constructor

- A default constructor has no parameters
- Java defines a constructor (default) if you do not define your own constructor
- Java will not provide default constructor, if you include even one constructor in your class
- If you include any constructors in your class, be sure to provide your own no-argument constructor
- You can invoke another method within the definition of a constructor

```
public Date(int monthInt, int day, int year){
    setDate(monthInt, day, year);
}
```

Points About Constructor

- In constructor, instance variables must have valid values.

```
public Student(String stdName, int stdAge, double stdMarks)
{
          name = stdName;
          if ((stdAge< 0) || (stdMarks< 0)){
                  System.out.println("Error: Negative age or marks.");
                  System.exit(0);
          }
          else{
                  age = stdAge;
                  marks= stdMarks;
          }
}
```

Common constructor bugs

- Re-declaring fields as local variables ("shadowing"):

```
public Point(int initialX, int initialY) {
        int x = initialX;
        int y = initialY;
    }
```

  - This declares local variables with the same name as the fields, rather than storing values into the fields. The fields remain 0.

- Accidentally giving the constructor a return type:

```
public void Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }
```

  - This is actually not a constructor, but a method named `Point`

- Missing Assignment to data:

```
public Point(int initialX, int initialY) {
        initialX = x;
        initialY = y;
}
```

  - This declares local variables with the same name as the fields, rather than storing values into the fields. The fields remain 0.

Exercise

- Create Time class with four overloaded constructors.
- Create three objects in the runner using any three of the constructors

Lecture – 4

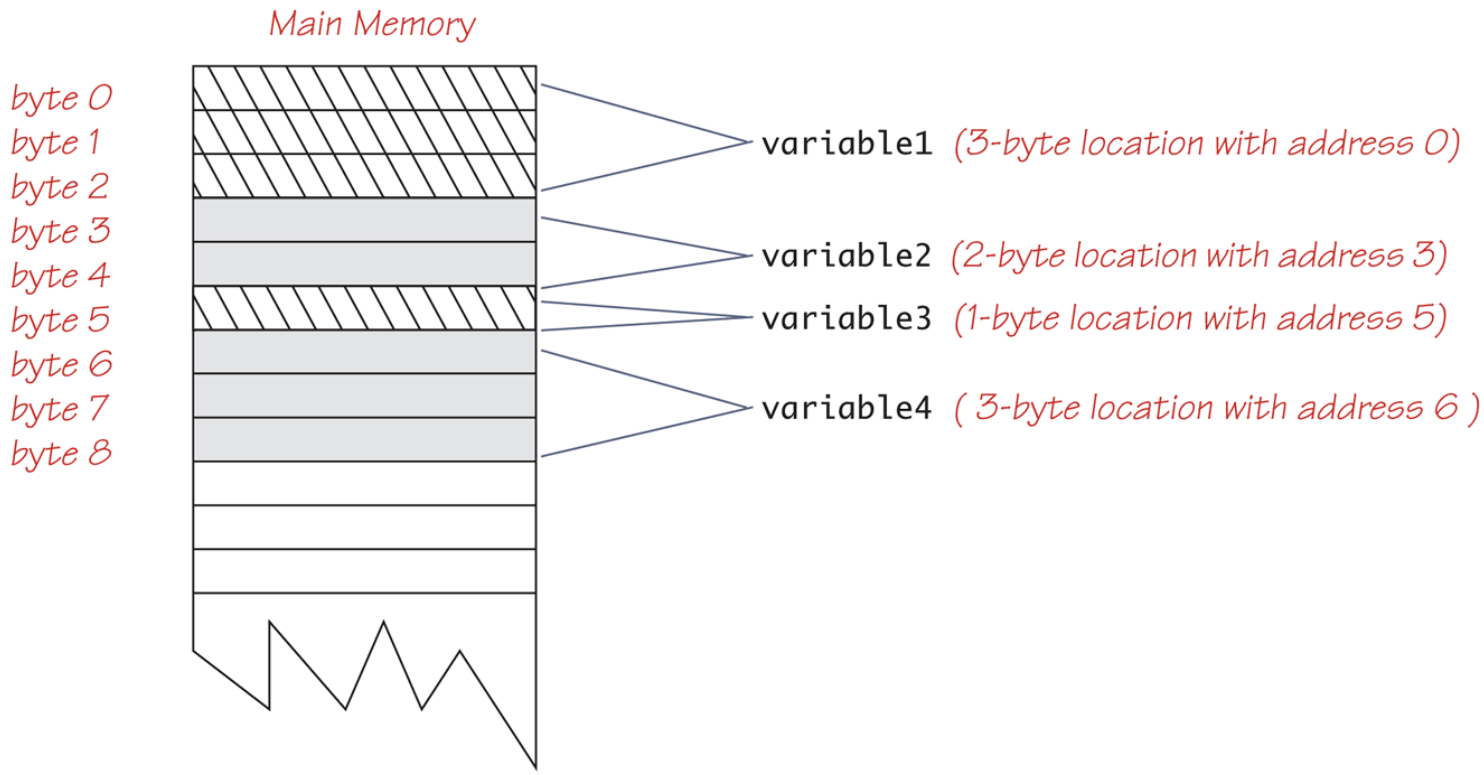
Objects and Memory

Variables and Memory

- Values of most data types require more than one byte of storage
  - Several adjacent bytes are then used to hold the data item
  - The entire chunk of memory that holds the data is called its *memory location*
  - The address of the first byte of this memory location is used as the address for the data item
- A computer's main memory can be thought of as a long list of memory locations of *varying sizes*

Variables in Memory

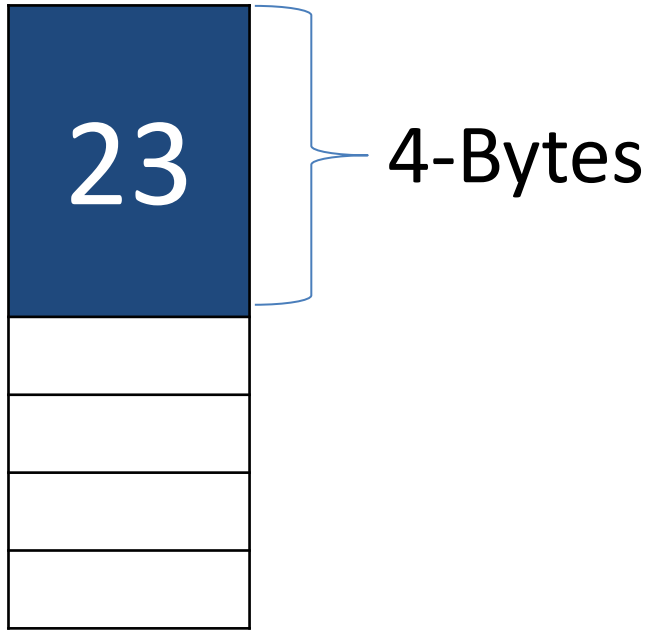
Display 5.10 Variables in Memory



References

- Every variable is implemented as a location in computer memory
- When the variable is a primitive type, the value of the variable is stored in the memory location assigned to the variable
  - Each primitive type always require the same amount of memory to store its values

```
int age = 23
```



References

- When the variable is a class type, only the memory address (or *reference*) where its object is located is stored in the memory location assigned to the variable
  - The object named by the variable is stored in some other location in memory
  - Like primitives, the value of a class variable is a fixed size
  - Unlike primitives, the value of a class variable is a memory address or reference
  - The object, whose address is stored in the variable, can be of any size

Class Type Variables Store a Reference (Part 1 of 2)

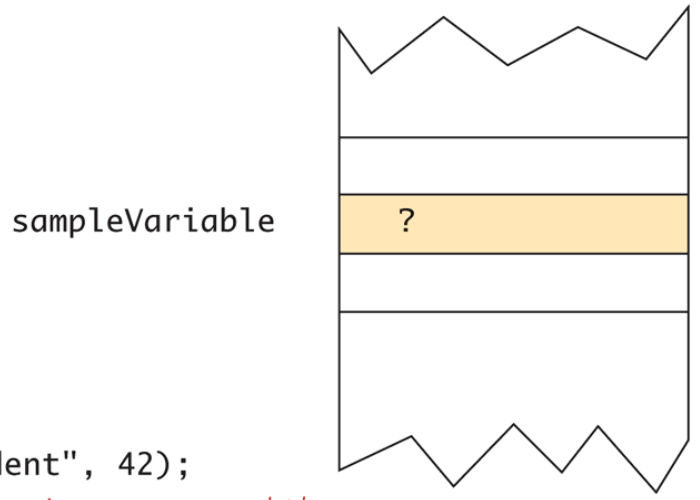
Display 5.12 Class Type Variables Store a Reference

```
public class ToyClass
{
    private String name;
    private int number;
```

The complete definition of the class `ToyClass` *is given in Display 5.11.*

```
ToyClass sampleVariable;
```

Creates the variable `sampleVariable` *in memory but assigns it no value.*



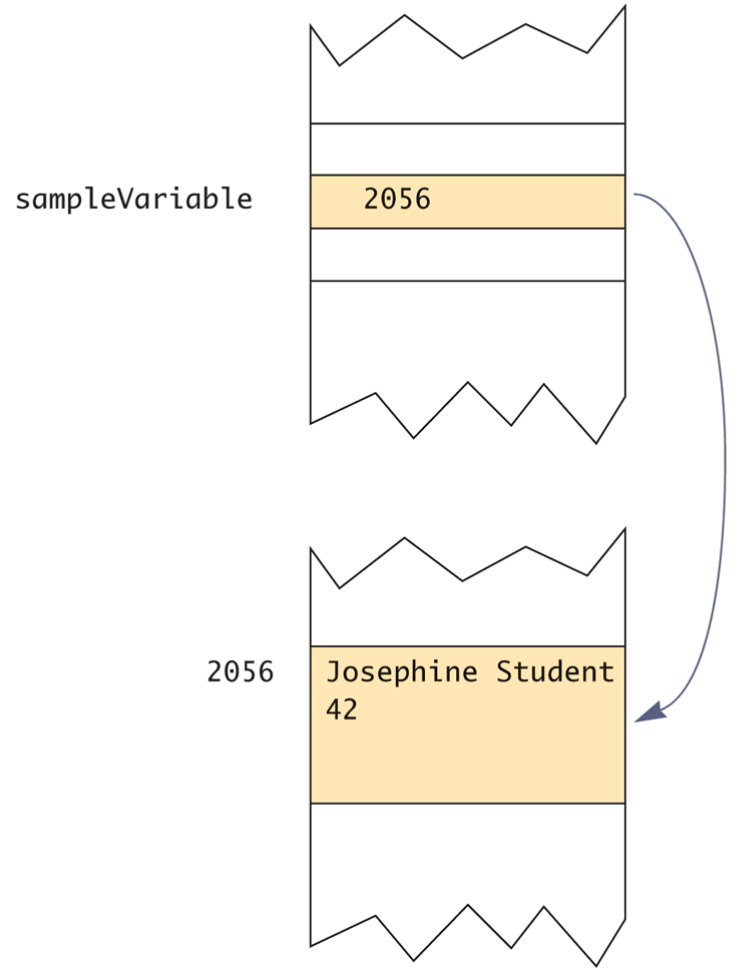

```
sampleVariable =
new ToyClass("Josephine Student", 42);
```

Creates an object, places the object someplace in memory, and then places the address of the object in the variable `sampleVariable`*. We do not know what the address of the object is, but let's assume it is* `2056`*. The exact number does not matter.*

(continued)

Class Type Variables Store a Reference (Part 2 of 2)

Display 5.12 Class Type Variables Store a Reference



For emphasis, we made the arrow point to the memory location referenced.

Assignment Operator with Class Type Variables

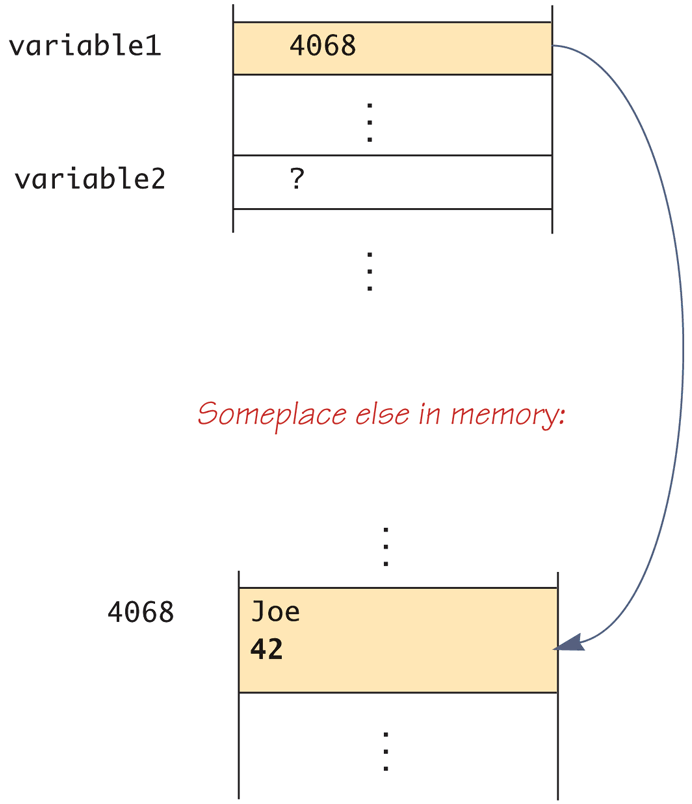
- Two reference variables can contain the same reference, and therefore name the same object
  - The assignment operator sets the reference (memory address) of one class type variable equal to that of another
  - Any change to the object named by one of these variables will produce a change to the object named by the other variable, since they are the same object

```
variable2 = variable1;
```

Assignment Operator with Class Type Variables (Part 1 of 3)

Display 5.13 Assignment Operator with Class Type Variables

```
ToyClass variable1 = new ToyClass("Joe", 42);
ToyClass variable2;
```



We do not know what memory address (reference) is stored in the variable `variable1`. *Let's say it is* `4068`. *The exact number does not matter.*

Note that you can think of

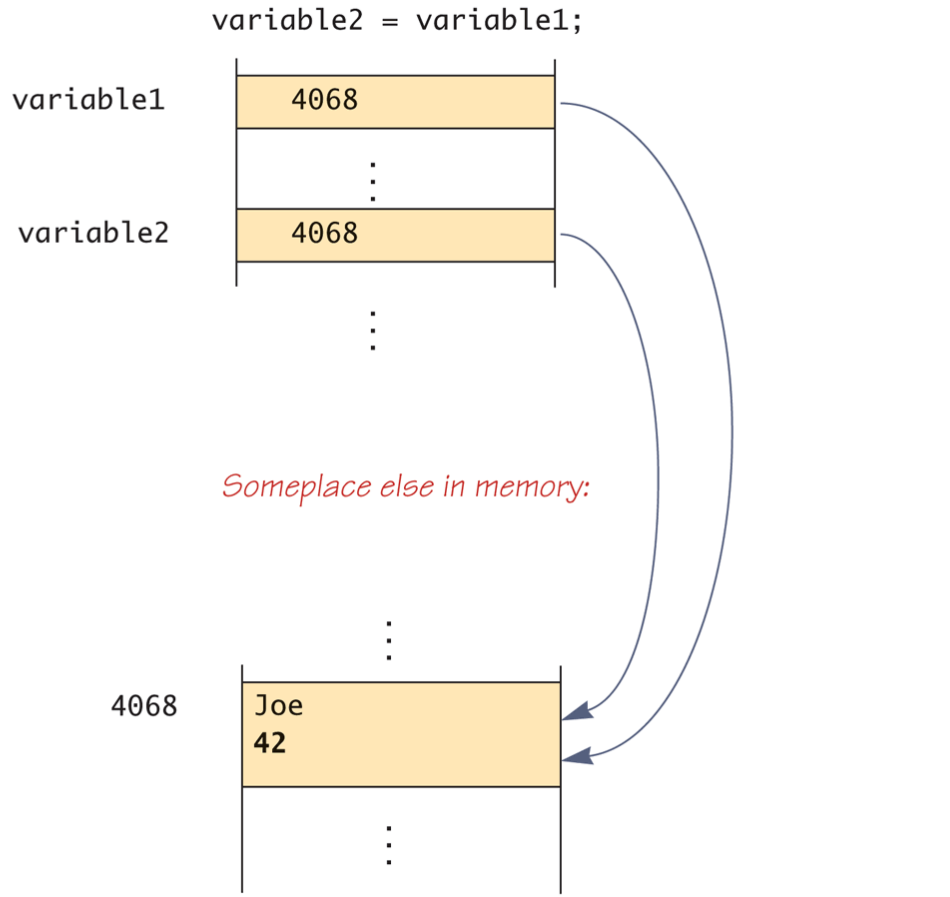
```
new ToyClass("Joe", 42)
```

as returning a reference.

(continued)

Assignment Operator with Class Type Variables (Part 2 of 3)

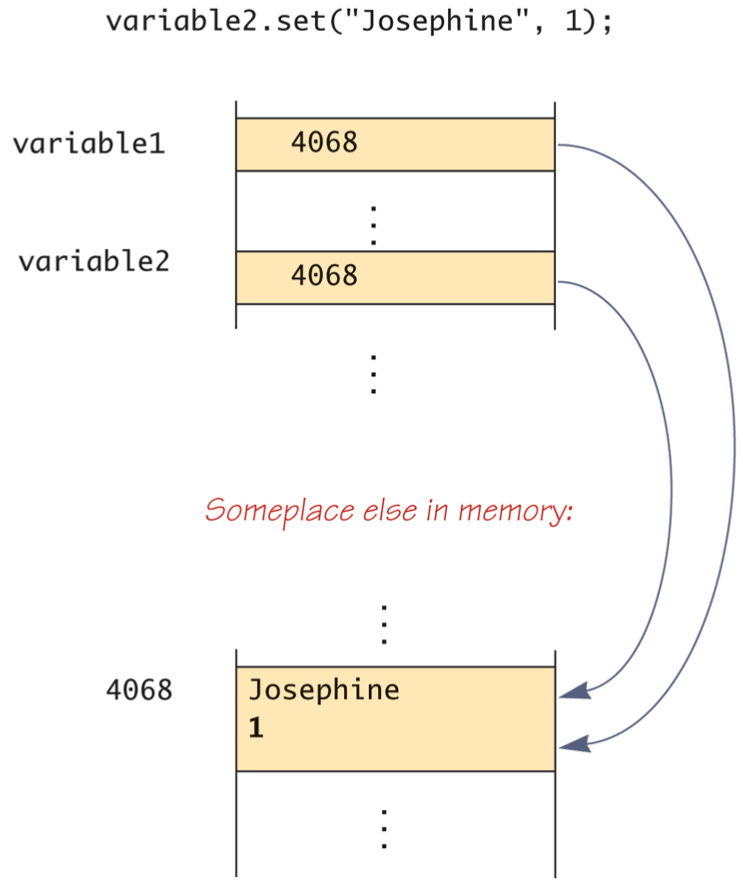
Display 5.13 Assignment Operator with Class Type Variables



(continued)

Assignment Operator with Class Type Variables (Part 3 of 3)

Display 5.13 Assignment Operator with Class Type Variables



The `equals` Method

- When the == operator is used with reference variables, the memory address of the objects are compared.
- The contents of the objects are not compared.

```
Stock stock1 = new Stock("GMX", 55.3);
Stock stock2 = new Stock("GMX", 55.3);
if (stock1 == stock2) // This is a mistake.
  System.out.println("The objects are the same.");
else
  System.out.println("The objects are not the same.");
```

- In above code segment only the addresses of the objects are compared.

The `equals` Method

- Java expects certain methods, such as **`equals`** to be in all, or almost all, classes
- The purpose of **`equals`**, a **`boolean`** valued method, is to compare two objects of the class to see if they satisfy the notion of "being equal"
  - Note: You cannot use **`==`** to compare objects

```
public boolean equals(ClassName objectName)
```

The `equals` Method

- Instead of using the == operator to compare two `Stock` objects, we should use the `equals` method.

```
public boolean equals(Stock object2)
{
   boolean status;

   if(symbol.equals(Object2.symbol) && sharePrice == Object2.sharePrice)
      status = true;
   else
      status = false;
   return status;
}
```

- Now, objects can be compared by their contents rather than by their memory addresses.

Methods That Copy Objects

- There are two ways to copy an object.
  - You cannot use the assignment operator to copy reference types

  - Reference only copy (shallow Copy)
    - This is simply copying the address of an object into another reference variable.

```
Stock stock1 = new Stock("GMX", 55.3);
Stock stock2 = stock1;
```

    Deep copy (correct)
    - This involves creating a new instance of the class and copying the values from one object into the new object.

Copy Constructors

- A copy constructor accepts an existing object of the same class and clones it

```
public Stock(Stock object1)
{
 if (object1 == null) //Not a real stock.
   {
     System.out.println("Fatal Error.");
     System.exit(0);
   }
      symbol = object1.symbol;
      sharePrice = object1.sharePrice;
}

// Create a Stock object
Stock company1 = new Stock("XYZ", 9.62);

//Create company2, a copy of company1
Stock company2 = new Stock(company1);
```

Lecture – 5

Passing Objects

Primitive Parameters

- Primitive types: boolean, byte, char, short, int, long, float, double
- In Java, all primitives are passed by value. This means a copy of the value is passed into the method
- Modifying the primitive parameter in the method does NOT change its value outside the method

Object Parameters

- Objects can be passed natively, just like primitives
- It is often misstated that Object parameters are passed by Reference.
- While it is true that the parameter is a reference to an Object, the reference itself is passed by Value.
- What we pass in method is a *handle of an object*, and in the c*alled method* a **new handle created** and **pointed to the same object**.
- Now when more than one handles tied to the same object, it is known as **aliasing**.

Object Parameters

Display 5.14 Parameters of a Class Type

```
1   public class ClassParameterDemo
2   {
3       public static void main(String[] args)
4       {
5           ToyClass anObject = new ToyClass("Mr. Cellophane", 0);
6           System.out.println(anObject);
7           System.out.println(
8                       "Now we call changer with anObject as argument.");
9           toy2.changer(anObject);
10          System.out.println(anObject);
11       }
12  }
```

ToyClass is defined in Display 5.11.

Notice that the method **changer** *changed the instance variables in the object* **anObject**.

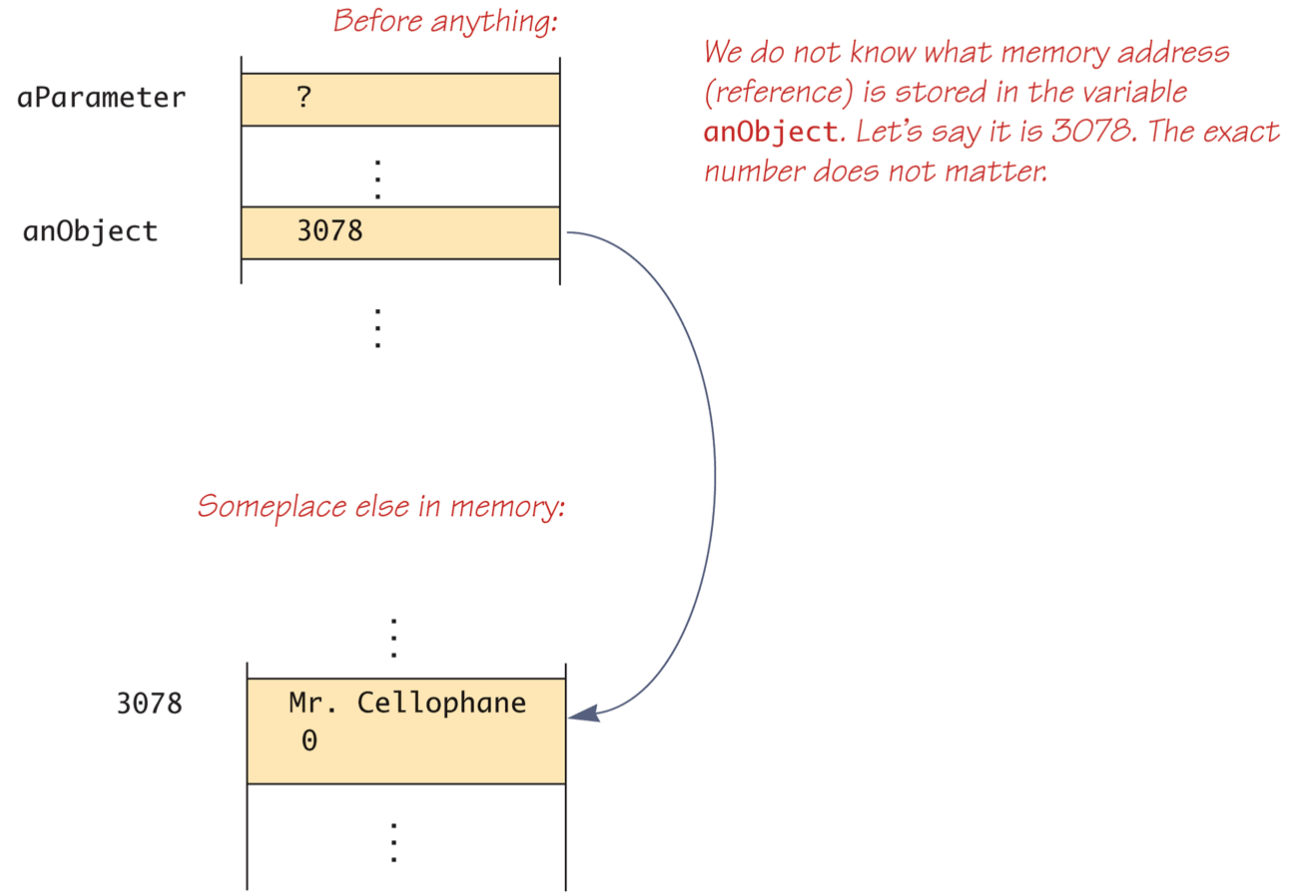
SAMPLE DIALOGUE

```
Mr. Cellophane 0
Now we call changer with anObject as argument.
Hot Shot 42
```

```
class ToyClass{
    private String name;
    private int number;
    public ToyClass(String initialName, int initialNumber){
        name = initialName;
        number = initialNumber;
    }
    public ToyClass(){
        name = "No name yet.";
        number = 0;
    }
    public static void changer(ToyClass aParameter){
        aParameter.name = "Hot Shot";
        aParameter.number = 42;
    }
    public void tryToMakeEqual(int aNumber){
        aNumber = number;
    }
    public boolean equals(ToyClass otherObject){
        return ((name.equals(otherObject.name)) && (number == otherObject.number) );
    }
    public String toString(){
        return (name + " " + number);
    }
}
```

Object Parameters - Memory Picture(Part 1 of 3)

Display 5.15 Memory Picture for Display 5.14

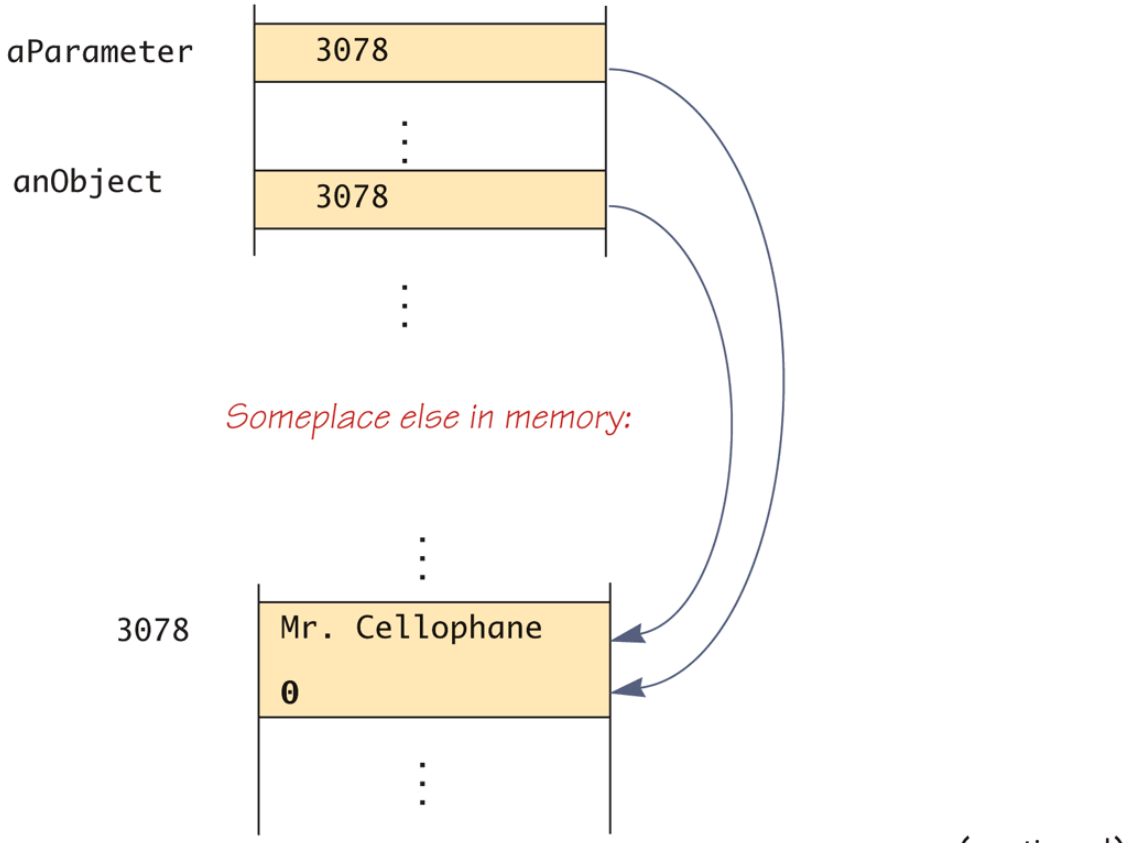


(continued)

Memory Picture for Display 5.14 (Part 2 of 3)

Display 5.15 Memory Picture for Display 5.14

`anObject` *is plugged in for* `aParamter`.
`anObject` *and* `aParameter` *become two names for the same object.*



(continued)

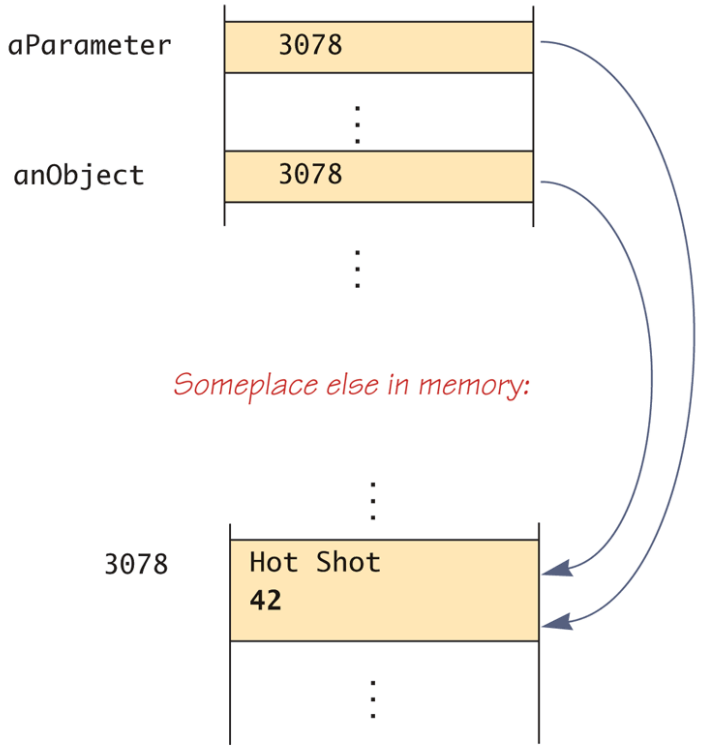
Memory Picture for Display 5.14 (Part 3 of 3)

Display 5.15 Memory Picture for Display 5.14

`ToyClass.changer(anObject);` *is executed and so the following are executed:*

```
aParameter.name = "Hot Shot";
aParameter.number = 42;
```

As a result, `anObject` *is changed.*



Return Objects From Methods

```
public class Complex{
    private double real;
    private double img;
    //Default Constructor
    public Complex(){
        real = 0.0;
        img  = 0.0;
    }
    //Overloaded Constructor
    public Complex(double r, double im){
        real = r;
        img  = im;
    }
    //Adding Two Complex objects and return Complex object
    public Complex addComplex(Complex b){
        double r = real + b.real;
        double i = img + b.img;
        //Create a temporary Complex to return it
        Complex temp = new Complex(r , i);
        return temp;
        //Or return new Complex(r , i);
    }
    //toString Method to display object values in instance variables

    public String toString(){
        return(real+" "+img);
    }
}
```

Main class

```
Complex c1 = new Complex(11 , 2.3);
Complex c2 = new Complex(9 , 2.7);
System.out.println("Complex-1: "+c1);
System.out.println("Complex-2: "+c2);

Complex c3 = c1.addComplex(c2);

System.out.println("Complex-3: "+c3);
```

```
Complex-1: 11.0 2.3
Complex-2: 9.0 2.7
Complex-3: 20.0 5.0
```

Objects can be passed natively, just like primitives

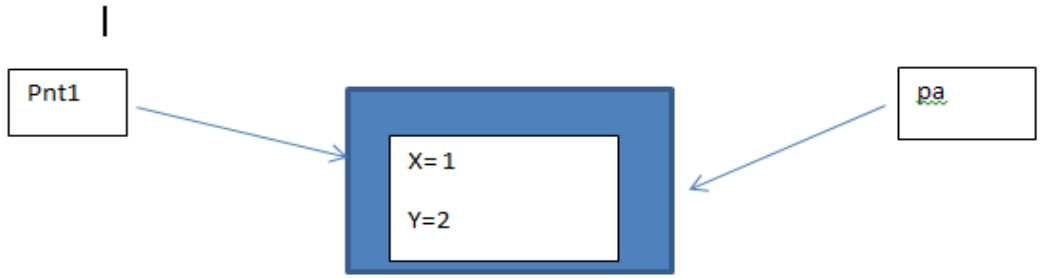
```
public class Point{
     public int x;
     public int y;
     public Point(int a, int b){
          x = a;
          y = b;
     }
     public Point(){}
     public void tricky(Point pa , Point pb){
          Point temp = new Point();
          temp = pa;
          pa   = pb;
          pb   = temp;
          System.out.println("pa.X: "+pa.x + " pa.Y: "+pa.y);
          System.out.println("pb.X: "+pb.x + " pb.Y: "+pb.y);
     }

}
```

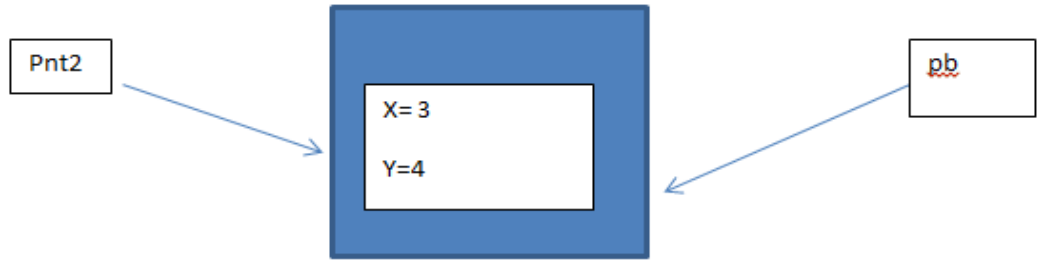
The method “tricky” is not performing swapping of object passed by main(), it swaps the objects in the function “tricky”

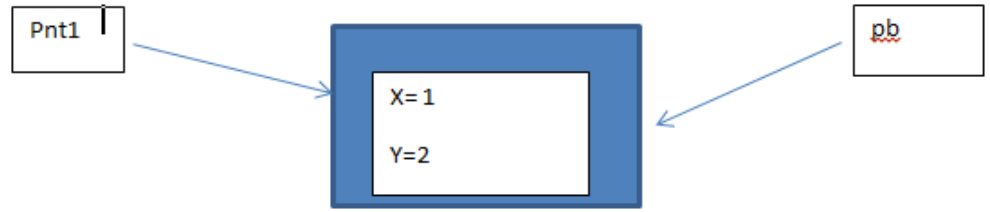
Main Method

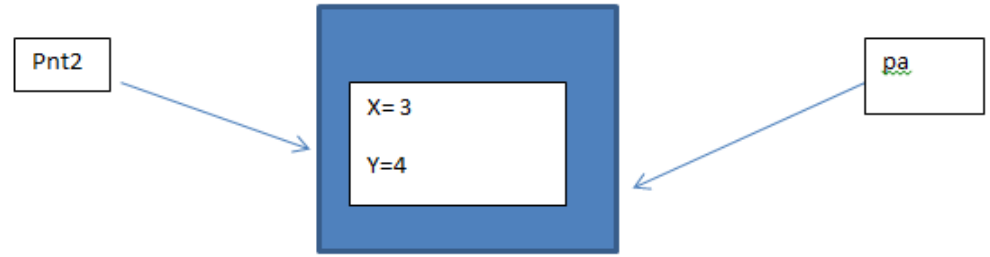
```
Point pnt1 = new Point(1,2);
Point pnt2 = new Point(3,4);
System.out.println("pnt1.X: "+pnt1.x + " pnt2.Y: "+pnt1.y);
System.out.println("pnt2.X: "+pnt2.x + " pnt2.Y: "+pnt2.y);

pnt1.tricky(pnt1 , pnt2);
System.out.println("pnt1.X: "+pnt1.x + " pnt1.Y: "+pnt1.y);
System.out.println("pnt2.X: "+pnt2.x + " pnt2.Y: "+pnt2.y);
```

```
pnt1.X: 1 pnt2.Y: 2
pnt2.X: 3 pnt2.Y: 4
pa.X: 3 pa.Y: 4
pb.X: 1 pb.Y: 2
pnt1.X: 1 pnt1.Y: 2
pnt2.X: 3 pnt2.Y: 4
```


Pnt1
X= 1
Y=2
pa


Pnt2
X= 3
Y=4
pb


Pnt1
X= 1
Y=2
pb


Pnt2
X= 3
Y=4
pa

The Constant `null`

- `null` is a special constant that may be assigned to a variable of any class type

```
YourClass yourObject = null;
```

- It is used to indicate that the variable has no "real value"
  - It is often used in constructors to initialize class type instance variables when there is no obvious object to use
- `null` is not an object: It is, rather, a kind of "placeholder" for a reference that does not name any memory location
  - Because it is like a memory address, use `==` or `!=` (instead of `equals`) to test if a class variable contains null

```
if (yourObject == null)
   System.out.println("No real object here.");
```

Pitfall: Null Pointer Exception

- Even though a class variable can be initialized to **`null`**, this does not mean that **`null`** is an object
  - **`null`** is only a placeholder for an object

- Any attempt to do this will result in a "Null Pointer Exception" error message

```
ToyClass2 aVariable = null ;
String representation = aVariable.toString();
```

Anonymous Objects

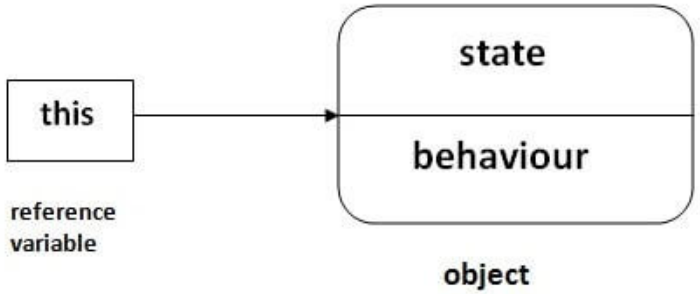
- Sometimes the object created is used as an argument to a method, and never used again
  - In this case, the object need not be assigned to a variable, i.e., given a name
- An object whose reference is not assigned to a variable is called an **anonymous object**

```
if (variable1.equals( new ToyClass("JOE", 42)))
    System.out.println("Equal");
else
    System.out.println("Not equal");
```

=

```
ToyClass temp = new ToyClass("JOE", 42);
if (variable1.equals(temp))
    System.out.println("Equal");
else
    System.out.println("Not equal");
```

this Pointer



- `this` is a **reference variable** that refers to the current object
- `this` can be used to refer current class instance variable
- `this` can be used to invoke current class method
- `this()` can be used to invoke current class constructor
- `this` can be passed as an argument in the method call
- `this` can be passed as argument in the constructor call
- `this` can be used to return the current class instance from the method

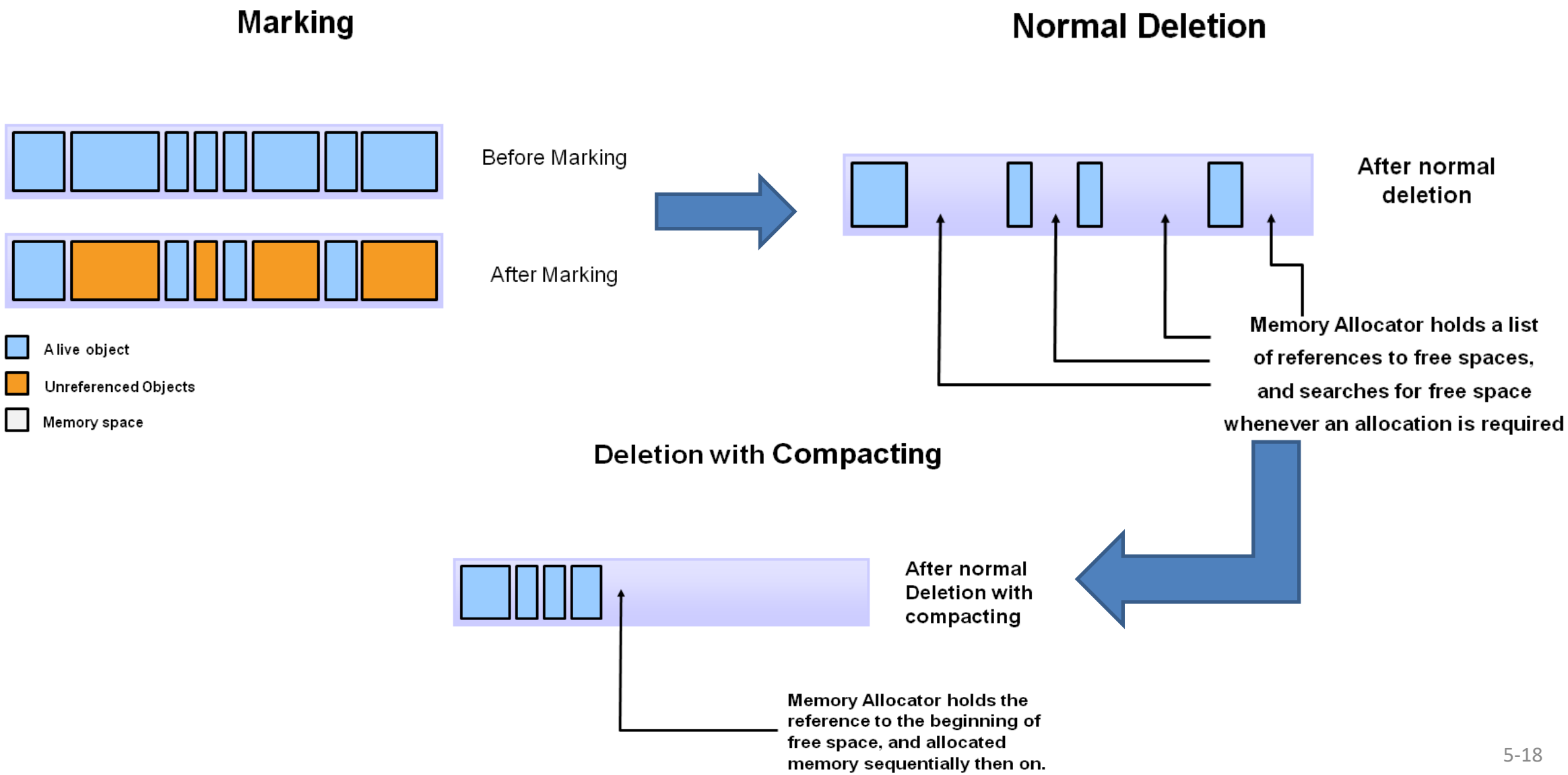
Important Points

- If another object is required for the operation of a method , we need to pass it through the argument.
- Class name is a user defined type.
- Class references can be used as function argument
- Class references can be returned from Functions
- Object is a composite entity
- Do not apply any arithmetic and logical operation on object name directly.

Java Garbage Collection

- In java, garbage means unreferenced objects.
- Garbage Collection is process of reclaiming the runtime unused memory automatically.
- Advantages
  - It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
  - It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

Garbage Collection - Basic Process



How can an object be unreferenced?

- By nulling a reference:

```
Employee e=new Employee();
e=null;
```

- By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

- By anonymous object:

```
new Employee();
```

Java Object finalize() Method

- Finalize() is the method of Object class
- Called just before an object is garbage collected

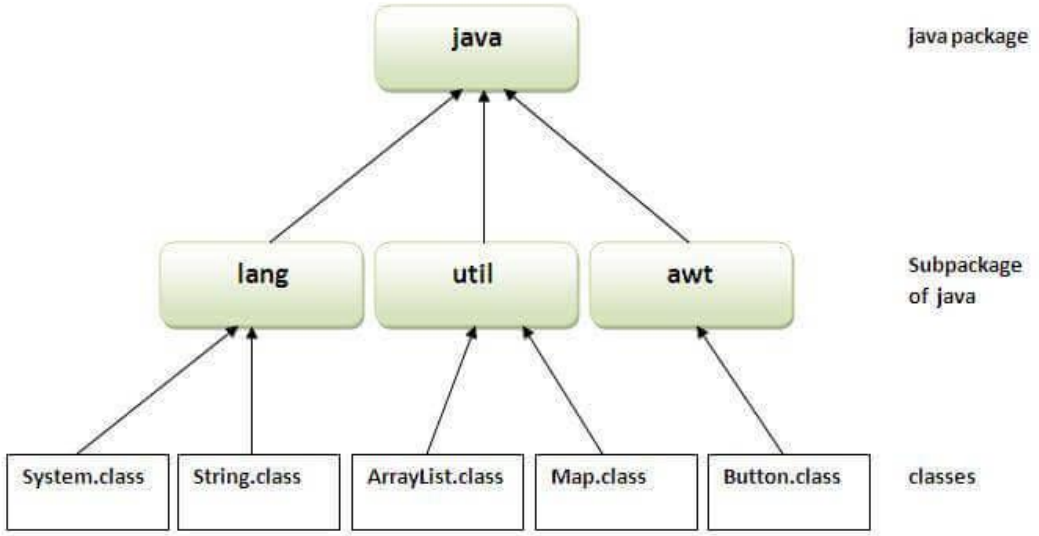
```
protected void finalize(){}
```

Lecture – 6

Encapsulation

Java Package

- A **java package** is a group of similar types of classes, interfaces and sub-packages.
  - **Built-in Packages** (java, lang, awt, javax, swing, net, io, util)
  - **User defined Packages**
- The **package keyword** is used to create a package in java.



Example

```
1.//save as Simple.java
2.package mypack;
3.public class Simple{
4. public static void main(String args[]){
5.    System.out.println("Welcome to package");
6.   }
7.}
```

How to compile java package

javac -d directory javafilename

Example

javac -d . Simple.java

- The -d switch specifies the destination where to put the generated class file.
- You can use any directory name like d:/abc (windows).
- If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

Java mypack.Simple

How to send the class file to another directory or drive?

To Compile:

e:\sources> javac -d c:\classes Simple.java

To Run:

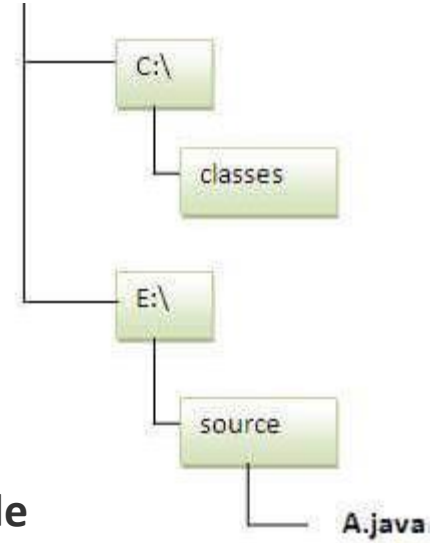
Set the class path and then run

e:\sources> set classpath=c:\classes;

e:\sources> java mypack.Simple

OR

e:\sources> java -classpath c:\classes mypack.Simple



Information Hiding

- **Information hiding** means that you separate the description of how to use a class from the implementation details,
  - Such as how the class methods are defined
  - Another term for information hiding is **abstraction**
  - To drive a car, do you need to know how the engine works? Why?
    - `println` method
      - need to know ***what*** the method does
      - but not ***how*** `println` does it
- Provide a more abstract view and hide the details

Defining Encapsulation

- **Encapsulation** means that the data and the actions are combined into a single item (in our case, a class object) and that the details of the implementation are hidden.
- ***Information hiding*** and ***Encapsulation*** are two sides of the same coin.
- If a class is well designed, a programmer who uses a class need not know all the details of the implementation of the class but need only know a much simpler description of how to use the class.

Controlling access to class members

- **Access Modifier**
  - Determines access rights for the class and its members
  - Defines where the class and its members can be used

Why use these

- It is important in many applications to hide data from the programmer
  - E.g., a password program must be able to read in a password and compare it to the current one or allow it to be changed
  - But the password should **never be accessed directly!**

```
public class Password {
        public String my_password;
                …
}
```

```
Password ProtectMe;
        …
ProtectMe.my_password = “backdoor”; // this is bad
```

Access Modifiers

- Member modifiers change the way class members can be used
- *Access modifiers* describe how a member can be accessed

Modifier	Description
`(no modifier) / default`	The access level is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
`public`	The access level is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.
`Protected`	The access level is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
`Private`	The access level is only within the class. It cannot be accessed from outside the class.

Access Modifiers

Access Modifier	Within class	Within Package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Encapsulating a Class

- Members of a class must always be declared with the minimum level of visibility.
- Provide *setters and getters* (also known as accessors/mutators) to allow *controlled* access to private data.
- Provide other public methods (known as *interfaces*) that other objects must adhere to in order to interact with the object.

Accessors and Mutators

- **Accessor methods**: Public methods that allow attributes (instance variables) to be read
  - Get methods are also commonly called accessor methods
  - Much better than making instance variables public
- **Mutator methods**: Public methods that allow attributes (instance variables) to be modified
  - Set methods are also commonly called mutator methods, because they typically change an object's state

Setters and Getters

- Setters and Getters allow controlled access to class data
- *Setters* are methods that (only) alter the state of an object
  - Use setters to validate data before changing the object state
- *Getters* are methods that (only) return information about the state of an object
  - Use getters to format data before returning the object's state

Example

```
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    Person myObj = new Person();
    myObj.name = "John";  // error
    System.out.println(myObj.name); // error
  }
}
```

Correct Solution

```
public class Main {
  public static void main(String[] args) {
    Person myObj = new Person();
    myObj.setName("John"); // Set the value of the name
variable to "John"
    System.out.println(myObj.getName());
  }
}
```

Using Reference of an Object Directly in a Setter

Inside the '**Array**' class we have a private array which is an '**element**'.

We have written a setElement() setter function in which values of one array are copied into the array declared inside the class.

So the '**element**' array values are set which is the same as the '**Arr**' array which is declared inside the main method.

```
class Array {
   private int[] element;

   //setter method to copy the value of a in element array
   void setElement(int[] a) {
      this.element = a;
   }

   //to print the value of the element array
   void display() {
      //using .length function
      //to find length of an array
      int len = (this.element).length;

      for(int i = 0; i < len; i++) {
         System.out.print(this.element[i] + " ");
      }
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      Array a1 = new Array();

      int Arr[] = {5, 8, 11, 22, 33};

      // calling the setter function
      a1.setElement(Arr);

      // calling the display function
      a1.display();

      // new value is set at the 0th index
      Arr[0] = 2;
      System.out.println();

      // calling the display function one more time
      a1.display();
   }
}
```

```
5 8 11 22 78
2 8 11 22 78
```

```
class Array {
   private int[] element;
    void setElement(int[] a) {
      int len2 = a.length;
      // dynamically allocating the memory to element[]
      // according to the a[] array length
      element = new int[len2];
      for(int i = 0; i < len2; i++) {
        // copying the value one by one
        // into the element array
         this.element[i] = a[i];
      }
   }
   // to print the value of the element array
   void display() {                  //using .length function
      //to find length of an array
      int len = (this.element).length;

      for(int i = 0; i < len; i++) {
         System.out.print(this.element[i] + " ");
      }
   }
}
```

```
5 8 11 22 78
5 8 11 22 78
```

```
public class Main {
   public static void main(String[] args) {
      Array a1 = new Array();

      int Arr[] = {5, 8, 11, 22, 78};

      // calling the setter function
      a1.setElement(Arr);

      // calling the display function
      a1.display();

      // new value is set at the 0th index
      Arr[0] = 2;
      System.out.println();

      // calling the display function one
more time
      a1.display();
   }
}
```

Lecture – 7

Static Data and Methods

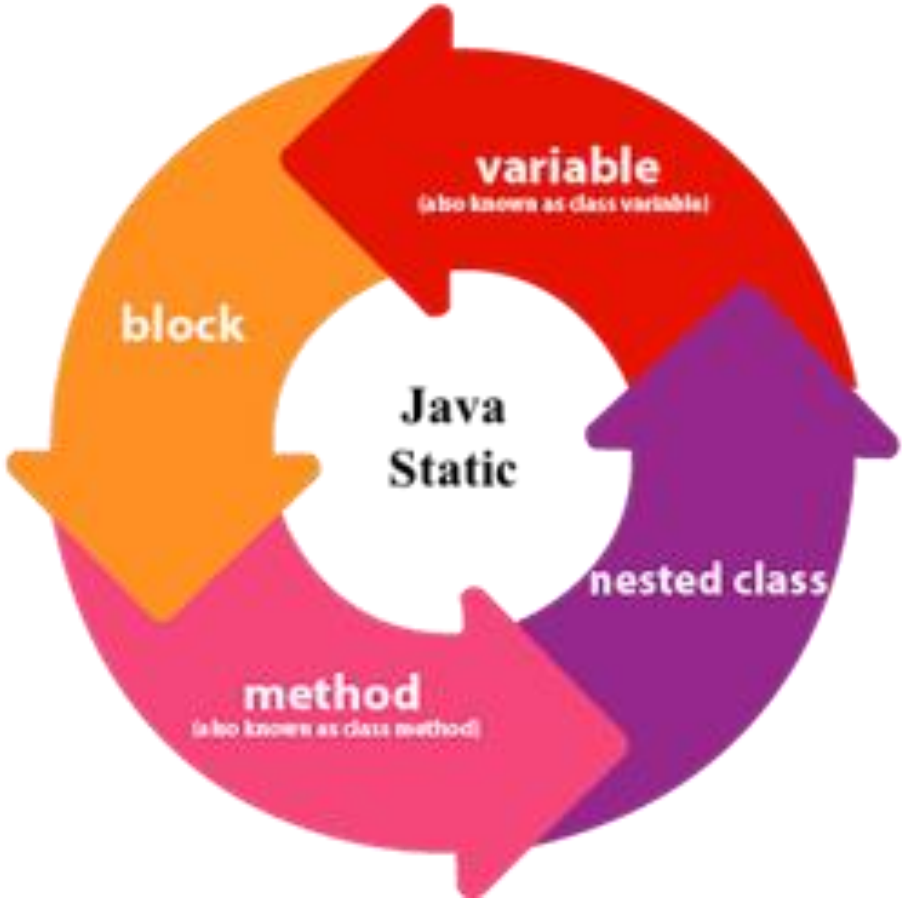
Java static keyword

- Java supports definition of global methods and variables that can be accessed without creating objects of a class. Such members are called **Static members**.
- Define a variable by marking with the static keyword .
- This feature is useful when we want to create a variable common to all instances of a class.
- One of the most common example is to have a variable that could keep a count of how many objects of a class have been created.
- Note: Java creates only one copy for a static variable which can be used even if the class is never instantiated.

Java static keyword

The static can be:

- Variable (also known as a class variable)
- Method (also known as a class method)
- Block
- Nested class



Understanding the Problem without `static`

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all **objects**. If we make it static, this field will get the memory only once.

```
class Student{
    int rollno;
    String name;
    String college="ITS";
}
```

- `static` is used for a constant variable or a method that is same for every instance of a class.

Example Static

```
class Student{
   int rollno;//instance variable
   String name;
   static String college ="CUI";//static variable
   Student(int r, String n){
     rollno = r;
     name = n;
   }
   Student(){}
   //method to display the values
   void display (){
      System.out.println(rollno+" "+name+" "+college);
   }
}
```

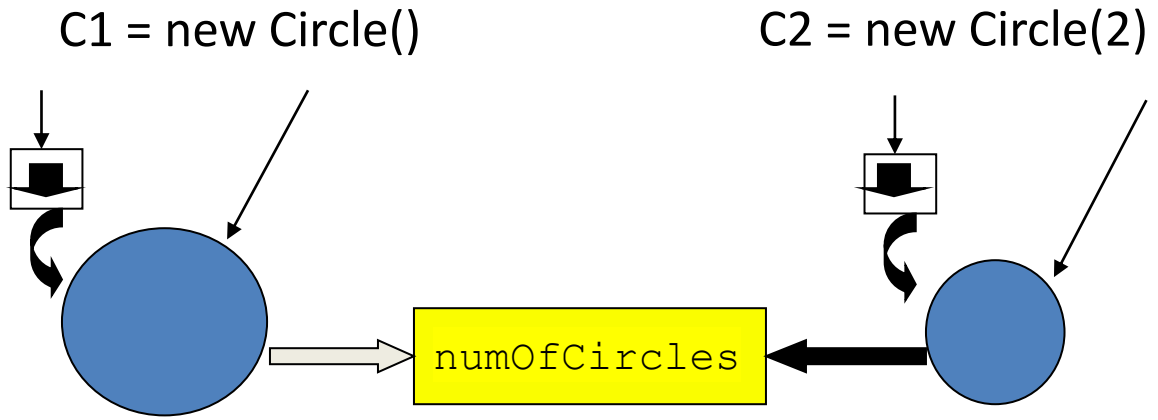
```
public class StudentTest{
  public static void main(String[] args){
     Student s1 = new Student(111,"Ahmad");
      Student s2 = new Student(222,"Zainab");
      s1.display();
      s2.display();

   }
}
```

```
111 Ahmad CUI
222 Zainab CUI
```

Defining Class Variable as Static

```
public class Circle{
    public static int numOfCircles;
    private int radius;
    private final double PI = 3.14;
    public Circle(){
        radius = 0;
        numOfCircles++;
    }
    public void setRadius(int r){
        this.radius = r;
    }
    public Circle(int radius){
        this.radius = radius;
        numOfCircles++;
    }
    public double getArea(){
        return radius * radius * PI;
    }

}
```

```
public class CircleTest{
    public static void main(String[] args){
        Circle c1 =  new Circle();
        c1.setRadius(2);
        System.out.println("C1 Area: "+ c1.getArea());
        Circle c2 = new Circle(4);
        System.out.println("C2 Area: "+ c2.getArea());
        System.out.println("So far we have "+
                Circle.numOfCircles+" circle objects");

    }
}
```



Non-static Vs Static Variables

- Non-static variables : One copy per object. Every object has its own instance variable.
  - E.g. `radius` in the circle

- Static variables : One copy per class.
  - E.g. `numOfCircles` (total number of circle objects created)

Important Points

- *Use a static variable when all objects of a class must use the same copy of the variable.*
- Static variables have class scope. We can access a class's public static members through a reference to any object of the class(`c1.numOfCircles`), or with the class name and a dot (.)(Circle.numOfCircles)

- A class's private static class members can be accessed by client code only through methods of the class.

Static Methods

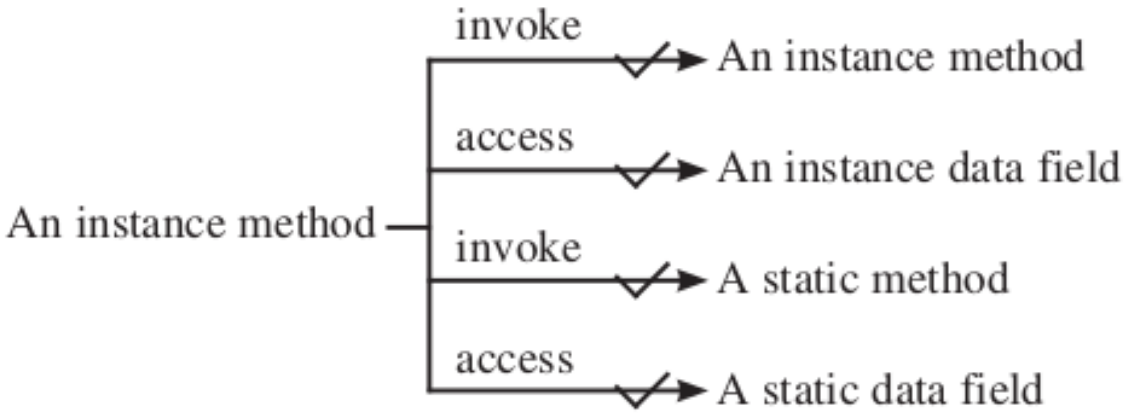
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

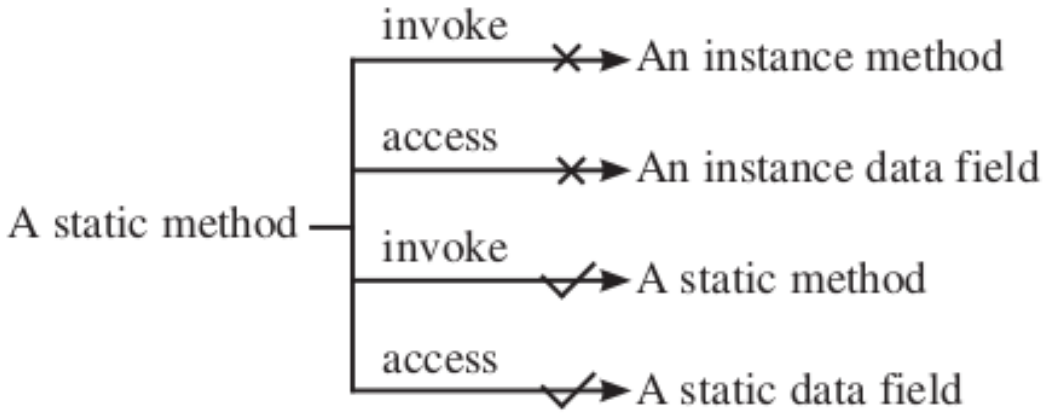
```
class Student{
   int rollno;//instance variable
   String name;
   private static String college ="CUI";//static variable
   Student(int r, String n){
     rollno = r;
     name = n;
   }
   Student(){}
   void display (){
      System.out.println(rollno+" "+name+" "+college);
   }
    public static void changeUni(String uni){
        university = uni;
   }

}
```

```
111 Ahmad QAU
222 Zainab QAU
```

```
public class StudentTest{
  public static void main(String[] args){
     Student s1 = new Student(111,"Ahmad");
     Student s2 = new Student(222,"Zainab");
     Student.changeUni("QAU");
     s1.display();
     s2.display();

   }
}
```

Important Points





- They can only call other static methods.
- They can only access static data.
- They cannot refer to "this" or "super" (more later) in anyway.

Example

```
1  public class A {
2    int i = 5;
3    static int k = 2;
4
5    public static void main(String[] args) {
6      int j = i; // Wrong because i is an instance variable
7      m1(); // Wrong because m1() is an instance method
8    }
9
10   public void m1() {
11     // Correct since instance and static variables and methods
12     // can be used in an instance method
13     i = i + k + m2(i, k);
14   }
15
16   public static int m2(int i, int j) {
17     return (int)(Math.pow(i, j));
18   }
19 }
```

Example

```
public class Test {
  public int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i ++)
      result *= i;

    return result;
  }
}
```

(a) Wrong design

```
public class Test {
  public static int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

(b) Correct design

Independent of any specific instance.

Example

- Suppose that the class F is defined in (a). Let f be an instance of F. Which of the statements in (b) are correct?

```
public class F {
  int i;
  static String s;

  void imethod() {
  }

  static void smethod() {
  }
}
```

(a)

```
System.out.println(f.i);
System.out.println(f.s);
f.imethod();
f.smethod();
System.out.println(F.i);
System.out.println(F.s);
F.imethod();
F.smethod();
```

(b)

Example – static data and methods

- Create a `SavingsAccount` class.
- Use a static data member `annualInterestRate` to store the annual interest rate.
- The class contains a private data member `savingsBalance` indicating the balance of account.
- Provide member function `calculateMonthlyInterest` that calculates the monthly interest by multiplying the balance by `annualInterestRate` divided by 12; this interest should be added to `savingsBalance`.
- Provide a static member function `modifyInterestRate` that sets the static `annualInterestRate` to a new value.

Example – static data and methods

- Write a driver program to test class `SavingsAccount`. Instantiate two different objects of class `SavingsAccount`, saver1 and saver2, with balances of $2000.00 and $3000.00, respectively.
- Set the `annualInterestRate` to 3 percent.
- Then calculate the monthly interest and print the new balances for each of the savers.
- Then set the `annualInterestRate` to 4 percent, calculate the next month's interest and print the new balances for each of the savers.

Example - Calculator

- Create a class `TwoDigitCalculator` which allows user to perform addition, subtraction, multiplication and division on 2 digits.

- Analyse the data members and methods of this class and implement

Lecture – 8

Array of Objects

Introduction

- Arrays can be used to hold multiple objects of same type.
- Creating an array of class type results in the creation of a list of references that will point to the objects.

String class Example

- Strings Arrays
  - String city[] = new String[5];

  - city[0] = new String(“Melbourne”);
  - city[1] = new String(“Sydney”);

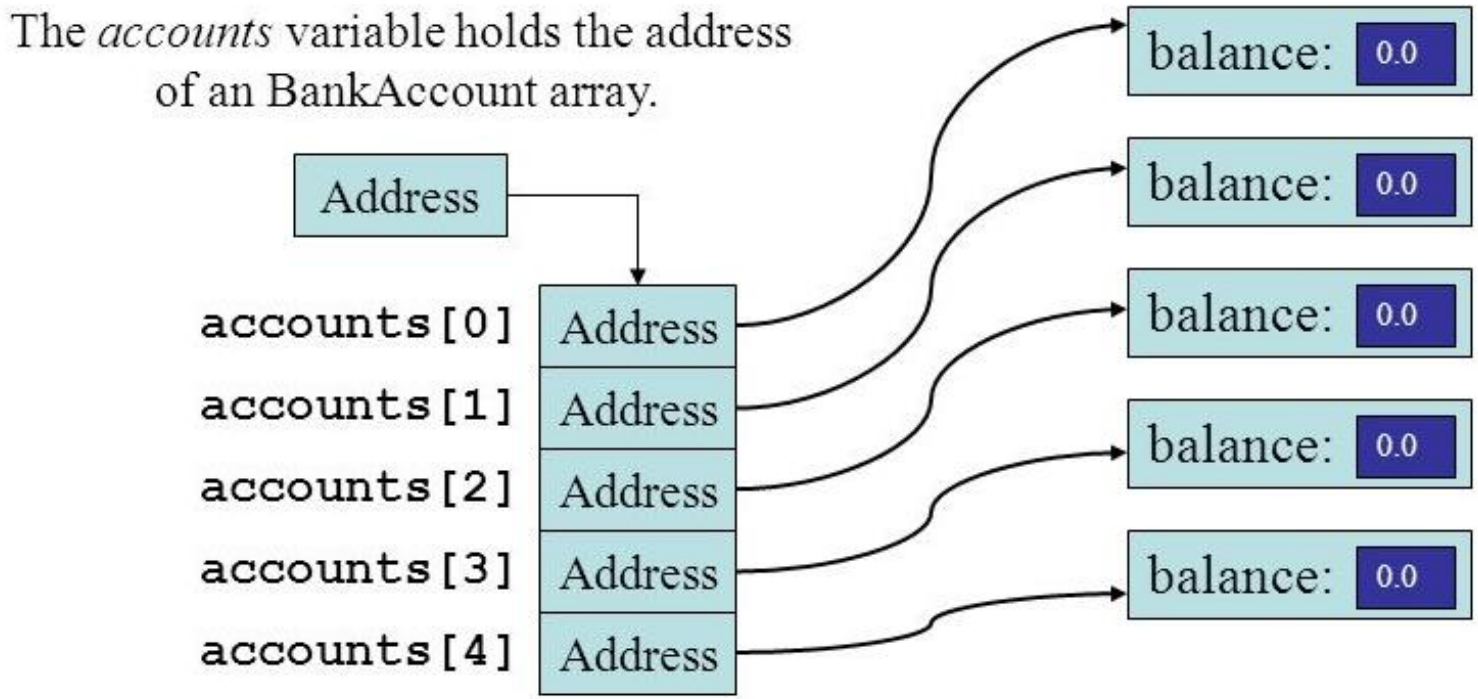
BankAccount class Example

- Consider a class BankAccount with balance as data member:

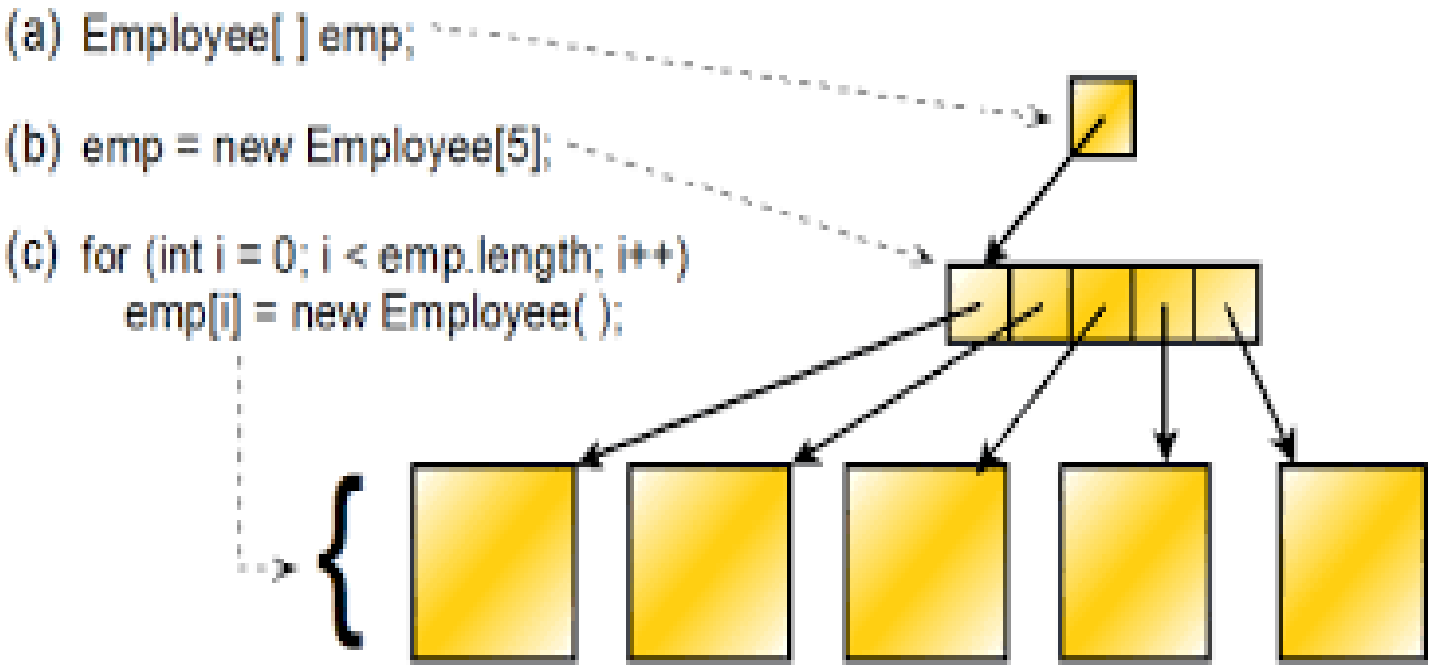
```
public class BankAccount {

    private double balance;

    public BankAccount(double balance) {
        this.balance = balance;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }

}
```

```
public class BankAccountRunner {
    public static void main (String [] arg)
    {
        BankAccount accounts [] = new BankAccount[5];
        for ( int i = 0; i<5 ; i++)
        {
        accounts[i] = new BankAccount (0);
        }
    }

}
```

Memory Structure – Object array



Memory Structure – Object array



Lecture – 9

Immutable Classes and
String as Immutable Class

Mutable Classes

- **mutation**: A modification to the state of an object.
- Objects can be mutable or immutable, depending on whether they can be changed after they are created.
- Mutable classes are classes whose instances can be changed after they are created.
  - Examples of mutable classes in Java include ArrayList, StringBuilder, and HashMap.

Immutable Classes

- **immutable**: Unable to be changed (mutated).
  - Basic idea: A class with no "set" methods (*mutators*).
- In Java, Strings are immutable.
  - Many methods appear to "modify" a string.
  - But actually, they create and return a new string (*producers*).

"Modifying" strings

- What is the output of this code?

```
String name = "pakistan";
name.toUpperCase();
System.out.println(name);
```

- The code outputs `pakistan` in lowercase.
- To capitalize it, we must reassign the string:

```
name = name.toUpperCase();
```

  – The `toUpperCase` method is a producer, not a mutator.

If Strings were mutable...

- What could go wrong if strings were mutable?

```
public Employee(String name, ...) {
    this.name = name;
    ...
}

public String getName() {
    return name;
}
```

  - A client could accidentally damage the Employee's name.

```
String s = myEmployee.getName();
s.substring(0, s.indexOf(" "));  // first name
s.toUpperCase();
```

Making a class immutable

- Don't provide any methods that modify the object's state.
- Declare the class as final (class cannot be extended)(later)
- Make all fields `final`.
- Make all fields `private`. (ensure encapsulation)
- Ensure exclusive access to any mutable object fields.
  - Don't let a client get a reference to a field that is a mutable object(Example Student dateCreated).

Mutable Fraction class

```
public class Fraction {
    private int numerator, denominator;

    public Fraction(int n)
    public Fraction(int n, int d)
    public int getNumerator(),  getDenominator()

   public void add(Fraction other) {
     numerator = numerator * other.denominator
               + other.numerator * denominator;
     denominator = denominator * other.denominator;
}

}
```

Immutable methods

```
// mutable version
public void add(Fraction other) {
    numerator = numerator * other.denominator
              + other.numerator * denominator;
    denominator = denominator * other.denominator;
}

// immutable version
public Fraction add(Fraction other) {
    int n = numerator  * other.denominator
          + other.numerator * denominator;
    int d = denominator * other.denominator;
    return new Fraction(n, d);
}
```

- former mutators become *producers*
  - create/return a new immutable object rather than modifying this one

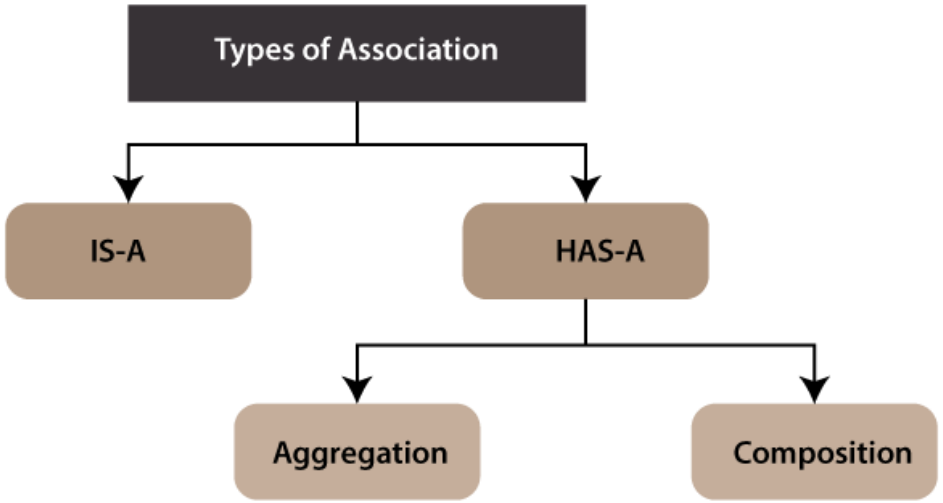
Lecture – 10

Association, Aggregation
and Composition

Association

- Interaction of different objects in OO model (or in problem domain) is known as **association**
- In object-oriented model, objects interact with each other to perform some useful work
  - Modeling these objects (entities) is done using the association
- This association can be represented with a line along an arrowhead () or without arrowhead

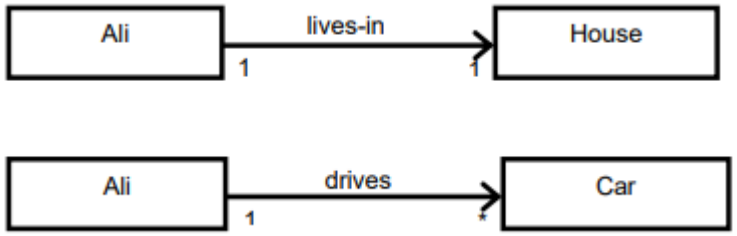
Types of Association



- **Class Association – Inheritance (IS-A)**
- **Object Association**
  - It is the interaction of stand-alone objects of one class with other objects of anther class.
  - **Simple Association or Association**
  - **Composition**
  - **Aggregation**

Types of Association

- Simple Association
  - The two interacting objects have no intrinsic relationship with other object
  - It is the weakest link between objects.
  - It is a reference by which one object can interact with some other object
  - Example:
    - Customer gets cash from cashier
    - Employee works for a company
    - Ali lives in a house
    - Ali drives a car

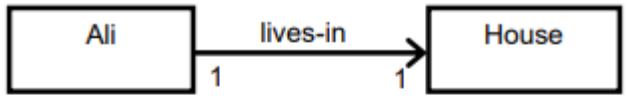


Types of Association

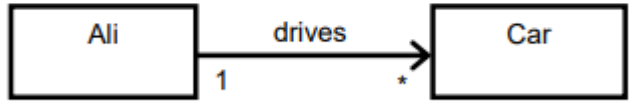
- Simple association can be categorized in two ways

With respect to direction (navigation)

One-Way Association

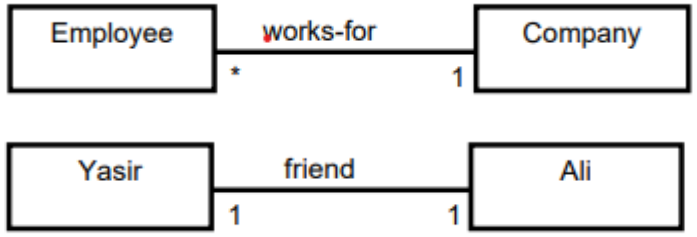


Ali lives in a House



Ali drives his Car

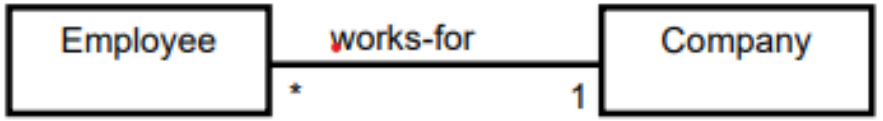
Two-Way Association



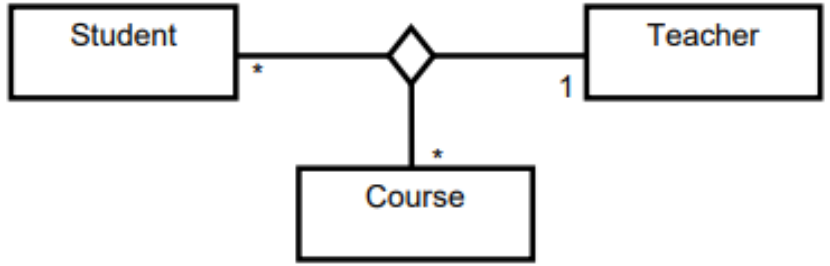
With respect to number of objects (cardinality)

Binary Association

It associates objects of exactly two classes; it is denoted by a line, or an arrow between the associated objects.

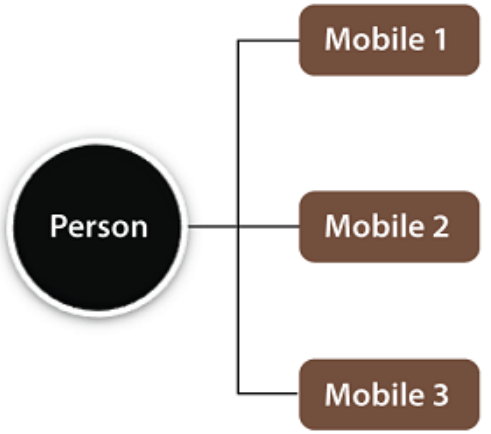


Ternary Association



Association Relationships Types

- One-to-One
  - One instance of a class is related to one instance of another class
  - **Example:** A person can have only one passport.
- One-to-Many
  - One instance of a class is related to many instances of another class
  - **Example:** A College can have many students
- Many-to-Many
  - Many instances of one class are related to many instances of another class.
  - **Example:** A single student can associate with multiple teachers, and multiple students can also be associated with a single teacher.



Aggregation(Has-A)

- The relationship between two classes called whole/part
- It is unidirectional relationship.
- Implementation:
  - One class has a reference to another class, but the second class can exist independently of the first class.
- Benefit: Code Reusability
- Examples
  - Student Has-A Address (Has-A relationship between student and address)
  - Staff Has-A Address (Has-A relationship between staff and address)
  - Department Has-A Teacher (Has-A relationship between department and teacher)

Aggregation(Has-A)

- Example:
  - Department(A) Has-A Teacher(B)
  - A owns B
  - The lifetime of object B does not depend on lifetime of object A
  - If there is no department, it does not mean teacher doesn’t exist.

Example

- Consider two classes Student class and Address class. Every student has an address so the relationship between student and address is a Has-A relationship. But if you consider its vice versa then it would not make any sense as an Address doesn't need to have a Student necessarily.

```
public class Address{
  private int streetNum;
  private String city;
  private String state;
  private String country;
  Address(int street, String c, String st, String coun){
     this.streetNum=street;
     this.city =c;
     this.state = st;
     this.country = coun;
  }
  public int getStreetNum(){
    return this.streetNum;
  }
  public String getCity(){
    return this.city;
  }
  public String getState(){
    return this.state;
  }
  public String getCountry(){
    return this.country;
  }
}
```

```
public class Student{
  private int rollNum;
  private String studentName;
  //Creating HAS-A relationship with Address class
  private Address studentAddr;
  Student(int roll, String name, Address addr){
     this.rollNum=roll;
     this.studentName=name;
     this.studentAddr = addr;
  }
  public int getRollNum(){
    return this.rollNum;
  }
  public String getStudentName(){
    return studentName;
  }
  public Address getAddress(){
    return this.studentAddr;
  }
   public int getStreetNum(){
    return studentAddr.getStreetNum();
  }
```

```
public String getCity(){
    return studentAddr.getCity();
  }
  public String getState(){
    return studentAddr.getState();
  }
  public String getCountry(){
    return studentAddr.getCountry();
  }
}
```

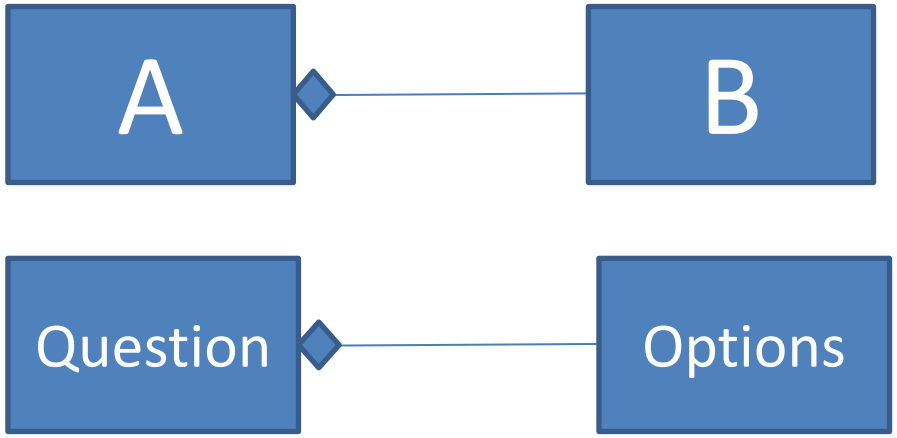
```
public class StudentAddressTest{
public static void main(String args[]){
    Address addr = new Address(55, "Islamabad", "Fedral", "Pakistan");
    Student ahmad = new Student(123, "Ahmad", addr);
    System.out.println(ahmad.getRollNum());
    System.out.println(ahmad.getStudentName());
    System.out.println(ahmad.getAddress().getStreetNum());
    System.out.println(ahmad.getCity());
    System.out.println(ahmad.getState());
    System.out.println(ahmad.getCountry());
  }
}
```

Composition

- Composition is a specialized form of aggregation.
- In composition, if the parent object is destroyed, then the child objects will not exist.
- Composition is a strong type of aggregation and is sometimes referred to as a “death” relationship
- Example:
  - House Has-A Room or a house may be composed of one or more rooms
  - If the house is destroyed, then all of the rooms that are part of the house are also destroyed

Composition

- In composition the life cycle of the part or child is controlled by the whole or parent that owns it
- Example
  - Question(A) Has-A Options(B)




```
public class House
{
  private Room room;
  public House()
  {
     room = new Room();
  }
}
```

```
public class Question{
    private String questionText;
    private Option option1, option2, option3;
    public Question(String questionText) {
        this.questionText = questionText;
        option1 = new Option("Option 1", false);
        option2 = new Option("Option 2", true);
        option3 = new Option("Option 3", false);
    }
    public String getQuestionText() {
        return questionText;
    }
    public String getOption1() {
        return option1.getOptionText();
    }
    public String getOption2() {
        return option2.getOptionText();
    }
    public String getOption3() {
        return option3.getOptionText();
    }
}
```

```
public class Option{
    private String optionText;
    private boolean isCorrect;

    public Option(String optionText, boolean isCorrect) {
        this.optionText = optionText;
        this.isCorrect = isCorrect;
    }

    public String getOptionText() {
        return optionText;
    }

    public boolean isCorrect() {
        return isCorrect;
    }

}
```

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


      Question question = new Question("Which option is correct?");

      System.out.println(question.getQuestionText());
      System.out.println(question.getOption1());
      System.out.println(question.getOption2());
      System.out.println(question.getOption3());
   }
}
```

```
Which option is correct?
Option 1
Option 2
Option 3
```

Lecture – 11 - 15

Inheritance

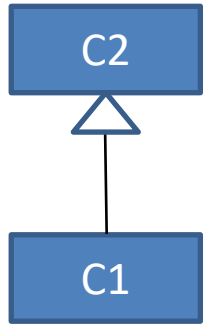
Introduction

- Inheritance is the process by which a new class is created from another class
  - The new class is called a ***derived class***
  - The original class is called the ***base class***
- A derived class automatically has
  - All the instance variables and methods that the base class has
  - And additional methods and/or instance variables as well
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.
- Advantage is the code reusability

Introduction

- Syntax

```
class Subclass-name extends Superclass-name {
  //methods and fields
}
```



Superclass/Parent Class or Base Class

A class C1 extended from another class C2 is called a subclass/Child class/Extended class, or a Derived class.

- The **extends keyword** indicates that you are making a new class that derives from an existing class.
- The meaning of "extends" is to increase the functionality.

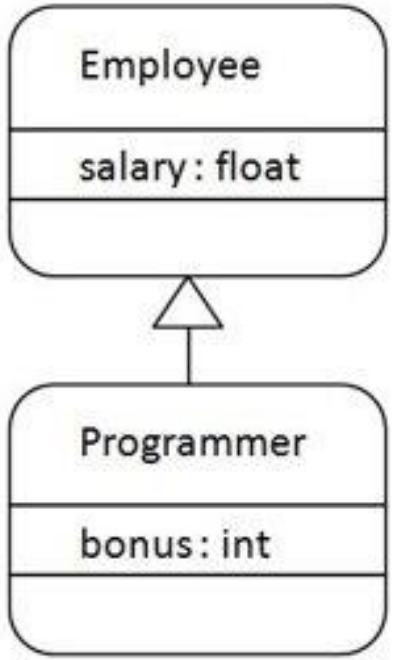
Examples

Superclass	Subclasses
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount

Example

```
class Employee{
  float salary=40000;
}
class Programmer extends Employee{
  int bonus=10000;
  public static void main(String args[]){
     Programmer p=new Programmer();
     System.out.println("Programmer salary is:"+p.salary);
     System.out.println("Bonus of Programmer is:"+p.bonus);
  }
}
```

```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

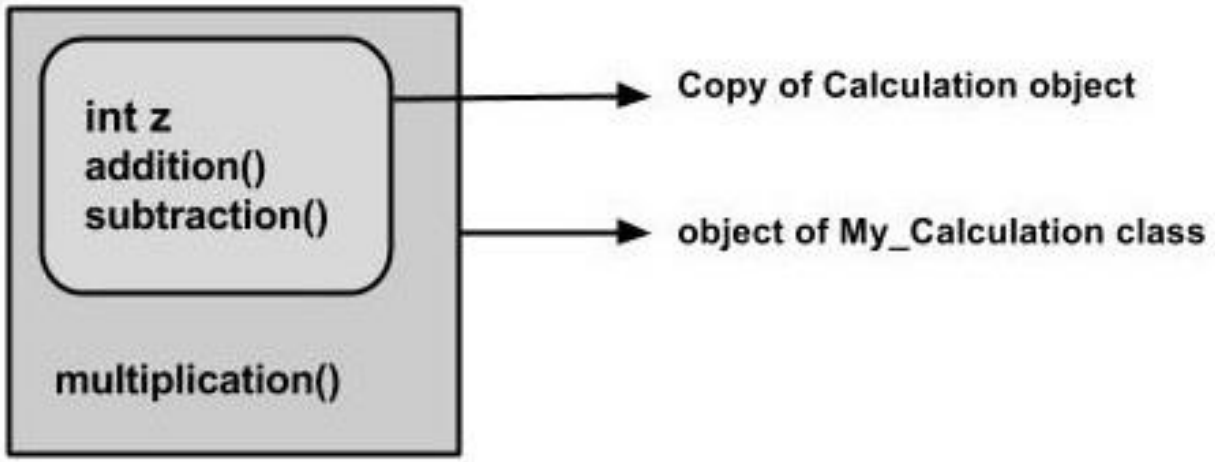


Relationship between two classes
Programmer IS-A Employee

```
public class Calculation {
  int z;
  public void addition(int x, int y) {
    z = x + y;
    System.out.println("The sum of the given numbers:"+z);
  }
  public void Subtraction(int x, int y) {
    z = x - y;
    System.out.println("The difference between the given
numbers:"+z);
  }
}
```

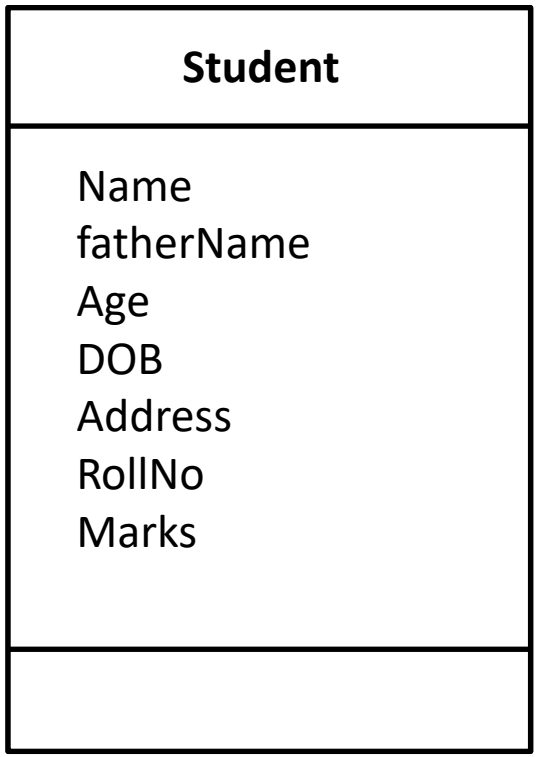
```
public class My_Calculation extends Calculation {
  public void multiplication(int x, int y) {
    z = x * y;
    System.out.println("The product of the given numbers:"+z);
  }
}
```

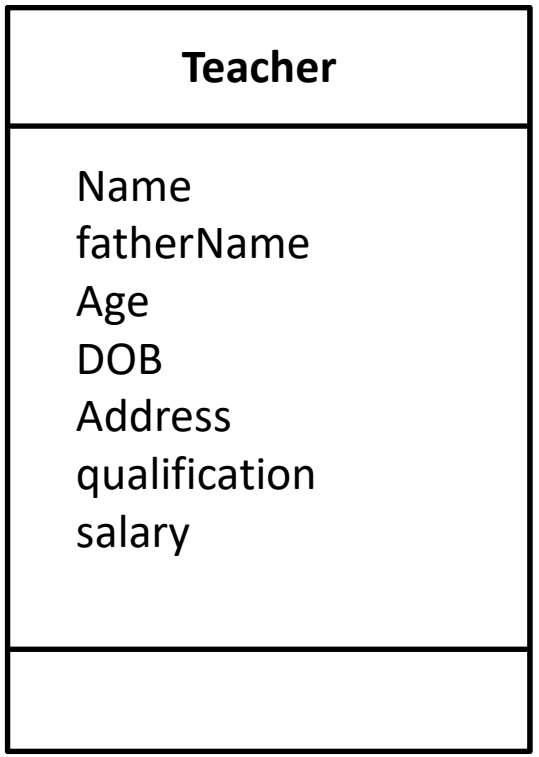
```
 public class Demo{
            public static void main(String args[]) {
      int a = 20, b = 10;
      My_Calculation demo = new My_Calculation();
      demo.addition(a, b);
      demo.Subtraction(a, b);
      demo.multiplication(a, b);
    }
  }
```

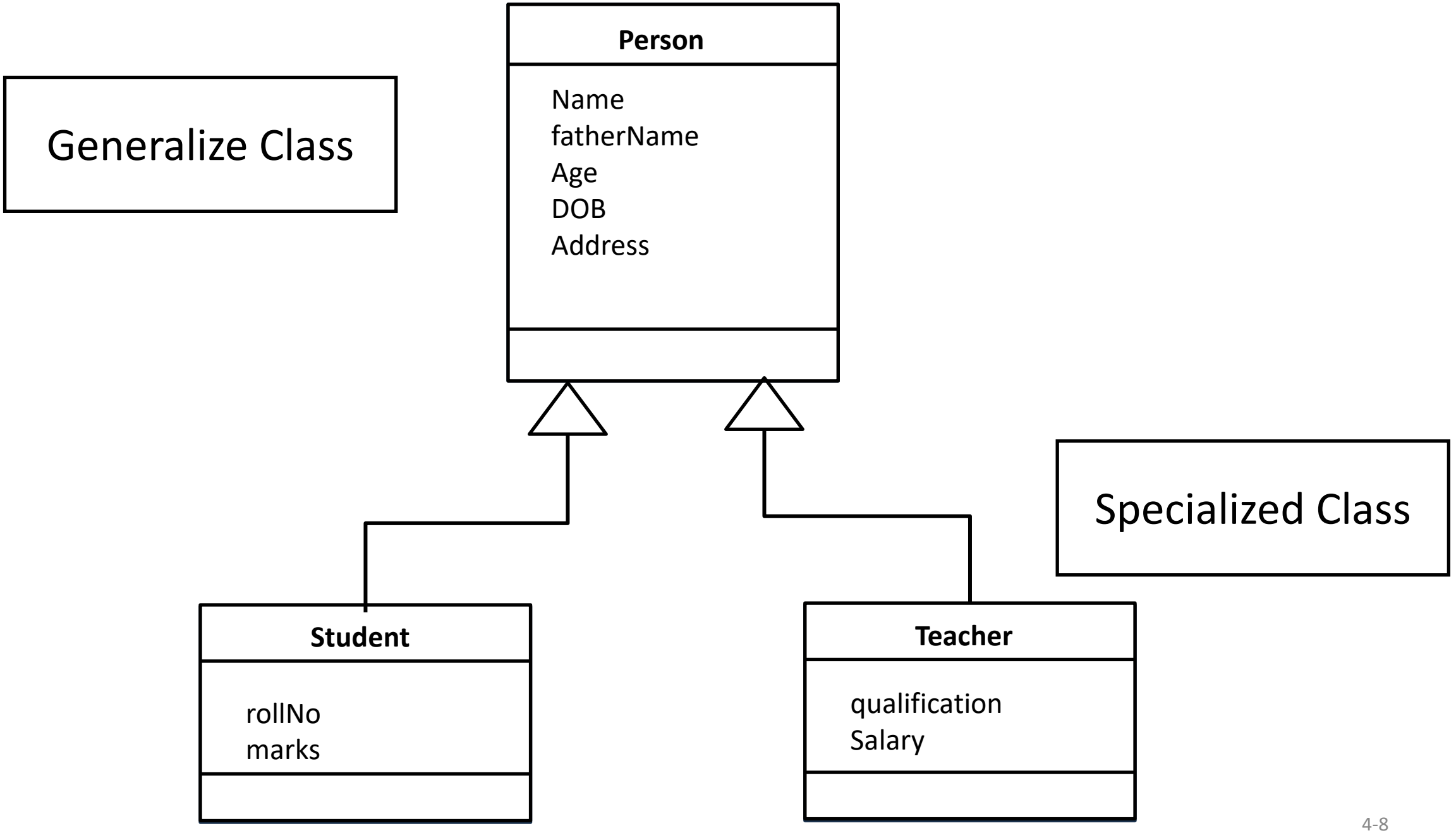


Introduction

- Generalization and Specialization
  - Inheritance enable you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses).



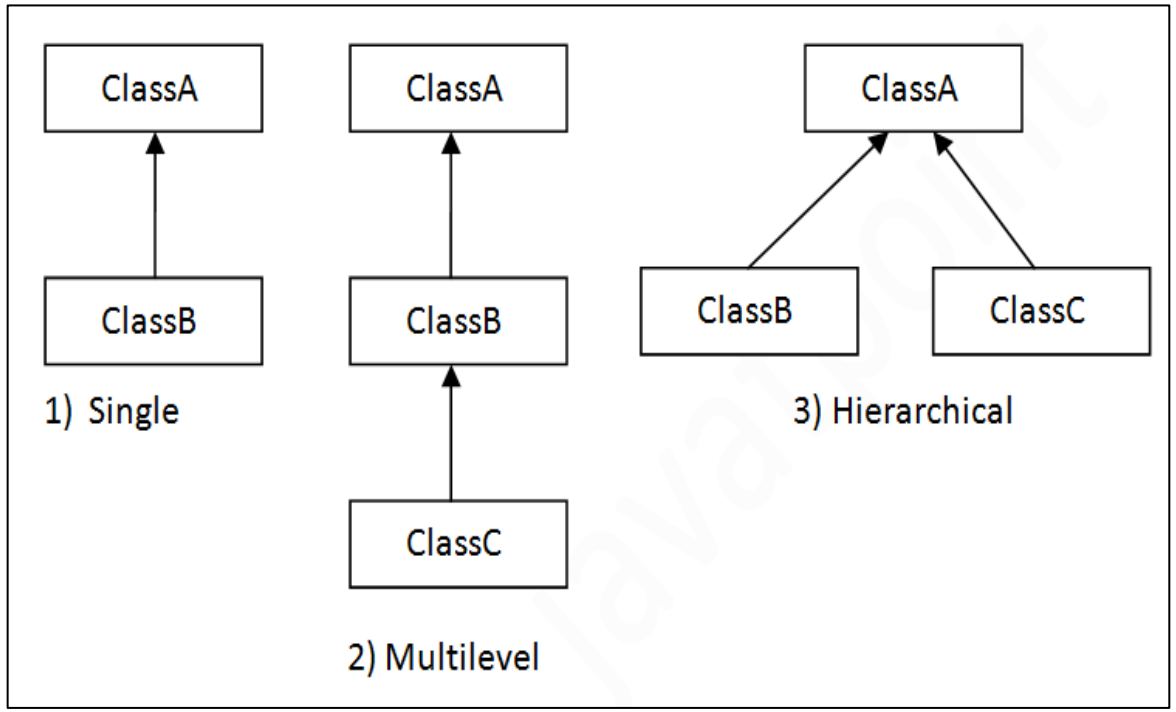


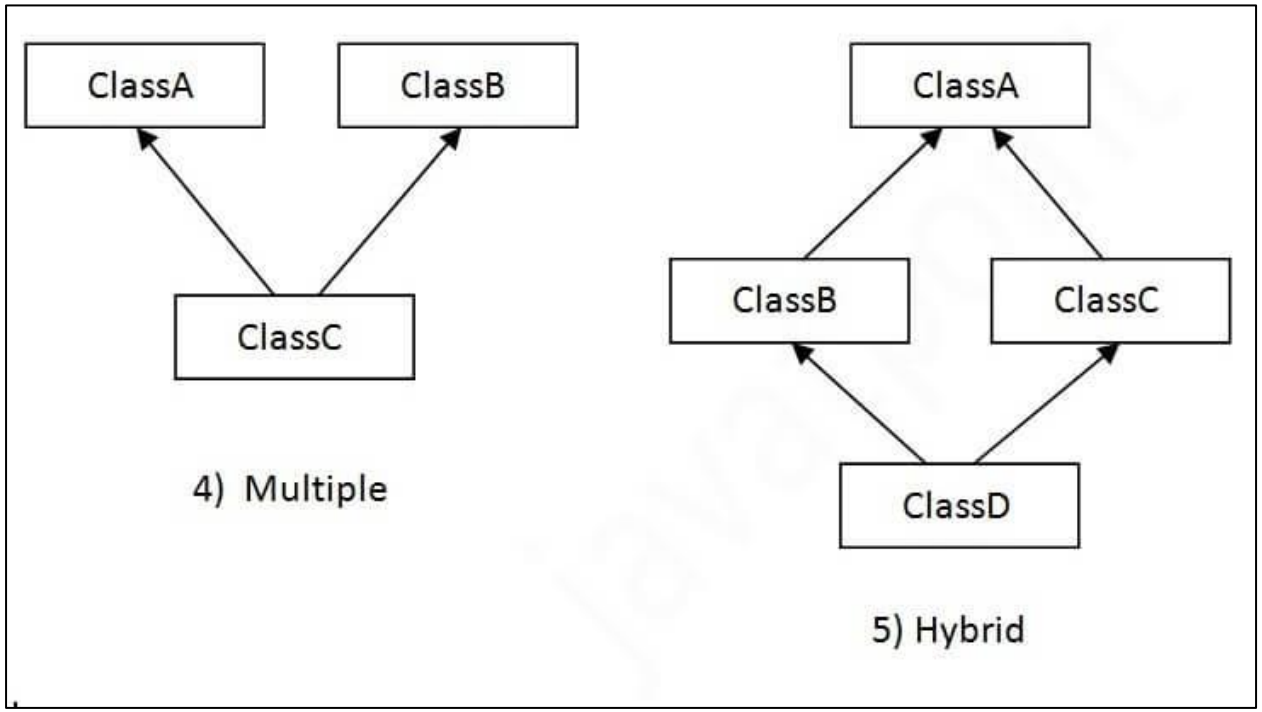

Generalize Class
Person
Name
fatherName
Age
DOB
Address
Specialized Class
Student
rollNo
marks
Teacher
qualification
Salary

What is Inherited in Subclass

- All non-private **fields**(Instance Variables) and **Methods** (declared with any access modifier `public`, `protected`, or `default`) of the superclass are inherited by the subclass
  - Subclass can override methods or hide fields with its own implementation
- Constructors are not inherited by subclasses
- A subclass can call a constructor of the superclass using the **super()** keyword

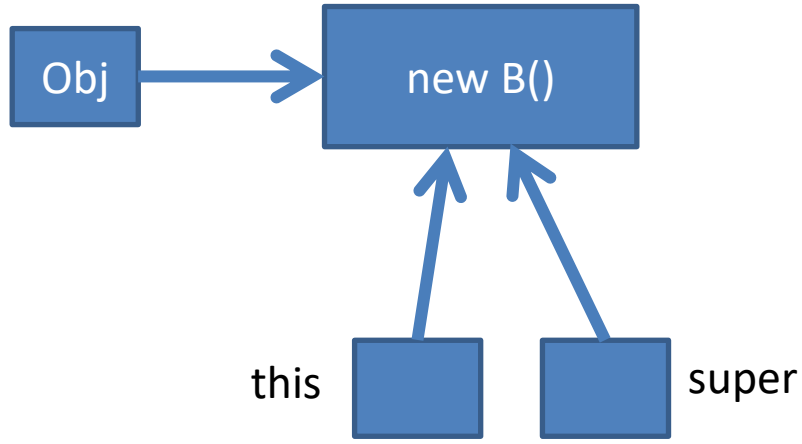
Types of inheritance in java





Super keyword

- In inheritance, subclass object, when call an instance member function of subclass only
  - The function contains two reference variables **this** and **super** referring to current object (object of subclass)
- The only difference in **this** and **super** is only **type**
  - ***This*** reference variable is of subclass type
  - ***Super*** reference variable is of superclass type



The super keyword

- **super** can be used to refer immediate parent class instance variable.

```
super.variable;
```

- **super** can be used to invoke immediate parent class method.

```
super.method();
```

- It is used to **invoke the superclass** constructor from subclass.

```
super(values);
```

Constructors in Inheritance

- Constructors are not inherited in inheritance
- Subclass constructors invokes constructor of the super class
- Implicit call and explicit call to the super class constructor
  - Implicit call – java automatically place call by placing super
  - Explicit call – use super() in subclass constructor and it must be the first line in the subclass constructor.

Constructors in Inheritance

- Scenarios
  - Implicit constructors in superclass and subclass (default constructor)
  - Implicit constructor in subclass and explicit constructor in superclass (no-arg constructor)
  - Implicit constructor in superclass and explicit constructor in subclass
  - Explicit constructor in superclass and subclass
  - For parameterized constructor in superclass, subclass must use super(x) in its constructor to call superclass constructor

Find Errors

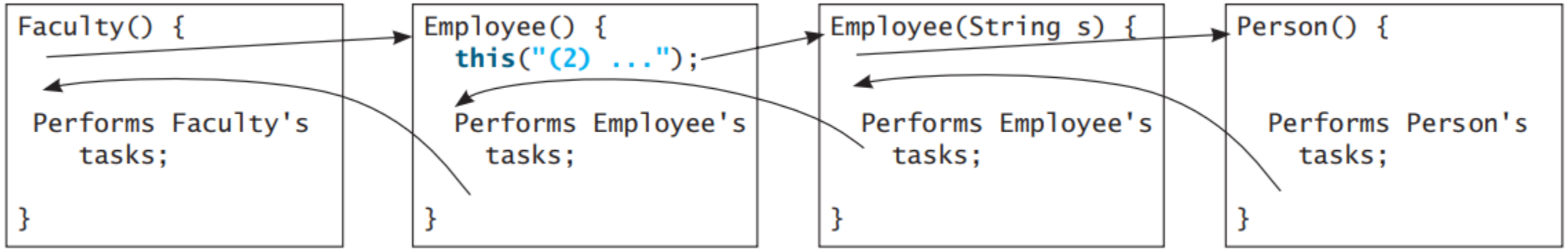
```
public class Apple extends Fruit{
}
class Fruit{
        public Fruit(String name){
                System.out.println("Fruit's constructor is invoked");
        }
}
```

Since `Apple` has no constructor, therefore no-arg constructor is defined implicitly. `Apple` is subclass of `Fruit`, `Apple` default constructor invokes `Fruit's` no-arg constructor. However `Fruit` does not have no-arg constructor, because it has an explicit constructor defined

You should provide a no-arg constructor for every class to make the class easy to extend and to avoid errors

Constructor Chaining

- Constructor can call other constructors of the same class or superclass
- Constructor call from a constructor must be the first step(call should appear in the first line)
- Such series of invocation of constructors is known as constructor chaining



```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("Faculty's no-arg constructor is invoked");
  }
}
class Employee extends Person {
  public Employee() {
     System.out.println("Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
  }
}
class Person {
  public Person() {
    System.out.println("Person's no-arg constructor is invoked");
  }
}
```

Overloading Methods

- If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but different signatures, then the method name is said to be overloaded
- Method overloading is a way to implement polymorphism

Example

```
class A{
        public void f1(int x){
                System.out.println("A");
        }
}
class B extends A{
        public void f1(int x, int y){
                System.out.println("B");
        }
}
public class Example{
        public static void main(String[] args){
                B obj = new B();
                obj.f1(3);
                obj.f1(3,4);
        }
}
```

Overriding Methods

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
- Provide the specific implementation of a method which is already provided by its superclass.
- Rules for Java Method Overriding
  - The method must have the same name as in the parent class
  - The method must have the same parameter as in the parent class.
  - There must be an IS-A relationship (inheritance).

Example

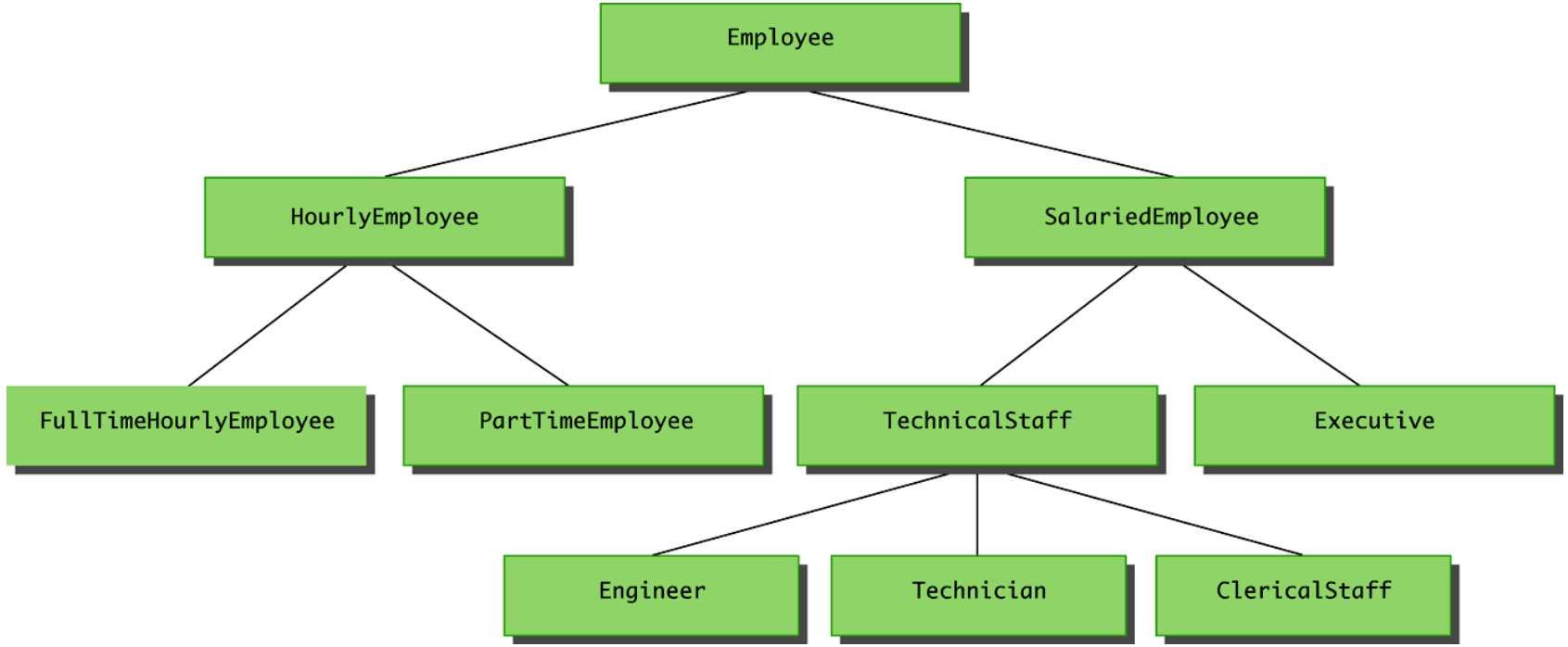
```
class A{
        public void f1(int x){
                System.out.println("Class A");
        }
}
class B extends A{
        public void f1(int x){
                System.out.println("Class B");
        }
}
public class ExampleOverRiding{
        public static void main(String[] args){
                B obj = new B();
                obj.f1(8);
        }
}
```

Overriding and Access-Modifiers

- The access modifier for an overriding method can allow more, but not less, access than the overridden method
- For example
  - A protected instance method in superclass can be made public, but not private, in the subclass.
  - Doing so will generate compile time error

Inheritance hierarchy for Employee

- Each arrow in the hierarchy represents an *is-a* relationship
- Not every class relationship is an inheritance relationship



Employee - Example

- Class **`Employee`** defines the instance variables **`name`** and **`hireDate`** in its class definition
- Class **`HourlyEmployee`** also has these instance variables, but they are not specified in its class definition
- Class **`HourlyEmployee`** has additional instance variables **`wageRate`** and **`hours`** that are specified in its class definition
- The class **`HourlyEmployee`** inherits the methods **`getName`**, **`getHireDate`**, **`setName`**, and **`setHireDate`** from the class **`Employee`**

Lecture – 16

Object Class, and toString() Method

Object Class

- Every class in Java is descended from the `java.lang.Object` class.
- If no inheritance is specified when a class is defined, the superclass of the class is `Object` by default.
- This means that all classes in Java are subclasses of Object, and inherit certain default behavior and methods from Object, such as the `"equals()", "toString()",` and `"hashCode()"` methods

```
Public class ClassName{
 …
}
```

Equivalent

```
Public class ClassName extends Object{
 …
}
```

toString() Method

- `toString()` method is instance method of `Object` Class
- Signature of `toString(): public String toString()`
- Invoking `toString()` on an object returns a string that describes the object

```
Loan loan = new Loan();
System.out.println(loan.toString());
```

```
Loan@15037e5
```

- This output is not informative or helpful.
- Usually we override the toString() method
  - It returns a descriptive string representation of the object.

Example

```
public class Shape {
   private String color;
   private boolean filled;

   public Shape() {
      color = "red";
      filled = true;
   }

   public Shape(String color, boolean filled) {
      this.color = color;
      this.filled = filled;
   }
. . .
 @Override
   public String toString() {
      return "Shape[color=" + color + ", filled=" + filled + "]";
   }
```

```
public class Demo{
        public static void main(String[] args){
                Shape s1 = new Shape("red", false);
                System.out.println(s1.toString());
                System.out.println(s1 );
        }
}
```

```
Shape[color=red, filled=false]
Shape[color=red, filled=false]
```