

University of Salahaddin-Hawler  
College of Engineering  
Software Engineering Department  
High Diploma



# Advanced Object Oriented Programming

Lecturer

Kanar Shukr Muhamad

2023-2024

# Objectives

- Understand major concepts of object-oriented programming
- Knowledge and skills in OO design and program development.
- Experience in Java programming and program development within an integrated development environment
- Certain skills in internet and windows programming and using graphical user interface.
- Finishing this course successfully learns to put object oriented methodologies under practice.

Subject Title	Date		Subject Title	Date	Week
Coursebook overview and Introduction				22-Oct	1
Recursion	29-Oct		Methods, and method overloading	29-Oct	2
Initialization & Cleanup	5-Nov		Java Classes, objects and access modifiers	5-Nov	3
Java Image Classes	12-Nov		Object Oriented Programming: Encapsulation	12-Nov	4
Package	19-Nov		Object Oriented Programming: Inheritance	19-Nov	5
Exception Handling	26-Nov		Object Oriented Programming: Polymorphism	26-Nov	6
Algorithm Design	3-Dec		Inner Classes	3-Dec	7
Abstract Class and Interface	10-Dec		MultiThreading	10-Dec	8
Java Two Dimensional Arrays	17-Dec		Accessing Databases with JDBC	17-Dec	9
Generic in Java	24-Dec		Network Programming	24-Dec	10
Generic Collection Class				7-Jan	11
Graphical User Interface (GUI) and event handling				14-Jan	12

# References

- Paul Deitel, Harvey Deitel., " Java How to Program ", 10<sup>th</sup> Edition, ISBN 978-0132575669.
- Ralph Morelli, Ralph Walde, “Java, Java, Java™: Object Oriented Problem Solving”, 3<sup>rd</sup> Edition, June 25, 2017
- Herbert Schildt , “Java™ : The Complete Reference”, 7<sup>th</sup> Edition, ISBN: 978-0-07-163177-8, 2007.
- B. Eckel , “Thinking in Java”, 4<sup>th</sup> ed., Prentice Hall, 2006

# Object Oriented Programming

- ❑ Object-oriented programming is a new way of programming.
- ❑ Since its early days, programming has been practiced using a number of various methodologies.
- ❑ At each new stage, a new approach was created to make programming easier and help the programmer handle more complex programs.
- ❑ Java is a pure object-oriented programming language.
- ❑ Structured programming relies on control structures, code blocks, procedures or functions and facilitates recursion.

# Object Oriented Programming

- ❑ But after a certain point even structured programming becomes very hard to follow.
- ❑ To write larger and more complex programs, a new programming approach was invented: object-oriented programming or OOP for short.
- ❑ Object-oriented programming combines the best features of structured programming with some new powerful concepts that allows writing more complex and more organized programs.

# Object Oriented Programming 2

- ❑ All OOP languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, polymorphism and reusability.
- ❑ Any programming language that supports these three concepts is said to be an OO programming language like C++, Java, Smalltalk, c#....
- ❑ Object-oriented programming encourages programmers to break problems into related subgroups.

# Object Oriented Programming 3

- ❑ Each subgroup becomes a self-contained object with its own instructions and data.
- ❑ An object is similar to an ordinary variable but with its own member methods.
- ❑ Each object is a self-contained entity, it is an autonomous entity that can be used and reused in other programs.
- ❑ This also allows for composition of objects to create more complex programs.



# Object Oriented Programming 4

- ❑ In procedural programming languages, programming tends to be **action-oriented**, whereas in OO programming languages such as Java and Smalltalk, programming **object-oriented**.
- ❑ In action-oriented programming, the focus is on actions or functions; in OO programming, the focus is on objects.

# Object Oriented Programming 6

□ Suppose `ob` is an object:

```
print(ob) ; //focus is function/action
```

```
ob.print() ; //focus is object
```

# Difference between Procedural Programming and OOP

## OOP

- Object-oriented Programming is a programming language that uses classes and objects to create models based on the real world environment. In OOPs it makes it easy to maintain and modify existing code as new objects are created inheriting characteristics from existing ones.

## Procedural Oriented Programming

- On other hand Procedural Oriented Programming is a programming language that follows a step-by-step approach to break down a task into a collection of variables and routines (or subroutines) through a sequence of instructions

# Difference between Procedural Programming and OOP

OOP	Procedural Oriented Programming
<p>Modifying and updating the code is easier. due to modularity in its programs is less complex and hence new data objects can be created easily from existing objects making object-oriented programs easy to modify</p>	<p>Modifying the code is difficult as compared to OOPs, there's no simple process to add data in POP at least not without revising the whole program.</p>

# Difference between Procedural Programming and OOP

OOP	Procedural Oriented Programming
3- Due to abstraction in OOPs data hiding is possible and hence it is more secure than POP.	3- On other hand POP is less secure as compare to OOPs.

# What is Object Oriented Programming (OOPs)?

- Object Oriented Programming (OOP) is a programming paradigm where the complete software operates as a bunch of objects talking to each other.

# What is a Class?

- A **class** is a building block of Object Oriented Programs. It is a user-defined data type that contains the data members and member methods that operate on the data members.
- ❑ Most things in the world are classified: a class of students, a class of fish, a class of birds, a class of objects.
- ❑ A class is a collection of **data** (stored in named fields) and **code** (organized into named methods that operate on that data).

# What is an Object

- An **object** is an instance of a class. Data members and methods of a class are used an object (or instance) of the class that have a state and behavior.
- **Q1/** The code below shows is an example of how an object of a class is created.



# What are the main features of OOPs?

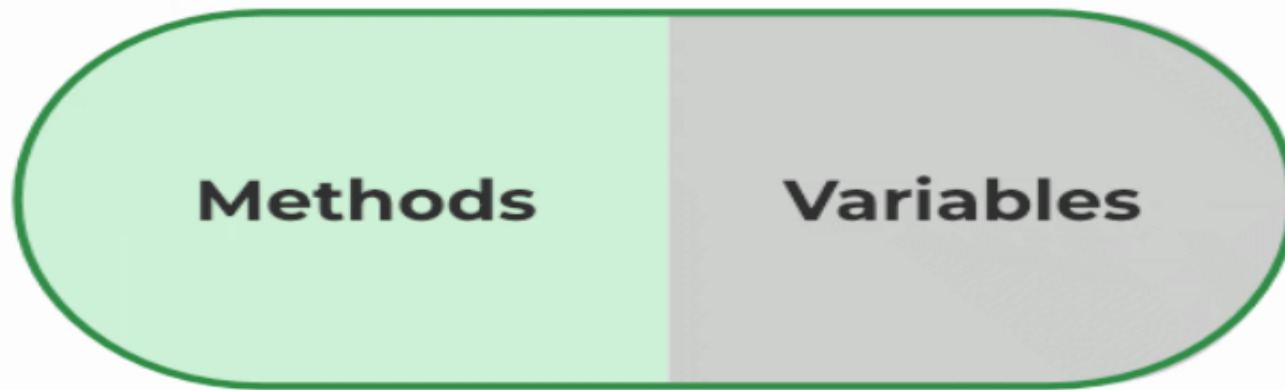
- The main feature of the OOPs are as follows:
  1. Encapsulation
  2. Polymorphism
  3. Data Abstraction
  4. Inheritance

# What is Encapsulation?

- Encapsulation is the binding of data and methods that the sensitive data is hidden from the users. It is implemented as:
  - 1. Data hiding:** A language feature to restrict access to members of an object. For example, private and protected members.
  - 2. Bundling of data and methods together:** Data and methods that operate on that data are bundled together they are wrapped into a single unit known as a class.

# What is Encapsulation?

## Encapsulation



## Class

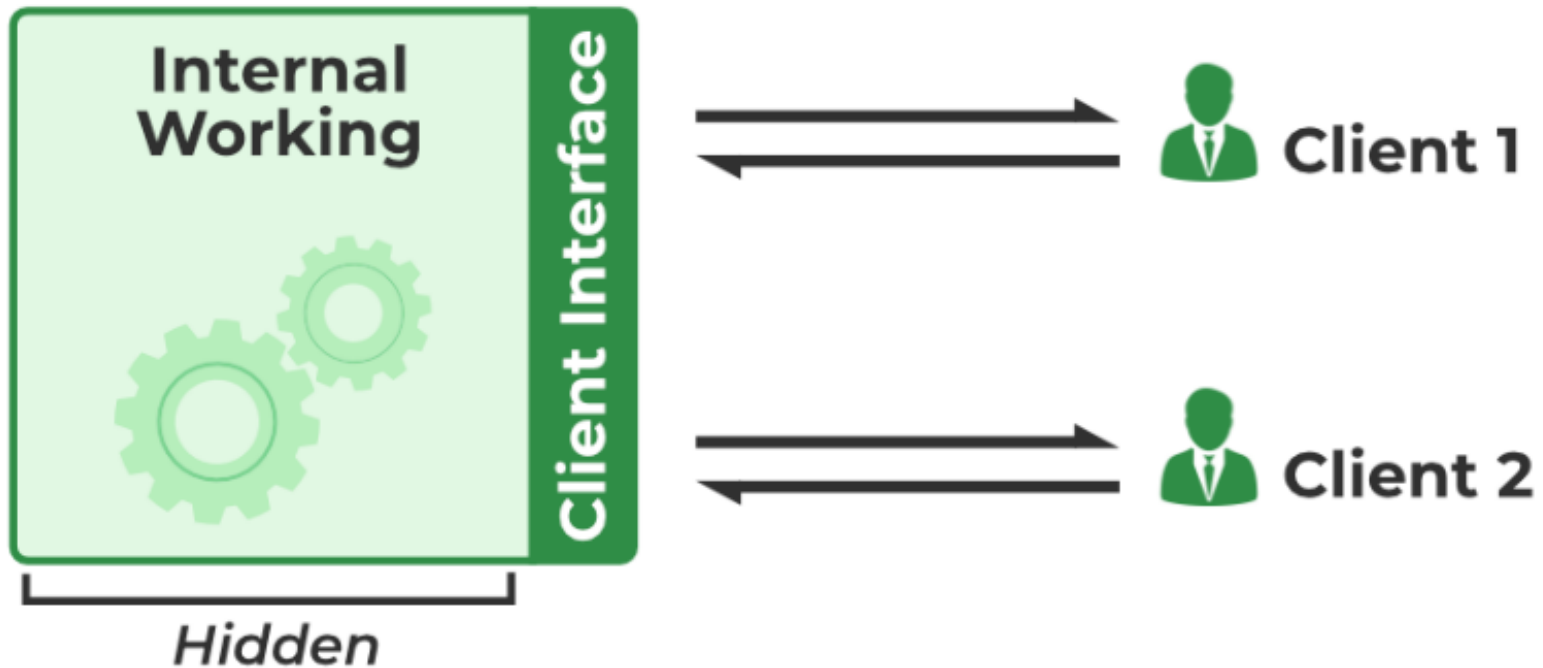
**Q2/** The code below shows is an example demonstration of encapsulation, It has a private data member and getter and setter methods.

# What is Abstraction?

- Abstraction is similar to data encapsulation and is very important in OOP.
- It means showing only the necessary information and hiding the other irrelevant information from the user.
- Abstraction is implemented using classes and interfaces.

# What is Abstraction?

## Abstraction



# What are the main features of OOPs?

**Q3/** What is an abstract class and implement abstraction through abstract class.

# What is Polymorphism?

- The word “**Polymorphism**” means having many forms. It is the property of some code to behave differently for different contexts.
- Polymorphism can be classified into two types based on the time when the call to the object or function is resolved.
  - ✓ Compile Time Polymorphism
  - ✓ B. Runtime Polymorphism

# What is Polymorphism?

**Q4/** Write a program to display method overload.

**Q5/** Write a program to display method overriding.



# What is Inheritance? What is its purpose?

- The idea of inheritance is simple, a class is derived from another class and uses data and implementation of that other class.
- The main purpose of Inheritance is to increase code reusability.
- It is also used to achieve Runtime Polymorphism

# What are the main features of OOPs?

Q6/ create classes to demonstrate inheritance.

# What are access specifiers in OOPs?

- Access specifiers (**Private**, **Public**, and **Protected** ) are special types of keywords that are used to specify or control the accessibility of entities like classes, methods, members, and so on.
- The key components of OOP are largely achieved because of these access specifiers

# What are the advantages and disadvantages of OOPs?

Advantages of OOPs	Disadvantages of OOPs
OOPs provides enhanced code reusability.	The programmer should be well-skilled and should have excellent thinking in terms of objects as everything is treated as an object in OOPs.
The code is easier to maintain and update.	Proper planning is required because OOPs is a little bit tricky.
It provides better data security by restricting data access and avoiding unnecessary exposure.	OOPs concept is not suitable for all kinds of problems.
Fast to implement and easy to redesign resulting in minimizing the complexity of an overall program.	The length of the programs is much larger in comparison to the procedural approach.

# Encapsulation?

- Encapsulation is the binding of data and methods that the sensitive data is hidden from the users.
- Encapsulation is the binding together of code and data and keeping both safe from outside interference and misuse.
- ❑ When code and data are bound together like this an object is created.
- ❑ Inside an object, code and data may be **private** or **public** to that object.

# Encapsulation?

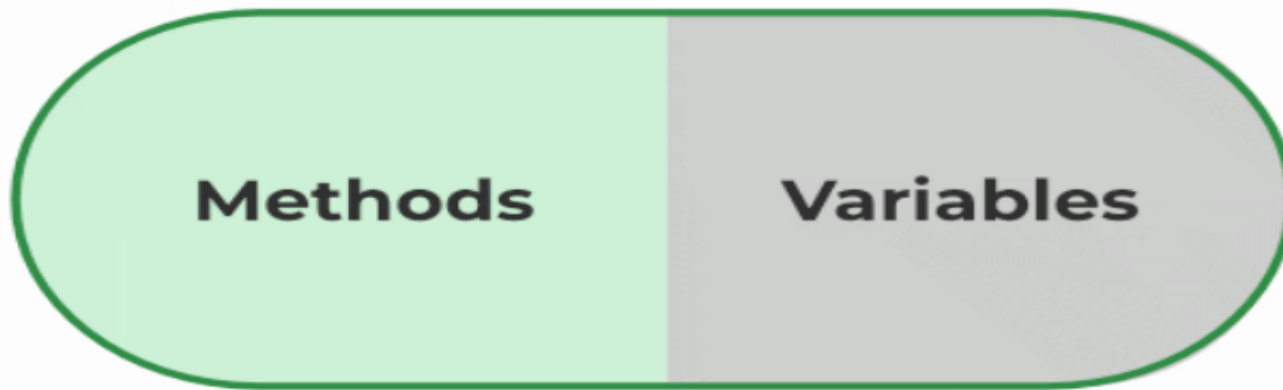
- ❑ Private data or code is known and accessible to other parts of the object only.
- ❑ The object dictates or determines how its private data and methods (code) should be accessed and used.
- ❑ Better control of class attributes and method, Increased security of data.

# What is Encapsulation?

- It is implemented as:
  - 1. Data hiding:** A language feature to restrict access to members of an object. For example, private and protected members.
  - 2. Bundling of data and methods together:** Data and methods that operate on that data are bundled together they are wrapped into a single unit known as a class.

# What is Encapsulation?

## Encapsulation



**Class**



# Polymorphism

- ❑ Polymorphism is the mechanism which allows one name to be used for two or more related but technically different purposes.
- ❑ Earlier on we saw overloading of methods which is an example of polymorphism. The general concept of polymorphism is **“one interface, multiple methods”**.
- ❑ In other words, you use the same method or mechanism to perform a group of related tasks, it's helps reduce complexity.

# Polymorphism

- ❑ Another type of polymorphism is achieved using inheritance, this is the more interesting type and we will look at it a bit later.

# Inheritance

- ❑ Inheritance is another important feature of OOP, that allows an object inherit the properties of another object.
- ❑ The object that inherits another object acquires all the properties of the parent object and can add its own extra features specific only to itself.
- ❑ Inheritance provides for hierarchical classification which is very important in making information manageable.

# Inheritance

- ❑ For example, a square is a kind of rectangle; in turn, a rectangle is a kind of closed geometric shape; in turn, a closed geometric shape is a kind of geometric shape.
- ❑ In each case, the child object inherits all the properties of the parent object and adds some extra features specific to itself.

# Reusability

- ❑ Once a class has been written, created and debugged, it can be distributed to other programmers for use in their own program.
- ❑ This is called reusability, In OOP, however, inheritance provides an important extension to the idea of reusability.
- ❑ A programmer can use an existing class and without modifying it, add additional features to it.

# Objects & Classes

- ❑ A class is like a factory for creating objects.
- ❑ The fields and methods of a class are called members of the class, can be of two types: **static** or class members associated with the class itself, that there is only a single copy of it. While **instance** members associated with individual instances of the class (i.e., with objects):
  - ❑ The static modifier says that the field is a class field.
  - ❑ The final modifier says that this field does not change.
  - ❑ The public modifier says that this field can be used by any code; it's a visibility modifier which we will cover in more detail later.

# Objects & Classes 2

```
public class Circle
{//a class/static field and method
    public static final double PI=3.14159;
    public static double radiansToDegrees(double rads)
    {    return rads * 180 / PI;    }
    private double r; //an instance field
//two instance methods
    public double area()
    {    return PI * r * r;    }
    public double circumference()
    {    return 2 * PI * r;    }
}
```

# Objects & Classes 3

- ❑ Class methods and fields are declared with the `static` modifier, they can be accessed inside the class by writing its name only `PI`, and to access it from outside the class you must write `Circle.PI`.
- ❑ A class field is similar to a global variable; it is accessible from all parts and methods of that class.
- ❑ Any field declared without the `static` modifier is an instance field:

**`public double r;`**



# Objects & Classes 4

- ❑ Instance fields are associated with instances or objects of the class; each instance of the class has its own copy of an instance field.
- ❑ Any method not declared with the static modifier is an instance, it's operate on instances of the class, not on the class itself, they have access to both class fields and class methods also.
- ❑ They know which object they operate on because they are passed an implicit reference to the object they operate on; this reference is called **this**.

# Constructor

- ❑ A constructor is a special method that is automatically called every time you create a new object, it's used to initialize objects.
- ❑ A constructor method has the same name as the class name and returns no values.
- ❑ To create and initialize an object: **Circle c=new Circle();**
- ❑ The operator new creates a new object of type Circle.
- ❑ In the class Circle defined earlier, no constructors were written; Java provided a default constructor that takes no parameters and performs no initialization.
- ❑ It is always better to specify a constructor for every new class you define to specify how a new object of that class would be initialized:

```
public Circle(double r) {this.r=r;}
```

# Constructor 2

- ❑ To create an instance/object of type Circle:

`Circle c=new Circle(5.0);`**//object creation & initialization on one line**

- ❑ **Constructor syntactic and operations:**

- ✓ The constructor name is always the same as the class name
- ✓ The constructor has no return values but may take parameters
- ✓ The constructor should perform initialization of the new object immediately upon creation.

- ❑ To provide flexibility in initializing a new object, often multiple constructors are defined:

```
public Circle() { r=1.0;} public Circle(double r) { this.r=r;}
```

The two constructors must have different parameter lists or signatures (method overloading).

# Constructor 3

- ❑ One important thing about having multiple constructors, if you need a constructor, make sure you have defined a default constructor.
- ❑ One of the uses of the **this** keyword is to invoke a constructor from within another constructor:

```
public Circle(double r) { this.r=r;}
```

```
public Circle() { this(1.0);}
```

- ❑ Instance **fields** and class **fields** are initialized by default: Variables of types byte, short, int, long, float, and double are initialized to 0, variables of type boolean are initialized to false , variables of type char are intialized to space( ), and reference-type variables are initialized to null. But local variables (declared inside methods) are not initialized by default. You should initialize them before use.

# Set and Get Methods

- ❑ As you know, the class's private fields can be manipulated only by methods of that class.
- ❑ **Set and Get methods vs. public data**
- ❑ It would seem that providing set and get capabilities is essentially the same as making the variables public. This is subtlety of Java that makes the language so desirable for software engineering.
- ❑ A public instance variable can be read or written by any method that has a reference to an object that contains the instance variable.
- ❑ If a variable is declared private, a public get method certainly allows other methods to access the variable, but the get method can control how other methods can access the variable.
- ❑ A public set method can- and should- carefully scrutinize attempts to modify the variable's value to ensure that the new value is appropriate for that data item.

# Garbage Collection

- ❑ In Java, memory occupied by an object is automatically reclaimed when the object is no longer needed. This is achieved by a process called **Garbage Collection**.
- ❑ The programmer does not have to worry about releasing or reclaiming memory used by object, this greatly reduces bugs and helps programmers be more productive.
- ❑ The Java interpreter knows which objects it has allocated, so the interpreter can determine which objects are no longer referenced by any variable and it then destroys them.
- ❑ The Java garbage collector runs as a low-priority thread, so it does most of its work when nothing else is going on.
- ❑ The only time that it must run even when some high-priority thread is going on is when available memory is dangerously low, but this doesn't happen often because the low-priority thread is running in the background and cleans unused objects.

# Finalizers

- ❑ Constructors are used to create objects: obtain memory, obtain resources, initialize object data... **Finalizers** are used to return allocated resources back to the system such as file, print and network connections.
- ❑ A class's finalizer is called just before the garbage collector destroys the object. It always has the name `finalize`, returns no values, has return type `void` and takes no parameters.
- ❑ If you don't define a `finalize` method for your class, a default one is created that does nothing.
- ❑ Finalizers are a bit similar to C++'s destructor functions which are used to return resources to the system.
- ❑ Finalizers are not as useful and necessary as C++'s destructors and are not often used in normal Java programming.

# Finalizers 2

**E.x3/** The following is an example demonstrating how finalizers are used:

```
import javax.swing.*;
public class EmployeeTest
{
    public static void main(String[] args)
    {
        Employee e= new Employee("X", "YZ");
        e=null;          //mark for garbage collection
        System.gc();    //suggest that GC be called
    }
}
```



# Finalizers 3

```
class Employee
{
    private String fName, lName;
    public Employee(String fName, String lName)
    {
        this.fName=fName; this.lName=lName;
        System.out.println("Constructor:" + fName+ " "+lName);
    }
    protected void finalize()
    { System.out.println("Finalizer Called"); }
}
```

- ❑ Finalizer methods are usually declared as protected so that subclasses can directly access and run them.

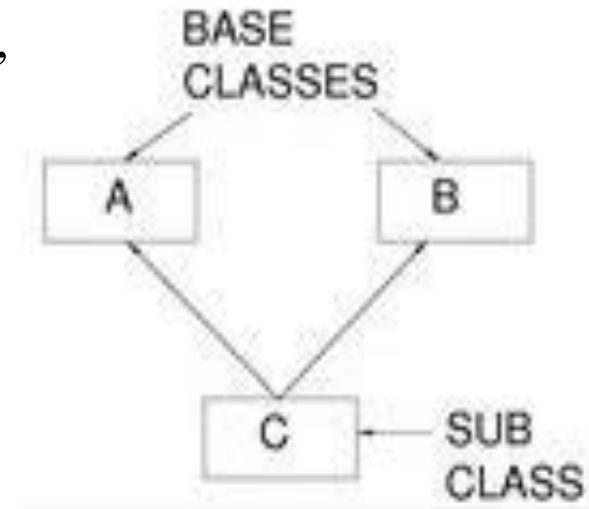
❑ The output of this program is the message: Constructor: xyz

❑ Finalizer Called“”.

# Inheritance

- ❑ Inheritance allows new classes to be created by reusing existing classes, thus saving time in software development.
- ❑ New classes acquire proven and debugged properties of existing classes.
- ❑ In Java, the keyword **extends** is used to inherit a new class from an existing class:  

```
class Child extends Parent {...}
```
- ❑ The new class Child is the subclass and the Parent is the superclass.
- ❑ Unlike C++, Java does not support multiple inheritance, but it supports interfaces which allow Java achieve many of the advantages of multiple inheritance without the associated problems.



# Inheritance 2

- ❑ Every object of the subclass is also an object of the super-class but not the other way round.
- ❑ Subclass methods and methods of other classes in the same package as the superclass can directly access protected superclass members.
- ❑ Every class in Java must inherit from a superclass; if a new class does not explicitly extend another class, Java implicitly uses the `Object` class as the superclass for the new class `Class Object` provides a set of methods that can be used with any object of any class.
- ❑ Consider the following example which is taken from the textbook:

# Inheritance 3

```
class Point{  
    private int x, y;  
    public Point() {          setPoint(0, 0);    }  
    public Point(int a, int b) {setPoint(a, b);}  
    public void setPoint(int a, int b)  
    {    x=a; y=b;    }  
    public int getX() { return x;}  
    public int getY() { return y;}  
    public String toString()  
    {    return "[" + x + ", " + y + "]; }  
} //end class point
```

# Inheritance 4

```
public class Circle extends Point
{
    protected double radius;
    public Circle() { setRadius(0); }
    public Circle(double r, int a, int b)
    {    super(a, b); setRadius(r);    }
    public void setRadius(double r)
    {radius= (r >=0.0 ? r : 0.0); }//see slide note
    public double getRadius() { return radius; }
    public double area()
    {return Math.PI * radius * radius; }
    public String toString()
    {
        return "Center= " + "["+ getX() + ", " +
getY() +";          Radius= " + radius; }
}
```

# Inheritance 5 (Note)

**if** ( $a > b$ )

$\text{max} = a;$

**else**

$\text{max} = b;$

- ❑ You can rewrite the above example in a single line like this:

**$\text{max} = (a > b ? a : b);$**

- ❑ Is an expression which returns one of two values, **a** or **b**.
- ❑ The condition, (**a > b**) is tested, if it is true the first value **a** is returned, if it is false, the second value **b** is returned.

# Inheritance 6

```
import java.text.DecimalFormat;
import javax.swing.JOptionPane;

public class InheritanceTest
{
    public static void main(String[] args)
    {
        Point pointRef, p; Circle circleRef, c;
        String output;

        p=new Point(30, 50);
        c=new Circle(2.7, 120, 89);
```

# Inheritance 7

```
output="Point p: " + p.toString() +
"\nCircle c: " +c.toString();
pointRef=c;//since a circle is-a point
output+="\nCircle c (via pointRef): " +
pointRef.toString();
circleRef=(Circle) pointRef; //downcast
output+="\nCircle c (via circleRef): " +
circleRef.toString();
DecimalFormat precision2=new DecimalFormat("0.00");
output+="\nArea of c (via circleRef): " +
precision2.format(circleRef.area());
```



# Inheritance 8

```
if (p instanceof Circle)
{
    circleRef=(Circle) p;
    output+="\nCast Successful";
}
else
    output+="\np does not refer to a Circle";
JOptionPane.showMessageDialog(null, output,
    "Demonstrating the \"is-a \" relationship",
    JOptionPane.INFORMATION_MESSAGE);
System.exit(0);
```

# Inheritance 9

- ❑ In this example, class Circle inherits from class Point and adds members specific to itself:
  - ✓ Circle overrides Point's toString method (**polymorphism**)
  - ✓ Point and Circle both have a default constructor as well as a parameterized constructor
  - ✓ Subclass Circle needs to call superclass Point's parameterized constructor using super along with any required arguments, and this statement must come before any other statements
  - ✓ Default constructors are invoked automatically.
  - ✓ Superclass objects or references can be used to refer to subclass objects because of the **is-a relationship**, hence the statement **pointRef=c;**

# Inheritance 10

- ✓ Explicit casting is needed to make a subclass (`Circle`) object to refer to a superclass (`Point`) object, hence the statement: **`circleRef= (Circle) pointRef;`**
- ✓ Attempting to cast a `Point` object to a `Circle` object is an error, so the statement: **`circleRef= (Circle) p;`** is illegal because `p` refers to a point object.
- ✓ The operator `instanceof` is used to check whether the object to which it refers is a `Circle`.
- ✓ Superclass constructors are not inherited; subclass constructors can call superclass constructors using the `super` reference.

# Inheritance 11

- ✓ If a class defines a finalize method, any subclass finalize method should call the superclass finalize method as its last action.

**E.X5/** The following simplified example illustrates the order in which constructor and finalize methods are called:

```
class Point {  
  
private int x, y;  
  
public Point()  
  
{  
    System.out.println("Point Cons: " + this);  
}
```

# Inheritance 12

```
public Point(int a, int b)
{
    x=a;  y=b;
    System.out.println("Point Cons" + this);
}

protected void finalize()
{ System.out.println("Point finalize: " + this); }

public String toString()
{
    return "[" + x + ", " + y + "]";
}
}
```

# Inheritance 13

```
class Circle extends Point
{
    protected double radius;
    public Circle()
    {
        System.out.println("Circle Cons: " + this);
    }
    public Circle(double r, int a, int b)
    {
        super(a, b);radius=r;
        System.out.println("Circle Cons: " + this);
    }
}
```

# Inheritance 14

```
public String toString()
{
    return "Center= "+super.toString()+ "
    Radius= " + radius;
}
protected void finalize()
{
    System.out.println("Circle finalize:"+this);
    super.finalize();
}
```

# Inheritance 15

- ❑ The keyword `super` can be used to access parent class members

```
public class TestI
{
    public static void main(String[] args)
    {
        Circle c1;
        c1=new Circle(4.5,72,29);
        c1=null;
        System.gc();
    }
}
```



# Inheritance 16

- ❑ The output is:

```
Point Cons Center=[72, 29]; Radius=0.0
```

```
Circle Cons: Center=[72, 29]; Radius=4.5
```

```
Circle finalize: Center=[72, 29]; Radius=4.5
```

```
Point finalize: Center=[72, 29]; Radius=4.5
```

- ❑ The class Object has a number of methods which are inherited by any created class. The method `toString()` is one such method which returns a textual representation of the object.

# Inheritance 17

- ❑ You should always try to override this method as in this example.
- ❑ Outputting the **this** reference of an object, invokes the `toString()` method.
- ❑ In is example, after we finish with `c1` object, we set them to null to indicate that they are no longer needed and then ask that the system's garbage collector be called with the call `System.gc()`.
- ❑ Java guarantees that before the garbage collector runs to reclaim the space for each object, the `finalize` method for each object is called.

# Polymorphism

- ❑ You can have an inheritance hierarchy where a number of classes extend from a parent class:

**Person ---> Students ---> Undergraduate ---> ...**

- ❑ This is called an inheritance hierarchy.
- ❑ In OOP, when you invoke a method on an object, you send that object a message.
- ❑ When a method is applied to an object of class in an inheritance hierarchy the following occurs:
  - ✓ The class (subclass) checks whether or not it has a method with that name and with exactly the same parameters. If so, it uses it. If not:

# Polymorphism 2

- ✓ The parent class becomes responsible for handling the message and looks for a method with that signature. If so, it uses it. This process continues until a match is found. If not match is found, a compile-time error is reported.
- ✓ Remember that inheritance defines the **is-a** relationship.
- ✓ The is-a relationship allows subclass objects to be treated as superclass objects, because **a subclass object IS A superclass object**, because a student IS A person.
- ✓ **Method-overriding** refers to the idea of having a subclass contain a method with the same signature as that of a method in its parent class.
- ✓ An objects ability to decide what method to apply to itself, depending on where it is in the inheritance hierarchy, is called **polymorphism**.

# Polymorphism 3

- ✓ What makes polymorphism work is **late-binding**, which means that the compiler does not generate the code to call a method at compile-time; instead the compiler generates code to calculate which method to call, using type information from the object.
- ✓ (In C++, you had to declare functions as virtual for dynamic binding but in Java this is the default behavior).

**E.x6/** In the following example, we see how polymorphism is achieved in java:

```
public class ManagerTest
{
    public static void main(String[] args)
    {
        Manager boss = new Manager("Razawa", 5);
        boss.setSecretaryName("Lava");
    }
}
```

# Polymorphism 4

```
Employee[] staff = new Employee[3];  
  
staff[0] = boss; //is-a relation  
  
staff[1] = new Employee("Lava", 5);  
  
staff[2] = new Employee("Aza", 3);  
  
for (int i = 0; i < 3; i++)  
    staff[i].raiseSalary(10);  
  
for (int i = 0; i < 3; i++)  
    staff[i].print();  
  
}
```

```
}
```

# Polymorphism 5

```
class Employee
{
    public Employee(String n, double s)
    {   name = n;   salary = s;}
    public void print()
    {   System.out.println(name + " " + salary);   }

    public void raiseSalary(double byPercent)
    {   salary = salary + byPercent / 100;   }

    public String Name() { return name;}

    private String name;   private double salary;
```

# Polymorphism 6

```
class Manager extends Employee
{
    public Manager(String n, double s)
    {    super(n, s);        secretaryName = "";    }

    public void raiseSalary(double byPercent)
    {    super.raiseSalary(byPercent + 10);    }

    public void setSecretaryName(String n)
    {    secretaryName = n;    }

    public String getSecretaryName()
    {    return secretaryName;    }

    private String secretaryName;
}
```



# Polymorphism 9

- ❑ Class Manager extends class Employee.
  - ❑ In the test class we create 3 objects, 2 of type employee and one of type Manager.
  - ❑ Method raiseSalary() is overridden in subclass Manager.
  - ❑ Then we create an array of type Employee and store the 3 objects in the array, even the object boss of type Manager, After that we invoke the method raiseSalary() on all the array elements.
  - ❑ Since this method is overridden, its correct version is applied to each individual element.
  - ❑ If you don't want a class to be inherited or a method overridden, declare them as
- final, you may do this for efficiency and safety.

# Packages

- ❑ To make classes easier to find and to use, to avoid naming conflicts, and to control access, programmers bundle groups of related classes and interfaces into packages.
- ❑ **A package is a collection of related classes and interfaces providing access protection and namespace management.**
- ❑ The classes and interfaces that are part of the Java platform are members of various packages that bundle classes by method.
- ❑ Fundamental classes are in `java.lang`, classes for reading and writing (input and output) are in `java.io`, and so on.
- ❑ In few slides we learn to put our classes and interfaces in packages, too.

# Packages 2

- ❑ To create a new package, first create a folder; the name of this folder will also be the name of the package.
- ❑ Create your classes and interfaces and save them in this new folder.
- ❑ Include a package statement at the top of every source file that defines a class or an interface that is to be a member of that package.
- ❑ If you put multiple classes in a single source file, only one may be public, and it must share the name of the source files base name.
- ❑ Only public package members are accessible from outside the package.
- ❑ To create a package within another package, just create a subfolder in the original folder and give the subfolder the same name as your sub-package or nested package.

# Packages 3

## ❑ To use package members:

- ✓ Refer to the member by its long (qualified) name

**(package.myClass ob;)**

- ✓ Import the package member

**(import package.myClass;)**

- ✓ Import the members entire package

**(import package.\*;)**

# Packages 4

**E.x25/** A demo program importing a user-defined sub-package:

```
import myPackage.subPackage.*;
public class TestPackage
{
    public static void main(String[] a)
    {
        MySubClass ob=new MySubClass(); ob.myMethod();
    }
}
```

❑ And the class in the package:

```
package myPackage.subPackage;
public class MySubClass
{
    public void myMethod()
    {
        System.out.println("This is in a subpackage");
    }
}
```

# Packages 5

- ❑ Note that this class is saved in a folder called **subPackage** and in turn it is saved in another folder called **myPackage**.
  
- ❑ **Ex26/** In this exercise you will create a package called MyPackage and create a few classes and save them in the package. Remember that all classes in the package should have as their first line “ package MyPackage” . Also make your classes public, otherwise they would not be accessible outside the package.

# Exception Handling

- ❑ A program often encounters problems as it executes.
- ❑ It may have trouble reading data, there might be illegal characters in the data, or an array index might go out of bounds.
- ❑ Java Exceptions enable the programmer deal with such problems.
- ❑ You can write a program that recovers from errors and keeps on running, this is important.
- ❑ A word processor program should not crash when the user makes an error!

# Exception Handling 2

- ❑ **Typical problems:** User input errors: specifying a syntactically wrong URL; typing errors; Device errors: printer may be off; printer out of paper; connection may be down;
- ❑ **Physical Limitations:** disks fill up; run out of memory;
- ❑ **Code errors:** computing an invalid array index;
- ❑ Traditional error handling mechanisms involved functions returning special error codes which the calling method analyzed.
- ❑ Further, the flow of program is cluttered with error-checking code.
- ❑ Finally, this approach is not very Object-oriented; it would be far better to have more information about the error than a simple code value.



# Exception Handling 3

- ❑ **An exception is a problem that occurs when a program is running.** Often the problem is caused by circumstances outside the control of the program, for example, bad user input or a bad sector in a disk file.
- ❑ A Java program may have statements that catch exceptions.
- ❑ When an exception occurs, the Java virtual machine creates an object of class Exception (big 'E') which holds information about the problem. The Exception object is used to recover from the problem.
- ❑ **Using Exceptions has several advantages:**
  - 1- Error handling is separated from regular program flow

# Exception Handling 4

2- errors are encapsulated in objects and provide more information about the error.

This approach is more object-oriented.

3- You can either use standard Java exceptions or create your own exception classes to define new error situations.

❑ There is a class called **Exception** which describes general error situations. This class has two subclasses: **IOException** and **RuntimeException**.

❑ **Runtime exceptions** occur when you have made a programming error: bad cast, out-of-bounds array access, or a null pointer.

# Exception Handling 5

- ❑ Any other exception occurs because something bad has happened to your otherwise good program: I/O problems, malformed URL, disk failure.
- ❑ The general rule is that if it is **RuntimeException**, it was your fault.
- ❑ For methods which threaten to throw I/O exceptions, you should advertise this in the method declaration. For Runtime exceptions you should not do this.

```
E.x27/ import java.io.* ;
```

```
public class Square
```

```
{
```

```
    public static void main ( String[] a ) throws IOException
```

# Exception Handling 6

```
BufferedReader stdin = new BufferedReader ( new
    InputStreamReader( System.in ) );

String inData; int num ;

System.out.println("Enter an integer:");

inData = stdin.readLine();

num = Integer.parseInt( inData );

System.out.println("The square of " + inData + " is
    " + num*num );

}

}
```

Here if the user enters bad input, the program will throw some type of **IOException**.

# Exception Handling 7

- ❑ In the main method we advertise that this method might throw an exception, but we don't specify any handlers for the exception; instead we say that the called (Java Virtual Machine) should handle the exception which will stop the program and print a trace of what went wrong.

**Ex.28/** Now the same example with try and catch blocks:

```
import java.io.* ;

public class SquareUser
{
    public static void main ( String[] a ) throws IOException
    {
```

# Exception Handling 8

```
BufferedReader stdin = new BufferedReader ( new
InputStreamReader( System.in ) );

String inData = null;

int num = 0; boolean goodData = false;

while ( !goodData )
{

    System.out.println("Enter an integer:");

    inData = stdin.readLine();

    try {

        num = Integer.parseInt( inData );

        goodData = true;

    }
```

# Exception Handling 9

```
        } catch (NumberFormatException ex )
        {System.out.println("Bad data, try again.\n" );}
    }
    System.out.print("The square of"+ inData+"is"+
num*num );
}
}
```

- ❑ In this program, the user is prompted again if the input is bad. Note that after the catch {} block is executed, execution continues with the statement that follows the catch{} block. (Execution does not return to the try{} block.) **What is this the case?**

# Exception Handling 10

- ❑ You may want to catch more than one type of exception in a method.
- ❑ To do this you need to define more than one catch statement for each type of exception you want to handle.
- ❑ **E.x29/** Here is a program that asks the user for an integer and for an array index where it is to be placed. Only indexes 0 through 9 are allowed, any other index causes an **IndexOutOfBoundsException**:

```
import java.io.* ;

public class IndexPractice
{
    public static void main(String[]a) throws IOException
```



# Exception Handling 11

```
{  
    BufferedReader stdin = new BufferedReader ( new  
        InputStreamReader( System.in ) );  
    String inData;  
    int data=0, slot=0 ;  
    int[] value = new int[10];  
    try  
    {  
        System.out.println("Enter the data:");  
        inData = stdin.readLine();  
        data = Integer.parseInt( inData );  
        System.out.println("Enter the array index:");
```

# Exception Handling 12

```
        inData = stdin.readLine();  
  
        slot = Integer.parseInt( inData );  
  
        value[slot] = data;  
  
    } catch (NumberFormatException ex )  
  
    { System.out.println("The problem: " +  
ex.getMessage() + "At"); ex.printStackTrace(); }  
  
    catch (IndexOutOfBoundsException ex )  
  
    { System.out.println("The problem: " +  
ex.getMessage() + "At"); ex.printStackTrace(); }  
  
}
```

# Exception Handling 13

❑ Some typical exceptions to catch:

- ✓ **IOException:** Signals that an I/O exception of some sort has occurred. This class is the general class of exceptions produced by failed or interrupted I/O operations .
- ✓ **EOFException:** Signals that an end of file or end of stream has been reached unexpectedly during input.
- ✓ **MalformedURLException:** indicate that a malformed URL has occurred. Either no legal protocol could be found in a specification string or the string could not be parsed.
- ✓ **UnknownHostException:** Thrown to indicate that the IP address of a host could not be determined.

# OO programming (Abstract Classes)

- ❑ Abstract classes are classes that cannot be instantiated, but you can have abstract class variables that refer to some extended class object.
- ❑ Abstract classes are used to factor out common behavior into a general class.
- ❑ Abstract classes may contain abstract methods: methods that have no definition in the original class and **must** be defined in non-abstract subclasses (pure virtual classes in C++).

```
abstract class Message
{
    public abstract void play(); //must be overridden
}
class TextMessage extends Message
{
    public void play()
    { System.out.println("Text Message"); }
}
```

# OO programming (Abstract Classes) 2

## ❑ Rules about abstract classes/methods:

- ✓ Any class with an abstract method is automatically abstract and **must be declared as such**. Also, an abstract class cannot be instantiated.
- ✓ A subclass of an abstract class can be instantiated only if it overrides each of the abstract methods of its superclass and provides an implementation for all of them.  
(Concrete class, as opposed to abstract)
- ✓ If a subclass of an abstract class does not implement all the abstract methods it inherits, that subclass is itself abstract, **and must be declared as such**.
- ✓ static, private and final methods cannot be abstract, since they cannot be overridden by a subclass.
- ✓ A class can be declared abstract even though it does not have any abstract methods.

# Generic Programming and Methods

- ❑ As mentioned before, the Object class is the; every class in Java extends this class.  
ultimate ancestor

- ❑ You can use a variable or object of type Object to refer to any type:

```
Object obj=new Employee("H. Hacker", 10000);
```

# Generic Programming and Methods 2

For example, for the Employee class:

```
public boolean equals(Object obj) {  
    if (!(obj instanceof Employee))  
        return false;  
    Employee e = (Employee) obj;
```

- ❑ Note that the equals method for the String class is overridden by Java.
- ❑ Since objects of any type or class can be held in variables of type Object, we can use this for generic programming.
- ❑ For example, suppose we need a method that takes an array and a value as parameters and we want the method to return the index of that value in the array:

# Generic Programming and Methods 3

```
static int find(Object[] ob, Object value)
```

```
{
```

```
    for (int i=0; i<ob.length; i++)
```

```
        if(ob[i].equals(value)) return i;
```

```
    return -1; //not found
```

```
}
```

```
    return name.equals(e.name) && ...}
```

```
Employee[] staff=new Employee[10];
```

```
Employee e=new Employee("Hacker", 10000);
```

```
//...
```

```
int n=find(staff, e); //works on any object array
```



# OO programming (Interfaces)

- ❑ Java does not support multiple inheritance, it's class can extend only one class.
  - ❑ When need classes that inherit behavior from more than one parent class. the solution to this is using Java **interfaces**.
  - ❑ Many believe that multiple inheritance, in C++, introduces more complexity and work on the part of the programmer than solve problems.
  - ❑ **An interface:** is a class but a class which can only contain abstract methods and constants (finals).
  - ❑ It cannot contain any implementation for its methods nor can have any instance fields because **an interface is a specification and has no implementation detail**.
- Its methods are implicitly abstract.

# OO programming (Interfaces) 2

- ❑ Any class that implements an interface must define the interface methods or must itself be an abstract class, it can implement as many interfaces as it needs.
- ❑ **E.x31/** The following example is taken from the textbook:

```
interface Shape {  
  
    public abstract double area();  
  
    public abstract double volume();  
  
    public abstract String getName();    }  
  

```

# OO programming (Interfaces) 3

**class Point implements Shape**

```
{  
  
    private int x, y; // coordinates of the Point  
  
    public Point() { setPoint( 0, 0 ); }  
  
    public Point( int a, int b ) { setPoint( a, b ); }  
  
    public void setPoint( int a, int b ){x = a; y = b; }  
  
    public int getX() { return x; }  
  
    public int getY() { return y; }  
  
    public String toString()  
  
    { return "[" + x + ", " + y + "]; }  
}
```

# OO programming (Interfaces) 3

```
public double area() { return 0.0; }

public double volume() { return 0.0; }

public String getName() { return "Point"; }

}

public class Circle extends Point {

    private double radius;

    public Circle() { setRadius( 0 ); }

    public Circle( double r, int a, int b )

{
    super( a, b );        setRadius( r ); }
}
```

# OO programming (Interfaces) 4

```
public void setRadius( double r )
{ radius = ( r >= 0 ? r : 0 ); }

public double getRadius()
{ return radius; }

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

public String toString()
{
    return "Center = " + super.toString() + "; Radius =
    " + radius; }

public String getName() { return "Circle"; }
```

# OO programming (Interfaces) 5

```
public class Cylinder extends Circle
```

```
{
```

```
    private double height; // height of Cylinder
```

```
    public Cylinder() {        setHeight( 0 ); }
```

```
    public Cylinder( double h, double r, int a, int b)
    { super( r, a, b );    setHeight( h ); }
```

```
    public void setHeight( double h )
```

```
    { height = ( h >= 0 ? h : 0 ); }
```

```
    public double getHeight() { return height; }
```

```
    public double area()
```

```
    {
        return 2 * super.area() + 2 * Math.PI * radius
            * height; }
    }
```

# OO programming (Interfaces) 6

```
public double volume()
{ return super.area() * height; }

public String toString()
{ return super.toString() + "; Height = " + height; }

    public String getName() { return "Cylinder"; }
}

import javax.swing.JOptionPane;
import java.text.DecimalFormat;

public class Interface
{

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

# OO programming (Interfaces) 7

```
Point point = new Point( 7, 11 );  
Circle circle = new Circle( 2, 22, 8 );  
Cylinder cylinder = new Cylinder( 10, 3, 10, 10 );  
Shape arrayOfShapes[] = new Shape[ 3 ];  
arrayOfShapes[ 0 ] = point;  
arrayOfShapes[ 1 ] = circle;  
arrayOfShapes[ 2 ] = cylinder;  
String output =point.getName() + ": " +  
point.toString() + "\n" +circle.getName() + ": " +  
circle.toString() + "\n" +cylinder.getName() + ": "  
+ cylinder.toString();
```



# OO programming (Interfaces) 8

```
DecimalFormat precision2 = new DecimalFormat( "0.00" );  
    for ( int i = 0; i < arrayOfShapes.length; i++ )  
    {  
        output += "\n\n" + arrayOfShapes[  
            i].getName() + ": " + arrayOfShapes[ i  
                ].toString() + "\nArea = " + precision2.format(   
                    arrayOfShapes[ i ].area() ) + "\nVolume = "  
                + precision2.format( arrayOfShapes[ i  
                    ].volume() ) );  
    }  
JOptionPane.showMessageDialog( null, output, "Demonstrating  
    Polymorphism", JOptionPane.INFORMATION_MESSAGE );  
    System.exit( 0 );  
}}}
```

# OO programming (Interfaces) 9

- ❑ **Abstract classes or interfaces:** Interfaces are used in place of abstract classes when there is **no default implementation**.
- ❑ In addition to providing support for **multiple inheritance**, interfaces are commonly used in **GUI programming** as will soon see.
- ❑ Usually interfaces are defined in classes of their own with the same name as the interface name and in a **.java file**.
- ❑ Since they both may contain abstract methods it is not possible to **instantiate objects** from them and they **may not define constructor** methods.
- ❑ **Interfaces** can only contain **abstract instance methods** and **constants** whereas **abstract classes** can contain **instance fields** and a **mixture of abstract and instance methods**.

# OO programming (Interfaces) 10

- ❑ If you add a new method to an interface which has already been implemented by some class, you break that subclass. This is not a problem with abstract classes.
- ❑ All methods of an interface are **implicitly public**, even if the public modifier is omitted. also it is an error to define **protected or private** methods in an interface.

**E.x32/** Here is another example using interfaces:

```
import java.util.*;

public class ArrayAlg
{
    public static void main(String[] l)
    {
```

# OO programming (Interfaces) 11

```
Employee[] staff=new Employee[3];  
  
staff[0]=new Employee("Harry", 35000);  
staff[1]=new Employee("Barry", 32000);  
staff[2]=new Employee("Jerry", 29000);  
  
ArrayM.Sort(staff);  
  
for(int i=0; i<staff.length;i++)  
    System.out.println(staff[i]);  
  
}  
  
}
```

# OO programming (Interfaces) 12

```
interface Sortable
```

```
{ public int compareTo(Sortable b);}
```

```
class Employee implements Sortable
```

```
{
```

```
    public Employee(String n, double s)
```

```
    {    name=n; salary=s;}
```

```
    public void raiseSalary(double byPercent)
```

```
    {    salary*=1+byPercent/100;}
```

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

```
    public double getSalary() {return salary;}
```

```
    public String toString()
```

```
    {return name + " " + salary}
```

# OO programming (Interfaces) 13

```
public int compareTo(Sortable b)
{
    Employee eb=(Employee) b;
    if (salary<eb.salary) return -1;
    if (salary>eb.salary) return 1;
    return 0; }

private String name;

private double salary;

}
```

# OO programming (Interfaces) 14

```
class ArrayM{
    public static void Sort(Sortable[] a){
        for(int i=0; i<a.length; i++)
        {
            for(int j=0; j<a.length-1; j++)
            {
                if(a[j].compareTo (a[j+1])>=1)
                {
                    sortable temp=a[j];
                    a[j]=a[j+1];
                    a[j+1]=temp;
                }
            }
        }
    }
}
```

# OO programming (Interfaces) 16

- ❑ In the example, the `employee` class implements the **Sortable** interface which has only one method, the **compareTo** method.
- ❑ The class **ArrayAlg** contains a static method **Sort** which takes an array of objects of **any class which implements the Sortable** interface and sorts the elements of the array in descending order.
- ❑ This example used an interface; you could have used an abstract class to achieve the same effect, but if the **employee** class had already inherited (extended) another class, say a class **Person**, then you wouldn't have been able to use abstract classes.



# Data base Programming

- ❑ A database system is a repository or store of data, it's systems organize data in an orderly way making the data easily accessible and updatable.
- ❑ Instead of using a database, you could use an ordinary text file to store your data.
- ❑ But we all know how a database management system makes updating, organizing and accessing data more efficient and user-friendly.
- ❑ Being able to access a database through programming is also important and allows the programmer to write **custom applications**.
- ❑ Software users (companies, universities, governments, stores...) prefer customized software, software that is designed to need their requirements and that is easy to

# Data base Programming 2

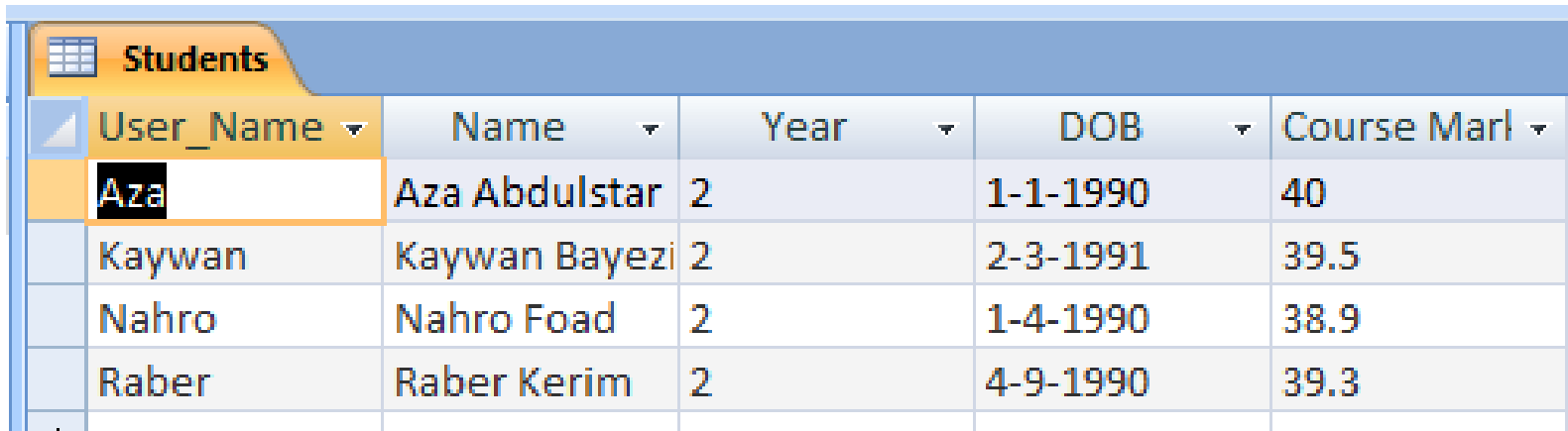
- ❑ With **JDBC** you can access almost all current database systems (MS SQL Server, **MS Access**, Oracle, DB2, Informix...).
- ❑ JDBC works like this: database vendors provide drivers for their particular database system to work with JDBC driver manager.
- ❑ JDBC provides an abstraction layer on top of these drivers.
- ❑ Programs written according to JDBC API would talk to the JDBC driver manager, which in turn, would use the drivers that it has to talk to the actual database.
- ❑ Using JDBC, Java programmers can write applications to access any database, using standard SQL statements.

# Data base Programming 3

- ❑ Java Database Connectivity or JDBC is designed to provide Java programmers with a uniform way of accessing database systems.
- ❑ JDBC is an interface to SQL which is an interface to all modern relational databases.
- ❑ So basically, JDBC let's you pass and execute SQL statements to databases.
- ❑ You can think of a database as a group of named tables with rows and columns.
- ❑ Each column has a column name. The rows contain the actual data (records).

# Data base Programming 4

- ❑ We assume we have a small **MS Access** database with a single table.
- ❑ The program which follow will use this database for testing, the table Students has the following design:



User_Name	Name	Year	DOB	Course Marl
Aza	Aza Abdulstar	2	1-1-1990	40
Kaywan	Kaywan Bayezi	2	2-3-1991	39.5
Nahro	Nahro Foad	2	1-4-1990	38.9
Raber	Raber Kerim	2	4-9-1990	39.3

- ❑ In SQL, to select all the records from the above table you would use:

**SELECT \* FROM Students**

# Data base Programming 5

- ❑ The FROM clause tells database which table to access. Or you can restrict the columns:

```
SELECT Name, CourseMark FROM Students
```

- ❑ To restrict the rows or records, you use the WHERE clause:

```
SELECT * FROM Students WHERE CourseMark >40
```

- ❑ You can also use SQL to update the data in a table:

```
UPDATE Students SET CourseMark=CourseMark + 2
```

- ❑ To insert values into a table, use the INSERT statement:

```
INSERT INTO Students Values ('Lava', 'Lava Burhan', 21, '10-  
Oct-1993', 21.5)
```

# Data base Programming 6

- ❑ To create a new table:

```
CREATE TABLE Departments ( Dept_Name CHAR(20) , Head  
CHAR(20) , No_Of_Students INT)
```

- ❑ You also need to create a Data Source Name (DSN) for your Access database, to use it when you connect to the database (This will be demonstrated in the lab).
- ❑ **E.x33/** Write a simple Java program that will access the above database .

```
import java.sql.*;  
  
public class CreateMarks  
{  
  
    public static void main(String args[])  
{
```

# Data base Programming 7

```
String url = "jdbc:odbc:JavaDSN1";  
Connection con;String createString; Statement stmt;  
createString = "select * from Students";  
try { //load odbc-jdbc bridge driver  
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
}catch(Exception e){System.out.println(e.getMessage());}  
try {  
    con = DriverManager.getConnection(url, "", "");  
    stmt = con.createStatement();  
    ResultSet rs=stmt.executeQuery(createString);
```

# Data base Programming 8

```
while(rs.next())

    System.out.println(rs.getString(2) + " | "
+rs.getString(3) + " | " + rs.getString(4) +
" |" + rs.getString(5));

rs.close();    stmt.close();        con.close();

}catch(SQLException ex)

{System.err.println("SQLExc.:" + ex.getMessage());}

}
```



# Data base Programming 9

- ❑ The string url is used to represent a protocol, here **jdbc:odbc:** means that JDBC interfaces Microsoft's ODBC which then connects to the database server - whose Internet address is denoted by the DSN Data Source Name **StudentsDSN**.
- ❑ A driver is needed which bridges between the Java JDBC and Microsoft's ODBC.
- ❑ Then the **connection** to the database, specified by URL, user name and password, is established.
- ❑ Finally, a **statement** object is created, which will allow us to pass SQL statements to the database.

# Data base Programming 10

- ❑ You create an SQL statement and pass this statement to the **executeQuery** method of the statement object.
- ❑ The result of this method is a **ResultSet** object ( a set of tuples/records).
- ❑ Then we use a loop to iterate through the elements of the resultset object.
- ❑ A **resultset** object is a table of data representing a database result set, which is usually generated by executing a statement that queries the database, it's maintains a cursor pointing to its current row of data, initially the cursor is positioned before the first row.

# Data base Programming 11

- ❑ The next method moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set.
- ❑ Note that the method getString is given a number; this number represents the column number. They start with 1. You could use column names (in double quotes) instead:

```
rs.getString("Name")    Or    rs.getInteger("Year");
```

- ❑ To make updates to a database you can use the **executeUpdate** method of the Statement object. For example, to add a new record to the Students table:

```
stmt.executeUpdate("INSERT INTO Students values('username','User Name',3,'2-
```

# Data base Programming 12

- ❑ This statement would add the record to the table. Note that it is customary to use uppercase letters for SQL keywords words. **SQL is case insensitive.**
- ❑ The **executeUpdate** method returns the number of records affected by the update operation, for example the following SQL statement would return 3:

```
int n=stmt2.executeUpdate("UPDATE Students SET [cousework mark]=40  
where[cousework mark]<40");  
  
System.out.println(n);
```

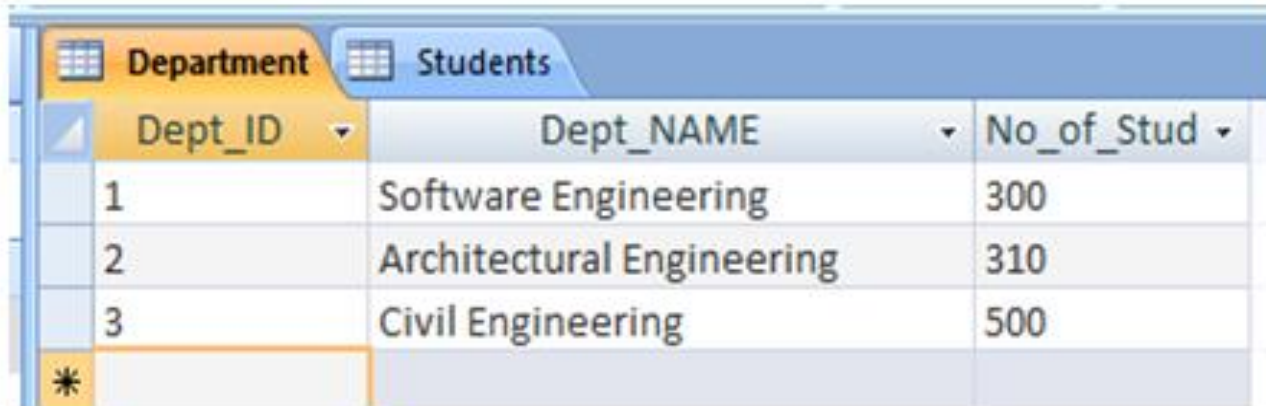
- ❑ You can create a new table as follows:

```
String createTable = "CREATE TABLE Department (Dept_ID INTEGER,  
Dept_NAME VARCHAR(40))";
```

# Data base Programming 13

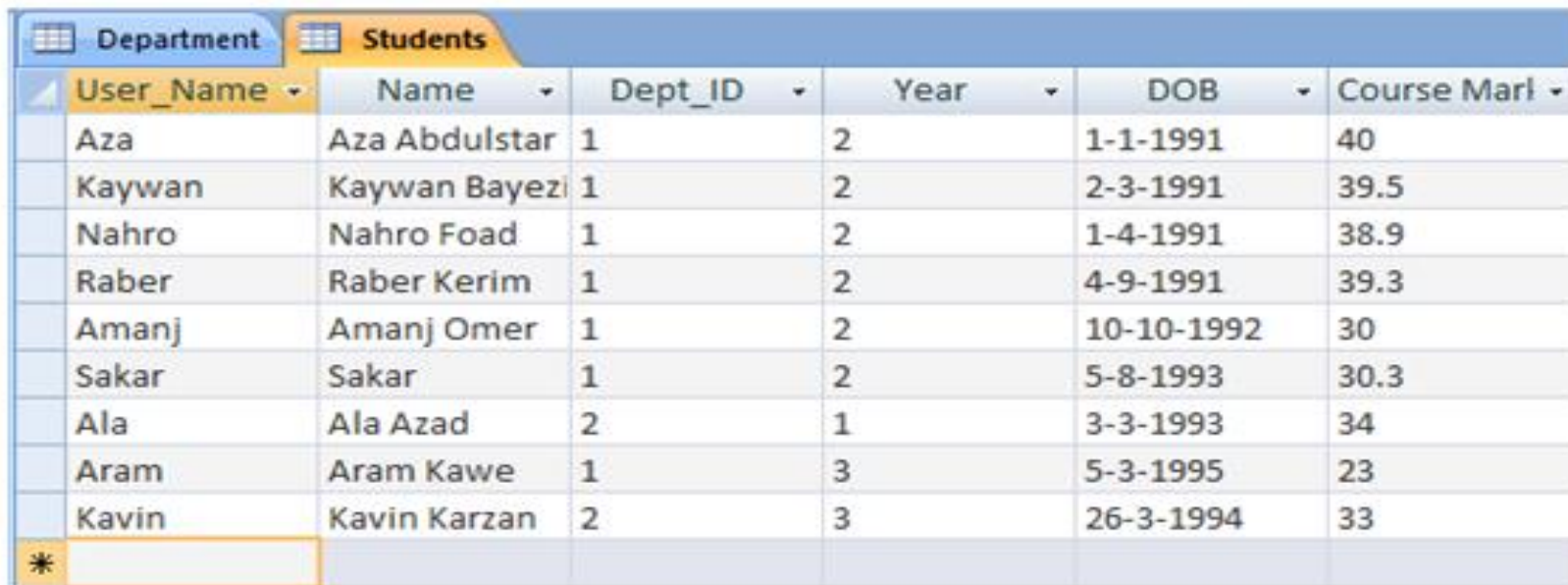
- ❑ There is a lot more to JDBC than what we have looked at so far.
- ❑ You must know SQL to write more advanced database applications, especially when you need to access databases with many tables and when there are relationships between those tables.
- ❑ There are times when you don't want a statement to be executed unless another statement/statements are also executed. For example, consider the following two tables:

# Data base Programming 14



A screenshot of a database interface showing a table named 'Department'. The table has three columns: 'Dept\_ID', 'Dept\_NAME', and 'No\_of\_Stud'. The data is as follows:

Dept_ID	Dept_NAME	No_of_Stud
1	Software Engineering	300
2	Architectural Engineering	310
3	Civil Engineering	500
*		



A screenshot of a database interface showing a table named 'Students'. The table has six columns: 'User\_Name', 'Name', 'Dept\_ID', 'Year', 'DOB', and 'Course Marl'. The data is as follows:

User_Name	Name	Dept_ID	Year	DOB	Course Marl
Aza	Aza Abdulstar	1	2	1-1-1991	40
Kaywan	Kaywan Bayezi	1	2	2-3-1991	39.5
Nahro	Nahro Foad	1	2	1-4-1991	38.9
Raber	Raber Kerim	1	2	4-9-1991	39.3
Amanj	Amanj Omer	1	2	10-10-1992	30
Sakar	Sakar	1	2	5-8-1993	30.3
Ala	Ala Azad	2	1	3-3-1993	34
Aram	Aram Kawe	1	3	5-3-1995	23
Kavin	Kavin Karzan	2	3	26-3-1994	33
*					

# Data base Programming 15

- ❑ Now suppose you would like to update the Students table by adding a new record to it.
- ❑ If this update operation is successful, you would also like the second table, Departments, to be updated too, otherwise your database would be in an **inconsistent** state.
- ❑ You would want to update the value of the third column in the Departments table.
- ❑ A **transaction** is a set of one or more statements that are executed together as a **unit**, so either all of the statements are executed, or none of the statements is executed.
- ❑ When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and will be automatically committed right after it is executed.

# Data base Programming 16

- ❑ This is the default behavior, the way to allow two or more statements to be grouped into a transaction is to disable auto-commit mode.
- ❑ This is demonstrated in the following line of code, where con is an active connection: `con.setAutoCommit(false);`
- ❑ Once auto-commit mode is disabled, no SQL statements will be committed until you call the method `commit` **explicitly**.

```
try {  
  
    con = DriverManager.getConnection(url, "", "");  
  
    con.setAutoCommit(false);  
  
    stmt1 = con.createStatement();  
  
    stmt2 = con.createStatement();
```



# Data base Programming 17

```
stmt1.executeUpdate("INSERT INTO Students
values('username9','User Name 9',1, 4,'2-Apr-
80',45)");

stmt2.executeUpdate("Update Department SET
No_of_Student=No_of_Student+1 where
Dept_ID=1");

con.commit();

con.setAutoCommit(true);

con.close();

}catch(SQLException
e){System.out.println("SQLException:");}
```

# Data base Programming 19

## ❑ The steps involved for making a connection with a database

- **Loading the driver** : To load the driver, Class. forName() method is used.

Class. forName("sun. jdbc. odbc. JdbcOdbcDriver");

- **Making a connection with database:** To open a connection to a given database, DriverManager. getConnection() method is used.

Connection con = DriverManager. getConnection ("jdbc:odbc:somedb", "user", "password");

- **Executing SQL statements** : To execute a SQL query, java. sql. statements

class is used. Statement stmt = con. createStatement();

# Data base Programming 20

- executeQuery() method of Statement executes the statement and returns a java.sql. ResultSet that encapsulates the retrieved data: `ResultSet rs = stmt.executeQuery("SELECT * FROM some table");`

## Connecting to Database

- ✓ Either by giving database path in detail as follows:

```
Connection con = DriverManager.getConnection("jdbc:odbc:Driver={Microsoft  
Access Driver (*.mdb)};DBQ=d:\\Students.mdb", "", "");
```

- ✓ Or by creating DSN for the database as follows:

```
Connection con = DriverManager.getConnection(" jdbc:odbc:Students“,”,””);
```

# Network Programming

To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection. A socket is one end-point of a two-way communication link between two programs running on the network. The `java.net` package provides two classes: `Socket` and `ServerSocket` that implement the client side of the connection and the server side of the connection, respectively.

# Network Programming2

Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined. The computer is identified by its 32-bit IP address. So in order to establish communication between a client and a server, we need a socket for the client and socket for the server. Then we need to specify a port and bind the socket (client or server ) to this port, depending on the application. Your applications can use any ports not in the range of well-known ports.

# Network Programming3

In the following example, we will write a client and a server program. The client will try to establish a connection with the server. After the connection is made between the two, the client can send strings to the server and the server will return or echo what the client sent it. We will make the server to listen on port 4444, so the client will use this port number to connect to the server. Of course, the client will also need the host name of the server. You can use this client/server application on a network or on the same computer.

# Network Programming4

First the code for the client machine:

```
import java.io.*;import java.net.*;

public class EchoClient{

    public static void main(String[] args) throws
    IOException

    {

Socket echoSocket = null;    PrintWriter out = null;

        BufferedReader in = null;

        try{echoSocket = new Socket("localhost", 4444);
out = new PrintWriter(echoSocket.getOutputStream(), true);
```

# Network Programming5

```
in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));  
  
    }catch (UnknownHostException e)  
  
    {System.err.println("Don't know host!"); System.exit(1);}  
  
    catch (IOException e) {  
  
System.err.println("Couldn't get I/O for the connection to!");System.exit(1);}  
  
BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));  
  
    String userInput; System.out.println(in.readLine());
```



# Network Programming6

```
while ((userInput = stdIn.readLine()) != null)
{
    out.println(userInput);
    System.out.println("echo: " + in.readLine());
}
out.close(); in.close(); stdIn.close(); echoSocket.close();
}
}
```

# Network Programming7

The program first creates a socket which will be used to connect to a machine “localhost” on port 4444, then gets the socket's output stream and input stream. The last statement in the while loop reads a line of information from the `BufferedReader` connected to the socket. The `readLine` method waits until the server echoes the information back to `EchoClient`. When `readline` returns, `EchoClient` prints the information to the standard output. The while loop continues until the user types an end-of-input character. That is, `EchoClient` reads input from the user, sends it to the Echo server, gets a response from the server, and displays it, until it reaches the end-of- input. The while loop then terminates and the program continues, executing the next four lines of code:

```
out.close(); in.close(); stdin.close(); choSocket.close();
```

These statements close the readers and writers connected to the socket and to the standard input stream, and close the socket connection to the server.

# Network Programming8

The server program begins by creating a new `ServerSocket` object to listen on a specific port. When writing a server, choose a port that is not already dedicated to some other service. The constructor for `ServerSocket` throws an exception if it can't listen on the specified port (for example, the port is already being used). If the server successfully connects to its port, then the `ServerSocket` object is successfully created and the server continues to the next step - accepting a connection from a client:

```
clientSocket = serverSocket.accept();
```

The `accept` method waits until a client starts up and requests a connection on the host and port of this server (in this example, the server is running on the hypothetical machine `taranis` on port `4444`).

# Network Programming9

**Here is the server program:**

```
import java.net.*; import java.io.*;

public class MultiServer {

    public static void main(String[] args) throws IOException{

        ServerSocket serverSocket = null; boolean listening = true;

        try { serverSocket = new ServerSocket(4444);} catch (IOException e) {

            System.err.println("Could not listen on port: 4444."); System.exit(-1);}

        while (listening)

            new MultiServerThread(serverSocket.accept()).start();

        serverSocket.close(); }}


```

# Network Programming10

```
class MultiServerThread extends Thread { private Socket socket = null;
    public MultiServerThread(Socket socket) { this.socket = socket; }
public void run() { try
    { PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        String inputLine, outputLine; out.println("Welcome to Echo Server");
        while ((inputLine = in.readLine()) != null) {
            outputLine = inputLine; out.println(outputLine);
            if (outputLine.equals("Bye")) break;}
        out.close(); in.close(); socket.close();
    } catch (IOException e) { e.printStackTrace();} }}
```

# Network Programming11

Every time a new socket connection is established (when method `accept` succeeds) a new thread is created and started to take care of the new connection. Here a new class is created which extends the `Thread` class. Any class that subclasses the `Thread` class should provide override the `run()` method. You should not call the `run` method. Instead call the object's `start()` method which will call the `run` method. You provide the desired functionality of the thread in the `run` method.

# Network Programming12

You should realize the importance of protocols by now. Look at the last two programs again and test them. If the client and the server do not understand each other, the client/server application will not function. They cannot communicate. For this simple client/server pair, it was not necessary to specify and document a protocol but for more complicated applications, you will need to specify and develop protocols before writing you client and server programs. Who should talk first? Second? When should they disconnect?

For more information consult your reference books.

**Good Luck**