# Object Oriented Programming with Java II


**Dr. Mohamed K. Hussein**

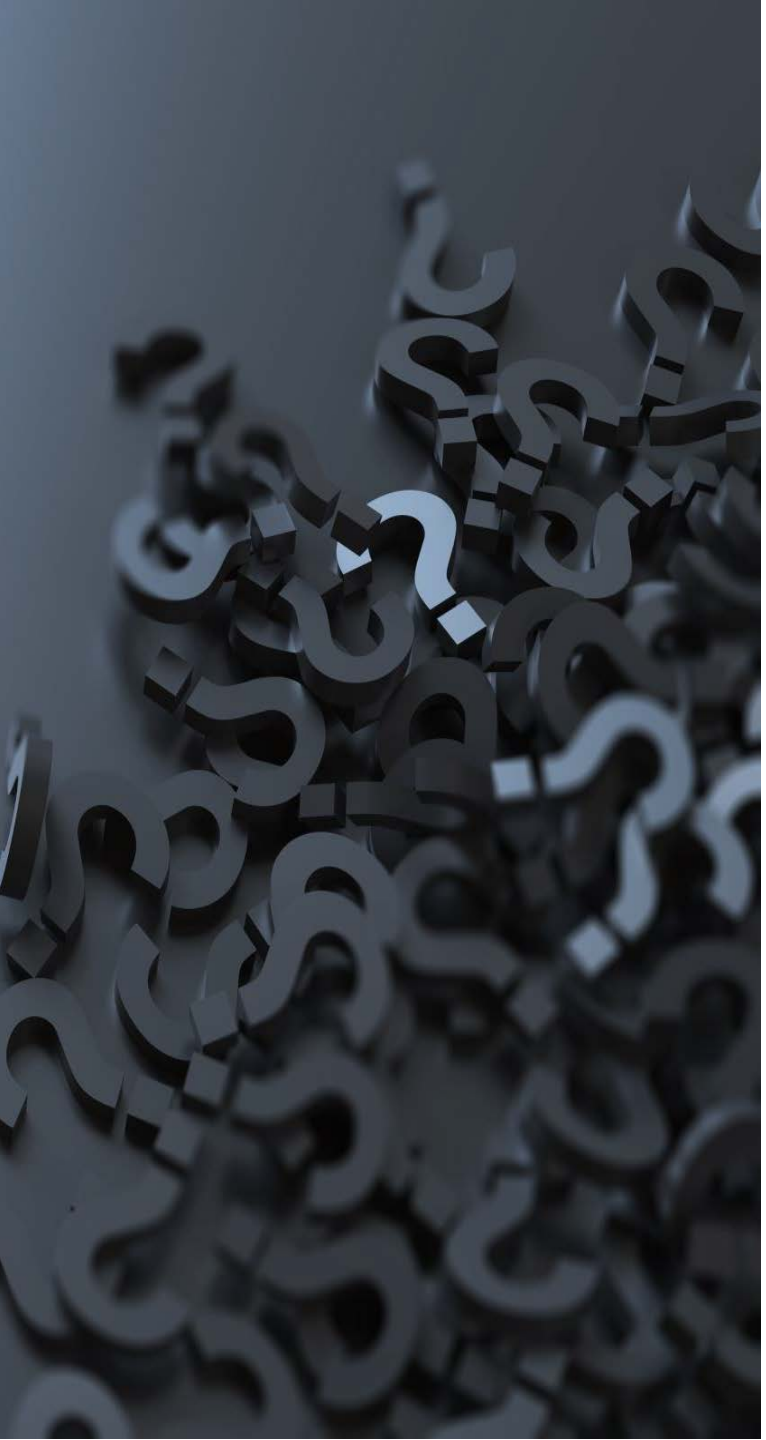

**Lecture 4**


Associate Prof., Computer Science department,

Faculty of Computers & Information Science,

Suez Canal University

**Email: m_khamis@ci.suez.edu.eg**



# Lecture outcomes

- Visibility Modifiers
- Inheritance
  - Superclasses and Subclasses
  - Deriving Subclasses
  - Public & Private members
  - Single versus Multiple inheritance
  - The super key word
  - Constructor Chaining
  - Calling Superclass Methods
  - Overriding methods
  - Overriding versus Overloading
  - The Object Class

# Visibility Modifiers

- Visibility modifiers can be used to specify the visibility of a class and its members.
- If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package.
  - This is known as *package-private* or *package-access*.
  - Packages can be used to organize classes: **package** packageName;

```
package p1;
public class C1 {
    public int x;
    int y;
    private int z;
    public void m1() {}
    void m2() {}
    private void m3() {}
}
```

```
package p1;
public class C2 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        can access o.y;
        cannot access o.z;
        can invoke o.m1();
        can invoke o.m2();
        cannot invoke o.m3();
    }
}
```



```
package p2;
public class C3 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        cannot access o.y;
        cannot access o.z;
        can invoke o.m1();
        cannot invoke o.m2();
        cannot invoke o.m3();
    }
}
```

# Visibility Modifiers

- If a class is not defined as public, it can be accessed only within the same package.

```
package p1;
class C1 {
...
}
```

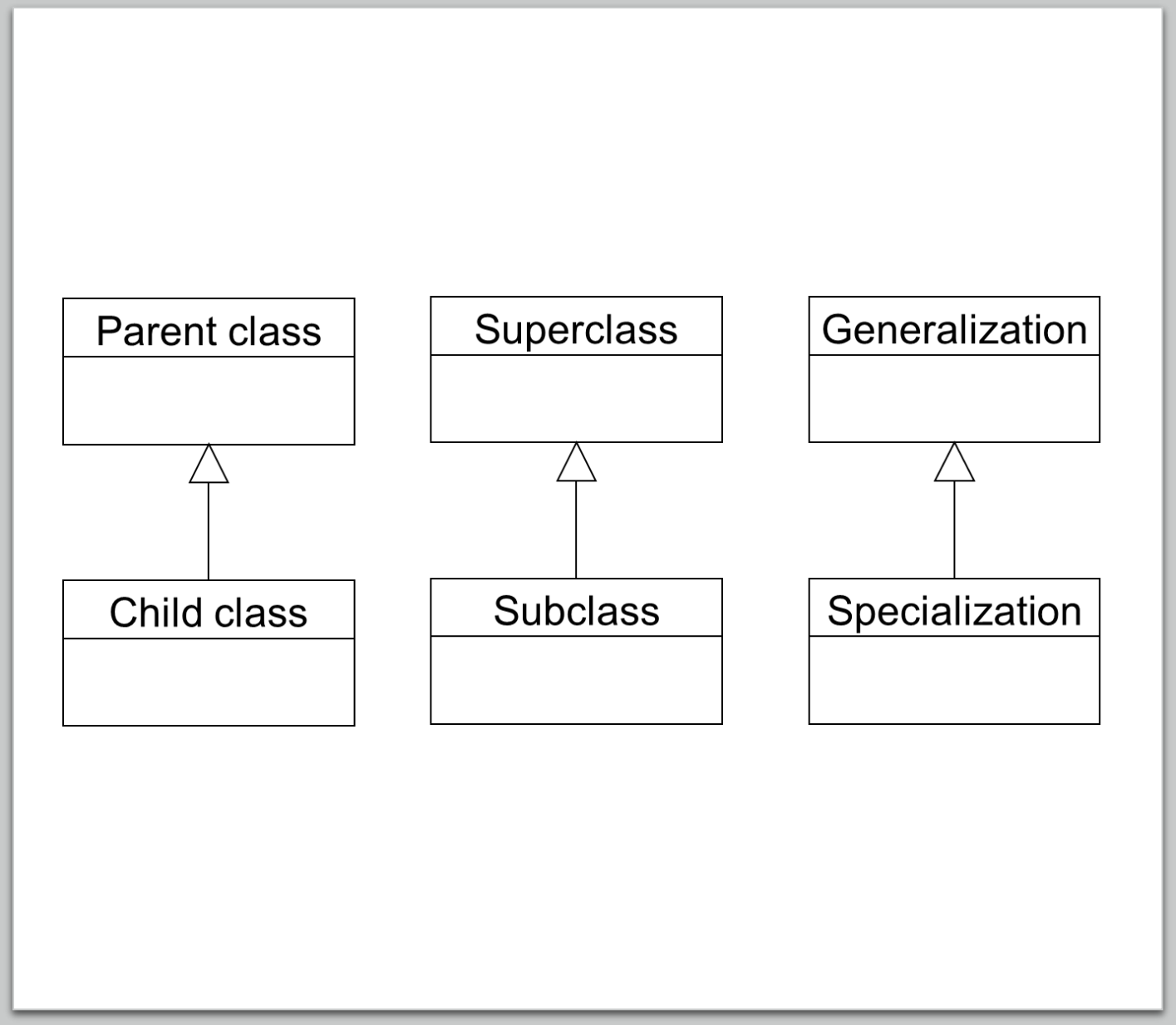
```
package p1;
public class C2 {
        can access C1
}
```

```
package p2;
public class C3 {
    cannot access C1;
    can access C2;
}
```

# *Inheritance*

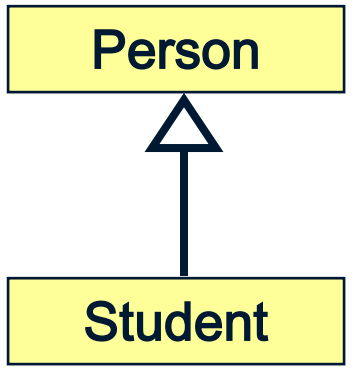
# Inheritance

- ***Inheritance*** allows a software developer to derive a new class from an existing one
  - The existing class is called the *parent class*, or *superclass*, or *base class*
  - The derived class is called the *child class* or *subclass*.
  - As the name implies, the child inherits characteristics of the parent
    - That is, the child class inherits the methods and data defined for the parent class



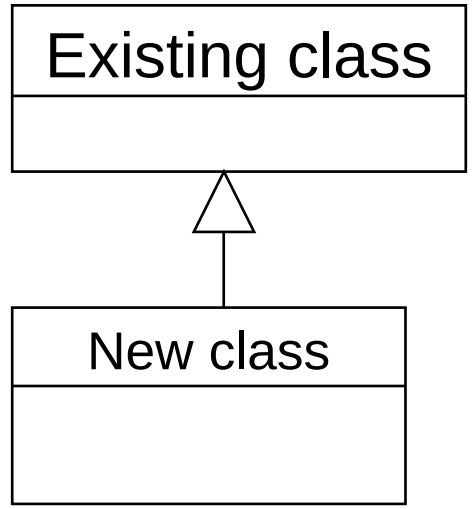
# Inheritance

- In the derived class, the programmer can add new variables or methods, or can modify the inherited ones
  - *Software reuse* is at the heart of inheritance
  - By using existing software components to create new ones



# Inheritance

- Inheritance relationships often are shown graphically in a UML class diagram, with an arrow with an open arrowhead pointing to the parent class




**Inheritance: UML representation**

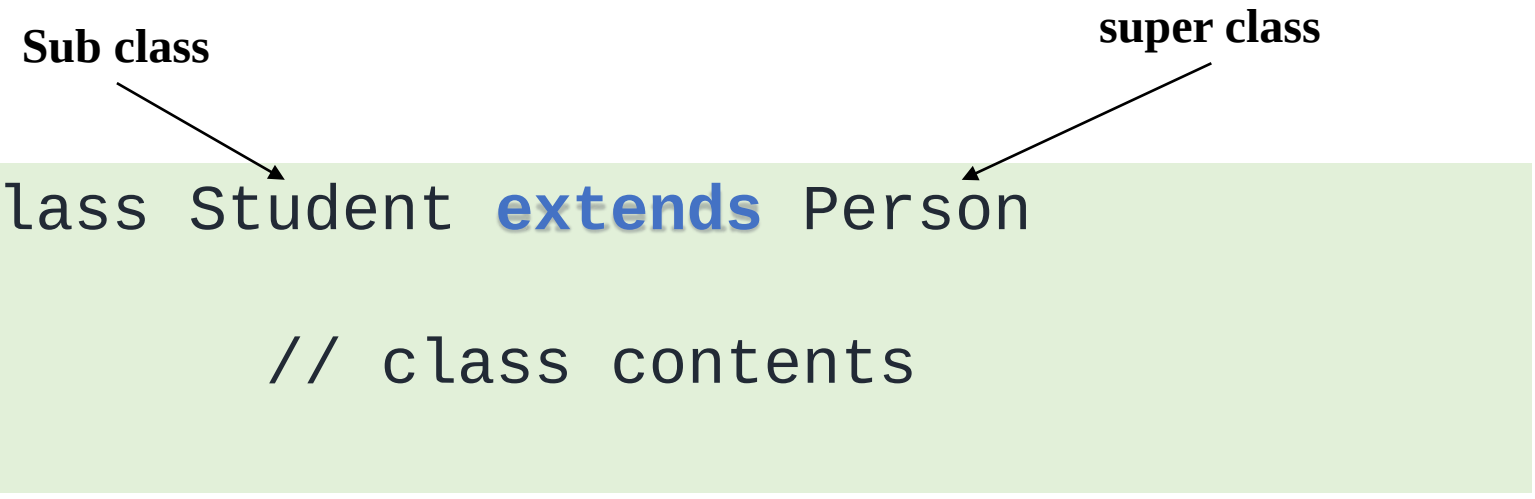
***Inheritance should create an is-a relationship, meaning the child is a more specific version of the parent***

# Deriving Subclasses

- In Java, we use the reserved word ***extends*** to establish an inheritance relationship

**Format**:

```
public class <Name of Subclass > extends <Name of Superclass>
{
    // Definition of subclass – only what is unique to subclass
}
```



# Example

```
public class Person{
    private int age;
    public Person(){age = 0;}
    public Person(int age){
        this.age = age;
    }
    public int getAge(){
        return age;
    }
    public void setAge(int age){
        this.age = age;
    }
    public void printData(){
        System.out.println("Age  = " +age);
    }
}
```

```
class Student extends Person{
    private int gpa;
    public Student(){
      setAge(0);
      gpa = 0;
    }
    public void printData(){
     System.out.println("Age=" + getAge());
     System.out.println("gpa = " + gpa);
    }
}
```

```
public class Main{
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.printData();
    }
}
```

# Private & public members

- A subclass is not a subset of its superclass.
  - A subclass usually contains more information and methods than its superclass.
- Private data fields in a superclass are not accessible outside the class.
  - They cannot be used directly in a subclass.
  - They can be accessed/mutated through public accessors/mutators if defined in the superclass.

# Single versus Multiple inheritance

- Some programming languages allow you to derive a subclass from several classes.
  - This capability is known as *multiple inheritance*.
  - Java, however, does not allow multiple inheritance.
- A Java class may inherit directly from only one superclass.
  - This restriction is known as *single inheritance*

# The super Keyword

- The keyword **super** refers to the superclass and can be used to invoke the superclass's methods and constructors.
- The **this** Reference, introduced to reference the calling object.
- The keyword **super** refers to the superclass of the class in which **super** appears.
- It can be used in two ways:
  - To call a superclass constructor.
  - To call a superclass method.

# The super Keyword

- A constructor is used to construct an instance of a class.
  - Unlike properties and methods, the constructors of a superclass are not inherited by a subclass.
  - They can only be invoked from the constructors of the subclasses using the keyword **super**.
- The syntax to call a superclass's constructor is:
  - **super**(), or **super**(parameters);

# Example

```
public class Person{
    private int age;
    public Person(){age = 0;}
    public Person(int age){
        this.age = age;
    }
    public int getAge(){
        return age;
    }
    public void setAge(int age){
        this.age = age;
    }
    public void printData(){
        System.out.println("Age  = " +age);
    }
}
```

```
class Student extends Person{
    private int gpa;
    public Student(){
      super();
      gpa = 0;
    }
    public void printData(){
      super.printData();
     System.out.println("gpa = " + gpa);
    }
}
```

```
public class Main{
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.printData();
    }
}
```

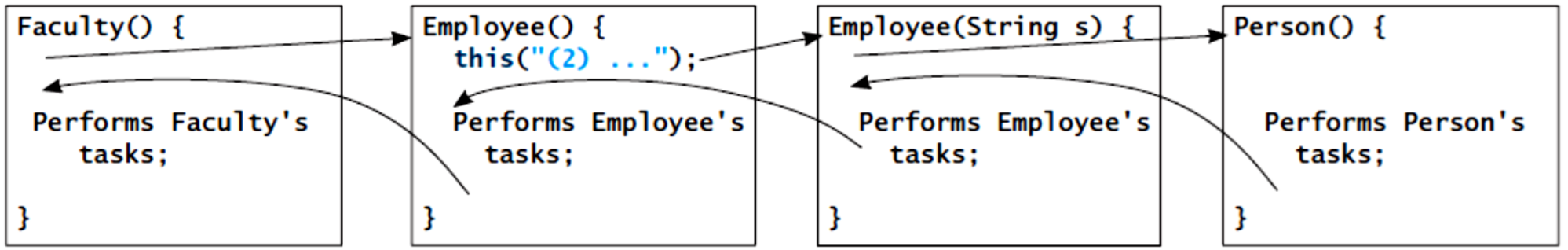
# Constructor Chaining

- A constructor may invoke an overloaded constructor or its superclass constructor.
- If neither is invoked explicitly,
  - the compiler automatically puts **super()** as the first statement in the constructor.
  - constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain.
- When constructing an object of a subclass,
  - the subclass constructor first invokes its superclass constructor before performing its own tasks.
  - If the superclass is derived from another class, the superclass constructor invokes its parent-class constructor before performing its own tasks.
  - This process continues until the last constructor along the inheritance hierarchy is called.
  - This is called *constructor chaining*.

# Constructor Chaining - Example

```
public class Faculty extends Employee {
    public static void main(String[] args) { new Faculty();  }
    public Faculty() { System.out.println("(4) Performs Faculty's tasks"); }
}
class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Performs Employee's tasks ");
    }
    public Employee(String s) { System.out.println(s);}
}
class Person {
    public Person() { System.out.println("(1) PerformsPerson's tasks"); }
}
```

(1) Performs Person's tasks
(2) Invoke Employee's overloaded constructor
(3) Performs Employee's tasks
(4) Performs Faculty's tasks



# Constructor Chaining

- No constructor is explicitly defined in **Apple**,
  - **Apple**'s default no-arg constructor is defined implicitly.
- **Apple** is a subclass of **Fruit**, **Apple**'s default constructor automatically invokes **Fruit**'s no-arg constructor.
  - However, **Fruit** does not have a no-arg constructor, because **Fruit** has an explicit constructor defined.
  - Therefore, the program cannot be compiled.

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

# Calling Superclass Methods

- The keyword **super** can also be used to reference a method other than the constructor in the superclass.
- **The syntax is:**

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

# Overriding Methods

- To override a method,
  - the method must be defined in the subclass using the same signature and the same return type as in its superclass.
- A subclass inherits methods from a superclass.
  - Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass.
  - This is referred to as *method overriding*.

# Example

```
public class Person{
    private int age;
    public Person(){age = 0;}
    public Person(int age){
        this.age = age;
    }
    public int getAge(){
        return age;
    }
    public void setAge(int age){
        this.age = age;
    }
    public void printData(){
        System.out.println("Age  = " +age);
    }
}
```

```
class Student extends Person{
    private int gpa;
    public Student(){
        gpa = 0;
    }
    public void printData(){
     super.printData();
     System.out.println("gpa = " + gpa);
    }
}
```

```
public class Main{
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.printData();
    }
}
```

# Overriding Methods

- An instance method can be overridden only if it is accessible.
  - A private method cannot be overridden, because it is not accessible outside its own class.
  - If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.
- Like an instance method, a static method can be inherited.
  - However, a static method cannot be overridden.
  - If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.

# Overriding vs. Overloading

- Overloading means to define multiple methods with the same name but different signatures.
- Overriding means to provide a new implementation for a method in the subclass.

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

# The Object Class and Its toString() Method

- *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. For example, the following two class definitions are the same:

```
public class ClassName {
  ...
}
```

Equivalent

```
public class ClassName extends Object {
  ...
}
```

- The signature of the **toString()** method is:

  **public** String toString()

- Invoking **toString()** on an object returns a string that describes the object.
- By default, it returns a string consisting of a class name of which the object is an instance, an at sign (@), and the object's memory address in hexadecimal.

# The Object Class and Its toString() Method

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

- The output for this code displays something like **Loan@15037e5**.
  - This message is not very helpful or informative.
  - You should override the **toString** method so that it returns a descriptive string representation of the object.

```
public String toString() {
        return "The person age is " + age;
}
```

# Assignment 4

II. Imagine a publishing company that markets both printed book and soft versions of its works. Create a class publication that stores the title (a string) and price (type float) of a publication. From this class derive two classes: book, which adds a page count (type int), and soundBook, which adds a playing time in minutes (type float). Each of these three classes should have a readData() method to get its data from the user at the keyboard, and a printData() method to display its data.

III. Write a main() program to test the hardBook and soundBook classes by creating instances of them, asking the user to fill in data with readData(), and then displaying the data with printData().

Thank you