Introduction:
Object Oriented Programming in Java

- Object Oriented Programming (OOP)
- Inheritance in OOP
- OO Programming - General Hints
- Introduction to Classes and Objects
- Subclasses and Inheritance
- Abstract Classes
- Single Inheritance versus Multiple Inheritance
- Interfaces
- Method Forwarding
- Overriding Methods

Object Oriented Programming (OOP)

In procedural programming languages (like 'C'), programming tends to be **action-oriented**, whereas in Java - programming is **object-oriented**.

In procedural programming,

groups of actions that perform some task are formed into functions and functions are grouped to form programs.

In OOP,

programmers concentrate on creating their own user-defined types called **classes**.

Each **class** contains **data** as well as the set of **methods** (procedures) that manipulate the data.

An instance of a user-defined type (i.e. a **class**) is called an **object**.

OOP encapsulates data (attributes) and methods (behaviours) into **objects**, the data and methods of an object are intimately tied together.

Objects have the property of *information hiding* .

Inheritance in OOP

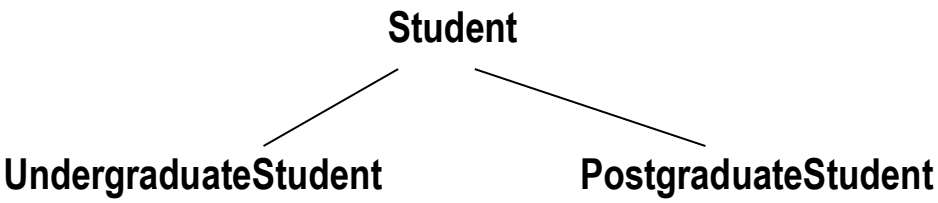
Inheritance is a form of software reusability in which new classes are created from the existing classes by absorbing their attributes and behaviours.

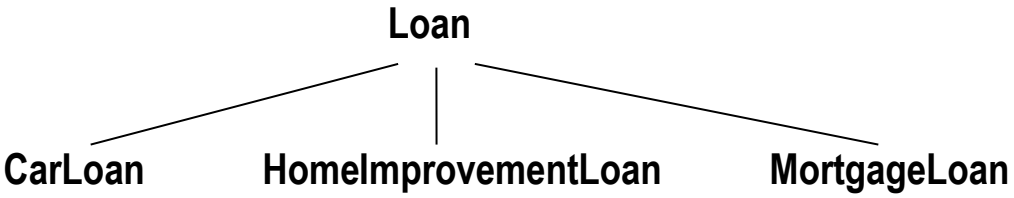
Instead of defining completely (separate) new class, the programmer can designate that the new class is to **inherit** attributes and behaviours of the existing class (called **superclass**). The new class is referred to as **subclass**.

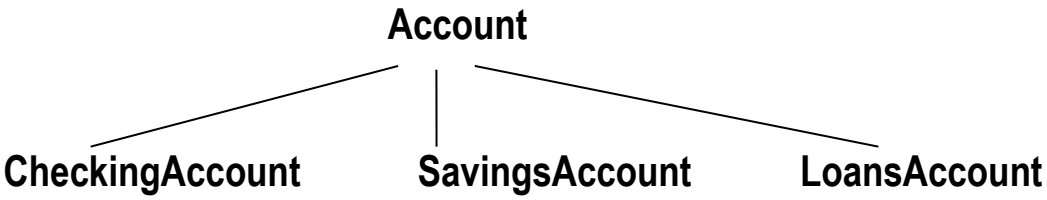
Programmer can add more attributes and behaviours to the **subclass**, hence, normally subclasses have more features than their **super classes**.

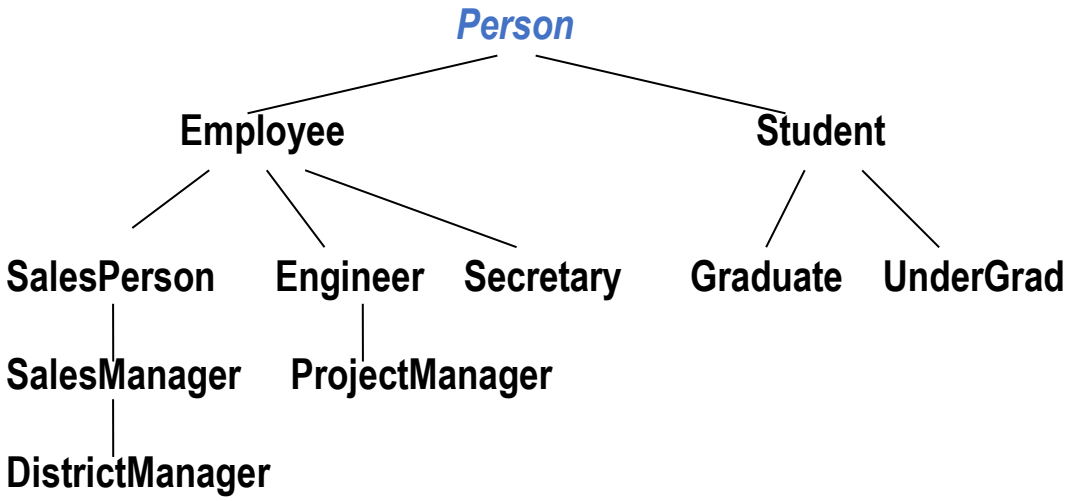
Inheritance relationships form tree-like hierarchical structures.

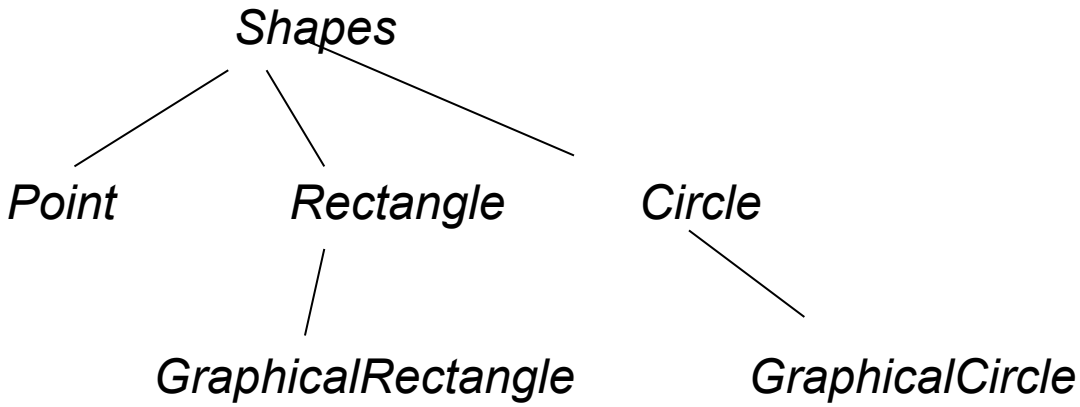
For example,








Person
Employee
Student
SalesPerson
Engineer
Secretary
Graduate
UnderGrad
SalesManager
ProjectManager
DistrictManager


Shapes
Point
Rectangle
Circle
GraphicalRectangle
GraphicalCircle

General Hints:

Designing Class Hierarchy

"Is-a" / Inheritance relationship

In an "is-a" relationship, an object of a subclass may also be treated as an object of the superclass.

For example, `UndergraduateStudent` can be treated as `Student` too.

You should use ***inheritance*** to model **"is-a" relationship**.

Important: Don't use inheritance unless **all or most** inherited attributes and methods make sense.

For example, mathematically a circle is-a (an) oval, however you should **not** inherit a class circle from a class oval. A class oval can have one method to set width and another to set height.

"Has-a" / Association relationship

In a "has-a" relationship, a **class object has an object of another class** to store its state or do its work, ie it "has-a" reference to that other object.

For example, a Rectangle Is-NOT-a Line. However, we may use a Line to draw a Rectangle.

The "has-a" relationship is quite different from an "is-a" relationship. **"Has-a" relationships** are examples of creating new classes by ***composition*** of existing classes (as oppose to extending classes).

Important:

- Getting "Is-a" versus "Has-a" relationships correct is both **subtle** and potentially **critical**. You should **consider** all **possible** future **usages** of the classes before finalising the hierarchy.
- It is possible that **obvious solutions may not work** for some applications.

Designing a Class

- Think carefully about the functionality (methods) a class should offer.
- Always **try to keep data private** (local).
- Consider **different ways** an object may be **created**.
- Creating an object may require different actions such as initializations.
- Always initialize data.
- If the object is no longer in use, free up all the associated resources.
- Break up classes with too many responsibilities.
- In OO, classes are often closely related. "**Factor out**" common attributes and behaviours and place these in a class. Then use inheritance to form subclasses, OR use method forwarding (discussed later).

Introduction to Classes and Objects

- A class is a collection of data and methods (procedures) that operate on that data.

For example,

a **circle** can be described by the **x, y position** of its centre and by its **radius**.

We can define some useful methods (procedures) for circles, compute **circumference**, compute **area**, check whether pointes are inside the circle, etc.

By defining the **Circle class** (as below), we can create a new data type.

```java
public class Circle  {

    protected static final double pi = 3.14159;
    protected  int  x, y;
    protected  int  r;

   // Very simple constructor
    public Circle(){
        this.x = 1;
        this.y = 1;
        this.r = 1;
    }
    // Another simple constructor
    public Circle(int x, int y, int r){
        this.x = x;
        this.y = y;
        this.r = r;
    }

    /**
     * Below, methods that return the circumference
     * area of the circle
     */
    public  double  circumference( ) {
        return 2 * pi * r ;
    }
    public  double  area ( ) {
        return  pi * r * r ;
    }

}
```

For simplicity, the methods for getter and setters are not added in the above code.

Objects are Instances of a class

In Java, objects are created by instantiating a class.

For example,

```
Circle  c ;
c =  new  Circle ( ) ;
```

Accessing Object Data

We can access data fields of an object.

For example,

```
Circle  c  =  new  Circle ( ) ;

// Initialize our circle to have centre (2, 5)
// and radius  1.
// Assuming, x, y and r are not private

c.x = 2;
c.y = 5;
c.r = 1;
```

Using Object Methods

To access the methods of an object, we can use the same syntax as accessing the data of an object:

```
Circle  c = new  Circle ( ) ;
double  a;

c.r = 2;       // assuming r is not private

a  =  c.area( );

//Note that its not :   a = area(c)
```

Subclasses and Inheritance

We want to implement **`GraphicalCircle.`**
This can be achieved in at least **3 different ways** ….

First approach ….

```
// The class of graphical circles

public class GraphicalCircle {
     int  x, y;
     int  r;
     Color  outline, fill;

     public  double  circumference( ) {
          return 2 * 3.14159 * r ;
     }
     public  double  area ( ) {
          return  3.14159 * r * r ;
     }

     public void draw(Graphics  g) {
          g.setColor(outline);
          g.drawOval(x-r, y-r, 2*r, 2*r);
          g.setColor(fill);
          g.fillOval(x-r, y-r, 2*r, 2*r);
     }
}
```

In this approach we are creating the new **separate class** for `GraphicalCircle` and **re-writing** the code already available in the class `Circle.`

Hence this approach is NOT elegant, in fact its the **WORST** possible solution.
Note again.. its the **WORST** possible solution ….

Second approach ……

We want to implement `GraphicalCircle` so that it can make use of the code in the class `Circle`.

```
public  class  GraphicalCircle2 {
    // here's the math circle
    Circle c;
    //The new graphics variables go here
    Color outline, fill;

    // Very simple constructor
    public GraphicalCircle2(){
       c = new Circle();
       this.outline = Color.black;;
   this.fill = Color.white;
    }
    // Another simple constructor
    public GraphicalCircle2(int x, int y,
                    int r, Color o, Color f){
        c = new Circle(x, y, r);
        this.outline = o; this.fill = f;
     }

    // Here are the old methods
    public double area() {
        return c.area();
    }
    public double circumference(){
        return c.circumference();
    }

    // draw method , using object 'c'
    public void draw(Graphics  g) {
        g.setColor(outline);
        g.drawOval(c.x-c.r, c.y-c.r, 2*c.r, 2*c.r);
        g.setColor(fill);
        g.fillOval(c.x-c.r, c.y-c.r, 2*c.r, 2*c.r);
    }
}
```

The above approach uses "**Has-a" relationship**. That means, a `GraphicalCircle` has a (mathematical) `Circle`.

It uses methods from the class `Circle` (area and circumference) to define some of the new methods. This technique is also known as ***method forwarding***.

It is possible in this example to use “Is-a” relationship instead of “Has-a” relationship (see third approach).

Third approach ……

Extending a Class

We can say that GraphicalCircle **is-a** Circle. Hence we can define `GraphicalCircle` as an *extension*, or *subclass* of `Circle`.

```
import java.awt.Color;
import java.awt.Graphics;

public class GraphicalCircle extends Circle {

    Color  outline, fill;
    public GraphicalCircle(){
            super();
            this.outline = Color.black;
            this.fill = Color.white;
    }
    // Another simple constructor
    public GraphicalCircle(int x, int y,
                    int r, Color o, Color f){
        super(x, y, r);
        this.outline = o; this.fill = f;
    }

    public void draw(Graphics  g) {
        g.setColor(outline);
        g.drawOval(x-r, y-r, 2*r, 2*r);
        g.setColor(fill);
        g.fillOval(x-r, y-r, 2*r, 2*r);
    }

}
```

The subclass `GraphicCircle` *inherits* all the variables and methods of its superclass `Circle`.

Example,

```
GraphicCircle  gc;
    ...
double  area = gc.area();
```

We can assign an instance of `GraphicCircle` to a `Circle` variable.

Example,

```
GraphicCircle  gc;
    ...
Circle  c = gc
```

Considering the variable "**c**" is of type "**Circle**",

- we can only access attributes and methods available in the class "**Circle**".
- we cannot call "**draw**" method for "**c**".

Superclasses, Objects, and the Class Hierarchy

Every class has a superclass.

If we don't define the superclass, by default, the superclass is the class **`Object`**.

`Object` Class:

- It's the only class that does not have a superclass.
- The methods defined by Object can be called by any Java object (instance).
- Often we need to override the following methods:
  - **`toString()`**
  - **`equals()`**
  - **`hasCode()`**

Abstract Classes

Using Abstract classes,
we can declare classes that define only part of an implementation, leaving extended classes to provide specific implementation of some or all the methods.

The benefit of an **`abstract`** class
is that methods may be declared such that the programmer knows the interface definition of an object,

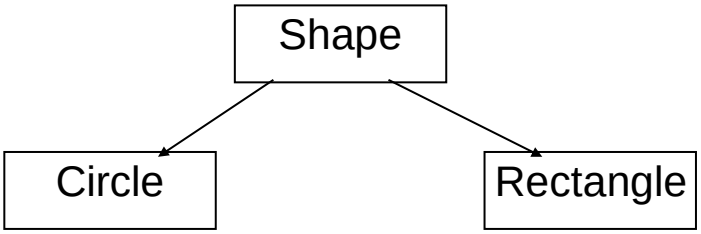
however, methods can be implemented differently in different subclasses of the `abstract` class.

Some rules about `abstract` classes:

- An abstract class is a class that is declared `abstract`.
- If a class includes `abstract` methods, then the class itself must be declared `abstract`.
- An `abstract` class cannot be instantiated.
- A subclass of an `abstract` class can be instantiated if it overrides each of the `abstract` methods of its superclass and provides an implementation for all of them.

- If a subclass of an `abstract` class does not implement all the `abstract` methods it inherits, that subclass is itself `abstract`.

For example,




```
public  abstract class Shape {

     public abstract double area();
     public abstract double circumference();

}
```

```
public class Circle  extends Shape {

    protected static final double pi = 3.14159;
    protected  int  x, y;
    protected  int  r;

   // Very simple constructor
    public Circle(){
        this.x = 1;
        this.y = 1;
        this.r = 1;
    }
    // Another simple constructor
    public Circle(int x, int y, int r){
        this.x = x;
        this.y = y;
        this.r = r;
    }

    /**
     * Below, methods that return the circumference
     * area of the circle
     */
    public  double  circumference( ) {
        return 2 * pi * r ;
    }
    public  double  area ( ) {
        return  pi * r * r ;
    }

}
```

```java
public class Rectangle extends Shape {

    protected double width, height;

    public Rectangle() {
            width = 1.0;
            height = 1.0;
    }

    public Rectangle(double w, double h) {
         this.width = w;
         this.height = h;
    }

    public double area(){
        return width*height;
    }

    public double circumference() {
            return 2*(width + height);
    }


}
```

We can now write code like this:

```
// create an array to hold shapes
Shape[] shapes = new Shape[4];
shapes[0] = new Circle(4, 6, 2);
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);
shapes[3] = new GraphicalCircle(1, 1, 6,
                Color.green, Color.yellow);

double total_area = 0;
for(int i = 0; i < shapes.length; i++) {
    // compute the area of the shapes
    total_area += shapes[i].area();
}
```

Some points to note:

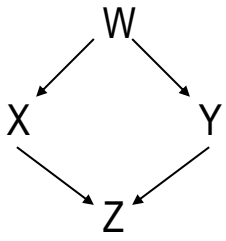
- As `Shape` is an abstract class, we cannot instantiate it.
- Instantiations of `Circle` and `Rectangle` can be assigned to variables of `Shape`. No cast is necessary
- In other words, subclasses of `Shape` can be assigned to elements of an array of `Shape`. No cast is necessary.
- We can invoke `area()` and `circumference()` methods for `Shape` objects.

Single Inheritance versus Multiple Inheritance

- In Java, a new class can extend exactly one superclass - a model known as *single inheritance*.

- Some object-oriented languages employ *multiple inheritance*, where a new class can have two or more *super classes*.

- In multiple inheritance, problems arise when a superclass's behaviour is inherited in two ways.



(also known as *diamond inheritance* problem)

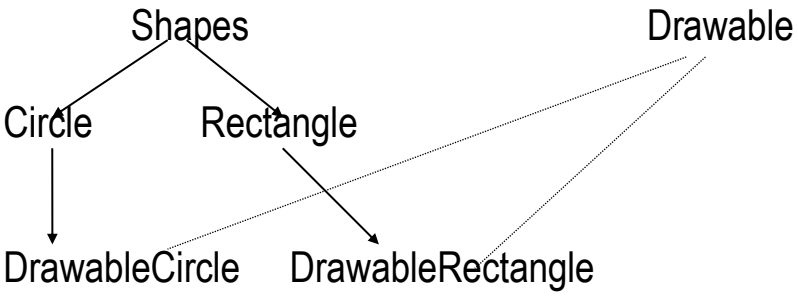
- Single inheritance precludes some useful and correct designs.

- In Java, **`interface`** in the class hierarchy can be used to add multiple inheritance.

Interfaces

- Interfaces are like `abstract` classes, but with few **important differences**.
- All the methods defined within an `interface` are implicitly `abstract`. (We don't need to use `abstract` keyword, however, to improve clarity one can use `abstract` keyword).
- Variables declared in an `interface` must be `static` and `final`, that means, they must be constants.
- Just like a class **`extends`** its superclass, it also can optionally **`implements`** an interface. In order to implement an interface, a class must first declare the interface in an **`implements`** clause, and then it must provide an implementation for all of the `abstract` methods of the interface.
- A class can **`implements`** more than one interfaces.

For example,




```
public  interface Drawable {
     public void setColor(Color c);
     public void setPosition(double x, double y);
     public void draw(Graphics  g);
}

public class DrawableRectangle
                  extends Rectangle
                  implements Drawable  {

     private  Color c;
     private double x, y;

          ......
          ......
     // Here are implementations of the
     // mehtods in Drawable
     // we also inherit all public methods
     // of Rectangle

     public void setColor(Color c) { this.c = c;}
     public void setPosition(double x, double y){
          this.x = x; this.y = y;}
     public void draw(Graphics g) {
          g.drawRect(x,y,w,h,c); }
}
```

Using Interfaces

- When a class implements an interface, instance of that class can also be assigned to variables of the interface type.

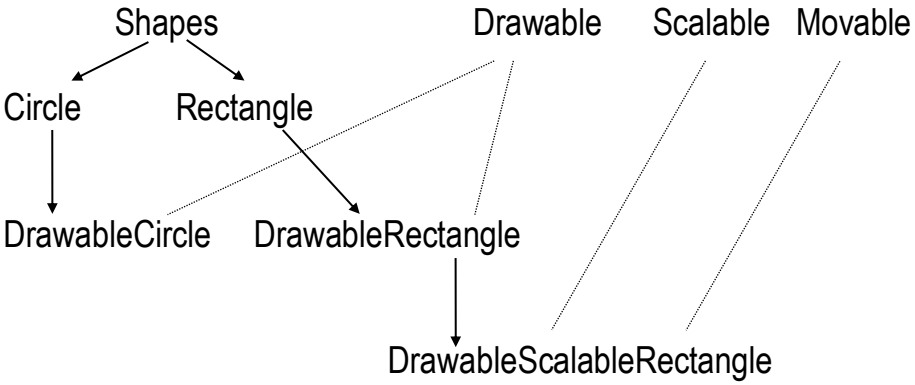
For example,

```
Shape[] shapes = new Shape[3];
Drawable[] drawables = new Drawable[3];


DrawableCircle dc = new DrawableCircle(1.1);
DrawableSquare ds = new DrawableSquare(2.5);
DrawableRectangle dr = new DrawableRectangle(2.3,
4.5);

// The shapes can be assigned to both arrays
shapes[0] = dc; drawables[0] = dc;
shapes[1] = ds; drawables[1] = ds;
shapes[2] = dr; drawables[2] = dr;

// We can invoke abstract method
// in Drawable and Shapes

double total_area = 0;
for(int i=0; i< shapes.length; i++) {

     total_area += shapes[i].area();

     drawables[i].setPosition(i*10.0, i*10.0);

     // assume that graphic area 'g' is
     // defined somewhere
     drawables[i].draw(g);
}
```

Implementing Multiple Interfaces

A class can `implements` more than one interfaces.

For example,




```
public class DrawableScalableRectangle
     extends DrawableRectangle
     implements  Movable,  Scalable {

//  methods go here ....

}
```

Extending Interfaces

Interfaces can have sub-interfaces, just like classes can have subclasses.

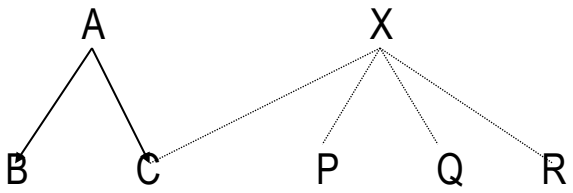
A sub-interface inherits all the abstract methods and constants of its super-interface, and may define new abstract methods and constants.

Interfaces can extend more than one interface at a time.

For example,

```
public interface Transformable
         extends Scalable,Rotable,Reflectable {}

public interface DrawingObject
         extends Drawable, Transformable{}

public class Shape implements DrawingObject {
              ... }
```

Method Forwarding



Suppose class C extends class A, and also implements interface X.

As all the methods defined in interface X are abstract, class C needs to implement all these methods.

However, there are three implementations of X (in P,Q,R).
In class C, we may want to use one of these implementations, that means, we may want to use some or all methods implemented in P, Q or R.

Say, we want to use methods implemented in P. We can do this by creating an object of type class P in class C, and through this object access all the methods implemented in P.

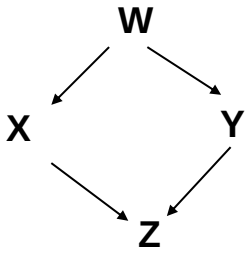
Note that, in class C, we do need to provide required stubs for all the methods in the interface X. In the body of the methods we may simply call methods of class P via the object of class P.

This approach is also known as **method forwarding**.

Another example,

In Java, one of the possible solutions for diamond inheritance problem is as shown below (note that we can resolve it in many ways),

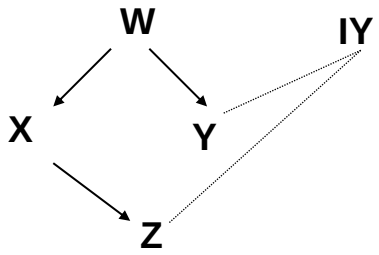
Using **multiple inheritance** (in C++):



we achieve the following:

- In class Z, we can use methods and variables defined in X, W **and** Y.
- Objects of classes Z and Y can be assigned to variables of **type Y.**
- and more ...

Using **single inheritance** in Java:




```
class W { }
interface IY { }
class X extends W { }
class Y extends W implements IY { }
class Z extends X implements IY { }
```

we achieve the following:

- In class Z, we can use methods and variables defined in X and W. In class Z, if we want to use methods implemented in class Y, we can use ***method forwarding*** technique. That means, in class Z, we can create an object of type class Y, and via this object we can access (in class Z) all the methods defined in class Y.
- Objects of classes Z and Y can be assigned to variables of **type IY** (instead of Y).
- and more ....

Methods Overriding

When a class defines a method using the same name, return type, and by the number, type, and position of its arguments as a method in its *superclass*, the method in the class **overrides** the method in the *superclass*.

If a method is invoked for an object of the class, it's the new definition of the method that is called, and not the superclass's old definition.

polymorphism

An object's ability to decide what method to apply to itself, depending on where it is in the inheritance hierarchy, is usually called ***polymorphism***.

In the example below,

- if **`p`** is an instance of class **`B`**,
  `p.f()` refers to **`f()`** in class **`B`**.
- However, if `p` is an instance of class **`A`**,
  `p.f()` refers to **`f()`** in class **`A`**.

The example also shows how to refer to the overridden method using **`super`** keyword.

```
class A {
     int i = 1;
     int f() { return i;}
}

class B extends A {
     int i;                              // shadows i from A
     int f() {                           // overrides f() from A
          i = super.i + 1;               // retrives i from A
          return  super.f() + i;         // invokes f() from A
     }
}
```

Another Example,

Suppose class C is a subclass of class B, and class B is a subclass of class A. Class A and class C both define method **`f().`**

From class C, we can refer to the overridden method by,

```
super.f()
```

This is because class B inherits method **`f()`** from class A.

However,

if **all the three** classes define **`f()`**, then calling **`super.f()`** in class C invokes class B's definition of the method.

Importantly, in this case, there is **no way** to invoke `A.f()` from within class C.

Note that **`super.super.f()`** is **NOT legal** Java syntax.

Data Hiding and Encapsulation

We can hide the data within the class and make it available only through the methods.

This can help in maintaining the consistency of the data for an object, that means the state of an object.

Visibility Modifiers

Java provides five access modifiers (for variables/methods/classes),

`public`	- visible to the world
`private`	- visible to the class only
`protected`	- visible to the package and all subclasses
No modifier (default)	- visible to the package

Method Overloading

Defining methods with the **same name** and **different** argument or return types is called method ***overloading***.

In Java,

a method is distinguished by its **method signature** - its name, return type, and by the number, type, and position of its arguments

For example,

```
double add(int, int)
double add(int, double)
double add(float, int)
double add(int, int, int)
double add(int, double, int)
```

Constructors

- Good practice to **define** the required constructors for **all** classes.

- **If** a constructor is **not defined** in a class,
  - **no-argument** constructor is implicitly **inserted**.
  - this no-argument constructor invokes the superclass's no-argument constructor.
  - if the parent class (superclass) doesn't have a visible constructor with no-argument, it results in a compilation **error**.

- If the **first statement** in a constructor is **not** a call to **super**() or **this**(), a call to **super** () is **implicitly** inserted.

- If a constructor is defined with one or more arguments, no-argument constructor is **not** inserted in that class.

- A class can have **multiple** constructors, with different *signatures.*

- The word "**this**" can be used to call another constructor in the same class.