Class 11 Notes: Polymorphism → many forms

```
public  abstract class MyPet
{
 protected String name;
 protected int age;
 // constructors  and getters and setters
 public String toString()
 {
  return "Name: "+ name + " age: "+ age;
 }}
```

```
public class MyDog extends
                        MyPet
{
 public MyDog(String name,
                    int age)
 {
   super(name, age);
 }
 public void speak()
 {
  System.out.println("Woof");
 }}
```

```
public class MyCat extends
                        MyPet
{
 public MyCat(String name,
                    int age)
 {
   super(name, age);
 }
 public void speak()
 {
  System.out.println ("Meow");
 }}
```

```
public class MyFrog extends
                        MyPet
{
 public MyFrog(String name,
                    int age)
 {
   super(name, age);
 }
 public void speak()
 {
  System.out.println ("Croak");
 }}
```

```
import java.util.*;
public class MyZoo
{
 public static void main(String[] args)
 {
  Random r = new Random();
  int n = r.nextInt(3); //0 1 or 2
  MyPet pet;  // declared type is MyPet
  if (n == 0)
  {
   pet= new MyDog("Fido", 4);
   pet.speak();
  }
  if (n == 1)
  {
   pet= new MyCat("Sylvester", 6);
   pet.speak();
  }
  if (n == 2)
  {
   pet= new MyTFrog("Michigan J. ", 1);
   pet.speak();
  }
 }
}}
```

Will This compile? Why or why not?

Here is one way to fix it ….**but there is a better way:**

Downcast:

```
import java.util.*;
public class MyZoo1
{
  public static void main(String[] args)
  {
    Random r = new Random();
    int n = r.nextInt(3);
    MyPet pet;
    if (n == 0)
    {
      pet= new MyDog("Fido", 4);
      ((MyDog)pet).speak();
    }
    if (n == 1)
    {
      pet= new MyCat("Sylvester", 6);
      ((MyCat)pet).speak();
    }
    if (n == 2)
    {
      pet= new MyFrog("Michigan J. ", 1);
     ((MyFrog) pet).speak();
    }
  }
}

}
```

```
public  abstract class MyPet1
{
  protected String name;
  protected int age;
  // constructors…getters and settere
  public String toString()
  {
    return "Name: "+ name + " age: "+ age;
  }
  public abstract void speak();
}
```

.. MyDog, MyCat and MyFrog as before

Now look at the same code. It will compile. Why?

```
import java.util.*;
public class MyZooX
{
  public static void main(String[] args)
  {
    Random r = new Random();
    int n = r.nextInt(3); //0 1 or 2
    MyPet1 pet;
    if (n == 0)
    {
      pet= new MyDogX("Fido", 4);
      pet.speak();
    }
    if (n == 1)
    {
      pet= new MyCatX("Sylvester", 6);
      pet.speak();
    }
    if (n == 2)
    {
      pet= new MyFrogX("Michigan j, ", 1);
      pet.speak();
    }
  }
}
```

The compiler is satisfied.

OK The code will compile. The compiler is satisfied. But which version of speak() will be chosen?
That decision is deferred until runtime.

When the program is run, the Java Virtual Machine chooses the version of speak() that corresponds to the object that was created by the "new" operator. In other words, the **JVM looks at the real type**. This is called runtime or dynamic binding.

There are three versions of speak() and when the program runs the "correct" version is chosen. What is the correct version…It is the version the corresponds to the object that was actually created. Usin the "new" operator

This is polymorphism. There are three forms and one is chosen.

So

The compiler checks that there will be a speak() but leaves the decision of which version to the Java Virtual Machine during the run of the program – runtime or dynamic binding.

speak() has three forms → poly(many) morphism(forms)

The basic idea is that the appropriate version of speak() is selected at RUNTIME and the choice is based on the **real type** of the calling object,that is, what kind of object was created. A dog? A cat? Or a frog?

Notice , it MyPet did not have the abstract method

```
public void abstract speak()
```

the compiler would have complained.

Here is another example:

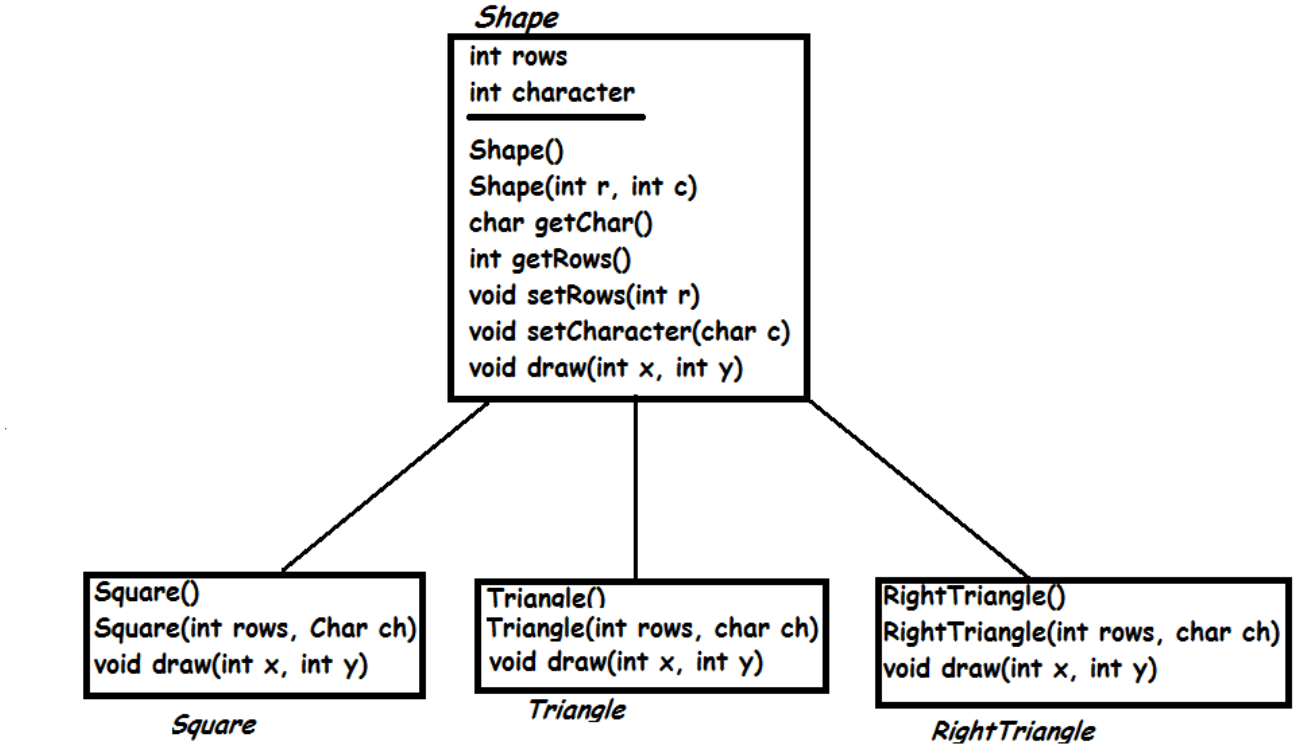
I want to write a program that draws some basic shapes such as a square, a triangle, or a right triangle. Here is a typical run:

```
Enter 1: Square, 2: RightTriangle, 3: Equilateral Triangle:  1
$$$$$$
$$$$$$
$$$$$$
$$$$$$
$$$$$$
$$$$$$
Enter 1: Square, 2: RightTriangle, 3: Equilateral Triangle:  2
#
##
###
####
#####
```

Here is some basic code that is used to accomplish this

```
 System.out.print("Enter 1: Square, 2: RightTriangle, 3: Equilateral Triangle: ");
  shapeNumber = input.nextInt()
   switch (shapeNumber)
   {
       case  1 : shape = new Square(6,'$');  //size 6, draw using $
                  break;
      case  2 : shape = new RightTriangle(5,'#');  //size 5, draw using  #
                 break;
      case  3 : shape = new Triangle(6,'+');  //size 6, draw using +
                  break;
           default : System.out.println("Invalid entry");  // shapeNumber is not 1,2,

        }

        shape.draw(1,1);
```

Now let's look at the classes that I used to do this



Shape is an abstract class. The method draw(..) is abstract.

Each of the subclasses of Shape will implement draw(..)
draw(x,y) → y means scroll down lines, xmeans indent x spaces.

So draw(5,3) means go down 3 lines (3 printlin()s) and indent 5 spaces.

```
public  abstract class Shape
{
    protected int rows;                    // figure drawn on rows rows
    protected char character;             // the drawing character

    public Shape()
    {
        rows  = 0;
        char character = ' ';
    }

    public Shape(int x, char ch)
    {
        rows = x;
        character = ch;
    }

    public int getRows()
    {
        return rows;
    }

    public char getCharacter()
    {
        return character;
    }
    public void setRows(int y)
    {
         rows = y;
    }
    public void setCharacter(char ch)
    {
         character = ch;
    }
   public abstract void   draw(int x, int y) // must be implemented in subclasses
}
```

```
public class RightTriangle
                        extends Shape
{
  public RightTriangle()
{
   super();
}

public RightTriangle(int x, char ch)
{
    super(x,ch);
}

public void   draw(int x, int y)
   {
      // move down y lines
      for ( int i = 1; i <= y; i++)
         System.out.println();

      // for each row
     for (int len = 1; len<= rows; len++)
     {
       //indent x spaces
        for (int i = 1; i <= x; i++)
            System.out.print(' ');
       for (int j = 1; j <= len; j++)
            System.out.print(character);
        System.out.println();
     }
   }
}
```

```
public class Triangle
            extends Shape
{
public Triangle ()
{
      super();
}

public Triangle (int x, char ch)
{
    super(x,ch);
}

public void   draw(int x, int y)
{
    // move down y lines
 for ( int i = 1; i <= y; i++)
       System.out.println();

// for each row
  for(int len=1; len<=rows; len++)
{
//indent; the vertex is centered

 for(int i=0; i <= rows-len+x; i++)
   System.out.print(" ");

for(int i =1; i<=len; i++)
  System.out.print(character +" " );
        System.out.println();
   }
 }
}
```

```
public class Square
                    extends Shape
{
public Square()
{
    super();
}

public Square(int x, char ch)
{
    super(x,ch);
}

public void   draw(int x, int y)
{
  // move down y lines
  for ( int i = 1; i <= y; i++)
         System.out.println();

  // for each row
for (int len = 1; len<=rows;len++)
{
   // indent x spaces
   for (int i = 1; i <= x; i++)
          System.out.print(' ');
   for(int  j = 1; j <=rows; j++)
      System.out.print(character);
       System.out.println();
   }
 }
}
```

```
import java.util.*;
public class TestDraw
{
    public static void main(String args[])
    {
        Scanner input = new Scanner(System.in);
        Shape shape = null;                         // all references can be upcast to Shape
        int shapeNumber;                            //code number for each type of figure
        System.out.print("Enter 1: Square, 2: RightTriangle, 3: Equilateral Triangle: ");
        shapeNumber = input.nextInt();

        switch (shapeNumber)
        {
            case  1 : shape = new Square(4,'*');          //size 4, draw with *
                      break;
            case  2 : shape = new RightTriangle(5,'#');           //size 5, draw with #
                      break;
            case  3 : shape = new Triangle(6,'+');         //size 6, draw with +
                      break;
            default : System.out.println("Invalid entry"); // shapeNumber is not 1,2, or 3
        }
          shape.draw(1,1);                          //which draw????????
    }
}
```

Let's look at the line

shape.draw(1,1);

When the program is compiled, the compiler sees that the **declared** type od shape is Shape.

So when the program is compiled, the compiler check the class Shaope and sees there is an abstract method draw(…). So the subclasses of Shape have a draw(..). The compiler is satisfied.

Obviously, the compiler cannot decide which draw method to use because that is determined at run time. When the program is run, version of draw(..) that is chosen depends on the kind of Shape object that was created using "new." That is dynamic or runtime binding.

There are three versions of draw(..). The one chosen depends on the kind of Shape object that was created when the program runs. Again→Runtime/dynamic binding

Another form of polymorphism is method overloading:

> Java allows two or more methods of the same class to share the same name. This practice is called *method overloading*.

For example, Java's Math class has several overloaded methods including Math.max(...), which has two forms:

1. int Math.max(int x, int y)
2. double Math.max (double x, double y)

Notice, that the parameter lists of the two methods differ. The first version of Math.max(...) accepts two integer parameters and the second version accepts two double parameters.

So the method **max(..) in the Math class has two forms – polymorphism**

In order for the Java compiler to distinguish between methods of the same name, overloaded methods **must** differ in the types and/or number of parameters.

But the appropriate overloaded method is chosen during compile time.
This is called static binding.

Here is a very simple example of overloaded operators:

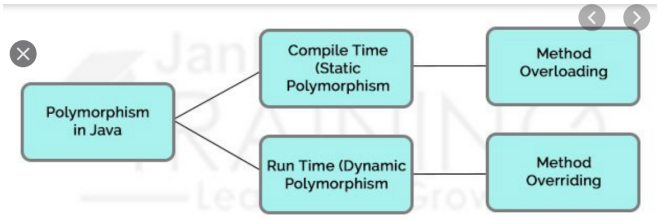
```
public class Arithmetic
{
  public int add(int x, int y)
  {
    System.out.println("I'm the integer version");
    return x + y;
  }

  public double add(double  x, double  y)
  {
    System.out.println("I'm the double version");
    return x + y;
  }

  public char add(char x)
  {
    System.out.println("I'm the char version");
    return (char) (x + 1);
  }

  public static void main(String[] args)
  {
    Arithmetic a = new Arithmetic();
    System.out.println(a.add(3,5));
    System.out.println(a.add(2.4,5.7));
    System.out.println(a.add('b'));
  }
}
```

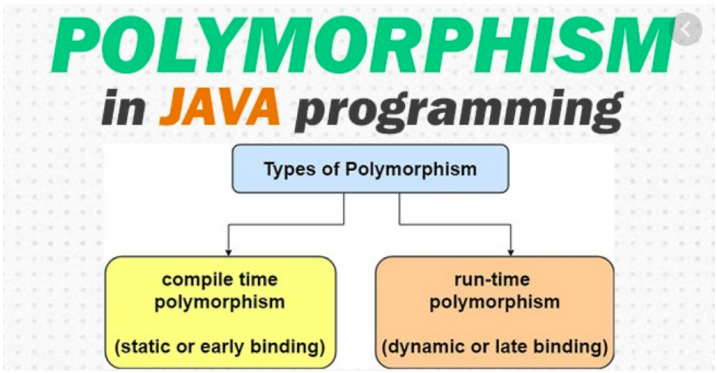
Output:
I'm the integer version 8
I'm the double version 8.1
I'm the char version c

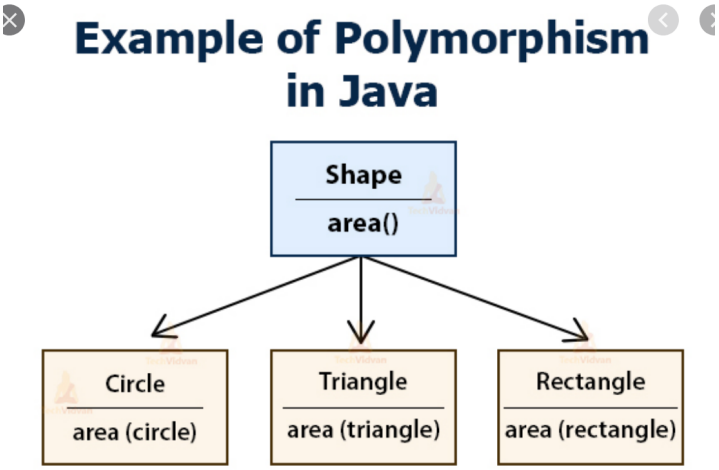
The compiler can decide which version to choose by the number and type of the parameters. The decision does not have to be postponed to runtime.

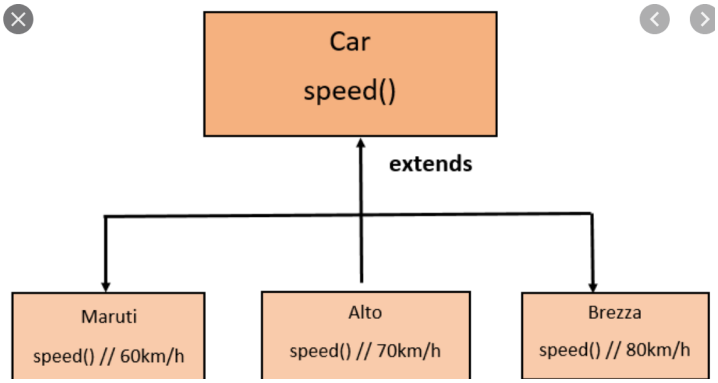
Method overloading is a simpler form of polymorphism.

Here are a few pictures from the web









Three principles of Object Oriented Programming:

1. Encapsulation
2. Inheritance
3. Polymorphism

Inheritance deals with similarities in a hierarchy –

Cat is-a Animal, Square is-a Shape

Polymorphism deals with differences in a hierarchy

Three different forms of draw() for example

method overloading (add(int, int), add(double, double), add(char)