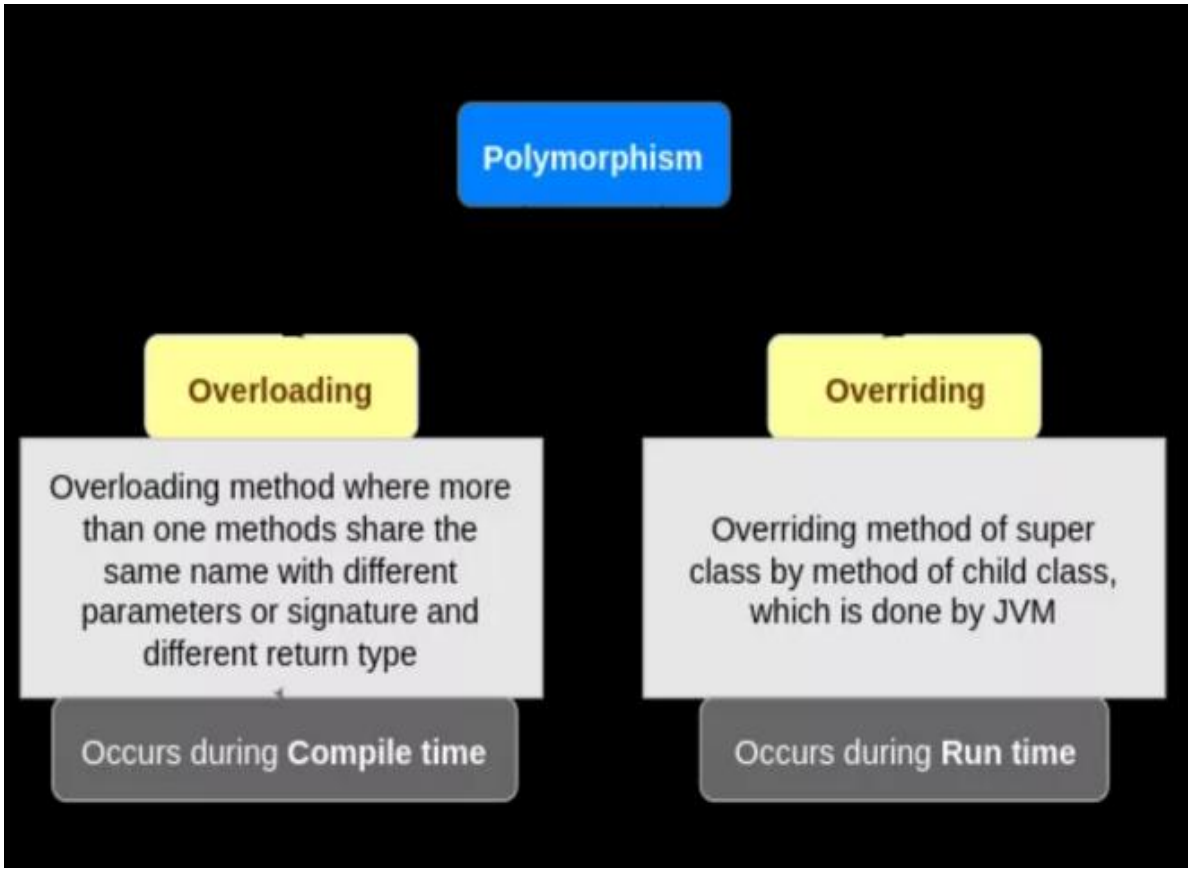
Polymorphism

Polymorphism

- Polymorphism means *many* (poly) *shapes* (morph)
- In Java, polymorphism is a concept in which we can perform a single action in different
- Polymorphism allows improved code organization and readability
- There are two kinds of polymorphism:
  - Overloading
    - Compile-time polymorphism
  - Overriding
    - Runtime polymorphism

Polymorphism



Overloading (Compile-time polymorphism)

- Overloading is a way to realize Polymorphism
- If you overload a static method, it is the example of compile time polymorphism.

Example :

```
class Test {
    public static void main(String args[]) {
        myPrint(5);
        myPrint(5.0);    }

    static void myPrint(int i) {
        System.out.println("int i = " + i);     }

    static void myPrint(double d) { // same name, different parameters
        System.out.println("double d = " + d);
    }
}
```

Output

```
int i = 5
double d = 5.0
```

Multiple constructors I

- You can “overload” constructors as well as methods:

```
Counter() {
    count = 0;
}

Counter(int start) {
    count = start;
}
```

Multiple constructors II

- One constructor can "call" another constructor in the same class, but there are special rules
  - You call the other constructor with the keyword this
  - The call must be the *very first thing* the constructor does
  - ```
    Point(int x, int y) {
        this.x = x;
        this.y = y;
        sum = x + y;
    }
    ```
  - ```
    Point() {
        this(0, 0);
    }
    ```
  - A common reason for overloading constructors is (as above) to provide default values for missing parameters

Superclass construction I

- The very first thing any constructor does, automatically, is call the *default* constructor for its superclass
  - class Foo extends Bar {
    Foo() { // constructor
    super(); // ***invisible*** call to superclass constructor
    ...
- You can replace this with a call to a *specific* superclass constructor
  - Use the keyword super
  - This must be the *very first thing* the constructor does
  - class Foo extends Bar {
    Foo(String name) { // constructor
    super(name, 5); // ***explicit*** call to superclass constructor
    ...

Superclass construction II

- Unless you specify otherwise, every constructor calls the *default* constructor for its superclass
  - class Foo extends Bar {
    Foo() { // constructor
    super(); // ***invisible*** call to superclass constructor
    …
- You can use this(…) to call another constructor in the same class:
  - class Foo extends Bar {
    Foo(String message) { // constructor
    this(message, 0, 0); // your ***explicit*** call to another constructor
    …
- You can use super(…) to call a specific *superclass* constructor
  - class Foo extends Bar {
    Foo(String name) { // constructor
    super(name, 5); // your ***explicit*** call to some superclass constructor
    …
- Since the call to another constructor must be the *very first thing you do* in the constructor, you can only do *one* of the above

Shadowing

```
class Animal {
    String name = "Animal";
    public static void main(String args[]) {
        Animal animal = new Animal();
        Dog dog = new Dog();
        System.out.println(animal.name + " " + dog.name);
    }
}

public class Dog extends Animal {
    String name = "Dog";
}
```

Animal Dog

- This is called shadowing—name in class Dog shadows name in class Animal

Overriding (Runtime Polymorphism)

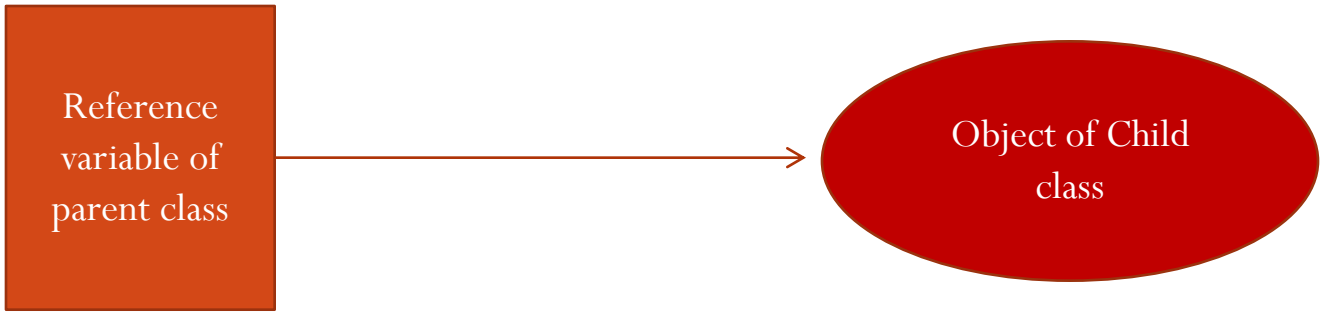
- Runtime polymorphism or dynamic binding is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

- In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

- Let us revisit upcasting before runtime polymorphism.

Upcasting

- If the reference variable of parent class refers to the object of child class, it is known as upcasting. For Example:



- Class A{ }
- Class B extends A { }
- A a= new B (); //upcasting

Runtime Polymorphism without upcasting

```
class Animal {
    public static void main(String args[]) {
        Animal animal = new Animal();
        Dog dog = new Dog();
        animal.print();
        dog.print();
    }
    void print() {
        System.out.println("Superclass Animal");
    }
}

public class Dog extends Animal {
    void print() {
        System.out.println("Subclass Dog");
    }
}
```

Superclass Animal
Subclass Dog

- This is called overriding a method.
- Method print in Dog overrides method print in Animal
- A subclass variable can *shadow* a superclass variable, but a subclass method can *override* a superclass method

Runtime Polymorphism with upcasting

Example 1: In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime. Example:

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

Output

running safely with 60km

Another Example

```
class Shape{
void
draw(){System.out.println("drawing...");}
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing
rectangle...");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing
circle...");}
}
class Triangle extends Shape{
void draw(){System.out.println("drawing
triangle...");}
}
```

```
class TestPolymorphism2{
public static void main(String args[]){
Shape s;
s=new Rectangle();
s.draw();
s=new Circle();
s.draw();
s=new Triangle();
s.draw();
}
}
```

Output

```
drawing rectangle...
drawing circle...
drawing triangle...
```

How to override a method

- Create a method in a subclass having the same *signature* as a method in a superclass

- That is, create a method in a subclass having the same name and the same number and types of parameters
  - Parameter *names* don't matter, just their *types*

- Restrictions:
  - The return type must be the same
  - The overriding method cannot be *more private* than the method it overrides

Calling an overridden method

- When your class overrides an inherited method, it basically "hides" the inherited method

- Within this class (but not from a different class), you can still call the overridden method, by prefixing the call with super.
  - Example: super.printEverything();

- You would most likely do this in order to observe the DRY principle
  - The superclass method will do most of the work, but you add to it or adjust its results
  - This isn't a call to a constructor, and can occur anywhere in your class (it doesn't have to be first)

Overload Vs. Override

Parameter	Overloading	Overriding
Resolved by	Compiler	JVM
Resolved during	Compile time	Run Time
Is achieved By	Function overloading or parameter overloading	Virtual methods and references to object instances
Effect on execution time	Fast (because it is done at compile time only once	Slow (because it is done at run time whenever overriding happens)
Other names	Early Binding (or) Static Binding	Late Binding (or) Dynamic binding

Instanceof Operator

- The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).
- Example 1:

```
class Simple1{
  public static void main(String args[]){
    Simple1 s=new Simple1();
    System.out.println(s instanceof Simple1);//true
 }
}
```

Output
true

Instanceof

- Example 2: An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

```
class Animal{   }
class Dog1 extends Animal{//Dog inherits Animal
        public static void main(String args[]){
 Dog1 d=new Dog1();
        System.out.println(d instanceof Animal);//true
        }
}
```

Output
true

Abstract Classes

- A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- Abstraction lets you focus on what the object does instead of how it does it.

- **Ways to achieve Abstraction**
  - There are two ways to achieve abstraction in java
    - Abstract class (0 to 100%)
    - Interface (100%)

Abstract Classes

- **Abstract class in Java**
- A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.
- **Points to Remember**
  - An abstract class must be declared with an abstract keyword.
  - It can have abstract and non-abstract methods.
  - It cannot be instantiated.
  - It can have constructors and static methods also.
  - It can have final methods which will force the subclass not to change the body of the method.

Abstract Classes

- Abstract class

  abstract class A{}

- Abstract Method: A method which is declared as abstract and does not have implementation is known as an abstract method.

  abstract void printStatus();//no method body and abstract

Example of Abstract class that has an abstract method

- In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{
    abstract void run();
}
class Honda4 extends Bike{
    void run(){System.out.println("running safely");}
    public static void main(String args[]){
    Bike obj = new Honda4();
    obj.run();
}
}
```

Output
running safely

Another Example

```
abstract class Shape{
abstract void draw();
} //In real scenario, implementation is provided by others i.e.unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}   //In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In a real scenario, object is provided through method, e.g.
    , getShape() method
s.draw();
}
```