<u>Polymorphism</u> - Polymorphism refers to the ability of OOPs programming languages to differentiate between entities with the same name efficiently. This is done by Java with the help of the signature and declaration of these entities.

Example:

```
// Overloaded sum().
// This sum takes two int parameters
public int sum(int x, int y) {
    return (x + y);
}
// Overloaded sum().
// This sum takes three int parameters
public int sum(int x, int y, int z) {
    return (x + y + z);
}
// Overloaded sum().
// This sum takes two double parameters
public double sum(double x, double y) {
    return (x + y);
}
```

Polymorphism in Java are mainly of 2 types:

1. **<u>Overloading</u>** –

➢ It allows different methods to have the same name, but different signatures. It can be done by -

- The <u>number of parameters</u> in two methods.
- The <u>data types of the parameters</u> of methods.
- The <u>order of the parameters</u> of methods.

➢ We cannot overload by return type. This behavior is same in C++.

➢ We can overload **static** methods.

➢ We **cannot** overload two methods in Java if they **differ only by static keyword** (number of parameters and types of parameters is same).

➢ We **can** overload **main()** in Java.

➢ Unlike C++, Java **doesn't allow user-defined overloaded operators.** Internally it overloads operators, like, + is overloaded for concatenation.

2. **<u>Overriding</u>** -

➢ Feature that allows a child class to **provide a specific implementatio**n of a method that is **already provided by one of its parent classes**.

➢ Overridden method in child class has the same name, same parameters, and **same return type** as a method in its super-class.

➢ Method overriding is one of the way by which java achieve Run Time Polymorphism.

➢ The version of a method that is executed will be determined by the object that is used to invoke it.

➢ If an **object of a parent class** is used to invoke the method, then the version in the parent class will be executed and vice-versa for **object of child class**.

➢ In other words, it is the **type of the object** being referred to (**not the type of the reference variable**) that determines which version will be executed.

➢ Example of Overriding and Run Time Polymorphism:

```
Class Parent { // Base Class
   void show()  {
      System.out.println("Parent's show()");
   }
}
class Child extends Parent { // Inherited class
   // This method overrides show() of Parent
   void show() {
      System.out.println("Child's show()");
   }
}
class Main {
   public static void main(String[] args) {
      // If a Parent type reference refers to a Parent object, then Parent's
      // show is called
      Parent obj1 = new Parent();
      obj1.show();
      // If a Parent type reference refers to a Child object Child's show()
      // is called. This is called RUN TIME POLYMORPHISM.
      Parent obj2 = new Child();
      obj2.show();
   }
}
```

Output:

```
Parent’s show()
Child’s show()
```

➢ The **access modifier** for an overriding method **can allow more**, **but not less**, access than the overridden method. Doing so (providing less access), will generate compile-time error. For example, a **protected method** in the **super-class** can be made **public, but not private,** in the subclass.

➢ **Final methods can not be overridden.** If we don’t want a method to be overridden, we declare it as final.

➢ When you define a static method with same signature as a static method in base class, it is known as method hiding (i.e. **static methods can’t be overridden**).

➢ **Summary** of what happens when we define a method with **same signature in child class** in different cases:-

	SUPERCLASS INSTANCE METHOD	SUPERCLASS STATIC METHOD
SUBCLASS INSTANCE METHOD	Overrides	Generates a compile-time error
SUBCLASS STATIC METHOD	Generates a compile-time error	Hides

- **Example of Method Hiding:-**

```
class Parent {
    // Static method in base class which will be hidden in subclass
    static void m1()  {
        System.out.println("From parent " + "static m1()");
    }

    // Non-static method which will be overridden in derived class
    void m2()   {
        System.out.println("From parent " + "non-static(instance) m2()");
    }
}

class Child extends Parent {
    // This method hides m1() in Parent
    static void m1()  {
        System.out.println("From child static m1()");
    }

    // This method overrides m2() in Parent
    public void m2()  {
        System.out.println("From child "  + "non-static(instance) m2()");
    }
}
class Main {
    public static void main(String[] args) {
        Parent obj1 = new Child();
        // As per overriding rules this should call to class Child static overridden method. Since static
        // method can not be overridden, it calls Parent's m1()
        obj1.m1();
        obj1.m2(); // Here overriding works and Child's m2() is called
    }
}
```

Output:

```
From parent static m1()
From child non-static(instance) m2()
```

- Private methods **can not be overridden**. It is so because **private methods are bonded during compile time** (No Run Time Polymorphism) and it is the **type of the reference variable** – not the type of object that it refers to – that determines what method to be called. This behavior is **different from C++**. In C++, we can have **virtual private methods.**

- The overriding method must have **same return type** (or **subtype**).

- We can call parent class method inside the definition of overriding method using **super** keyword. Example

```
void show()  {
    super.show();  // Calling Overridden Parent Class Method.
    System.out.println("Child's show()");
```

}

- We **can not override constructor** as parent and child class can never have constructor with same name.

- **Overriding and Exception-Handling:**
  Rule 1: If the overridden method **does not throws an exception**, overriding method can only throws the **unchecked exception**. Throwing checked exception will lead to compile-time error.
  Rule 2: If the overridden method **does throws an exception**, overriding method can only throw **same or a subclass exception**. Throwing parent exception in Exception hierarchy will lead to compile time error. Also there is no issue if subclass overridden method is not throwing any exception.

- **Abstract methods** in an interface or abstract class **must be overridden** in **derived concrete classes** otherwise a compile-time error will be thrown.

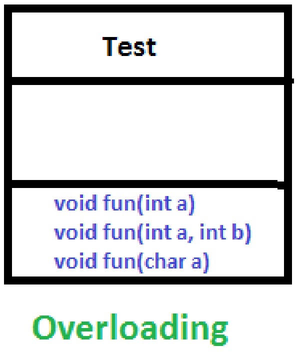
- The presence of **synchronized/strictfp** modifier with method have **no effect** on the rules of overriding.

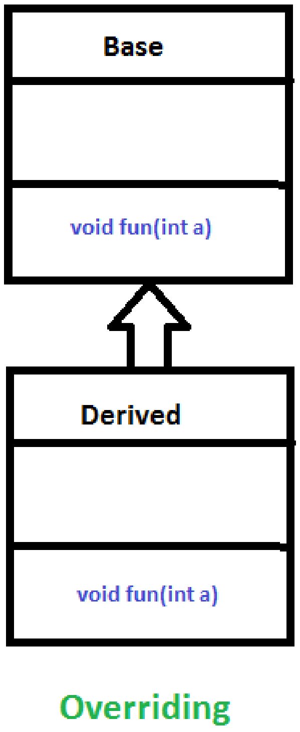
- In **C++**, we need **virtual keyword** to achieve **overriding** or **Run Time Polymorphism**. In **Java**, methods are virtual by **default**.

- We can have multilevel method-overriding i.e. GrandChildren can also override original Parent's methods.

Difference b/w Overloading and Overriding -

Overloading	Overriding
1. Example of Compile-time Polymorphism. 2. It is about same method have **different signatures**.	1. Example of Run-time Polymorphism. 2. It is about **same** method, same signature but **different classes** connected through inheritance.





Run Time Polymorphism (Dynamic Method Dispatch) – It is the mechanism by which a call to an overridden method is resolved at run time by JVM, rather than compile time.

Upcasting - A superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time.
Example:

```
// Inside main() method
Parent obj = new Child(); // Reference of Parent class is referring to object of Child Class.
```

- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

- In Java, we can **override methods only**, **not the variables**(data members), so runtime polymorphism cannot be achieved by data members.

Example:

```
// Java program to illustrate the fact that runtime polymorphism cannot be achieved by data members
class A {
   int x = 10;
}
class B extends A {
   int x = 20;
}
public class Test {
   public static void main(String args[])  {
       A a = new B(); // object of type B
       System.out.println(a.x); // Data member of class A will be accessed
   }
}
```

Output:

```
10
```

Advantages of Dynamic Method Dispatch -

- Dynamic method dispatch allow Java to support overriding of methods which is central for run time polymorphism.
- It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

Compile Time Polymorphism – In this an object is bound with their functionality at the compile-time. Also k/a or static or early binding.

- Method Overloading / Operator Overlaoding are example of compile time polymorphism.
- At compile-time, java knows which method to call by checking the method signatures.
- **private**, **final** and **static methods** and **variables** uses static binding and bonded by compiler.
- It is **Faster** than Run Time Polymorphism.
- It is **less flexible** than Run Time Polymorphism.