Compile Time Polymorphism: Whenever an object is bound with its functionality at the compile time, this is known as the compile-time polymorphism. At compile-time, java knows which method to call by checking the method signatures. So this is called compile-time polymorphism or static or early binding. Compile-time polymorphism is achieved through method overloading. Method Overloading says you can have more than one function with the same name in one class having a different prototype. Function overloading is one of the ways to achieve polymorphism but it depends on technology and which type of polymorphism we adopt. In java, we achieve function overloading at compile-Time. The following is an example where compile-time polymorphism can be observed.

```
// compile-time polymorphism
public class A {
// First addition function
        public static int add(int a, int b)
        {
                return a + b;        }
// Second addition function
        public static double add(
                double a, double b)
        {
                return a + b;
}
```

```
// Driver code
    public static void main(String args[])
    {
        // Here, the first addition
        // function is called
        System.out.println(add(2, 3));

        // Here, the second addition
        // function is called
        System.out.println(add(2.0, 3.0));
    }
}
```

- **Run-Time Polymorphism:** Whenever an object is bound with the functionality at run time, this is known as runtime polymorphism. The runtime polymorphism can be achieved by **method overriding**. **Java virtual machine** determines the proper method to call at the runtime, not at the compile time. It is also called dynamic or late binding. Method overriding says the child class has the same method as declared in the parent class. It means if the child class provides the specific implementation of the method that has been provided by one of its parent classes then it is known as method overriding. The following is an example where runtime polymorphism can be observed.

```
// Java program to demonstrate
        }
}
// runtime polymorphism

// Implementing a class
class Test {

        // Implementing a method
        public void method()
        {
                System.out.println("Method 1");
        }
}
```

```
// Defining a child class
public class GFG extends Test {

    // Overriding the parent method
    public void method()
    {
        System.out.println("Method 2");
    }

    // Driver code
    public static void main(String args[])
    {
        Test test = new GFG();

        test.method();
```

Sr. No.	Key	Compile-time polymorphism	Runtime polymorphism
1	Basic	Compile time polymorphism means binding is occuring at compile time	R un time polymorphism where at run time we came to know which method is going to invoke
2	Static/Dynamic Binding	It can be achieved through static binding	It can be achieved through dynamic binding
4.	Inheritance	Inheritance is not involved	Inheritance is involved
5	Example	Method overloading is an example of compile time polymorphism	Method overriding is an example of runtime polymorphism

- When the method signature (name and parameters) are the same in the superclass and the child class, it's called *overriding*. When two or more methods in the same class have the same name but different parameters, it's called *overloading*.

Overriding	**Overloading**
Implements "runtime polymorphism"	Implements "compile time polymorphism"
The method call is determined at runtime based on the object type	The method call is determined at compile time
Occurs between superclass and subclass	Occurs between the methods in the same class
Have the same signature (name and method arguments)	Have the same name, but the parameters are different
On error, the effect will be visible at runtime	On error, it can be caught at compile time

- **Overriding and overloading example**

```
import java.util.Arrays;

public class Processor {

        public void process(int i, int j) {
                System.out.printf("Processing two integers:%d, %d", i, j);          }

        public void process(int[] ints) {
                System.out.println("Adding integer array:" + Arrays.toString(ints));
        }

        public void process(Object[] objs) {
                System.out.println("Adding integer array:" + Arrays.toString(objs));
        }
}
```

```
class MathProcessor extends Processor {

        @Override
        public void process(int i, int j) {
                System.out.println("Sum of integers is " + (i + j));
        }

@Override       public void process(int[] ints) {
                int sum = 0;
                for (int i : ints) {
                        sum += i;
                }                       System.out.println("Sum of integer array elements is " + sum);
        }

}
```

- **Overriding**
- The process() method and int i, int j parameters in Processor are overridden in the child class MathProcessor. Line 7 and line 23:

```
public class Processor {
 public void process(int i, int j) { /* ... */ }


}
/* ... */

class MathProcessor extends Processor {

 @Override
   public void process(int i, int j) {  /* ... */ }

}
```

- And process() method and int[] ints in Processor are also overridden in the child class. Line 11 and line 28:

```
public class Processor {
 public void process(int[] ints) { /* ... */ }

}

/* ... */

class MathProcessor extends Processor {

    @Override
    public void process(Object[] objs) { /* ... */ }

}
```

- **Overloading**
- The process() method is overloaded in the Processor class. Lines 7, 11, and 15:
- public class Processor {
- public void process(int i, int j) { /* ... */ }
- public void process(int[] ints) { /* ... */ }
- public void process(Object[] objs) { /* ... */ }
- }

- **Conclusion**
- In this article, we covered overriding and overloading in Java. Overriding occurs when the method signature is the same in the superclass and the child class. Overloading occurs when two or more methods in the same class have the same name but different parameters.

Processor.java

```java
package com.journaldev.examples;

import java.util.Arrays;

public class Processor {

    public void process(int i, int j) {
        System.out.printf("Processing two integers:%d, %d", i, j);
    }

    public void process(int[] ints) {
        System.out.println("Adding integer array:" + Arrays.toString(ints));
    }

    public void process(Object[] objs) {
        System.out.println("Adding integer array:" + Arrays.toString(objs));
    }
}

class MathProcessor extends Processor {

    @Override
    public void process(int i, int j) {
        System.out.println("Sum of integers is " + (i + j));
    }

    @Override
    public void process(int[] ints) {
        int sum = 0;
        for (int i : ints) {
            sum += i;
        }
        System.out.println("Sum of integer array elements is " + sum);
    }

}
```

Overloading: Same Method Name but different parameters in the same class

Overriding: Same Method Signature in both superclass and child class

1

2

1

2