Objective:

- To understand the concept of Method Overloading and Method Overriding
- To learn about Polymorphism in Java
- To understand the difference between two types of polymorphism: Compile Time Polymorphism and Runtime Polymorphism

Method Overloading: Method overloading means to define multiple methods with the same name but different signatures, means different parameter list. It is not necessary that overloaded methods have to be in the same class, it can be in different classes related by inheritance. There are two ways to overload the method in java: *(i) By changing the number of arguments, (ii) By changing the data type of the parameters*. Different types of method overloading techniques are shown below.

(i) By changing number of arguments

```
public class Adder {
	public void sum(int a, int b) {
		System.out.println(a + "+" + b + " = " + (a+b));
	}

	public void sum(int a, int b, int c) {
		System.out.println(a + "+" + b + "+" + c + " = " + (a+b+c));
	}
}
```

```
public class Main {

	public static void main(String[] args) {
		Adder add = new Adder();
		add.sum(10, 20);
		add.sum(10, 20, 30);
	}
}
```

(ii) By changing the data type of the parameters

```
public class Adder {
	public void sum(int a, int b) {
		System.out.println(a + "+" + b + " = " + (a+b));
	}

	public void sum(double a, double b) {
		System.out.println(a + "+" + b + " = " + (a+b));
	}
}
```

```java
public class Main {

	public static void main(String[] args) {
		Adder add = new Adder();
		add.sum(10, 20);
		add.sum(10.5, 20.2);
	}
}
```

Method Overloading in different classes

```java
public class Greeting1 {
	public void hello() {
		System.out.println("Hello!");
	}
}
```

```java
public class Greeting2 extends Greeting1 {
	public void hello(String name) {
		System.out.println("Hello, " + name + "!");
	}
}
```

```java
public class GreetingMain {

	public static void main(String[] args) {
		Greeting2 g = new Greeting2();
		g.hello();
		g.hello("Biplob");
	}
}
```

Method Overriding: Method overriding means to provide a new implementation for a method in the subclass. In more simplified form, if a subclass provides the specific implementation of the method that has been declared by one of its parent classes, it is known as method overriding. Overridden methods exist in different classes related to inheritance. An example of method overriding is shown below.

```java
public class Doctor {
	public void treatPatient() {
		System.out.println("Doctor treated the patient!");
	}
}
```

```java
public class Surgeon extends Doctor {
	@Override
	public void treatPatient() {
		System.out.println("Surgeon performed the patient operation!");
	}
}
```

```java
public class Main {

	public static void main(String[] args) {
		Doctor doc = new Doctor();
```

```
            doc.treatPatient();
            Surgeon sur = new Surgeon();
            sur.treatPatient();
      }
}
```

Overloading vs Overriding: The main difference between overloading and overriding given below,

- Overloaded methods can be either in the same class or different classes related by inheritance; overridden methods are in different classes related by inheritance.
- Overloaded methods have the same name but a different parameter list; overridden methods have the same signature and the return type.

A special Java syntax called override annotation has been used to place @Override before the method in the subclass to avoid mistakes.

Polymorphism:

The word *Polymorphism* is derived from two Greek words: *poly* means many, and *morphs* means forms. So, Polymorphism means *'many forms'*. In simple words, Polymorphism is a concept by which specific action can be performed, but in different ways. Polymorphism in Java has two types:

- *Compile Time Polymorphism* or *Static Polymorphism* or *Static Binding*
- *Runtime Polymorphism* or *Dynamic Polymorphism* or *Dynamic Binding* or *Dynamic Method Dispatch*

Compile Time Polymorphism: In Java, *static polymorphism* can be achieved through *method overloading*. Java knows which method to invoke at compile time by checking or matching the method signatures. This means, the compiler search for a matching method according to the parameter type, number of parameters, and order of the parameters at compile time. And this is called *compile-time polymorphism* or *static binding*. An example of a *static polymorphism* is given below.

Adder.java

```
public class Adder {
      public void sum(int a, int b) {
            System.out.println(a + "+" + b + " = " + (a+b));
      }

      public void sum(int a, int b, int c) {
            System.out.println(a + "+" + b + "+" + c + " = " + (a+b+c));
      }

      public void sum(double a, double b) {
            System.out.println(a + "+" + b + " = " + (a+b));
      }

      public void sum(double a, double b, double c) {
            System.out.println(a + "+" + b + "+" + c + " = " + (a+b+c));
      }
}
```

StaticPolyMain.java

```
public class StaticPolyMain {

	public static void main(String[] args) {
		Adder add = new Adder();
		add.sum(10, 20);
		add.sum(10, 20, 30);
		add.sum(10.5, 20.2);
		add.sum(10.5, 20.5, 10.0);
	}
}
```

Runtime Polymorphism: In Java, *dynamic polymorphism* can be achieved through *method overriding*. Additionally, *Dynamic method dispatch* is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

However, we need to know about two terms: ***declared type*** and ***actual type***. The type that declares a variable is called the variable's *declared type*. The *actual type* of the variable is the actual class for the object referenced by the variable. Most important part is, a method maybe implemented in several class along the inheritance chain. The JVM dynamically binds the implementation of the method at runtime, decided by the *actual type* of the variable rather than *declared type* of the variable. Example of a *runtime polymorphism* is given below.

Person.java

```
public class Person {
	@Override
	public String toString() {
		return "Person!";
	}
}
```

Employee.java

```
public class Employee extends Person {
	@Override
	public String toString() {
		return "Employee!";
	}
}
```

Faculty.java

```
public class Faculty extends Employee{
	@Override
	public String toString() {
		return "Faculty!";
	}
}
```

DynamicPolyMain.java

```
public class DynamicPolyMain {
```

```
	public static void main(String[] args) {
		Person p;

		p = new Person();
		System.out.println(p.toString());

		p = new Employee();
		System.out.println(p.toString());

		p = new Faculty();
		System.out.println(p.toString());
	}
}
```

Lab Task:

Implement the following UML class diagram. In the main class create several objects whose *declared data types* will be *Shape*, but the *actual type* will be *Shape*, *Circle*, *Rectangle*, and *Square* respectively. Then call the *toString()* methods for each one of them to show the details. In the *toString()* method display the area, and perimeter along with the attributes value.