Learning Objectives - Polymorphism

- **Define polymorphism**
- **Explain how method overriding is an example of polymorphism**
- **Overload a method**
- **Override a method**
- **Use an abstract method as a form of polymorphism**

Method Overriding

What is Polymorphism?

Polymorphism is a concept in object-oriented programming in which a single interface takes different forms (polymorphism means “many forms”). Often this means similar operations are grouped together with the same name. However, these operations with the same name will produce different results. You have already encountered a few examples of polymorphism. Enter the following code into the IDE.

```
    //add code below this line

    int a = 5;
    int b = 10;
    System.out.println(a + b);

    String c = "5";
    String d = "10";
    System.out.println(c + d);

    //add code above this line
```

Notice how the plus operator can add together two numbers and concatenate two strings. You have a single interface (the plus operator) taking different forms — one that works with integers and and another that works with strings. This is an example of polymorphism.

▼ **Operator Overloading**

Because the plus operator can work with different forms, we can say that it is overloaded. Java overloads this operator by default. However, a user cannot manually overload an operator.

challenge

Try this variation:

Change your code to look like this:

```
    //add code below this line

    int a = 5;
    boolean b = true;
    System.out.println(a + b);

    //add code above this line
```

▼ **Why is there an error?**
Polymorphism allows Java to use the plus operator with different data types, but that does not mean that the plus operator can be used with all data types. The example above causes an error message because the plus operator cannot be used with an integer and a boolean. There are limits to polymorphism.

Method Overriding

Method overriding is another example of polymorphism that you have already seen. Overriding a method means that you have two methods with the same name, but they perform different tasks. Again you see a single interface (the method name) being used with different forms (the superclass and the subclass). Create the following classes.

```
class Alpha {
  public void show() {
    System.out.println("I am from class Alpha");
  }
}

class Bravo extends Alpha {
  public void show() {
    System.out.println("I am from class Bravo");
  }
}
```

Instantiate an `Alpha` object and call the `show` method.

```
    //add code below this line

    Alpha testObject = new Alpha();
    testObject.show();

    //add code above this line
```

As expected, the script prints `I am from class Alpha`. Now change the line of code in which you instantiate the object `testObject`. Make no other changes and run the code again.

```
    Bravo testObject = new Bravo();
```

Now the script prints `I am from class Bravo`. The method call did not change, but the output did. A single interface (the `show` method) works with multiple forms (the `Alpha` and `Bravo` data types). This is why method overriding is an example of polymorphism.

challenge

Try this variation:

- Create and overload the method `hello` that prints `Hello from Alpha` and `Hello from Bravo`. Test the method with both class types.

  ▼ **Solution**

```java
class Alpha {
  public void show() {
    System.out.println("I am from class Alpha");
  }

  public void hello() {
    System.out.println("Hello from Alpha");
  }
}

class Bravo extends Alpha {
  public void show() {
    System.out.println("I am from class Bravo");
  }

  public void hello() {
    System.out.println("Hello from Bravo");
  }
}
```

Method Overloading

Method Overloading

Method overloading is another example of polymorphism. Method overloading occurs when you have a single method name that can take different sets of parameters. Imagine you want to write the method `sum` that can sum up to three numbers. The math involved with three parameters is slightly different than two parameters, which is different from 1 parameter, etc. Traditionally, if you declare a method that takes three parameters but only pass two, Java will throw an error message. Instead let's create a class that has two `sum` methods; one with two parameters and another with three parameters.

```
//add class definitions below this line

class TestClass {
  public int sum(int n1, int n2, int n3) {
    return n1 + n2 + n3;
  }

  public int sum(int n1, int n2) {
    return n1 + n2;
  }
}

//add class definitions above this line
```

Create an object of type `TestClass` and call both versions of the `sum` method. Be sure you are passing three arguments for one method and two arguments for the other.

```
    //add code below this line

    TestClass tc = new TestClass();
    System.out.println(tc.sum(1, 2, 3));
    System.out.println(tc.sum(1, 2));

    //add code above this line
```

Java looks at the number and types of arguments and, as long there is a matching method definition, runs the code without an error. Defining the same method with different sets of arguments is called overloading. It is also an example of polymorphism.

challenge

Try this variation:

- Continue to overload the `sum` method such that it can take up to five numbers as parameters. Be sure to test all possible method calls.

▼ **Solution**

```
//add class definitions below this line

class TestClass {
  public int sum(int n1, int n2, int n3, int n4, int n5) {
    return n1 + n2 + n3 + n4 + n5;
  }

  public int sum(int n1, int n2, int n3, int n4) {
    return n1 + n2 + n3 + n4;
  }

  public int sum(int n1, int n2, int n3) {
    return n1 + n2 + n3;
  }

  public int sum(int n1, int n2) {
    return n1 + n2;
  }

  public int sum(int n1) {
    return n1;
  }
}

//add class definitions above this line
```

Overloading the Constructor

Java will also allow you to overload the constructor so that objects are instantiated in a variety of ways. The `Person` class has a default constructor (no arguments) and a constructor with three arguments.

```
//add class definitions below this line

class Person {
  private String name;
  private int number;
  private String street;

  public Person() {}

  public Person(String na, int nu, String s) {
    name = na;
    number = nu;
    street = s;
  }

  public String info() {
    return name + " lives at " + number + " " + street + ".";
  }
}

//add class definitions above this line
```

When you create a `Person` object with no arguments, the `info` method still works. That is because Java will use the default values for integers and strings. You can also instantiate an object with three arguments. Like method overloading, constructor overloading is a form of polymorphism.

▼ Default Values

Here are the default values for various data types:

Data Type	Default Value
int	0
double	0.0
String	null
boolean	false

```
    //add code below this line

    Person p1 = new Person();
    Person p2 = new Person("Calvin", 37, "Main Street");
    System.out.println(p1.info());
    System.out.println(p2.info());

    //add code above this line
```

challenge

Try these variations:

- Comment out both of the constructors.

```
//   public Person() {}

//   public Person(String na, int nu, String s) {
//     name = na;
//     number = nu;
//     street = s;
//   }
```

Instantiate a `Person` object and call the `info` method.

```
    //add code below this line

    Person p1 = new Person();
    System.out.println(p1.info());

    //add code above this line
```

▼ **Why does this work?**
When you do not declare a constructor, Java will use the default constructor and give each of the attributes their default value.

- Uncomment only the constructor with three arguments.

```
//   public Person() {}

  public Person(String na, int nu, String s) {
    name = na;
    number = nu;
    street = s;
  }
```

Do not make any changes to the object instantiation.

```
    //add code below this line

    Person p1 = new Person();
    System.out.println(p1.info());

    //add code above this line
```

▼ **Why is there an error?**

Java automatically uses the default constructor when there are no constructors defined. If a constructor exists, you must declare a default constructor if you want to instantiate an object without any arguments.

Abstract Methods

Abstract Classes

Another form of polymorphism in Java involves abstract methods. These methods, however, require knowledge of abstract classes. So before we continue the discussion on polymorphism, we need to first talk about abstract classes.

▼ **Concrete Classes**
Any class that is not an abstract class is called a concrete class. You do not need to use a keyword to indicate that a class is concrete.

The two defining characteristics of abstract classes is that they use the `abstract` keyword and you cannot instantiate an object of an abstract class. The way to use an abstract class is through inheritance. Create the `Engineer` class that extends `Person`. Add the attribute `specialty`, and create the `greeting` method.

```
//add class definitions below this line

class Engineer extends Person {
  private String specialty;

  public Engineer(String n, String s) {
    setName(n);
    specialty = s;
  }

  public String greeting() {
    return "Hello, my name is " + getName() + " and I am a(n) "
        + specialty + ".";
  }
}

//add class definitions above this line
```

Instantiate an `Engineer` object and print the output from `greeting`.

```
    //add code below this line

    Engineer e = new Engineer("Calvin", "civil engineer");
    System.out.println(e.greeting());

    //add code above this line
```

challenge

Try this variation:

Instead of instantiating an `Engineer` object, create a `Person` object.

```
    //add code below this line

    Person p = new Person();

    //add code above this line
```

▼ **Why is there an error?**
The `Person` class is an abstract class. Java will not allow you to directly instantiate an object of an abstract class. You must use the abstract class through inheritance.

Abstract Methods

Once you have an abstract class, you can create an abstract method. These methods do not have a body. Rewrite the `Person` class so that the method `greeting` is a part of the class. Be sure to use the `abstract` keyword. Since the method does not have a body, you do not need the curly braces. However, you do need a semi-colon.

```
abstract class Person {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String newName) {
    name = newName;
  }

  public abstract String greeting();
}
```

When you create an abstract method, the subclass is required to override it. This is helpful when you want a subclass to have a specific method, but want to let the subclass decide how to implement it. To demonstrate this, create another subclass called `Artist`. This class is almost identical to the `Engineer` class except for the output of the `greeting` method.

```
//add class definitions below this line

class Engineer extends Person {
  private String specialty;

  public Engineer(String n, String s) {
    setName(n);
    specialty = s;
  }

  public String greeting() {
    return "Hello, my name is " + getName() + " and I am a(n) "
        + specialty + ".";
  }
}

class Artist extends Person {
  private String specialty;

  public Artist(String n, String s) {
    setName(n);
    specialty = s;
  }

  public String greeting() {
    return "My name is " + getName() + " and I work with " +
        specialty + ".";
  }
}

//add class definitions above this line
```

Instantiate an object of the `Artist` class and call the `greeting` method. Using an abstract method forces you to override a method, and overriding a method is a form of polymorphism. Therefore, abstract methods are also a form of polymorphism.

```
    //add code below this line

    Person p = new Engineer("Calvin", "civil engineer");
    Person a = new Artist("Maria", "water colors");

    System.out.println(p.greeting());
    System.out.println(a.greeting());

    //add code above this line
```

challenge

Try these variations:

- Comment out the `greeting` method in both the `Engineer` and `Artist` classes.

```
//add class definitions below this line

class Engineer extends Person {
  private String speciality;

  public Engineer(String n, String s) {
    setName(n);
    speciality = s;
  }

//   public String greeting() {
//     return "Hello, my name is " + getName() + " and I am
       a(n) " + speciality + ".";
//   }
}

class Artist extends Person {
  private String medium;

  public Artist(String n, String m) {
    setName(n);
    medium = m;
  }

//   public String greeting() {
//     return "My name is " + getName() + " and I work with
       " + medium + ".";
//   }
}

//add class definitions above this line
```

▼ **Why is there an error?**

A subclass **must** override an abstract method. Because the `greeting` methods are commented out in the subclasses, they do not override the abstract method in the `Person` class. That is why Java throws an error.

- Make the `Person` class concrete (remove the `abstract` keyword).

```
class Person {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String newName) {
    name = newName;
  }

  public abstract String greeting();
}
```

▼ **Why is there an error?**

The method `greeting` is an abstract method. Abstract methods can only appear in an abstract class. That is why Java throws an error.