Polymorphism

Mr. Poole
Java

Before Polymorphism,
let’s conquer **Compile vs Run**.

First: What does it mean to compile?

Second: What does it mean to run?

Compile vs Run

- Compiling
  - The **process of parsing a program written** and checking syntax, semantics, linking libraries, and creating a binary executable (.class file) program as an output.
- Running
  - Getting some binary executable and **executing it as a new process**!

Errors make it easy to make sense of

- Compile time error example
  - Syntax Error
  - Logic Error
  - Object not found
- These errors happen when the program is parsing through to see if it actually can run!

- Runtime error example
  - Array out of Bounds
  - Null Pointer Exception
- These errors happen after compiling. Logic and syntax work but until ran, the computer doesn’t know it reaches something it can’t find.

Now we can discuss
Polymorphism

What is Polymorphism?

- A reference variable is polymorphic when it can refer to objects from different classes at different points in the code.
  - A reference variable can store a reference to its declared class or to any subclass of its declared class
- A method is considered polymorphic when it is overridden in at least one subclass.
- Polymorphism is the act of executing an overridden non-static method from the correct class at **runtime** based on the actual object type.

In short:

Polymorphism defines the behavior of how variables act when instantiated as another class object of a subclass.

Example:

```
Role obj = new Warrior();

obj.charge();
```

Is the Role or Warrior's **`charge()`** method called?

Let's focus on **overriding methods** first!

```
public class Dog{
      private String name;
      private int age;

      public Dog() {...}
      public Dog(String n, int a){...}

      public void bark(){
            System.out.println("Bark!");
      }
}
```

Previously we have the **Dog** and **Greyhound** classes.

We overrode the **`bark`** method from Dog

```
public class Greyhound extends Dog{
      private String color;

      public Greyhound () {...}
      public Greyhound (String n, int a, String c){...}

      public boolean isFast(){
            return true;
      }
      public void bark(){
            System.out.println("LOUD BARK!");
      }
}
```

Methods with Polymorphism

```
public class Dog{
      private String name;
      private int age;

      public Dog() {...}
      public Dog(String n, int a){...}

      public void bark(){
            System.out.println("Bark!");
      }
}
```

```
public class Greyhound extends Dog{
      private String color;

      public Greyhound () {...}
      public Greyhound (String n, int a, String c){...}

      public boolean isFast(){
            return true;
      }
      public void bark(){
            System.out.println("LOUD BARK!");
      }
}
```

```
Greyhound rapid = new Greyhound("Rapid", 7, "grey");
rapid.bark();
```

Given the code above, this compiles and runs!

These are the following steps that happen when **`rapid.bark()`** is compiling:

1. Check what class the **`rapid`** object is.
   a. Greyhound is the reference **variable** type
2. Check if that class has a **`bark()`** method.
   a. If yes, compile it.
   b. If not, check if there's an inherited **`bark()`**.

Methods with Polymorphism

```
public class Dog{
	private String name;
	private int age;

	public Dog() {...}
	public Dog(String n, int a){...}

	public void bark(){
		System.out.println("Bark!");
	}
}
```

```
public class Greyhound extends Dog{
	private String color;

	public Greyhound () {...}
	public Greyhound (String n, int a, String c){...}

	public boolean isFast(){
		return true;
	}
	public void bark(){
		System.out.println("LOUD BARK!");
	}
}
```

```
Dog rapid = new Greyhound("Rapid", 7, "grey");
rapid.bark();
```

OKAY, same code BUT different reference variable!

These are the following steps that happen when **`rapid.bark()`** is compiling:

1. Check what class the **`rapid`** object is.
   a. The reference variable a **Dog!**
2. Check if that class has a **`bark()`** method.
   a. If yes, compile it.
   b. If not, check if there's an inherited **`bark()`**.

This still works, since Dog has bark().

Methods with Polymorphism

```
public class Dog{
      private String name;
      private int age;

      public Dog() {...}
      public Dog(String n, int a){...}

      public void bark(){
            System.out.println("Bark!");
      }
}
```

```
Dog rapid = new Greyhound("Rapid", 7, "grey");
rapid.bark();
```

Though this compiles because **Dog** has a **`bark()`** method.

During **runtime,** the Greyhound **`bark()`** is referenced.

```
public class Greyhound extends Dog{
      private String color;

      public Greyhound () {...}
      public Greyhound (String n, int a, String c){...}

      public boolean isFast(){
            return true;
      }
      public void bark(){
            System.out.println("LOUD BARK!");
      }
}
```

Methods with Polymorphism

```
public class Dog{
      private String name;
      private int age;

      public Dog() {...}
      public Dog(String n, int a){...}

      public void bark(){
            System.out.println("Bark!");
      }
}
```

```
public class Greyhound extends Dog{
      private String color;

      public Greyhound () {...}
      public Greyhound (String n, int a, String c){...}

      public boolean isFast(){
            return true;
      }
      public void bark(){
            System.out.println("LOUD BARK!");
      }
}
```

```
Dog rapid = new Greyhound("Rapid", 7, "grey");
rapid.isFast();
```

Now instead of **bark**, what if we try **isFast()**?

These are the following steps that happen when `rapid.bark()` is compiling:

1. Check what class the `rapid` object is.
   a. Variable is still a **Dog**!
2. Check if that class has a `isFast()` method.
   a. If yes, compile it.
   b. If not, check if there's an inherited `bark()`.

This doesn't compile, since Dog doesn't have isFast()

Let’s try an example question:

```
public class Parent{
    public void display1(){
        System.out.print("P");
    }
    public void display2(){
        System.out.print("W");
    }
}

public class Child extends Parent{
    public void display1(){
        super.display1();
        System.out.print("C");
    }
    public void display2(){
        System.out.print("X");
    }
    public void display3(){
        System.out.print("Y");
    }
}
```

What's the output?

```
public class Parent{
    public void display1(){
        System.out.print("P");
    }
    public void display2(){
        System.out.print("W");
    }
}

public class Child extends Parent{
    public void display1(){
        super.display1();
        System.out.print("C");
    }
    public void display2(){
        System.out.print("X");
    }
    public void display3(){
        System.out.print("Y");
    }
}
```

In a different class:

```
Parent obj = new Child();
obj.display1();
obj.display2();
```

What is displayed as a result of executing the above segment of code?

(A) PW
(B) PCW
(C) CX
(D) PCX
(E) CW

Answer

```
public class Parent{
    public void display1(){
        System.out.print("P");
    }
    public void display2(){
        System.out.print("W");
    }
}

public class Child extends Parent{
    public void display1(){
        super.display1();
        System.out.print("C");
    }
    public void display2(){
        System.out.print("X");
    }
    public void display3(){
        System.out.print("Y");
    }
}
```

In a different class:

```
Parent obj = new Child();
obj.display1();
obj.display2();
```

What is displayed as a result of executing the above segment of code?

(A) PW
(B) PCW
(C) CX
(D) PCX
(E) CW

Answer

```
public class Parent{
    public void display1(){
        System.out.print("P");
    }
    public void display2(){
        System.out.print("W");
    }
}

public class Child extends Parent{
    public void display1(){
        super.display1();
        System.out.print("C");
    }
    public void display2(){
        System.out.print("X");
    }
    public void display3(){
        System.out.print("Y");
    }
}
```

In a different class:

```
Parent obj = new Child();
obj.display1();
obj.display2();
```

What is displayed as a result of executing the above segment of code?

(A) PW
(B) PCW
(C) CX
(D) PCX
(E) CW

1. This compiles because Parent has a **`display1()`** and **`display2()`** method part of its class.

2. This runs **`Child`**'s **`display1()`** and **`display2()`** because **`obj`** is instantiated as a **`Child`**

So back to our **`bark()`** example

How do we call **`isFast()`** from Greyhound on a Dog?

```
Dog rapid = new Greyhound("Rapid", 7, "grey");
rapid.isFast();
```

Superclass Polymorphism call

```
public class Dog{
      private String name;
      private int age;

      public Dog() {...}
      public Dog(String n, int a){...}

      public void bark(){
            System.out.println("Bark!");
      }
}
```

```
Dog rapid = new Greyhound("Rapid", 7, "grey");
rapid.isFast();
```

This runs the Greyhound's **`isFast()`**.

But it doesn't compile

```
public class Greyhound extends Dog{
      private String color;

      public Greyhound () {...}
      public Greyhound (String n, int a, String c){...}

      public boolean isFast(){
            return true;
      }
      public void bark(){
            System.out.println("LOUD BARK!");
      }
}
```

Superclass Polymorphism call

```
public class Dog{
      private String name;
      private int age;

      public Dog() {...}
      public Dog(String n, int a){...}

      public void bark(){
            System.out.println("Bark!");
      }
}
```

```
Dog rapid = new Greyhound("Rapid", 7, "grey");
((Greyhound)rapid).isFast();
```

This compiles and runs the Greyhound's **`isFast()`**.

```
public class Greyhound extends Dog{
      private String color;

      public Greyhound () {...}
      public Greyhound (String n, int a, String c){...}

      public boolean isFast(){
            return true;
      }
      public void bark(){
            System.out.println("LOUD BARK!");
      }
}
```

Superclass Polymorphism call

This is called **object casting**,
where you change the object used at runtime.

Imagine it like, multiplying by some constant value.

```
Dog rapid = new Greyhound("Rapid", 7, "grey");
((Greyhound)rapid).isFast();
```

Polymorphism like this works for **methods NOT for variables**.

The next part is just some spooky behaviors you need to be careful about when working with inheritance.

We make a Dog with a Corgi instantiated

```
Dog a = new Corgi();
System.out.println(a.name);
```

Now given the below classes, what does the code above do?

```
public class Corgi extends Dog{
    String breed;
    String name;

    public Corgi() {
        breed = "Corgi";
        name = "Joey";
    }
}
```

```
public class Dog{
    String name;

    public Dog(){
        name = "Doggo";
    }

    public void bark(){
        System.out.println("Bark!");
    }
}
```

Polymorphism behaviors

```
Dog a = new Corgi();
System.out.println(a.name);
```

The code above, outputs “**Doggo**”.
This is because the instance variable is Dog still.

This isn’t the behavior expected.
This is an exception in Java.

Remember polymorphism doesn’t work on Variables

```
public class Dog{
    String name;

    public Dog(){
        name = "Doggo";
    }

    public void bark(){
        System.out.println("Bark!");
    }
}
```

And now we know why
Accessor/Mutator methods are amazing!

Lab: Polymorphism

1. Create an array of 4 Performers
2. Construct with any value
    a. 1 Performer
    b. 1 Musician
    c. 1 Apprentice
    d. 1 Actor
3. Call the following methods on the following objects
    a. Performer - perform() and practice()
    b. Musician - perform() and practice()
    c. Apprentice - practiceAtUniversity(), playInstrument()
    d. Actor - monologue() and perform()