

Polymorphism

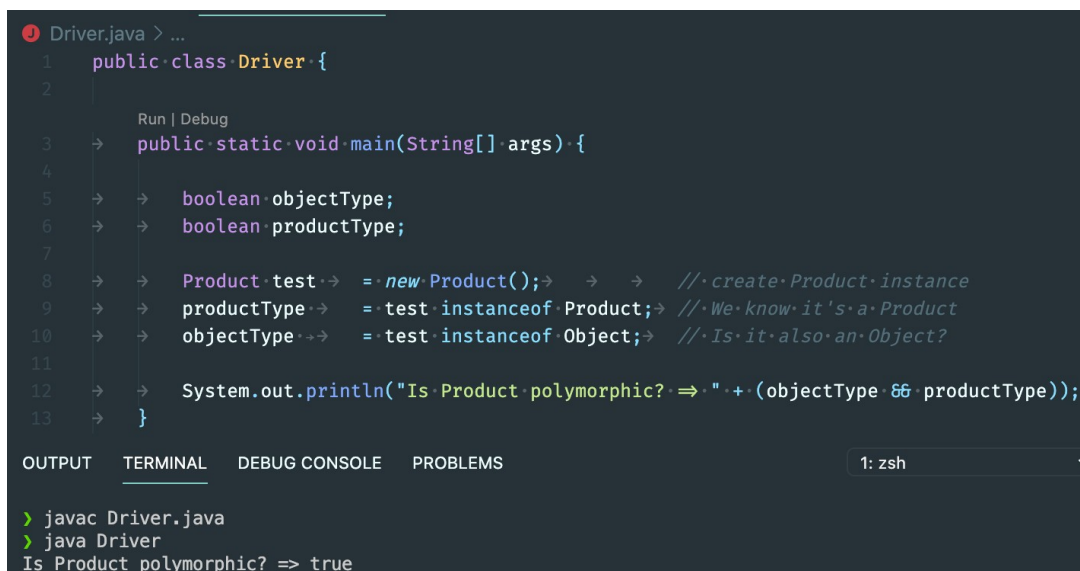
The word polymorphism is used in various contexts and describes situations in which something occurs in several different forms (“poly” = many; “morph” = form). In computer science, it describes the concept that objects of different types can be accessed through the same interface. Remember that the term **interface** here refers to the public methods which an object exposes to the “outside world” . . . the way in which objects can interact. Each type can provide its own, independent implementation of this interface. It is one of the core concepts of object-oriented programming (OOP).

If you’re wondering if an object is polymorphic, you can perform a simple test. If the object successfully passes multiple is-a or **instanceof** tests, it’s polymorphic. As I’ve described in earlier when discussing inheritance, all Java classes extend the class `Object`. Due to this, all objects in Java are polymorphic because they pass at least two **instanceof** checks. Let’s look at an example

The Product class: Notice that this class does not specify an **extends** clause. As we have seen, classes that do not **explicitly** extend a class **implicitly** extend the `Object` class.

```
1 public class Product{
2
3     → private String name;
4     → private double weight;
5     → private double price;
6 }
```

How can we prove this? We can utilize the **instanceof** operator. Here is a little toy program to illustrate this.



```
Driver.java > ...
1 public class Driver {
2
3     Run | Debug
4     → public static void main(String[] args) {
5         → → boolean objectType;
6         → → boolean productType;
7
8         → → Product test = new Product(); → → // create Product instance
9         → → productType = test instanceof Product; → // We know it's a Product
10        → → objectType = test instanceof Object; → // Is it also an Object?
11
12        → → System.out.println("Is Product polymorphic? => " + (objectType && productType));
13        → → }

```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 1: zsh

```
> javac Driver.java
> java Driver
Is Product polymorphic? => true

```

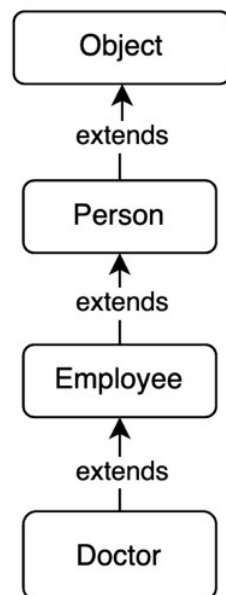
Different types of polymorphism

Java supports 2 types of polymorphism:

- **static or compile-time:** Method invocation is bound to specific method signature when the code is compiled.
- **dynamic or run-time:** Method invocation is bound to specific method signature when the code is executed.

Note: In Java an object variable has both a **declared (compile-time) type** and an **actual (run-time) type**. The *declared (compile-time) type* of a variable is the type that is used in the declaration. The *actual (run-time) type* is the class (constructor) that actually creates the object using new.

The following is perfectly legal and illustrates the **is a** property. Given the following inheritance hierarchy, the following declarations are legal. A subclass object can be assigned and **upcasted** to a super class type. This is legal for any subclass along the entire hierarchy. This does not work using composition or **has a** relationship



```
// compile-time type is Object  
// runtime type is Person  
Object o = new Person();  
  
// compile-time type is Person  
// runtime type is Person  
Person p = new Person();  
  
// compile-time type is Person  
// runtime type is Employee  
Person e = new Employee();  
  
// compile-time type is Person  
// runtime type is Doctor  
Person d = new Doctor();
```

Method Binding

Method binding is a mechanism of **associating a method call** in Java code **to the declaration** and implementation of the method being called. When we mention the phrase “declaration of the method”, we are referring to the method signature, which consists of the method name, and the order and the data type of its parameters. When we mention the phrase “implementation of the method”, we are referring to the body of the method that executes at runtime as part of the method call. The compiler always decides the signature of the called method. Depending on the method type (static, non-static, and interface), the compiler or the runtime decides what implementation of the method is executed at runtime.

Static Method Binding

Java, like many other object-oriented programming languages, allows you to implement multiple methods within the same class, or in an inheritance hierarchy that use the **same name** but a **different set of parameters**. That is called **method overloading** and represents a static form of polymorphism.

The parameter sets have to differ in at least **one of the following** three criteria:

- They need to have a different number of parameters, e.g. one method accepts 2 and another one 3 parameters.
- The types of the parameters need to be different, e.g. one method accepts a String and another one a long.
- The parameters can be in a different order, e.g. one method accepts a String and a long and another one accepts a long and a String. This kind of overloading is not recommended because it makes the API difficult to understand.

In most cases, each of these overloaded methods provides a different but very similar functionality.

Due to the different sets of parameters, each method has a **different signature**. That allows the compiler to identify which method has to be called and to bind it to the method call. This approach is called **static binding** or **static polymorphism**.

Let's take a look at an example.

Overloaded methods in the Date class. Below you can see two overloaded methods from the provided Date class. These methods allow for similar but different behavior. They both allow for the month to be set, but one allows an integer to be passed in and one allows for a String to be passed in. They both accomplish the same thing . . . setting the month.

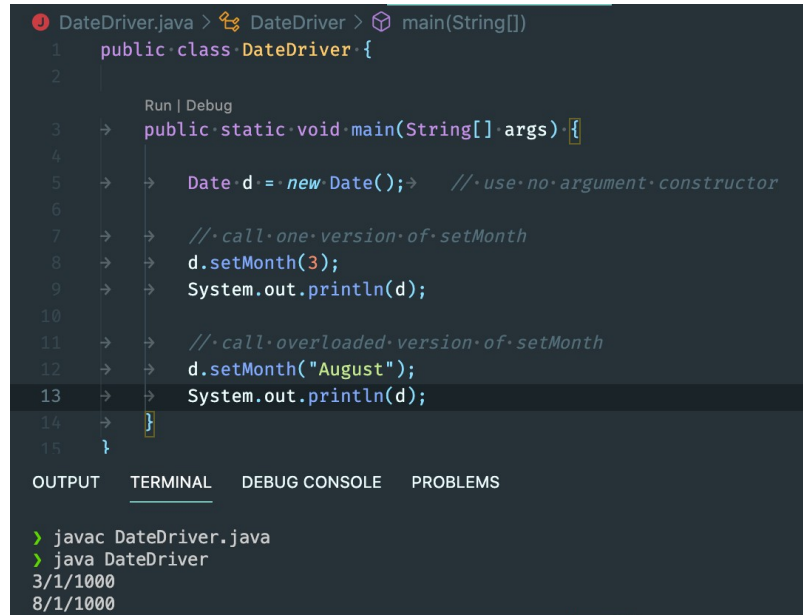
As described above, these methods are **overloaded** due to the same name, but differing signature. Overloaded methods illustrate **static polymorphism**. They meet the criteria for "polymorphism" due to the adherence to "same name, different form" principle.

```
public void setMonth(int m){
    ....//perform some domain validation
    ....if(m ≥ 1 && m ≤ 12)
    ....    month = m;
    ....else
    ....    month = 1;
} //end setMonth

//overloaded setMonth
public void setMonth(String month){
    ....int month_num = getMonthNumber(month); ....//call helper method
    ....this.setMonth(month_num); ....//pass it through setMonth
}
```

And a Driver class to show the results. →

It's easy to see how the compiler can differentiate the method calls on lines 8 and 12, **binding** them to the correct implementation. The compiler sees the String and int parameters as distinct calls. A **statically bound** method invocation will not change throughout the lifetime of the program.



```
1 public class DateDriver {
2
3     public static void main(String[] args) {
4
5         Date d = new Date(); // use no argument constructor
6
7         // call one version of setMonth
8         d.setMonth(3);
9         System.out.println(d);
10
11        // call overloaded version of setMonth
12        d.setMonth("August");
13        System.out.println(d);
14    }
15 }
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
> javac DateDriver.java
> java DateDriver
3/1/1000
8/1/1000
```

Dynamic Polymorphism (Dynamic Dispatch)

This form of polymorphism doesn't allow the compiler to determine the executed method. The JVM will make a decision based on the object's type at runtime, and then bind the call to the correct implementation. There are a couple of characteristics that must exist in order to benefit from dynamic polymorphism

1. You must have an inheritance hierarchy
2. You must have overridden methods

Method Overriding Reminder

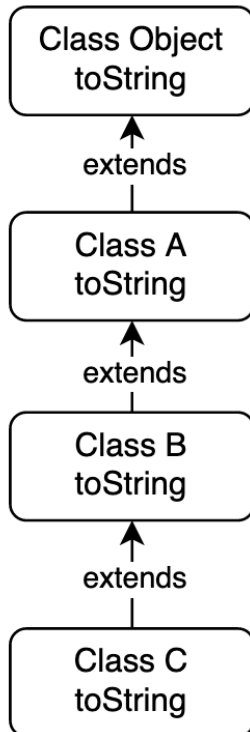
in object-oriented programming, **method overriding** is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes. The implementation in the subclass **overrides (replaces)** the implementation in the superclass by providing a method that has

- same name
- same parameters or signature
- same return type as the method in the parent class. (This is a special concept)

Within an inheritance hierarchy, a subclass can override a method of its superclass. That enables the developer of the subclass to customize or completely replace the behavior of that method.

It also creates a form of polymorphism. Both methods, implemented by the super and subclass, share the same name and parameters but provide different functionality. The version of a method that is executed will be determined by the **runtime type**, 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, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.

Let's take a look at an example. Here is a brief inheritance hierarchy. This is a toy example just to illustrate the concepts.



```
1  public class A{
2      ...private String name;
3  > ...public A(String name){ ...
6  > ...public String getName(){ ...
9  > ...private void helper(){ ...
12 ...@Override
13 > ...public String toString(){ ...
16 ...@Override
17 > ...public boolean equals(Object obj){ ...
36 } // end of A
37
38 class B extends A{
39     ...private double weight;
40 > ...public B(String name, double weight){ ...
44 ...@Override
45 > ...public String toString(){ ...
49 } // end of B
50
51 class C extends B{
52     ...private String color;
53 > ...public C(String name, double weight, String color){ ...
57 ...@Override
58 > ...public String toString(){ ...
61 } // end of C
62
```

Notes:

- Class A implicitly extends Object
- The Object class has a toString method
- 1. Each subsequent class A → B → C all provide overridden versions of toString as noted by the **@Override** annotation
- There are 4 different implementations of toString methods in this program.
- This organization creates the scenario for **dynamic polymorphism** around the multiple implementations of the toString method. Let's see what this means.

```
1 public class A{
2     ...private String name;
3     > ...public A(String name){ ...
6     > ...public String getName(){ ...
9     > ...private void helper(){ ...
12    ...@Override
13    ...public String toString(){
14    ...        return "Class A → Name is: " + name;
15    ...    }
16    ...@Override
17    > ...public boolean equals(Object obj){ ...
36    } // end of A
37
38    class B extends A{
39    ...private double weight;
40    > ...public B(String name, double weight){ ...
44    ...@Override
45    ...public String toString(){
46    ...        return "Class B → weight is: " + weight;
47    ...    }
48    } // end of B
49
50    class C extends B{
51    ...private String color;
52    > ...public C(String name, double weight, String color){ ...
56    ...@Override
57    ...public String toString(){
58    ...        return "Class C → color is: " + color;
59    ...    }
60    } // end of C
```

Calling a Method Polymorphically

The following code shows the method **polymorphicMethodCalls** that has a parameter of class type A which is the second highest level in the inheritance hierarchy. Based on the **is a** relationship between these objects, we can call this method passing any object **whose runtime type is A or a subclass of A**. The polymorphic behavior derives from the fact that the JVM will correctly call the appropriate toString method even though the method is invoked through a variable of type A

```
class PolymorphismDriver{
    Run | Debug
    ...public static void main(String[] args){
    ...
    ...C c = new C("warlock mini", 5, "Red");
    ...B b = new B("wizard mini", 3);
    ...A a = new A("halfling mini");
    ...
    ...polymorphicMethodCalls(a);
    ...polymorphicMethodCalls(b);
    ...polymorphicMethodCalls(c);
    ...}
    ...
    ...public static void polymorphicMethodCalls(A a){
    ...    System.out.println(a.toString());
    ...}
}
```


Notice this very interesting behavior. We do not have to use any decision statements to decide which method to call

```
62 class PolymorphismDriver{
63     public static void main(String[] args){
64         .....C c = new C("warlock mini", 5, "Red");
65         .....B b = new B("wizard mini", 3);
66         .....A a = new A("halfling mini");
67
68         .....polymorphicMethodCalls(a);
69         .....polymorphicMethodCalls(b);
70         .....polymorphicMethodCalls(c);
71     }
72
73     public static void polymorphicMethodCalls(A a){
74         .....System.out.println(a.toString());
75     }
76 }
77
```

toString is called via a variable of **compile time type A**

```
> javac A.java
> java PolymorphismDriver
Class A --> Name is: halfling mini
Class B --> weight is: 3.0
Class C --> color is: Red
```

Different toString methods are actually called **based on the runtime type** of the object

Note: The inherited method `getClass()` returns the runtime type of an object

Polymorphism and Arrays of Superclass Objects

You can also achieve polymorphic behavior by organizing objects into collections of super class types. Let's see this with the ultimate super class: Object.

```
• // declare an array of type Object
Object[] collection = new Object[5];

• // create some objects from the A, B, C hierarchy
collection[0] = new A("Warforged Wildebeast");
collection[1] = new B("Dire Wolf", 12);
collection[2] = new C("Dragonborn Warlock", 9, "green");
collection[3] = new A("Serpent Cat");
collection[4] = new B("Three-Eyed Raven", 4);
```

Compile time type of the array is Object

Runtime types of objects are A, B and C

Let's use an enhanced for loop to iterate through the array and call toString on each element.

```
74 .....//declare an array of type Object
75 .....Object[] collection = new Object[5];
76
77 .....//create some objects from the A, B, C hierarchy
78 .....collection[0] = new A("Warforged Wildebeast");
79 .....collection[1] = new B("Dire Wolf", 12);
80 .....collection[2] = new C("Dragonborn Warlock", 9, "green");
81 .....collection[3] = new A("Serpent Cat");
82 .....collection[4] = new B("Three-Eyed Raven", 4);
83
84 .....for (Object obj : collection)
85 .....    System.out.println(obj.toString());
86 .....}
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 1

```
> javac A.java
> java PolymorphismDriver
Class A --> Name is: Warforged Wildebeast
Class B --> weight is: 12.0
Class C --> color is: green
Class A --> Name is: Serpent Cat
Class B --> weight is: 4.0
```

The toString methods invoked through the **Object** variable get correctly bound to the appropriate implementation

In these examples we have focused on the toString method, but this situation can be created with any method, provided you have an inheritance hierarchy and overridden methods.

Summary

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- When an overridden method is called through a superclass reference, Java determines which version (superclass/subclass) of that method is to be executed based upon the type of the object being referred to at the time the call occurs (eg. The constructor used to build the object). Thus, this determination is made at run time.
- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
- A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.