Theory: Polymorphism

42 minutes

Verify to skip | Start practicing

§1. Kinds of polymorphism

In general, **polymorphism** means that something (an object or another entity) has many forms.

Java provides two types of polymorphism: **static (compile-time)** and **dynamic (run-time)** polymorphism. The first one is achieved by **method overloading**, the second one is based on inheritance and **method overriding**.

The more theoretical approach subdivides polymorphism into several fundamentally different types:

- **Ad-hoc polymorphism** refers to polymorphic functions that can be applied to arguments of different types, but behave differently depending on the type of the argument to which they are applied. Java supports it as **method overloading**.
- **Subtype polymorphism** (also known as subtyping) is a possibility to use an instance of a subclass when an instance of the base class is permitted.
- **Parametric polymorphism** is when the code is written without mention of any specific type and thus can be used transparently with any number of new types. Java supports it as **generics** or **generic programming**.

In this topic, we consider only **subtype (runtime) polymorphism** that is widely used in object-oriented programming.

§2. Runtime polymorphic behavior

A reminder: *method overriding* is when a subclass redefines a method of the superclass with the same signature.

Run-time polymorphism relies on two principles:

- a reference variable of the superclass can refer to any subtype object;
- a superclass method can be overridden in a subclass.

Run-time polymorphism works when an overridden method is called through the reference variable of a superclass. Java determines at runtime which version of the method (superclass/subclasses) is to be executed based on the type of the object being referred, not the type of reference. It uses a mechanism known as **dynamic method dispatching**.

Example. Here, you can see a class hierarchy. The superclass `MythicalAnimal` has two subclasses: `Chimera` and `Dragon`. The base class has a method `hello`. Both subclasses override this method.

2 required topics

- ✓ Referencing subclass objects — In project
- ✓ Hiding and overriding

1 dependent topic

- ✓ Abstract class

Table of contents:

```
class MythicalAnimal {

    public void hello() {
        System.out.println("Hello, I'm an unknown animal");
    }
}

class Chimera extends MythicalAnimal {
    @Override
    public void hello() {
        System.out.println("Hello! Hello!");
    }
}

class Dragon extends MythicalAnimal {
    @Override
    public void hello() {
        System.out.println("Rrrr...");
    }
}
```

We can create a reference to the class `MythicalAnimal` and assign the subclass object to it:

```
MythicalAnimal chimera = new Chimera();
MythicalAnimal dragon = new Dragon();
MythicalAnimal animal = new MythicalAnimal();
```

We can also invoke overridden methods through the base class references:

```
chimera.hello(); // Hello! Hello!
dragon.hello(); // Rrrr...
animal.hello(); // Hello, I'm an unknown animal
```

So, the result of a method call depends on the actual type of instance, not the reference type. It's a polymorphic feature in Java. The JVM calls the appropriate method for the object that is referred to in each variable.

Subtype polymorphism allows a class to specify methods that will be common to all of its subclasses. Subtype polymorphism also makes it possible for subclasses to override the implementations of those methods. Together with abstract methods and interfaces, which you'll learn about later, subtype polymorphism is a fundamental object-oriented design concept.

§3. Polymorphism within a class hierarchy

The same thing works with methods that are used only within a hierarchy and are not accessible from the outside.

In the following example, we have a hierarchy of files. The parent class `File` represents a description of a single file in the file system. It has a subclass named `ImageFile`. It overrides the method `getFileInfo` of the parent class.

```
class File {

    protected String fullName;

    // constructor with a single parameter

    // getters and setters

    public void printFileInfo() {

String info = this.getFileInfo(); // here is polymorphic behavior!!!
        System.out.println(info);
    }

    protected String getFileInfo() {
        return "File: " + fullName;
    }
}

class ImageFile extends File {

    protected int width;
    protected int height;
    protected byte[] content;

    // constructor

    // getters and setters

    @Override
    protected String getFileInfo() {

return String.format("Image: %s, width: %d, height: %d", fullName, wi
    }
}
```

The parent class has a public method `printFileInfo` and a protected method `getFileInfo`. The second method is overridden in the subclass, but the subclass doesn't override the first method.

Let's create an instance of `ImageFile` and assign it to a variable of `File`.

```
File img = new ImageFile("/path/to/file/img.png", 480, 640, someBytes); // as
```

Now, when we call the method `printFileInfo`, it invokes the overridden version of the method `getFileInfo`.

```
img.printFileInfo(); // It prints "Image: /path/to/file/img.png, width: 480,
```

So, **run-time polymorphism** allows you to invoke an overridden method of a subclass having a reference to the base class.