(http://baeldung.com)

Polymorphism in Java

Last modified: December 28, 2017

by baeldung (http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/) +

1. Overview

All Object-Oriented Programming (OOP) languages are required to exhibit four basic characteristics: abstraction, encapsulation, inheritance, and polymorphism.

In this article, we cover two core types of polymorphism: ***static or compile-time polymorphism*** *and* ***dynamic or runtime polymorphism***. Static polymorphism is enforced at compile time while dynamic polymorphism is realized at runtime.

2. Static Polymorphism

According to Wikipedia (https://en.wikipedia.org/wiki/Template_metaprogramming#Static_polymorphism), static polymorphism is an imitation of **polymorphism which is resolved at compile time and thus does away with run-time virtual-table lookups**.

For example, our *TextFile* class in a file manager app can have three methods with the same signature of the *read()* method:

```
1  public class TextFile extends GenericFile {
2      //...
3
4      public String read() {
5          return this.getContent()
6            .toString();
7      }
8
9      public String read(int limit) {
10         return this.getContent()
11           .toString()
12           .substring(0, limit);
13     }
14
15     public String read(int start, int stop) {
16         return this.getContent()
17           .toString()
18           .substring(start, stop);
19     }
20 }
```

During code compilation, the compiler verifies that all invocations of the *read* method correspond to at least one of the three methods defined above.

3. Dynamic Polymorphism

With dynamic polymorphism, the **Java Virtual Machine (JVM) handles the detection of the appropriate method to execute when a subclass is assigned to its parent form**. This is necessary because the subclass may override some or all of the methods defined in the parent class.

In a hypothetical file manager app, let's define the parent class for all files called *GenericFile*:

```
public class GenericFile {
    private String name;

    //...

    public String getFileInfo() {
        return "Generic File Impl";
    }
}
```

We can also implement an *ImageFile* class which extends the *GenericFile* but overrides the *getFileInfo()* method and appends more information:

```
public class ImageFile extends GenericFile {
    private int height;
    private int width;

    //... getters and setters

    public String getFileInfo() {
        return "Image File Impl";
    }
}
```

When we create an instance of *ImageFile* and assign it to a *GenericFile* class, an implicit cast is done. However, the JVM keeps a reference to the actual form of *ImageFile*.

The above construct is analogous to method overriding. We can confirm this by invoking the *getFileInfo()* method by:

```
public static void main(String[] args) {
    GenericFile genericFile = new ImageFile("SampleImageFile", 200, 100,
      new BufferedImage(100, 200, BufferedImage.TYPE_INT_RGB)
      .toString()
      .getBytes(), "v1.0.0");
    logger.info("File Info: \n" + genericFile.getFileInfo());
}
```

As expected, *genericFile.getFileInfo()* triggers the *getFileInfo()* method of the *ImageFile* class as seen in the output below:

```
File Info:
Image File Impl
```

4. Other Polymorphic Characteristics in Java

In addition to these two main types of polymorphism in Java, there are other characteristics in the Java programming language that exhibit polymorphism. Let's discuss some of these characteristics.

4.1. Coercion

Polymorphic coercion deals with implicit type conversion done by the compiler to prevent type errors. A typical example is seen in an integer and string concatenation:

```
1 String str = “string” + 2;
```

4.2. Operator Overloading

Operator or method overloading refers to a polymorphic characteristic of same symbol or operator having different meanings (forms) depending on the context.

For example, the plus symbol (+) can be used for mathematical addition as well as *String* concatenation. In either case, only context (i.e. argument types) determines the interpretation of the symbol:

```
1 String str = "2" + 2;
2 int sum = 2 + 2;
3 System.out.printf(" str = %s\n sum = %d\n", str, sum);
```

Output:

```
str = 22
str2 = 4
```

4.3. Polymorphic Parameters

Parametric polymorphism allows a name of a parameter or method in a class to be associated with different types. We have a typical example below where we define *content* as a *String* and later as an *Integer*.

```
public class TextFile extends GenericFile {
    private String content;

    public String setContentDelimiter() {
        int content = 100;
        this.content = this.content + content;
    }
}
```

It's also important to note that **declaration of polymorphic parameters can lead to a problem known as variable hiding** where a local declaration of a parameter always overrides the global declaration of another parameter with the same name.

To solve this problem, it is often advisable to use global references such as *this* keyword to point to global variables within a local context.

4.4. Polymorphic Subtypes

Polymorphic subtype conveniently makes it possible for us to assign multiple subtypes to a type and expect all invocations on the type to trigger the available definitions in the subtype.

For example, if we have a collection of *GenericFile*s and we invoke the *getInfo()* method on each of them, we can expect the output to be different depending on the subtype from which each item in the collection was derived:

```
GenericFile [] files = {new ImageFile("SampleImageFile", 200, 100,
  new BufferedImage(100, 200, BufferedImage.TYPE_INT_RGB).toString()
  .getBytes(), "v1.0.0"), new TextFile("SampleTextFile",
  "This is a sample text content", "v1.0.0")};

for (int i = 0; i < files.length; i++) {
    files[i].getInfo();
}
```

Subtype polymorphism is made possible by a combination of upcasting and late binding. Upcasting involves the casting of inheritance hierarchy from a supertype to a subtype:

```
ImageFile imageFile = new ImageFile();
GenericFile file = imageFile;
```

The resulting effect of the above is that *ImageFile*-specific methods cannot be invoked on the new upcast *GenericFile*. However, methods in the subtype override similar methods defined in the supertype.

To resolve the problem of not being able to invoke subtype-specific methods when upcasting to a supertype, we can do a downcasting of the inheritance from a supertype to a subtype. This is done by:

```
ImageFile imageFile = (ImageFile) file;
```

Late binding strategy helps the compiler to resolve whose method to trigger after upcasting. In the case of i*mageFile#getInfo* vs *file#getInfo* in the above example, the compiler keeps a reference to *ImageFile*'s *getInfo* method.

5. Problems with Polymorphism

Let's look at some ambiguities in polymorphism that could potentially lead to runtime errors if not properly checked.

5.1. Type Identification During Downcasting

Recall that we earlier lost access to some subtype-specific methods after performing an upcast. Although we were able to solve this with a downcast, this does not guarantee actual type checking.

For example, if we perform an upcast and subsequent downcast:

```
GenericFile file = new GenericFile();
ImageFile imageFile = (ImageFile) file;
System.out.println(imageFile.getHeight());
```

We notice that the compiler allows a downcast of a *GenericFile* into an *ImageFile*, even though the class actually is a *GenericFile* and not an *ImageFile*.

Consequently, if we try to invoke the *getHeight()* method on the *imageFile* class, we get a *ClassCastException* as *GenericFile* does not define *getHeight()* method:

```
Exception in thread "main" java.lang.ClassCastException:
GenericFile cannot be cast to ImageFile
```

To solve this problem, the JVM performs a Run-Time Type Information (RTTI) check. We can also attempt an explicit type identification by using the *instanceof* keyword just like this:

```
1  ImageFile imageFile;
2  if (file instanceof ImageFile) {
3      imageFile = file;
4  }
```

The above helps to avoid a *ClassCastException* exception at runtime. Another option that may be used is wrapping the cast within a *try* and *catch* block and catching the *ClassCastException.*

It should be noted that **RTTI check is expensive** due to the time and resources needed to effectively verify that a type is correct. In addition, frequent use of the *instanceof* keyword almost always implies a bad design.

5.2. Fragile Base Class Problem

According to Wikipedia (https://en.wikipedia.org/wiki/Fragile_base_class), base or superclasses are considered fragile if seemingly safe modifications to a base class may cause derived classes to malfunction.

Let's consider a declaration of a superclass called *GenericFile* and its subclass *TextFile*:

```
public class GenericFile {
    private String content;

    void writeContent(String content) {
        this.content = content;
    }
    void toString(String str) {
        str.toString();
    }
}
```

```
public class TextFile extends GenericFile {
    @Override
    void writeContent(String content) {
        toString(content);
    }
}
```

When we modify the *GenericFile* class:

```
public class GenericFile {
    //...

    void toString(String str) {
        writeContent(str);
    }
}
```

We observe that the above modification leaves *TextFile* in an infinite recursion in the *writeContent()* method, which eventually results in a stack overflow.

To address a fragile base class problem, we can use the *final* keyword to prevent subclasses from overriding the *writeContent()* method. Proper documentation can also help. And last but not least, the composition should generally be preferred over inheritance.

6. Conclusion

In this article, we discussed the foundational concept of polymorphism, focusing on both advantages and disadvantages.

As always, the source code for this article is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java).

CATEGORIES

SPRING (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
HTTPCLIENT 4 TUTORIAL (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
SPRING PERSISTENCE TUTORIAL (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SECURITY WITH SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)

THE COURSES (HTTP://COURSES.BAELDUNG.COM)

CONSULTING WORK (HTTP://WWW.BAELDUNG.COM/CONSULTING)

META BAELDUNG (HTTP://META.BAELDUNG.COM/)

THE FULL ARCHIVE (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)

WRITE FOR BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)

CONTACT (HTTP://WWW.BAELDUNG.COM/CONTACT)

COMPANY INFO (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)

TERMS OF SERVICE (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)

PRIVACY POLICY (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)

EDITORS (HTTP://WWW.BAELDUNG.COM/EDITORS)

MEDIA KIT (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)