Inheritance, Overriding, and Hiding

How JVM deals with methods in Java

By
Vaibhav Choudhary (@vaibhav_c)

Understanding Inheritance

- Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.

- When you inherit from an existing class, you can reuse methods and fields of the parent class.

- Overriding provides you, what you want extra.

Overriding example

```
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
}
class SlowPoint extends Point {
    int xLimit, yLimit;
    void move(int dx, int dy) {
        super.move(limit(dx, xLimit), limit(dy, yLimit));
    }
    static int limit(int d, int limit) {
        return d > limit ? limit : d < -limit ? -limit : d;
    }
}
```

Overriding Example

```
import java.io.OutputStream;
import java.io.IOException;

class BufferOutput {
    private OutputStream o;
    BufferOutput(OutputStream o) { this.o =
o; }
    protected byte[] buf = new byte[512];
    protected int pos = 0;
    public void putchar(char c) throws
IOException {
        if (pos == buf.length) flush();
        buf[pos++] = (byte)c;
    }
    public void putstr(String s) throws
IOException {
        for (int i = 0; i < s.length(); i++)
            putchar(s.charAt(i));
    }
    public void flush() throws IOException {
        o.write(buf, 0, pos);
        pos = 0;
    }
}
```

```
class LineBufferOutput extends BufferOutput {
    LineBufferOutput(OutputStream o) { super(o); }
    public void putchar(char c) throws IOException {
        super.putchar(c);
        if (c == '\n') flush();
    }
}
```

```
class Test {
    public static void main(String[] args) throws IOException {
        LineBufferOutput lbo = new LineBufferOutput(System.out);
        lbo.putstr("lbo\nlbo");
        System.out.print("print\n");
        lbo.putstr("\n");
    }
}
```

Hiding (by Class Method)

- If a class C declares or inherits a static method m, then m is said to hide any method m', where the signature of m is a subsignature of the signature of m', in the superclasses and superinterfaces of C that would otherwise be accessible to code in C.

Hiding Example

```
class Super {
    static String greeting() { return "Goodnight"; }
    String name() { return "Richard"; }
}
class Sub extends Super {
    static String greeting() { return "Hello"; }
    String name() { return "Dick"; }
}
class Test {
    public static void main(String[] args) {
        Super s = new Sub();
        System.out.println(s.greeting() + ", " + s.name());
    }
}
```

Shadowing Example

```
class Test {
    static int x = 1;
    public static void main(String[] args) {
        int x = 0;
        System.out.print("x=" + x);
        System.out.println(", Test.x=" + Test.x);
    }
}
```

* Since the scope of a class variable includes the entire body of the class, the class variable x would normally be available throughout the entire body of the method main. In this example, however, the class variable x is shadowed within the body of the method main by the declaration of the local variable x.

Shadowing Example

```
import java.util.*;
class Vector {
    int val[] = { 1 , 2 };
}

class Test {
    public static void main(String[] args) {
        Vector v = new Vector();
        System.out.println(v.val[0]);
    }
}
```

* *using the class Vector declared here in preference to the generic class java.util.Vector that might be imported on demand.*

Composition

- Is-a and Has-a concept
  - Tiger “is a” Cat - Inheritance
  - Cat “has” 4 legs - Composition
- Leg { }
- <Base> Cat { Leg() } - Composition
- <Derived> Tiger extends Cat - Inheritance

Composition over Inheritance

- All relationship are not pre-defined.
- Runtime behavior can change.
- Multiple inheritance can be a requirement
- Inheritance creates Fragile code

```
class Object
{
  public:
     void update() {};
     void draw() {};
     void collide(Object objects[]) {};
};

class Visible extends Object
{
  public:
    void draw() { // You must draw something };
};

class Solid extends Object
{
  public:
    void collide(Object objects[]) {
     // if collide, you are solid
    };
};

class Movable extends Object
{
  public:
    void update() { // if update, you are good};
};
```

```
class Cloud - which is Movable and Visible, but not Solid

class Building - which is Solid and Visible, but not Movable

class Player - which is Solid, Movable and Visible
```

How composition provides solution
https://en.wikipedia.org/wiki/Composition_over_inheritance

Polymorphism - Under hood

- Polymorphism in Java is a concept by which we can perform a single action in different ways.
- compile-time polymorphism and runtime polymorphism.
- We can perform polymorphism in java by method overloading and method overriding.
- If you overload a static method in Java, it is the example of compile time polymorphism.

Runtime Polymorphism

- Dynamic Method dispatch - runtime resolution of the method call.

```
class Base{}
class Derived extends Base{}
Base b=new Derived();//upcasting
```

```
interface I{}
class A{}
class B extends A implements I{}

B IS-A A
B IS-A I
B IS-A Object
```

Byte-code view

- invokestatic - invokestatic instruction means the method that will be invoked is a method of a class not an instance method. Also, they are called static because you need to use static keyword in Java to say that a method is a class method, not an instance method.
- invokespecial - invokespecial invokespecial instruction indicates an invocation of private instance methods, superclass methods or constructors.
- invokevirtual - invokevirtual invokevirtual instruction is generated for methods with public, package private or default and protected access modifiers. Those methods are virtual and could be overridden by subclasses.
- invokeinterface - invokeinterface instruction tells JVM that it should invoke a class implementation of an interface method.
- invokedynamic - instruction that facilitates the implementation of dynamic languages

JVM Method Invocation

- Before invoking a method, JVM performs two actions: method resolution and method lookup.

- Every method invocation instruction has an index to an entry in a class constant pool. The entry consists of a method name and a method descriptor.

- **invokestatic** - first time method resolution and lookup, not from second time.

- **invokespecial** same like invokestatic with additional check - instance is not null.

- **invokevirtual** - method resolution can save the method index and use it for method lookup the next time the method is invoked.

- **invokeinterface** - it completes both method resolution and method lookup every time a method is invoked.

JVM Method Invocation

```
def add(a, b)
  a + b
end
```

- We will get to know the type of a and b at runtime.
- How JVM will deal with it ?

Rescue from reflection ?

```
Method substring = String.class.getDeclaredMethod
                    ("substring", int.class, int.class);
String object = "Hello, World!";
Object hello = substring.invoke(object, 0, 5);
System.out.println(hello); // will print out "Hello"
```

- Slower than a direct invocation of a method

- invokedynamic resolves it

- Programmatically use MethodHandle

```
MethodHandles.Lookup lookup = MethodHandles.lookup();
MethodType methodType = MethodType.methodType(String.class, int.class, int.cla
MethodHandle substring = lookup.findVirtual(String.class, "substring",
methodType); //findStatic; findSpecial
Object hello = substring.invoke("Hello, World!", 0, 5);
System.out.println(hello); // will print out "Hello"
```

Poly, Mono and Bi - Morphic calls

- Polymorphic calls comes on a cost.
- JIT compiler can understand if polymorphism has not been used.
- It can convert calls into Mono or Bi - Morphic calls

References

- JLS - https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html

- JVM Spec - https://docs.oracle.com/javase/specs/jvms/se8/html/

- InvokeDynamic - https://www.baeldung.com/java-invoke-dynamic

- Method invocation inside JVM - https://www.lohika.com/methods-invocation-inside-of-a-java-virtual-machine

- Some reference from http://bangalorejug.org/Downloads/