Programming Fundamentals 2

Pierre Talbot

25 March 2021

University of Luxembourg

Chapter VII. (Almost) Everything is Object

Object class

(Almost) Everything is object

All classes created automatically inherit from an existing class called `Object`.

```
class Weapon { ... }
// is equivalent to
class Weapon extends Object { ... }
```

This class contains many methods that can be overridden in the subclasses.

Almost?

Only primitive types are not objects, thus do not inherit from `Object`. But, they have object equivalent such as `Double`, `Integer`, ... For instance, the class `Integer`[1] is written as:

```
public class Integer {
  private int value;
  public Integer(int v) { this.value = v; }
  // ...
}
```

Autoboxing cast

Autoboxing is a special casting mechanism in Java to automatically cast a primitive type to its class equivalent:

```
static void f(Double d) { ... }
f(2.4); // call f with a primitive type automatically casted into Double.
```

The converse operation exists too and is called *unboxing*.

[1] `https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/Integer.html`

`Object` class

```
class Object {
  // Utility methods
  public String toString() { ... }
  protected Object clone() throws CloneNotSupportedException { ... }
  public boolean equals(Object obj) { ... }
  public int hashCode() { ... }
  // Thread related.
  public final void notify() { ... }
  public final void notifyAll() { ... }
  public final void wait() throws InterruptedException { ... }
  public final void wait(long timeout)
    throws InterruptedException { ... }
  public final void wait(long timeout, int nanos)
    throws InterruptedException { ... }
  // Garbage collector related.
  protected void finalize() throws Throwable { ... }
  // Reflection related.
  public final Class<?> getClass() { ... }
}
```

We will focus on the utility methods which are the most common.

Utility methods

The apparent simplicity of these methods is *deceptive* and they must be implemented carefully.

Effective Java, 3rd Edition, Joshua Bloch

- `toString`: Item 12: Always override `toString`
- `clone`: Item 13: Override `clone` judiciously
- `equals`: Item 10: Obey the general contract when overriding `equals`
- `hashCode`: Item 11: Always override `hashCode` when you override `equals`

Introductory exercise

Instructions

```
git clone https://github.com/ptal/PF2-lab-B.git
cd PF2-lab-B
mvn compile
mvn exec:java -Dexec.mainClass="lab.B.Main" -q
```

Add to the right classes (possibly `Weapon`, `Axe` and `Hammer`) the method `toString`:

```
@Override public String toString() { ... }
```

Correction

toString

```
public abstract class Weapon {
  protected double damage;
  // ...
  @Override public String toString() {
    return damage + " damages";
  }
}
public class Hammer extends Weapon {
  // ...
  @Override public String toString() {
    return "Hammer of " + super.toString();
  }
}
```

- Although `Weapon` is only a concept, we implement `toString` in order to reuse its code in subclasses (DRY principle).
- To remember: always override `toString` (Item 12).

Copying

`clone` method

We distinguish three kinds of copies:

- Aliasing: Only the reference of the object is copied.

```
ArrayList<Integer> i = new ArrayList();
ArrayList<Integer> j = i;
j.add(3); // modify both i and j (see Chapters 2 and 3).
```

- Shallow copy: The attributes of the object are copied by aliasing.

```
public class IntegerList {
  public int x;
  public Array<Integer> y;
  public IntegerList shallowCopy() {
    return new IntegerList(x, y);
  }
}
// ...
IntegerList p1 = new IntegerList(3, new ArrayList());
IntegerList p2 = p1.shallowCopy();
p2.x = p2.x + 1; // modify only x in p2.
p2.y.add(3); // modify the array of both p1 and p2.
```

`clone` method

- Deep copy: The attributes are themselves copied using `clone`:

```
class IntegerList implements Cloneable {
  public int x;
  public Array<Integer> y;
  @Override public IntegerList clone() {
    return new IntegerList(x, y.clone());
  }
}
// ...
IntegerList p1 = new IntegerList(3, new ArrayList());
IntegerList p2 = p1.clone();
p2.x = p2.x + 1; // modify only x in p2.
p2.y.add(3); // modify the array of p2 only.
```

Additional comments on `clone`

- Usually, `clone` is implemented for deep copies. It is always good to mention the kind of copy in the documentation.
- The interface `Cloneable` must be used to indicate a class implements `clone`.
- According to Java documentation, although not strict requirements, you should have:

```
x.clone() != x
x.clone().getClass() == x.getClass()
x.clone().equals(x)
```

Your turn!

Add `clone` methods to the right classes in Lab B.

Covariant return type

When using `clone`, we must actually cast the object, *e.g.*,

```
Axe axe1 = new Axe();
Axe axe2 = (Axe)axe1.clone(); // Because clone returns an object.
```

We saw that two signatures are override-equivalent if they have the same name and the same parameters types. The return type can be *covariant*[2]: the return type can be a subtype of the return type of the parent's method. Therefore we can write:

```
@Override public Axe clone() // instead of Object clone()
```

while preserving override-equivalent signatures.

[2] http://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29

Equality

What does it mean to be equal?

From a mathematical perspective, we define equivalent things in a set S using an equivalence relation $\theta \subseteq S \times S$.

Equivalence relation

- Reflexive: $\forall x \in S,\ (x, x) \in \theta$,
- Symmetric: $\forall x, y \in S,\ (x, y) \in \theta \Leftrightarrow (y, x) \in \theta$,
- Transitive: $\forall x, y, z \in S,\ (x, y) \in \theta \wedge (y, z) \in \theta \Rightarrow (x, z) \in \theta$.

- The method `boolean equals(Object other)` should specify an equivalence relation (not always possible however).
- By default, `Object.equals` only compares the references. This corresponds to the *smallest equivalence relation*.

Additional requirements

We have two additional requirements for equivalence relations over objects:

- `x.equals(null)` must always be `false`,
- Consistent: `x.equals(y) == x.equals(y)` (multiple invocations return the same result).

Why not `boolean equals(Axe a)` instead of `boolean equals(Object o)`

Because `equals` would not be override-equivalent to `Object.equals` anymore, which means that the implementation of `equals` is selected at compile-time by overloading.

Your turn!

- Implement `equals` for the right classes.
- Add a new class:

```
public class MithrillAxe extends Axe {
  private boolean madeByDwarf;
  // ...
}
```

  Add the methods `toString`, `clone` and `equals` to this class, and test them.

- Can we guarantee `MithrillAxe.equals` to be an equivalence relation?

Implementing the `equals` method (part 1a)

```
public abstract class Weapon {
  @Override public boolean equals(Object o) {
    if(o == this) {
      return true;
    }
    else if (!(o instanceof Weapon)) {
      return false;
    }
    else {
      return ((Weapon)o).damage == damage;
    }
  }
}
```

Implementing the `equals` method (part 1b)

We must override `equals` in `Hammer` and `Axe` too, otherwise an axe and a hammer with the same amount of damages would be considered equal:

```
public class Axe extends Weapon {
  @Override public boolean equals(Object o) {
    return (o instanceof Axe) && super.equals(o);
  }
}
public class Hammer extends Weapon {
  @Override public boolean equals(Object o) {
    return (o instanceof Hammer) && super.equals(o);
  }
}
```

Implementing the `equals` method (part 2)

```
public class MithrillAxe extends Axe implements Cloneable {
  @Override public boolean equals(Object o) {
    if(o == this) {
      return true;
    }
    else if (!(o instanceof MithrillAxe)) {
      return false;
    }
    else {
      return super.equals(o)
        && ((MithrillAxe) o).madeByDwarf == madeByDwarf;
    }
  }
```

Unfortunately, this method is not symmetric:

```
MithrillAxe a1 = new MithrillAxe(true);
Axe a2 = new Axe();
a2.equals(a1); // is true, but
a1.equals(a2); // is false.
```

Implementing the `equals` method (part 3)

```
public class MithrillAxe extends Axe implements Cloneable {
  @Override public boolean equals(Object o) {
    if(o == this) {
      return true;
    }
    else if(o.getClass() == Axe.class) {
      return super.equals(o);
    }
    else if (!(o instanceof MithrillAxe)) {
      return false;
    }
    else {
      return super.equals(o)
        && ((MithrillAxe) o).madeByDwarf == madeByDwarf;
    }
  }
```

That's the best we can do, but this method is still not transitive:

```
MithrillAxe a1 = new MithrillAxe(true);
Axe a2 = new Axe();
MithrillAxe a3 = new MithrillAxe(false);
a1.equals(a2); // is true and
a2.equals(a3); // is true, but
a1.equals(a3); // is false.
```

A use-case of the `equals` method

The operation `ArrayList.contains` relies on `equals` to detect objects that are equal.

```
ArrayList<Weapon> store = new ArrayList<>();
store.add(new Axe());
store.add(new MithrillAxe(true));
System.out.println(store.contains(new Axe())); // prints true.
```

If `equals` is not implemented, the semantics of `contains` changes, and rely on reference equality.

Hash

Hashing

A hash is a function $\mathit{hashCode} : T \to \mathbb{N}$. That is, it maps a type `T` to an integer.

- `x.equals(y)` implies `x.hashCode() == y.hashCode()`.
- However, the converse is not necessarily true.
- To guarantee this implication, **you should always override** `hashCode` **if you override** `equals`**.**
- Normally, it is faster to compute a hash than comparing two objects.

An example

```
public abstract class Weapon {
  protected int damage;
  @Override public int hashCode() {
    return damage;
  }
}

public class MithrillAxe extends Axe implements Cloneable {
  private boolean madeByDwarf;
  @Override public int hashCode() {
    return super.hashCode() * 10 + (madeByDwarf ? 1 : 0);
  }
}
```

In that case, we have `x.hashCode() == y.hashCode()` $\Leftrightarrow$ `x.equals(y)` (assuming no overflow). But that is not always the case, imagine the hash function of an array.

A use-case of `hashCode`: HashSet

`java.util.Set` is an interface for collection implementing the semantics of a (mathematical) set. It has the property that if `x.equals(y)`, then only x or y is in the set.

From a practical standpoint, a set can be implemented by various data structures with different tradeoff.

```
HashSet<Weapon> store = new HashSet<>();
store.add(new Axe());
store.add(new MithrillAxe(true));
store.add(new MithrillAxe(false));
store.add(new Axe());
System.out.println(store.size()); // prints 3.
```

Another possible choice is `TreeSet`, however this one needs the class to implement the `Comparable` interface (imposes a total order on the elements).

Summary

- Object is the parent class of all classes.
- Four essential methods to override from Object.
- A lot of subtleties... Necessary to study these to become proficient in Java.

Correction of the exercises: `git checkout correction` in your repository (you need to commit your changes first).