The Object Class

Mark Allen Weiss
Copyright 2000

java.lang.Object

- All classes either extend Object directly or indirectly.
  - Makes it easier to write generic algorithms and data structures
  - Makes it easy to treat all objects same (for instance with respect to automatic calls to toString)
- Every non primitive IS-A Object
- Object has several methods.
- Object is not an abstract class, so all methods have implementations

Important Methods In Object

- getClass
- toString
- equals
- hashCode
- clone
- finalize
- wait
- notifyAll
wait and notifyAll

- Used for threading
- We'll discuss those in a few weeks, but when we do, remember that these methods are defined in Object.

getClass

- Returns a Class object that represents information about the type of the object.
- Every type has a single Class object.
- Two objects with same Class are of same type

```java
class Person { ... }
class Employee extends Person { ... }
Object o1 = new Person( ... );
Object o2 = new Employee( ... );
Object o3 = new Employee( ... );
Object o4 = new Person[ 5 ]; // Arrays are objects
Class c1 = o1.getClass(); // Returns Person.class
Class c2 = o2.getClass(); // Returns Employee.class
Class c3 = c1.getClass(); // Returns Employee.class
Class c4 = o4.getClass(); // Returns Person[].class
// Note: c2 == c1 is false, c2 == c3 is true
```

Class objects

- Will discuss more details when we talk about reflection.
- Can get name of the class with getName.
- Also, toString is defined.

```java
Object o1 = ...; // can reference any object
Class c1 = o1.getClass();
System.out.println("Type of o1 is "+ c1.getName( ) );
System.out.println("Type of o1 is "+ o1.toString( ) );
System.out.println("Type of o1 is "+ c1 );
```
toString

- Automatically called on an object when the object is concatenated with a String.
- The default prints the name of the class and object's hash code; you can expect that different objects (even with same state) will be identified differently by toString.
- Can override the default to print out your meaningful version.
- Common to chain calls to superclass.
- Don’t hard code class name into toString

Example of toString With Chaining

class Person
{
    public String toString()
    {
        return getClass().toString() + " " + getName();
    }
    public String getName()
    {
        return name;
    }
    private String name;
    private Date birthdate;
}
class Student extends Person
{
    public String toString()
    {
        return super.toString() + " " + getID();
    }
    public int getID()
    {
        return id;
    }
    private int id;
}

equals

- Used to determine if two references refer to Objects that have same state.
- Default in Object is to return true only if the two references are not null and are equal (cannot invoke equal with a null reference).
- Can override default; that’s what String does, for example.
- The method to override is public boolean equals( Object other )
- Common pitfall to use wrong signature.
Contract of equals

- If comparing with null, must return false.
- Reflexive: \( x.equals(x) \) must be true
- Symmetric: \( x.equals(y) \) is the same as \( y.equals(x) \), if neither is null
- Transitive: \( x.equals(y) \) and \( y.equals(z) \) both being true implies \( x.equals(z) \) must be true (if exactly one is true, \( x.equals(z) \) must be false).
- \( x.equals(y) \) should always give the same answer, unless the states of \( x \) or \( y \) change.

So What's The Big Deal?

- Contract is trickier than it looks when comparing base class objects with derived class objects.
  - some implementations crash because of null or assumption of correct type
  - some implementations uses instanceof in both classes and fail the symmetric requirement
  - there's an additional requirement that hashCode must be implemented consistent with equals
- JDK 1.3 source has over 130 incorrect equals implementations

Standard Preamble

- Generally, two objects should only compare equal if types match exactly, or types are in the same hierarchy, but equals is never overridden beyond initial base class (i.e. equals is final).
- In second case, can probably use instanceof.
- In first case, start code with:
  ```java
  public boolean equals( Object obj )
  {
    if( obj == null || getClass() != obj.getClass() ) return false;
  }
  ```
- When overriding equals in derived class, chain up to base class via super.
Example of `equals` With Chaining

```java
class Person
{
    public boolean equals(Object obj) {
        if (obj == null || getClass() != obj.getClass())
            return false;
        Person other = (Person) obj;
        return getName().equals(other.getName());
    }
}
class Student extends Person
{
    public boolean equals(Object obj) {
        if (!super.equals(obj))
            return false; // handles null and same class
        Student other = (Student) obj;
        return getID() == other.getID();
    }
}
```

hashCode

- Used in `Hashtable`, `HashSet`, and `HashMap` containers
- Returns an int
- Contract is that if `x.equals(y)` is true, `x.hashCode()` must equal `y.hashCode()`
- Expectation is that if `x.equals(y)` is false, hash codes are almost certainly different
- Same principles as before: use chaining
- If you mess up `hashCode`, your objects will not be found in the hashing containers.

Example of `hashCode` With Chaining

```java
class Person
{
    public int hashCode() {
        return getName().hashCode();
    }
}
class Student extends Person
{
    public int hashCode() {
        return super.hashCode() ^ getID(); // exclusive or
    }
}
Cloning

- **Object** defines a clone method that returns a new **Object** of the same type, with the expectation of the same state.
- Only objects that implement the **Cloneable** interface can call `clone` without generating a `CloneNotSupportedException`.
- The **Cloneable** interface is a tagged interface; no methods, just something you have to say.
- The implementation in **Object** is magic:
  - Does a shallow copy, so others can chain up to it
  - If called directly, however, will throw an exception

Tricky Stuff

- Never use a constructor to create the new object; instead delegate to `super.clone`.
- If possible, use `clone` on the additional reference fields in the derived class.
- Strings don't need cloning
- Implement the **Cloneable** interface
- Make `clone` method public

Example of `clone` With Chaining

```java
class Person {
    public Object clone() throws CloneNotSupportedException {
        Object copy = super.clone();
        // no other deep copies needed
        return copy;
    }
}
class Student extends Person {
    public Object clone() throws CloneNotSupportedException {
        // no other deep copies needed
        return super.clone();
    }
}
class Undergrad extends Student {
    // ...
}
```
finalize

- Not a reliable routine; might never be invoked
- If invoked by VM, will never be invoked again by VM
- Leave protected; should only be called by garbage collector
- Usual stuff if you implement: chain to the superclass (last!)
- Also, try to catch exceptions
- Probably never need to write finalize unless you are doing demos of the garbage collector

Summary

- Object class is root of all inheritance
- Defaults provided for all methods
- Implementations are tricky for classes that use inheritance
- equals and hashCode go together