Java's Object Methods

Technion - Israel Institute of Technology

Last Update : Guy Suday, August 2018
In This Tutorial

• In Java, the Object class contains a number of useful methods it is important to understand.

• In this tutorial, we’ll look at several of them:
  • clone()
  • equals()
  • hashCode()

• And in addition, we will learn about the interfaces:
  • Comparable / Comparator
• The equality operator == returns true if and only if both its operands are the same.
  – Compares values of primitive types.
  – Compares identities of objects:

```java
Integer i1 = new Integer("3");
Integer i2 = new Integer("3");
Integer i3 = i2;

i1 == i1;  // Result is true
i1 == i2;  // Result is False
i2 == i3;  // Result is true
```
Object Equality

• To compare between two objects the `boolean equals(Object o)` method is used:
  – Default implementation compares using the equality operator.
  – Most Java API classes provide a specialized implementation.
  – Override this method to provide your own implementation.

```java
i1.equals(i1)  // Result is true
i1 == i2;      // Result is false
i1.equals(i2)  // Result is true
```
Example: Object Equality

```java
public class Name {
    String firstName, lastName;

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Name)) return false;
        Name other = (Name)o;
        return firstName.equals(other.firstName) &&
            lastName.equals(other.lastName);
    }
}
```

More on the subtleties of equals() later in the course...
equals(), hashCode()

• Are defined in Object
• `equals(Object o)`
  – Used for state equality testing
    • as opposed to operator `==` (used for reference equality)
  – Used to test containment of object in collection
  – Used for storing an object in hash-based collection

• `hashCode()`
  – Returns an `int` value representing the object
  – Used for storing an object in hash-based collections
Overriding `equals()`

- Default implementation of `equals()`
  - is based on the `==` operator
  - Two objects are equal if and only if they are the same object
- Sometimes a program specific notion of equality is required:

```java
Date d1 = new Date(2007, 7, 7);
Date d2 = new Date(2007, 7, 7);
(d1 == d2) == false;
d1.equals(d2) == true;
```
equals() Contract

• The `equals()` method implements an equivalence relation:
  – It is **reflexive**
    • `a.equals(a)` is true
  – It is **symmetric**
    • `a.equals(b) ⇔ b.equals(a)`
  – It is **transitive**
    • `a.equals(b) and b.equals(c) ⇒ a.equals(c)`
  – It is **consistent**
    • repeated calls to the method must yield the same result unless the arguments are modified in-between  
  – **No object equals null**
    • `a.equals(null) == false`
equals() Contract

- The contract is defined in Java documentation:

- The contract is not enforced by the compiler or the runtime system!
A Naive Example

class Point {
    private final int x;
    private final int y;
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Point))
            return false;
        return ((Point)o).x == x
            && ((Point)o).y == y;
    }
}

- Meets all contract demands, right?
  - As long as inheritance is not involved...
class ColorPoint extends Point {
    private final Color color;
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof ColorPoint))
            return false;
        return super.equals(o) &&
                ((ColorPoint)o).color.equals(color);
    }
}

ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);

p2.equals(p1); // true
p1.equals(p2); // false 😞
class Point {
    private final int x;
    private final int y;
    public boolean equals(Object o) {
        if (o == null) return false;
        if (o.getClass() != this.getClass())
            return false;
        return ((Point)o).x == x
            && ((Point)o).y == y;
    }
}

- Meets all the contract’s requirements.
- Might be too strict
  - An object is never equals to an instance of its super class
  - Even if the derived class doesn’t add fields
Another Solution

• The idea: two objects must agree on their equality

```java
class Point {
    protected boolean eq(Object o) {
        if (!(o instanceof Point))
            return false;
        return ((Point)o).x == x && ((Point)o).y == y;
    }
    public boolean equals(Object o) {
        return (this.eq(o) && ((Point)o).eq(this));
    }
}

class ColorPoint extends Point {
    @Override
    protected boolean eq(Object o) {
        if (!(o instanceof ColorPoint))
            return false;
        return super.eq(o) && ((ColorPoint)o).color == color;
    }
}
```

Is this safe?
Another Solution - Example

- The idea: two objects must agree on their equality

```java
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);

p2.equals(p1); // false
p1.equals(p2); // false
```
hashCode() Contract

• Must generate equal values for **equal** objects
  \[ x.equals(y) \Rightarrow x.hashCode() = y.hashCode() \]

• **It is consistent**
  – Repeated calls to the method must yield the same int unless the objects are modified in-between.

• **We see that** `hashCode()` **is closely related to** `equals()`
  – if you override equals, you should override `hashCode`
Comparable Interface

- Used for comparing between objects.
  - Single method: int compareTo(T o)
- Why?
  - equals() is boolean – can’t be used for sorting
- A problem
  - we want compareTo to take a parameter of the same type as the receiver – why?
  - We would like a covariant parameter
- The Solution
  - Using generics:
    ```java
    public interface Comparable<T> { 
        int compareTo(T o);
    }
    ```
public class Person implements Comparable<Person> {
    public String name;
    public int age; // Order by age then by name

    public int compareTo(Person p) {
        int diff = this.age - p.age;
        if (diff != 0)
            return diff;
        // Use compareTo of String
        return this.name.compareTo(p.name);
    }
}

Implementing Comparable
class Utils {
    public static <T extends Comparable<T>>
    void bubbleSort(List<T> lst) {
        int top = lst.size();
        for (int i = 0; i < top - 1; ++i) {
            for (int j = 1; j < top - i; ++j) {
                T a = lst.get(j - 1);
                T b = lst.get(j);
                if (a.compareTo(b) > 0) {
                    lst.set(j - 1, b);
                    lst.set(j, a);
                }
            }
        }
    }
}
Comparator Interface

• Used for comparing objects of a class that does not implement Comparable (or not with the criteria we want)

```java
public interface Comparator<T> {
    public int compare(T o1, T o2);
}

public class Utils {
    public static <T> boolean isSorted(Comparator<T> comp, T[] ts) {
        for (int i = 1; i < ts.length; ++i)
            if (comp.compare(ts[i], ts[i-1]) < 0)
                return false;
        return true;
    }
}
```
• **Comparators** allow us to implement several different comparisons for the same type.
  – Define several classes, each implements **Comparator**.
• **Comparable** must be implemented by the type being compared.

```java
public class Student implements Comparable<Student> {
    public int id;
    public String firstName, lastName;
    public int compareTo(Student s) {
        // compare by ID
    }
}
public class CompareByFirstName implements Comparator<Student> {
    // implement compare according to student first name
}
public class CompareByLastName implements Comparator<Student> {
    // implement compare according to student last name
}
```
Comparator Interface

- An Ad-hoc λ — *expr* comparator class use:

```java
public class Student implements Comparable<Student> {
    public int id;
    public String firstName, lastName;
    public int compareTo(Student s) {
        // compare by ID
    }
}

public class Program {
    public static void main(String[] args) {
        Comparator<Student> firstNameComp =
            (s1, s2) -> s1.firstName.compareTo(s2.firstName);
        ...
        Comparator<Student> lastNameComp =
            (s1, s2) -> s1.lastName.compareTo(s2.lastName);
    }
}
```
clone()

• Defined in `Object`
• Creates an identical copy
  – Copies *pointers* to fields (does not copy fields of fields)
  – Makes a shallow copy
• It is intended that these will return true:
  1. `x.clone() != x`;
     - Modifying `x` no longer affects `x.clone()`
  2. `x.clone().getClass() == x.getClass()`;
  3. `x.equals(x.clone())`;
     ▪ However, none of these 3 are a requirement.
• if the object’s class doesn’t implement Cloneable, a
  `CloneNotSupportedException` is thrown
Cloneable

- Tagging interface – has no methods.
- Why isn’t clone() defined in Cloneable?
  - Has an efficient common implementation in Object.
  - Interfaces can't provide implementation.
- Why isn’t Cloneable a class then?
  - Will occupy the one inheritance slot.
- So why not make all classes cloneable?
  - No more singletons.
- Compromise: clone() is protected.
- This makes cloning an almost “dynamic” feature.

Before Java 8 (default methods)
• **So, `Object.clone()` does the following:**
  – Check if its class implements `Cloneable`. If not, throws a `CloneNotSupportedException`.
  – Allocate memory for a new object of the same type.
  – Shallow copy all the object’s fields.

• **All this is done in native code, hence efficiently.**
Why clone() is protected?

- Sometimes classes represent things that should not be cloned
- Before making clone() available to use, a class has to implement Cloneable
- The class can decide to make clone public after implementing Cloneable
CloneNotSupportedException Example

```java
class X {
    public X getAClone() throws CloneNotSupportedException {
        return (X) this.clone();
    }
}

public class Program {
    public static void main(String[] args) throws Exception {
        (new X()).getAClone();
    }
}
```

Output:
Exception in thread "main" java.lang.CloneNotSupportedException: X
    at java.lang.Object.clone(Native Method)
    at X.getAClone(Program.java:18)
    at Program.main(Program.java:11)

Just for demonstration, bad programming
public class X implements Cloneable {
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

Why Override clone()?

- First, Object’s clone is a protected method – Its visibility can be increased:

    The standard way to start a clone(), in any class
Why Override clone()?

Class A
private int x;
private int y;
...

A a = new A();
X   0
Y   0
Memory Addr: 1001

A a1 = (A)a.clone();
X   0
Y   0
Memory Addr: 2029

Class B
private int x;
private A y;
...

B b = new B();
x   0
y   4049
Memory Addr: 2787

A a = new A();
X   0
Y   0
Memory Addr: 4049

B b1 = (B)b.clone();
X   0
y   4049
Memory Addr: 3034
Why Override clone()?

• Sometimes a deeper copy of a field is required

```java
public class X implements Cloneable {
    private A a;
    @Override
    public Object clone() throws CloneNotSupportedException {
        X x = (X) super.clone();
        if (a != null)
            x.a = (A) a.clone();
        return x;
    }
}
```
Pitfalls in Overriding clone()

- Implementing `clone()` using a constructor

```java
public class X implements Cloneable {
    public Object clone() throws CloneNotSupportedException {
        return new X();
    }
}
```

1st try

```java
public class Y extends X {
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

Clone returns wrong type!

2nd try

```java
public class Y extends X {
    public Object clone() throws CloneNotSupportedException {
        Y y = (Y) super.clone();
        ...
    }
}
```

Exception! Casting a object of class X to Y
• Using Constructors to copy sub-objects:

```java
public class X implements Cloneable {
    private A a;
    @Override
    public Object clone () throws CloneNotSupportedException {
        X x = (X)super.clone();
        if(a != null)
            x.a = new A();
        return x;
    }
}
```

If this.a holds a subclass of A, x.a may be different from this.a, meaning y.equals(y.clone()) might return false.