Java's Object Methods

Technion - Israel Institute of Technology

Updated: April 2016

236703 - Object-Oriented Programming
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
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.
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.
CloneNotSupportedException Example

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

public class Program {
    public static void main(String[] args) throws Exception {
        (new B()).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)
Why Override clone()?

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

```java
public class X implements Cloneable {
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

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

- Problems?
Why Override clone()?

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

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

A a = new A();
x
y
Memory Addr: 1001

A a1 = (A)a.clone();
x
y
Memory Addr: 2029

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

B b1 = (B)b.clone();
x
y
Memory Addr: 3034

Memory Addr: 4049
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
Pitfalls in clone() – Cont.

• 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.

• Can “throws CloneNotSupportedException” be omitted if we do support cloning?
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;
```

== or equals?...
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`
The contract is defined in Java documentation:
- [http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object))
- The contract is not enforced by the compiler or the runtime system!
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;
    public boolean equals(Object o) {
        if (!(o instanceof ColorPoint))
            return false;
        return super.equals(o) &&
                ((ColorPoint)o).color == color;
    }
}

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

p2.equals(p1); // ???
p1.equals(p2); // ???
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;
    }
}
```
hashCode() Contract

• Must generate equal values for equal objects
  - \( x.equals(y) \implies x.hashCode() = y.hashCode() \)

• It is consistent
  - Repeated calls to the method must yield the same \( \text{int} \) unless the objects are modified in-between.

• We see that hashCode() is closely related to equals()
  - if you override equals, you should override hashCode
public class SimpleHashSet {
    private List<Object>[][] table = ...

    public void add(Object o) {
        int i = o.hashCode() % table.length;
        List<Object> lst = table[i];
        if (lst == null) {
            lst = new LinkedList<Object>();
            table[i] = lst;
        }
        lst.add(o); // if not there already...
    }
}
public class SimpleHashSet {
    ...

    public boolean exists(Object o) {
        int i = o.hashCode() % table.length;
        List<Object> lst = table[i];
        if (lst == null) return false;
        for (Object elem : lst)
            if (elem.equals(o)) return true;
        return false;
    }
}
class Point {
    final private int x, y;
    public boolean equals(Object o) {
        if (!(o instanceof Point))
            return false;
        return ((Point)o).x == x;
    }
    public int hashCode() {
        return 19 * y;
    }
}

public static void main(String[] args) {
    SimpleHashSet<Point> s = new ...;
    Point p1 = new Point(1, 2);
    s.add(p1);
    Point p2 = new Point(1, 3);
    System.out.println(p1.equals(p2)); // true
    System.out.println(s.exists(p2)); // false
}
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
  – We would like a covariant parameter
• The Solution
  – Using generics:

```java
public interface Comparable<T> {
    int compareTo(T o);
}
```
Implementing Comparable

```java
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);
    }
}
```
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);
                }
            }
        }
    }
}
• Used for comparing objects of a class that does not implement Comparable (or not with the criteria we want)
Comparator Interface

- **Comparators** allow us to implement several different comparisons for the same type.
  - Define several classes, each implements `Comparable`.
- **Comparable** must be implemented by the type being compared.