CSE 331
Lecture 4

Comparing objects; cloning

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia
http://www.cs.washington.edu/331/

Natural ordering, Comparable, Comparator

Comparing objects

- Operators like < and > do not work with objects in Java.
  - But we do think of some types as having an ordering (e.g. `Date`s).
  - (In other languages, we can enable <, > with *operator overloading*.)

- **natural ordering**: Rules governing the relative placement of all values of a given type.
  - Implies a notion of equality (like `equals`) but also < and > .
  - **total ordering**: All elements can be arranged in A ≤ B ≤ C ≤ ... order.

- **comparison function**: Code that, when given two values *A* and *B* of a given type, decides their relative ordering:
  - A < B,   A == B,   A > B

The Comparable interface

- The standard way for a Java class to define a comparison function for its objects is to implement the `Comparable` interface.

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

- A call of **A**.`compareTo(`**B**`)` should return:

  a value < 0 if **A** comes "before" **B** in the ordering,

  a value > 0 if **A** comes "after" **B** in the ordering,

  or exactly 0 if **A** and **B** are considered "equal" in the ordering.

- **Effective Java Tip #12**: Consider implementing `Comparable`.

compareTo example

```
public class Point implements Comparable<Point> {
    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;   // same x, smaller y
        } else if (y > other.y) {
            return 1;    // same x, larger y
        } else {
            return 0;    // same x and same y
        }
    }

    // subtraction trick:
    // return (x != other.x) ? (x - other.x) : (y - other.y);
}
```

compareTo and collections

- Java's binary search methods call compareTo internally.

```
String[] a = {"al", "bob", "cari", "dan", "mike"};
int index = Arrays.binarySearch(a, "dan");  // 3
```

- Java's TreeSet/Map use compareTo internally for ordering.
  - Only classes that implement Comparable can be used as elements.

```
Set<String> set = new TreeSet<String>();
for (int i = a.length - 1; i >= 0; i--) {
    set.add(a[i]);
}
System.out.println(s);
// [al, bob, cari, dan, mike]
```

Flawed compareTo method

```
public class BankAccount implements Comparable<BankAccount> {
    private String name;
    private double balance;
    private int id;
    ...
    public int compareTo(BankAccount other) {
        return name.compareTo(other.name);  // order by name
    }

    public boolean equals(Object o) {
        if (o != null && getClass() == o.getClass()) {
            BankAccount ba = (BankAccount) o;
            return name.equals(ba.name)
                && balance == ba.balance && id == ba.id;
        } else {
            return false;
        }
    }
}
```

- What's bad about the above? Hint: See Comparable API docs.

The flaw

```
BankAccount ba1 = new BankAccount("Jim", 123, 20.00);
BankAccount ba2 = new BankAccount("Jim", 456, 984.00);

Set<BankAccount> accounts = new TreeSet<BankAccount>();
accounts.add(ba1);
accounts.add(ba2);
System.out.println(accounts);     // [Jim($20.00)]
```

- Where did the other account go?
  - Since the two accounts are "equal" by the ordering of `compareTo`, the set thought they were duplicates and didn't store the second.

compareTo and equals

- `compareTo` should generally be consistent with `equals`.
  - `a.compareTo(b) == 0` should imply that `a.equals(b)`.

- from `Comparable` Java API docs:
  - ... sorted sets (and sorted maps) without explicit comparators behave strangely when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the equals method.
  - For example, if one adds two keys a and b such that (!a.equals(b) && a.compareTo(b) == 0) to a sorted set that does not use an explicit comparator, the second add operation returns false (and the size of the sorted set does not increase) because a and b are equivalent from the sorted set's perspective.

What's the "natural" order?

```
public class Rectangle implements Comparable<Rectangle> {
    private int x, y, width, height;

    public int compareTo(Rectangle other) {
        // ...?
    }
}
```

- What is the "natural ordering" of rectangles?
  - By x, breaking ties by y?
  - By width, breaking ties by height?
  - By area? By perimeter?

- Do rectangles have any "natural" ordering?
  - Might we ever want to sort rectangles into some order anyway?

Comparator interface

```
public interface Comparator<T> {
    public int compare(T first, T second);
}
```

- Interface `Comparator` is an external object that specifies a comparison function over some other type of objects.
  - Allows you to define multiple orderings for the same type.
  - Allows you to define a specific ordering for a type even if there is no obvious "natural" ordering for that type.

Comparator examples

```
public class RectangleAreaComparator
        implements Comparator<Rectangle> {
    // compare in ascending order by area (WxH)
    public int compare(Rectangle r1, Rectangle r2) {
        return r1.getArea() - r2.getArea();
    }
}

public class RectangleXYComparator
        implements Comparator<Rectangle> {
    // compare by ascending x, break ties by y
    public int compare(Rectangle r1, Rectangle r2) {
        if (r1.getX() != r2.getX()) {
            return r1.getX() - r2.getX();
        } else {
            return r1.getY() - r2.getY();
        }
    }
}
```

Using Comparators

- TreeSet and TreeMap can accept a Comparator parameter.

```
Comparator<Rectangle> comp = new RectangleAreaComparator();
Set<Rectangle> set = new TreeSet<Rectangle>(comp);
```

- Searching and sorting methods can accept Comparators.

```
Arrays.binarySearch(array, value, comparator)
Arrays.sort(array, comparator)
Collections.binarySearch(list, comparator)
Collections.max(collection, comparator)
Collections.min(collection, comparator)
Collections.sort(list, comparator)
```

- Methods are provided to reverse a Comparator's ordering:

```
Collections.reverseOrder()
Collections.reverseOrder(comparator)
```

Cloning objects

Copying objects

- In other languages (common in C++), to enable clients to easily make copies of an object, you can supply a *copy constructor* :

```
// in client code
Point p1 = new Point(-3, 5);
Point p2 = new Point(p1);       // make p2 a copy of p1


// in Point.java
public Point(Point blueprint) {    // copy constructor
    this.x = blueprint.x;
    this.y = blueprint.y;
}
```

  - Java has some copy constructors but also has a different way...

Object clone method

```
protected Object clone()
        throws CloneNotSupportedException
```

- Creates and returns a copy of this object. General intent:
  - `x.clone() != x`
  - `x.clone().equals(x)`
  - `x.clone().getClass() == x.getClass()`
    - (though none of the above are absolute requirements)

- The `Object` class's `clone` method makes a "shallow copy" of the object, but by convention, the object returned by this method should be **independent** of this object (which is being cloned).

Protected access

```
protected Object clone()
        throws CloneNotSupportedException
```

- **protected**: Visible only to the class itself, its subclasses, and any other classes in the same package.
  - In other words, for most classes you are not allowed to call `clone`.
  - If you want to enable cloning, you must override `clone`.
    - You should make it `public` so clients can call it.
    - You can also change the return type to your class's type. (good)
    - You can also not throw the exception. (good)
  - You must also make your class implement the `Cloneable` interface to signify that it is allowed to be cloned.

The Cloneable interface

```
public interface Cloneable {}
```

- Why would there ever be an interface with no methods?
  - Another example: `Set` interface, a sub-interface of `Collection`

- **tagging interface**: One that does not contain/add any methods, but is meant to mark a class as having a certain quality or ability.
  - Generally a wart in the Java language; a misuse of interfaces.
  - Now largely unnecessary thanks to *annotations* (seen later).
  - But we still must interact with a few tagging interfaces, like this one.

- Let's implement clone for a `Point` class...

Flawed clone method 1

```
public class Point implements Cloneable {
    private int x, y;
    ...
    public Point clone() {
        Point copy = new Point(this.x, this.y);
        return copy;
    }
}
```

- What's wrong with the above method?

The flaw

```
// also implements Cloneable and inherits clone()
public class Point3D extends Point {
    private int z;
    ...
}
```

- The above `Point3D` class's `clone` method produces a `Point`!
  - This is undesirable and unexpected behavior.
  - The only way to ensure that the clone will have exactly the same type as the original object (even in the presence of inheritance) is to call the `clone` method from class `Object` with `super.clone()` .

Proper clone method

```
public class Point implements Cloneable {
    private int x, y;
    ...
    public Point clone() {
        try {
            Point copy = (Point) super.clone();
            return copy;
        } catch (CloneNotSupportedException e) {
            // this will never happen
            return null;
        }
    }
}
```

- To call `Object`'s `clone` method, you must use `try/catch`.
  - But if you implement `Cloneable`, the exception will not be thrown.

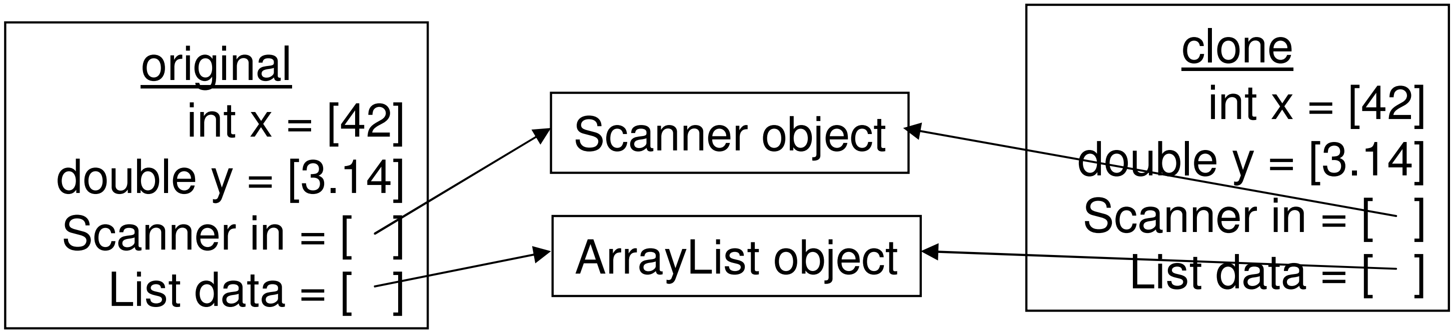
Flawed clone method 2

```
public class BankAccount implements Cloneable {
    private String name;
    private List<String> transactions;
    ...
    public BankAccount clone() {
        try {
            BankAccount copy = (BankAccount) super.clone();
            return copy;
        } catch (CloneNotSupportedException e) {
            return null;   // won't ever happen
        }
    }
}
```

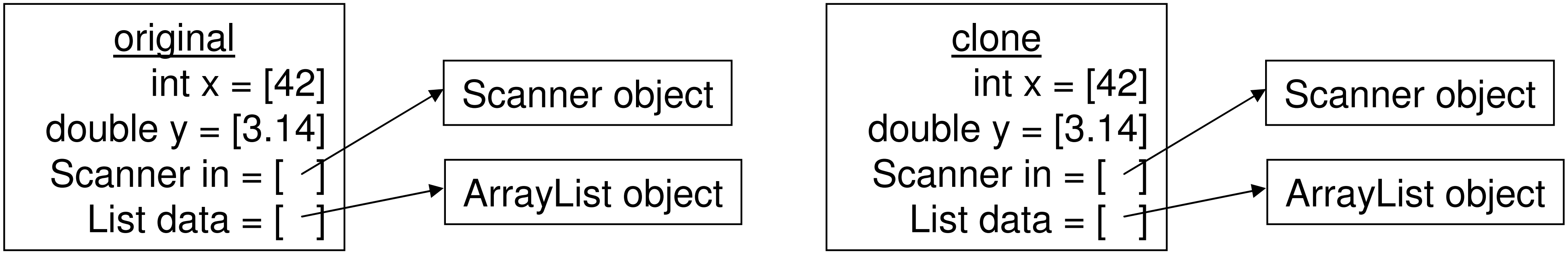
- What's wrong with the above method?

Shallow vs. deep copy

- **shallow copy**: Duplicates an object without duplicating any other objects to which it refers.



- **deep copy**: Duplicates an object's entire *reference graph*: copies itself and deep copies any other objects to which it refers.



  - `Object`'s `clone` method makes a shallow copy by default. (Why?)

Proper clone method 2

```
public class BankAccount implements Cloneable {
    private String name;
    private List<String> transactions;
    ...
    public BankAccount clone() {
        try {              // deep copy
            BankAccount copy = (BankAccount) super.clone();
            copy.transactions = (List<String>)
                                    transactions.clone();
            return copy;
        } catch (CloneNotSupportedException e) {
            return null;   // won't ever happen
        }
    }
}
```

- Cloning the list of transactions (and any other mutable reference fields) produces a deep copy that is independent of the original.

Effective Java Tip #11

- **Tip #11**: Override `clone` judiciously.

- Cloning has many gotchas and warts:
  - protected vs. public
  - flaws in the presence of inheritance
  - requires the use of an ugly tagging interface
  - throws an ugly checked exception
  - easy to get wrong by making a shallow copy instead of a deep copy