CSE 331

Cloning objects

slides created by Marty Stepp
http://www.cs.washington.edu/331/
Copying objects

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

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

```java
// 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...
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).

Object clone method
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.
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

```java
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.
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.

  ```
  original
  int x = [42]
double y = [3.14]
Scanner in = [ ]
List data = [ ]
  
  clone
  int x = [42]
double y = [3.14]
Scanner in = [ ]
List data = [ ]
  ```

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

  ```
  original
  int x = [42]
double y = [3.14]
Scanner in = [ ]
List data = [ ]
  
  clone
  int x = [42]
double y = [3.14]
Scanner in = [ ]
List data = [ ]
  ```

- Object's `clone` method makes a shallow copy by default. (Why?)
Proper clone method 2

```java
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 = new ArrayList<String>(transactions);
            return copy;
        } catch (CloneNotSupportedException e) {
            return null; // won't ever happen
        }
    }
}
```

- Copying the list of transactions (and any other modifiable 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