java.lang.Object: Equality

Lecture 14
java.lang.Object

- The root of all class hierarchies
  - This is a class called “Object”
  - There is also a class in java.lang called “Class”!
- Provides several useful methods
  - getClass() (Returns Class of the object instance)
  - String toString() (Returns String representing object value)
  - boolean equals(Object) (Returns true iff argument is equal to object)
  - int hashCode() (Returns an int “hash value” for object)
  - Object clone() [but the actual class of the object must implement interface java.lang.Cloneable] (Creates and returns a copy)
The getClass() Method

- Returns an instance of java.lang.Class
  - Generic class: Class<T>
  - String getName()
    - Name of the class as a string, eg “CseMajor”
- Think of it as representing the object’s class

```java
Student s1 = new OsuStudent();
Student s2 = new CseMajor();
System.out.println(s1.getClass().getName());
System.out.println(s2.getClass().getName());
if (s1.getClass() == s2.getClass()) {
    // ...
}
```
- Of course (?) java.lang.Class extends Object!
  - Try not to think about this too hard
Good Practice: Core Methods

- Always override `toString()`
  - Default implementation gives class name + @ + a meaningless hex number
  - eg “BankAccount@3d4606bf”

- Always override `equals()`
  - Default implementation checks object references for equality
    ```java
    Pencil p1 = new LeadedPencil();
    Pencil p2 = new LeadedPencil();
    assert(!p1.equals(p2));
    ```

- Always override `hashCode()`
  - Default implementation is memory address
  - What is a `hashCode`? Stay tuned...

- Overriding `clone()`: some tricky issues
  - “Shallow” vs. “deep” copies
Overriding toString()

- Spec in java.lang.Object
  - “A concise but informative representation that is easy for a person to read.”
- Automatically called when String needed
  ```java
  System.out.println(myAccount);
  String msg = "Cell phone: " + phoneNumber;
  ```
- Ideally provides *complete* information
  - Can be at odds with being “concise”
  - Information about *abstract* (ie interface) state
- Design decision: How specific to make spec?
  - Whatever is in spec, the client can use/exploit
  - Specific toString info ==> most useful to client
  - Vague toString info ==> most flexibility for future
Good Practice: String Conversion

- Provide matching constructor to create object from a String
  - String toString(): object --> String
  - Pencil(String): String --> new object

- Especially common for immutables
  - See java.lang.Integer
  - Notice how carefully toString() is documented
  - Caveat: Factory methods are better than constructors here (we’ll talk about these later)
Overriding equals()

- Spec requires it to be an equivalence relation
  - Should also be consistent with compareTo

1. Reflexive
   - x.equals(x) == true

2. Symmetric
   - x.equals(y) <=> y.equals(x)

3. Transitive
   - x.equals(y) && y.equals(z) ==> x.equals(z)

4. Consistent (ie over time)
   - x.equals(y) == x.equals(y) == x.equals(y) ... 

5. Robust to null
   - x.equals(null) == false
Naïve approach

class SmartPerson {
    private String firstName;
    private String lastName;

    public boolean equals (SmartPerson p) {
        return (firstName.equals(p.firstName) &&
                lastName.equals(p.lastName));
    }

}
Many Problems with Naïve Solution

- On the surface, it looks promising
  - Reflexive, symmetric, transitive, consistent

- But (1): Not robust to null
  
  ```java
  if (p1.equals(null)) {... //run-time error
  ```

- But (2): Wrong argument type
  - `equals()` has argument type `Object`
  - This implementation overloads (not overrides) `equals()` in `java.lang.Object`
class SmartPerson {
    private String firstName;
    private String lastName;

    @Override
    public boolean equals (Object o) {
        if (o == null) return false;
        SmartPerson p = (SmartPerson)o;
        return (firstName.equals(p.firstName) &&
                lastName.equals(p.lastName));
    }
}
New Problems

- Narrowing cast may fail
  ```java
  Person p = new SmartPerson();
  IceCreamFlavor i = new SaltyCaramel();
  if (p.equals(i)) {... //run-time error
  ```

- We could keep patching it
  - Add `instanceof` test of run-time type

- It would keep breaking
  - Inheritance complicates the analysis
  - Can an OsuStudent be equal to a CseMajor?

- Bottom line: You can not do both
  1. Have behavioral subtypes, and
  2. Satisfy all the equivalence relation requirements
class SmartPerson {

    @Override
    public boolean equals (Object o) {
        if (o == this) return true;
        if (o == null) return false;
        if (!o.getClass().equals(this.getClass()))
            return false;
        SmartPerson p = (SmartPerson)o;
        return (firstName.equals(p.firstName) &&
                lastName.equals(p.lastName));
    }
}
Complication: Extensions

class OsuStudent extends SmartPerson {
    private BuckID identity;

    @Override
    public boolean equals (Object o) {
        if (o == this) return true;
        if (!super.equals(o)) return false;
        OsuStudent s = (OsuStudent)o;
        return identity.equals(s.identity);
    }
}
Notes on equals()

- Initial comparison (ie \( o == this \))
  - Used only for performance reasons (a “shortcut”)

- Objects must be of exactly the same class
  - Subclass instance never equal to superclass instance
    - So much for “is a”!
    - For CseMajor \( c \), and OsuStudent \( s \)
      - assert(!c.equals(s))
  - Different classes that implement the same interface can never be equal
    - For SlowBigNatural \( b1 \), and FastBigNatural \( b2 \)
      - assert(!b1.equals(b2))

- Two recipes for implementing equals()
  - Version 1 when overriding equals for the first time
  - Version 2 when some parent overrides equals
Overriding hashCode() 

- This method returns a “random” int 
  - Must be consistent (ie repeatable) 
  - Default implementation: memory address 

- Equal objects must have equal hashes 
  - x.equals(y) ==> x.hashCode() == y.hashCode() 

- Must distinct objects have distinct hashes? 
  - Not required for correctness 
  - But helps performance when using collections 

- Rule: If you override equals(), override hashCode() 

- Immutable objects can pre-compute and then cache their hashcode value
Recipe for hashCode()

1. Initialize with a non-zero constant integer
   \[
   \text{int result} = 17; \quad //\text{must be non-zero}
   \]

2. For each field \( f \) that figures into equals:
   a. Compute int hash code \( c \) for \( f \)
      - For primitive \( f \), use \( f \)'s value
      - For reference \( f \), recurse
      - For array \( f \), examine each element
        \[
        c = (f \ ? \ 0 : 1); \quad //\text{for boolean } f
        \]
        \[
        c = (\text{int})f; \quad //\text{for byte/char/short/int } f
        \]
        \[
        c = \text{Float.floatToIntBits}(f); \quad //\text{for long } f
        \]
        \[
        c = f.\text{hashCode}(); \quad //\text{for reference } f
        \]
   b. Combine \( c \) into result through multiplication
      \[
      \text{result} = 37*\text{result} + c; \quad //\text{use an odd prime}
      \]

3. Return result
Basic Example

class SmartPerson {
    private String firstName;
    private String lastName;
    private int age;

    @Override
    public int hashCode () {
        int result = 17;
        result = 37*result + firstName.hashCode();
        result = 37*result + lastName.hashCode();
        result = 37*result + age;
        return result;
    }
}

Example: hashCode for Immutable

class SmartPerson {
    private int cachedHashCode = 0;

    @Override
    public int hashCode () {
        if (cachedHashCode == 0) {
            int result = 17;
            // code to compute hash from fields
            cachedHashCode = result;
        }
        return cachedHashCode;
    }
}
Supplemental Reading

- Bloch’s “Effective Java”, chapter 3
  - See Safari Books Online link
  - Warning: favors `instanceof` over `getClass()`
    - Better for behavioral subtyping
    - Worse for creating an equivalence relation

- IBM developerWorks paper
  - “Java Theory and practice: Hashing it out”

- Various blogs (all slightly broken)
Summary

- java.lang.Object
  - Root of all class hierarchies
  - Contains useful methods
  - Several core ones should be overridden
- toString()
  - Concise, complete, informative
- equals()
  - Spec: An equivalence relation
  - Default implementation compares references
  - Comparing values is subtle because of inheritance
  - Overriding helps with JUnit
- hashCode()
  - Equal objects must return equal hashes