Equality cont. and Generics

Saman Amarasinghe
Spring 2013
Today’s lecture

Finishing up on Equality

- Beneficent side effects
- Hash code
- Mutable objects

Generics
Beneficent Side Effects

Immutable objects can change internal state without losing their immutability

- As long as the state change has no observable effect on the object’s abstract value
- A change like this is called a **beneficent side-effect**
- Often done for performance reasons

Examples

- Caching results: e.g. Expr.eval() might store the result of the evaluation in the root of the tree, so that it doesn’t have to do it again
- Rebalancing: e.g., a splay tree is a binary search tree that moves the most recently accessed element to the top of the tree

State changes caused by beneficent side effects shouldn’t affect equality

© Robert Miller 2007
equals and hashCode Contract

equals() and hashCode() have a specific relationship

\[
a.equals(b) \implies a.hashCode() == b.hashCode()
\]

- Note that the converse is not necessarily true – a and b can have the same hash code even if they are different objects

This relationship is required by Object’s specification

- Without it, hash sets and hash maps don’t work correctly
Enforcing the Object Contract

Object’s default hashCode() implementation is consistent with its default equals()

```java
public class Object {
    public boolean equals (Object that) { return this == that; }
    public int hashCode () { return /* the address of this */; }
}
```

➢ For references a, b, if a == b, then the address of a == the address of b

But immutable objects need a different hashCode()

Duration d1 = new Duration(1,2);
Duration d2 = new Duration(1,2);
d1.equals(d2)
d1.hashCode()
d2.hashCode()
Override hashCode()

Always override hashCode() when you override equals()

➤ Your hash code should be computed from the same parts of the abstract value that equals() compares

```java
public class Duration {

    ...

    @Override
    public boolean equals(Object _that) {
        if (_that == null || !this.getClass().equals(_that.getClass())) return false;
        Duration that = (Duration) _that;
        return this.getSecs() == that.getSecs();
    }

    @Override
    public int hashCode() {
        return this.getSecs();
    }
}

➤ See also Bloch, Effective Java, item 8

© Robert Miller 2007
Equality of Mutable Objects

Recall our definition

- Two objects are equal when they cannot be distinguished by observation

With mutable objects, there are two ways to interpret this

- ... when they cannot be distinguished by observation that doesn’t change the current state of the program
  - i.e., by calling only observer, producer, and creator methods
  - This is often called observational equality, since it tests whether the two objects “look” the same, in the current state of the program
- ...when they cannot be distinguished by any observation, even state changes
  - i.e., allow calling any methods, including mutators
  - This is often called behavioral equality, since it tests whether the two objects will “behave” the same, in this and all future states
- For immutable objects, observational and behavioral equality are identical

© Robert Miller 2007
Consistency over Time

**Consistency property**

- Equality shouldn’t change over time
- If a.equals(b) now, then a.equals(b) later too

**Here’s why**

List<String> list = new ArrayList<String>();
List.add(“hello”);
Set<List<String>> set = new HashSet<List<String>>();
set.add(list);

set.contains(list) → true
set.contains(Arrays.asList(“hello”)) → true

list.add(“goodbye”);
set.contains(list) → false!
set.contains(Arrays.asList(“hello”, “goodbye”)) → false!
for (List<String> l : set) { set.contains(l) → false! }
Breaking the Rep

What’s going on?

- List<String> is a mutable object
- In the standard Java implementation of collection classes like List, mutations affect the result of equals() and hashCode()
  • i.e., equals() and hashCode() implement observational equality
- When the list is first put into the HashSet, it is stored in the hash bucket corresponding to its hashCode() result at that time
- When the list is subsequently mutated, its hashCode() changes, but HashSet doesn’t realize it should be moved to a different bucket
- So it can never be found again
True Confessions

Quote from the specification of java.util.Set

- Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.

Java library is inconsistent about equals()

- Collections use observational equality
- Other classes (like StringBuffer) use behavioral equality

Lesson: equals() should implement behavioral equality

- Mutable objects should just inherit equals() and hashCode() from Object
- For observational equality (whether two objects “look” the same in the current state), define a new method, e.g. similar()
The Full Object Contract

- equals must define an equivalence relation (reflexive, symmetric, transitive)
- equals must be consistent with itself (repeated calls must yield the same result unless the objects are mutated)
  - the Object contract permits equals() to implement observational equality, but that doesn’t mean you should do it
- x.equals(null) should return false (assuming x is non-null)
- equals and hashCode must be consistent with each other (a.equals(b) implies a.hashCode() == b.hashCode())
Summary

**Equivalence relations**
- equals() should be reflexive, symmetric, and transitive

**Abstraction function as a basis for equality**
- Two objects with the same abstract value should be equals()

**Object contract**
- a.equals(b) implies a.hashCode() == b.hashCode()

**Observational equality vs. behavioral equality**
- If clients can’t distinguish two objects by method calls, they’re equal

**Consistency over time**
- Using behavioral equality avoids messy issues with putting objects in collections
GENERICS
List of different things

How do you represent a list of integers
  public class ListIntegers { ... }

How do you represent a list of Strings
  public class ListStrings { ... }

Again...replication of code!

➤ We would like to have a single list class
public interface List {
    void add(Object x);
    Object get();
    Iterator iterator();
}

public interface Iterator{
    Object next();
    boolean hasNext();
}

List ls = new ArrayList(); // List should contain strings
ls.add("Hello");
ls.add(new Integer(5));
String hello = (String) ls.get(0);
Why Generics

For type checking purposes, you want to have

- ListInteger, ListString, ListFoo, …

But you can’t possibly maintain a separate list for each type

Prehistoric Java settled for a single list of objects

- After all, everything is an object
- This isn’t ideal

List ls = new ArrayList(); // List should contain strings
ls.add("Hello");
ls.add(new Integer(5));
String hello = (String) ls.get(0);
Generics = parameterized types

```java
public interface List<E> {
    void add(E x);
    E get();
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Only one interface/class to maintain
Can be instantiated with an arbitrary type
- List<Integer>, List<String>, List<Foo>
Benefits of generics

List<String> ls = new ArrayList<String>();
ls.add("Hello");
ls.add(new Integer(5)); // This is a compile time error
String hello = ls.get(0);

Comments are replaced by statically checked type parameters
Compiler enforces type safety
No need for messy typecasts
GENERICS AND INHERITANCE
Inheritance

What does it mean for A to be a subtype of B?

- A can be used anywhere in place of B

```java
class Vehicle { … }
class Car extends Vehicle { … }
class Spaceship extends Vehicle { … }

void rideVehicles(Vehicle v);
```
Subtyping

Trick Question:

Should List<Car> be a subtype of List<Vehicle>?

```java
class Vehicle { … }
class Car extends Vehicle { … }
class Spaceship extends Vehicle { … }

void rideVehicles(List<Vehicle> lv);
```

See L14-Transport
Subtyping

Just because Car is a subtype of Vehicle doesn’t mean List<Car> should be a subtype of List<Vehicle>

- This helps maintain the static guarantees of generic classes
- It can also be a pain when trying to write reusable code

```java
class Car extends Vehicle{ … }
class Spaceship extends Vehicle{ … }

void rideVehicles(List<Vehicle> lv);

List<Spaceship> myfriends = new ArrayList<Spaceship>();

rideVehicles(yourfriends);
rideVehicles(myfriends);
```
Wildcards

The symbol ? in the type parameter is a wildcard

- `List<?>` is a supertype of `List<X>` for any `X`.
- `List<? extends Y>` is a supertype of `List<X>` if `Y` is a supertype of `X`.

```java
class Car extends Vehicle{ … }
class Spaceship extends Vehicle{ … }

void rideVehicles(List<? extends Vehicle> lv);

List<Spaceship> myfriends = new ArrayList<Spaceship>();

rideVehicles(yourfriends);
rideVehicles(myfriends);
```
**Wildcards**

**What should the type parameter for the list below be?**

```java
static void addCars(List< > lv){
    lv.add(new Car());
    lv.add(new Car());
}

List<Vehicle> lveh = ... ;
List<Car> lcar = ... ;

addCars(lveh);
addCars(lcar);
```

See L14-Transport-addcars
Wildcards

Express the fact that any superclass of cars will be acceptable

```
static void addCars(List<? super Cars> lv){
    lv.add(new Car());
    lv.add(new Car());
}

List<Vehicle> lvveh = … ;
List<Car> lcar = … ;

addCars(lvveh);
addCars(lcar);
```
Wildcards

General rule:

- Use Container<? extends A> when you want to read from the container

- Use Container<? super A> when you want to write to the container
Generic Methods

Problem:
➢ Write a method that copies elements from one list to another

```java
static void copyList(List<?> out, List<?> in){
    ...
}
```

Will this work?
Generic Methods

Problem:
➢ Write a method that copies elements from one list to another

```java
static void copyList(List<?> out, List<?> in){
    for( _____ e : in){
        out.add(e);
    }
}
```

Will this work?
Generic Methods

Generic methods allow you to establish relationships between type parameters

```java
static <T> void copyList(List<T> out, List<T> in){
    for(T e : in){
        out.add(e);
    }
}
```
Can you copyList a List<subtype> to List<supertype>

```java
static <T> void copyList(List<?> out, List<?> in){
    for(T e : in){
        out.add(e);
    }
}
```

© Robert Miller 2009
Generic Methods

Can you copyList a List<subtype> to List<supertype>?

```java
static <T> void copyList(List<? super T> out, List<? extends T> in){
    for(T e : in){
        out.add(e);
    }
}
```
Conclusions

- Generics are a powerful mechanism to make your code more reusable
- You should be aware of subtle subtyping rules
- Writing good generic code can be tricky