Inheritance and Equality

Saman Amarasinghe
Spring 2014
Functions on Recursive Data Types

append: ImList x ImList \rightarrow ImList

- append(Empty, list2:ImList) = list2
- append(Cons(first:E, rest:ImList), list2:ImList) = cons(first, append(rest, list2))

reverse: ImList \rightarrow ImList

- reverse(Empty) = empty()
- reverse(Cons(first:E, rest:ImList)) = append(reverse(rest), cons(first, empty()))
Functions on Recursive Data Types

reverse: ImList $\rightarrow$ ImList

- reverse(Empty) = empty()
- reverse(Cons(first:E, rest:ImList)) = append(reverse(rest), cons(first, empty()))
**Functions on Recursive Data Types**

**reverse: ImList → ImList**

- \( \text{reverse}(\text{Empty}) = \text{empty}() \)
- \( \text{reverse}(\text{Cons}(\text{first}:E, \text{rest}:\text{ImList})) = \text{append}(\text{reverse}(\text{rest}), \text{cons}(\text{first}, \text{empty}())) \)
Functions on Recursive Data Types

reverse: ImList → ImList

- reverse(Empty) = empty()
- reverse(Cons(first:E, rest:ImList)) = append(reverse(rest), cons(first, empty()))
Functions on Recursive Data Types

**reverse**: ImList → ImList

- reverse(Empty) = empty()
- reverse(Cons(first:E, rest:ImList)) = append(reverse(rest), cons(first, empty()))
Functions on Recursive Data Types

**reverse: ImList \rightarrow ImList**

- reverse(Empty) = empty()
- reverse(Cons(first:E, rest:ImList)) = append(reverse(rest), cons(first, empty()))
Functions on Recursive Data Types

reverse: ImList → ImList

- reverse(Empty) = empty()
- reverse(Cons(first:E, rest:ImList)) = append(reverse(rest), cons(first, empty()))
Functions on Recursive Data Types

reverse: ImList → ImList

- reverse(Empty) = empty()
- reverse(Cons(first:E, rest:ImList)) = append(reverse(rest), cons(first, empty()))
Functions on Recursive Data Types

reverse: ImList \rightarrow ImList

- \text{reverse}(\text{Empty}) = \text{empty}()
- \text{reverse}(\text{Cons}(\text{first}:E, \text{rest}:\text{ImList})) = \text{append}(\text{reverse}(\text{rest}), \text{cons}(\text{first}, \text{empty}()))
Functions on Recursive Data Types

**reverse: ImList → ImList**

- reverse(Empty) = empty()
- reverse(Cons(first:E, rest:ImList)) = append(reverse(rest), cons(first, empty()))
Functions on Recursive Data Types

reverse: ImList → ImList

- reverse(Empty) = empty()
- reverse(Cons(first:E, rest:ImList)) = append(reverse(rest), cons(first, empty()))
Functions on Recursive Data Types

reverse: ImList → ImList

- reverse(Empty) = empty()
- reverse(Cons(first:E, rest:ImList)) = append(reverse(rest), cons(first, empty()))
Functions on Recursive Data Types

**reverse: ImList → ImList**

- reverse(Empty) = empty()
- reverse(Cons(first:E, rest:ImList)) = append(reverse(rest), cons(first, empty()))
Today’s lecture

Last Lectures were talked about Software Engineering, Today we will look at the impact of programming languages

➢ Type casting
➢ Power of inheritance
➢ Dynamic dispatch and method overloading
➢ Equality
➢ Subtleties of inheritance in equality
➢ Composition vs. Inheritance
➢ Impact of mutability on Equality

The moral of today’s lecture

➢ Languages provide powerful tools to engineer software
➢ When well understood and used properly, can be very powerful
➢ But a lot of subtle details that can really create huge problems

© Robert Miller 2007
Upcasting and Downcasting

**Upcasting**: an object of a derived class is assigned to a variable of a base class

- If you assign a superclass object to a subclass variable, Java will do this for you!

**Downcasting**: an object in a base class variable is assigned to a derived class variable

- Need to do this explicitly by the programmer
- Dangerous! May result in a runtime exception if the type is wrong

```java
public class DemoSuper { }
Public class DemoSub extends DemoSuper { }

DemoSuper supersub = new DemoSub();
DemoSub subsub = new DemoSub();
DemoSuper supersuper = new DemoSuper();
DemoSuper dsuper = subsub;
DemoSub dsub1 = (DemoSub)supersub;
DemoSub dsub2 = (DemoSub)supersuper;
```

See L12-Demo-1

This brings us to inheritance, why do we need inheritance?
Example

**Polygon:**
- int centerX()
- int centerY()
- int area()
- int circumference()
- int numSides()
- int sideLength()

**Ellipse:**
- int centerX()
- int centerY()
- int area()
- int circumference()
- int lengthX()
- int lengthY()

Both Polygon and Ellipse has parts of identical code

How can we share this code?
Inheritance

Polygon:
- int centerX()
- int centerY()
- int area()
- int circumference()
- int numSides()
- int sideLength()

Shape:

Ellipse:
- int centerX()
- int centerY()
- int area()
- int circumference()
- int lengthX()
- int lengthY()
Inheritance

Shape:

Polygon:
- int centerX()
- int centerY()
- int area()
- int circumference()
- int numSides()
- int sideLength()

Ellipse:
- int centerX()
- int centerY()
- int area()
- int circumference()
- int lengthX()
- int lengthY()
Inheritance

```java
public class Shape {
    public int centerX()
    public int centerY()
    public int area()
    public int circumference()
}

public class Polygon class Shape {
    public int numSides()
    public int sideLength()
}

public class Ellipse class Shape {
    public int lengthX()
    public int lengthY()
}
```
Power of Inheritance I

Can share common code

Can extend functionality without cluttering the original
But, Some computation are specialized

**Shape:**
- int centerX()
- int centerY()
- int area()
- int circumference()

**Polygon:**
- int numSides()
- int sideLength()

**Ellipse:**
- int lengthX()
- int lengthY()
Calculating the area of all the shapes

```java
import java.util.*;

public class Main {
    List<Shape> shapes = new ArrayList<Shape>();

    public int totalArea() {
        int area = 0;
        for(Shape myshape : shapes) {
            if(myshape instanceof Polygon) {
                Polygon polyshape = (Polygon)myshape;
                area += polyshape.area();
            } else if(myshape instanceof Ellipse) {
                Ellipse ellshape = (Ellipse)myshape;
                area += ellshape.area();
            }
        }
        return area;
    }
}
```

© Robert Miller 2007
Power in Inheritance II: Polymorphism

Can override a method in the superclass in the subclass

- Have an area method in Shape as well as Polygon and Ellipse

When invoking the method, one in the Object’s class, NOT in the class of the type of the variable pointing to the Object, will be called

- Dynamic Dispatch
- Invoking area() for a will call the area in either Polygon or Ellipse depending on the type of the object

Can do computation on the parent objects

- Don’t need to know how many different subtypes are there
- Adding a new subtype does not need to mess with the parent
public class Shape {
    public int area()
    public int circumference()
}

public class Polygon class Shape {

    @Override
    public int area() { … }

    @Override
    public int circumference() { … }
}
public class Main {
    List<Shape> shapes = new ArrayList<Shape>();

    public int totalArea() {
        int area = 0;
        for(Shape myshape : shapes) {
            area += myshape.area();
        }
        return area;
    }
}
public class DemoSuper {
    public void mytest() {
        System.out.println("in DemoSuper");
    }
}

public class DemoSub extends DemoSuper {
    public void mytest() {
        System.out.println("in DemoSub");
    }
}

DemoSuper supersuper = new DemoSuper();
DemoSuper supersub = new DemoSub();
DemoSub subsub = new DemoSub();

supersuper.mytest(); → ??
supersub.mytest(); → ??
subsub.mytest(); → ??
OK, what happens now…

```
public class DemoSuper {

    static void mytest2(DemoSuper ds) {
        System.out.println("arg DemoSuper");
    }

    static void mytest2(DemoSub ds) {
        System.out.println("arg DemoSub");
    }

    DemoSuper supersuper = new DemoSuper();
    DemoSuper supersub   = new DemoSub();
    DemoSub   subsub     = new DemoSub();

    mytest2(supersuper);   → ??
    mytest2(supersub);     → ??
    mytest2(subsub);       → ??
```
Overloading Methods

Can have multiple methods with the same name within a class

At compile time, the type signature of the method invocation is matched to select the method to call

- Will upcast the arguments as needed to find a matching type signature

Different from dynamic dispatch we just observed
Equality

In the physical world

- Every object is distinct -- at some level, even two snowflakes are different
  - Even if the distinction is just the position they occupy in space
- Not true of all subatomic particles, actually, but true enough of large objects like snowflakes and baseballs and people
- So two physical objects are never truly “equal” to each other, only degrees of similarity

In the world of human (or mathematical) language

- You can have multiple names for the same object
- So it’s natural to ask when two expressions represent the same object
  - 1+2
  - 3
  - \( \sqrt{9} \)
  - \((\lambda x \mid x+1) \ 2\)
Two Ways to Regard Equality

As a relation

- An equivalence is a relation \( R \subseteq T \times T \) that is:
  - \textbf{reflexive}: \( R(t,t) \) for all \( t \in T \)
  - \textbf{symmetric}: \( R(t,u) \Rightarrow R(u,t) \)
  - \textbf{transitive}: \( R(t,u) \land R(u,v) \Rightarrow R(t,v) \)

- To use \( R \) as a definition for equality: \( a \) equals \( b \) if and only if \( R(a,b) \)

As an interpretation function

- An interpretation function \( f: E \rightarrow V \) maps expressions to abstract values
- To use \( f \) as a definition for equality of expressions: \( a \) equals \( b \) if and only if \( f(a)=f(b) \)

These notions are equivalent

- an equivalence relation induces an interpretation function (the relation partitions \( T \), so \( f \) maps each element to its partition class)
- the relation induced by an interpretation function is an equivalence relation (check the three properties)
Two Views of Equality

Two possible definitions

- Two expressions are equal when they denote the same abstract value
- Two expressions are equal when they cannot be distinguished by observation

Example

- consider the set expressions \{1,2\} and \{2,1\}
- the unique abstract set they both denote contains exactly 1 and 2
- under the observers |...| and \(\in\), they’re indistinguishable
  - \(|\{1,2\}| = 2\) and \(|\{2,1\}| = 2\)
  - 1 \(\in\) \{1,2\} is true, and 1 \(\in\) \{2,1\} is true
  - 2 \(\in\) \{1,2\} is true, and 2 \(\in\) \{2,1\} is true
  - 3 \(\in\) \{1,2\} is false, and 3 \(\in\) \{2,1\} is false
- So both approaches produce the same results in the world of mathematical expressions
Equality in Immutable ADTs

Mapping these viewpoints to abstract data types

- Interpretation function is the abstraction function
  - i.e., two rep objects r1 and r2 are equal iff the abstraction function A maps them to the same abstract value, i.e. \( A(r1) = A(r2) \)

- “Indistinguishable by observation” means calling methods
  - i.e., two immutable objects are equal iff they cannot be distinguished by calling methods on the objects
public class Duration {
    // abstract duration is an integer representing
    // total number of seconds
    // A(r) = r.getSecs()
    private final int mins;
    private final int secs;
    public Duration(int m, int s) {
        mins = m;
        secs = s;
    }
    public int getSecs() { return mins*60+ secs; }
}

Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 3);
Duration d3 = new Duration (0, 62);

Which of these should be considered equal?
== vs. equals()

In Java

- == compares object locations
  - Or, more precisely, it tests *referential* equality: two references are == if they point to the same storage in memory
- equals() compares object contents
  - In other words, *object* equality

- Our first try at Duration.equals():

```java
public class Duration {
    ...
    public boolean equals(Duration that) {
        return this.getSecs() == that.getSecs();
    }
}
```

© Robert Miller 2007
What happens when...

```java
public class Duration {
    ...
    public boolean equals(Duration that) {
        return this.getSecs() == that.getSecs();
    }
}
```

Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 3);
Duration d3 = new Duration (0, 62);

d1.equals(d1) → ??
d1.equals(d2) → ??
d2.equals(d1) → ??
d1.equals(d3) → ??
d3.equals(d1) → ??
d2.equals(d3) → ??
d3.equals(d1) → ??
# Referential & Object Equality in Languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Referential Equality</th>
<th>Object Equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td><code>==</code></td>
<td><code>Equals()</code></td>
</tr>
<tr>
<td>Objective C</td>
<td><code>==</code></td>
<td><code>isEqual:</code></td>
</tr>
<tr>
<td>C#</td>
<td><code>==</code></td>
<td><code>Equals()</code></td>
</tr>
<tr>
<td>Python</td>
<td><code>is</code></td>
<td><code>==</code></td>
</tr>
<tr>
<td>Javascript</td>
<td><code>==</code></td>
<td>n/a</td>
</tr>
</tbody>
</table>
equals() must be an equivalence

Implementation of equals() must satisfy the 3 properties

- Reflexive: a.equals(a) for all non-null references a
- Symmetric: a.equals(b) \( \Rightarrow \) b.equals(a)
- Transitive: a.equals(b) \( \land \) b.equals(c) \( \Rightarrow \) a.equals(c)

Suprisingly easy to get this wrong

- What if b is null? Better return false if we want symmetry
- What if we wanted a tolerance in comparing Durations:

```java
public class Duration {
    private static final int CLOCK_SKEW = 5; // seconds
    public boolean equals(Duration that) {
        return Math.abs(this.getSeconds() - that.getSeconds()) <= CLOCK_SKEW;
    }
}
```

• Which property is violated?
What happens when…

```java
public class Duration {

    ......

    public boolean equals (Duration that) {
        return this.getSecs() == that.getSecs();
    }
}
```

Duration d = new Duration (1, 2);
Object o = new Duration(1,2);

```
d.equals(o) → ??
o.equals(d) → ??
```
Overriding vs. Overloading

What’s going on with d.equals(o)?

➢ Duration has overloaded the equals() method inherited from Object, instead of overriding it

```java
public class Object {
    public boolean equals (Object that) {
        return this == that;
    }
}

public class Duration extends Object {
    public boolean equals (Object o) {
        return o == this;
    }
    public boolean equals (Duration that) {
        return this.getSecs() == that.getSecs();
    }
}
```

© Robert Miller 2007
Overriding vs. Overloading

What’s going on with o.equals(d)?

- `equals()` in object is not overridden by `Duration`. Only one `equals()` in `Object`.
- Since `d` is also an object, it can match the `equals(Object)` method in `Object`.

```java
public class Object {
    public boolean equals (Object that) {
        return this == that;
    }
}

public class Duration extends Object {
    public boolean equals (Duration that) {
        return this.getSecs() == that.getSecs();
    }
}
```
Fixing Equals()

```java
@Override // compile-time error if doesn’t match a superclass method
public boolean equals (Object _that) {
    // also handles _that == null
    if (! (_that instanceof Duration)) return false;
    Duration that = (Duration) _that;
    return this.getSecs() == that.getSecs();
}
```

This fixes the immediate problem:

Duration d = new Duration(1, 2);
Object o = new Duration(1, 2);

d.equals(o) → ??
o.equals(d) → ??
Suppose we subclass Duration

```java
public class PreciseDuration extends Duration {
    private final int millisecs;
    public PreciseDuration(int m, int s, int ms) {
        super(m,s);
        millisecs = ms;
    }
    public int getMillisecs() {
        return super.getSecs() * 1000 + millisecs;
    }
}
```

How should equality be defined for PreciseDuration?

- Can we simply use the equals() inherited from Duration? No, because it ignores milliseconds.
Overriding equals

```java
public class PreciseDuration extends Duration {
    ...

    @Override
    public boolean equals (Object _that) {
        if (! (_that instanceof PreciseDuration))
            return false;
        PreciseDuration that = (PreciseDuration) _that;
        return this.getMillisecs() == that.getMillisecs();
    }
}
```

Duration d1 = new Duration(1, 2);
Duration d2 = new Duration(1, 2);
PreciseDuration pd1 = new PreciseDuration(1, 2, 100);
PreciseDuration pd2 = new PreciseDuration(1, 2, 100);

d1.equals(pd1) → ??
pd1.equals(d1) → ??
Overriding equals

Duration d1 = new Duration(1, 2);
Duration pd1 = new PreciseDuration(1, 2, 100);

pd1.equals(d1) → false
d1.equals(pd1) → true

Two calls use different methods
➢ First uses the equal method in PreciseDuration
➢ Second uses the equal method in Duration as pd1 is also a Duration
Another Try

Suppose we use the superclass definition of equals() in all cases except for comparing two PreciseDuration objects

```java
public class PreciseDuration extends Duration {
    ...
    @Override
    public boolean equals(Object _that) {
        if (!(_that instanceof PreciseDuration)) return super.equals(_that);
        PreciseDuration that = (PreciseDuration) _that;
        return this.getMillisecs() == that.getMillisecs();
    }
}
```
public class PreciseDuration extends Duration {

    ... 

    @Override 
    public boolean equals (Object _that) {
        if (! (_that instanceof PreciseDuration)) return super.equals(_that);
        PreciseDuration that = (PreciseDuration) _that;
        return this.getMillisecs() == that.getMillisecs();
    }

}

Duration d = new Duration(1, 2);
PreciseDuration pd1 = new PreciseDuration(1, 2, 100);
PreciseDuration pd2 = new PreciseDuration(1, 2, 200);

d.equals(pd1) → ??
d.equals(pd2) → ??
pd1.equals(pd2) → ??
public class PreciseDuration extends Duration {

    ...

    @Override
    public boolean equals (Object _that) {
        if (! (_that instanceof PreciseDuration)) return super.equals(_that);
        PreciseDuration that = (PreciseDuration) _that;
        return this.getMillisecs() == that.getMillisecs();
    }

    }

➤ Not transitive!
    • Allows two different PreciseDuration objects to be equal to the same Duration object
Solving the Subclassing Problem

No really satisfactory solution

- Standard approach: superclass equality should reject all subclass objects
  - i.e., instead of
    ```java
    if (! (_that instanceof Duration)) return false;
    ```
  - use:
    ```java
    if (!_that.getClass().equals(getClass())) return false;
    ```
- but this is inflexible
  - e.g., doesn’t permit a subclass that doesn’t add any new abstract values

Better solution

- avoid inheritance, and use composition instead
- see Bloch, Effective Java, Item 14
Use a member than subclass

```java
public class PreciseDuration {
    private final Duration duration;
    private final int millisecs;
    public PreciseDuration(int m, int s, int ms) {
        duration = new Duration(m, s);
        millisecs = ms;
    }

    public int getMillisecs() {
        return duration.getSecs()*1000 + millisecs;
    }

    @Override
    public boolean equals (Object _that) {
        // also handles _that == null
        if (! (_that instanceof PreciseDuration)) return false;
        PreciseDuration that = (PreciseDuration) _that;
        return this.getMillisecs() == that.getMillisecs();
    }
}
```
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 beneficial 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
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.

© Robert Miller 2007
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) → true
d1.hashCode() → 2392
d2.hashCode() → 4823
```
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
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 = makeList("hello");
Set<List<String>> set = new HashSet<List<String>>();
set.add(list);
set.contains(list) → true
set.contains(makeList("hello")) → true

list.add("goodbye");
set.contains(list) → false!
set.contains(makeList("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