how to implement an abstract type

Daniel Jackson
March 19, 2008
plan for today

topics
• steps to implementing a type
• equality and how to code it
• rep invariants & how to exploit them
• equality subtleties (to read after)

reminders
• lecture note handout
• P2a due tomorrow
• work breakdown due too
steps to implementing a type
step 1: design a rep

desiderata

• easy to program (and get right!)
• good enough performance

usually

• a couple of fields of existing types suffices
• before inventing a complex type, check the Java collections

sometimes

• a tricky structure or algorithm is needed
• first, see if someone’s done it before: copy from CLR, eg

always

• write a rep invariant to clarify the design
step 2: write the methods

required methods first
• from Object class: equals, hashCode, to String
• from any interface the class implements
• when overriding, mark with @Override

constructors
• for an immutable type, some private constructors often help

producers and observers
• whenever possible, build on each other
• separate core methods (eg, size) from those that sit on top (eg. isEmpty)

incomplete methods
• use UnsupportedOperationException to get it to compile
step 2: rep invariant

code the rep invariant

• as a `checkRep` method

• for immutables, call it at the end of all constructors

as you write the constructors and producers

• ask yourself **why** they preserve the rep invariant
step 4: assertions and tests

runtime assertions

• are your friend: use them freely
• turn on by adding -ea to VM args in Eclipse

write JUnit test suite for your class

• will help you find bugs earlier, and make debugging easier
• take the trouble to write a toString that produces helpful output
equality: basics
fundamentals

objects often used as keys

- need to compare them
- eg, `Literal` used as key in `Environment`

Java convention

- the class `Object` has a method that every class inherits
  
  
  `Object.equals: Object -> boolean`

- by convention, this method is used to compare objects for equality
- collections especially assume this: call `equals` on keys
- the inherited method is usually wrong for immutable types
- so must override by explicitly declaring a method
  
  `MyType.equals: Object -> boolean`
why inherited equality fails

the problem

⦁ `Object.equals` compares objects with `==`
⦁ this makes any two distinct objects unequal
⦁ even if they have the same value

example

⦁ the “same” pairs are unequal:

```java
public class Pair {
  private int fst, snd;
  public Pair (int f, int s) {fst=f; snd=s;}

  public static void main (String[] args) {
    Pair p1 = new Pair (1, 2);
    Pair p2 = new Pair (1, 2);
    System.out.println (p1 == p2 ? "yes" : "no");
    System.out.println (p1.equals(p2) ? "yes" : "no");
  }
}
```
standard equals method

correct code for Pair.equals

\cdot compare the fields

```java
@override
public boolean equals (Object that) {
    if (this == that) return true;
    if (!(that instanceof Pair)) return false;
    Pair p = (Pair) that;
    return p.fst == fst && p.snd == snd;
}
```

remember: comparison is with any object reference

\cdot need to check type of arg, and whether null

\cdot you may be tempted to write this, but don’t (see later slide)

  ```java
  public boolean equals (Pair that) {...}
  ```

\cdot write @Override and compiler will catch the bug
a design puzzle

interning objects

• suppose you have a structure containing objects of type C
• you want to intern them: that is, have one object for each value
• so you write this code, but it won’t work (why not?)

```java
public class C {
    private String s;
    public static Map<C, C> allocated = new ListMap<C, C>();

    public C intern() {
        C c = allocated.get(this);
        if (c == null) {
            allocated = allocated.put(this, this);
            return this;
        }
        return c;
    }
}
```
approaches

the problem: one equals method

- if it compares references with ==, then lookup won’t find match
- if it compares values, then interning is pointless!

have collection take equality predicate as argument

- see “higher-order functions” slides (lecture 10)
- can’t use standard Java collections: will have to make your own
- but see use of comparator objects in ordered types like java.util.TreeSet

use component as key instead of whole object

- eg, allocated maps String to C
- this is how the factory method of PosLiteral works (lecture 10)

for key, make wrapper around C object with its own equals

- not terrible, but a bit ugly
rep invariants
rep invariant R
  \* defines set of legal representation values

abstraction function A
  \* interprets legal representation values as abstract values
how to establish invariants

for state machines
- establish invariant in initial state
- ensure that all transitions preserve invariant

for mutable types, the same approach
- a mutable object is a state machine

for immutable types, a simpler story
- objects can’t change
- assume any argument you’re given satisfies the invariant
- ensure any result you create satisfies it too

who gets to preserve the invariant?
- by hiding the rep, can limit to the methods of the ADT itself
implications

a strong invariant means

• methods can assume more about arguments
• allows checks to be omitted and optimizations to be applied
• but methods must do more to ensure results satisfy invariant

rep design = rep invariant

• the choice of rep invariant characterizes the design of the rep!
common invariants

these invariants

• are commonly used
• provide concrete benefits

examples

• **no nulls**: no need to check before calling method
• **acyclic**: no need to worry about looping
• **ordered**: can navigate efficiently; can stop when key value is passed
• **no duplicates**: can stop when find first match
• **caching**: can do fast look up
example: Clause
writing the invariant

rep invariant for **Clause** written as executable method

```java
public class Clause {
    private final List<Literal> literals;
    static final boolean CHECKREP = true;
    void checkRep () {checkRep (literals);} 
    void checkRep (List<Literal> ls) {
        assert ls != null : "Clause, invariant: literals non-null";
        if (!ls.isEmpty()) {
            Literal first = ls.first(); List<Literal> rest = ls.rest();
            assert first != null : "Clause, invariant: no null elements";
            assert !rest.contains(first) : "Clause, invariant: no duplicates";
            assert !rest.contains(first.getNegation()) : "Clause, invariant: no literal and its negation";
            checkRep (rest);
        }
    }
    private Clause(List<Literal> literals) {
        this.literals = literals;
        if (CHECKREP) checkRep();
    }
    ...
}
```

flag to turn expensive check off

messages give invariant informally

check rep for each constructed value
exploiting the invariant

an equals method for Clause

```java
@override
public boolean equals (Object that) {
    if (this == that) return true;
    if (!(that instanceof Clause)) return false;
    Clause c = (Clause) that;
    if (size() != c.size()) return false;
    for (Literal l: literals)
        if (!(c.contains(l))) return false;
    return true;
}
```

how invariant is exploited

• since literals is non-null, can use in for-loop without null check
  implicit call to literals.iterator will throw exception if literals is null

• since no duplicate literals, can check containment in one direction only

  that is, given two sets \( S \) and \( T \): \( S = T \iff \#S = \#T \land S \subseteq T \)
paying the price

no free lunch

• you have to do some work to establish the invariant

```java
public Clause add(Literal l) {
    if (literals.contains(l)) return this;
    if (literals.contains(l.getNegation())) return null;
    return new Clause(literals.add(l));
}
```

• without invariant, first two statements wouldn’t be needed
exercise

how does reduce exploit the invariant?

/**
 * Requires: literal is non-null
 * @return clause obtained by setting literal to true
 * or null if the entire clause becomes true
 */
public Clause reduce(Literal literal) {
    List<Literal> reducedLiterals = reduce(literals, literal);
    if (reducedLiterals == null) return null;
    else return new Clause(reducedLiterals);
}
private static List<Literal> reduce(List<Literal> literals, Literal l) {
    if (literals.isEmpty()) return literals;
    Literal first = literals.first();
    List<Literal> rest = literals.rest();
    if (first.equals(l)) return null;
    else if (first.equals(l.getNegation())) return rest;
    else {
        List<Literal> restR = reduce(rest, l);
        if (restR == null) return null;
        return restR.add(first);
    }
}
more on equality
properties of equality

can define your own equality notion
  \- but is any spec reasonable?

reasonable equality predicates
  \- define objects to be equal when they represent the same abstract value

a simple theorem
  \- if we define $a \approx b$ when $f(a) = f(b)$ for some function $f$
  \- then the predicate $\approx$ will be an equivalence

an equivalence relation is one that is
  \- reflexive: $a \approx a$
  \- symmetric: $a \approx b \Rightarrow b \approx a$
  \- transitive: $a \approx b \land b \approx c \Rightarrow a \approx c$
a running example

a duration class

\[\text{represents durations measured in minutes}\]

```java
public class Duration {
    private final int hours;
    private final int mins;
    public Duration(int h, int m) {hours = h; mins = m;}
    public int getMins() {return hours*60 + mins;}
}
```
abstraction function

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

eres our first broken equality method

• violates transitivity: easy to see why

    public class Duration {
        private final int hours;
        private final int mins;
        static final int CLOCK_SKEW = ...;
        public boolean equals (Duration d) { // problematic, see next slide
            if (d == null) return false;
            return Math.abs(d.getMins()-this.getMins()) < CLOCK_SKEW;
        }
    }
bug #2

what happens if you fail to override equals

note that outcome depends on declaration, not runtime type (aagh!)

```java
public class Duration {
    private final int hours;
    private final int mins;
    public Duration(int h, int m) {hours = h; mins = m;}
    public boolean equals(Duration d) {
        return d.getMins() == this.getMins();
    }
}
```

```java
Duration d1 = new Duration(1,2);
Duration d2 = new Duration(1,2);
System.out.println(d1.equals(d2)); // prints true

Object d1 = new Duration(1,2);
Object d2 = new Duration(1,2);
System.out.println(d1.equals(d2)); // prints false!
```
explaining bug #2

what's going on?

- we've failed to override `Object.equals`
- method is chosen using compile-time type
- method has been overloaded, not overridden

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

public class Duration extends Object {
    public boolean equals (Object o) {
        return o == this;
    }
    public boolean equals (Duration d) {
        return d.getMins() == this.getMins();
    }
}
```
here's a fix to the problem

• compile-time declaration no longer affects equality

```java
@Override // compile error if doesn’t override superclass method
public boolean equals(Object o) {
    if (!(o instanceof Duration))
        return false;
    Duration d = (Duration) o;
    return d.getMins() == this.getMins();
}
```
equality and subclassing

now considering extending the type

• how should equality be determined?
• can’t rely on inherited equals method, because seconds ignored

```java
public class ShortDuration extends Duration {
    private final int secs;
    ...
    private ShortDuration (int h, int m, int s) {...};
    public int getSecs () {return 3600*hours + 60*mins + secs;}
    ...
}
```
an attempt at writing equals for subclass

```java
@Override public boolean equals(Object o) {
    if (! (o instanceof ShortDuration))
        return false;
    ShortDuration d = (ShortDuration) o;
    return d.getSecs () == this.getSecs();
}
```

will this work?

' no, now it's not symmetric!

```java
Duration d1 = new ShortDuration(1,2,3);
Duration d2 = new Duration(1,2);
System.out.println(d1.equals(d2)); // false
System.out.println(d2.equals(d1)); // true
```
yet another attempt
  ∙ this time not transitive

```java
@Override public boolean equals(Object o) {
    if (! (o instanceof Duration)) return false;
    if (! (o instanceof ShortDuration)) return super.equals (o);
    ShortDuration d = (ShortDuration) o;
    return d.getSecs () == this.getSecs();
}
```

```
Duration d1 = new ShortDuration(1,2,3);
Duration d2 = new Duration(1,2);
Duration d3 = new ShortDuration(1,2,4);
System.out.println(d1.equals(d2)); // true
System.out.println(d2.equals(d3)); // true
System.out.println(d1.equals(d3)); // false!
```
solving the subclassing snag

no really satisfactory solution

superclass equality rejects subclass objects
  • can write this
    ```java
    if (!o.getClass().equals(getClass())) return false;
    ```
  • but this is inflexible: can’t extend just to add functionality, eg

better solution
  • avoid inheritance, and use composition instead
  • see Bloch, *Effective Java*, Item 14
summary
summary

equality

› essential method for most types
› for immutables, reference equality (==) is no good
› writing the equals method is tricky
› with subclassing, even trickier

rep invariants

› characterize the design of representations
› a strong invariant helps: simpler and more efficient code
› but must ensure that producers preserve the invariant
› implement the invariant as a checkRep method
lecture exercise

for Monday after Spring Break

[fairly easy] **exploiting the rep invariant**

- study the code of the `removeAll` method of `ListSet` in the repository
- how could you make it more efficient by exploiting the rep invariant?
- modify the code, and run the JUnit test suite to make sure it works