Catch up: Equality, Testing and Exceptions
Based on material by Michael Ernst, University of Washington

Announcements
- HW4 is due today
- HW5 based off of HW4, will be out tonight, due Oct. 31th
- Konstantin will cover lecture Tuesday

Last class
- Testing
  - White-box testing
- Equality
  - Reference equality
  - "Value" equality with .equals
  - Equality and inheritance
    - equals and hashCode

Outline
- Equality
  - Equality and mutation
  - Equality in ADTs
- Testing
  - Def-use testing
  - Summary of testing
- Exceptions
  - Exceptions vs. preconditions
  - 2 uses of exceptions: failure and special result

Equality
- Simple idea:
  - 2 objects are equal if they have the same value
- Many subtleties
  - Same rep or same abstract value?
    - Remember the HW3 questions
  - Equality in the presence of inheritance?
  - Does equality hold just now or is it eternal?
  - How can we implement equality efficiently?

The int hashCode Method
- hashCode computes an index for the object (to be used in hashtables)
- Javadoc for Object.hashCode():
  ...
  Must be consistent with equals() method:
  a.equals(b) => a.hashCode() == b.hashCode()
  
RULE: If you override equals() make sure you override hashCode to make it consistent with equals()!!!
Implementations of `hashCode`

```java
public boolean equals(Object o) {
    if (!(o instanceof Duration)) return false;
    Duration d = (Duration) o;
    return this.min == d.min && this.sec == d.sec;
}
public class Duration {
    // Choice 1: don't override, inherit `hashCode` from `Object`
    // Choice 2: `public int hashCode() { return 1; }`
    // Choice 3: `public int hashCode() { return min; }`
    // Choice 4: `public int hashCode() { return min+sec; }`
}
```

`hashCode` Must Be Consistent with `equals`

- Suppose we change `Duration.equals` (returns true if `o` and `this` represent the same number of seconds)
```java
public boolean equals(Object o) {
    if (!(o instanceof Duration)) return false;
    Duration d = (Duration) o;
    return 60*min+sec == 60*d.min+d.sec;
}
```

- Can we leave `hashCode` be one of the implementations on the previous slide that we determined were consistent?

**Equality, Mutation and Time**

- If two objects are equal now, will they always be equal?
  - In mathematics, the answer is "yes"
  - In Java, the answer is "you chose"
  - The `Object` spec does not specify this
- For immutable objects
  - Value never changes, equality is eternal
- For mutable objects
  - We can either compare values now, or
  - be eternal (can't have both since value can change)

StringBuffer Example

- StringBuffer is mutable, and takes the eternal approach
```java
StringBuffer s1 = new StringBuffer("hello");
StringBuffer s2 = new StringBuffer("hello");
System.out.println(s1.equals(s1)); // true
System.out.println(s1.equals(s2)); // false
```
- `equals` is just reference equality (==). This is the only way to ensure eternal equality for StringBuffer
**Date Example**

- Date is mutable, and takes the “compare values now” approach

```java
Date d1 = new Date(0); // Jan 1, 1970 00:00:00 GMT
Date d2 = new Date(0);
System.out.println(d1.equals(d2)); // true
d2.setTime(1); // a millisecond later
System.out.println(d1.equals(d2)); // false
```

**Equality and Mutation**

- We can violate rep invariant of a Set container (rep invariant: there are no duplicates in set) by mutating after insertion

```java
Set<Date> s = new HashSet<Date>();
Date d1 = new Date(0);
Date d2 = new Date(1);
s.add(d1);
s.add(d2);
d2.setTime(1000); // mutation after d2 already in the Set!
for (Date d : s) { System.out.println(d); }
```

- Sets assume hash codes don’t change

```java
Set<Date> s = new HashSet<Date>();
Date d1 = new Date(0);
Date d2 = new Date(1000); // 1 sec later
s.add(d1);
s.add(d2);
d2.setTime(10000);
s.contains(d2); // false
s.contains(new Date(10000)); // false again
```

**Implementing equals Efficiently**

- equals can be expensive!

```java
public boolean equals(Object o) {
    if (this == o) return true;
    // class-specific prefiltering (e.g.,
    // compare file size if working with files)
    // Lastly, compare fields (can be expensive)
}
```

- How can we speed-up equals?

```java
// Assumes int arrays rep and rep inv: c[c.length-1]!=0
public boolean equals(Object o) {
    if (o instanceof Poly) {
        Poly p = (Poly) o;
        int i=0;
        while (i < Math.min(p.c.length,c.length)) {
            if (p.c[i] != c[i]) // Assume int arrays
                return false;
            i = i+1;
        }
        if (i != p.c.length || i != c.length) return false;
        return true;
    }
    return false;
}
```
Example: Better `equals`

```java
public boolean equals(Object o) {
    if (o instanceof Poly) {
        Poly p = (Poly) o;
        if (p.c.length != c.length)
            return false; // prefiltering
        for (int i=0; i < c.length; i++) {
            if (p.c[i] != c[i])
                return false;
        }
        return true;
    }
    else
        return false;
}
```

Implementing `hashCode`

```java
public int hashCode() {
    int h = this.hash; // rep. field hash
    if (h == 0) {       // caches the hashcode
        char[] val = value;
        int len = count;
        for (int i = 0; i < len; i++) {
            h = 31*h + val[i];
        }
        this.hash = h;
    }
    return h;          // This works only for immutable objects!
}
```

Rep Invariant, AF and Equality

- With ADTs we compare abstract values, not rep
- Usually, many valid reps map to the same abstract value
- If Concrete Object (rep) and Concrete Object' (rep') map to the same Abstract Value, then Concrete Object and Concrete Object' must be equal
- A stronger rep invariant shrinks the domain of the AF and simplifies `equals`

Example: Line Segment

```java
class LineSegment {
    // Rep invariant:
    // x1<x2 || x1=x2 && y1<y2
    float x1,y1;
    float x2,y2;
    ...
}
```

Other Examples

- RatNum
- RatPoly

Outline

- Equality
  - Equality and mutation
  - Implementing `equals` and `hashCode` efficiently
  - Equality in ADTs
- Testing
  - Def-use testing
  - Summary of Testing
- Exceptions
  - Exceptions vs. preconditions
  - 2 uses of exceptions: failure and special result
White Box Testing

```java
boolean[] primeTable[CACHE_SIZE]
// returns: true if x is prime,
//          false otherwise
boolean isPrime(int x) {
    if (x > CACHE_SIZE) {
        for (int i=2; i<x/2; i++)
            if (x%i==0) return false;
        return true;
    } else return primeTable[x];
}
```

Fall 17 CSCI 2600, A Milanova (example due to Michael Ernst)

White Box Testing

- Ensure test suite covers (covers means executes) "all of the program"
- We approximate "all of the program" using coverage targets
  - CFG nodes
  - CFG edges
  - CFG paths
- Assumption: higher coverage implies fewer remaining errors in the program

White Box Testing: Dataflow-based Testing

- A definition (def) of x is x at the left-hand-side
  - E.g., x = y+z, x = x+1, x = foo(y)
- A use of x is when x is at the right-hand-side
  - E.g., z = x+y, x = x+y, x> y, z = foo(x)
- A def-use pair of x is a pair of nodes, k and n in the CFG, s.t. k is a def of x, n is a use of x, and there is a path from k to n free of definition of x

A Buggy gcd

```java
// requires a,b > 0
static int gcd(int a, int b) {
    int x=a;
    int y=b;
    while (x != y) {
        if (x > y) {
            x = x - 2y;
        } else {
            y = y - x;
        }
    }
    return x;
}
```

Let's test with gcd(15, 6) and gcd(4, 8). What's the statement coverage? Branch?
Def-use Coverage Targets
- The **All-defs** coverage target: for every \texttt{def} \texttt{x}, cover at least one path (free of definition of \texttt{x}), to at least one \texttt{use} \texttt{x}
- The **All-uses** coverage target: for every \texttt{def-use pair of x}, cover at least one path (free of definition of \texttt{x}) from the \texttt{def} \texttt{x} to the \texttt{use} \texttt{x}
- The **All-du-paths** coverage target: for every \texttt{def-use pair of x}, cover every path (free of definition of \texttt{x}) from the \texttt{def} \texttt{x} to the \texttt{use} \texttt{x}

Order def-use targets by strength (implication)
- Coverage target A is stronger than (i.e., implies) coverage target B if a test suite that achieves 100% coverage under A, achieves 100% coverage under B
- All-du-paths => All-uses => All-defs

White Box Testing: Dataflow-based Testing
- Def-use coverage forces more targeted tests
- Higher probability to find errors
- Research has shown it leads to higher quality test suites

Nevertheless, def-use coverage is rarely used in practice. Why?
- Difficult to find ground truth, i.e., the 100%
- Aliasing: x.f = A and B = y.f can be a def-use pair

In Practice
1. Write test suite based on spec (using paths-in-spec, equivalence partitioning, boundary values). Write test suite \textbf{before} code!
2. Write code
3. Run test suite, fix bugs, measure coverage (typically branch)
4. If coverage is inadequate, write more tests. Go to Step 3
- Good “coverage” of paths-in-spec and boundary values typically \textbf{(but not always!)} yields good program coverage

Specification Tests vs. Implementation Tests
- Specification tests are black-box tests
  - Based entirely on the specification
  - If some behavior is undefined in spec, then we cannot write a test case to test this behavior
- Implementation tests are white-box tests
  - Based on code
  - Covers control-flow, different algorithmic paths
  - Code may define behavior undefined in spec. Implementation tests must test this behavior
Regression Testing

Regression testing is the process of re-running test suite (e.g., after a bug fix).
- Whenever you find a bug:
  - Add test case with input that elicited the bug
  - Verify that test suite fails
  - Fix the bug
  - Verify that test suite succeeds
- This ensures that we populate test suite with good tests

Testing, Summary

- Testing is extremely important. Two rules:
  - Test early and test often
  - Be systematic
- Large test suite does not mean quality test suite:
  - Can miss relevant test cases
  - Use black box and white box heuristics
  - Write tests based on
    - Specification (black-box heuristics)
    - Implementation (white-box heuristics)
- Testing cannot prove absence of bugs:
  - But can increase confidence in program

Outline

- Equality
  - Equality and mutation
  - Implementing equals and hashCode efficiently
  - Equality in ADTs
- Testing
  - Def-use testing
  - Testing Summary
- Exceptions
  - Exceptions vs. preconditions
  - 2 uses of exceptions: failure and special result

What to Do When Something Goes Wrong?

- Exceptions help us deal with failure
- Two rules of failure: fail friendly, fail early to prevent harm
- Goal 1: Give information
  - To the programmer, to the client code
- Goal 2: Prevent harm
  - Abort: inform a human, cleanup, log error, etc.
  - Retry: problem might be temporary
  - Skip subcomputation: permit rest of program to continue
  - Fix the problem (usually infeasible)

Preconditions vs. Exceptions

- A precondition prohibits misuse of your code
  - Adding a precondition weakens the spec
- A precondition ducks the problem
  - Behavior of your code when precondition is violated is unspecified!
  - Does not help clients violating precondition of your code
- Removing the precondition requires specifying the behavior. Strengthens the spec
  - Example: specify that an exception is thrown

Preconditions vs. Exceptions

- In certain cases, a precondition is the right choice
  - When checking would be expensive. E.g., array is sorted
  - In private methods, usually used in local context
- Whenever possible, remove preconditions from public methods and specify behavior
  - Usually, this entails throwing an Exception
  - Stronger spec, easier to use by client
Square Root, With Precondition and Assertions

// requires: x >= 0
// returns: approximation to square root of x
public double sqrt(double x) {
    assert x >= 0 : "Input must be >=0";
    double result;
    // compute result
    assert(Math.abs(result*result – x) < .0001);
    return result;
}

Better: Square root, Specified for All Inputs

// throws: IllegalArgumentException if x < 0
// returns: approximation to square root of x
public double sqrt(double x) throws IllegalArgumentException {
    double result;
    if (x < 0)
        throw new IllegalArgumentException("...");
    // compute result
    return result;
}

Client code:
try {
    y = sqrt(-1);
} catch (IllegalArgumentException e) {
    e.printStackTrace(); // or take same other
}

Exception is handled by catch block associated with nearest dynamically enclosing try
Top-level handler: print stack trace, terminate program

Throwing and Catching

Exceptions allow non-local error handling
- A method far down the call stack can handle a deep error!
- Java maintains a call stack of methods that are currently executing
- When an exception is thrown, control transfers to the nearest method with a matching catch block
- If none found, top-level handler

Propagating an Exception through the Call Chain

// throws: IllegalArgumentException if no real solution exists
// returns: x such that ax^2 + bx + c = 0
double solveQuad(double a, double b, double c) throws IllegalArgumentException {
    ...
    // exception thrown by sqrt is declared,
    // no need to catch it here
    return (-b + sqrt(b*b - 4*a*c))/(2*a);
}

Informing the Client of a Problem

Special value
- null = Map.get(x)
- -1 = List.indexOf(x)
- NaN = sqrt of negative number

Problems with using special value
- Hard to distinguish from real values
- Error-prone: programmer forgets to check result?
  The value is illegal and will cause problems later
- Ugly
- Better solution: exceptions
Two Distinct Uses of Exceptions

Failures
- Unexpected by your code (code is written so that failure does not happen)
- Usually unrecoverable. If condition is left unchecked, exception propagates up the stack

Special results
- Expected by your code
- Always check and handle locally. Take special action and continue computing

Java Exceptions: Checked vs. Unchecked Exceptions

Checked exceptions. For special results
- Method: must declare in signature
- Caller: must either catch or declare in signature
- It is guaranteed there is a dynamically enclosing catch

Unchecked exceptions. For failures
- Library: no need to declare
- Client: no need to catch
- RuntimeException and Error

Java Exception Hierarchy (Part of)

Exceptions, review

Use an exception when
- Checking the condition is feasible
- Method is used in a broad or unpredictable context

Use a precondition when
- Checking would be prohibitive
  - E.g., requiring that a list is sorted
- Method is used in a narrow context in which calls can be checked

Don’t Ignore Exceptions

An empty catch block is poor style!
- Often done to hide an error or get to compile
  try {
    readFile(filename);
  } catch (IOException e) {} // do nothing on error
- At a minimum, print the exception
  } catch (IOException e) {
    e.printStackTrace();
  }

Avoid preconditions because
- Caller may violate precondition
- Program can fail in an uninformative or dangerous way
- Want program to fail as early as possible

Use checked exceptions most of the time
- Handle exceptions sooner than later