

Catch up: Equality, Testing and Exceptions

Based on material by Michael Ernst,
University of Washington

Announcements

- HW4 is due today
- HW5 bases off of HW4, will be out tonight, due Oct. 30th

2

Last class

- Equality
 - Reference equality
 - “Value” equality with `.equals`
 - Equality and inheritance
 - `equals` and `hashCode`
- Testing
 - White-box testing

Fall 15 CSCI 2600, A Milanova

3

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

4

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?

Fall 15 CSCI 2600, A Milanova

5

`equals()` and `hashCode()`

- Important when using Hash-based containers

```
class Duration {
    public final int min;
    public final int sec;
    public Duration(int min, int sec) {
        this.min = min;
        this.sec = sec;
    }
}

Set<Duration> ds = new HashSet<>();
ds.add(new Duration(5,10)); // true or false?
ds.add(new Duration(5,10)); // true or false?
ds.contains(new Duration(5,10)); // T or F?
```

6

equals() and hashCode()

- Now, let's add `equals` and `hashCode()`:

```
public boolean equals(Object o) {
    if (!(o instanceof Duration)) return false;
    Duration d = (Duration o);
    return min == d.min && sec == d.sec;
}

public int hashCode() { return min+sec; }
```

```
Set<Duration> ds = new HashSet<>();
ds.add(new Duration(5,10)); // T or F?
ds.add(new Duration(5,10)); // T or F?
ds.contains(new Duration(5,10)); // T or F?
```

7

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)

Fall 15 CSCI 2600, A Milanova (slide by Michael Ernst)

8

StringBuffer Example

- `StringBuffer` is mutable, and takes the **eternal** approach

```
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`

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

9

Date Example

- `Date` is mutable, and takes the **"compare values now"** approach

```
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
```

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

10

Equality and Mutation

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

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

11

Equality and Mutation

- Sets assume hash codes don't change

```
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
s.contains(new Date(1000)); // false again
```

Fall 15 CSCI 2600, A Milanova

12

Equality and Mutation

- Be very careful with elements of Sets
- Ideally, elements will be immutable objects
- Java spec for Sets warns about using mutable objects as set elements
- Same problem applies to keys in maps

Fall 15 CSCI 2600, A.Milanov

13

Implementing `equals` Efficiently

- `equals` can be expensive!
- How can we speed-up `equals`?

```
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)  
}
```

14

Example: A Naïve `Poly.equals`

```
// 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;  
    }  
    else  
        return false;  
}
```

15

Example: Better `equals`

```
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;  
}
```

Fall 15 CSCI 2600, A.Milanov

16

Implementing `hashCode`

```
// returns: the hashCode value of this String  
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!

Fall 15 CSCI 2600, A.Milanov

17

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`

18

Example: Line Segment

```
class LineSegment {
// Rep invariant:
// !(x1=x2 && y1=y2)
float x1,y1;
float x2,y2;
...
}
// equals must
// return true for
// {x1:1,y1:2,x2:4,y2:5}
// and {4,5,1,2}
class LineSegment {
// Rep invariant:
// x1<x2 ||
// x1=x2 && y1<y2
float x1,y1;
float x2,y2;
...
}
// equals is simpler:
// {4,5,1,2} is not
// valid rep anymore
```

Fall 15 CSCI 2600, A Milanova

19

Other Examples

- RatNum
- RatPoly

Fall 15 CSCI 2600, A Milanova

20

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

21

White Box Testing

```
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];
}
```

Spring 15 CSCI 2600, A Milanova (example due to Michael Ernst)

22

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

Spring 15 CSCI 2600, A Milanova

23

White Box Testing: Dataflow-based Testing

- A **definition (def)** of x is x at the left-hand-side
 - E.g., $x = y+z$, $x = x+1$, $x = \text{foo}(y)$
- A **use of x** is when x is at the right-hand side
 - E.g., $z = x+y$, $x = x+y$, $x > y$, $z = \text{foo}(x)$
- 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



24

White Box Testing: Dataflow-based Testing

- Dataflow-based testing targets: write tests that cover paths between **def-use pairs**
- Intuition:
 - If code computed a wrong value at a **def of x**, the more uses of this **def of x** we “cover”, the higher the probability we’ll expose the error
 - If code had erroneous **use of x**, the more def-use pairs we “cover”, the higher the probability we’ll expose the use

25

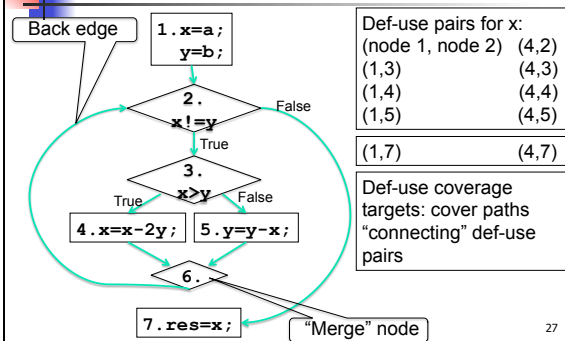
A Buggy gcd

```
// 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?

26

CFG for Buggy GCD



27

Def-use Coverage Targets

- The **All-defs** coverage target: for every **def x**, cover at least one path (free of definition of **x**), to at least one **use x**
- The **All-uses** coverage target: for every **def-use pair of x**, cover at least one path (free of definition of **x**) from the **def x** to the **use x**
- The **All-du-paths** coverage target: for every **def-use pair of x**, cover every path (free of definition of **x**) from the **def x** to the **use x**

Spring 15 CSCI 2600, A Milanova

28

Def-use Coverage Targets

- 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

Spring 15 CSCI 2600, A Milanova

29

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

Spring 15 CSCI 2600, A Milanova

30

White Box Testing

- Covering “all of the program”
 - Statement coverage
 - Branch coverage
 - Def-use coverage (a kind of path coverage)
 - Path coverage
- Just a heuristic
 - Even 100% all-uses coverage (one of the strongest targets) can still miss bugs
- High coverage increases confidence in test suite

31

In Practice

1. Write test suite based on spec (using paths-in-spec, equivalence partitioning, boundary values). Write test suite 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 (**but not always!**) yields good program coverage

32

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

Spring 15 CSCI 2600, A Milanova

33

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
 - Ensures that we populate test suite with good tests

Spring 15 CSCI 2600, A Milanova

34

Rules of Testing

- First rule of testing: **do it early and do it often**
 - Best to catch bugs soon, before they hide
 - Automate the process
 - Regression testing will save time
- Second rule of testing: **be systematic**
 - Writing tests is a good way to understand the spec
 - Specs can be buggy too!
 - When you find a bug, write a test first, then fix

Spring 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

35

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

36

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

37

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)

Fall 15 CSCI 2600, A. Milanova

38

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

Fall 15 CSCI 2600, A. Milanova

39

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

Fall 15 CSCI 2600, A. Milanova

40

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;
}
```

Fall 15 CSCI 2600, A. Milanova

41

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;
}
```

Fall 15 CSCI 2600, A. Milanova

42

Better: Square root, Specified for All Inputs

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**

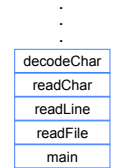
Top-level handler: print stack trace, terminate program

Fall 15 CSCI 2600, A Milanova

43

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



Fall 15 CSCI 2600, A Milanova

44

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);
}
```

Fall 15 CSCI 2600, A Milanova

45

The **finally** Block

- **finally** is always executed
 - No matter whether exception is thrown or not
 - Useful for clean-up code
- ```
FileWriter out = null;
try {
 out = new FileWriter(...);
 ... write to out; may throw IOException
} finally {
 if (out != null) {
 out.close();
 }
}
```

Fall 15 CSCI 2600, A Milanova

46

## 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

47

## Two Distinct Uses of Exceptions

- Failures (e.g., null pointer)
  - Unexpected by your code
  - 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
  - Unknowable for the client of your code

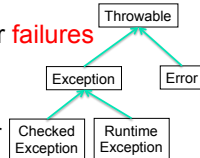
Fall 15 CSCI 2600, A Milanova

48



## Java Exceptions: Checked vs. Unchecked Exceptions

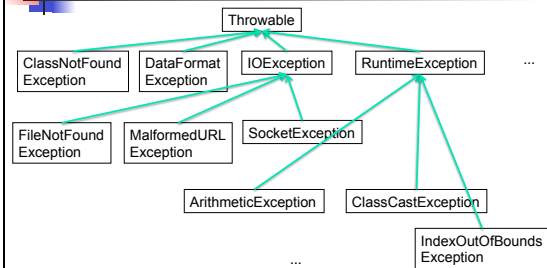
- **Checked** exceptions. For **special results**
  - Method: must declare in signature
  - Caller: must either catch or declare in signature
  - It is guaranteed there is a dynamically enclosing catch
- **Unchecked** exceptions. For **failures**
  - Library: no need to declare
  - Client: no need to catch
  - RuntimeException and Error



Fall 15 CSCI 2600, A. Milanova

49

## Java Exception Hierarchy (Part of)



Fall 15 CSCI 2600, A. Milanova

50

## 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();
}

```

Fall 15 CSCI 2600, A. Milanova

51

## Exceptions, review

- Use an exception when
  - Checking the condition is feasible
  - Used in a broad or unpredictable context
- Use a precondition when
  - Checking would be prohibitive
    - E.g., requiring that a list is sorted
  - Used in a narrow context in which calls can be checked

Fall 15 CSCI 2600, A. Milanova

52

## Exceptions, review

- 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

Fall 15 CSCI 2600, A. Milanova

53