

HW Server Survey:

<http://goo.gl/forms/k6ReldoPNG>

Or just go to the **Announcements** page, the link is there

As a thank you for filling up the survey, we give an extra late day to use on HW4 or later

Testing, cont., Equality and Identity

Based on material by Michael Ernst,
University of Washington

Announcements

- HW4 is due October 20th
 - **LONG!!!**
 - Several tasks:
 - 1) design the ADT following the ADT methodology
 - Mutable vs. immutable
 - Operations (creators, mutators, etc.) + their specs
 - 2) write a test suite based on those specs (before coding, test-first principle)
 - 3) then code
 - 4) then augment test suite based on actual implementation and measure coverage
 - When done, Submit in Homework Server

3

Outline

- Testing
 - Strategies for choosing tests
 - Black box testing
 - **White box testing**
- Equality and identity
- **Quiz 5 at the end of class**

Fall 15 CSCI 2600, A Milanova

4

Why Is Testing So Hard?

// requires: $1 \leq x, y, z \leq 10000$

// returns: computes some $f(x, y, z)$

```
int proc(int x, int y, int z)
```

- Exhaustive testing would require 1 trillion runs! And this is a trivially small problem
- The key problem: **choosing test suite**
 - Small enough to finish quickly
 - Large enough to validate program

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

5

Testing Strategies

- Test case: specifies
 - Inputs + pre-test state of the software
 - Expected result (outputs and post-test state)
- **Black box testing**:
 - We ignore the code of the program. We look at the specification (roughly, given some input, was the produced output correct according to the spec?)
 - Choose inputs without looking at the code
- **White box (clear box, glass box) testing**:
 - We use knowledge of the code of the program (roughly, we write tests to "cover" internal paths)
 - Choose inputs with knowledge of implementation

Fall 15 CSCI 2600, A Milanova

6

Black Box Testing Heuristics

- Paths in specification
 - A form of equivalence partitioning
- Equivalence partitioning
- Boundary value analysis
 - Arithmetic (Inputs: smallest/largest values)
 - Objects (Inputs: null objects, circular list, aliasing)

7

White Box Testing

- Ensure test suite covers (covers means executes) "all of the program"
- Measure quality of test suite with % coverage
- Assumption: high coverage implies few remaining errors in program
- Focus: features in code not described in specification
 - Performance optimizations
 - Alternate algorithms (paths) for different cases

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

8

White Box Complements Black Box

```
boolean[] primeTable[CACHE_SIZE]
// requires: x > 0
// 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 15 CSCI 2600, A Milanova (example due to Michael Ernst)

9

White Box Testing: Control-flow-based Testing

- Control-flow-based white box testing:
 - Extract a control flow graph (CFG)
 - Test suite must cover (execute) certain elements of this control flow graph
- Idea: Define a coverage target and ensure test suite covers target
 - Targets: CFG nodes, branch edges, paths, etc.
 - Coverage target approximates "all of the program"

Fall 15 CSCI 2600, A Milanova

10

Aside: Control Flow Graph (CFG)

- Assignment $x=y+z \Rightarrow$ node in CFG: $x=y+z$
- If-then-else

$\text{if (b) S1 else S2} \Rightarrow$

Fall 15 CSCI 2600, A Milanova

11

Aside: Control Flow Graph (CFG)

- Loop

$\text{while (b) S} \Rightarrow$

Fall 15 CSCI 2600, A Milanova

12

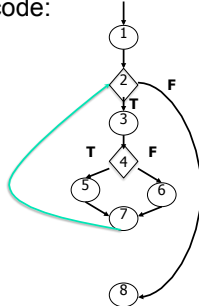
Aside: Control Flow Graph (CFG)

- Draw the CFG for this code:

```

1 s = 0;
  x = 0;
2 while (x < y) {
3   x = x + 3;
  y = y + 2;
4   if (x + y < 10)
5     s = s + x + y;
  else
6     s = s + x - y;
7 }
8 res = s;

```



13

Statement Coverage

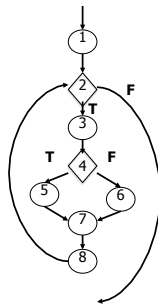
- Traditional target: **statement coverage**. Write test suite that covers **all statements**, or in other words, **all nodes in the CFG**
- Intuition: code that has never been executed during testing may contain errors
 - Often this is the “low-probability” code

Fall 15 CSCI 2600, A Milano

14

Example

- Suppose that we run two test cases
- Test case #1: follows path 1-2-exit (e.g., we never take the loop)
- Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit (loop twice, and both times take the true branch)
- Problems?



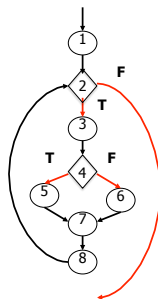
Branch Coverage

- Target: write test cases that cover all **branch edges** at predicate nodes
 - True and false branch edges of each if-then-else
 - The two branch edges corresponding to the condition of a loop
 - All alternatives in a SWITCH statement
- In modern languages, branch coverage implies statement coverage!
 - I.e., a test suite that achieves 100% branch coverage achieve 100% statement coverage

16

Example

- We need to cover the **red** branch edges
- Test case #1: follows path 1-2-exit
- Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit
- What is % branch coverage?



Branch Coverage

- Statement coverage does not imply branch coverage
 - I.e., a suite that achieves 100% statement coverage does not necessarily achieve 100% branch coverage
 - Can you think of an example?
- Motivation for branch coverage: experience shows that many errors occur in “decision making” (i.e., branching). Plus, it implies statement coverage

18

Example

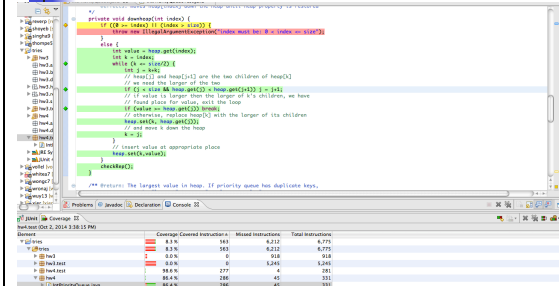
```
static int min(int a, int b) {
    int r = a;
    if (a <= b)
        r = b;
    return r;
}
```

- Let's test with min(1,2)
- What is the statement coverage?
- What is the branch coverage?

Fall 15 CSCI 2600, A. Milanova

19

Code Coverage in Eclipse



Fall 15 CSCI 2600, A. Milanova

20

Other White Box Heuristics

- White box equivalence partitioning and boundary value analysis
- Loop testing
 - Skip loop
 - Run loop once
 - Run loop twice
 - Run loop with typical value
 - Run loop with max number of iterations
 - Run with boundary values near loop exit condition
- Branch testing
 - Run with values at the boundaries of branch condition

21

Outline

- Reference equality
- "Value" equality with `equals`
- Equality and inheritance
- `equals` and `hashCode`
- Equality and mutation
- Implementing `equals` and `hashCode` efficiently
- Equality in ADTs

Fall 15 CSCI 2600, A. Milanova

22

Equality

- Simple idea:
 - 2 objects are equal if they have the same value
- Many subtleties
 - Same reference, or same value?
 - 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

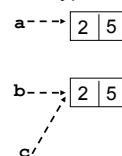
23

Equality: `==` and `equals`

- Java uses the reference model for class types

```
class Point {
    int x; // x-coordinate
    int y; // y-coordinate
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}


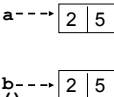
a = new Point(2,5);
b = new Point(2,5);
c = b;
```



true or false? `a == b` ?
 true or false? `b == c` ?
 true or false? `a.equals(b)` ?
 true or false? `b.equals(c)` ?

24

Equality: == and equals

- In Java, `==` tests for **reference equality**. This is the **strongest** form of equality
- Usually we need a **weaker** form of equality, **value equality**

- In our `Point` example, we want `a` to be "equal" to `b` because the `a` and `b` objects hold the same value
 
 - Need to override `Object.equals()`

Fall 15 CSCI 2600, A Milanova

25

Properties of Equality

- Equality is an **equivalence relation**
 - Reflexive** `a.equals(a)`
 - Symmetric** `a.equals(b) ⇔ b.equals(a)`
 - Transitive** `a.equals(b) ∧ b.equals(c) ⇒ a.equals(c)`
- Is reference equality an equivalence relation?
 - Yes

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

26

Object.equals method

- `Object.equals` is very simple:

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

27

Object.equals Javadoc spec

Indicates whether some other object is "equal to" this one. The `equals` method implements an equivalence relation:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return true.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` should return true.
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

28

Object.equals Javadoc spec

For any non-null reference value `x`, `x.equals(null)` should return false.

The `equals` method for class `Object` implements **the most discriminating possible** (i.e., the **strongest**) equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns true if and only if `x` and `y` refer to the same object (`x == y` has the value true)...

Parameters:

`obj` - the reference object with which to compare.

Returns:

true if this object is the same as the `obj` argument;
false otherwise.

See Also:

`hashCode()`, `HashMap`

29

The Object.equals Spec

- Why this complex specification? Why not just **returns**: true if `obj == this`, false otherwise
- `Object` is the superclass for all Java classes
 - The specification of `Object.equals` must be as **weak** (i.e., general) as possible
- Subclasses must be **substitutable** for `Object`
 - Thus, subclasses need to provide stronger **equals**!
 - `obj == this` is already the strongest form of equality! Places undue restriction on subclasses: no subclass can weaken **equals** and still be substitutable for `Object`!
 - Javadoc spec lists the properties of equality, the **weakest** possible specification of **equals**

30

Adding equals

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

Duration d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
System.out.println(d1.equals(d2)); // prints?
```

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

31

First Attempt to Add equals

```
public class Duration {
    public boolean equals(Duration d) {
        return
            this.min == d.min && this.sec == d.sec;
    }
}

Duration d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
System.out.println(d1.equals(d2)); Yields what?
■ Is equals reflexive, symmetric and transitive?
■ This equals is incorrect though. Why? Fix?
```

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

32

What About This?

```
public class Duration {
    public boolean equals(Duration d) {
        return
            this.min == d.min && this.sec == d.sec;
    }
}

Object d1 = new Duration(10,5);
Object d2 = new Duration(10,5);
System.out.println(d1.equals(d2)); Yields what?
```

d1's compile-time type is Object.
d1's runtime type is Duration.

Compiler looks at d1's compile-time type.
Chooses signature equals(Object).

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

33

A Correct equals

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

Object d1 = new Duration(10,5);
Object d2 = new Duration(10,5);
System.out.println(d1.equals(d2)); Yields what?
```

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

34

Outline

- Reference equality
- "Value" equality with .equals
- Equality and inheritance
- equals and hashCode
- Equality and mutation
- Implementing equals and hashCode efficiently
- Equality and ADTs

Fall 15 CSCI 2600, A Milanova

35

Add a Nano-second Field

```
public class NanoDuration extends Duration {
    private final int nano;
    public NanoDuration(int min,
                        int sec,
                        int nano) {
        this.super(min, sec); // initializes min&sec
        this.nano = nano;
    }
}

■ What if we don't add NanoDuration.equals?
(Assume Duration.equals as in slide 34)
```

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

36

First Attempt at NanoDuration.equals

```
public boolean equals(Object o) {
    if (! (o instanceof NanoDuration) )
        return false;
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nd.nano == nano;
}
```

```
Duration d1 = new NanoDuration(5,10,15);
Duration d2 = new Duration(5,10);
d1.equals(d2); Yields what?
d2.equals(d1); Yields what?
```

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

37

Possible Fix for NanoDuration.equals

```
public boolean equals(Object o) {
    if (! (o instanceof Duration) )
        return false;
    if (! (o instanceof NanoDuration) )
        return super.equals(o); //compare without nano
    NanoDuration nd = (NanoDuration) o;
    return super.equals(o) && nd.nano == nano;
}
```

- Does it fix the symmetry bug?
- What can go wrong?

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

38

Possible Fix for NanoDuration.equals

```
Duration d1 = new NanoDuration(10,5,15);
Duration d2 = new Duration(10,5);
Duration d3 = new NanoDuration(10,5,30);
```

```
d1.equals(d2); Yields what?
d2.equals(d3); Yields what?
d1.equals(d3); Yields what?
```

equals is not transitive!

d1 →

10	5	15
----	---	----

d2 →

10	5
----	---

d3 →

10	5	30
----	---	----

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

39

One Solution: Checking Exact Class, Instead of instanceof

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

- Problem: every subclass must implement **equals**; sometimes, we want to compare distinct classes!

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

40

Another Solution: Composition

```
public class NanoDuration {
    private final Duration duration;
    private final int nano;
    ...
}
```

Composition does solve the **equals** problem:
Duration and **NanoDuration** are now unrelated,
so we'll never compare a **Duration** to a
NanoDuration

Problem: Can't use **NanoDuration** instead of
Duration. Can't reuse code written for **Duration**.

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

41

A Reason to Avoid Subclassing Concrete Classes. More later

- In the JDK, subclassing of concrete classes is rare. When it happens, there are problems

- One example: **Timestamp** extends **Date**

- Extends **Date** with a nanosecond value
- But **Timestamp** spec lists several caveats
 - E.g., **Timestamp.equals(Object)** method is **not symmetric** with respect to **Date.equals(Object)** (the symmetry problem we saw on the previous slide)

42

Abstract Classes

- Prefer subclassing abstract classes
 - Just like in real life. “Superclasses” in real life cannot be instantiated
- There is no equality problem if superclass cannot be instantiated!
 - E.g., if `Duration` were abstract, the issue of comparing `Duration` and `NanoDuration` never arises

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

43

Outline

- Reference equality
- “Value” equality with `.equals`
- Equality and inheritance
- `equals` and `hashCode`
- Equality and mutation
- Implementing `equals` and `hashCode` efficiently
- Equality and ADTs

Fall 15 CSCI 2600, A Milanova

44

The `int hashCode` Method

- `hashCode` computes an index for the object (to be used in hashtables)
- Javadoc for `Object.hashCode()`:
 - “Returns a hash code value of the object. This method is supported for the benefit of hashtables such as those provided by `HashMap`.”
 - Self-consistent: `o.hashCode() == o.hashCode()`
... as long as `o` does not change between the calls
 - Consistent with `equals()` method: `a.equals(b)`
=> `a.hashCode() == b.hashCode()`

45

The `Object.hashCode` Method

- `Object.hashCode`’s implementation returns a **distinct integer** for each **distinct object**, typically by converting the object’s address into an integer
- `hashCode` must be consistent with equality
 - `equals` and `hashCode` are used in hashtables
 - If `hashCode` is inconsistent with `equals`, the hashtable behaves incorrectly
 - Rule: if you override `equals`, override `hashCode`; must be consistent with `equals`

46

Implementations of `hashCode`

Remember, we defined

`Duration.equals(Object)`

```
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; }`

Fall 15 CSCI 2600, A Milanova

47

`hashCode` Must Be Consistent with `equals`

- Suppose we change `Duration.equals`
// Returns true if `o` and `this` represent the same number of // seconds

```
public boolean equals(Object o) {
    if (!(o instanceof Duration)) return false;
    Duration d = (Duration) o;
    return 60*min+sec == 60*d.min+d.sec;
}
```
- Will `min+sec` for `hashCode` still work?

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

48

Outline of today's class

- Reference equality
- "Value" equality with `.equals`
- Equality and inheritance
- `equals` and `hashCode`
- Equality and mutation
- Implementing `equals` and `hashCode` efficiently
- Equality and ADTs

Fall 15 CSCI 2600, A Milanova

49

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
 - Abstract value never changes, equality is **eternal**
- For mutable objects
 - We can either compare abstract values **now**, or
 - be **eternal** (can't have both since value can change)

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

50

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 mutable objects

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

51

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)

52

Behavioral and Observational Equivalence

- Two objects are **"behaviorally equivalent"** if there is no sequence of operations that can distinguish them
- Two objects are **"observationally equivalent"** if there is no sequence of observer operations that can distinguish them
 - We are excluding mutators
 - Excluding `==`

Fall 15 CSCI 2600, A Milanova

53

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); }
```

54

Equality and Mutation

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

Fall 15 CSCI 2600, A Milanova

55

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

56

Equality and Mutation

- Redefining `equals` and `hashCode` makes most sense for immutable, “value”, objects
 - E.g., String, RatNum
- Be careful with `equals` and `hashCode` on mutable objects
 - From spec of `Object.equals`: It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

57

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)  
}
```

58

Example: A Naive `RatPoly.equals`

```
public boolean equals(Object o) {  
    if (o instanceof RatPoly) {  
        RatPoly rp = (RatPoly) o;  
        int i=0;  
        while (i<Math.min(rp.c.length,c.length)) {  
            if (rp.c[i] != c[i]) // Assume int arrays  
                return false;  
            i = i+1;  
        }  
        if (i != rp.c.length || i != c.length) return false;  
        return true;  
    }  
    else  
        return false;  
}
```

Fall 15 CSCI 2600, A Milanova

59

Example: Better `equals`

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

Fall 15 CSCI 2600, A Milanova

60

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 Milanova

61

Outline of today's class

- Reference equality
- “Value” equality with `.equals`
- Equality and inheritance
- `equals` and `hashCode`
- Equality and mutation
- Implementing `equals` and `hashCode` efficiently
- Equality and ADTs

Fall 15 CSCI 2600, A Milanova

62

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**

63

Example: Line Segment

<pre>class LineSegment { // Rep invariant: // !(x1=x2 && y1=y2) float x1,y1; float x2,y2; ... } // equals must // return <u>true</u> for // {x1:1,y1:2,x2:4,y2:5} // and {4,5,1,2}</pre>	<pre>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</pre>
--	---

Fall 15 CSCI 2600, A Milanova

64

Other Examples

- RatNum
- RatPoly

Fall 15 CSCI 2600, A Milanova

65