Testing, cont., Equality and Identity

Thanks to Michael Ernst,
University of Washington

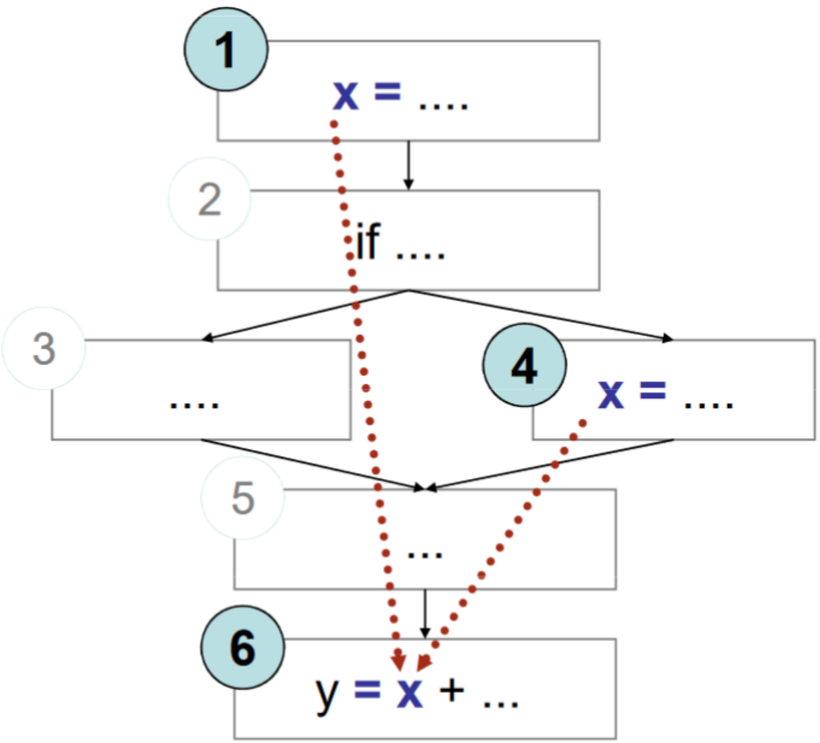
Outline

- Testing
  - Strategies for choosing tests
    - Black box testing
    - White box testing
    - Definition-usage
- Equality and identity

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

Difficulties



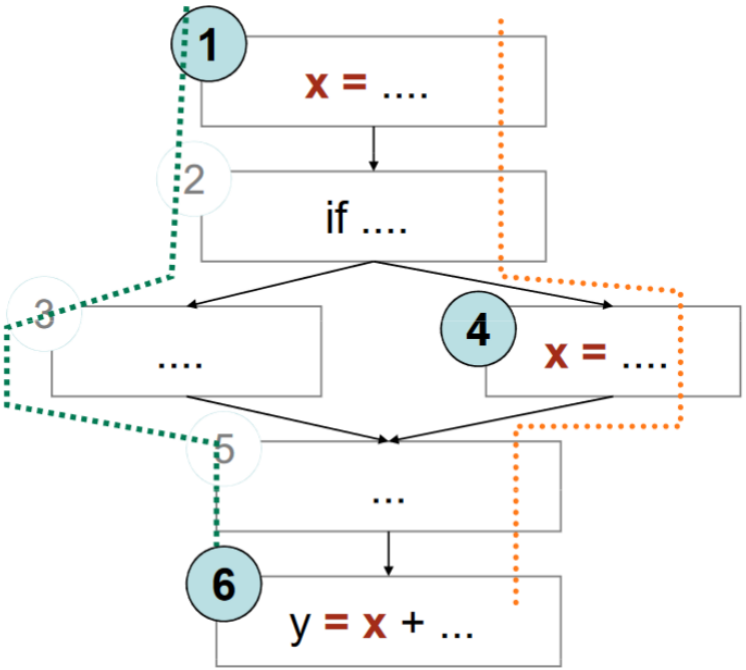
- Value of x at 6 could be computed at 1 or at 4
- Bad computation at 1 or 4 could be revealed only if they are used at 6
- (1,6) and (4,6) are *def-use (DU) pairs*
  - defs at 1,4
  - use at 6

http://www.inf.ed.ac.uk/teaching/courses/st/2015-16/Ch13.pdf

Definition-use Pairs

- A def-use (DU) pair
  - A pair of a definition and use of a variable such that at least one path exists from the definition to the use
  - x = 1;   // definition
  - y = x + 3 // use
- DU path
  - A path from the definition of a variable to a use of the same variable with no other definition of the variable on the path
  - Loops can create infinite DU paths

Definition-clear path



- 1,2,3,5,6 is a definition-clear path from 1 to 6
  - x is not re-assigned between 1 and 6
- 1,2,4,5,6 is not a definition-clear path from 1 to 6
  - the value of x is “killed” (reassigned) at node 4
- (1,6) is a DU pair because 1,2,3,5,6 is a definition-clear path

http://www.inf.ed.ac.uk/teaching/courses/st/2015-16/Ch13.pdf

Adequacy

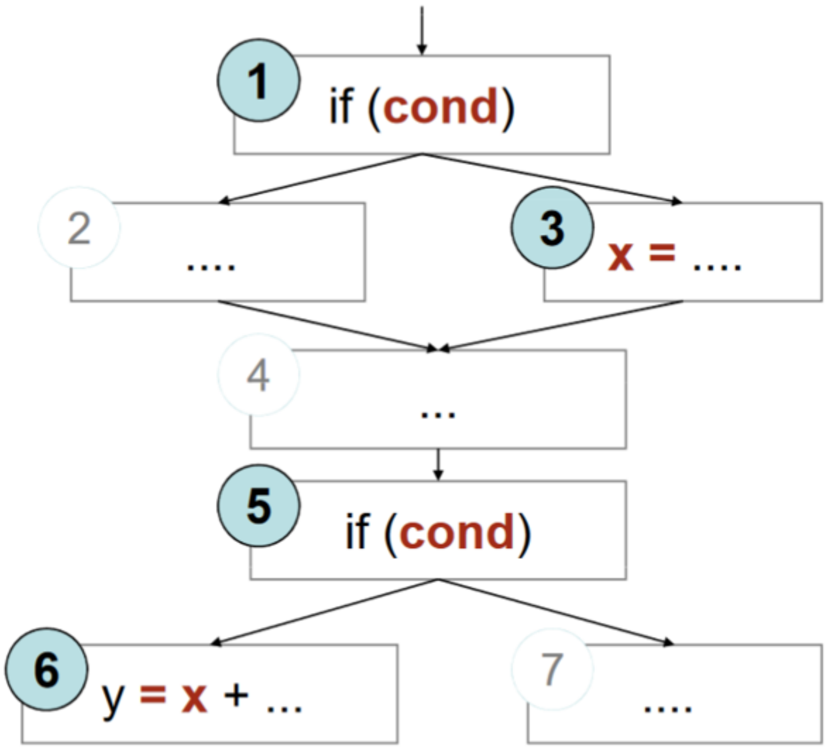
- We want to test:
- All DU pairs
  - Each DU pair tested at least once
- All DU paths
  - Each path is tested at least once
- All definitions
  - For each definition, there is at least one test that exercises a DU path containing it
  - Every computed value is used at least once

Difficulties

- x[i] = some_value; y = x[j];
  - DU pair only if i == j
- Obj x = new Obj(); x = some_value; y = x;
  - y is an alias of x
  - What happens when x or y is used?
- m.putFoo(); y = n.getFoo();
  - Are m and n the same object?
  - Do m and n share a foo?
- Aliases can be a problem

Infeasibility

- Suppose cond doesn't change between 1 and 5
  - Or conditions could be different, but 1 implies 5
- (3, 6) is not a feasible DU path
- It is very difficult to find infeasible paths
- Infeasible paths are a problem
  - Difficult to find
  - Difficult to test



Infeasibility

- Detecting infeasibility can be difficult
  - Combination of elements matter
  - No general way to detect infeasible paths
- In practice the goal is reasonable coverage
  - Number of paths can be large
  - Doing all DU paths might be impractical
- Problems
  - Aliases
  - Infeasible paths
  - Worst case is bad
    - Exponential number of paths
    - Undecidable properties
  - Be pragmatic

Outline

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

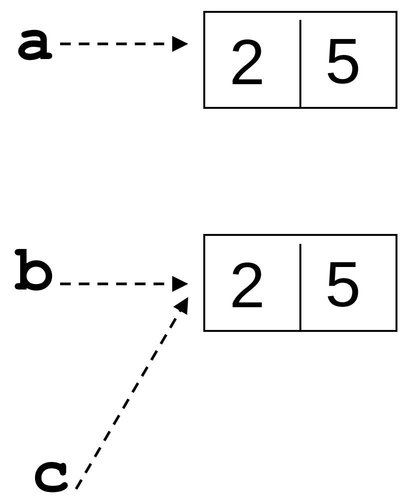
Equality

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

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;
```

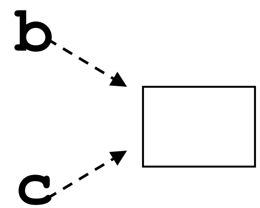


true or false? `a == b` ?
true or false? `b == c` ?
true or false? `a.equals(b)` ?
true or false? `b.equals(c)` ?

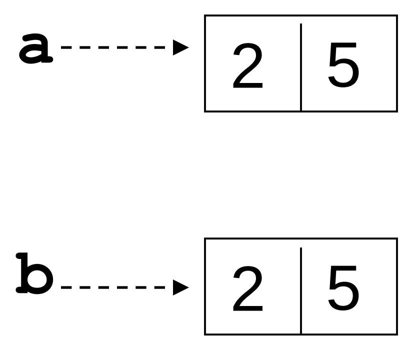
Equality: **`==`** and **`equals`**

- In Java, **`==`** tests for reference equality. This is the strongest form of equality

- Often 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()`**



Properties of Equality

- Equality is an equivalence relation
  - Reflexive $\quad$ `a.equals(a)`
  - Symmetric $\quad$ `a.equals(b)` ⇔ `b.equals(a)`
  - Transitive $\quad$ `a.equals(b)` ∧ `b.equals(c)` ⇒ `a.equals(c)`

- Is reference equality an equivalence relation?
  - Yes

`Object.equals` method

- **`Object.equals`** is very simple:
  - Point extends Object
  - all objects extend Object, implicitly

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

`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, <u>provided no information used in equals comparisons on the objects is modified</u>.

`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`**

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 <u>weak</u> (i.e., general) as possible
- Subclasses must be substitutable for Object
    - Thus, subclasses need to provide stronger **`equals`**!
    - No subclass can weaken **`equals`** and still be substitutable for Object!
    - Javadoc spec lists the properties of equality, the <u>weakest</u> possible specification of **`equals`**

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?
```

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 not quite correct. Why?

Overriding vs. Overloading

- Method overloading is when two or more methods in the same class have the exact same name but different parameters
  - When overloading, one must change either the type or the number of parameters for a method that belongs to the same class. Overriding means that a method inherited from a parent class will be changed.
- Method overriding is when a derived class requires a different definition for an inherited method,
  - The method can be redefined in the derived class.
  - *In overriding* a method, everything remains exactly the same except the method definition – what the method does is changed slightly to fit in with the needs of the child class.
  - the method name, the number and types of parameters, and the return type will all remain the same.
  - Happens at runtime

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)`.

A More 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?

Outline

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

Add a Nano-second Field

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

- What if we don't add **`NanoDuration.equals?`**

`(Assume Duration.equals as in slide 24)`

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?

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
                            // Is this what we want?
  NanoDuration nd = (NanoDuration) o;
  return super.equals(o) && nd.nano == nano;
}
```

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

Possible Fix for **`NanoDuration.equals`**

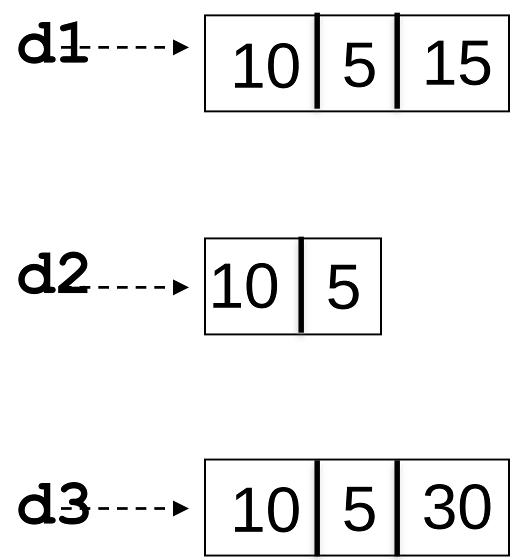
```
Duration d1 = new NanoDuration(10,5,15);
Duration d2 = new Duration(10,5);
Duration d3 = new NanoDuration(10,5,30);
```

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

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

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

`equals` is not transitive!



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!

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.`**

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)

Abstract Classes

- Prefer subclassing abstract classes
  - "Superclasses" 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

Outline

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

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()`**
  - Collections such as HashMap calculate *unicity* using .equals and .hashCode

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`**

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; }`

```
}
```

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?

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

Equality, Mutation and Time

- If two objects are equal now, will they always be equal?
  - In mathematics, the answer is “yes”
    - Given that the object is not a function of time
  - 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)

`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

`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(10000); //some time later
System.out.println(d1.equals(d2)); // false
```

Behavioral and Observational Equivalence

- Two objects are "behaviorally equivalent" if there is no sequence of operations that can distinguish them
  - This is "eternal" equality
  - Two Strings with same content are behaviorally equivalent, two Dates or StringBuffers with same content are not

- Two objects are "observationally equivalent" if there is no sequence of observer operations that can distinguish them
  - •We are excluding mutators
  - •Excluding ==
  - •Two Strings, Dates, or StringBuffers with same content are observationally equivalent.

Equality and Mutation

- Date class implements observational equality
- We can violate the 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) { // prints 2 identical dates
   System.out.println(d);
}
```

Equality and Mutation

- Be very careful with elements of Sets
- Ideally, elements should 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

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); //  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
```

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.

Equality and Mutation

- From JavaDoc
  - 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.
- HashSet.contains() uses hashCode()
- HashSet.contains() method relies on hash values to stay immutable
  - There is an assumption that the hashCode() does not change
- contains() computes the hashCode() of the object it is looking for
  - It searches only the *bucket* that contains the hash value
- The moral
  - If you put mutable objects in a Set, don't modify them and expect operations like contains() to work as expected.

Mutation and hash codes

- Sets assume that the hash codes don't change
- Mutation can break this assumption

```
List<String> friends =
     new LinkedList<String>(Arrays.asList("yoda", "zaphod"));
List<String> enemies =
     new LinkedList<String>(Arrays.asList("Darth Vader", "Joker"));

Set<List<String>> h = new HashSet<List<String>>();
h.add(friends);
h.add(enemies);
friends.add("BatMan");

System.out.println(h.contains(friends)); // probably false
System.out.println();
for (List<String> lst : h) {
     System.out.println(lst.equals(friends));
}  // one "true" will be printed - inconsistent
```

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)
}
```

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;
}
```

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;
}
```

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!

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

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

```
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
```

Rules for overriding equals()

- Overriding equality seems easy but many ways to get it wrong
- Obey the general contract
- Don't do it if
  - Each instance of the class is inherently unique.
  - You don't care whether the class provides a "logical equality" test
    - Random numbers
  - A superclass has already overridden equals, and the behavior inherited from the superclass is adequate for this class.
    - Set inherits its equals from AbstractSet
  - The class is private or package-private, and you are certain that its equals method will never be invoked.

Rules for overriding equals()

- If you need to:
  - Use the == operator to check if the argument is a reference to this object
  - Use the instanceof operator to check if the argument is of the correct type
  - Cast the argument to the correct type
  - For each “significant” field in the class, check to see if that field of the argument matches the corresponding field of this object.
  - When you are finished writing your equals method, ask yourself three questions: Is it symmetric, is it transitive, and is it consistent?
  - Always override hashCode when you override equals
  - Don’t substitute another type for Object in the equals declaration.
  - Eclipse can generate Java hashCode and equals methods
    - Source->Generate hashCode() and equals()‘.