CSE 3302
Lecture 23: Equality and identity
13 April 2010
Nate Nystrom
UTA
Today

• A smörgåsbord:
  • Equality and identity
  • Real numbers
  • Dynamic typing
Equality and identity

• What does it mean for two variables to be equal?

• equality vs. identity
  • identical = names refer to the same object
  • equal = names refer to objects with the same value
Java

• `==` on primitives: equality
• `==` on objects: identity
• `.equals` on objects: equality
Java primitives

- == implements equality
- numbers:
  - 1.0 == 1 == 1L
  - (byte) 256 == 0  // same bit pattern
- floats:
  - +0. == -0.
  - Float.NaN != Float.NaN
Java objects

• Use == for identity, .equals for equality

• new Integer(1) != new Integer(1)
• (new Integer(1)).equals(new Integer(1))
Java strings

• Use == for identity, .equals for equality

• "hello".equals("hello")
• "hello" == "hello"
• "hello" == new String("hello")
Java strings

• Use \texttt{==} for identity, \texttt{.equals} for equality

• \texttt{“hello”
equals\texttt{“hello”}} \hspace{1cm} true
• \texttt{“hello” == “hello”} \hspace{1cm} true
• \texttt{“hello” == new String(“hello”)} \hspace{1cm} false
Java objects

• Classes can override `.equals`
• Default implementation does `==`  
  • Thus `new Object() != new Object()`
Java .equals

- Things to be careful of when overriding .equals:
  - reflexivity, transitivity, symmetry
  - mutable state
  - null
  - consistency
  - hashCode
  - toString
Reflexivity

• `==` should imply `.equals`
• `! .equals` should imply `!=`

• Broken:

```java
class C {
    public boolean equals(Object o) {
        return false;
    }
}
```
Symmetry

- Broken?

```java
class C {
    int x;
    public boolean equals(Object o) {
        return o instanceof C
                && x == ((C) o).x;
    }
}
```
class C {
    int x;
    public boolean equals(Object o) {
        return o instanceof C && x == ((C) o).x;
    }
}

class D extends C {
    int y;
    public boolean equals(Object o) {
        return super.equals(o)
            && o instanceof D && y == ((D) o).y;
    }
}

(new D()).equals(new C()) -- false
(new C()).equals(new D()) -- true
Symmetry

• Solution 1:
  
  class C {
    int x;
    public boolean equals(Object o) {
      return o instanceof C && x == ((C) o).x
      && o.equals(this);
    }
  }
Symmetry

• Solution 2:
  class C {
    int x;
    public boolean equals(Object o) {
      return o.getClass() == this.getClass() && x == ((C) o).x;
    }
  }
}
Symmetry

• Guideline:
  • If a class \( C \) overrides \texttt{equals}, every subclass of \( C \) should override \texttt{equals} also
null

• $x.equals(null)$ should return $false$ for any $x != null$
Consistency

• Multiple invocations of `x.equals(y)` should consistently return either `true` or `false` provided no fields of `x` or `y` change

• Broken (obviously):

```java
class C {
    boolean equals(Object o) {
        return java.lang.Math.random() < .5;
    }
}
```
Mutable state

class C {
    int x;

    boolean equals(Object o) {
        return o.getClass() == getClass()
            && ((C) o).x == x;
    }
}

C c1 = new C();
C c2 = new C();
c1.equals(c2); // true
c1.x = 1;
c1.equals(c2); // false
Mutable state

- Be careful!
  - Object might be changed by another thread, or might change in a non-obvious way

```java
if (x.equals(y)) {
    ... // other thread modifies x.f
    assert x.equals(y); // fails!
}
```

- Guideline: `equals` should not depend on mutable state
hashCode

• If `x.equals(y)`, then
  
  `x.hashCode() == y.hashCode()`

• Otherwise, collections break!

• Guideline: `hashCode` should be overridden whenever `equals` is overridden
hashCode bug

- Guideline: hashCode should not depend on mutable state

- Bug:
  ```java
  hashMap.put(k, v);
  k.mutate();
  assert hashMap.get(k) == v; // fails!
  ```
Deep vs. shallow equality

- Shallow:
  - `.equals` compares fields with `==`
- Deep:
  - `.equals` compares fields with `.equals`
Language and tool support

• Eclipse:
  • warns if hashCode not overridden with equals

• Scala case classes
  • == does structural comparison of fields rather than pointer identity

• Ruby:
  • == same value
  • .eql? same value and same type
  • .equal? same object
Real numbers
Real numbers

• Read:
  • What Every Computer Scientist Should Know About Floating-Point Arithmetic
  • http://docs.sun.com/source/806-3568/ncg_goldberg.html

• Several different possible representations

• Not possible to represent exactly, of course

• => rounding errors
Floats

• Most common representation of real numbers

• Base B and precision p
• B=10, p=3
  • $0.1 = 1.00 \times 10^{-1}$ (exact)
• B=2, p=24
  • $0.1 = 1.10011001100110011001101 \times 2^{-4}$ (inexact)

• In general
  • +/- $d_0.d_1d...d_{p-1} \times B^e$
  • $d.dd...d = \text{significand}$ (or \text{mantissa}), has p digits
  • $e = \text{exponent}$
### IEEE Floats

<table>
<thead>
<tr>
<th>Name</th>
<th>Common name</th>
<th>Base</th>
<th>Digits</th>
<th>$E_{\text{min}}$</th>
<th>$E_{\text{max}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary16</td>
<td>half</td>
<td>2</td>
<td>11</td>
<td>-14</td>
<td>15</td>
</tr>
<tr>
<td>binary32</td>
<td>single, float</td>
<td>2</td>
<td>24</td>
<td>-126</td>
<td>127</td>
</tr>
<tr>
<td>binary64</td>
<td>double</td>
<td>2</td>
<td>53</td>
<td>-1022</td>
<td>1023</td>
</tr>
<tr>
<td>binary128</td>
<td>quadruple</td>
<td>2</td>
<td>113</td>
<td>-16382</td>
<td>16383</td>
</tr>
<tr>
<td>decimal32</td>
<td></td>
<td>10</td>
<td>7</td>
<td>-95</td>
<td>96</td>
</tr>
<tr>
<td>decimal64</td>
<td></td>
<td>10</td>
<td>16</td>
<td>-383</td>
<td>384</td>
</tr>
<tr>
<td>decimal128</td>
<td></td>
<td>10</td>
<td>34</td>
<td>-6143</td>
<td>6144</td>
</tr>
</tbody>
</table>
Extended precision

- 80-bit floating point
- Often “long double” in C
- Supported on x86
  - instruction to switch processor precision mode
  - expensive to switch
  - can’t be done on a per-instruction basis

- Although:
  - sometimes “long double” = double (Visual C++)
  - sometimes “long double” = quad
Rounding errors

- Suppose B=10, p=3

- **Without** rounding:
  - \( x = 2.15 \times 10^{12} \)
  - \( y = 1.25 \times 10^{-5} \)
  - \( x - y = 2.149999999999999875 \times 10^{12} \)
Rounding errors

• Suppose B=10, p=3

• **With** rounding:
  • $x = 2.15 \times 10^{12}$
  • $y = 1.25 \times 10^{-5}$
  • ->
  • $x = 2.15 \times 10^{12}$
  • $y = 0.00 \times 10^{12}$
  • $x - y = 2.15 \times 10^{12}$
Rounding errors

• Suppose B=10, p=3

• 10.1 - 9.93
  • Without rounding: 0.17
  • With rounding:
    • $x = 1.01 \times 10^1$
    • $y = 0.99 \times 10^1$
    • $x - y = 0.02 \times 10^1$
  • error = $|0.2 - 0.17| / 0.17 = 17.65%$
  • rounding error, even though result (0.17) could be represented exactly
Rounding errors

• Theorem:

• if x and y are floats with representation \((B,p)\) with rounding error \(\leq \varepsilon\), then:
  • if subtraction is done with \(p\) digits, \(x - y\) has rounding error \(\leq B-1\)

• If \(B=2\), rounding error \(\leq 1 = 100\%\)!
Guard bits

• Theorem:

• if $x$ and $y$ are floats with representation $(B,p)$ with rounding error $\leq \varepsilon$, then:
  • if subtraction is done with $p+1$ digits, $x - y$ has rounding error $\leq 2\varepsilon$

• IEEE floating point uses an extra guard bit to make rounding errors manageable
Cancellation

• When **subtracting** nearby numbers, most significant digits match and cancel each other

• If operands have rounding error, result of subtraction can be ALL rounding error
Cancellation

- \( b^2 - 4ac \)
- \( b = 3.34, a = 1.22, \) and \( c = 2.28 \)
  - Exact: 0.0292

- \( B=10, \) \( p=3 \)
- \( b^2 = 3.34 \times 3.34 = 11.1556 \rightarrow 11.2 \)
  - error: \( \frac{|11.2 - 11.1556|}{11.1556} = 0.40\% \)
- \( 4ac = 4 \times 1.22 \times 2.28 = 11.1264 \rightarrow 11.1 \)
  - error: \( \frac{|11.1 - 11.1264|}{11.1264} = 0.24\% \)
- \( b^2 - 4ac = 11.2 - 11.1 = 0.1 \)
  - error: \( \frac{|0.1 - 0.0292|}{0.0292} = 242.5\% \) !!!

5:31:37 PM Tuesday, April 27, 2010
Cancellation

- Cancellation is **benign** if operations are exact
- Cancellation is **catastrophic** if operations are rounded
  - most significant bits cancel out
  - rounding error becomes the most significant bits of result — result can be just garbage
Cancellation

- $x^2 - y^2$ catastrophic since both operands are the result of float operations --> both operands rounded

- Exact:
  - $x^2 - y^2 = 2.25$

- $(B=10,p=3)$:
  - $x^2 = 127.69$ -- $128.$
  - $y^2 = 125.44$ -- $125.$
  - $x^2 - y^2 = 128. - 125. = 3.00$

- Error:
  - $|3.00 - 2.25| / 2.25 = 33.3\%$
Area of triangle:
\[ A = \sqrt{s(s-a)(s-b)(s-c)}, \quad s = \frac{(a+b+c)}{2} \]

Suppose \( a \sim b + c \), then \( s \sim a \)

\[ \Rightarrow s - a \text{ cancels out} \]

e.g., \( a = 9.0, \ b = c = 4.53 \)

- exact: \( s = 9.03, \ A = 2.342 \)
- computed: \( s = 9.05, \ A = 3.04 \) !!!
Cancellation

• Rewriting to avoid cancellation:

  • \( A = \frac{\sqrt{((a+(b+c)) \cdot (c - (a - b)) \cdot (c + (a - b)) \cdot (a + (b - c)))}}{4} \)
  • \( a \geq b \geq c \)

  • e.g., \( a = 9.0, b = c = 4.53 \)
  • exact: \( A = 2.342 \)
  • computed: \( A = 2.35 \)
Special “numbers”

- **NaN** - “not a number”
  - produced when result is undefined, indeterminate
  - result of 0/0, sqrt(-1), Inf + -Inf, ...
  - $x \neq x$

- **Inf** - “infinity”
  - produced when result is defined but overflows
  - result of $x/0$, $x > 0$
  - $x/\text{Inf} = +0$, $x > 0$
  - $x/-\text{Inf} = -0$, $x > 0$
Zero

- IEEE floating point has two zeros:
  - +0
  - -0

- +0 == -0

- Ensures $1/(1/x) == x$, even if $x == +/-\text{Inf}$

- Also: underflow for functions like log, not defined at 0
  - +0 can be thought of as +epsilon
Dynamic typing
Dynamic typing

• This lecture:
  • how to handle incomplete type information at run time

• Arises in dynamically typed languages
• Also in statically typed languages because only supertype is known
Dynamically typed languages

- Scheme, CLOS, Dylan, PostScript
- Ruby, Python, PHP, Perl

- Variables do not have a declared type
- They can contain any kind of value

- Operations can be invoked without knowing type of value

- Strong typing: must check value to ensure operations is supports => need to be able to figure out the run-time type of a value
Primitive types

x = 238932;
x = new Foo();

• If variables are untyped, how to know x is actually an int (or not)?
• Must change representation of integers! (and other primitives)
  • box everything into an object?
  • use two words per value?  – type and value
Boxing

- Represent primitives as normal objects
- All variables hold an object (or null)

- Ex: represent an integer with `java.lang=Integer`

\[ a + b \]

\[ \Rightarrow \]

\[ \text{Integer}.\text{valueOf}(((\text{Integer})a).\text{intValue}()) + ((\text{Integer})b).\text{intValue}()) \]

- Compiler can optimize code to unbox when necessary
Tag bits

- Reserve 1-3 bits in each word to identify primitive values
- Advantage: variable fits in a single word
- Disadvantage: extra overhead smaller range of representable values, pointers

- \[12 = 00001100 \rightarrow 001100[00]\]
- \[\text{‘f’} = 00001100 \rightarrow 001100[01]\]
- \[\text{new Foo} = 00110000 \rightarrow 001100[11]\]
Tag bit tricks

• Integers: use zero bit pattern so integer \( n \) represented by number \( 4n \)
  • adding two integers \( a + b \): just add tagged representation
  • multiply: \( a * b \rightarrow a * (b >> 2) \)

• Pointers: represent 4-byte aligned pointer to address \( p \) by \( p' = p+3 \)
  • (don’t need to be able to address every byte)
  • \( [p+k] = [p'+k-3] \)
  • new Foo = 00110000 \rightarrow 001100[11] \)
Primitive types and subtyping

• Java, C#: primitives have no subtyping relationship with other types
  
• Prevents some code reuse,
  • Must write 9 (10) different versions of generic code
    • Object, boolean, byte, short, char, int, long, float, double, (void)

• Workaround: boxing
  • use java.lang.Integer rather than int
Subtyping for primitives

- Solution 1: objects
- Solution 2: tagging
- Solution 3: autoboxing

- Only works in statically-typed languages
- Allows multiple representations of primitive values: boxed and unboxed
- Primitives are represented in efficient way when type is known, as objects when type is unknown

Tuesday, April 27, 2010
Automatic boxing

- Use static type to decide when to box
  - `int x = 9;    // don’t box`
  - `Object x = 9; // box`

- When assigning a primitive to Object, compiler boxes the primitive
  - `Object x = new Integer(9);`

- When casting from object to primitive, unbox
  `int y = (int) x;`

  =>
  `int y = ((Integer) x).intValue();`
Summary

• Equality
  • same value

• Identity
  • same object, implies equality

• Languages support both with different operators

• Java `equals` — be careful

• Floats — be careful, especially with subtraction

• Dynamic typing
  • need to handle variables that can store both objects and primitives