Lecture 10

Equality and Hashcode
Announcements
Announcements

This coming week is the craziest part of the quarter!
• Quiz 4 due tomorrow 10 pm
• HW4 due tomorrow 10 pm

• HW5 due next Thursday
  – Hardest hw in 331 and future hws build on it
• Section tomorrow!
  – important things you need to know for HW5

• Midterm review session Friday 3:30-5 in this room
• Midterm Monday 1:10-2:10 in this room

• Mid-quarter course evaluation Friday (during part of class)
  – Visitor: Jamal from the Center for Teaching and Learning
Equality
Object equality

A simple idea??
- Two objects are equal if they have the same value

A subtle idea: intuition can be misleading
- Same object or same contents?
- Same concrete value or same abstract value?
- Same right now or same forever?
- Same for instances of this class or also for subclasses?
- When are two collections equal?
  - How related to equality of elements? Order of elements?
  - What if a collection contains itself?
- How can we implement equality efficiently?
Mathematical properties of equality

**Reflexive**  
\[ a \text{.equals}(a) == \text{true} \]  
- An object equals itself

**Symmetric**  
\[ a \text{.equals}(b) \iff b \text{.equals}(a) \]  
- Order doesn’t matter

**Transitive**  
\[ a \text{.equals}(b) \land b \text{.equals}(c) \implies a \text{.equals}(c) \]  
- “transferable”

In mathematics, a relation that is reflexive, transitive, and symmetric is an *equivalence relation*.
Reference equality

• Reference equality means an object is equal only to itself
  – \( a == b \) only if \( a \) and \( b \) refer to (point to) the same object

• Reference equality is an equivalence relation
  – Reflexive \( a==a \)
  – Symmetric \( a==b \iff b==a \)
  – Transitive \( a==b \land b==c \Rightarrow a==c \)

• Reference equality is the \textit{smallest} equivalence relation on objects
  – “Hardest” to show two objects are equal (must be same object)
  – Cannot be any more restrictive without violating reflexivity
  – Sometimes but not always what we want
What might we want?

- Sometimes want equivalence relation bigger than ==
  - Java takes OOP approach of letting classes *override* `equals`
Overriding Object’s equals
Object.equals method

```java
public class Object {
    public boolean equals(Object o) {
        return this == o;
    }
    ...
}
```

- Implements reference equality
- Subclasses can override to implement a different equality
- But library includes a contract equals should satisfy
  - Reference equality satisfies it
  - So should any overriding implementation
  - Balances flexibility in notion-implemented and what-clients-can-assume even in presence of overriding
equals specification

public boolean equals(Object obj)
    Indicates whether some other object is “equal to” this one.

The equals method implements an equivalence relation:

• It is reflexive: for any reference value x, x.equals(x) should return true.
• It is symmetric: for any 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 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 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 object is modified.
• For any non-null reference value x, x.equals(null) should return false.
equals specification

- equals contract is:
  - Weak enough to allow different useful overrides
  - Strong enough so clients can assume equal-ish things
    - Example: To implement a set
  - Complete enough for real software

- So:
  - Equivalence relation
  - Consistency, but allow for mutation to change the answer
  - Asymmetric with null
    - null.equals(a) raises exception
    - for non-null a, a.equals(null) must return false
An example

A class where we may want `equals` to mean equal contents

```java
public class Duration {
    private final int min; // RI: min>=0
    private final int sec; // RI: 0<=sec<60

    public Duration(int min, int sec) {
        assert min>=0 && sec>=0 && sec<60;
        this.min = min;
        this.sec = sec;
    }
}
```

– Should be able to implement what we want and satisfy the `equals` contract…
How about this?

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

Two bugs:
1. Violates contract for null (not that interesting)
   - Can add `if(d==null) return false;`
   - But our fix for the other bug will make this unnecessary
2. Does not override `Object`'s `equals` method (more interesting)
Overloading: `String.indexOf`

```java
int indexOf(int ch)
```
Returns the index within this string of the first occurrence of the specified character.

```java
int indexOf(int ch, int fromIndex)
```
Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

```java
int indexOf(String str)
```
Returns the index within this string of the first occurrence of the specified substring.

```java
int indexOf(String str, int fromIndex)
```
Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
Overriding: `String.equals`

**In Object:**

```java
public boolean equals(Object obj)
```

... The equals method for class Object implements the most discriminating possible 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) ...

**In String:**

```java
public boolean equals(Object anObject)
```

Compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.
Overriding vs. Overloading

Consider the following classes

```java
class Foo extends Object {
    Shoe m(Shoe x, Shoe y){ ... }
}

class Bar extends Foo {...}
```

![Class hierarchy diagram]
Overriding vs. Overloading

- The result is method overriding
- The result is method overloading
- The result is a type-error
- None of the above

Method in Foo
Shoe m(Shoe x, Shoe y){ ... }

Possible Methods in Bar
Shoe m(Shoe q, Shoe z) { ... } overriding
HighHeeledShoe m(Shoe x, Shoe y) { ... } overriding
Shoe m(FootWear x, HighHeeledShoe y) { ... } overloading
Shoe m(FootWear x, FootWear y) { ... } overloading
Shoe m(HighHeeledShoe x, HighHeeledShoe y) { ... } overloading
Shoe m(Shoe y) { ... } overloading
FootWear m(Shoe x, Shoe y) { ... } type error
Shoe z(Shoe x, Shoe y) { ... } new method
Overloading versus overriding

In Java:
- A class can have multiple methods with the same name and different parameters (number or type)
- A method *overrides* a superclass method only if it has the same name and exact same argument types

So *Duration’s* `boolean equals(Duration d)` does *not* override *Object’s* `boolean equals(Object d)`

- Overloading is sometimes useful to make several closely related functions with the same name
- Overloading is sometimes confusing since the rules for what-method-gets-called are complicated
- [Overriding covered in CSE143, but not overloading]
Overload resolution

Java’s language spec for resolving Method Invocations (including overload resolution) is about 18 pages long.

In summary

• The declared types of parameters and the object it’s called on determine the signature of the method to call
  – declared type is also known as compile-time type
• The runtime type of the object it’s called on determines which implementation of that method signature gets called
  – this is called dynamic dispatch
Example: Overloading

```java
public class Duration {
    public boolean equals(Duration d) {...}
    ...
}
Duration d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
Object o1 = d1;
Object o2 = d2;
d1.equals(d2);  // true
o1.equals(o2);  // false(!)
d1.equals(o2);  // false(!)
o1.equals(d2);  // false(!)
do1.equals(o1);  // true [using Object’s equals]
```
Overload resolution

In summary

- The declared types of parameters and the object it’s called on determine the signature of the method to call.
- The runtime type of the object it’s called on determines which implementation of that method signature gets called.

```java
do1.equals(d2)
```

- `do1` has declared type `Object`, so the signature `equals(Object)` is chosen.
- The runtime type of `do1` is `Duration`, so `Duration`’s `equals(Object)` method gets called. Since `Duration` doesn’t implement `equals(Object)`, the superclass `Object`’s implementation is called.
Overload resolution

In summary

- The declared types of parameters and the object it’s called on determine the signature of the method to call.
- The runtime type of the object it’s called on determines which implementation of that method signature gets called.

```java
o1.equals(o2)
```

- `o2` has declared type `Object` so the signature `equals(Object)` is chosen.
- The runtime type of `o1` is `Duration`, so `Duration`’s `equals(Object)` method is chosen. Since `Duration` doesn’t implement `equals(Object)`, the superclass `Object`’s implementation is called.
public class Duration {
    public boolean equals(Object d) {...}
    ...
}
Duration d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
Object o1 = d1;
Object o2 = d2;
d1.equals(d2); // true
o1.equals(o2); // true [overriding]
d1.equals(o2); // true [overriding]
o1.equals(d2); // true [overriding]
d1.equals(o1); // true [overriding]
But wait!

This doesn’t actually compile:

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

• Cast cannot fail
• We want equals to work on any pair of objects
• Gets null case right too (null instanceof C always false)
• So: rare use of cast that is correct and idiomatic
  – This is what you should do (cf. Effective Java)
Satisfies the contract

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

- Reflexive: Yes
- Symmetric: Yes, even if o is not a Duration!
  - (Assuming o’s equals method satisfies the contract)
- Transitive: Yes, similar reasoning to symmetric
Even better

- Great style: use the @Override annotation when overriding

```java
public class Duration {
    @Override
    public boolean equals(Object o) {
        ...
    }
}
```

- Compiler warning if not actually an override
  - Catches bug where argument is Duration or String or ...
  - Alerts reader to overriding
    - Concise, relevant, checked documentation
Summary: Overriding Equals

Equals contract – Equals must implement an equivalence relation
• Reflexive  \( \text{a.equals(a)} \)
• Symmetric  \( \text{a.equals(b) } \Leftrightarrow \text{b.equals(a)} \)
• Transitive  \( \text{a.equals(b) } \land \text{b.equals(c) } \Rightarrow \text{a.equals(c)} \)

Equals must override, not overload Object’s equals
• Must take in a parameter of type Object
• After checking instanceof, can cast argument to the right class
Equals and Subclassing
Okay, so are we done?

- Done:
  - Understanding the `equals` contract
  - Implementing `equals` correctly for `Duration`
    - Overriding
    - Satisfying the contract [for all types of arguments]
- Alas, matters can get worse for subclasses of `Duration`
  - No perfect solution, so understand the trade-offs…
Two subclasses

```java
class CountedDuration extends Duration {
    public static numCountedDurations = 0;
    public CountedDuration(int min, int sec) {
        super(min, sec);
        ++numCountedDurations;
    }
}

class NanoDuration extends Duration {
    private final int nano;
    public NanoDuration(int min, int sec, int nano) {
        super(min, sec);
        this.nano = nano;
    }
    public boolean equals(Object o) { ... }
    ...
}
```
CountedDuration is good

- CountedDuration does not override equals

- Will (implicitly) treat any CountedDuration like a Duration when checking equals

- Any combination of Duration and CountedDuration objects can be compared
  - Equal if same contents in min and sec fields
  - Works because o instanceof Duration is true when o is an instance of CountedDuration
Now NanoDuration [not so good!]

- If we don’t override `equals` in `NanoDuration`, then objects with different `nano` fields will be equal

- So using everything we have learned:

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

- But we have violated the `equals` contract
  - Hint: Compare a `Duration` and a `NanoDuration`
The symmetry bug

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

This is *not symmetric*!

```java
Duration d1 = new NanoDuration(5, 10, 15);
Duration d2 = new Duration(5, 10);
d1.equals(d2);  // false
```

```java
d2.equals(d1);  // true
```
Fixing symmetry

This version restores symmetry by using Duration’s equals if the argument is a Duration (and not a NanoDuration)

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

Alas, this still violates the equals contract

- Transitivity: $a.equals(b) \land b.equals(c) \Rightarrow a.equals(c)$
The transitivity bug

```
Duration d1 = new NanoDuration(1, 2, 3);
Duration d2 = new Duration(1, 2);
Duration d3 = new NanoDuration(1, 2, 4);
d1.equals(d2);   // true
d2.equals(d3);   // true
d1.equals(d3);   // false!
```
No great solution

- *Effective Java* says not to (re)override `equals` like this
  - Unless superclass is non-instantiable (e.g., abstract)
  - “Don’t do it” a non-solution given the equality we want for `NanoDuration` objects

- Two far-from-perfect approaches on next two slides:
  1. Don’t make `NanoDuration` a subclass of `Duration`
  2. Change `Duration`’s `equals` such that only `Duration` objects that are not (proper) subclasses of `Duration` are equal
Bad idea: the `getClass` trick

Different run-time class checking to satisfy the `equals` contract:

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

But now `Duration` objects never equal `CountedDuration` objects

- Subclasses do not “act like” instances of superclass because behavior of `equals` changes with subclasses
- Generally considered wrong to “break” subtyping like this
Composition

Choose composition over subclassing

– Often good advice: many programmers overuse (abuse) subclassing [see future lecture on proper subtyping]

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

NanoDuration and Duration now unrelated

– No presumption they can be compared to one another

Solves some problems, introduces others

– Can’t use NanoDurations where Durations are expected (not a subtype)

– No inheritance, so need explicit forwarding methods
Slight alternative

- Can avoid some method redefinition by having `Duration` and `NanoDuration` both extend a common abstract class
  - Or implement the same interface
  - Leave overriding `equals` to the two subclasses

- Keeps `NanoDuration` and `Duration` from being used “like each other”

- But requires advance planning or willingness to change `Duration` when you discover the need for `NanoDuration`
Summary: Equals and Subclassing

• Be careful when creating subclasses – `equals` needs to work!
• `NanoDuration` is not a proper Java subclass of `Duration` since we can’t get `equals` to work
  – More on the nuances of subclassing later!

• Unresolvable tension between
  – “What we want for equality”
  – “What we want for subtyping”
• This is one of the limitations of Java
Announcements
Announcements

This coming week is the craziest part of the quarter!
• Quiz 4 due tomorrow 10 pm
• HW4 due tomorrow 10 pm
• HW5 due next Thursday
  – Hardest hw in 331 and future hws build on it
• Section tomorrow!
  – important things you need to know for HW5
• Midterm review session Friday 3:30-5 in this room
• Midterm Monday 1:10-2:10 in this room
• Mid-quarter course evaluation Friday (during part of class)
  – Visitor: Jamal from the Center for Teaching and Learning
Equals and Collections
hashCode

Another method in Object:

public int hashCode()

“Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.HashMap.”

Contract (again essential for correct overriding):

- **Self-consistent:**
  
  o.hashCode() == o.hashCode()
  
  ...so long as o doesn’t change between the calls

- **Consistent with equality:**

  a.equals(b) ⇒ a.hashCode() == b.hashCode()
Think of it as a pre-filter

- If two objects are equal, they *must* have the same hash code
  - Up to implementers of `equals` and `hashCode` to satisfy this
  - If you override `equals`, you *must* override `hashCode`

- If two objects have the same hash code, they *may or may not* be equal
  - “Usually not” leads to better performance
  - `hashCode` in `Object` tries to (but may not) give every object a different hash code

- Hash codes are usually cheap[er] to compute, so check first if you “usually expect not equal” – a pre-filter
Asides

• Hash codes are used for hash tables
  – A common collection implementation
  – See CSE332
  – Libraries won’t work if your classes break relevant contracts

• Cheaper pre-filtering is a more general idea
  – Example: Are two large video files the exact same video?
    • Quick pre-filter: Are the files the same size?
Doing it

- So: we have to override `hashCode` in `Duration`
  - Must obey contract
  - Aim for non-equals objects usually having different results

- Correct but expect poor performance:
  ```java
  public int hashCode() { return 1; }
  ```

- Correct but expect better-but-still-possibly-poor performance:
  ```java
  public int hashCode() { return min; }
  ```

- Better:
  ```java
  public int hashCode() { return min ^ sec; }
  ```
Correctness depends on `equals`

Suppose we change the spec for `Duration`’s `equals`:

```java
// true if o and this represent same # 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;
}
```

Must update `hashCode` – why?

– This works:

```java
public int hashCode() {
    return 60*min+sec;
}
```
Equality, mutation, and time

If two objects are equal now, will they always be equal?
- In mathematics, “yes”
- In Java, “you choose”
- Object contract doesn't specify

For immutable objects:
- Abstract value never changes
- Equality should be forever (even if rep changes)

For mutable objects, either:
- Stick with reference equality
- “No” equality is not forever
  - Mutation changes abstract value, hence what-object-equals
Examples

StringBuffer is mutable and sticks with reference-equality:
StringBuffer s1 = new StringBuffer("hello");
StringBuffer s2 = new StringBuffer("hello");
s1.equals(s1); // true
s1.equals(s2); // false

By contrast:
Date d1 = new Date(0); // Jan 1, 1970 00:00:00 GMT
Date d2 = new Date(0);
d1.equals(d2); // true
d2.setTime(1);
d1.equals(d2); // false
Behavioral and observational equivalence

Two objects are “behaviorally equivalent” if there is no sequence of operations (excluding ==) that can distinguish them
   – they look the same forever
   – might live at different addresses

Two objects are “observationally equivalent” if there is no sequence of observer operations that can distinguish them
   – Excludes mutators (and ==)
   – they look the same now, but might look different later
Equality and mutation

Set class checks equality only upon insertion

Can therefore violate rep invariant of a Set by mutating after insertion

```java
Set<Date> s = new HashSet<Date> ();
Date d1 = new Date(0);
Date d2 = new Date(1000);
s.add(d1);
s.add(d2);
d2.setTime(0);
for (Date d : s) {
    // prints two of same date
    System.out.println(d);
}
```
Pitfalls of mutability and collections

From the spec of **Set**:

“**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.”

Same problem applies to **keys in maps**

Same problem applies to mutations that **change hash codes** when using **HashSet** or **HashMap**

(Libraries choose not to copy-in for performance and to preserve object identity)
Another container wrinkle: self-containment

equals and hashCode on containers are recursive:

class ArrayList<E> {
    public int hashCode() {
        int code = 1;
        for (Object o : list)
            code = 31*code + (o==null ? 0 : o.hashCode());
        return code;
    }
}

This causes an infinite loop:
List<Object> lst = new ArrayList<Object>();
lst.add(lst);
lst.hashCode();

From the List documentation: Note: While it is permissible for lists to contain themselves as elements, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a list.
Summary: Equals and Collections

• **Reference equality** (strongest)
  – a and b are the same iff they live at the same address

• **Behavioral equality** (weaker than Reference equality)
  – if a and b are the same now, they will be the same after any sequence of method calls (immutable objects)

• **Observational equality** (weaker than Behavioral equality)
  – if a and b are the same now, they might be different after mutator methods are called (mutable objects)

• Java’s `equals` has an elaborate specification, but does not require any of the above notions
  – Also requires consistency with `hashCode`
  – Concepts more general than Java

• Mutation and/or subtyping make things even less satisfying
  – Good reason not to overuse/misuse either