CSSE2002/7023

Semester 2, 2021

Programming in the Large

Week 6.2: More Miscellaneous Java

In this Session

- `abstract`
- Operators
- Short Circuit Evaluation
- StringBuilder/StringBuffer
- Copying
- `.equals()` and Overriding
- Hash Codes

abstract class

An abstract method has no implementation

```
public abstract void doStuff();
```

If a class contains any abstract methods, the class must also be declared abstract

```
public abstract class X {...}
```

Abstract classes can not be instantiated, but can be extended

```
public class Y extends X {

    @Override
    public void doStuff() {...}
}
```

`X v = new Y();` is legal

`X v = new X();` is not.

Rest of the Operators[1]

```
ex++   ex--
++ex   --ex   +ex   -ex   ~   !
*   /   %
+   -
<<   >>   >>>
<   >   <=   >=   instanceof
==   !=
&
^
|
&&
||
? :
=   +=   -=   /=   %=   &=   ^=   |=   <<=   >>=   >>>=
```

[1] From `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html`

Short Circuit Evaluation

Both the logical and (&&) operator and the logical or operator (||) are "short circuit" operators. That is, if we already know the answer, stop.

```
f(x) || g(x) || h(x)
```

If `f(x)` returns true, then `g` and `h` won't be called. If `f(x)` is false, then `g(x)` will be checked and so on.
This matters if the functions have "side-effects".

Use?

```
if ((args.length > 0) && args[0].equals("zzzz"))
```

StringBuilder/StringBuffer[2]

Strings are immutable, but this is not always convienient when creating strings.

```
StringBuilder sb = new StringBuilder("primes: 2");
for (int i = 3; i < 1000; i++) {
    if (isPrime(i)) {
        sb.append(',');
        sb.append(i);
    }
}

sb.insert(6, " under 1000");   // "primes under 1000:"
sb.setCharAt(0, 'P');          // Capitalise "Primes"

String s = sb.toString();      // Once we have the string
                               // the way we want it
```

[2] Older and slower but thread safe

Copying

At the shallowest level, `Object x = y` will make `x` and `y` reference the same object ... which *isn't* really a copy.

- If the objects are immutable, does that matter?

`Object` class has a protected `.clone()` method

- As it is protected, some work is required to be able to use it

See `CopyDemo.java` and `MessageHolder.java`

- Class needs to implement `Cloneable` interface
  - marker interface – does not define any methods
    - tells `Object.clone()` that copying is legal
- Typically, call `Object.clone()`
- Copy mutable objects that represent this class' state
- Consider the impact of different levels of "deep" copying

Cloning Complexities

`Object.clone()` will make a new object of the same type with *copies* of all the values.

Javadoc refers to the "intent" as being:

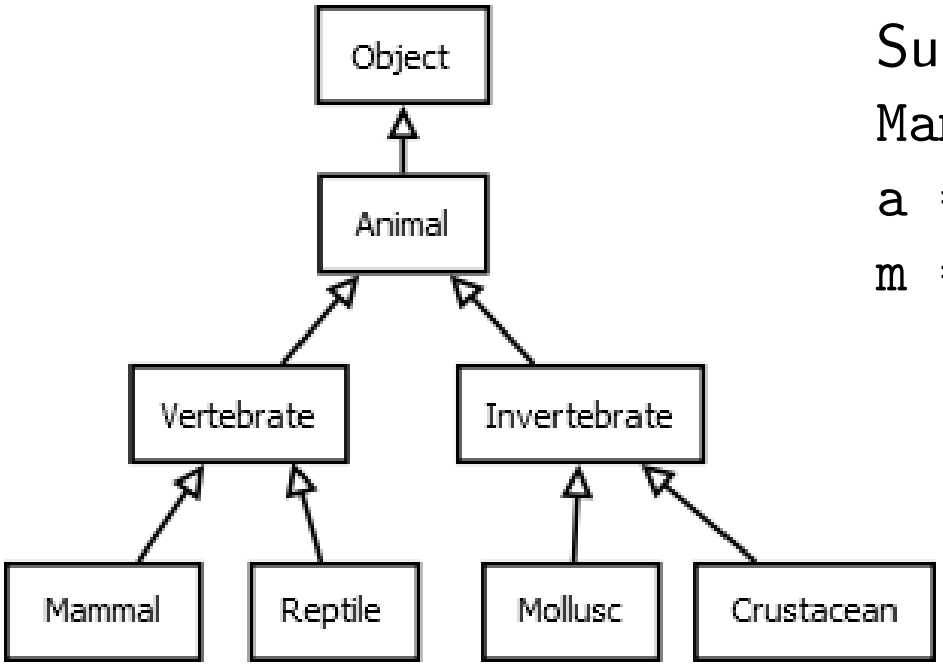
- `x.clone() != x`
- `x.clone().getClass() == x.getClass()`
- `x.clone().equals(x)  // typical, but optional`

Properties of `.equals()`

From the Javadoc for `Object.equals()`, (for `x, y, z != null`)

- `x.equals(x)` (reflexive)
- `x.equals(y)` $\Leftrightarrow$ `y.equals(x)` (symmetric)
- `x.equals(y) and y.equals(z)` $\Rightarrow$ `x.equals(z)` (transitive)
- `x.equals(y) should give a consistent result` (*deterministic*)
- `x.equals(null) == false`

Multiple Overrides of `.equals()`



Suppose both `Animal` and `Mammal` override `.equals()`

```
a = new Animal();
m = new Mammal();
```

Does `a.equals(m)` $\Leftrightarrow$ `m.equals(a)` hold?

- `a.equals(m)` uses the definition from `Animal`
- `m.equals(a)` uses the definition from `Mammal`

`.equals()` and `.hashCode()`

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

- if `.equals()` is overridden, `.hashCode()` should be as well

Which parts of state should be used for `.equals()` and `.hashCode()` calculations?

There are at least two schools of thought:

- these methods are about object identity
  - since an object's identity should not change, no mutable parts should be used (concerns about mutable objects as keys)
- they are for computing whether two objects currently have equivalent state
  - mutable parts should be included

The Java language does not take a position on this

What's a "Hash Code" Anyway?

Hash codes are computed values

Hash functions

- take an input
- perform a calculation
- return the resulting hash code
  - usually a numeric value
- e.g. `Object.hashCode()`
  - and overriding it in subclasses

Intent is that the hash code can be used to identify an object

- not necessarily *uniquely* identify an object
- `x.equals(y)` $\Rightarrow$ `x.hashCode() == y.hashCode()`
- **not** `x.hashCode() == y.hashCode()` $\Rightarrow$ `x.equals(y)`

Collisions between hash codes should be rare

What are "Hash Codes" Used For?

Fast matching of items

- hash function must be efficient

Common uses

- Searching for items
  - tables, databases, ...
- Cryptography
  - match passwords without needing to compare actual password values

Revisiting `.equals()` Calculation

Which parts of state should be used to calculate `.equals()`?

- `Comparable` and `Comparator`
  - both say their ordering is consistent with `.equals()`
  - if it is only object identity, their ordering is not flexible or useful

My stance is that `.equals()` is about state not identity

- an object's reference provides its identity

Revisiting `.hashCode()` Calculations

Which parts of state should be used to calculate `.hashCode()`?

- Hash codes are not necessarily unique
  - Can't be used for object identity

But, hash codes are used as "keys" when searching for objects

- Need to ensure we find the correct object

Need to consider:

- Does changing state make it a different object?
- Or, is it different internal state of the same object?
- Context in which objects are used will have different answers.
- That will determine whether only immutable parts of state should be used to calculate hash code or also mutable

My stance is that `.hashCode()` is about state not identity

- but which aspect of its state, is important in determining how to calculate it