CSSE2002/7023

Programming in the large

Week 6.1: More Java

Todays lecture

... will only go for one hour. We will cover:

- abstract methods
- Java operators and short circuiting
- StringBuilder
- Copying and Object.clone()
- Object.hashCode()
- Some information on Assignment 1

Does Java have aliasing for packages?

- Does Java have something like Python's `import X as Y`

Does Java have aliasing for packages?

- Does Java have something like Python's `import X as Y`
- No.

Does Java have aliasing for packages?

- Does Java have something like Python's `import X as Y`
- No.
- It has been previously requested as an enhancement `https://bugs.java.com/bugdatabase/view_bug.do?bug_id=4214789`

Does Java have aliasing for packages?

- Does Java have something like Python's `import X as Y`
- No.
- It has been previously requested as an enhancement `https://bugs.java.com/bugdatabase/view_bug.do?bug_id=4214789`
- But was not implemented.

Does Java have aliasing for packages?

- Does Java have something like Python's `import X as Y`
- No.
- It has been previously requested as an enhancement `https://bugs.java.com/bugdatabase/view_bug.do?bug_id=4214789`
- But was not implemented.
- From the comments "This is not an unreasonable request, though hardly essential." ".. it doesn't pass the bar of price/performance for a language change."

`abstract` methods

An abstract method is one with no body:

```
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 but `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 threadsafe

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?

The `Object` class has a protected `.clone()` method, but it's protected so we can't use it without some work.
See `CopyDemo.java` and `MessageHolder.java` (Note the impact of different levels of "deep" copying).

Cloning complexities

The `Object.clone()` will make a new object of the same type with *copies* of all the values.
The javadoc refers to the "intent" as being:

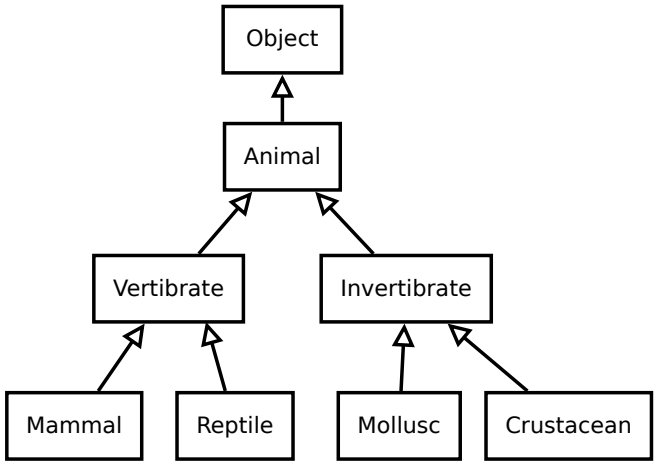
- `x.clone() != x`
- `x.clone().getClass() == x.getClass()`
- `x.clone().equals(x) // optional`

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`.

See `http://www.angelikalanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals-2.html`

What's "Hash..." anyway?

Let's talk about hashtables.

.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 seem to be at least two schools of thought here:

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

It is important to note that the java language does not take a position on this.

Assignment — some caveats to avoid

DO NOT IMPORT ANYTHING THAT DOES NOT BEGIN WITH `java.` IN YOUR SOLUTION

Assignment — some caveats to avoid

DO NOT IMPORT ANYTHING THAT DOES NOT BEGIN WITH `java.` IN YOUR SOLUTION

- `import java.util.List;`

Assignment — some caveats to avoid

DO NOT IMPORT ANYTHING THAT DOES NOT BEGIN WITH `java.` IN YOUR SOLUTION

- `import java.util.List;` ✔

Assignment — some caveats to avoid

DO NOT IMPORT ANYTHING THAT DOES NOT BEGIN WITH `java.` IN YOUR SOLUTION

- `import java.util.List;` ✔
- `import java.util.Map;`

Assignment — some caveats to avoid

DO NOT IMPORT ANYTHING THAT DOES NOT BEGIN WITH `java.` IN YOUR SOLUTION

- `import java.util.List;` ✔
- `import java.util.Map;` ✔

Assignment — some caveats to avoid

DO NOT IMPORT ANYTHING THAT DOES NOT BEGIN WITH `java.` IN YOUR SOLUTION

- `import java.util.List;` ✔
- `import java.util.Map;` ✔
- `import com.sun.istack.internal.NotNull;`

Assignment — some caveats to avoid

DO NOT IMPORT ANYTHING THAT DOES NOT BEGIN WITH `java.` IN YOUR SOLUTION

- `import java.util.List;` ✔
- `import java.util.Map;` ✔
- `import com.sun.istack.internal.NotNull;` ✘

Assignment — some caveats to avoid

DO NOT IMPORT ANYTHING THAT DOES NOT BEGIN WITH `java.` IN YOUR SOLUTION

- `import java.util.List;` ✔
- `import java.util.Map;` ✔
- `import com.sun.istack.internal.NotNull;` ✘
- `import org.apache.http.client.HttpClient;`

Assignment — some caveats to avoid

DO NOT IMPORT ANYTHING THAT DOES NOT BEGIN WITH `java.` IN YOUR SOLUTION

- `import java.util.List;` ✔
- `import java.util.Map;` ✔
- `import com.sun.istack.internal.NotNull;` ✘
- `import org.apache.http.client.HttpClient;` ✘

Assignment — some caveats to avoid

Additionally for your tests:

- `import org.junit.*` ✔
- `import static org.junit.Assert.*;` ✔
- `import org.junit.rules.Timeout;` ✔

Assignment — some caveats to avoid

DO NOT ADD ANY PUBLIC OR PROTECTED MEMBERS TO YOUR SOLUTION THAT ARE NOT IN THE SPEC

Assignment — some caveats to avoid

DO NOT ADD ANY PUBLIC OR PROTECTED MEMBERS TO YOUR SOLUTION THAT ARE NOT IN THE SPEC

- Task sheet says “You will implement precisely the public and protected items described in the supplied documentation (no extra members or classes).”

Assignment — some caveats to avoid

DO NOT ADD ANY PUBLIC OR PROTECTED MEMBERS TO YOUR SOLUTION THAT ARE NOT IN THE SPEC

- Task sheet says "You will implement precisely the public and protected items described in the supplied documentation (no extra members or classes)."
- We are testing classes in a leave-one-out style.

Assignment — some caveats to avoid

DO NOT ADD ANY PUBLIC OR PROTECTED MEMBERS TO YOUR SOLUTION THAT ARE NOT IN THE SPEC

- Task sheet says "You will implement precisely the public and protected items described in the supplied documentation (no extra members or classes)."
- We are testing classes in a leave-one-out style.
- Extra public members may result in your code failing to compile. You will then get 0 for that entire class.

Assignment — some caveats to avoid

DO NOT ADD ANY PUBLIC OR PROTECTED MEMBERS TO YOUR SOLUTION THAT ARE NOT IN THE SPEC

- Task sheet says "You will implement precisely the public and protected items described in the supplied documentation (no extra members or classes)."
- We are testing classes in a leave-one-out style.
- Extra public members may result in your code failing to compile. You will then get 0 for that entire class.
- `class Builder { public static int MY_USEFUL_CONSTANT = 4;}` ✘

Assignment — some caveats to avoid

DO NOT CHANGE METHOD SIGNATURES DESCRIBED IN THE SPEC

Assignment — some caveats to avoid

DO NOT CHANGE METHOD SIGNATURES DESCRIBED IN THE SPEC

- Do not allow a method to throw exceptions that are not specified.

Assignment — some caveats to avoid

DO NOT CHANGE METHOD SIGNATURES DESCRIBED IN THE SPEC

- Do not allow a method to throw exceptions that are not specified.
- Do not change the arguments or accessibility of a specified method.

Assignment 1

- Due Friday 31st, 11:59 PM
- Some additional checking utilties released today on blackboard:
  - TestMethods.java — a set of JUnit tests for ***method signatures*** (not functionality)
  - CheckZipAssignment1.java — a simple utility that checks that files included in the zip file match those on the task sheet