September 14th Lecture Notes

September 14, 2020

September 14: More Java

More Objects

Objects are Pointers

- All instances of classes are implemented as pointers to blocks of memory on the heap.
- So, when you make a copy of an object, it is a shallow copy of the pointer only
- If we want a deep copy, we need to do that explicitly
  - Never use `clone`, it is a one-level-deep copy which is almost never what we want
  - We will write our own deep-copy methods

Equality

- `==` checks for *physical equality*, i.e., that the pointers are the same
- `.equals` checks for *structural equality*, i.e., that the data itself is the same
  - We'd need to implement this ourselves
- Generally we should avoid `==`.

Strings

- `String`s are objects, not primatives
- `String`s can be created by writing a new literal
- They have methods because they are objects.

For example...

```
/* true */
"foo" == "foo";

/* true */
String x = "foo";
x == "foo";
```

```
/* true */
"foo" == "f" + "oo";

/* false */
"foo" == "f".concat("oo");
```

`null`

- `null` is Java's null pointer
- We can use `null` wherever an object is expected
- It's an error to invoke a method or access a field of `null`

Garbage Collection

- We don't need to `delete` things.
- The JVM will look around and find things that we are done with and recycle the memory for us

Inner Classes

- Classes can be nested

```
class A {
    static class B {
        void f() { System.out.println("B.f!\n"); }
    }
    void test() {
        b = new B();
        b.f();
    }
}
```

- The JVM doesn't actually understand inner classes, they are syntactic sugar
- `B` gets compiled to `A$B.class`

Linked List Example

- LinkedList.java
- Cell.java
  - The code in here was moved to be a private inner class of `LList`
- LList.java

Arrays

- Array syntax is similar to C

```
int[] a = new int[5];

a[0] = 42;
int x = a[0];
a[5] = 43     /* whoops, this is off the end of the array */
int y = a[-1] /* ArrayIndexOutOfBoundsException           */
```

- We can uses `a.length` to get the length of `a`
  - This is **not** a method call (note the lack of `()`)
  - It's its own special kind of thing
- We can also do multidimensional arrays

```
int[][] b = new int[2][3];
```

- *But* these aren't necessarily contiguous. There is an extra level of indirection here

The Standard Library

Exceptions

- Exceptions are instance of classes that are tailor-made to represent errors
- There are tons of exceptions in the standard library
- We throw an exception like this

```
throw new IndexOutOfBoundsException();
```

`LinkedList` and Packages

- Many things are in the `java.lang` package. These are included by default, kinda like a preamble
- The `LinkedList` class is instead in the `java.util` package.
- We can either use the full name, `java.util.LinkedList`

```
java.util.LinkedList ll = new java.util.LinkedList();
```

- Or we can import `java.util` into our working namespace

```
import java.util.*;
LinkedList ll = new LinkedList();
```

Interfaces

For example...

```
interface List {
    void insert(int x);
    int  size();
}

class ArrayList implements List {
```

```
    void insert(int x) { ... }
    int  size()        { ... }
}

class LinkedList implements List {
    void insert(int x) { ... }
    int  size()        { ... }
}

List l1 = new ArrayList();
List l2 = new LinkedList();

l1.insert(42);
l2.size();
```

Woah!

- An interface declares the publicly visible methods in a `class`
- A class that `implements` and interface must have at least the methods in the interface
  - You can have extra ones though
- If a class `implements` an interface, then we can use the class wherever the interface is expected
  - We call this *subtyping* or *subtype polymorphism*
- Interfaces must be specified explicitly
- Classes can implement more than one interface

We get cool information hiding here

```
interface I {
    void m1();
}

class C implements I {
    public void m1();
    public void m2();
}

I i = new C();
i.m1(); /* a-okay */
i.m2(); /* whoops this is an error */
```

The dynamic dispatch system only allows methods of the *interface* to be called, then looks up the method in the appropriate class.

Generics

Basic Generics

Some collection-type types can contain more than one kind of thing. This is similar to templates in C++ or generics in Rust.

```
import java.util.*;
LinkedList<Integer> ll = new LinkedList<>();
```

Generics with Interfaces

```
import java.util.*;
List<Integer> l = new LinkedList<>();
```

`instanceof` and type casting

here's how we'd typecast in Java

```
List l = ...;

if (l instanceof ArrayList) {
    al = (ArrayList) l;
    /* use al as an ArrayList */
} else if (l instanceof LinkedList) {
    ll = (LinkedList) l;
    /* use ll as a LinkedList */
}
```

But this is not ideomatic.

- We have to know all possible implemtors of List
- Kinda verbose
- Not great style
- Should probably avoid it.