Classes In Java (Not Comp 215)

Prof. Chris Jermaine
cmj4@cs.rice.edu

What Is a “Class”?

- Bundle of data + functionality
- Standard, classic idea in OO programming
- First made popular in Smalltalk-80
- Makes sense if you agree abstraction is important goal
  - — Say you have a list data structure
  - — Don’t want a user to understand inner workings
  - — Makes sense to provide Insert, Remove, GoToEnd, etc. with the data structure
- Note: nothing prevents you from doing this in a non-OO lang.
  - — People have been associating functions with C structs for decades
- But classes give you language-level support for this

Variables, References, Classes In Java

- A *variable* is always of one of the 8 + 1 primitive types
- A *reference* contains the address of an *object*
- An *object* is always an instance of a *class*
  - — The associated class is the object's type
  - — So both primitive types and classes are types
  - — But variables take one of the 8 + 1 primitive types
  - — And objects take a class type

Classes In Java

- Classes have data ("members", "member variables" in OO-speak)
  - — "Instance variables"™ is the "official" Java name
- Classes have functions/procedures ("methods" in OO-speak)

```
class Foo {
    private int a; // member variable
    public Foo () {a = 12;} // method
    public void AddOne () {a++;} // method
}
```

- In fact, all functions in Java must reside in a class
- This militancy leads to some weirdness
  - — Because Java is not so militant in other ways (allows arrays, primitive types)
  - — So we get things like the weird Math functions, which are attached to a class
  - — Where's the data?

Constructors in Java

- "Constructor" is a standard OO idea
- Function that is called when an object is created via new
  - — No return type needed or wanted
  - — Java gives a default, zero-arg one if you don't
  - — Ex:

```
class Foo {
    private int a; // member variable
    public Foo () {a = 12;} // method
    public void AddOne () {a++;} // method
}
...
Foo temp = new Foo ();  // this sets temp.a to 12
```

- Useful because allows you to write code you know will have to be run on every instance of that class, now and forever

Access Control In Java Classes

- All members/methods have a certain “privacy” level
- One of:
  - — public
  - — protected
  - — <nothing>
  - — private
- Ex:

```
class Foo {
    private int a;
    public Foo () {a = 12;}
    public void AddOne () {a++;}
}
```

- Will define these now...

But First a Foray Into Packages

- A “package” in Java is (meant to be) a set of classes working towards a single goal
  - — A geometry package might have point, line, circle, square, etc.
- When you do “import java.lang.Math” you are importing the class Math from the package java.lang
- Can also say “import java.util.*” but not so good an idea
- Generally, everything shipped with JVM in “java.something”
- Use of packages important to keep compilation time short, prevent naming conflicts, have interfaces stretching accross many classes
- We’ll not organize code into packages
- ‘Nuff said (but you should at least understand this much)

Public

- A public method/member can be called by anyone

Protected

- A protected method can be called (and a protected member accessed by)...
  - — The class
  - — Any subclass
  - — Any other class in the package
- But not by the outside world

<Nothing>

- A method with no specified access control can be called (and a similar member accessed by)...
  - — The class
  - — Any other class in the package
- But not by the outside world or subclasses of the class

Private

- Private means only the class can look at it or use it directly

Private

- Private means only the class can look at it or use it directly

If you **EVER** have a non-private member variable...

Private

- Private means only the class can look at it or use it directly

If you **EVER** have a non-private member variable...

- There's **NEVER** a good reason to do this
  - — Even if force it to be immutable, are locking yourself into an implementation

Classes Can Be Public or <Nothing>

- Ex:

```
class Foo {
    private int a;
    public Foo () {a = 12;}
    public void AddOne () {a++;}
}
```

- Means that outside this package, Foo is invisible

Static Members and Methods

- A static member is one shared by all objects that are instances of a given class

```
class Foo {
    private static int a; // static member variable
    public Foo () {a = 12;}
    public void addOne () {a++;}
    public void print () {System.out.format (“%d\n”, a);}
}
...
Foo foo1 = new Foo ();
foo1.AddOne ();
foo1.Print ();
Foo foo2 = new Foo ();
foo1.Print ();
```

- This will (surprisingly?) output a 13 followed by a 12

Static Members and Methods (cont)

- Can also declare a static method
- Such a method can only access static members
- Rixner always says (paraphrase) “no one should use ‘em”
- I disagree, but Scott has a key point
  - — People wayyyyy over-use static
  - — Why? Two main reasons.
  - — (1) Mis-guided attempt at performance tuning
  - — (2) Too lazy to put data into a class:

```
double x = 27;
double foo = Math.sqrt (x); // bad! But way it works in Java
double bar = x.sqrt (); // good! attach sqrt to Number
```

- We’ll generally never have a reason to use ‘em (other than main)

The Final Keyword

- Lots of things can be labeled final
- For this lecture, we'll consider final members
  - — Means value must be set at declaration
  - — Or by the time constructor ends
  - — Can never change after that

```
class Foo {
    private final int a; // final member variable
    public Foo () {a = 12;}
    public void addOne () {a++;}
    public void print () {System.out.format ("%d\n", a);}
}
```

- This won't compile because we have the addOne method

Abusing the Final Keyword

- Final is very useful for avoiding bugs
  - — If you mean to never modify a in last example, the final declaration means you can't screw up and write addOne
- But many people use "final" as part of the interface
  - — As a "safe" way to expose public member variables

```
class Foo {
    public final int a; // final member variable
    ...
}
...
if (myFoo.a > 12)
...
```

  - — Don't do it! Wrap it up in a function
  - — Otherwise, you make it part of interface (doesn't anyone understand abstraction?)