Java Introduction

This is the outline for the Java section going over all the language basics in one session.

Java Doc Links

http://java.sun.com/docs/
  lots of docs about Java.
http://java.sun.com/docs/books/tutorial/index.html
  Java tutorial -- includes separate tracks for many topics.
http://java.sun.com/products/jdk/1.2/docs/api/index.html
  The "API" section is a reference for the many built-inthe library classes (String, ArrayList, ...)
http://www.bruceeckel.com/
  Bruce Eckel's Book, Thinking in Java, in a free online form
http://www.afu.com/javafaq.html
  The comp.lang.java FAQ -- maintained by Peter van der Linden -- great!
http://www.javaworld.com/
  lots of Java articles

Java Qualities

Compiled to bytecode
Interpreted at runtime -- potentially slow
  Interpreters are slower than compiled code, but they enable true portability
  Modern systems are not true interpreters -- they use some form of Just-in-time-Compile to build native code at runtime. Sun's "hot spot" system does this in a sophisticated way at runtime.
  Realistically, not as fast as C, but getting closer. Java's dynamic features: safe pointers, garbage collection, etc. extract a penalty vs. C code.
Portable
  Runs the same everywhere without even a re-compile. Very true for the core language. Less true for first GUI library, AWT, but the new GUI library Swing has fixed this.
Robust/Safe
  Hard to crash, hard to write viruses
Structured -- typed
  Java has a formal type system that must be obeyed at compile-time and run-time. This is helpful for larger projects, where the structure helps keep the various parts consistent. Contrast to Perl, which as more of a quick-n-dirty feel.
C/C++ Syntax
  Fool the C/C++ programmers into thinking this is not much of a change
OOP
  The best way to structure code

Official "Blessed" libraries for common problems
- Common classes: String, Collection, Date ...
- GUI libs (AWT, Swing)

Threaded
- Supports multiple threads of control built in to the language

Dynamic
- Types/behaviors are determined at run-time (most flexible, at the cost of some efficiency).
- Automatic garbage collector takes care of memory management.

Point: Somewhat slow + portable + programmer efficiency features

Programmer Efficiency

Faster Development
- Building an application in Java takes about 30% less time than in C or C++

Libraries
- Code re-use at last -- String, ArrayList, ... (C++ can also do this to an extent)

Memory errors
- Memory errors largely disappear because of the safe pointers and garbage collector. I suspect the lack of memory errors accounts for much of the increased programmer productivity.

Java Niche

CPU time vs. Programmer time tradeoff

Portability
- Send code from one place to another
- Avoid vendor lock-in
- Microsoft hates this feature (in any domain, it's natural for the largest vendor to dislike portability, while the small vendors and the customers like portability).

Applets
- Java code, runs in the browser in a "sandbox"
- Did not work that well
- Not well implemented the first time -- AWT not well done, in-the-browser concept is fragile
- Microsoft not helping

Custom server side applications
- A good match for Java qualities -- robust/safe, programmer efficient, portable (e.g. develop on NT, deploy on Solaris, Linux, ...)

Procedural vs. OOP

Nouns and Verbs

Nouns -- data
Verbs -- operations

Procedural Structure

C/Pascal/etc. ...
Verb oriented
  decomposition around the verbs -- dividing the big operation into a series of smaller and smaller operations.
Nouns/Verb structure is not formal
  The programmer can group the verbs and nouns together (ADTs), but it's just a convention and the compiler does not especially help out.

OOP Structure..

Objects

State
  Stores state, like a regular variable
Class
  Belongs to a class which defines its state and behavior.
  An object always remembers its class (in Java).
Anthropomorphic -- self-contained
  Procedural variables are passive -- they just sit. A procedure is called and it changes the variable.
  Objects are anthropomorphic-- the object has some state, and the object acts to change its own state.

Class

Exists once -- there is one copy of the class in memory.
Definitions for its objects
Storage
  Define the storage that objects of this class will have.
  "instance variables" -- every object of the class will have its own copy of each instance variable (aka ivar).
Behavior
  Define the behaviors that objects of this class will be able to execute (methods).

Message / Receiver

Sent to an object -- Request -- "getUnits()"

a.getUnits()

Receiver

The object receiving the message.

obj.units = 15; NO

The receiver should operate on its state, not the client

obj.setUnits(15) YES

Send a message to the obj, maps to a method (below), that method is code that actually changes the receiver state.

Method (code)

A method, such as setUnits() { .... } defines actual code -- it's like a function in C. The method is defined in a class -- it is the code for the objects of that class.

Message -> Method resolution

Suppose a message is sent to an object --- a.getUnits();

1. The receiver is of some class -- suppose the object a is of the Student class
2. Look in that class for a matching method (code)
3. Execute that code "against" the receiver -- using its ivars, etc.

In Java this is "dynamic" -- the process uses the true, run-time class of the receiver.

OOP Design - Anthropomorphic

1. Objects responsible for their own state
2. Objects can send messages to each other -- requests

OOP Part 1 -- Encapsulation

Objects "protect" their own state from direct access by other objects. Other objects can send requests, but only the receiver actually changes its own state. This allows more reliable software -- once a class is correct and debugged, putting it in a new context should not create new bugs in the class.

Abstraction vs. Implementation

This is the old Abstract Data Type (ADT) style of separating the abstraction from the implementation, but re-cast as messages (abstraction) vs. methods (implementation)

OOP Technique 1 --Receiver Relative Coding

Think about the objects that make up an application

Think about the behaviors or capabilities those objects should have

Endow the objects with those abilities as methods

Co-operation

Objects sent each other messages to co-operate

But each method operates on its own receiver

Tidy style

Having each object operate on its own state turns out to be a tidy way to organize things

Java Client Side

Allocate objects with "new" -- calls constructor
Objects are always accessed through pointers -- shallow, pointer semantics
Send messages -- methods execute against the receiver
Can access public, but not private/protected from client side

Client Side Code

```
// Make two students
Student a = new Student(15);
Student b = new Student(12);

// They respond to getUnits() and getStress()
System.out.println("a units:" + a.getUnits() +
   " stress:" + a.getStress());

System.out.println("b units:" + b.getUnits() +
   " stress:" + b.getStress());


//a.setUnits(a.getUnits() - 4);  // drop that class!
a.dropAClass(4);  // Now there's a method that just does it


System.out.println("a units:" + a.getUnits() +
   " stress:" + a.getStress());
```

Constructor (ctor)

Initializes new objects
Bug control
- Make it easier for the client to do the right thing since objects are always put into an initialized state when created.

Every ivar goes in Ctor
- Every time you add an instance variable to a class, go add the line to the ctor that inits that variable.

Method

Code stored in class
This code will execute against instances of the class
Message-Method Lookup
- Message sent to a receiver
- Receiver looks in its class for matching method
- That method executes against the receiver

Receiver Relative (Method, Ctor)

The code runs "on" or "against" the receiving object
Any ivar read/write operations happen to the receiver

Method code is written with a "receiver relative" style
e.g. "units" instance variables automatically that of the receiver
Self message send
    "setUnits(units - drop);" -- easy to send a message keeping the same receiver

"this" -- receiver

"this" in a method
    "this" is a pointer to the receiver
    Don't write "this.units", write: "units"
    Don't write "this.setUnits(5)", write "setUnits(5);"
ivar vs. local var
    Usually, just refer to the ivar by name directly. Sometimes you have a local var with the same name as the ivar, in which case the expression this.ivar refers to the ivar. Having a local var with the same name as an ivar is a stylistically questionable, but it can be handy sometimes. Some people prefer to give ivars a name always starting with "m" -- mUnits, etc.
Receiver/Noun Style
    You think a little differently about your code -- code is grouped around the noun it operates on

"private"

Implementation visibility
    Essentially, only code that is implementing the class can access the method or ivar...
Ivars
Methods
Ctors
"Sibling Access"
    This does not protect against one object in the class access the state of **another** object in the class. Such access is slightly less desirable OOP style, but the private keyword does not guard against it.

"public"

Visible to all
"Official" class interface
Not removed (vs. deprecated)
    public features will not be removed in a future rev
    Other classes can depend on these features
    Sun deprecates some public features, so new code won't be written with them, but it very rarely removes a formerly public feature
    private things can be removed from an implementation at will

"protected"

Similar to "private" but allows access to subclasses

Student.java Code Example

```
// Student.java
/*
 Demonstrates the most basic features of a class.
 Language points (not the usual sort of comments I would
 put in production code) are marked with "NOTE".
*/

/*
 A student is defined by their current number of units.
 There are standard get/set accessors for units.

 The student responds to getStress() to report
 their current stress level which is a function
 of their units.

 NOTE A well documented class should include an introductory
 comment like this. Don't get into all the details -- just
 introduce the landscape.
*/
public class Student extends Object {
   // NOTE this is an "instance variable" named "units"
   // Every Student object will have its own units state
   // "protected" and "private" mean clients do not get access
   protected int units;


   /* NOTE
    "public static final" declares a public readable constant that
    is associated with the class -- it's full name is Student.MAX_UNITS.
    It's a convention to put constants like that in upper case.
   */
   public static final int MAX_UNITS = 20;


   // Constructor for a new student
   public Student(int initUnits) {
      units = initUnits;
      // NOTE this is example of "Receiver Relative" coding --
      // "units" refers to the ivar of the reciever.
      // Code is written relative to an implicitly present receiver.
   }


   // Standard accessors for units
   public int getUnits() {
      return(units);
   }

   public void setUnits(int units) {
      if ((units < 0) || (units > MAX_UNITS)) {
         return;
         // Could use a number of strategies here: throw an
```

```
        // exception, print to stderr, return false
     }
     this.units = units;
     // NOTE: "this" trick to allow param and ivar to use same name
  }


  /*
   Stress is units *10.

   NOTE another example of "Receiver Relative" coding
  */
  public int getStress() {
     return(units*10);
  }


  /*
   Try to drop the given number of units.
   Does not drop if would go below 9 units.
   Returns true if the drop succeeds,
  */
  public boolean dropAClass(int drop) {
     if (units-drop >= 9) {
        setUnits(units - drop);    // NOTE send self a message
        return(true);
     }
     return(false);
  }

  /*
   Here's a static test function with some simple
   client-of-Student code.
   NOTE Invoking the "Student" class from the command line runs this.
   It's handy to put test/demo/sample client code in the main() of a class.
  */
  public static void main(String[] args) {
     // Make two students
     Student a = new Student(15);
     Student b = new Student(12);

     // They respond to getUnits() and getStress()
     System.out.println("a units:" + a.getUnits() +
        " stress:" + a.getStress());

     System.out.println("b units:" + b.getUnits() +
        " stress:" + b.getStress());


     //a.setUnits(a.getUnits() - 4);  // drop that class!
     a.dropAClass(4);  // Now there's a method that just does it


     System.out.println("a units:" + a.getUnits() +
        " stress:" + a.getStress());


     // Now "b" points to the same object as "a"
```

```
        b = a;
        b.setUnits(10);


        // So the "a" units have been changed
        System.out.println("a units:" + a.getUnits() +
           " stress:" + a.getStress());

        /*
         OUTPUT...
           a units:15 stress:150
           b units:12 stress:120
           a units:11 stress:110
           a units:10 stress:100
        */
    }
}

/*
 Things to notice...

 -Demonstrates the Object-lifecycle -- clients create the object (must go
 through constructor), then send it messages. Hard for the client to mess
 up the state of the object. Note how setUnits() can maintain the internal
 correctness of the object.

 -The implementation code can refer to instance variables like "units"
 by name. It automatically binds to the ivar of the receiver.

 -"units" is declared protected. Thereore, a client cannot write something like
 "a.units++;". The client must go through public messages like setUnits().
 This promotes a less fragile design. The client may access things declared
 "public".

 -State vs. Computation -- notice that the client can't really tell if stress is
 stored or computed. It just appears to be a property that Students have. Whether
 it is stored or computed is just a detail. This is a nice separation between
 the abstraction exposed by client and how it is actually implemented.
*/
```

Compiling and Running

1. % javac Student.java
   compiles Student.java and generates .class file
2. % java Student
   runs the main() method in the class named Student. In Java 2, the java runtime looks in the current directory for class definitions

Can gather .class files in a .jar file -- e.g. Student.jar -- then double clicking the .jar just runs the program. See the "Jar file" track in the tutorial.

OOP Part 2 — Inheritance

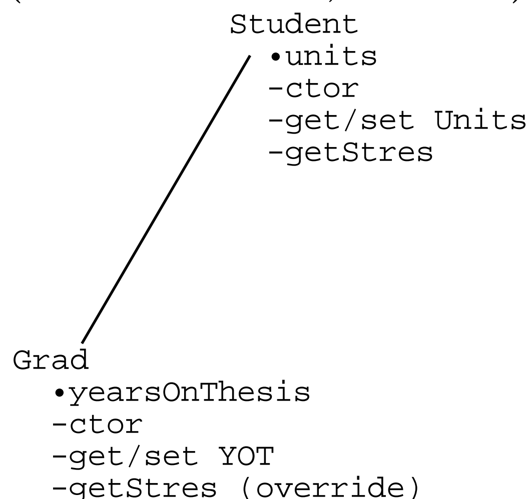
Arrange several related classes in a way to avoid duplication / promote code re-use.
OOP Hierarchy
Superclass / Subclass
Inheritance
Overriding

Student Inheritance

Student defined by int units
Grad is everything that a Student is + the idea of yearsOnThesis
"isa" relationship with its superclass -- Grad isa Student
Subclass has **all** the properties of its superclass + a few
Grad overrides getStress() with a specialized version
Grad (subclass) has more properties / is more constrained / more specific
Student (superclass) has fewer properties / is less constrained./ more general

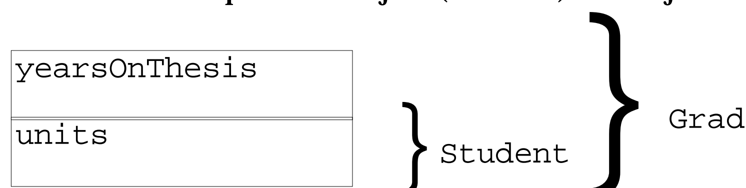
Student/Grad Design Diagram

The following is an excellent sort of diagram to make early in the design to think about the division of responsibility between a Superclass and its Subclass.
('•' = instance variable, '-' = method)



Student/Grad Memory Layout

The ivars of the subclass are layered on top of the ivars of the superclass
Result: have a pointer to the base of the object - can treat subclass object (Grad) as if it were a superclass object (Student) and it just works.



Never Forget Class

In Java, no matter what code is being executed, the receiver is the same receiver (even if the code is in a different class) and the receiver never forgets its class.
EG even getUnits() (Student class) executing on a Grad, remembers that the receiver is a Grad

Semantics of "Student s;"

NO: "s points to a Student object"
YES: "s points to an object that is at least a Student"
YES: "s points to an object that responds to all the messages Students respond to"
YES: "s points to a Student, or a subclass of Student"

Substitution Example

Subclass can be used in a context which call for the superclass
This works because of the ISA property -- Grad ISA Student

```
Student s = new Student(10);
Grad g = new Grad(10);
s.getStress();  // ok -- goes to Student.getStress
g.getStress();          // ok -- goes to Grad.getStress -- OVERRIDING
g.getUnits();  // ok -- goes to Student.getUnits -- INHERITANCE
```

Insomnia Example

```
static boolean insomnia(Student s) {
  return(s.getStress() > 100);
}
```

1. Can pass it a Student or Grad instance to operate on
2. The s.getStress() lines still does the right thing

Insomnia 2

Suppose insomnia is implemented up in the Student class...

```
...
public boolean insomnia() {
  return(getStress() > 100);
// Pops DOWN to Grad.getStress()
// if the receiver is a Grad
}
..
client code...
Student s = new Student(...);
Grad g = new Grad(....);
s.insomnia();               // does the right thing
g.insomnia();               // does the right thing
```

g.insomnia() Series

Where does the code flow go....
1. Student.insomnia()
2. Grad.getStress() // pop-down
3. Student.getStress() // the super.getStress call in Grad.getStress

Up-Down Rule

The receiver knows its class
The flow of control jumps around different classes
No matter where the code is executing, the receiver knows its class and does message->method mapping correctly

Grad.java

```
/*
 Grad is a subclass of Student.
 Grad adds the state of yearsOnThesis.

 Grad overrides getStress() to provide a Grad specific version.
*/
public class Grad extends Student {

   private int yearsOnThesis;


   public Grad(int units, int yearsOnThesis) {
      // NOTE "super" must be first if used --
      // chains up to the superclass constructor
      super(units);

      this.yearsOnThesis = yearsOnThesis;
   }


   /*
    Grad stress is based on twice the Student stress
    plus an additional factor for the yearsOnThesis.

    NOTE: avoid code repetition between subclass/superclass
    at all costs --that's why we use Student.getStress()
    for the core of our computation.
   */
   public int getStress() {
      // NOTE "super" still invokes message/method resolution
      // but it starts the search one class higher up
      // (there is no super.super)
      int student = super.getStress();

      return(student*2 + yearsOnThesis);
   }

   // Standard accessors
   public void setYearsOnThesis(int yearsOnThesis) {
      this.yearsOnThesis = yearsOnThesis;
   }
```

```
   public int getYearsOnThesis() {
      return(yearsOnThesis);
   }


   public static void main(String[] args) {
      Student s = new Student(10);
      Grad g = new Grad(15, 2);
      Student x = null;

      System.out.println("s " + s.getStress());
      System.out.println("g " + g.getStress());

      g.dropAClass(3);  // drop that class!
      // Note how g responds to everthing s responds to

      System.out.println("g " + g.getStress());

      /*
       OUTPUT...
      s 100
      g 302
      g 242
      */



      // Substitution rule -- subclass may play the role of superclass
      x = g;   // ok

      // At runtime, this goes to Grad.getUnits()
      // Point: message/method resolution uses the RT class of the receiver,
      // not the CT class in the source code.
      // This is essentially the objects-know-their-class rule at work.
      x.getUnits();


      //g = x; // NO -- CT error -- substitution does not work this direction
      g.setYearsOnThesis(2);  // how could this work if the above were allowed?
   }

}
```

```
/*
 Things to notice...

 -The ctor takes both Student and Grad state -- the Student state is passed up
 to the Student ctor by the first "super" line in the Grad ctor.

 -getStress() is a classic override. Note that it does not _repeat_ the code
  from Student.getStress(). It calls it using super, and fixes the result.
  The whole point of inheritance is to avoid code repetition.

 -Grad responds to every message that a Student responds to -- either
 a) inherited such as getUnits()
 b) overriden such as getStress()

 -Grad also esponds to things that Students do not,
 such as getYearsOnThesis().
*/
```

Inheritance / Notification Style

Suppose there is a Car class with go(), stop(), and turn() methods
You want to create your won car, but that turns differently...
    Subclass off Car
    Override the turn() method with your own definition
The existing methods go(), stop(), etc. continue to work
A turn() message will pop down, and use your definition.

Notification Style Results

Use this style to benefit from already written code, but put in a little segment of your own code.
E.g., this is how Servlets work -- use a key override to insert your own code.

Misc. Java features...

static

Exists once for the whole class

Not associated with an instance, associated with the whole class. Exists exactly once, instead of once for each instance.

Static methods

A static method is not associated with a particular instance. There is no receiver. Static methods operate more like traditional functions (no receiver), however they are still in the class namespace: e.g.. Student.main() instead of just main().

Static variable

A static variable is like a global variable for that class. The variable exists just once as a global, instead of once for each instance. It is in the namespace of the class, such as Student.someStaticVariable.

Static variables are a little rare.

public static void main(String[] args);

The special method begins at a method like this

Student.foo(), NOT a.foo();

a.foo() actually compiles, but it discards a as a receiver and translates to the same thing as Student.foo() (there is no receiver object for a static method). The a.foo() syntax is misleading, since it makes it look like a regular message send.

Array

int[] a; -- an array of ints

int a[]; -- alternate syntax for C refugees

Heap allocated

int[] a; -- allocs the pointer, not the pointee (the array itself)

Implementations currently zero the array, but that may go away someday (auto-zero is a poor design IMHO)

a = new int[100];

Run time allocate the array, just like an object

100 == a.length

Arrays know their length -- cool!

NOT a.length()

a[0] = new Student(10); // NO -- CT Err

At CT, arrays know their type (int in this case) and trap errors such as above

The other Java collections WILL NOT have this CT type system error catching (d'oh!), although it is rumored that CT types are being added for Java 1.4 or Java .15.

Student b[] = new Student[100];

Allocate 100 Student pointers

Does not allocate any Student objects -- that's a separate pass

String

String (and char) use 2-byte unicode characters.
"Hello World!\n" -- String consts like C
String object is "immutable"
- It never changes once created
- Handy way to finesse memory sharing and multi-threading issues

String + String
- String a = "foo";
- a + " bar"
- + concats strings together -- creates a new String based on the other two

See the docs
- Look in the String class docs for the many messages it responds to
- length() -- number of chars

StringBuffer
- Similar to String, but can change the chars over time. More efficient to change one StringBuffer over time, than to create 20 slightly different String objects over time.

```
StringBuffer buff = new StringBuffer();
buff.add(someString);          // efficient append
....
buff.toString();          // make a String once done with appending
```