CS 126 Lecture S1: Introduction to Java

"Systems" Part of the Class

- What is the "system"?
  - Loosely defined as anything that's not your application
- Why should you care?
  - Learn more about the pieces that constitute a large part of your daily computing life: compilers, operating systems, ...
  - The boundaries between the different pieces are becoming increasingly fussy in this age, so an "application" can have elements of compilers and OS built in.
  - For example, a browser that has a Java Virtual Machine and a Just-In-Time compiler built in is simultaneously an application, a compliler, and to some extent, an OS!
  - Synthesis of much stuff that we learned about programming, hardware, and theory

Roadmap

- S1-S2: Java
  - More like a continuation of the programming part of the class
  - So, really, it's an excuse to teach you some Java :-)
  - But, there is a profound connection between Java and OS, as we shall see: fundamental question of how to structure a system in terms of issues such as protection. So Java is far more than just another programming language
- S3: Compilers
  - A good meeting place of three previous pieces: programming, hardware, and theory
- S4: Operating systems
  - The missing link between hardware and applications

Outline

- **Introduction**
  - **History**
  - **Java vs. C**
  - **How to learn**
- The basics
- Object-oriented niceties
- Conclusions

History

- Bill Joy and Sun
  - BSD god at Berkeley
  - Founding of Sun (early 80s)
  - "The network is the computer" (a little ahead of its time)
  - Missed the boat on PC revolution
  - Sun Aspen Smallworks (1990)
- James Gosling
  - Early fame as the author of "Gosling Emacs" (killed by GNU)
  - Then onto Sun's "NeWS" window system (killed by X)
  - Lesson 1: keeping things proprietary is kiss of death
  - Lesson 2: power of integrating three things:
    + an expressive language
    + network-awareness, and
    + a GUI (graphical user interface)

History (cont.)

- Joy and Gosling joined forces, FirstPerson, Inc. (1992)
  - Targeting consumer electronics: PDAs, appliances, phones, all with cheap infra-red kind of networks
- Need a language that's small, robust, safe, secure, wired
  - Started working on **C++−−**
  - Soon gave up hope, decided to start from scratch
- Again, a little ahead of its time
  - PDAs died with the demise of Apple Newton
  - Switched to interactive TV (ITV)
  - The resulting language was called "Oak"
  - Then ITV died too
- The net exploded in 1993
  - Oak became Java!

History (cont.)

- Many success stories in CS
  - Very much like what we said about Unix
  - Not a technological breakthrough
  - All of the features of Java were present in earlier research systems
  - The "genius" lies in the good taste of assembling a small and elegant set of powerful primitives that fit together well and tossing everything else!
- Luck helps a lot too!

Java vs. C

- Comparison inevitable, but...
- "Java is best taught to people not contaminated by C"
  - Important to "think Java", instead of "translating C to Java"
- Similarities between C and Java are skin-deep
  - Syntatic sugar to make it easy to swallow
  - Terseness is good
  - Underlying philosophies are like day and night
- Theme of this class: levels of abstraction
  - C exposes the raw machine
  - Java hides a lot of it

Java vs. C (cont.)

- Bad things you **can** do in C that you **can't** do in Java
  - Shoot yourself in the foot (safety)
  - Others shoot you in the foot (security)
  - Ignoring wounds (error handling)
- Dangerous things you **have to** do in C that you **don't** in Java
  - Handling ammo (memory management: malloc/free)
- Good things that you **can** do in C but you don't; Java **makes** you
  - Good hunting practices (objected-oriented methodology)
- Good things that you **can't** do in C but you **can** now
  - Kills with a single bullet (portability)
- An interesting lesson in abstraction (and politics?): making things better by "taking away" power
- [We will revisit these differences after we learn more about Java]

How to Learn

- The best language to learn on-line, which is the best way to learn Java!
  - http://www.javasoft.com
  - http://java.sun.com/docs/books/tutorial/index.html
  - http://java.sun.com/products/jdk/1.1/docs/api/packages.html
  - http://java.sun.com/products/jdk/1.2/docs/api/index.html
- Start with existing code, read code, read docs
- Experiment by making small changes and adding functionality progressively
- My personal opinion: learning a second programming language in a class is a waste of time :-)
- So, it's really just a highlight

Outline

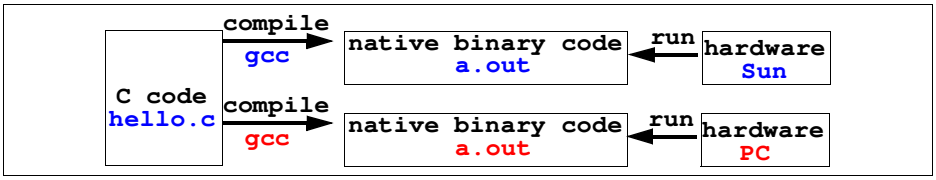
- ~~Introduction~~
- **The basics**
  - **First Java program and tools of trade**
  - **Classes, methods, and objects**
  - **Arrays**
  - **"Pointers"**
  - **Libraries**
- Object-oriented niceties
- Conclusions

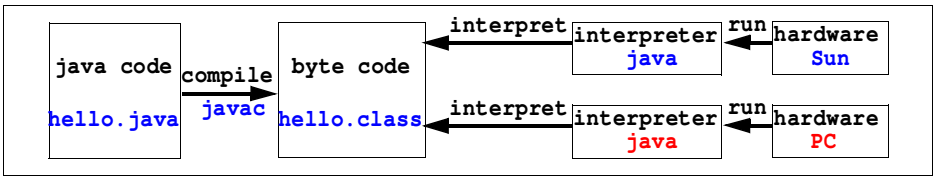
Your First Java Program

```
mocha:tmp% cat > hello.java

class hello {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}

mocha:tmp% javac hello.java

mocha:tmp% ls hello.*
hello.class  hello.java

mocha:tmp% java hello
Hello World!
```

- Source file: "**`hello.java`**"
- Java compiler: **`javac`**
- Byte code: "**`hello.class`**"
- Java interpreter: **`java`**
- Can install JDK on any machine, including your PC
- Other tools in JDK: **`jdb`**, **`javadoc`**

Compiling vs. Interpreting





- Interpreter: a level of abstraction: the "virtual machine"
- The advantage of interpreting is beyond portability
- A convenient place to exercise all sorts of control
- Disadvantage: slower

Classes, Methods, and Objects

```
public class MyStack {
  Object[] items;
  int n;

  public MyStack() {
    items = new Object[1000];
    n = 0;
  }
  public void push(Object item) {
    items[n++] = item;
  }
  public Object pop() {
    return items[--n];
  }
  public boolean empty() {
    return n == 0;
  }
```

MyStack.java

```
import MyStack;

class StackTest {
  public static void
    main(String[] args) {

    MyStack s = new MyStack();
    s.push("first");
    s.push("second");
    s.push("third");
    while (!s.empty())
      System.out.println
        (s.pop());
  }
}
```

StackTest.java

- (Don't need to understand everything in this code, yet)
- A program is a sequence of classes (no .h files!)
- A class is like a struct, one difference: methods: operations that act on the data that makes up the class
- A method is like a function. (Note how they are invoked.)
- An object to a class in Java is like a variable to a type in C

More Thoughts/Details on This Example

```
public class MyStack {
  Object[] items;
  int n;

  public MyStack() {
    items = new Object[1000];
    n = 0;
  }
  public void push(Object item) {
    items[n++] = item;
  }
  public Object pop() {
    return items[--n];
  }
  public boolean empty() {
    return n == 0;
  }
```

MyStack.java

```
import MyStack;

class StackTest {
  public static void
    main(String[] args) {

    MyStack s = new MyStack();
    s.push("first");
    s.push("second");
    s.push("third");
    while (!s.empty())
      System.out.println
        (s.pop());
  }
}
```

StackTest.java

- Other than the primitives such as int, char, boolean, all variables are objects
- Concepts of object declaration, allocation, and a constructor
- How to design a Java program: think objects!
  - What objects do I break the problem into?
  - What operations do they allow?
  - How do I implement them using even smaller objects?

Arrays (still same example)

```
public class MyStack {
  Object[] items;
  int n;

  public MyStack() {
    items = new Object[1000];
    n = 0;
  }
  public void push(Object item) {
    items[n++] = item;
  }
  public Object pop() {
    return items[--n];
  }
  public boolean empty() {
    return n == 0;
  }
```

declaration

allocation

MyStack.java

- Arrays are first class citizen of Java.
- No other back-doors of accessing them, for example, no pointer arithmetic
- Array reference bounds are checked at run time
  - No seg faults possible, tremendous help in reducing headaches
  - Also important implications for safety, security, and encapsulation

Pointers and Linked List

```
class MyNode {
  Object item;
  MyNode next;

  MyNode(Object item,
         MyNode next) {
    this.item = item;
    this.next = next;
  }
}
```

```
public class MyStack {
  MyNode list = null;

  public MyStack() {}
  public void push(Object item) {
    list = new MyNode
      (item, list);
  }
  public Object pop() {
    Object obj = list.item;
    list = list.next;
    return obj;
  }
  public boolean empty() {
    return list == null;
  }
}
```

MyStack.java

- Officially no pointers anywhere, behind the scene, each object is a pointer, called a reference, special **`null`** reference part of language
- No pointer arithmetic, no **`*`**, no **`->`**, no **`free()`**, no pointer bugs, no pain
- Reimplement stack using a linked list
  - **`push()`** code tricky: it allocates a new node, made by calling the constructor, which puts the old list head into the next field of the new node.

Java Libraries (Packages)

- Huge number of pre-written libraries
- Always check before you reinvent something of your own
- Watch out for version differences
  - http://java.sun.com/products/jdk/1.1/docs/api/packages.html
  - http://java.sun.com/products/jdk/1.2/docs/api/index.html
  - Reading these docs is a major part of learning/programming Java
  - Get a big picture of what they are but read details on-demand
- 1.2 is a significant improvement, for CS126, the “java.util” library has everything you can ask for: linked list, stacks, ...
- On the next slide, I will give a third implementation of the stack using a library class: Vector is an array that doesn’t require you to pre-specify a size and doesn’t fill up!

Example Use of Library

```
import java.util.*;

public class MyStack {
  Vector items;
  public MyStack() {
    items = new Vector();
  }
  public void push(Object item) {
    items.addElement(item);
  }
  public Object pop() {
    int end = items.size()-1;
    Object obj = items.elementAt
                 (end);
    items.removeElementAt
                 (end);
    return obj;
  }
  public boolean empty() {
    return items.isEmpty();
  }
}
```

MyStack.java

Sort of like #include

Vector is a class implemented by the java.util library, called a package

All of these are operations implemented by the package. You find out about them by reading the documentation, which you can download as a whole or read online.

Outline

- ~~Introduction~~
- ~~The basics~~
- **Object-oriented niceties**
  - **Inheritance**
  - **Encapsulation**
  - **Code reuse**
  - **Multiple implementations**
- Conclusions

Inheritance

```
public class MyImprovedStack extends MyStack {
  public Object pop() {
    if (n <= 0) {
      return null;
    }
    return items[--n];
  }
  public Object peek() {
    if (n <= 0) {
      return null;
    }
    return items[n-1];
  }
}
```

Inherits everything from MyStack

Overwrites old implementation

Adds new functionality

MyImprovedStack.java

- MyImprovedStack is a subclass of MyStack
- This example: adding functionality
- Another example use: "specialization"--a student class inherits from a person class

Encapsulation and Access Control

```
public class MyStack {
  protected Object[] items;
  protected int n;

  public MyStack() {
    items = new Object[1000];
    n = 0;
  }
  public void push(Object item) {
    items[n++] = item;
  }
  public Object pop() {
    return items[--n];
  }
  public boolean empty() {
    return n == 0;
  }
```

MyStack.java

- User of this class sees only what he's allowed to see
- Three key words:
  - **`private`**: accessible only by this class
  - **`protected`**: subclasses can see it too
  - **`public`**: accessible to all
  - (additional deals for "packages", read about them on-line if you care)

Code Reuse

```
import MyStack;
class StackTest {
  public static void main(String[] args) {
    MyStack s1 = new MyStack();
    s1.push ("first"); s1.push ("second");
    while (!s1.empty()) System.out.println(s1.pop());
    MyStack s2 = new MyStack();
    s2.push(new Integer(1)); s2.push(new Integer(2));
    while (!s2.empty()) System.out.println(s2.pop());
  }
}
```

Same code, same type

But different things in the stacks

StackTest.java

- This example: no need to write different codes for stack of Strings and stack of Integers

Multiple Implementations

```
import MyStack;
import MyArrStack;
import MyListStack;
class StackTest {
  public static void main(String[] args) {
    MyStack s;
    s = new MyArrStack();
    s.push ("first"); s.push ("second");
    while (!s.empty()) System.out.println(s.pop());

    s = new MyListStack();
    s.push ("first"); s.push ("second");
    while (!s.empty()) System.out.println(s.pop());
  }
}
```

Common interface

Different implementations

StackTest.java

- As long as a common interface is agreed upon
- We can pick and choose different implementations
- How's this done? Next slide...

Abstract Classes

```
public abstract class MyStack {
    public abstract void push(Object item);
    public abstract Object pop();
    public abstract boolean empty();
}
```

MyStack.java

```
import MyStack;
public class MyArrStack extends MyStack {
  ......
}
```

MyArrStack.java

```
import MyStack;
public class MyListStack extends MyStack {
  ......
}
```

MyListStack.java

- Abstract classes specify interfaces, no implementation
- Implementations inherit abstract classes and fill in implementation details

Outline

- ~~Introduction~~
- ~~The basics~~
- ~~Object-oriented niceties~~
- **Conclusions**

Java vs. C (Revisit)

- Bad things you **can** do in C that you **can't** do in Java
  - Shoot yourself in the foot (safety)
  - Others shoot you in the foot (security)
  - Ignoring wounds (error handling)
- Dangerous things you **have to** do in C that you **don't** in Java
  - Handling ammo (memory management: malloc/free)
- Good things that you **can** do in C but you don't; Java **makes** you
  - Good hunting practices (objected-oriented methodology)
- Good things that you **can't** do in C but you **can** now
  - Kills with a single bullet (portability)

Closing

- These are highlights, by no means complete
- Best way of learning
  - Study the tutorial online
  - Read and experiment with existing code
  - Read docs
- **I don't expect people to memorize or be able to reproduce syntatic details**
- I **do** expect people to be able to **read** and **understand** given code and concepts discussed