Java

Java is “Just another Object Oriented language”

Its syntax is similar to C++ (very similar, but there are
lots of small differences which result in a high frequency
of syntax errors if you are using both Java and C++)

Java’s style is closer to “purer” object oriented
languages like Eiffel and Smalltalk.

Object-Oriented Style

• Java (like Eiffel and Smalltalk) supports only the
object oriented style.

• Libraries defining elaborate hierarchies of classes are an intrinsic part of these languages

• Programs have a “main()” that creates the
principal object (this main() may be hidden, provided by
the compiler system), all subsequent control flow
involve object interactions.

C++ programming styles (1)

• C++ supports several programming styles:
  – conventional procedural program
    • main(), functions, global data
    • design by top down functional decomposition
  – hybrid style with objects (instances of simple classes)
    • objects perform well defined roles
    • overall control in “free functions” from which calls to objects are made
    • design
      – create some useful classes for component objects,
      – work out overall flow of control and put in free functions

Design with Libraries

• You design your programs in the context of the
class libraries

• When making the initial breakdown of a problem,
you think in terms of the reusable classes from the
library

• Some of these classes define complete subsystems,
  – you may think and design in terms of using an instance of class X
  – what you get is an X object and a number of objects of other
classes, these work together to achieve goals (library code defines
their interactions)

C++ programming styles (2)

• fully object based
  • main() creates single instance of principal object
  • program works through interactions among objects as defined in their member functions
  • design - “world of interacting objects”

• object oriented
  • object based, plus use of inheritance (allowing polymorphism etc) and use of elaborate class
libraries
  • design in the context of the class libraries
Learning Java

- Language?
  - Yes, you have to learn a new language; but if you know C++ there isn’t really much new to learn (just which parts of C++ to leave out)
- Libraries?
  - “learning Java” will to a larger extent involve learning about the libraries, and design ideas for using libraries, etc

Why Java?

- Why a new language?
- Why didn’t the originators simply provide their class libraries in C++ and work with C++?
- Let’s review the origins of Java and its development

Java origins

- Java comes from Sun’s research labs.
- Sun has several groups working on medium term speculative research - things that may become products in a few years, or may prove impractical.
- One project, started around 1990/1991, was to develop software for consumer durables.
  - cable TV decoders, VCR controllers, automobile controllers, etc.
  - limited memory, limited cpu power, lots of different architectures

Why not C++?

- First reason was the problem of multiple architectures
  - some having under powered and exotic CPUs
- Too difficult to get cross-compilers (compilers that run on your main development machine and produce code for different target machine) for all these different CPUs
- C++ didn’t match the applications
  - too concerned with efficiency of code
    - example: C++ programmers take responsibility for allocating and releasing memory
      - you need this for efficiency (as best performance when these operations are carefully tuned)
      - but it adds to complexity of code
  - too many programming paradigms
    - easy for programmers to drop back into old procedural style habits
- C++ too complex; e.g.
  - multiple inheritance
    - C++ rules are complex
    - Complexity relates to support for specialized, atypical uses of multiple inheritance
  - templates
    - A sophisticated approach to generic code, - but can achieve much the same results by cruder expedients
Why not C++? (4)

- C++ has maintained “backward compatibility” with C.
- Not required for intended applications.
- But is a source of a large number of errors
  - C permits (encourages) operations on pointers, operations that are unsafe
  - C provides a very poor, insecure model for array
  - ...

Something simpler than C++ (1)

- Cut back on the sophisticated but rarely used features
  - multiple inheritance
  - user defined types having same support as built in types
  - operator overloading
  - ...

Something simpler than C++ (2)

- Leave out error prone features
  - no #defines, macros, ...
  - no access to pointers (no int *p; p++;
  - no global data
  - no free functions
- Substitute simple expedients for things like templates.

Something simpler than C++ (3)

- Support for the programmer
  - provide more automation for free storage management
  - “garbage collection” of unused data structures
  - less efficient, more time consuming, but eliminates many errors and simplifies programming; efficiency not critical in intended language applications

Something easier to implement on multiple platforms

- How do you get code that can run on all sorts of different machines?
- Generate code for a single idealized computer.
- Simulate that ideal computer on all the real machines.
- Compiling code for an “ideal” machine and running simulators on actual computers is an old trick.

Conventional compiled programs & “Byte” code interpreters

- Conventional approach (C, C++ etc):
  - compiler translates program source to assembly language of target machine
  - assembly language converted to bit patterns representing actual instructions as interpreted by hardware CPU
  - tightly bound to target, efficient
Conventional compiled programs & “Byte” code interpreters

• Byte code approach:
  – have an imaginary “virtual” computer with a simple instruction set (often, one byte is used to represent the instruction - i.e. 256 different instructions possible, though usually less - hence name “byte code”)
  – compiler generates code for this imaginary computer
  – simulator program models the computer

Web interaction limited!

• Fancy Graphics?
  – Web pages could be set up that automatically asked server for a different picture every 15 seconds or so (very crude animation?)
  – “animated gifs” (several pictures concatenated together as if a single picture; display code shows each in turn, starts over when gets to end)

Byte code interpreter

• Virtual machine designed to make it easy to write a simple, compact simulator.

• So, write one compiler
• and lots of simulators - one for each target CPU
• Simulator will need to invoke a function (and execute tens to hundreds of instructions) to model each interpreted virtual instruction - so relatively inefficient.

Client side computations?

• Many applications that would be nice to have on Web couldn’t work this way --- too slow, too clumsy
  – e.g. imagine a “molecular modeller” that displays ball-and-stick molecules and allows them to rotate, it would be horribly slow if each 1º rotation needed an interaction with server.

• But client running browser program is a computer, why shouldn’t it do the work?

World Wide Web

• By 1994, use of the World Wide Web had started to grow substantially and capabilities of browsers were growing
  – first browsers, text and hypertext links
  – Mosaic browser, had added images etc
  – now had interactive web use
    • Web pages with forms entry,
    • User input sent to programs running on the server

Client side computations?

• Client side computing?
• Nice idea, but LOTS of problems

• Lots of different clients
  – e.g. as minimum have 6 varieties of Unix, PC (with three PC op. Systems), Mac
  – do you have different downloadable code for each?
• Also, the Internet isn’t a safe place
  – There are people out there who write things like viruses; “Trojan Horse” programs, worms
Byte code interpreter and the Web

- **Platform neutral**: you compile a single set of byte codes, these downloaded and run on interpreters specially written for each platform.
- **Web safe**: the byte code interpreter applies restrictions, e.g. can’t access local disk

Sun’s Java release

- Java released early 1996; v1.0.2 (minus some bugs) became the initial standard on Unix, Mac, PC.
- Version 1.1 (now 1.1.8 for bug fixes) 1997
- Version 1.2 (now 1.2.2 for bug fixes) 1998
- Version 1.3 has now been released

A niche where Java may survive?

- Java has just what the Web needs for client side computing
  - platform independence
  - potentially, some mechanisms for security
  - easy to integrate into existing browsers

JDK style development

- Use standard editor, enter Java code
- Compile with `javac`
- If “application”, run with `java`
- If Web dependent “applet”, run using a Java browser (e.g., Netscape, Explorer, or HotJava) or the simple “`appletviewer`” program

Sun’s Java release

- For Unix, Mac, and PC:
  - compiler (javac) compiles Java source to “class” files
  - interpreter (java) loads and runs class files
  - these distributed free along with class library.

Java Language

- Language features:
  - Data types
  - Variables (and constants)
  - Operators
  - Expressions (same as always)
  - Control flow
  - Functions
Data types

- A few “built in” types, everything else will be an instance of a class
  - Unlike C++, you can’t make classes “first class types” comparable in all respects to these “built in” types.
- Built in types (aka “primitive” types)
  - integer arithmetic
  - characters
  - real arithmetic
  - boolean

Real number types

- float about 6 digit accuracy
  \( \pm 3.402823 \times 10^{38} \)
- double about 15 digit accuracy
  \( \pm 1.7976931348623 \times 10^{308} \)
- (always use double for engineering or scientific calculations, too much round off with float)

Integer types

- C/C++ programs have portability problems related to integer types
  - the number range handled by a specific type is not defined in the standard!
    (Reason? This allows compiler writer to optimize code to suit capabilities of specific hardware)
- Java deliberately specifies number ranges to avoid such portability problems
  - may result in slightly less efficient arithmetic on some architectures, but this far less important than benefits of standardization

Character types

- Java has adopted Unicode 16 bit characters as its standard
  (these are available in C/C++ as wchar_t type, and there are analogues of strcmp() etc for 16-bit characters; however support is a little patchy)
- Most development systems and browsers still likely to supply only standard ASCII character set (now just one of many possible alphabets)

Integer types

- byte -128 to 127
- short -32768 to 32767
- int -2,147,483,648 to 2,147,483,647
- long -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807L
- can specify hex numbers, start 0x

Character types

- char ‘a’, ‘b’, ‘7’, ‘%’, ‘\b’, ‘\u’, ‘\u009b’ ‘\u00c1’ ‘\u177a’
- Java requires specific cast if you want to assign an integer value to a character
boolean

- boolean true false
- boolean is not compatible with integer types
- Significant tidy up from C where have automatic conversion of integer type to/from boolean (0 false, non-zero true)
  - an automatic conversion that is often source of errors

Variable declarations

- Variables can be declared as -
  - instance data members (each object created from class has its own copy)
  - class data members (one copy of variable shared by all instances of class)
  - local variables of member functions

Variables

- Different treatment of built-ins and class instances
  - built in types on stack (or “static data segment”)
  - class instances on heap
  - (In C++, don’t have this restriction).
- No global (or filescope) variables of any type.

Variable declarations

- Simple declarations -
  int counter;
  double slope, intercept;
  char mark;
  - data members are initialized (zero numerics, false booleans)
  - local variables of functions are not automatically initialized

Variable names

- Variable name must start with a “letter”
  - any letter from any Unicode alphabet, but stick to a..z, A..Z, _, $
- Variable name can contain letters, digits

Variable declarations

- Declaration may provide an initial value (should do this for all local variables)
  double sumxsq = 0.0;
  char hash = ‘#’;
constants

- The word const is a reserved word in Java language; currently it is not used.
- Java has a qualifier “final” that can modify declarations
  - can specify a class as final (can’t make subclasses)
  - can specify a member function as final (subclasses cannot redefine that function)
  - can specify an initialized variable as final (initial value cannot be changed, so it is a constant)

Derived types? No

- In C/C++ you can declare variables with “derived types” like pointer types, arrays, references.
- You can’t do that in Java.
- Arrays will come soon, they are considered as being much the same as objects (class instances)

operators

- Generally similar to those in C/C++.
- No operators relating to pointer types.
- An extra shift operator to resolve ambiguities that causes portability problems for C/C++ programs.

Operator precedence table

```
[] array subscripting, . member access, () function call
! logical not, ~ bit not, ++, --,
* / %
+ -
<< >> >>>
< <= > >=
== !=
&
^ 
||
&&
? : 
= += -= etc
```

Use parentheses on any expression complex enough to make you concerned about operator precedence!
Shift operators

- \( << \) shift left \( n << 3 \)
- \( >> \) "arithmetic shift right"
- \( >>> \) "logical shift right"

- C/C++ leave meaning of \( >> \) undefined, compiler writer can chose either arithmetic or logical right shift (hence portability problems!).

Control flow

- Some oddities
  - "labels" and modified forms of "break" and "continue" (keyword goto is reserved but not used)
  
```java
getRowData:
    for(int row=1; row<=nRows; row++) {
      ...  
      if(!ValidateDataSoFar(row)){
        System.out.writeln("data inconsistent!");
        continue getRowData;  
      }  
      ...  
    }
```

operators

- operators are essentially only defined for built in (primitive types)
- Unlike C++, you cannot define a ++ operator to work with instance of your classes
- One exception, Java did define operator+( ) for class String; can use + with String variables, it produces a new String by concatenating existing Strings (this operator will do type coercion, String + number invokes function to convert the number to a String).

Functions

- Form is generally similar to C/C++.
- But
  - all functions must be declared as member functions of a class (possibly static member functions eg. Math.sin(double), etc)

```java
int Largest(int data[], int num) {
  int largest = data[0];
  for(int i=1;i<num; i++)
    if(data[i]>largest) largest = data[i];
  return largest;
}
```

Control flow

- The usual -
  - selection
    - if
    - if ... else ...
  - switch
  - iteration
    - while
    - do ... while (repeat)

  - better to use "Enumerators" (iterators) when working through collections
Program structure

- Superficial similarities to C/C++, but underneath there are significant differences.
- C & C++ build program by compiling source files to .o (‘object’) files and then linking these with libraries to get an executable.
- Java interpreter loads code for classes when it needs to, there is no real “link” step.

Compiler

- With Java
  - a file will contain declaration of a principal class (and optionally some auxiliary classes)
  - all details of class in the one file (unlike C++, no separate header file, and no possibility of defining some member functions in one file and others in a different file)
  - name of file must match name of (principal) class (with file extension .java)
  - compiler creates files with byte codes (separate ‘.class’ file for each class defined in the file being compiled)

Runtime

- Two possibilities
  - simple applications run by the ‘java’ interpreter
  - applets (meant to be run by a browser, such as Netscape, which includes a Java interpreter)
- Either way,
  - byte codes of ‘principal’ class get loaded
  - “appropriate function” called, normally this will create one (or more) objects and get one to run
  - code for other classes loaded

Compiler

- A ‘.java’ file will start with “import” statements
  - role has some similarities with #includes in C/C++
    - if you are going to be declaring and using instances of classes defined elsewhere, compiler is going to have to read descriptions of those classes so that it can verify that they support the operations that you invoke on the instances that you create

Runtime

- Runtime system must ‘know’ where byte codes for other classes can be found
  - current directory
  - default directory (for classes supplied as part of standard Java release)
  - directories specified by CLASSPATH environment variable
  - possibly, loaded from remote site via Internet

Conclusions

- Java never intended as just a fancy language for small Web based applications.
- Original role, embedded systems, has been revived and may yet prove Java’s principal area of application.
- Java can be used for large applications, more typically done in C++.
- Java can be compiled (to real machine code, not byte code), so can be efficient.