Overview of the Java Programming Language

History:
- In the beginning, machines are programmed by hardwiring physical components!
- Later, idea of machine language (1GL) and assembly language (2GL).
  → Design language to reflect what underlying machine is capable of.
- Still later, idea of High-Level Language (HLL) (aka 3GL)
  → Design language to model abstract problem solving rather than details of what particular machine can do efficiently.
  e.g., iteration as an abstract loop, rather than complicated rat’s nest of GOTOs/branches.
- More recently, ideas of 4GLs (oriented to particular application domains, e.g., databases) and 5GLs (intelligent systems).

Evolution of HLLs (3GLs)
- First HLLs (FORTRAN, COBOL) were clunky, but they caught on because they were there first.
  → The designers were pioneers, so we forgive them.
  → These languages are still with us, mostly because so many big systems were written in them and it's too expensive to just rewrite them from scratch.
- Later, more attention paid to efficiency (e.g., C) and good design of abstractions (e.g., Algol, Pascal).
- Object-oriented programming caught on in the 1980s, mostly due to C++. Easier to write large programs correctly and efficiently. Much better support for high-level abstractions, enforceable user-defined rules.

C and C++
- The C programming language [AT&T Research] caught on in the Unix world in the 1970s.
  → Low level of abstraction; programmer must manage lots of details correctly (or risk catastrophe).
  → Little help when (not if) things go wrong.
  → Experienced programmers can write very efficient code.
  → Can write reasonably portable code if you try (usually).
  → Arcane, cryptic, hard to learn, beginners likely to shoot themselves in the foot repeatedly.
  → Poor support for abstractions in general; abstractions are mostly by programmer convention and discipline and are not enforceable.

C and C++
- The C++ programming language [AT&T Research] added better support for abstractions, which meant larger programs are easier to write and more likely to be correct.
  → OOP an older idea; C++ made it household word.
  → C++ is a proper extension to C; major selling point as can add C++ on top of existing C systems.
  → Abstractions can be enforced by language! (but can often get around them if you try)
  → Code can be just as efficient as C.
  → Big, hairy, complicated language. Often several ways of doing the same thing.

[99% true]
Enter Java

Current hypothesis: Java cures cancer.

- OOP!
- SAFE!
- PORTABLE!
- APPLETS!

- Looks a lot like C/C++, especially at the low level (e.g., int, double, while, switch etc.)
- disallows some of the “nastier” and more complicated aspects of C and C++ (e.g., no multiple inheritance, no operator overloading)

... but ...

- Tools lagging (esp. good “compilers”).
- Efficiency a concern.

Short History of Java

- 1991 — James Gosling (Sun Microsystems) wants to write platform independent code for embedded systems (e.g., toasters, elevators). Tries C++ but is unhappy with it. Starts to think about another creating language.
- 1993 — Develops new language code name “Oak”; superficially looks a lot like C++, but aim is to be portable and safe.
- 1994 — WWW starts to appear. Mosaic (first significant web browser) provides tempting application area.
- 1995 — Sun announces availability of Java language and HotJava browser to support it. Java applets contribute to explosion of interest in WWW.
- Sun’s official release of Java and a barebones development kit is called JDK. Current release is version 1.1.5. Others have their own development kits (CodeWarrior, VJ++, etc.).

Applets and Applications

Java applications:

- Tell system to run the main function of some class.
- Dynamically load other classes as needed.
- Applications are considered to be trusted (this doesn’t mean they are trustworthy!) They are allowed to read, write, create, delete files as they execute.

Compiling source to byte code:

hockey.java source code -> hockey.class byte code

Java compiler

Executing the application byte code:

hockey.class byte code -> Sun’s JVM, MS’s JVM (Running program)

Java byte code interpreter

Java Applets:

- “applet” == application-let
- Usually meant to be run by web browser (or JDK appletviewer)
- Compilation same as before; execution model is different:

Executing applet byte code:

- Browsers (Netscape, MS-IE) have their own JVM embedded in them!
- To execute an applet, grab the byte code from other internet site and execute locally (i.e., same machine as browser user).
**Applets and Security**

- Browsers usually consider applets to be untrustworthy; by default, applets have some restrictions on what they are allowed to do:
  - No touching (creating/deleting, etc.) files of user.
  - No other security surprises (e.g., no connecting to other computers, printing).
  - No secret communication with local or non-local programs.

[User can alter these policies by tweaking the Java SecurityManager.]

- Security is a huge, important area! And it’s only going to get more important.

---

**Other Java Features**

- Basic language is small and simple:
  - Syntax is very C-like.
  - BIG library for advanced users. Learn as you need to.
  - However, one you start into non-trivial programs, you have to start poring over the reference manual. Eventually, it becomes second-nature.

---

**Portable!**

Traditional (non-Java) view of compilation:

```
source code -> compiler -> object code
```

Java view of compilation:

```
source code -> Java compiler -> bytecode -> JVM
```

**Portable graphics:**

- Graphics (GUIs in particular) are usually OS-specific; traditionally, if you want a Mac and a Windows version of your program:
  1. Separate out GUI from underlying engine (as much as possible).
  2. Write single portable engine (as much as possible).
  3. Write a different GUI for each OS.

In practice, this is very messy!

- Java AWT is guaranteed to be implemented by any JVM. Use same source code for all versions of engines and GUIs!

- AWT GUIs look a bit different on different OSs, because each VM can decide what the “look and feel” should be like: e.g., size of buttons, colour schemes, etc.

(Java version 1.2 will attempt to alleviate this somewhat.)
Overview of Basic Java

Basic Types:

`int, double, boolean, char` etc.

```java
int numGallonsSold, ageOfCar = 10;
boolean creditCardValid = false;
final double area = Math.PI * r * r;
```

- Can declare variables as you need them (unlike C, Pascal) as long as before first use.
  [Good idea to declare as needed.]
- Use `double` rather than `float` for reals.
- Booleans are not integers and vice versa (unlike C).
- `String` type is very nice but a bit subtle (more later).
- Add `final` modifier in declaration means variable is a constant (must be give a value as declaration and can’t be changed).

Simple Output:

```java
System.out.println ("I sold 
 + numGallonsSold + " gallons.");
System.err.println ("Bad Visa card: 
 + visaNumber);
```

- Use `System.out.println` (or `print`) for simple output.
  [println adds a newline, print doesn’t]
- Use `System.err.println` for error message logs.
- For printing to a file or a GUI component, we’ll let you know.
- These routines take a character string as an arg.; can print multiple items by using “+” to concatenate.
- `print(b)` where `b` is a basic type (int, boolean, etc.) will print what you would expect.
- `print(p)` where `p` is a reference to an object invokes will result in printing `p.toString()`.

• Configurable security:
  → Secure by default, esp. when using other people’s code.
  → Can decide to use different security policies by choosing/configuring a security manager.

• Safety by default:
  → Object “pointers” are abstract references to objects, not raw memory addresses as in C/C++.
    - Cannot ask “what’s in byte 34578?”
    - Array bounds are always checked; cannot wander off the edge of an array.
  → Safe, simple automated storage allocation and reclamation.
  → Don’t shoot yourself in the foot quite so often.
  
  \(^4\text{Internet worm}\)

• Networked:
  → Can access files anywhere on the internet easily (modulo security considerations).
  → Previously, this was a real headache.

• Threaded:
  → Multiple “threads” of execution can be active at the same time, interacting in interesting ways.
  → Can allow multiple processors to communicate and co-operate to solve a large problem.
  → Java provides language-level support for concurrent threads.
Simple Input:

- ... is not so simple in Java.
- We will provide you with a class called TokenReader that can be set up to either read from a small GUI input widget or from a text file.
- We will tell you how to use it too.
- TokenReader is not a standard part of Java; we created it for you. You can get it from the course web page.
- Not a bad idea to put these two lines at the beginning of your main programs:

```java
SystemInput sysIn = new SystemInput();
TokenReader in = new TokenReader(System.in);
```

This will stop CodeWarrior from running a program and then shutting everything down before you can watch it execute.

if Statement:

```java
int age = in.readInt();
if (age >= 18) {
    System.out.println("You can vote!");
    if (age < 21) {
        System.out.println("but no beer for you");
    } else if (age >= 65) {
        System.out.println("Senior's discount?!");
    }
}
```

- if is just like C/C++
- Test in if and loops must be a boolean!
- Therefore `if (a=b)` is illegal! (unlike C)
- Also, switch is also like C/C++, including “fall through” problem in cases
  [Don’t forget to break after each distinct case.]

Loops and Repetition:

```java
while (i<N && !found) {
    ...
}
do {
    ...
} while (input == dummyValue);
for (i=0; i<N; i++) {
    ...
}
```

- Just like in C/C++.
- Get in the habit of defining boolean flags! Makes conditions more readable.

Advice on Style

- I don’t claim that my style is the only true way, just that it’s a good one. You will develop your own style in time, but copying an “old hand” is a good way to start.
- If a loop body or if statement body consists of only one statement, then it is legal to leave of the enclosing curly brackets.
  I strongly suggest you always include them!
  i.e., DO THIS:
  ```java
  for (i=0; i<N; i++) {
      A[i] = 0;
  }
  ```
- DON’T DO THIS:
  ```java
  for (i=0; i<N; i++)
      A[i] = 0;
  ```
  Trust me, it will save you time and again from dumb mistakes that are hard to spot.
• Where to put the curly brackets is a matter of taste. My style is a very good one. [See code examples throughout notes.]

• Indenting is also a matter of taste. Watch what I do and imitate (I use four spaces).

• Choose good variable names; be consistent!

<table>
<thead>
<tr>
<th>Good names</th>
<th>Bad names</th>
</tr>
</thead>
<tbody>
<tr>
<td>int numGallonsSold,</td>
<td>int myInt, nGallSld,</td>
</tr>
<tr>
<td>customerAge;</td>
<td>n1, n2, n3;</td>
</tr>
<tr>
<td>boolean inputOK,</td>
<td>boolean flag, boolVal,</td>
</tr>
<tr>
<td>cardValid;</td>
<td>input, inputStatus;</td>
</tr>
</tbody>
</table>

[It’s OK to use i, j, k as loop counters, ‘tho.]

• Use symbolic constants, not “magic numbers”.

```
// This is bad:
int [] grades = new int [271];
// This is better:
final int numStudents = 271;
int [] grades = new int [numStudents];
```

• Syntax is slightly different from C/C++.

• Declaration does not perform allocation of elements; need to do a new to actually get the array allocated!*

• If you declare an array \( A \) of \( N \) elements, then they are indexed by \( A[0], A[1], \ldots, A[N-1] \).

• Unlike C/C++, array bounds are always verified for you by the system. If you try to access \( A[N] \) or \( A[-3] \), the system will catch this as an error. This is great news!

[You will get a message about an ArrayIndexOutOfBoundsException being “thrown”.

*Use square brackets in new not parentheses.

• Comment, comment, comment!

  → Put comment before class definition, explaining what this class does or represents.

  → Put comment before or beside all top-level variable declarations explaining what variable’s purpose is.*

  → Comment each method to explain what parameters are (including the return value), and what function the method performs.

  → Put comments within method definitions judiciously. You don’t need to explain what “i=i+1;” does, but you should, say, explain what a loop or a block of statements does.

*a “Top-level” means not defined within a method body; i.e., a static or instance variable.

Arrays:

```
final int numFriends = 3;
String [] names;
names = new String[numFriends];
names[0] = "Harvey the Rabbit";
names[1] = "Spot the Dog";
names[2] = "Fluffy bunny";
```

Methods (aka Functions/Procedures)

```
public static int pow (int b, int p) {
    if (p < 0) {
        return 0;
    } else {
        int ans=1, i;
        for (i=0; i<p; i++) {
            ans *= b;
        }
        return ans;
    }
}
```

• Very C-like in syntax.

• All methods must be part of a class; no “standalone” functions.

• Usually preceded by access modifier:
  public, protected, or private

• May be preceded by static, which means it may access top-level variables or methods only if they are also declared as static.

[main functions are static]
No Scalar Reference Parameters:

- Unlike C++ and Pascal (but like C) Java does not have “reference parameters” for scalars:
  → Changes made to “scalar” parameters (i.e., non-objects, i.e., int, boolean etc.) are legal but do not percolate back to caller.
- Changes made to fields (variables) of objects are permanent, but if you change the object-reference parameter to point to a different object, that change will not percolate back either.

Output from following program is:

```java
class Employee {
    // Don't make instance variables public!
    public String name;

    public static void test (Employee e, int i) {
        System.out.println (e.name + " " + i);
        // This change is permanent:
        e.name = "Felix Potvin";
        System.out.println (e.name + " " + i);
        Employee e2 = new Employee();
        e2.name = "Terry Sawchuk";
        // These two changes are legal but will // not percolate back to the caller:
        e = e2;
        i = 1;
        System.out.println (e.name + " " + i);
    }

    public static void main (String [] args) {
        Employee e = new Employee();
        e.name = "Ken Dryden";
        int i = 29;
        System.out.println (e.name + " " + i);
        test (e, i);
        // Neither e nor i changed, but e.name did.
        System.out.println (e.name + " " + i);
    }
}
```

Ken Dryden 29
Ken Dryden 29
Felix Potvin 29
Terry Sawchuk 1
Felix Potvin 29