An Overview of Java

The Java Platform:

- APIs and libraries
- Java Virtual Machine
- Java

What is Java?

- An object oriented programming language with strong type-checking and automatic garbage collection.
- A “virtual machine” that performs run-time linking and run-time verification to guarantee the integrity of code.
- A set of libraries that abstract user interface issues to a platform neutral set of operations.
- A system that can support many levels of code: embedded applications; “applets” that run inside Web browsers; full-blown applications; or even an operating system.
History

- Began in 1990 with a project for consumer electronic device controllers.
- 1991: Programming language “Oak” for this project.
- 1992: Separate company (FirstPerson, Inc.) started for this project, but didn’t thrive.
- 1994: Sun began adapting Oak for the Internet.
- April 1995: Initial release (including HotJava), and rename to Java.
- May 1995: Netscape agrees to incorporate Java with Navigator.
- April 1996: Microsoft announces incorporation of Java into Windows 95.
- Following months: Lots and lots of people announce Java projects and support.

What makes Java so popular?

- Platform independence
- Pure (although slightly incomplete) object oriented design
- Secure for running untrusted code
Is this new?

No!

- Platform independence has been seen many places: The UCSD P-System, the Zork-language virtual machine, etc.
- Object oriented languages are in abundance: Smalltalk, Eiffel, Dylan, etc.
- Secure for untrusted code: Some anti-virus techniques, restricted system access, etc. (Java’s only big improvements come in this area)

So if it’s all old technology, why all the fuss?

- Timing, timing, timing. Emergence as the web was "growing up" was perfect timing.
- Familiarity — large base of C++ programmers made learning curve very short.
- The right combination of technologies.
- Good, free tools: compiler, debugger, and libraries
- Very good and cheap commercial tools: visual development environments, etc.
- Good side effect of being safe for untrusted code: safer for untrusted programmers! Strong type-checking and verification catches many subtle programmer bugs, increasing programmer productivity.
Platform Independence

- Goal is "Write Once, Run Anywhere"™ (yes, that phrase is a trademark of Sun)
- Don’t have to have a PC version, a Mac version, a Solaris version, a Digital Unix version, an HP-UX version, ... One version works on any system that supports the JVM.
- Still not quite there — but the bugs are being worked out.
- Absolutely vital to maintain standards that allow Java code written for one system to work on other systems: Sun vs. Microsoft lawsuit.

Remainder of presentation: How to make programming more reliable.

Or: What causes program errors and unpredictable behavior?

Note: Some of the following problems are corrected with the language Java, and others are corrected in the Java Virtual Machine.
Problem 1: Unchecked pointer arithmetic treats memory as one big block.

Example:

```c
int var1;
int *ptr = &var1;
int var2 = *(ptr + 100); // This is legal but nonsensical!
```

Solution 1: No pointers in Java! Reference types in the Java VM, but cannot be mixed with integers (no pointer arithmetic), so at compile time can verify that they will always point to something reasonable.

Related problem/solution: All array bounds are checked before indexing is performed.

Problem 2: Stack problems/overwriting critical info.

Example: (Intel x86 assembly language)

```assembly
mysubr proc
    pop ax
    mov ax, 0
    push ax
    ret
mysubr endp
```

Solution 2: Stack operations are restricted to be predictable and bounded to remain in the appropriate region. Somewhat restrictive, but loss is efficiency not capability.

Not allowed: (Intel x86 assembly language)

```assembly
mov bx, 10
mov cx, 0
prloop: xor dx, dx
        div bx
        push dx
        inc cx
        cmp ax, 0
        jnz prloop
```
Problem 3: Uninitialized variables cause unpredictable results.

Example:

```cpp
Object obj; // Assume no constructor that initializes values
int var;
cout << obj.data; // Unpredictable value!
cout << var; // Unpredictable value!
```

Solution 3: All class fields get an initial, predictable default value. Local variable initializations verified at run-time.

Problem 4: Memory allocation problems (pointer/reference not pointing to a valid object but used anyway).

Example:

```c
char *str;
strcpy(str, "Hello"); // What does str point to?!!
```

Solution 4: All dereference operators checked at run time.
Problem 5: Memory deallocation problems (memory not freed properly or used after freeing).

Example 1 (basic problem):

```cpp
Object *obj = new Object;
    // ... use *obj
delete obj;
   // ... more stuff
obj->data = 1;
```

Example 2 (more subtle):

```cpp
Node *curr = first;
    while (curr != 0) {
        delete curr;
            curr = curr->next;  // *curr no longer there!
    }
```

Example 3 (losing allocated memory):

```cpp
Node *curr = new Node;
    curr = first;  // What happened to new node?
```

Solution 5: No explicit deallocation available. All memory management is through automatic garbage collection, insuring consistency.

Problem 6: Types used inconsistently (at machine level or through unions in C/C++).

Example:

```cpp
union {
    int ival;
    float fval;
} u;
    u.ival = 10;
    cout << u.fval;  // Not really a float value stored there!
```

Solution 6: Memory locations in Java VM have values with a type, so not just treated as strings of bits. Type consistency enforced at run-time (in Sun JDK this is done through load-time code validation).
Problem 7: Changing types/class signatures without proper recompilation of relevant modules causes “offset shift”.

Example: Code compiled with this definition

```cpp
class myclass {
    public:
        int field1;
        float field2;
};
```

may use memory offset 0 within the object to refer to field 1. If later changed to

```cpp
class myclass {
    public:
        float field2;
        int field1;
};
```

now previously compiled code does not work correctly because fields have moved!

Solution 7: Java bytecode (i.e., object code) contains no offsets (which would require pointer arithmetic!), only name based references resolved at run-time.

Java Overview

Problem 8: Overloading integers and booleans allows incorrect test conditions to slip by.

Example:

```cpp
if (x = 1)
    cout << "x is one!";
```

Solution 8: Addition of a separate boolean type, which is required for all test conditions. No casting between boolean and integer types.

Java Overview
**Problem 9:** At machine level, improperly coded transfers may not jump to the beginning of an instruction, causing totally unpredictable results.

**Example:** (Intel x86 machine code)

```
0FFF:0100 B80001 MOV AX,0100
0FFF:0103 B83D01 CALL 0243
0FFF:0106 48 DEC AX
0FFF:0107 75FB JNZ 0104
0FFF:0109 C3 RET
```

**Solution 9:** All jump targets verified at loading time, before the code is executed.

---

**Problem 10:** For untrusted (downloaded) programs, machine code allows arbitrary OS calls (file modifications, deletions, etc.).

**Solution 10:** Only machine interface is through local, trusted libraries with access screened by a *security manager* class.
Final Note

These Java/JVM design decisions were made mostly to protect machine against rogue code or unpredictable behavior, but also excellent protection against programmer error!

- It’s difficult to catch all the subtle machine-failure-type bugs in a C++ program.
- It’s difficult for a subtle machine-failure-type bugs to get by testing in a Java program.

Of course, logic bugs are still the responsibility of the programmer!