# Intro to Java

Ron Even
December 4, 1995

# Motivation

- Multiple incompatible hardware architectures.
- Multiple incompatible OS's for each architecture.
- Multiple incompatible GUI's for each OS.
- Applications must work in a distributed client-server environment.
- Want to take advantage of internet, WWW, and "electronic commerce".

# Java Selling Points

- Object-oriented and familiar, yet simple
- Faster development cycle since Java is interpreted
- Machine and Platform Independent
- Robustness—Java run-time system manages memory
- Multithreading built into the language
- Fully Specified—no behavior is implementation-dependent or undefined
- Applications dynamically adaptable—can dynamically load code modules from anywhere in the network.
- Security—run-time system protects against viruses and tampering.

# Java Portability and Byte Code

- Compiler compiles down to byte codes. These are codes for a “virtual machine”. The Java interpreter and run-time system for a particular machine take care of interpreting the byte codes.

- Boon to software distribution. Need only one version of software.

- Sizes of various primitive types are built into the language—don’t depend on word size of a particular platform.

- Fully Specified

# Fully Specified

- All variables are assigned before they are used.

- All variables can refer only to objects of the correct types. The run-time system keeps track of type of each object.

- All operands to all operators are guaranteed to have the appearance of being evaluated in left-to-right order. There are no legal expressions which have undefined behavior.

# Object Technology in the Java Language

- Encapsulation—to implement information hiding and modularity
- Inheritance—code re-use and code organization
- Dynamic Bindings—for maximum flexibility while a program is executing

## Dynamic Loading and Binding

- Classes are linked in as required and can be downloaded from across networks.

- Can look up a class definition given a string containing its name—can compute a data type name and have it easily dynamically-linked into the running system.

# Automatic Garbage Collection

- Once you allocate an object, run-time system keeps track of the object's status and automatically reclaims memory when objects are no longer used.

- Memory manager keeps track of references to an object. When there are no more references to it, it is a candidate for garbage collection(GC).

- GC runs as low priority thread, hopefully avoiding the problem of interrupting the user to garbage collect.

## Security

- Need to be able to run “untrusted” code securely.
- No pointers in traditional sense.
- Memory layout decisions made by the run-time system—not the compiler.
- Can’t infer the physical memory layout of a class by looking at its declaration. Can’t manufacture pointers to memory.
- These arguments are fine as long as we’re restricted to using the Java compiler.

# Byte Code Verifier

- Last phase of the byte code loader.
- Checks that code won't break the Java run-time system.
  1. There are no operand stack overflows or underflows.
  2. The types of the parameters of all byte code instructions are known to always be correct.
  3. No illegal data conversions are done, like converting integers to pointers.
  4. Object field accesses are known to be legal—private, public, or protected.
  5. Allows the interpreter to go faster (fewer checks).

## Class Loader

- Environment is broken down into name spaces.
- Separated by source (one space for local file system, one for each network source).
- Class references are resolved by looking in the local name space first; prevents “spoofing” a built-in class.

# Network Security

The Java networking package allows different configurations for different levels of security.

- Disallow all network accesses.
- Allow all network accesses.
- Allow network accesses to only the hosts from which the code was imported.
- Allow network accesses only outside the firewall if the code came from outside.

# Multithreading

- Java libraries are all “thread safe”.
- Thread class provides methods to start a thread, run a thread, stop a thread, and make inquiries about a thread’s status.
- Threads are pre-emptive—only relinquish control when a higher priority thread comes along.

# Exceptions

- *try* block followed by one or more *catch* blocks.
- If an exception occurs in a *try* block, the following *catch* blocks are examined.
- The first *catch* block whose argument type matches the exception is executed.
- A *finally* clause may be attached after the *catch* blocks. In this case, the code in the clause gets executed after any *catch* block is executed.

# Types

- Primitive Types—Part of language

- Classes—All derived from Object class

- Interfaces—Guarantee that certain methods will be provided

- Arrays—One array type for each of the other types

# Values

- Primitive Types—variables which have primitive types are always passed by value; therefore, the only way to change the value of a primitive type variable $x$ is by explicitly changing the value of $x$.

- Dynamically Allocated Objects—all other variables (non-primitive) are passed by reference. If you want to pass one of these by value, you must explicitly copy it.

# Primitive Types

- Arithmetic
  - *byte*—8-bit two's-complement integer
  - *short*—16-bit two's-complement integer
  - *int*—32-bit two's-complement integer
  - *long*—64-bit two's-complement integer
  - *float*—32-bit IEEE 754 floating-point numbers
  - *double*—64-bit IEEE 754 floating-point numbers
- Characters: *char*—16-bit Unicode characters
- Boolean: *boolean*—possible values are *true* and *false*

## Primitive Types cont.

Any integral type may be cast to any other arithmetic type.

Java provides all the integral operators that C++ does.

Integer division by zero throws ArithmeticException.

Floating point types may be cast to any other arithmetic type and char.

# Classes

- Similar to C++ classes.

- All classes are derived from the class *Object*.

- Instance variables and methods can be *public*, *protected*, or *private* (same definitions as C++).

- Anything not declared as *public*, *protected*, or *private* is visible throughout its *package*. This is the only way to get “friendly” behavior.

- No multiple inheritance. Each class (except *Object*) has exactly one superclass.

- Initialized to *null*. Takes on value only through the = operator or the *new* operator.

# Static and Final Methods and Variables

- Static variables and methods are attached to a specific *class* instead of to a particular instance of a *class*. They are accessed through the *class* name instead of through a variable.

- Static methods may not reference instance variables or methods.

- Final variables must be initialized and may not be changed. They are constants.

- Final methods may not be overridden.

## Abstract Classes and Methods

- An abstract class is one that cannot be instantiated—only subclassed. It provides prototypes for methods that it doesn't implement.
- An abstract class definition is preceded by the word *abstract*.
- An abstract method (also preceded by the word *abstract*) is one which isn't implemented.

## Strings

- Provided as a Class in the java.lang package.
- Not just a string of *char*s.
- Provides + operator (concatenation).
- String length is fixed by constructor.
- Use StringBuffer class for variable length strings.

# Polymorphism

- Overriding methods.

- *instanceof* operator—check at run-time if a variable is instantiated with a particular subclass.

# Overloading

Method overloading occurs when two methods have the same name. In Java, this can only be done if the two methods' argument lists differ in some way. They may differ either in number, type, or order (or some combination of these. Note that return type is not considered, so you may not have two methods of the same name that differ only by return type.

There is no facility for operator overloading in Java.

# Interfaces

- An interface specifies a group of method prototypes and field variables.
- All field variables are implicitly static and final and must be initialized with a constant expression.
- An interface, $I$, may extend other interfaces. Any class that implements $I$ must implement all the interfaces that $I$ extends.
- A class that implements the interface must instantiate each method in the interface.
- A variable of type interface $v$ can be instantiated with a reference to any class that implements $v$.

# Arrays

- Each class type has an array type created automatically.

- Array type declared by adding [] to variable.

- Initialization through the *new* operator or =.

- All arrays are single-dimensional. Must use arrays of arrays instead of multi-dimensional arrays.

- An array is an object with a number of variable. Instead of having names, these variables are referenced by non-negative integers (their indices).

- The array length is not part of its type; thus, over its lifetime, a given array variable may refer to arrays of different lengths.

- Every array has a *.length* field, which is a final variable. Once an array object is allocated, its length never changes.

- Array bounds are checked at run-time; *ArrayIndexOutOfBoundsException* is thrown if an attempt to reference an index out of the range [0..length-1] is made.

# Storage Class

- Determines lifetime of a variable.

- Local variables—declared and allocated within a block. Discarded at end of block. Method parameters are considered local.

- Static variables—local to a class. Allocated when class is loaded and discarded when class is unloaded.

- Dynamic objects—instances of classes and arrays. Allocated by *new* expression. May be referenced by more than one variable. Garbage collector handles reclamation of storage used by dynamic objects.

# Structure

- Package—Made up of compilation units

- Compilation Unit—File made up of classes and interfaces with at most one public class or interface

## Synchronization

- Methods declared as *synchronized* do not run concurrently.

- Each class and each instantiated object has its own monitor that controls access to its variables.

```
public synchronized void stop() {
 if (kicker != null) {
      kicker.stop();
      kicker = null;
 }
}

public synchronized void startSort() {
 if (kicker == null || !kicker.isAlive()) {
      kicker = new Thread(this);
      kicker.start();
 }
}
```

## What Java doesn't have

- Header files, typedefs, #define, and preprocessor—makes Java more context free.
- Structures and Unions—just use classes.
- Functions—force programmers to stick to objects.
- Multiple Inheritance—interfaces address some of these capabilities.
- Goto statement—everyone hates these anyway...
- Operator Overloading
- Automatic Coercions—must explicitly cast if a loss of precision may occur.
- Pointers—cause of buggy code; major security hole.
- Templates—well, you can't have everything.

## Java Class Libraries

- Language Foundation Classes—Wrappers for primitive types and fundamental classes. Also Math routines.

- I/O Class Library—File and Stream input and output.

- Another Window Toolkit Class Library—Everything you need to build a GUI.

- Utility Class Library—Implements a variety of encoder and decoder techniques, data and time, hash table, vector, and stack.

- Network Interface Class Library—Extends the functionality of the I/O class library with socket interfaces and Telnet interfaces.

## Resources

- http://java.sun.com
- Gamelan—http://www.gamelan.com
- newsgroup—comp.lang.java