G22.2210

Programming Languages: PL

B. Mishra
New York University.

Lecture # 11

—Slide 1—

Java: History

- Spring 1990–April 1991: Naughton, Gosling and Sheridan ("The Green Team") of Sun Microsystems formulate a mission statement: "*Behind the Green Door*"

  > "To develop and license an operating environment [...] that enables services and informations to be persuasively presented via the emerging digital infrastructure."

- August 1992:
  - Gosling creates an "industrial strength" object oriented programming language named Oak (`(C++)--`).
  - Naughton designs an interface called "killer app."
  - A character named Duke (imp with a red nose) is created with "killer app" to guide the user through a cartoon house.

—Slide 2—

Java: History (Contd.)

- October 1992: Sun is ecstatic. Sun tries to market Oak.
- March 1993: Sun loses to Silicon Graphics. Sun is NOT ecstatic. The project is practically killed.
- June 1993: The Web and the Mosaic Browser find wide-spread use.
- January 1995: Oak is renamed Java. "Killer app" becomes an interpreter for a Web browser and is renamed HotJava. Java becomes famous.
- Happy Ending: Netscape buys Java. Microsft acquires Java. Jim Gosling becomes a house-hold name in the entire civilized world.

—Slide 3—

Java: 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".

—Slide 4—

Java: Properties

Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language.
— "The Java Language: A White Paper"

- *Simple*
  –Based on `C` and `C++`
  –Many problematic features of `C++` removed
- *Object-oriented*
  –*Class*: A collection of data and methods that operate on that data
  –Data and methods describe the state and behavior of an *object*
  –Classes are arranged in a hierarchy—Subclass can inherit behavior from its superclass.

—Slide 5—

Java: Properties (Contd.)

- *Distributed*
  –Java classes can be transmitted over the net
  –Library routines for network I/O
- *Interpreted*
  –Java interpreter executes the compiled byte codes
  –Byte codes provide an architecture neutral object file format
  –Java Virtual Machine (JVM)
- *Robust*
  –Strongly typed
  –Many checks performed at runtime
  –Automatic Memory Management—Garbage Collection(GC)
  –Exception Handling
- *Secure*
  –No way to forge pointers
  –Java code is verified
  –Restricted access to file systems and network

—Slide 6—

Java: Properties (Contd.)

- *Architecture Neutral*
  –Compiles to byte codes
  –Easy to interpret or compile on any CPU/OS
- *Portable*
  –Size and representation of primitive data types, defined by the language
  –Standard library, specified
- *High-Performance*
  –Byte codes can be compiled to native code
  –Potentially as fast as `C`
- *Multithreaded*
  –Built-in threads and synchronization primitives
- *Dynamic*
  –Adapts to an evolving environment
  –Classes are loaded in as needed
  –Run-time class definition allows classes to be dynamically linked

—Slide 7—

Portability

- Compiler compiles down to "byte codes" to run on a "virtual machine" (JVM).
  –The Java interpreter and run-time system for a particular machine take care of interpreting the byte codes.
- Easy software distribution.
  –Need only one version of software
  –Various primitive types are built into the language—Don't depend on word size of a particular platform.
- **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.

—Slide 8—

Object Oriented Programming

- *Encapsulation*
  –To implement information hiding and modularity
- *Inheritance*
  –Code re-use and code organization
- *Dynamic Loading and Binding*
  –For maximum flexibility while a program is executing
  - 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.

—Slide 9—

Automatic Garbage Collection

- Allocate an Object
  –Run-time system keeps track of the object's status
  –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
  –Thus (hopefully) avoids the problem of interrupting the user to garbage collect.

—Slide 10—

Java Security

- Must be able to run "untrusted" code securely.
- No pointers in traditional sense.
- Memory layout decisions made by the run-time system— not the compiler.
- Cannot infer the physical memory layout of a class by looking at its declaration
  Cannot manufacture pointers to memory.
- These arguments are "trustable" as long as restricted to using the Java compiler.

—Slide 11—

Java Types

- **Primitive Types:** Part of language
- **Classes:** Derived from Object class
- **Interfaces:** Guarantee that methods will be provided
- **Arrays:** One array type for each of the other types (Primitive, Classes and Interfaces)
- *Values:-*
  - *Primitive Types*
    –Variables which have primitive types are always passed by value
    –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 wish to pass one of these by value, *you must explicitly copy it*

—Slide 12—

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`
- Java can assign "narrower types" to "wider types"

```
byte < short < int < long < float < double
        char < int
```


—Slide 13—

Primitive Types (contd)

- 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`

—Slide 14—

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
- *Single Inheritance*
  Each class (except `Object`) has exactly one superclass.
- Initialized to `null`. Takes on value only through the = operator or the `new` operator.

—Slide 15—

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.

—Slide 16—

Abstract Classes and Method

- An abstract class is one that cannot be instantiated—only subclassed.
- Abstract class provides prototypes for methods that it does not implement.
- An abstract class definition is preceded by the word `abstract`.
- An abstract method (also preceded by the word `abstract`) is one which is not implemented.

—Slide 17—

Java Example

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

- `HelloWorld` is defined to be `class`
- No instance variable
- A single `public` method called `main`. `main` is static—It is attached to the class itself and not an instance of the class
- The method `main` contains in its body a single method invocation to display the string `"Hello World!"` on the standard output

—Slide 18—

Classes in Java: Example

- A class defines the instance variables and methods of an object. It is a template that defines how an object will look and behave when it is instantiated.

```
class Point{
   public float x;
   public float y;

   Point(){
      x = 0.0;
      y = 0.0;
   }
}
```

- *Instantiation*

```
Point myPoint;
myPoint = new Point();
```

- *Manipulation*

```
myPoint.x = 10.0;
myPoint.y = 25.7;
```


—Slide 19—

Classes in Java: Example (Contd.)

- *Constructor* Performs initialization when you instantiate objects from a class

```
public final class Integer extends Number{
   private int value;
   public Integer(int value){this.value = value;}
}

Integer myIntegerObject = new Integer(123);
```

- *Finalizer* Performs necessary "tear-down" (or "wills-and-testament") before the garbage collector is about to free the object

```
protected finalize(){
   try{
      close();
   } catch (Exception e){ }
}
```

—Slide 20—

Strings

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

—Slide 21—

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
- Interface involves dynamic method binding—There is a small performance penalty to using them
- *Combining Interfaces* Interface can incorporate one or more other interfaces (using `extend`)—This gives multiple inheritance over the interfaces.

—Slide 22—

Interfaces: Example

```
public interface Storing {
   void freezeDry(Stream S);
   void reconstitute(Stream S);
}
public interface Painting{
   ...
}

public class Image implements Storing, Painting {
   ...
   void freezeDry(Stream S){//JPEG compression of image...}
   void reconstitute(Stream S){//JPEG decompression of image...}
}

interface DoesItAll extends Storing, Painting {
   void doesSomethingElse();
}
```


—Slide 23—

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 variables. 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

—Slide 24—

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.

—Slide 25—

Structure

- **Package**
  –Made up of compilation units
- **Compilation Unit**
  –File made up of classes and interfaces with at most one public class or interface

—Slide 26—

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.

—Slide 27—

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

—Last Slide—

What Java Lacks

- *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*
- *Operator Overloading*
- *Automatic Coercions* –Must explicitly cast if a loss of precision may occur.
- *Pointers* –Cause of buggy code; major security hole.
- *Templates*

[End of Lecture #11]