Evolution of Programming Languages

- **40's machine level**
  - raw binary
- **50's assembly language**
  - names for instructions and addresses
  - very specific to each machine
- **60's high-level languages**
  - Fortran, Cobol, Algol
- **70's system programming languages**
  - C
  - Pascal (more for teaching structured programming)
- **80's object-oriented languages**
  - C++, Ada, Smalltalk, Modula-3, Eiffel, ...
    strongly typed (to varying degrees)
    better control of structure of really large programs
    better internal checks, organization, safety
- **90's scripting, Web, component-based, ...**
  - Perl, Java, Visual Basic, ...
    strongly-hyped languages
- **00's cleanup, or more of the same?**
  - C#, Python, ...
    increasing focus on interfaces, components

Java

- **invented mainly by James Gosling (Sun)**
- **1990: Oak language for embedded systems**
  - toasters, microwave ovens
  - needs to be reliable, easy to change, retarget
  - efficiency is secondary
  - implemented as interpreter, with virtual machine
- **1993: run it in a browser instead of a microwave**
  - renamed "Java"
  - HotJava browser supports Java applets, run JVM
- **1994: Netscape supports Java in their browser**
  - enormous hype: a viable threat to Microsoft
- **1995-present: rapid growth of libraries**
  - language is relatively stable
  - libraries grow and change rapidly
  - compiler technology improvements (but still runs slow)
  - significant commercial use
    but interface/glue, not applets, as originally thought
  - AP computer science language as of fall 2003
  - Sun sues Microsoft multiple times over Java

- **lots of documentation**
  - http://java.sun.com/docs

Java is fully buzzword-compliant

- **Sun: "simple, object-oriented, distributed interpreted, robust, secure, architecture neutral, portable, high performance, multi-threaded, dynamic"**
- **simple: a reaction to complexity of C++ and risks of C**
  - no goto, no header files, no preprocessor, no pointers
  - garbage collection
- **object-oriented: everything is a class**
  - no independent variables or functions
- **distributed: classes for networking, URL's, etc.**
- **interpreted: compiled into byte codes for a virtual machine**
  - JVM interprets byte codes on the target environment
  - the same everywhere
- **robust: eliminates unsafe constructs**
  - strongly typed, no pointers, garbage collection, exception handling
- **secure: language is safer; security model**
  - byte code verifier, run-time checks (e.g., array bounds, casting)

Buzzwords, continued

- **architecture neutral: runs on anything**
  - byte codes + JVM; large set of libraries
- **portable: runs the same on anything**
  - bytes codes + JVM;
  - sizes, behaviors, etc., fully specified
  - "write once, run anywhere" (in theory)
- **high performance: (not really)**
  - just-in-time compilation, native mode extensions
- **multi-threaded:**
  - language and library facilities for multiple threads in a single process
- **dynamic:**
  - classes loaded as needed (like .DLL or shared libraries)
  - run-time type identification, etc.

Java vs. C and C++

- **no preprocessor**
  - **`import`** instead of **`#include`**
  - constants use **`static final`** declaration
- **C-like basic types, operators, expressions**
  - sizes, order of evaluation are specified
    byte, short, int, long: signed integers (no unsigned)
    char: unsigned 16-bit Unicode character
    boolean: true or false
- **really object-oriented**
  - everything is part of some class
  - objects all derived from **Object** class
  - **`static`** member function applies to whole class
- **references instead of pointers for objects**
  - null references, garbage collection, no destructors
  - == is object identity, not content identity
- **all arrays are dynamically allocated**
  - int[] a; a = new int[100];
- **strings are more or less built in**
- **C-like control flow, but**
  - labeled break and continue instead of goto
  - exceptions: try {...} catch(Exception) {...}
- **threads for parallelism within a single process**
  - in language, not a library add-on

Hello world in Java

```
public class hello {

  public static void main(String[] args)
  {
       System.out.println("hello, world");
  }
}
```

- **compile creates hello.class**
  javac hello.java
- **execution starts at main in hello.class**
  java hello

- **filename has to match class name**

- **libraries in packages loaded with `import`**
  - java.lang is core of language
    System class contains stdin, stdout, etc.
  - java.io is basic I/O package
    file system access, input & output streams, ...

Fahrenheit / Celsius example

```
public class fahr {

 public static void main(String[] args){
   for (int fahr = 0; fahr < 300; fahr += 20)
     System.out.println(fahr + "   " +
                  5.0 * (fahr - 32) / 9.0);
 }

}
```

- **System.out.println is only for a single string**
- **formatted output needs the Format class**
  which is a total botch

2 versions of **echo** command

```
public class echo {
 public static void main(String[] args)
 {
   for (int i = 0; i < args.length; i++)
      if (i < args.length-1)
         System.out.print(args[i] + " ");
      else
         System.out.println(args[i]);
 }
}

public class echo1 {
  public static void main(String[] args)
  {
     String s = "";  // must be initialized

     for (int i = 0; i < args.length-1; i++)
         s += args[i] + " ";
     if (args.length > 0)
         s += args[args.length-1];
     if (s != "")
         System.out.println(s);
  }
}
```

- **arrays have a length field**
- **subscripts always checked**

Java I/O and file system access

- **import java.io.***

- **byte I/O**
  - InputStream and OutputStream
- **exceptions**
- **file access**
  - FileInputStream, FileOutputStream
- **buffering**
  - BufferedInputStream, BufferedOutputStream
- **etc.**

- **character I/O**
  - InputReader and OutputWriter
  - InputStreamReader, OutputStreamWriter
  - BufferedReader, BufferedWriter

- **String library**
- **miscellaneous useful stuff**

Byte-at-a-time I/O

```
// cat <input >output

import java.io.*;

public class cat1 {

 public static void main(String args[]) {
   int b;

   try {
      while ((b = System.in.read()) >= 0)
         System.out.write(b);
   } catch (IOException e) {
      System.err.println("IOException " + e);
   }
 }
}
```

- **System.in, .out, .err like stdin, stdout, stderr**
- **read() returns next byte of input**
  - returns -1 for end of file
- **any error causes an IO Exception**
  - caught by the catch() statement

Exceptions

- **C-style error handling**
  - ignore errors -- can't happen
  - return a special value from functions, e.g.,
    -1 from system calls like open()
    NULL from library functions like fopen()
- **leads to complex logic**
  - error handling mixed with computation
  - repeated code or goto's to share code
- **limited set of possible return values**
  - extra info via errno and strerr: global data
  - some functions return all possible values
    no possible error return value is available

- **Exceptions are the Java solution (also in C++)**
- **exception indicates unusual condition or error**
- **occurs when program executes a throw statement**
- **control unconditionally transferred to catch block**
- **if no catch in current function, passes to calling method**
- **keeps passing up until caught**
  - ultimately caught by system at top level

try {...} catch {...}

- **a method can catch exceptions**

```
public void foo() {
 try {
        // if anything here throws an IO exception
        // or a subclass, like FileNotFoundException
 } catch (IOException e) {
       // this code will be executed
       // to deal with it
 }
```

- **or it can throw them, to be handled by caller**

- **a method must list exceptions it can throw**
  - exceptions can be thrown implicitly or explicitly

```
public void foo() throws IOException {
    // if anything here throws an exception
    // foo will throw an exception
    //     to be handled by its caller
 }
```

Why exceptions?

- **reduced complexity**
  - if a method returns normally, it worked
  - each statement in a **try** block knows that the previous statements worked, without explicit tests
  - if the **try** exits normally, all the code in it worked
  - error code grouped in a single place
- **can't unconsciously ignore possibility of errors**
  - have to at least think about what exceptions can be thrown

```
public static void main(String args[])
    throws IOException {
        int b;
        while ((b = System.in.read()) >= 0)
                System.out.write(b);
}
```

- **don't use exceptions for normal flow of control**
- **don't use for "normal" unusual conditions**
  - e.g., in.read() returns –1 for EOF
  - instead of throwing an exception
  - should a file open that fails throw an exception?

File I/O of bytes

```
import java.io.*;

public class cp1 {

   public static void main(String[] args) {
      int b;

      try {
         FileInputStream fin =
            new FileInputStream(args[0]);
         FileOutputStream fout =
            new FileOutputStream(args[1]);

         while ((b = fin.read()) > -1)
            fout.write(b);
         fin.close();
         fout.close();
      } catch (IOException e) {
         System.err.println("IOException "+e);
      }
   }

}
```

- **this is very slow because I/O is unbuffered**

Buffered byte I/O

```
import java.io.*;

public class cp2 {

  public static void main(String[] args) {
     int b;

     try {
        FileInputStream fin =
           new FileInputStream(args[0]);
        FileOutputStream fout =
           new FileOutputStream(args[1]);
        BufferedInputStream bin =
           new BufferedInputStream(fin);
        BufferedOutputStream bout =
           new BufferedOutputStream(fout);

        while ((b = bin.read()) > -1)
           bout.write(b);
        bin.close();
        bout.close();
     } catch (IOException e) {
        System.err.println("IOException " +e);
     }
  }

}
```

Unicode (www.unicode.org)

- **universal character encoding scheme**

- **UTF-16**
  - 16 bit internal representation
  - encodes all characters used in all languages
    - numeric value and name for each
    - semantic info like case, directionality, …

- **UTF-8**
  - byte-oriented external form
    - variable-length encoding
  - compatible with ASCII 7-bit form
    - ASCII characters occupy 1 byte in UTF-8

- **expansion mechanism for > 2^{16} characters**
  - 94000+ characters today

- **Java supports Unicode**
  - **char** data type is 16 bits
  - **String** data type is 16-bit Unicode chars
  - **\uhhhh** is Unicode character hhhh

Character I/O (char instead of byte)

- **use a different set of functions for char I/O**
- **works properly with Unicode**

- **InputStreamReader adapts from bytes to chars**
- **OutputStreamWriter adapts from chars to bytes**

- **use Buffered(Reader|Writer) as well**

```
public class cp4 {
public static void main(String[] args) {
  int b;
  try {
     BufferedReader bin = new BufferedReader(
        new InputStreamReader(
           new FileInputStream(args[0])));
     BufferedWriter bout = new BufferedWriter(
        new OutputStreamWriter(
           new FileOutputStream(args[1])));
     while ((b = bin.read()) > -1)
        bout.write(b);
     bin.close();
     bout.close();
  } catch (IOException e) {
     System.err.println("IOException " + e);
  }
}
```

Line at a time character I/O

- **handles Unicode**

```
public class cat3 {

public static void main(String[] args) {
   BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
   BufferedWriter out = new BufferedWriter(
      new OutputStreamWriter(System.out));
   try {
      String s;
      while ((s = in.readLine()) != null) {
         out.write(s);
         out.newLine();
      }
      out.flush();
   } catch (Exception e) {
      System.err.println("IOException " + e);
   }
}
```

String library functions

- **String is sequence of Unicode chars**
  - immutable: each update makes a new String
  - indexed from 0 to str.length()-1
- **search, comparison, etc.:**
  - substring, toUpperCase, toLowerCase
  - compareTo, equals, equalsIgnoreCase
  - startsWith, endsWith, indexOf, lastIndexOf
  - ...

- **StringTokenizer**

```
StringTokenizer st = new
  StringTokenizer(str);
while (st.hasMoreTokens()) {
   String s = st.nextToken();
   ...
}
```

- **StringBuffer vs String**
  - String can be inefficient
    have to create new ones instead of changing existing
  - StringBuffer is mutable
    grows & shrinks to match size
  - append, insert, setCharAt, ...

Runtime, Process, exec

```
public class runtime1 {
   public static void main(String[] args) {
        runtime1 r = new runtime1();
   }

 runtime1() {
   try {
      Runtime rt = Runtime.getRuntime();
      BufferedReader bin = new BufferedReader(
         new InputStreamReader(System.in));
      String[] cmd = new String[3];
      cmd[0] = "/bin/sh";   // Unix-specific
      cmd[1] = "-c";
      String s;
      while ((s = bin.readLine()) != null) {
         cmd[2] = s;
         Process p = rt.exec(cmd);
         BufferedReader pin = new BufferedReader(
            new InputStreamReader(p.getInputStream()));
         while ((s = pin.readLine()) != null)
            System.out.println(s);
         pin.close();
         p.waitFor();
         System.err.println("status = " + p.exitValue());
      }
   } catch (InterruptedException e) {
      e.printStackTrace();   // ignored
   } catch (IOException e) {
      e.printStackTrace();
   }
 }
```