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?**
  - Python, PHP, C#, ...
    increasing focus on interfaces, components

Program structure issues

- **objects**
  - user-defined data types
- **components**
  - related objects
- **interfaces**
  - detailed boundaries between code that provides a service and code that uses it
- **information hiding**
  - what parts of an implementation are visible
- **resource management**
  - creation and initialization of entities
  - maintaining state
  - ownership: sharing and copying
  - memory management
  - cleanup
- **error handling**

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: reaction to complexity of C++, risks of C**
  - no goto, no header files, no preprocessor, no pointers
  - garbage collection
- **object-oriented: <u>everything</u> 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
- **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

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

- **compiler 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, ...

Basic data types

```
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);
 }
}
```

- **basic types:**
  - boolean   true / false (no conversion to/from int)
  - byte   8 bit signed
  - char   16 bit unsigned (Unicode character)
  - int   32 bit signed
  - short, long, float, double
- **String is sort of built in**
  - "..." is a String
  - holds chars, NOT bytes
  - does NOT have a null terminator
  - + is string concatenation operator; += appends

- **System.out.println(s) is only for a single string**
  - formatted output is a total botch

C interface for an RE package

- **functions analogous to assignment 1:**

```
RE *RE_new(char *)
int RE_match(RE *, char *)
int RE_start(RE *)
int RE_end(RE *)
void RE_free(RE *)
...
```

- **"RE" is an *opaque type***
  - conceals the implementation as much as possible
- **implementation uses a structure like this**

```
typedef struct RE {
   ...
} RE;
```

- **user code sees only**

```
typedef struct RE *RE;
```

- **analogous to FILE* in C stdio**

- **in real life, there would be a header file RE.h**

Design issues

- **what functions?**
  - relatively few
  - fundamental, most commonly used
  - not easily synthesized from others
    but others can be synthesized from them
  - not easily implemented by users

- **this would be sufficient**

```
int RE_match(char *re, char *str,
             int *start, int *end);
```

- **but not really convenient or efficient:**
  - typically compile once, test matches often
  - often don't care about the matched string

- **better: different functions for different ops**
  - create a new regexp from a string (constructor)
  - match a string
  - access matched substring
  - free any resources (destructor)

Convenience & usability issues

- **small things, but they make a difference**
- **should there be functions for common operations**
  - immediate match of a regexp and string
    like Java's Pattern.matches(regexp, string)
    (which is an anchored match!!)
- **which of these is best?**

```
char *RE_start(),  int RE_length()
char *RE_start(),  char *RE_end()
char *RE_matched_substr()
```

- **how should errors be reported and returned?**
  - bool, int, struct, pointers?
  - print? assertion failure?
- **consistency in choices, naming, order of args, ...**

Resource management issues

- **when are start() and end() valid?**
  - what if source string changes?
  - what if multiple matches are in process?
- **what if you want successive searches, as in Java's Matcher.find?**
  - who remembers where you were?
  - what if the source string has changed in the interim?
  - how do you make it re-entrant?
  - why is C's strtok is a botch?
- **what if there were an array of matched substrings?**
  - like Perl's $1, $2, ...
- **suppose RE's were to be cached as in Awk**
  - how are they coordinated?
- **how would you know if the RE had changed?**
  - is the string saved? hashed? quietly assumed ok?

Who manages what memory when?

- **a big, fundamental interface issue**
  - getting it wrong or inconsistent is a major problem
  - making it hard for users is a major problem
- **char *RE_substr(RE *) needs space for string**
- **who allocates space for the string?**
- **should it grow? without limit?**
- **who grows it?**
- **who complains if it gets too big? how?**
- **who owns it?**
- **who can change its contents? how?**
- **who sees the changes? re-entrant?**
- **what is its lifetime?**
  - when are pointers into the data structure invalidated?
- **who frees it?**
- **these issues are not all solved by garbage collection**

Classes and objects

- **language support for design and implementation of data structures and operations on them**
  - data abstraction and protection mechanism
  - originally from Simula 67

```
class thing {
    public part:
        methods: functions that define what operations
            can be done on this kind of object
        visible outside the class
    private part:
        functions and variables that implement the
            operation
        invisible outside the class
}
```

- **a class defines a new data type**
  - can declare variables and arrays of this type, pass to functions, return them, etc.
- **object: an instance of a class variable**
- **method: a function defined in the class**
  - (and visible outside)
- **from outside, can't tell HOW the operations are implemented, only WHAT they do**
- **localizes all aspects of design & implementation**

Classes & objects in Java

- **in Java, <u>everything</u> is part of some object**
  - all classes are derived from class Object

```
public class RE {
   String re;      // regular expression
   int start, end; // of last match

   public RE(String r) {...} // constructor
   public int match(String s) {...}
   public int start() { return _start; }
   int matchhere(String re, String text) {...}
    // etc.
}
```

- **member functions are defined inside the class**
  - internal functions shouldn't be public (e.g., matchhere)
  - internal variables shouldn't be public

Constructors: making a new object

- **all objects are created dynamically, by a special member function called a constructor**

```
public RE(String str) { // same name as class
   re = str;
}
```

- **have to call new to construct an object**

```
RE re; // null: doesn't yet refer to an object

re = new RE("abc*"); // now it does

int m = re.match("abracadabra");
int start = re.start();
int end = re.end();
```

- **can define multiple constructors with different arguments to construct in different ways**

```
public RE() { /* ??? */ }
```

Class variables & instance variables

- **every object is an instance of some class**
  - created dynamically by calling **new**
- **class variable: a variable declared static in class**
  - only one instance of it in the entire program
  - exists even if the class is never instantiated
  - the closest thing to a global variable in Java

```
public class RE {
   static int num_REs = 0;

   public RE(String re) {
      num_REs++;
      ...
   }
   public static int RE_count() {
       return num_REs;
   }
```

- **class methods**
  - most methods associated with an object instance
  - if declared static, associated with class itself, not a specific instance
    e.g., main()

Class methods

- **most methods associated with an object instance**
- **if declared static, amounts to a global function**

```
class RE {
   public boolean equals(RE r) {
      return re.equals(r.re);
   }
   public static boolean equals(RE r1, RE r2) {
      return r1.re.equals(r2.re);
   }

   public static void main(String[] args) {
      RE r1 = new RE(args[0]);
      RE r2 = new RE(args[1]);
      if (equals(r1, r2)) ...    // compare contents
      if (r1.equals(r2)) ...     // compare contents
      if (r1 == r2) ...          // object equality
   }
```

- **some classes are entirely static members and class functions, e.g., Math, System, Color**
  - can't make a new one: no constructor

Destruction & garbage collection

- **interpreter keeps track of what objects are currently in use**
- **memory can be released when last use is gone**
  - release does not usually happen right away
  - has to be garbage-collected
- **garbage collection happens automatically**
  - separate low-priority thread manages garbage collection
- **no control over when this happens**
  - can set object reference to **null** to encourage it
- **Java has no destructor (unlike *C++*)**
  - can define a finalize() method for a class to reclaim other resources, close files, etc.
  - no guarantee that a finalizer will ever be called
- **garbage collection is a great idea**
  - but this is not a great design

Typical program structure

```
class RE {

  private class variables
  private object variables
  public RE methods, including constructor(s)
  private functions

  public static void main(String[] args) {
    re = args[0];
    for (i = 1; i < args.length; i++)
      fin = open file args[i]
      grep(re, fin)
  }
  static int grep(String regexp, FileReader fin) {
    RE re = new RE(regexp);
    for each line of fin
      if (re.match(line)) ...
  }
}
```

- **order of declarations doesn't matter**

Scope and visibility

- **only one public class per file**
  - public class hello { } has to be in hello.java
- **public methods of the class are visible outside the file**
- **other methods are not**
  - default is file private
- **other classes in a file are visible within the file**
- **but not visible outside the file**
- **variables of a class are always visible within the class**
- **and to other classes in the same file unless private**
- **static variables are visible to all class instances**

```
class Math {
   public static double PI = 3.141592654;
}
double d = Math.cos(Math.PI)
```

"Real" example: regular expressions

- **simple class to look like RE**
- **uses the Java 1.4 regex mechanism**
- **provides a better interface** (or at least less clumsy)

```
import java.util.regex.*;

public class RE {
        Pattern p;
        Matcher m;

   public RE(String pat) {
        p = Pattern.compile(pat);
   }
   public boolean match(String s) {
        m = p.matcher(s);
        return m.find();
   }
   public int start() {
        return m.start();
   }
   public int end() {
        return m.end();
   }
}
```

I/O and file system access

- **import java.io.***
- **byte I/O**
  - InputStream and OutputStream
- **character I/O (Reader, Writer)**
  - InputReader and OutputWriter
  - InputStreamReader, OutputStreamWriter
  - BufferedReader, BufferedWriter
- **file access**
- **buffering**
- **exceptions**
- **in general, use character I/O classes**

Byte-at-a-time I/O

```
// cat <stdin >stdout

import java.io.*;

public class cat1 {

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

     while ((b = System.in.read()) >= 0)
        System.out.write(b);
  }
}
```

- **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**
  - which is passed on by main

File I/O of bytes

```
// cp infile outfile
import java.io.*;
public class cp1 {
  public static void main(String[] args)
        throws IOException {
     int b;

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

     while ((b = fin.read()) > -1)
        fout.write(b);
     fin.close();
     fout.close();
  }
}
```

- **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)
         throws IOException {
      int b;

      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();
   }
}
```

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 <u>throw</u> statement**
- **control unconditionally transferred to <u>catch</u> block**
- **if no <u>catch</u> 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
 }
```

With exceptions

```
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);
     }
   }
}
```

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?

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) for speed**
  - and it has a readLine method

```
public class cat3 {
  public static void main(String[] args)
        throws IOException {
     BufferedReader in =
        new BufferedReader(
          new InputStreamReader(System.in));
     BufferedWriter out =
        new BufferedWriter(
          new OutputStreamWriter(System.out));
     String s;
     while ((s = in.readLine()) != null) {
        out.write(s);
        out.newLine();
     }
     out.flush();   // required!!
  }
}
```

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

Strings

- **a String is a sequence of (Unicode) chars**
  - immutable: each update makes a new String
    s += s2 makes a new s each time
  - indexed from 0 to str.length()-1

- **useful String methods**
  - charAt(pos) return character at pos
  - substring(start, len) return substring

```
for (i = 0; i < s.length(); i++)
   if (s.charAt(i) != s.substring(i, 1))
      // can't happen
```

- **String parsing**

```
String[] fld = str.split("\\s+");

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

String methods

- **search, comparison, etc.:**
  - substring, toUpperCase, toLowerCase
  - compareTo, equals, equalsIgnoreCase
  - startsWith, endsWith, indexOf, lastIndexOf
  - …

- **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();
    }
  }
```