Java in 21 minutes

• Hello world
• Basic data types
• Classes & objects
• Program structure
• Constructors
• Garbage collection
• I/O
• Strings
• Exceptions

Hello World

```java
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, etc.

Java in 21 minutes
Basic data types

• Basic types:
  – boolean: true / false
  – 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

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

• Arrays have a length field (a.length)
  – subscripts are always checked

• System.out.println(s) is only for a single string
  – + is string concatenation operator
  – does NOT have a null terminator
  – holds chars, NOT bytes
  – "" is a String

• Arrays have a length field (a.length)
  – subscripts are always checked

• Strings have a length() function
  – System.out.println(s) ("...")

2 versions of echo

Basic data types

public class echo
{
  public static void main(String[] args)
  {
    String s = "";
    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);
  }
}

public class echo1
{
  public static void main(String[] args)
  {
    for (int i = 0; i < args.length - 1; i++)
      System.out.print(args[i] + " ");
    if (i < args.length - 1)
      System.out.println(args[i]);
  }
}

String s = "";
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);
Classes, objects and all that

• data abstraction and protection mechanism
  • originally from Simula 67, via C++ and others

class thing {
  public part:
  methods: functions that define what operations can be done on this kind of object

  private part:
  functions and variables that implement the operation
}

• defines a new data type "thing"
  – 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 within the class
  – and visible outside (and visible outside)
  – internal functions should be public (e.g., matchhere)
  – internal variables defined but should be public
  – internal variables and functions are not accessible from outside the class

• not possible to determine HOW the operations are implemented, only WHAT they do
  from outside the class

• private variables and functions are not accessible

+ all objects are created dynamically
  + have to call new to construct an object

  + all objects are derived from class Object
  + member functions are defined inside the class

  + constructor: a function defined within the class, return the "thing"
    
    {  
      if (matchhere(regex, re, start, text, end)) {  
        return start;  
      }  
      public RegExp(String exp) {  
        constructor // {  
          start = 0;  
          end = 0;  
          exp = exp;  
        }  
      }  
      public RegExp(String re) {  
        constructor // {  
          start = 0;  
          end = 0;  
          exp = re;  
        }  
      }  
    }

  + member variables are declared from class Object
  + in Java, everything is part of some object

Classes & objects
Constructors: making a new object

```java
public RE(String re) {
    this.re = re;
}
RE r;
r = new RE(s);
```

• "this" is the object being constructed or running

Public class RE {
    static int num_REs = 0;
    public RE(String re) {
        num_REs++;
    }
}

• Class methods
  • most methods associated with an object instance
  • can use multiple constructors with different arguments to construct in different ways:
  • can use multiple constructors with different arguments in a single program

Class variables & instance variables
• every object is an instance of some class
  • created dynamically by calling new
  • every object is an instance of some class

• Class variables: a variable declared static in class
  • the closest thing to a global variable in Java
  • exists even if the class is never instantiated
  • only one instance of it in the entire program

• Instance variables: a variable declared not static in class
  • if declared static, associated with class itself
  • if not static, associated with an object instance
```
Program structure

- typical structure is a great idea

  class RE {
  private variables
  public RE methods, including constructor(s)
  private functions
  public static void main(String[] args) {
    extract re
    for (i = 1; i < args.length; i++)
      fin = open up the file...
      grep(re, fin)
  }
  static int grep(String regexp, FileReader fin) {
    RE re = new RE(regexp);
    for each line of fin
      if (re.match(line)) ...
  }

Java has no destructor (unlike C++)

- order doesn't matter

  { ... (re.match(t))
    for each line of fin :
      re = new RE(regex);  
  } (fin)

  grep(re, fin)

  ... open up the file...
  ... for (i = 1; i > 0; length ++)
  extract re

  public static void main(String[] args) {
    private functions
    private RE methods, including constructor(s)
    private variables
  }

  class RE

  typical structure is a great idea

  garbage collection is a great idea

  - no guarantee that a finalizer will ever be called
  - other resources, close files, etc.
  - can define a finalize() method for a class to reclaim
  - Java has no destructor (unlike C++)

  - can set object reference to null to encourage it
  - no control over when this happens
  - separate low-priority thread manages garbage
  - garbage collection happens automatically
  - has to be garbage-collected
  - release does not usually happen right away
  - memory can be released when last use is gone
  - interpreter keeps track of what objects are

  Destruction & garbage collection

  - no guarantee that a finalizer will ever be called
  - other resources, close files, etc.
  - can define a finalize() method for a class to reclaim

  - Java has no destructor (unlike C++)
    - can define a finalize() method for a class to reclaim
  - has no destructor (unlike C++)
  - order doesn't matter
I/O and file system access

- import java.io.*
- byte I/O
- character I/O
- file access
- buffering
- exceptions

Character I/O

- BufferedReader
- BufferedWriter
- InputStreamReader
- OutputStreamWriter

In general, use character I/O classes because it has a readLine method for speed.

- use BufferedReader and BufferedWriter
- InputStreamReader and OutputStreamWriter

BufferedReader and BufferedWriter

I/O and file system access

import java.io.*
Exceptions

Exceptions are the Java solution (also in C++)

Exceptions indicate unusual condition or error

• 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

Exceptions

Exceptions

- C-style error handling
  - null from library functions like fopen()
  - return a special value from functions, e.g.
  - ignore errors -- can't happen
  - limited set of possible return values
- limited set of possible return values
  - repeated code or goto's to share code
  - error handling mixed with computation
  - leads to complex logic

• some functions return all possible values
• no possible error return value is available
• leads to complex logic

Exceptions

Exceptions

- NULL from library functions like fopen()
  - i from system calls like open()
- return a special value from functions, e.g.
  - C-style error handling
  - leads to complex logic

Exceptions

Exceptions

- C-style error handling
  - null from library functions like fopen()
  - return a special value from functions, e.g.
  - ignore errors -- can't happen
  - limited set of possible return values

Exceptions

Exceptions

- some functions return all possible values
- no possible error return value is available
- leads to complex logic

Exceptions

Exceptions

- C-style error handling
  - null from library functions like fopen()
  - return a special value from functions, e.g.
  - ignore errors -- can't happen
  - limited set of possible return values

Exceptions

Exceptions

- some functions return all possible values
- no possible error return value is available
- leads to complex logic

Exceptions

Exceptions

- C-style error handling
  - null from library functions like fopen()
  - return a special value from functions, e.g.
  - ignore errors -- can't happen
  - limited set of possible return values

Exceptions

Exceptions

- some functions return all possible values
- no possible error return value is available
- leads to complex logic

Exceptions

Exceptions

- C-style error handling
  - null from library functions like fopen()
  - return a special value from functions, e.g.
  - ignore errors -- can't happen
  - limited set of possible return values
• A method can catch exceptions
  ```java
  public void foo() {
      try {
          // if anything here throws an IO exception
          // or a subclass, like FileNotFoundException
      } catch (IOException e) {
          // this code will be executed
          // if anything here throws an exception
      }
  }
  ```

• A method must list exceptions it can throw
  ```java
  public void foo() throws IOException {
      // if anything here throws an exception
      // foo will throw an exception
      // to be handled by its caller
  }
  ```

• Exceptions can be thrown implicitly or explicitly

Why exceptions?

- Reduced complexity
  - if a method returns normally, it worked
  - if a method returns normally, it worked

- Error code grouped in a single place
  - if the try exits normally, all the code in it worked
  - statements worked, without explicit tests

- Can't unconsciously ignore possibility of errors
  - have to at least think about what exceptions can be thrown

- Or it can throw them, to be handled by caller
  ```java
  public static void main(String args[]) throws IOException {
      int b;
      while ((b = System.in.read()) >= 0)
          System.out.write(b);
  }
  ```
String methods

• a String is 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)
  – substring(start, len)

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
  fld = st.nextToken...

"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();
  }
}

...
Java vs. C and C++

- In language, not a library add-on
- Threads for parallelism within a single process
  ```
  try { catch(Exception) }
  ```
  - Exception: try { catch(Exception) }
  - Labelled break and continue instead of goto
- C-like control flow, but
- Strings are more or less built in
  ```
  int [] a = new int[100];
  ```
  - Arrays are dynamically allocated
  - References instead of pointers for objects
  - Static member function applies to whole class
  - Objects all derived from Object class
  - Everything is part of some class
  - Really object-oriented
- Boolean: true or false
- Character: unsigned 16-bit Unicode character
- Byte, short, int, long: signed integers (no unsigned)
- sizeof, order of evaluation are specified
- C-like basic types, operators, expressions
  - Constants use static final declaration
  - Import instead of #include

Java vs. C and C++