Java in 21 minutes

- hello world
- basic data types
- classes & objects
- program structure
- constructors
- garbage collection
- I/O
- exceptions
- Strings
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

```java
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
  - 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
2 versions of `echo`

```java
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 echol {
    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);
    }
}
```

- arrays have a length field `a.length`
  - subscripts are always checked
- Strings have a length() function `(s.length())`
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)

• private variables and functions are not accessible from outside the class

• not possible to determine HOW the operations are implemented, only WHAT they do
Classes & objects

• in Java, everything 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) {...}
    // or matchhere(String re, int ri, String text, int ti)
}

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

• all objects are created dynamically
• 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();
Constructors: making a new object

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

```java
RE r;
r = new RE(s);
```

- "this" is the object being constructed or running the code

- can use multiple constructors with different arguments to construct in different ways:

```java
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

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

    public RE(String re) {
        num_REs++;
    }
}
```

• class methods
  - most methods associated with an object instance
  - if declared static, associated with class itself
  - e.g., main()
Program structure

- typical structure is

```java
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)) ...
    }
}

- order doesn't matter
```
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
I/O and file system access

- import java.io.*

- byte I/O
  - InputStream and OutputStream

- character I/O (Reader, Writer)
  - InputReader and OutputStream
  - InputStreamReader, OutputStreamWriter
  - BufferedReader, BufferedWriter

- file access
- buffering
- exceptions

- in general, use character I/O classes
Character I/O

- InputStreamReader reads Unicode chars
- OutputStreamWriter write Unicode chars

- use Buffered(Reader|Writer)
  - for speed
  - because it has a readLine method

```java
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);
        }
    }
}
```
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(); // required!!!
        } catch (Exception e) {
            System.err.println("IOException "+ e);
        }
    }
}
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 strerror: 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

```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
        // 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

```java
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

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

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

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

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

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

- **String parsing**

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

StringTokenizer st = new StringTokenizer(str)
while (st.hasMoreTokens()) {
  String s = 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)

```java
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++

• 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