

Java history

- **invented mainly by James Gosling (Sun Microsystems)**
- **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 in a browser instead of a microwave**
 - renamed "Java"
 - Java Virtual Machine (JVM) runs in browser
- **1994: Netscape supports Java in their browser**
 - enormous hype: a viable threat to Microsoft
- **1997-2002: Sun sues Microsoft multiple times over Java**
 - MSFT guilty of anti-competitive actions
 - mostly settled by 4/04
- **significant language changes in Java 1.5 (9/04)**
 - generics, auto box/unbox, for loop, annotations, ...

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
- **object-oriented**
 - everything is part of some class
 - objects all derived from **Object** class
 - klunky mechanisms for converting basic \leftrightarrow object
- **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 is now null
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**

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.printf("%3d %7.2f\n"
                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 (an Object)**

- "." is a String
- holds 16-bit Unicode chars, NOT bytes
- does NOT have a null terminator
String.length() returns length
- + is string concatenation operator; += appends
- immutable: string operations make new strings

Classes & objects in Java

- **everything** is part of some object
 - all classes are derived from class Object

```
public class RE {
    String re;        // regular expression
    public RE(String regexp) {...} // constructor
    public boolean match(String text) {...}
    public int start() {...}
    public int end() {...}

    boolean matchhere(String re, String text) {...}
    // ...
}
```

- **member functions defined inside the class**
 - internal functions shouldn't be public (e.g., matchhere)
 - variables shouldn't ever be public
- **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
boolean m = re.match("abracadabra");
```

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

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)
```

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 does garbage collection
- **no control over when this happens**
 - can set object reference to **null** to encourage it
- **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 does not seem like a great design

I/O and file system access

- `import java.io.*`
- **byte I/O**
 - `InputStream` and `OutputStream`
- **character I/O (Reader, Writer)**
 - `InputStreamReader` and `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

Buffered byte I/O to/from files

```
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();
    }
}
```

- buffering required; too slow otherwise

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

```
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  
    } finally {  
        // this is done regardless  
    }  
}
```

- 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 any kind of IO 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**
 - ~100,000 characters
- **Java supports Unicode**
 - **char** data type is 16 bits
 - **String** data type is 16-bit Unicode chars
 - **\uhhhh** is Unicode character hhhh (h == hex digit)
 - use in `"..."` and `'.'`