Java history

• invented mainly by James Gosling ([formerly] Sun Microsystems)
• 1990: Oak language for embedded systems
  - needs to be reliable, easy to change, retarget
  - efficiency is secondary
  - implemented as interpreter, with virtual machine
• 1993: renamed "Java"; use in a browser instead of a microwave
  - 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 found 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 1.6 (== 6.0) 12/06 is mostly incremental changes
  - Java 1.7 (7/11) seems to be as well
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 <-> object
- references instead of pointers for objects
  - null references, garbage collection, no destructors
  - == is object identity, not content identity
- all arrays are dynamically allocated
  ```java
  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
Basic data types

- Java tries to specify some of the unspecified or undefined parts of C and C++

- 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`

- **member functions & variables defined inside class**
  - internal functions should not be public, variables should never be public

- **every object is an instance of some class**
  - created dynamically by calling `new`

- **class variable: a variable declared static in class**
  - only one instance in whole program, exists even if 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++;
        ...  
    }
    public static int RE_count() {
        return num_REs;
    }
}
```
Class methods

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

class RE {
    String 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 (r1.equals(r2)) ... // member function
        if (equals(r1, r2)) ... // static function
        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
Scope and visibility

• only one public class per file
  – public class hello {} has to be in file 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;  // etc.
  }
  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

• byte I/O for raw data
  - read(), write(), InputStream, OutputStream

• character I/O for Unicode (Reader, Writer)
  - InputReader and OutputWriter
  - InputStreamReader, OutputStreamWriter
  - BufferedReader, BufferedWriter

• byte-at-a-time I/O
  - System.in, .out, .err like stdin, stdout, stderr
  - read() returns next byte of input, -1 for end of file
  - any error causes an I/O Exception

import java.io.*;

public class cat1 {
    public static void main(String args[]) throws IOException {
        int b;
        while ((b = System.in.read()) != -1) {
            System.out.write(b);
        }
    }
}
Buffered byte I/O to/from files

- buffering is usually required; too slow otherwise

```java
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();
    }
}
```
Character I/O (char instead of byte)

- use a different set of functions for char I/O
- works properly with Unicode (\u1234 literals)
- InputStreamReader adapts from bytes to chars
- OutputStreamWriter adapts from chars to bytes
- use Buffered(Reader|Writer) for speed

```java
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
  - ~110,000 characters today

• UTF-16: 16 bit internal representation
  - encodes all characters used in all languages
    numeric value, name, case, directionality, ...
  - expansion mechanism for > $2^{16}$ characters

• UTF-8: byte-oriented external form
  - variable-length encoding, self-synchronizing within a couple of bytes
  - ASCII compatible: 7-bit characters occupy 1 byte
    
    \[
    \begin{align*}
    00000000 & \rightarrow 0bbbbbbb \\
    00000bbb & \rightarrow 110bbbb 10bbbbbb \\
    bbbbbbbb & \rightarrow 1110bbbb 10bbbbbb 10bbbbbb \\
    \end{align*}
    \]
  - analogous longer encoding for chars in extended set

• Java supports Unicode
  - char data type is 16-bit Unicode
  - String data type is 16-bit Unicode chars
  - \uhhhh is Unicode character hhhh (h == hex digit); use in "..." and '.'
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
    so no possible error return value is available for use
- **exceptions are the Java solution (also in C++)**
- an 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
        } finally {
            // this is done regardless
        }
    }
    // 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

```java
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 previous statements worked, without explicit tests
  - if the `try` exits normally, all the code in it worked
  - error code is 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);
}
```

- **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?