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 over time**
  - Java 1.5 (9/04) generics, auto box/unbox, for loop, annotations, ...
  - Java 1.8 (3/14) lambdas / closures

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**

```
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) {…} finally {…}`
- **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

Unicode (www.unicode.org)

- **universal character encoding scheme**
  - ~113,000 characters
- **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

```
0bbbbbbb
110bbbbb 10bbbbbb
1110bbbb 10bbbbbb 10bbbbbb
11110bbb 10bbbbbb 10bbbbbb 10bbbbbb
```

- **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 '.'

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

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

    so no possible error return value is available for use
- **exceptions are the Java solution (also in C++, Python, …)**
- **an 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 or dealt with**
  - 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
}
```

How exceptions help

```
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

- **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?

Virtual functions

- **in Java, all functions are implicitly *virtual***
- **if a reference to a superclass type is really a reference to a subclass object, a function call with that reference calls the subclass function**
- **polymorphism: proper function to call is determined at run-time**
  - *e.g.*, drawing Shapes in an array:

```
draw(Shape[] sa) {
  for (int i = 0; i < sa.length; i++)
    sa[i].draw();
}
```

- **virtual function mechanism automatically calls the right draw() function for each object**
  - a subclass may provide its own version of this function, which will be called automatically for instances of that subclass
  - the superclass can provide a default implementation
- **the loop does not change if more subclasses of Shapes are added**

Interfaces

- **an interface is like a class**
- **declares a new data type**
- **only declares methods (not implementations) and constants**
  - methods are implicitly **`public`**
  - constants are implicitly **`public static final`**

- **any class can implement the interface**
  - i.e., provide implementations of the interface methods
  - and can provide other methods as well
  - and can implement several interfaces

```
class foo implements bar {
   // implementation of bar methods
}
```

- **the only way to simulate function pointers and function objects**

Comparison interface for sorting

```
interface Cmp {
   int cmpf(Object x, Object y);
}
class Icmp implements Cmp {  // Integer comparison
   public int cmpf(Object o1, Object o2) {
      int i1 = ((Integer) o1).intValue();
      int i2 = ((Integer) o2).intValue();
      if (i1 < i2) return -1;
      else if (i1 == i2) return 0;
      else return 1;
   }
}
class Scmp implements Cmp {  // String comparison
   public int cmpf(Object o1, Object o2) {
      String s1 = (String) o1;
      String s2 = (String) o2;
      return s1.compareTo(s2);
   }
}
```

- **whole lot of casting going on**
- **can't do an illegal cast, but don't find out till runtime**

Sort function using an interface

```
void sort(Object[] v, int left, int right, Cmp cf) {
   int i, last;

   if (left >= right)  // nothing to do
      return;
   swap(v, left, rand(left,right));
   last = left;
   for (i = left+1; i <= right; i++)
      if (cf.cmpf(v[i], v[left]) < 0)
         swap(v, ++last, i);
   swap(v, left, last);
   sort(v, left, last-1, cf);
   sort(v, last+1, right, cf);
}

Integer[] iarr = new Integer[n];
String[] sarr = new String[n];
Quicksort.sort(iarr, 0, n-1, new Icmp());
Quicksort.sort(sarr, 0, n-1, new Scmp());
```

Wrapper types

- **most library routines work only on Objects**
  - don't work on basic types like int
- **have to "wrap" basic types in objects to pass to library functions, store in Vectors, etc.**
  - Character, Integer, Float, Double, etc.
- **wrappers also include utility functions and values**

```
Integer I = new Integer(123); // constructor
int i = I.intValue();         // get value
i = Integer.parseInt("123");  // atoi
I = Integer.valueOf("123");   // …
String s = I.toString();

Double D = new Double(123.45);
double d = D.doubleValue();
d = Double.parseDouble("123.45");  // atof
D = Double.valueOf("123.45");      // ...
String s = D.toString();

double atof(String str) { return Double.parseDouble(str); }
System.out.println(Double.MAX_VALUE);
```

Boxing and unboxing

- **Java 1.5 autobox and unbox somewhat clean up this mess**

```
Integer I = 123; // no need for new Integer()
int i = I;       // no need for I.intValue()
String s = I.toString();

Double D = 123.45;
double d = D;
d = Double.parseDouble("123.45");  // atof
D = Double.valueOf("123.45");
s = D.toString();
```

Collections and collections framework

- **"collection" == container in C++, etc.**
  - Set, List (includes array), Map
- **interfaces for standard data types**
  - abstract data types for collections
  - can do most operations independently of real type
  - include standard interface for add, remove, size, member test, ...
- **implementations (concrete representations)**
  - HashSet, TreeSet
  - ArrayList, LinkedList
  - HashMap, TreeMap
- **algorithms**
  - standard algorithms like search and sort
  - work on any Collection of any type that provides standard operations like comparison
  - "polymorphic"
- **iterators**
  - uniform mechanism for accessing each element

Collections sort

- **ArrayList is an implementation of List**
  - like Vector but better
  - adds some of its own methods, like get()
- **Collections.sort is a polymorphic algorithm**
  - specific type has to implement Comparable

```
class qsort1 {
  public static void main(String[] argv) throws IOException {
    FileReader f1 = new FileReader(argv[0]);
    BufferedReader f2 = new BufferedReader(f1);
    String s;
    List al = new ArrayList();
    while ((s = f2.readLine()) != null)
       al.add(s);
    Collections.sort(al);
    for (int j = 0; j < al.size(); j++)
       System.out.println(al.get(j));
  }
}
```

Generics, for-each

- **generics tell compiler what type a Collection holds**
  - compiler can do more type checking at compile time
- **for-each loop cleans up iterator code**

```
String s;
List<String> al = new ArrayList<String>();
while ((s = f2.readLine()) != null)
   al.add(s);
Collections.sort(al);
for (String j : al)
   System.out.println(j);
```

- **<?> as a type in a generic matches any type**

- **<? extends T> matches any type that extends T**
  - "bounded wildcard"

Interface example: map

- **interface defines methods for something**
- **says nothing about the implementation**

```
interface Map
    void put(String name, String value);
    String get(String name);
    // ...
}
```

- **classes implement it by defining functions**
- **have to implement all of the interface**

```
class Hashmap implements Map {
    Hashtable h;
    Hashmap() { h = new Hashtable(); }
    void put(String name, String value) {h.put(name, value); }
    String get(String name) { return h.get(name); }

class Treemap implements Map {
    RBTree t;
    Treemap() { t = new RBTree(); }
    void put(String name, String value) { … }
    String get(String name) { … }
```

Word frequency count: Java

```
public class freqhash {
  public static void main(String args[]) throws IOException {
    FileReader f1 = new FileReader(args[0]);
    BufferedReader f2 = new BufferedReader(f1);

    Map<String, Integer> hs = new HashMap<String,Integer>();
    String buf;
    while ((buf = f2.readLine()) != null) {
      String nv[] = buf.split("[   ]+");
      for (int i = 0; i < nv.length; i++) {
        Integer oldv = hs.get(nv[i]);
        if (oldv == null)
          hs.put(nv[i], 1);
        else
          hs.put(nv[i], oldv+1);
      }
    }
    for (String n : hs.keySet()) {
      Integer v = hs.get(n);
      System.out.println(n + " " + v);
    }
  }
}
```

Word frequency count: C++ STL

```
#include <iostream>
#include <map>
#include <string>

int main() {
    string temp;
    map<string, int> v;
    map<string, int>::const_iterator i;

    while (cin >> temp)
        v[temp]++;
    for (i = v.begin(); i != v.end(); ++i)
        cout << i->second << " " << i->first << "\n";
}
```

Sorting: Java v. C++

```
String s;
List<string> al = new ArrayList<string>();
while ((s = f2.readLine()) != null)
    al.add(s);
Collections.sort(al);
for (String j : al)
    System.out.println(j);
```

```
string tmp;
vector<string> v;
while (getline(cin, tmp))
    v.push_back(tmp);
sort(v.begin(), v.end());
copy(v.begin(), v.end(),
        ostream_iterator<string>(cout,"\n"));
```