Java: simple, object-oriented, distributed, interpreted, robust, secure, architecture
neutral, portable, high-performance, multithreaded, and dynamic language.”
— Sun

Java: The Language

Hello World!

```
/** Application HelloWorld */
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

% javac HelloWorld.java
% java HelloWorld
Hello World!

Java vs. C++

```
/** Application HelloWorld */
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Unlike C++, Java has....

- No global functions — everything is in a class!
- Real String objects — not just char[]
- No pointers — everything is a reference
- No user operator overloading
- No preprocessor — cpp not needed
- unicode instead of ascii
**Command Line Arguments**

```java
/** Application PrintArgs
 * prints the command line arguments */
public class PrintArgs {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++)
            System.out.println(args[i]);
    }
}
```

---

**Java vs. C++, Revisited**

```java
Ball ball = new Ball(50, 50);
PinballAnimationPane pap = new PinballAnimationPane();
pap.addObject(ball);
ball.animate();
```

```java
Ball *pball = new Ball(50,50);
PinballAnimationPane *pxpap = new PinballAnimationPane();
pxpap->addObject(pball);
pball->animate();
```

```java
Ball ball(50,50); // creates ball on stack
PinballAnimationPane xpap(); // creates xpap on stack
xpap.addObject(ball); // calls: addObject(Ball &b);
ball.animate();
```

---

**Primitive types in Java**

- boolean
- char (16-bit) //unicode
- byte (8-bit signed)
- short (16-bit signed)
- int (32-bit signed)
- long (64-bit signed)
- float (32-bit signed)
- double (64-bit signed)

**Floating point types**

---

**Java's Hybrid Object Model**

- Primitive types on stack
  - May be *wrapped* or *boxed* into a real object
    - Integer an Integer = new Integer(43);
      - (useful for storing in java.util.*'s collections)
  - Unboxed primitives very similar to in C++
  - All object instances live in the heap (not stack)
    - all object creation is done with new
  - No "delete" — Java uses garbage collection, but also provides finalize() method

---

**Java's Class Hierarchy**

- String
- Boolean
- Number
- Compiler
- Component
- Container
- Panel
- Applet
Java Documentation

HelloWorld Applet

```
import java.applet.*; import java.awt.*

public class HelloWorldApplet extends Applet {
    static final String message = "Hello World";
    private Font font;

    public void init() { // one-time initialization
        font = new Font("Helvetica", Font.BOLD, 48);
    }

    public void paint(Graphics g) {
        g.setColor(Color.yellow);
        g.fillOval(10, 10, 330, 100);

        g.setColor(Color.red);
        g.drawOval(10, 10, 330, 100);
        g.drawOval(9, 9, 332, 102);
        g.drawOval(8, 8, 334, 104);
        g.drawOval(7, 7, 336, 106);

        g.setColor(Color.black);
        g.setFont(font);
        g.drawString(message, 40, 75);
    }
}
```

Running the HelloWorld Applet

```
<APPLET code="HelloWorldApplet.class" width=350 height=120> Java Missing
</APPLET>
```

Methods: A Closer Look

```
public class Point {
    public void move(int dx) {
        x += dx;
        moved();
    }

    private void moved() { .. }

    private int x, y;
}
```

More on Methods

- Instance methods (no `static` keyword)
  - have implicit `this` argument
  - can use `super` keyword
  - no need to use `->` operator as in C++
    - just `.` operator since this, `super` are references
- `static` (class) methods
  - do not have implicit `this` argument
  - cannot use the `super` keyword

Default Arguments

```
public class Point {
    public Point() {
        this(0,0);
    }

    public Point(int x, int y) {
        this.x = x; this.y = y;
    }

    public void move() {
        moved();
    }

    public void move(int dx) {
        x += dx;
        moved();
    }

    private int x, y;
}
```

Note: two different x's and y's
"Override" vs. "Overload"

- **Override**
  - replace a superclass's method with a specialized version
  - signatures must match
    (including return type; C++ permits narrowing of return types, Java does not)

- **Overload**
  - write several methods for a given class with the same name
  - language can disambiguate based on number or types of arguments

Java's Class Hierarchy

Objects and Identities

Test object identity:
- `ball == sameBall` ➞ true

Test object value's equality:
- `ball.equals(sameBall)` ➞ true

Cloning Objects

Test object identity:
- `ball == anotherBall` ➞ false

Test object value's equality:
- `ball.equals(anotherBall)` ➞ true
### Changing a Ball

```java
testBall.setPosition(20, 35);
```

Test object identity:
- `ball == sameBall` ⇒ `???
- `ball == anotherBall` ⇒ `???

Test object value's equality:
- `ball.equals(sameBall)` ⇒ `???
- `ball.equals(anotherBall)` ⇒ `???
```

### Inequality in Balls!

```java
ball.setPosition(20, 35);
sameBall anotherBall
```

Test object identity:
- `ball == sameBall` ⇒ `true`
- `ball == anotherBall` ⇒ `false`

Test object value's equality:
- `ball.equals(sameBall)` ⇒ `true`
- `ball.equals(anotherBall)` ⇒ `false`
```

### Assignment just changes the pointer

```java
ball = sameBall = anotherBall;
sameBall anotherBall
```

Test object identity:
- `ball == sameBall` ⇒ `false`
- `ball == anotherBall` ⇒ `false`

Test object value's equality:
- `ball.equals(sameBall)` ⇒ `false`
- `ball.equals(anotherBall)` ⇒ `false`
```

### Java variables hold...

- **primitive**
  - `boolean foo;` // boolean, not bool as in C++
  - `char aChar = 'a';` // 16 bit char (unicode)

- **Object reference (may be null)**
  - ColoredBall cball = new ColoredBall();
  - Ball ball = cball;

- **Array reference**
  - `int[] intArray = { 1, 2, 3, 4, 5, };`
  - `String[] strArray = { "Hello", "World", };`
    // same as
  - `String[] strArray = new String[2];`
  - `strArray[0] = new String("Hello");`
  - `strArray[1] = new String("World");`

### Arrays

- Java arrays are 1st-class Objects
- 0-indexed
- Bounds checking performed
- Store/Retrieve using [] operator
- Have implicit length field
  ```java
  String[] strArray;
  strArray.length = 2;
  ```
```

### 2-d and 3-d Arrays

- No special language support for 2-d arrays -- just make an array of arrays
  ```java
  public class myArray {
  public static void main (String[] args) {
  double[][] mat = {{1., 2., 3., 4.}, {5., 6., 7., 8.},
  {9., 10., 11., 12.}, {13., 14., 15., 16.}};
  for (int y = 0; y < mat.length; y++) {
  for (int x = 0; x < mat[y].length; x++)
  System.out.print(mat[y][x] + " ");
  System.out.println();    }}
  ```
Strings

- The `String` class provides read-only strings and supports operations on them.
- A `String` can be created implicitly either by using a quoted string (e.g., "HUB food") or by the concatenation of two `String` objects, using the `+` operator.

Strings are Immutable

Since you cannot modify existing strings, there are methods to create new strings from existing ones.

- `public String substring(int beginIndex, int endIndex)`
- `public String replace(char oldChar, char newChar)`
- `public String concat(String str)`
- `public String toLowerCase()`
- `public String toUpperCase()`
- `public String trim()`

Identifiers

- Everything has a globally-unique name.
  - `Java.lang.String`  
  - `Java.util.Hashtable`  
  - `Java.applet.Applet`  
  - `EDU.Washington.grad.gjb.cassowary.Variable.toString()`
  - Pretty wordy, so...

import statement

- Two forms:
  - `import java.util.HashTable;`
    - Just make the `HashTable` class available from package `java.util`
  - `import EDU.Washington.grad.gjb.cassowary.*;`
    - Make all classes from package available on demand
- Always an implicit "import java.lang.*"
- Permits using simple (short) names

How Java Finds a Class...

- Package names mirror the directory structure.
- `package` statement informs the compiler.

Compilation of Source File

```bash
% ls
Variable.java  
% javac Variable.java
% ls
Variable.java  
Variable.class  
Helper.class
```

One java source file may create multiple class files containing the byte-compiled code.
Class Access Protection

- Only one public class per file
- No specifier ⇒ package protection
  visible to all classes in the package
  no "package" keyword — remember it is a statement

Java Accessibility vs. C++

- No "friend" keyword
- Every field or method has an access specifier
  (no "public:" sections)
- Default is package-visibility which has no
  associated keyword (not private)

Final Fields

- final fields correspond to C++'s "const"
- final fields cannot be changed once initialized
- cannot use final in function signatures
  (less flexible than C++'s const is an unused reserved word in Java)

Private: most restrictive access modifier

- Only one public class per file
- No specifier ⇒ package protection
- visible to all classes in the package
  no "package" keyword — remember it is a statement

No Need for Forward Declarations

- public class Point {
  private int x, y;
  void setXY(int x, int y) {
    this.x = x; this.y = y;
  }
  protected void move(int x, int y) {
    setXY(this.x+x, this.y+y);
  }
  public int getX() { return x; }
  public int getY() { return y; }
}

Ball and CBall Example

- public final class Circle {
  private final double MY_PI = 3.1415;
  public double area() { return MY_PI * r*r; }
}
  
  final fields correspond to C++'s const
  final fields cannot be changed once initialized
  cannot use final in function signatures
  (less flexible than C++'s const is an unused reserved word in Java)

- package BallExample;
  public class Ball implements Bounceable {
    private int x, y;
    public Ball(int x, int y) {
      this.x=x; this.y=y;
    }
    public void Bounce() {
      System.err.println("Ball bounces");
    }
  }
  
  - package BallExample;
    public class CBall extends Ball {
      private int colorSelector;
      public CBall(int x, int y) {
        super(x,y);
        colorSelector = 0; // for black
      }
      public void Bounce() {
        System.err.println("CBall bounces");
      }
    }
  
  - package BallExample;
    public class Circle {
      private PointColor c;
      public double area() { return MY_PI * r*r; }
    }

Inheritance Mechanisms

- **extends superclass**
  - similar to "public" in C++
  - for expressing an "is-a" relation
- **implements superinterface**
  - similar in use to C++’s multiple inheritance
  - for expressing an "is-capable-of" or "knows-how-to" relation

Accessing Inherited Methods

- As previously discussed, the keyword **this** refers to the object on which the method was invoked (even if the method itself was found by chasing up the superclass hierarchy).
- The keyword **super** functions similarly, except that the method lookup starts in the **superclass** of the class in which the method was found.

Java Interfaces

- Interfaces can only specify public methods
- Similar to protocols in Smalltalk
- May be used as a type for a variable
- Can specify sub-interfaces and can extend multiple interfaces at a time

Bounceable Interface

- `public interface Bounceable {
  public void Bounce();
  private void BounceNow(); // error
}

Ball Example Output and Errors

- % java BallExample.BallTest
  Ball bounces
  CBall bounces
  CBall.ClassFn()
  Ball.ClassFn()

Types vs. Classes

- **Types**
  - variables have types
  - used for checking validity of method invocations
  - may be an **interface**
- **Classes**
  - objects (i.e., instances) have classes
  - used for dynamic dispatch (binding of non-static function call)
  - Each class has a corresponding type — that hierarchy of types mirrors the class hierarchy
Multiple Inheritance in Java

- A Java class can extend (subclass) another class and implement multiple interfaces

```
public class TopLevelWindow extends Window
    implements Drawable, Cloneable, Streamable {
    ...
}
```

Abstract Methods and Abstract Classes

- Abstract methods correspond to C++’s “pure virtual functions” (But C++ uses “=0” syntax, and permits an implementation)
- Abstract methods must be overridden in concrete subclasses
- Only abstract classes can have abstract methods (C++ infers abstract classes, Java requires you mark the class explicitly)

```
// Note abstract keyword is used for the class, too
public abstract class Shape {
    public abstract void rotate(int); // no definition
    public abstract double area(); // no definition
}
```

Final Methods

```
public class Circle {
    private double r; // radius
    ...
    public final double area() { return Math.PI * r*r; }
}
```

- Final methods cannot be overridden
- Final methods may be inlined (no “inline” keyword)
- Similar to non-virtual member functions in C++ (but those can be overridden, they just do not dispatch dynamically)

Final Classes

```
public final class Circle {
    private double r; // radius
    ...
    public double area() { return Math.PI * r*r; }
}
```

- Final classes cannot be subclassed — they are leafs in the class hierarchy
- Methods in final classes are implicitly final
- Provides compiler with optimization opportunities

try { throw } and catch, finally (exceptions)

```
class ExceptionExample {
    static public void main(String args[]) {
        try {
            // allocate some resource (besides memory)
            doSomething();
            if (!FThingsAreOkay()) {
                throw new RuntimeException("Things not ok");
            }
            doSomethingElse();
        } catch (RuntimeException e) {
            System.err.println("Runtime Exception: " + e);
        } catch (Exception e) { // similar to "catch (...)" in C++
            System.err.println("Exception: " + e);
        } finally { // finally is not in C++
            // cleanup resource
        }
    }
}
```

Exception Hierarchy
Threads

```java
public class Pendulum extends Applet implements Runnable {
    private Thread myThread;
    public void start() {
        if (myThread == null) {
            myThread = new Thread(this, "Pendulum");
            myThread.start();
        }
    }
    public void run() {
        while (myThread != null) {
            try { myThread.sleep(100); }
            catch (InterruptedException e) { /* do nothing */ }
            myRepaint();
        }
    }
    public void stop() { myThread.stop(); myThread = null; }
}
```

Summary:
What Java Left Out from C++

- No stack objects, only heap objects
- No destructors, only `finalize()` method
- No pointers, everything is a reference
- No delete, garbage collector instead
- No const, only `final` (methods, fields, classes)
- No templates, no preprocessor
- No operator overloading
- No multiple inheritance of classes
- No enumerations or typedefs

Summary:
What Java Put In (vs. C++)

- Garbage collector
- Object-rooted, rich class hierarchy
- Strings, first-class arrays with bounds checking
- Package system with `import`
- `interface`, `implements`, `extends`, `abstract`
- `finally` blocks, static/instance initializers
- Secure and portable JavaVM, threads
- Dynamic reflection capabilities, inner classes
- JavaDoc system