Assumptions

- You already know C++
- You understand that this presentation is just a feature overview. Only a fraction of Java’s features are presented and we barely scratch the surface of the Java API.
Programs written in Java are executed on a **Java Virtual Machine (JVM)**

- Java can be run any platform for which a JVM has been implemented
  - “Write once, run anywhere”

- Java is compiled to an intermediate language called **bytecode**
  - Bytecode is either interpreted, instruction by instruction by the JVM (slow), or sent through a Just-in-time compiler (JIT) which translates some of it into machine code just prior to execution (much faster!)

- Code is written in `.java` files; These are converted into `.class` files (bytecode)
History

- Designed in the early 90’s by Sun Microsystems (now part of Oracle)

**Motivations:**

- Provide an alternative to C++ which reduced developer errors:
  - Cleaner syntax (no pointers!)
  - Garbage collection vs. manual memory management
  - Pure object-oriented language (no global code or data)

- Execute on a wide range of devices
- Execute code directly on a web page
  - Java Applets (deprecated); now Java Web Start

- Since 2008: Primary language for apps on Android
Comparison with other Languages

  - 1-4 times slower than C/C++
  - Approximately the same as other JIT compiled languages such as C#
  - Much faster than pure interpreted scripting languages such as Perl, Python, and Ruby

- Development time:
  - Twice as fast as C++ (from “Thinking in Java”)
  - Slower than scripting languages
  - (Hard to find an objective source for this information)
The two components of the Java Platform are the JVM and the Java API. The API provides a massive set of classes for numerous applications:

- String processing
- Data structures
- Networking
- Handling media files (images, video, audio, ...)
- Graphical User Interfaces (GUI): AWT, Swing, and JavaFX
- ...etc
Everything is an Object

In Java, there is no code that exists outside of a class. Even the main method must appear within a class:

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

<table>
<thead>
<tr>
<th>Similar to C++</th>
<th>Different from C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes are similarly defined (although no .h and .cpp separation)</td>
<td><strong>main</strong> exists within a class.</td>
</tr>
<tr>
<td><strong>public</strong> has roughly the same meaning, although here it is used twice for both the class and the main method</td>
<td>There is a standard <strong>String</strong> class</td>
</tr>
</tbody>
</table>
public class Point {
    private double x, y;

    /* Constructor. */
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() { return x; }
    public void setX(double inX) { x = inX; }
    // ...
A Point Class in C++

class Point {
private:
    double x, y;
public:
    /* Constructor. */
    Point(double x, double y) {
        this->x = x;
        this->y = y;
    }

    double getX() { return x; }
    void setX(double inX) { x = inX; }
    // ...
};

In C++ we can implement methods within the .h file or the .cpp file. In Java there is only the .java file.
// No including type stuff required (yet)

public class TestPoint {
    public static void main(String[] args) {
        // Objects are always constructed on heap
        Point p = new Point(4, 10);

        // p is called a reference variable
        p.setX(5);

        // String concatenation with +
        System.out.println("x: " + p.getX());
    }
}

Andrew Vardy  Brief Intro. to Java: Part 1
#include <iostream>
#include "Point.h"
using namespace std;

int main(int argc, char **argv) {

    // Here we construct the object on the stack.
    Point p(4, 10);

    // Calling a public method
    p.setX(5);

    // Using the overloaded "<<" to concatenate.
    cout << "x: " << p.getX() << endl;
}

Variables: Primitive Types

All variables are either primitive types or references.

Primitive Types (most common in bold):

- byte, short, **int**, long, float, **double**, **boolean**, char.
- **int**: 32-bit integers from -2,147,483,648 to 2,147,483,647
- **double**: 64-bit rational numbers
  - 15 significant decimal digits; range of about $\pm 1.7 \times 10^{308}$
- **boolean**: Boolean values written as true or false
- **char**: 16-bit characters
public class Primitives1 {
    public static void main(String[] args) {
        // Declare and utilize as in C++.
        int i = 4;
        i++;
        System.out.println("i: " + i);
        // Error to use an uninitialized value!
        // double x;
        // System.out.println("x: " + x);
        // Logic and comparison.
        boolean a = false;
        boolean output = a && (i < 100);  // Lazy!
        System.out.println("output: " + output);
    }
}
Initialization of Primitive Data Members

Primitive members initialized to 0 (false for booleans).

```java
public class Primitives2 {
    private int i, j; // Will be initialized to 0
    private long k = 12;

    public Primitives2() {
        j = 7; // j was already initialized to 0.
    }

    public void printOut() {
        System.out.println("i: " + i + ", j: " + j "+ ", k: " + k);
    }

    public static void main(String[] args) {
        Primitives2 p2 = new Primitives2();
        p2.printOut(); // A method call!
    }
}
```
Conversion between Primitive Types

You can convert between types where the appropriate widening relationships exist:

long >>> int >>> short >>> byte    double >>> float

```java
public class Primitives3 {
    public static void main(String[] args) {
        // Maximum values for each type.
        byte b = 127;
        long l = 9223372036854775807L; // Suffix l
        // l = b;
        // b = l - 1; // Can't do this

        // If you know the loss of precision is acceptable you can cast between types.
        b = (byte) (l - 1);
    }
}
```
Conversion Issues

int: 10 digits max., long: 19 digits max.
float 6-7 significant digits, double 15 significant digits

```java
public class Primitives4 {
    public static void main(String[] args) {
        long l = 9223372036854775807L;
        float f = 2.0F;  // Use suffix F for floats

        // Loss of precision!
        f = l;
        System.out.println("l: " + l);
        System.out.println("f: " + f);

        // System.out.println("l == f: " + (l == f));
        // The long is converted to a float
        // in the comparison, so the result is true!
    }
}
```
public class ControlFlow {
    public static void main(String[] args) {
        // Loops, if-else, statements, all as C++
        for (int i=0; i<3; i++)
            System.out.println("i: " + i);
        // But the conditions for these statements
        // only accept booleans!
        int i = 3;
        // while (i) {
        while (i > 0) {
            System.out.println("i: " + i);
            i--;
        }
    }
}
Reference Variables

Objects are referred to through reference variables, which are essentially pointers without the horrible syntax.

```java
public class References {
    public static void main(String[] args) {
        // Declare a reference to a new Point
        Point a = new Point(0, 0);

        // Another reference 'pointing' at
        // the same object
        Point b = a;

        //
        b.setX(42);
        System.out.println("a.getX(): " + a.getX());
    }
}
```

Andrew Vardy | Brief Intro. to Java: Part 1
Local variables, including both primitive types and references live on the stack.

Objects (but not references) are allocated with `new` and live on the heap.

There is no `delete` keyword! When there are no references to an object remaining, the object becomes available to the garbage collector (GC).

- The GC uses its own logic to determine when to re-claim unused memory. Therefore, you should not make any assumptions about when your object is deleted.
- There is no destructor in Java, but there is a `finalize` method that is used in unusual circumstances to de-allocate memory allocated using a non-standard mechanism (e.g. via C++).
Garbage collection may occur when no references to an object remain. References can go away by going out of scope or by being explicitly set to `null`. (Aside: Uninitialized ref’s are set to `null`).

```java
public class GarbageCollection {
    public static void main(String[] args) {
        Point a = new Point(0, 0); //
        {
            Point b = a;
            {
                Point c = b;
                // Now three ref’s to object
            }
            // Now two
            //
            a = null; // Now just one
        }
        // No ref’s to object. It can be garbage
        // collected. (But don’t count on it!)
    }
}
```
Reference Equivalence vs. Object Equivalence

If a comparison operator such as `==` is applied to a reference variable, it is applied to the reference, not the object.

```java
public class RefEquiv {
    public static void main(String[] args) {
        Point p1 = new Point(2, 0);
        Point p2 = new Point(2, 0);
        Point alias1 = p1;

        // p1 and alias1 refer to same object
        System.out.println("p1 == alias1: " + (p1 == alias1));

        // p1 and alias1 refer to different objects
        System.out.println("p1 == p2: " + (p1 == p2));

        // Output:
        // p1 == alias1: true
        // p1 == p2: false
    }
}
```
The Java class hierarchy, including standard Java API classes and your classes, all inherit from the **Object** class. This provides several useful methods that can be applied to any object.

```java
public class ObjectExample {
    public static void main(String[] args) {
        Point p = new Point(2, 0);
        //
        System.out.println( p.toString() );
        // Output: Point@6d06d69c
        //
        Class pClass = p.getClass();
        System.out.println( pClass.getName() );
        // Output: Point
    }
}
```
Arrays in Java: (1) They are actual objects and have a public **length** data member; (2) Arrays of primitives are automatically initialized; (3) Out-of-bounds access generates an exception.

```java
public class Arrays1 {
    public static void main(String[] args) {
        // Array declaration and initialization
        int[] array = new int[10];

        // Array access. Also, use of length
        for (int i = 0; i < array.length; i++)
            assert array[i] == 0;

        // Java arrays check their index!
        int i = array[10]; // Exception thrown
    }
}
```
Creating an array of objects does not create the actual objects.

```java
public class Arrays2 {
    public static void main(String[] args) {
        String[] array = new String[3];
        // No actual strings have been created!

        // The for-each construct
        for (String s : array)
            System.out.println(s); // Prints null!

        for (int i=0; i<array.length; i++)
            array[i] = new String("string #" + i);
            // OR array[i] = "string #" + i;

        // Aggregate initialization is possible
        Point[] points = {
            new Point(1,1),
            new Point(2,2)
        };
    }
}
```
Two uses: (1) Refer to current object; (2) Call other constructor

```java
public class Rectangle {
    private int x, y, width, height;

    public Rectangle(int x, int y, int w, int h) {
        this.x = x;
        this.y = y;
        width = w; // 'this' not needed here
        height = h;
    }

    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }

    public Rectangle() {
        this(0, 0, 0, 0); // Unnecessary
    }
}
```