

Procedural Java

Requisite 'Hello World' in Java

```
1 public class HelloWorld {  
2     public static void main(String[] args) {  
3         System.out.println("Hello World");  
4     }  
5 }
```

Now what?

- How to compile?
 - Must specify the file(s)
- How to execute?
 - Must specify the class (without .class)
- Does the filename matter?
 - Must have '.java' suffix
 - Must be named the same as class

Primitives

- 8 types (9 including void)
 - `boolean` - one bit (but size undefined)
 - true or false
 - `byte` - signed 8 bits
 - (-128, 127)
 - `short` - signed 16 bits
 - (-32,768, 32,767)
 - `char` - unsigned 16 bits. Unicode character
 - from '\u0000' to '\uFFFF' or (0, 65,535)
 - `int` - signed 32 bits
 - (-2^{31} , $2^{31}-1$)
 - `long` - signed 64 bits
 - (-2^{63} , $2^{63}-1$)
 - `float` - single-precision 32-bit IEEE 754 floating point
 - 1.40129846432481707e-45 to 3.40282346638528860e+38
 - `double` - double-precision 64-bit IEEE 754 floating point
 - 4.94065645841246544e-324d to 1.79769313486231570e+308d

Where'd unsigned go?

Removed for simplicity.

What's this C code print?

```
1  #include<stdio.h>
2
3  int oldEnough(unsigned int age) {
4      return (age >= 18);
5  }
6
7  int main() {
8      int age = -1;
9      if (oldEnough(age)) {
10         printf("Old enough!\n");
11     } else {
12         printf("Not old enough\n");
13     }
14 }
```

Default Primitive Types / Specifying

```
1 // int and double are default integer / real number primitives
2 /* int */ foo = 10;
3 /* double */ bar = 10.0;
4
5 // to specify long or float use 'L' and 'f' respectively
6 /* long */ fool = 10L;
7 /* float */ barf = 10.0f;
8
9 // can also explicitly initialize with 'd' (or 'D')
10 // useful to force as a real number instead of integer
11 /* double */ food = 10d;
12
13 // nothing analogous for byte / short
14 /* short */ foos = (short) 10;
15 /* byte */ barb = (byte) 10;
```

Fun with Primitives

```
1  double nan = Double.NaN;
2  if (nan == Double.NaN) {
3      System.out.println("What I thought!");
4  } else {
5      System.out.println("WAT?!");
6  }
```

Fun with Primitives (cont)

```
1  int truthy = 1;
2  if (truthy) {
3      System.out.println("truthy");
4  } else {
5      System.out.println("falsy");
6  }
```


Variables

- Declaration
 - Define a variable with a name.
 - Multiple on a single line
- Initialization
 - Can be inline with declaration
 - Can be after declaration
 - Cannot reference a variable until initialized
- Naming
 - camelCase
 - only 'static final' use UPPER_SNAKE_CASE
 - Be descriptive with naming (don't use aBox as a name)

Fun with Variables

```
1  int foo, bar = 1;
2  if (System.currentTimeMillis() > 0L) {
3      foo = bar;
4  }
5  if (true) {
6      System.out.printf("%d%n", foo);
7  }
```

Fun with Variables (cont)

```
1  int bar = 1, foo = bar;
2  if (System.currentTimeMillis() > 0L) {
3      foo = bar;
4  }
5  if (true) {
6      System.out.printf("%d%n", foo);
7  }
```

Descriptive Variable Names

```
1 private static final double KILOMETER_TO_MILE = 0.621371d;  
2  
3 private static double convertKilometerToMile(double kilometers) {  
4     double miles = kilometers * KILOMETER_TO_MILE;  
5     return miles;  
6 }
```

Descriptive Variable Names (cont)

DO NOT DO - COUNTEREXAMPLE

```
1 public static final double radius = 6378.137d;
2
3 public double compute(double[] a, double[] b) {
4     double diff1 = b[0] - a[0];
5     double diff2 = b[1] - a[1];
6
7     double tmp1 = Math.sin(diff1 / 2);
8     double tmp2 = Math.sin(diff2 / 2);
9     double aa = (tmp1 * tmp1) + (tmp2 * tmp2 * Math.cos(a[0]) * Math.cos(b[1]));
10    double c = 2 * Math.atan2(Math.sqrt(aa), Math.sqrt(1 - aa));
11    return (radius * c);
12 }
```

Descriptive Variable Names (cont)

```
1 public static final double EARTH_RADIUS_KM = 6378.137d;
2
3 /**
4  * Use the Haversine (i.e., as crow flies) formula to calculate distance between two points.
5  *
6  * @param fromLat from-point's Latitude in radians
7  * @param fromLng from-point's Longitude in radians
8  * @param toLat to-point's Latitude in radians
9  * @param toLng to-point's Longitude in radians
10  * @return the Haversine distance between from and to points in kilometers.
11  */
12 public double haversineDistance(double fromLat, double fromLng, double toLat, double toLng) {
13     double deltaLat = toLat - fromLat;
14     double deltaLng = toLng - fromLng;
15
16     double deltaLatSin = Math.sin(deltaLat / 2);
17     double deltaLngSin = Math.sin(deltaLng / 2);
18
19     double squareHalfChordLength = (deltaLatSin * deltaLatSin) +
20                                     (deltaLngSin * deltaLngSin * Math.cos(fromLat) * Math.cos(toLat));
21     double angularDistance = 2 * Math.atan2(Math.sqrt(squareHalfChordLength),
22                                              Math.sqrt(1 - squareHalfChordLength));
23     return (EARTH_RADIUS_KM * angularDistance);
24 }
```

Operators

```
1  int foo = 10, bar = 2;
2
3  // PLUS
4  int result = foo + bar; // result = 12
5  // MINUS
6  result = foo - bar; // result = 8
7  // MULTIPLICATION
8  result = foo * bar; // result = 20
9  // DIVISION
10 result = foo / bar; // result = 5
11 // MODULOS
12 result = foo % bar; // result = 0
13
14 // CAVEAT on divide by 0
15 double food = 10d, bard = 0d;
16 bar = 0;
17 double resultd = food / bard; // result = NaN/Infinity
18 result = foo / bar; // Exception -> "java.lang.ArithmeticException: / by zero"
```

The rarest of keywords...strictfp

```
1  private static double getX(double a, double b, double c) {  
2      return a * b / c;  
3  }  
4  
5  private static strictfp double getY(double a, double b, double c) {  
6      return a * b / c;  
7  }  
8  
9  double a = 1.11d, b = 2.22d, c = 3.33d;  
10  
11  // what's the difference between x and y?  
12  double x = getX(a, b, c);  
13  double y = getY(a, b, c);
```


Increment / Decrement Operators

- Postfix increment / decrement operator
 - `a++; a--;`
- Prefix increment / decrement operator
 - `++a; --a;`

```
1  int a = 0, b = 0;
2  boolean equals = (a++ == ++b);
3  System.out.printf("%d and %d are equals? %s\n", a, b, equals);
```

Relational Operators

```
1  int a = 0, b = 1;
2
3  // equality
4  boolean equals = a == a;
5  // inverse equality - a not-equals b?
6  boolean notEquals = a != b;
7  // less than
8  boolean lessThan = a < b;
9  // greater than
10 boolean greaterThan = b > a;
11 // less than or equals
12 boolean lessThanOrEquals = a <= a;
13 // greater than or equals
14 boolean greaterThanOrEquals = b >= b;
15 // conditionals - and
16 boolean and = (a == a) && (a < b);
17 // conditionals - or
18 boolean or = (b < a) || (b > a);
```

Short Circuit

```
1  private boolean evaluateViaLookup(int id1, int id2) {  
2      int id1Result = loadFromDB(id1);  
3      int id2Result = loadFromDB(id2);  
4      return (id1Result > id2Result);  
5  }  
6  
7  int id1 = 1, id2 = 1;  
8  
9  // method `evaluateViaLookup` never called - saved a DB Lookup  
10 boolean equals = (id1 == id2) || evaluateViaLookup(id1, id2);
```

Ternary

```
1 private int maximum(int a, int b) {  
2     if (a > b) {  
3         return a;  
4     } else {  
5         return b;  
6     }  
7 }
```

Bitwise Operators

```
1  int foo = 1, bar = 2, foobar = 3;
2  // bitwise 'and'
3  int result = foo & bar; // result = 0
4  // bitwise 'or'
5  result = foo | bar; // result = 3
6  // bitwise 'exclusive or'
7  result = bar ^ foobar; // result = 1
8  // bitwise 'not'
9  result = ~foo; // result = -2
```

Bitwise Shift Operators

```
1  int a = 1, b = 2, c = -2;
2  // bit shift to the right (arithmetic)
3  int result = b >> 1; // result = 1
4  // bit shift to the right (arithmetic)
5  result = c >> 1; // result = -1
6  // bit shift to the left
7  result = a << 1; // result = 2
8
9  // Logical right shift (there is no Logical Left shift)
10 result = b >>> 1; // result = 1
11 result = c >>> 1; // result = 2147483647;
```

Conversions

- Memorize diagram on page **61** of **Core Java Vol 1**

```
1  12 + 14.1; // 12 is converted to double
2  10f - 17;  // 17 is converted to float
3  50L + 10;  // 10 is converted to long
4
5  short foo = (short) 10;
6  10 + foo;  // foo is converted to int
```

Casts

- May be necessary to explicitly cast to other types
 - Have already seen in needing to get to short type
- Careful as down-casting results in loss of precision

```
1 // downcast from double to short
2 double foo = 99999.99999d;
3 short bar = (short) foo;
4 // what is bar?
```


Order of Operations

- Memorize diagram on page 63 of Core Java Vol 1

```
1  int a = -1, b = -1, c = 3;  
2  c += b *= a;  
3  // what's c?
```

Strings!

- First, not primitives!
 - They're Objects (next lecture)
- Any Unicode character, surround in quotation marks
 - "This is a string in Java"
 - Escape Unicode - "This is the snowman character \u2603"
- Unlike other objects they're literals do not need 'new' and they have an overloaded '+' operator

```
1 String brianLangel = "Brian Langel";
2 String snowman = "\u2603";
3 // can concatenation with '+' however prefer `String.format`
4 String brianLangelAndSnowman = brianLangel + snowman;
5 // concatenation via String.format
6 String combined = String.format("%s %s", brianLangel, snowman);
```

String.format

- Similar to C style printf

```
1 String.format("%d int | %.f float | %.2f float", 10, 10.121212, 10.121212);  
2 String.format("0x%x - hexadecimal!", 10);  
3 String.format("%s %s %s concatenated", "this", "and", "that");
```

StringBuilder / StringBuffer

- Prefer StringBuilder as you should never be needing to build strings shared across threads.

```
1  int lineLimit = 10;
2  StringBuilder buffer = new StringBuilder();
3  for (int i = 0; i < characters.size(); i++) {
4      char character = characters;
5      buffer.append(character);
6      if ((i != 0) && ((i % lineLimit) == 0)) {
7          buffer.append('\n');
8      }
9  }
10 String text = buffer.toString();
```

Control Flow - Blocks

- Surrounded by braces {} they define a logical scope for variables.
- Methods, if statements, loops, try/catch are all examples of blocks.
- Can define arbitrary blocks yourself

```
1 private void blockOne() {  
2     int value = 1;  
3     if (true) { // block two  
4         int scoped = 0;  
5         value++;  
6         scoped++;  
7     }  
8     for (int i = 0; i < 1; i++) { // block three  
9         int scoped = 0;  
10        value++;  
11        scoped++;  
12    }  
13    try { // block four  
14        int scoped = 0;  
15        value++;  
16        scoped++;  
17        throw new RuntimeException();  
18    } catch (Exception e) { // block five  
19        int scoped = 0;  
20        value++;  
21        scoped++;  
22    }  
23    // custom block, block six  
24    {  
25        int scoped = 0;  
26        value++;  
27        scoped++;  
28    }  
29    System.out.printf("Scoped is not accessible, value = %d\n", value);  
30 }
```

Control Flow - If Statements

- The else is optional
- Although allowed for single lined statements, ALWAYS surround with braces

```
1  int foo = 1, bar = 1;
2  if ((foo == bar) && (bar != 2)) {
3      System.out.printf("Foo & Bar not equal to 2%n");
4  }
5  if ((foo != bar) && (foo == 2)) {
6      System.out.printf("Bar not equal to 2%n");
7  } else {
8      System.out.printf("Bar may equal 2%n");
9  }
10 if ((foo == bar) && (foo == 2)) {
11     System.out.printf("Foo & Bar equal to 2%n");
12 } else if (foo == 2) {
13     System.out.printf("Bar not equal to 2%n");
14 } else {
15     System.out.printf("Bar may equal 2%n");
16 }
```

Control Flow - Loops

```
1 String[] array = new String[] { "foo", "bar" };
2 // for Loop
3 for (int i = 0; i < array.length; i++) {
4     // do something
5 }
6 // while Loop
7 int i = 0;
8 while (i < array.length) {
9     // do something
10    i++;
11 }
12 // do-while Loop
13 i = 0;
14 do {
15     // do something
16 } while (i < array.length);
17 // for-each Loop
18 for (String entry : array) {
19     // do something
20 }
```

Control Flow - Switch Statement

- Available for primitive types and String (as of Java 7)
- Careful not to “fall through”

```
1  String value = "foo";
2  switch (value) {
3      case "foo":
4      case "bar":
5          // do something
6          break;
7      default:
8          // do something
9  }
```


Arrays

- Zero based
- Protected at runtime with bounds checking
 - Will throw an `ArrayIndexOutOfBoundsException` exception
- Cannot use pointer arithmetic (as you can in C++) to increment
- Always initialize with array brackets associated with type

```
1  int[] array = new int[100];
2  // can also initialize with known values
3  int[] values = new int[] { 1, 2, 3, 4, 5 };
4  // has length member
5  int size = values.length; // equals 5
6  // access and assign with common syntax
7  int firstValue = values[0];
8  values[0] = firstValue + 1;
```

Multidimensional Arrays

```
1  // initialize with first dimension
2  int[][] multi = new int[100][];
3  // initialize second in loop
4  for (int i = 0; i < multi.length; i++) {
5      multi[i] = new int[100];
6  }
7  // can also inline initialize
8  multi = new int[][] { { 1, 2, 3 },
9                        { 4, 5, 6 } };
```

Arrays Object

- Contains helper methods
 - Use these methods whenever necessary
 - Popular ones are `fill` and `sort`

```
1  int a[] = new int[10];  
2  Arrays.fill(a, 2);  
3  Random random = new Random();  
4  a[random.nextInt(10)] = random.nextInt();  
5  Arrays.sort(a);
```

Read Chapter 4

All sections will be covered in next lecture

Homework 2

<https://github.com/NYU-CS9053/Fall-2016/homework/week2>