# Object Oriented Programming with Java

Lecture 1: Basics

# Simple Java Program

```
public class HelloWorld {

   public static void main (String[] args) {

      //instructions go here

      System.out.println("Hello World!");

   }

}
```

*Each word of this should make sense by the semester's end!*

*For now it is boilerplate code—just the template we'll use to write code.*

# Whitespace

- Whitespace means the 'blank' characters:
  space, tab, newline
- Whitespace is (almost) irrelevant in Java
  - Whitespace is used to separate tokens
    **int x** vs **intx**
  - we can't type <enter> to span lines within a string
- Syntax is not based on indentations
  - but indentation is highly recommended (required for this class)

# Bad Whitespace Example #1

Valid, but horribly written, code.
(excessive, meaningless spacing)

```
public                          class
   Spacey { public
      static


void
      main(String[
               ] args
         ){ System      .
         out.println("Weirdly-spaced code still runs!"

                     );}}
```

# Bad Whitespace Example #2

Valid, but horribly written, code.
(one-liners aren't always best!)

```
public class Spacey2{public static void main(String[]args){System.out. println("space-devoid code also runs...");}}
```

*Code like this might not receive any credit! Seriously, don't do this in anything you ever turn in. Never make the grader unhappy.*

# Good Whitespace Example

```
public class GoodSpacing {
    public static void main (String[] args) {
        int x = 5;
        int y = 12;
        System.out.println("x+y = "+ (x+y));
    }
}
```

*indentation levels for each block: class, method definitions, control structures…*

# Identifiers

- Identifiers are the ***names*** we choose for variables, methods, classes, interfaces, etc.
- An identifier can be made up of letters, digits, the underscore character ( _ ), and the dollar sign ($)
- Identifiers cannot begin with a digit
- Java is *case sensitive* - `Total`, `total`, and `TOTAL` are different identifiers
- Identifiers cannot be reserved keywords

# Java Keywords

Keywords are identifiers that have meanings in the language definition. They cannot be used by programmers as new identifiers.

*Keyword: one of*

| | | | | |
|---|---|---|---|---|
| abstract | continue | for | new | switch |
| assert | default | if | package | synchronized |
| boolean | do | goto | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
| case | enum | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp | volatile |
| const | float | native | super | while |

# Identifiers

By convention, programmers use different case styles for different types of identifiers, such as:

- *lower case* or *camel case* for variables and methods
  `response, previousIndex`
- *title case* or *upper camel case* for classes
  `Person, MasonStudent`
- *upper case* for constants
  `MASON, MAX_INT`

# Identifier Examples

Legal Identifier Examples:

hello

camelCaseName

__$_09abizzare

user_input18

anyArbitrarilyLongName

Illegal Identifier Examples:

two words

Extra-Characters!

1st_char_a_number

transient *(it's a keyword)*

Dots.And.Hooks?

# Types

- Java is **strongly typed**: every expression has a specific type that never changes whether it's a variable, literal value, method call, or any other expression
- Java is **statically-typed**: the type of a variable is known at compile-time instead of at run-time
- Java has two kinds of types:
  - **primitive** types (containing literal values)
  - **reference** types (containing objects of some class).

# Primitive Types

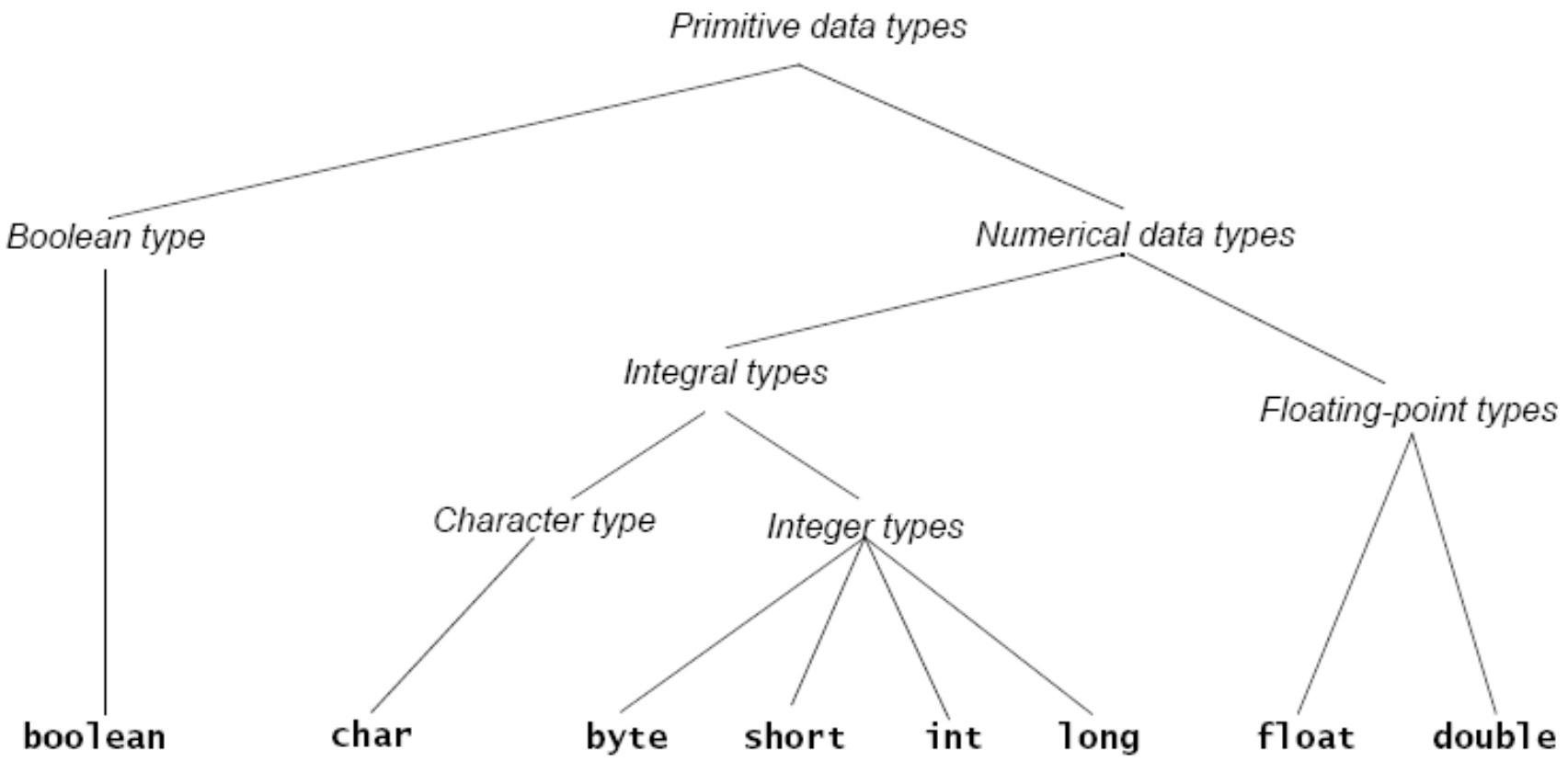
Primitive Types contain the basic values of the language, such as numbers, characters, and booleans.

**boolean**: truth values. Only possible values:
`true` and `false`

**char**: one character in single-quotes (or code number):
'a' 'H' '\n' '5'

**numbers:** there are many versions of integers (all with limited ranges), and two versions of real numbers (different precision/ranges).

# Primitive Types



# Integer Types

Java provides integral numbers of different sizes (number of bits), and different ranges of values:

| | | | | |
|---|---|---|---|---|
| `byte` | (8 bits) | -128 | *to* | 127 |
| `short` | (16 bits) | -32768 | *to* | 32767 |
| `int` | (32 bits) | -2147483648 | *to* | 2147483647 |
| `long` | (64 bits) | $-2^{63}$ | *to* | $2^{63} - 1$ |
| `char` | (16 bits) | 0 | *to* | 65535 |

* `char` *can use a Unicode # but it still is printed as a character, not a number*

* all integer **literals** are internally stored as **int** type. Therefore, larger values will be truncated unless you add an **L** to the end *(or **l** which looks like 1)*

```
9823412674152 34L
```

# Literal Integer Representation

Integers can be represented in decimal, hexadecimal, or octal.
*(different representations of the **same values**!)*

***decimal*** (base 10): 0 or start with 1-9, followed by one or more of 0-9

0 10 483 −9876501234 66045

***hexadecimal*** (base 16): prefix 0x, followed by one or more of 0123456789ABCDEF (case insensitive: a-f are equivalent to A-F)

0x0 0xfade 0x1B2C 0x9 −0x10

***octal*** (base 8): prefix 0, followed by one or more of 0-7

00 071 −045306 01777 010

# Note on Different Representations

All three inputs are alternatives you can use to describe the **same** values. Similar to what happens with **char**

You also know:

- Roman Numerals (e.g., XLVI)
- tally-marks (e.g. ||||) which is practically **base 1** system

All of these represent integers!

Don't confuse representation with meaning

# Floating Point Numbers

**Approximating** the real #s: called floating point numbers.

**We just write things in normal base 10 as always.**

internal binary representation: like scientific notation.


S: sign bit. E: bias-adjusted exponent M: adjusted fractional value

$$value = (-1)^S * 2^E * M$$

Also representable: infinity (+/−), NaN ("not a number")

**float:** 32-bit representation. (1 sign, 8 exp, 23 frac)
**double:** 64-bit representation. (1 sign, 11 exp, 52 frac)

# Representing Floating Point Numbers

Floating Point numbers:

- may have a decimal point followed by digits
  2.32  1.21  450000
- may be written in scientific notation:
  2.32e5 = $2.32 \times 10^5$ = 232000
- may be very large:
  2E35F  2e250 -2e250

  Float.POSITIVE_INFINITY

* all floating point **literals** are internally stored as **double** type.
You must add an **F** or **f** to the end if you want to make it a float
  0**f**  3.14159**f**  -2E3**F**  59023**f**

# Wrapper classes for primitive types

| primitive data type | wrapper class |
|---|---|
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| boolean | Boolean |
| char | Character |

# Practice Problem

To what types can these numbers be stored?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | char | byte | short | int | long | float | double |
| 150 | char | | short | int | long | float | double |
| 3.14 | | | | | | | double |
| 7e200 | | | | | | | double |
| −0x3F | | byte | short | int | long | float | double |
| 371F | | | | | | float | double |

# Creating Variables

- Variables must be **declared** and **initialized** before use.
- **Declaration:** creates the variable. It includes a type and a name. The variable can only hold values of that type.

```
int x;     char c;     boolean ok;     Person p;
```

- **Initialization:** assign an expr. of the variable's type to it.

```
x=7+8;    c='M';     ok = true;  p = new Person();
```

- **Both:** we can declare and instantiate all at once:

```
int x = 5;  char c = 'S';  Person p = new Person();
```

# Type Casting

- With all these numerical types, changing between them is possible, and has implications.
- a **cast** is a conversion from one type to another. We cast things by placing the type in parentheses in front of it:

```
(int) 3.14f
```

- One use: forcing floating-point division.

```
int x=3, y=4;
double z = ((double)x)/y;
System.out.println(z);  //prints 0.75
```

# Constants

- Constant is a variable that has a value that never changes
- The compiler will issue an error if you try to change the value of a constant
- We use the **`final`** modifier to declare a constant

```
final int MIN_HEIGHT = 60;
```

* keyword **`final`** has different semantics when applied to methods and classes

# Why use Constants?

Constants are useful for three important reasons:

1. they give meaning to otherwise unclear literal values
   → `MAX_LOAD` versus the literal `250`

2. they facilitate program maintenance
   → use a constant in multiple places, only update in one place

3. formally establish that a value cannot change, avoiding inadvertent errors by other programmers
   → This is why we use the `final` keyword

# Java Comments

There are two main styles of comments in Java:

- **Single-Line**: from **//** to the end of the line.

- **Multi-Line**: all content between **/*** and ***/**, whether it spans multiple lines or part of one line.

*JavaDoc: convention of commenting style that helps generate code documentation/API. More on this later.*

# Expressions, Statements

**Expression:** a representation of a calculation that can be evaluated to result in a single value. There is no indication what to do with the value.

**Statement:** a command, or instruction, for the computer to perform some action. Statements often contain expressions.

# Basic Expressions

- literals (all our numbers, booleans, characters)
- operations

| | |
|---|---|
| relational ops: | `<  <=  >  >=  ==  !=` |
| math ops: | `+  -  *  /  %` |
| boolean ops: | `&&  ||  !` |
| ternary op: | `exp ? exp : exp` |

- variables
- parenthesized expressions: `(exp)`

# Conditional Expression

The ternary operator is a conditional *expression*: it evaluates the boolean expression, and then results in the middle expression when true or the last expression when false.

```
boolexpr ? expr_when_true : expr_when_false
```

We **must** have all three operands, and the type of 2nd and 3rd expressions must agree with the resulting type of the entire expression. Example:

```
int c = a>b ? 6 : 7;
```

# Increment/Decrement operator

Shorthand allows us to increment or decrement a number:

```
x++   ++x   //increment    (after/before stmt)
x--   --x   //decrement   (after/before stmt)
```

Often these are one-liners or isolated:

```
x++;
for(int i=0; i<10; i++)
```

# Increment/Decrement

**Suffix form:**

- **x++**, **x--** : **first** use the current value of the variable and **after** that increment/decrement the variable

```
int x=1;
int y = (x++) * 5;        →        int y = x * 5;        x = x + 1;
```

**Prefix form:**

- **++x**, **--x** : **first** increment/decrement the variable and **after** that use the new value of the variable

```
int x=1;
int y = (++x) * 5;        →        x = x + 1;        int y = x * 5;
```

# Expression Examples

Legal:

```
4+5
x%2==1
(3>x++) && (! true)
(2+3)*4
(x<y)&&(y<--z)
```

Illegal:

```
x>y>z
4 && false x=3
7(x+y)
```

# Basic Statements

- Declaration: announce that a variable exists.
- Assignment: store an expression's result into a variable.
- method invocations (may be stand-alone)
- blocks: multiple statements in **{ }**'s
- control-flow: if-else, for, while, …

# Statement Examples

```
int x;  // declare x
x = 15; // assignment to x
int y = 7;      // decl./assign. of y
x = y+((3*x)-5);  // assign. with operators
x++; // increment stmt (x = x+1)
System.out.println(x); // method invocation


if (x>50) {     //if-else statement
   x = x - 50;
}
else {
   y = y+1;
}
```