IntelliJ and Java Primitive Types

Objectives

- After this lecture, you will be able to:
  - describe the difference between a .java and .class file
  - define several Java keywords
  - declare and use primitive types
  - create arithmetic expressions

Java JDK

- Your laptop likely already has some Java stuff installed
- You need the Java Development Kit version 8 or higher
  - Prefer 17
  - Use:

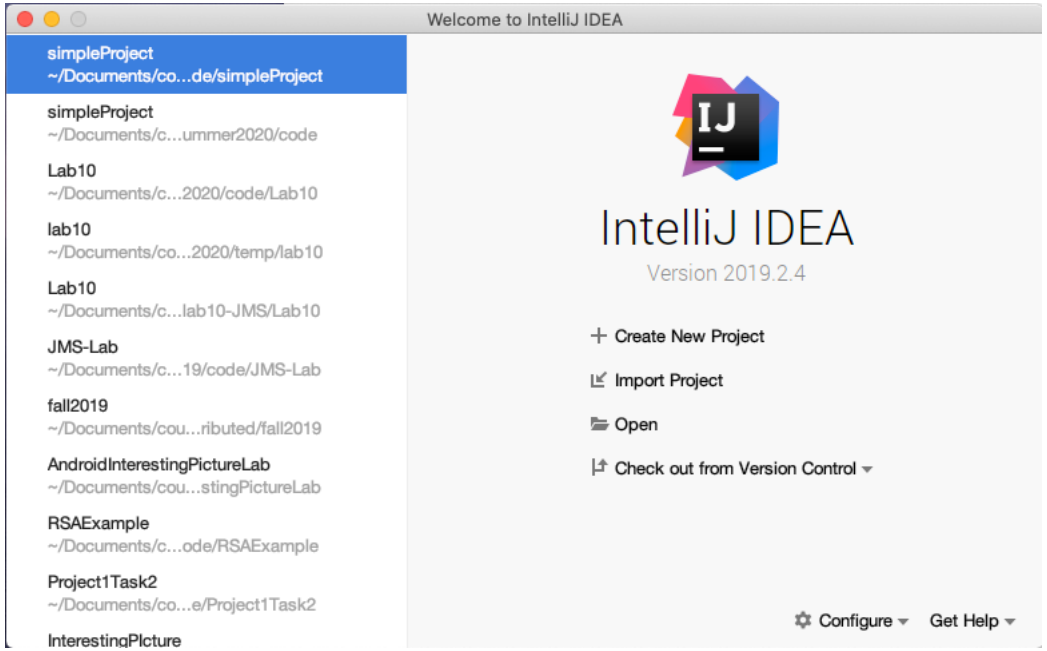
  Oracle: https://www.oracle.com/java/technologies/javase-downloads.html#javasejdk
  - Windows environment variables
- JDK versus JVM
- Command line check: javac - version and java -version

IntelliJ

- Establish your student credentials with JetBrains for the free version of IntelliJ for educational use: https://www.jetbrains.com/student/
- Download and install IntelliJ IDEA Ultimate.

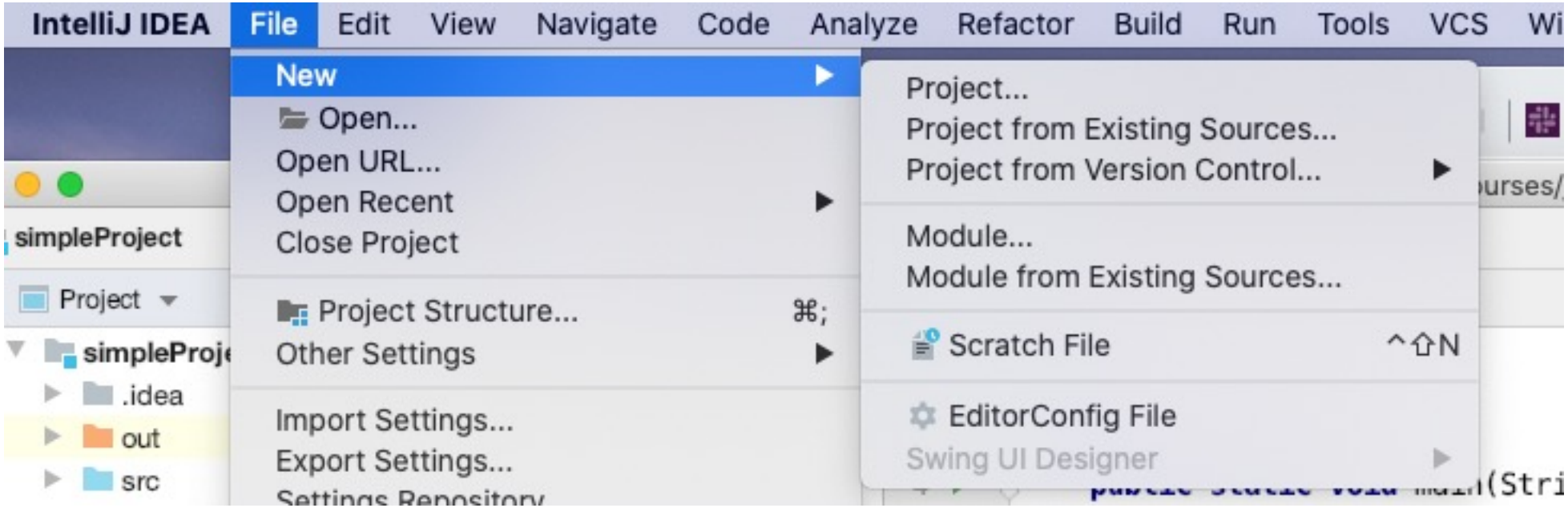
Projects in IntelliJ

- Before this, create a Folder to hold your projects
- Either this: Create New Project



Projects in IntelliJ, cont.

- or this: File -> New -> Project

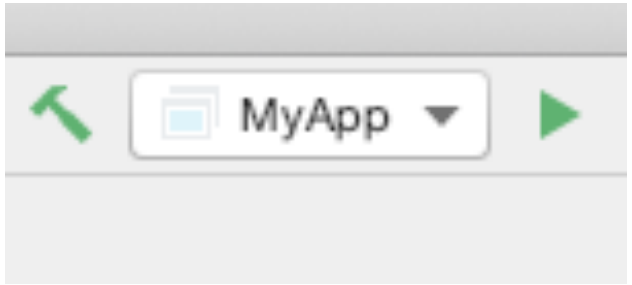


Projects in IntelliJ, cont.

- Can also use File->Open or File->Open Recent
  - And File->Close Project
- Project Name: up to you
- Project location – use that pre-defined location

Sample Project

- Everyone's favorite, Hello World
- Create a project named simpleProject
- Use File -> New -> Java Class to create class MyApp
  - Classes usually start with capital letter, use camelCase
  - Picked a bad name, or misspelled it? Highlight it, then Code -> Refactor->Rename. Either choose a suggestion, or Shift-Option to get a dialog box.
- Run it with Green Arrow, Run, or Run …
- Running the .class file (not recommended)



Projects Folders

- Project structure
  - src directory: one file per class
    - With a few exceptions, one class per file
  - Alternative when using packages: right-click on src, choose New -> Package
    - Industry practice: reverse-url for uniqueness, as in: edu.cmu.ds
  - out->production->myproject: contains .class files (compiled Java code)

.java versus .class versus .jar

- Three kinds of files:
  - ***.java***: code files containing one class (sometimes more, but be careful)
  - ***.class***: ***byte code*** files created by the Java compiler
    - executable by the JVM
    - usually one main( ) (sometimes more, but be careful)
  - ***.jar***: Java executables for distribution or execution
    - kind of equivalent to .exe files in Windows
    - ***shaded jar***: jar file that contains all dependent jars – useful if your code depends on some non-system jar files

Data Primitive Types

Basic Data Typing in Java

- Java has two basic kinds of data types
  - ***primitive*** or "built-in" types
  - ***class*** types
- You can't add to the built-ins, but you can create new classes
- Java is ***strongly typed*** and requires ***declare before use*** and no name reuse (within a scope)
  - unlike Python, for example

Strong Typing

- Why is strong typing a good thing?
  - Variables don't change type from one place to another
  - Method calls check parameter types and don't fail inside the method because of wrong types
    - The compiler checks this, so it's prevention
- Is weak typing ever a good thing?
  - Scripting languages like Python don't do compile-time type checking

Primitive Types

These are the ones we'll use (mostly)

Primitive	Size	Min value	Max value
`boolean`	(word)	-	-
`char`	16 bits	0 (Unicode)	2^16 – 1 (Unicode)
`byte`	8 bits	-128	+127
`short`	16 bits	-2^15	+2^15 – 1
`int`	32 bits	-2^31	+2^31 – 1
`long`	64 bits	-2^63	+2^63 – 1
`float`	32 bits	+/-1.4E-45	+/-3.4028235E+38
`double`	64 bits	+/-4.9E-324	+/-1.79769E+308
`void`	(word)	-	-

"word" means the standard addressable memory unit in your computer – either 32 bits or 64 bits

Primitive Types, cont.

- Most used: `boolean, int, double, void`
- Needed in some applications: `char, long, float`
- Rarely used: `byte, short, unsigned char`
- Math library typically uses `double`

Primitive Types, cont.

- Why so many types?
  - Borrowed from C/C++
  - C dates back to when computer memory was small and expensive
  - So you'd choose the low-precision types if possible
- Memory is now big enough and cheap enough that you should never care about a primitive's size
  - Unless you're using a library that requires them

Primitive Types, cont.

- Type ***casting***: changing from one type to another
  - Not the same as converting: getting a String and changing it to an int, for example
- ***Widening***: from smaller precision to larger precision – this is automatic

```
int i = 7;
double x = i;              // No problem
```

- This is sometimes called ***upcasting***
- Upcasting is used for mixed expressions, as in `x + i` which will be `double`

Primitive Types, cont.

- ***Narrowing***: use the explicit typecast in parentheses

```
double x = 7.25;
int i = (int)x;   // Possible problem
```

- This is sometimes called ***downcasting***. The loss of precision can be a problem
- Constants are `int` and `double`, unless you add `L` or `F`
- While a String can be changed to an int or double, this isn't casting, it's conversion

Primitive Types, cont.

```
public class Example {
    public static void main(String[] args) {
        double x = 2.75;
        int i = 7;
        int ix = (int) x;     // Truncates x to 2
        double di = i;        // Upcasts i to 7.0

        float y = 1.25F;      // Cast to float - decimal constants are double
        long j = 15L;         // Cast to long - integer constants are int
        char letter = 'a';    // Note the single quotes, not double
        boolean b = true;     // or false
    }
}
```

cast 1.25 to float

cast 15 to long

Arithmetic Operators

- The usual set: `+, -, *, /`
  - `/` for two int's is integer division; `%` is modulus
  - No exponentiation operator; use `Math.pow(base, power)`
  - Unary + and -
- And miscellaneous bitwise and shift operators – ignore these for now
- There's a set of ***precedence*** rules for operators. Instead of memorizing it, use parentheses to force the order you want: stuff inside parentheses goes first

Arithmetic Operations, cont.

```
int a = 7, b = 3, c = 5;

// Integer division: 8*2/5 = 3
System.out.println( ((a+1)*(b-1))/c );

// Cast c to double, forces floating point division: 3.2
System.out.println( ((a+1)*(b-1))/(double)c );

// Integer mod, take the remainder: also 3
System.out.println(((a+1)*(b-1))%c);

// 5 to the third power: 125
System.out.println( Math.pow(c, b) );
```

Increment and Assignment Operators

- `++` and `--` can be either ***prefix*** or ***postfix***
  - If used in an expression, pay attention!
  - `i++` means, use i first, then increment; `++i` means, increment i first, then use it. By themselves, they are equivalent
  - Inside expressions, they are *not* equivalent. This is a style issue – I try to avoid using these inside expressions, but you'll see it a lot
- Besides =, Java has the increment-and-assign operators:
  - `+=, -=, *=, /=`

Increment and Assignment Operators, cont.

```
int a = 7, b = 3, c = 5;
System.out.println( ((a++)*(b--))/++c );  // Bad style #1

  ------------------------------------------------------------

a = 7; b = 3; c = 5;                      // Start over
c++;                                      // Increment before use
System.out.println( ((a)*(b))/c );        // Better style #1
a++; b--;                                 // Increment/decrement
                                          //   after use
```

Increment and Assignment Operators, cont.

Can you tell the difference in this example from the previous one?

```
a = 7; b = 3; c = 5;                          // Start over
System.out.println( ((++a)*(--b))/c++ );      // Bad style #2
```

```
a = 7; b = 3; c = 5;                          // Start over
a++; b--;                                     // Increment/decrement
                                              //    before use
System.out.println( ((a)*(b))/c );            // Better style #2
c++;                                          // Increment after use
```