Java Fundamentals

CS 240: Advanced Programming Concepts

Topics and Topic Order

- We won’t cover everything you will need to know in class
  - Read the assigned chapters
- We will focus on things that are significantly different from C++
- Topic order will be driven by the programming projects

Where Java Came From

- Early 1991 - Green project started at Sun Microsystems
- Tried to write a better C++ compiler
- Late 1992 - Completed Oak
- 1993 - Mosaic introduced
- Early 1994 - Green team (FirstPerson) disbanded
- Oak renamed Java and HotJava Browser Created
- May 23, 1995 - Netscape announcement
- 2010 – Oracle Acquired Sun Microsystems, and Java

What is Java?

> ***“A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language”.***

- The Java Language: An Overview (Sun Whitepaper)

Java Overview

- Similar syntax but in many cases different semantics from C++
- Differences between Java and C++
  - Built-in garbage collection
  - References instead of pointers
  - Data types are always the same size in Java
  - Specific boolean datatype and language constructs made to use it
    - if(x = 1) is a compile error in Java
  - Classes dynamically linked at runtime (no separate link step)
  - Java is a hybrid, compiled / interpreted language
  - Several other differences

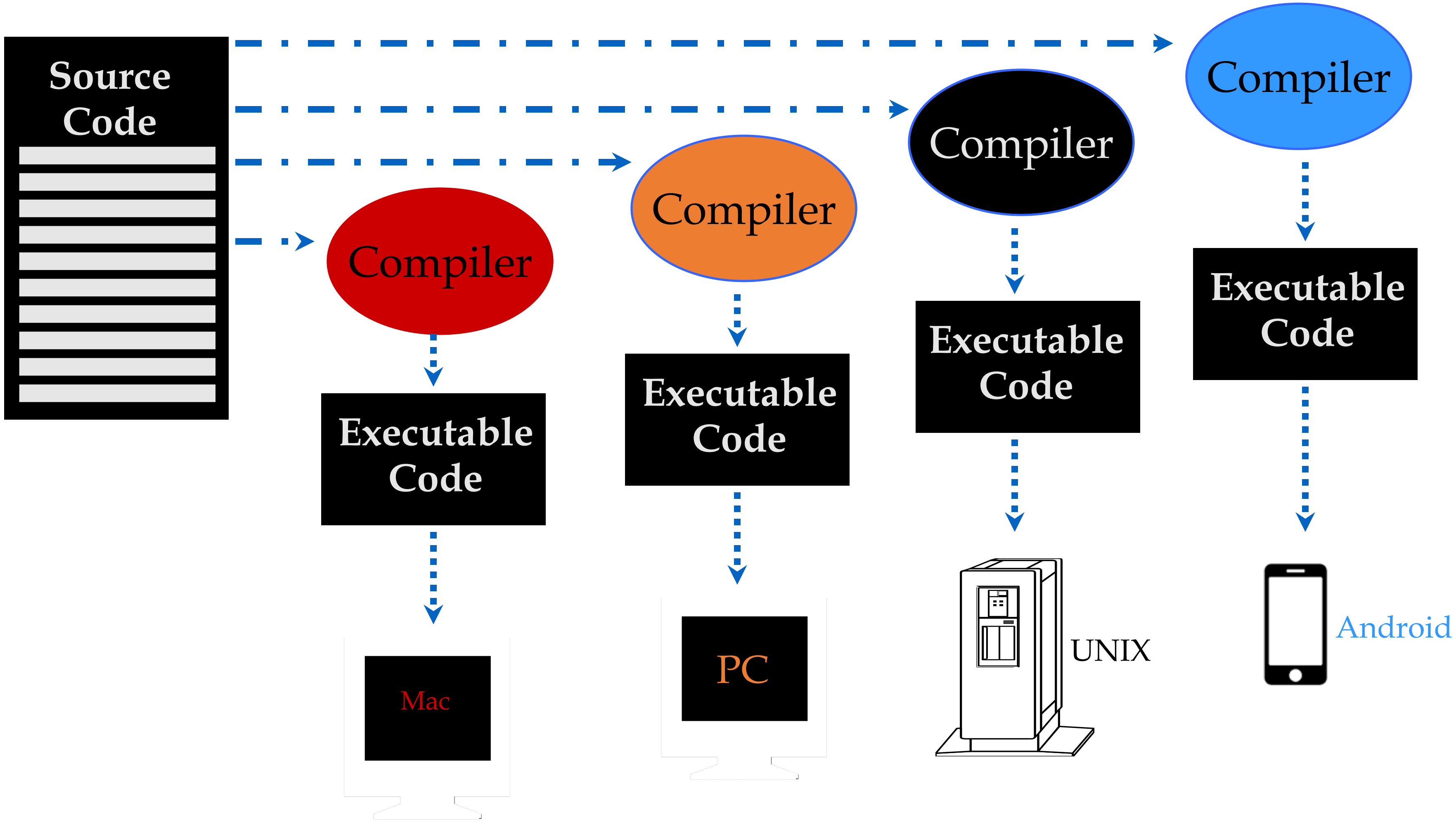
Getting and Installing Java

- Download the latest version of the JDK from Oracle’s website
  - https://www.oracle.com/technetwork/java/javase/downloads/index.html
- Find platform specific installation instructions on the download page

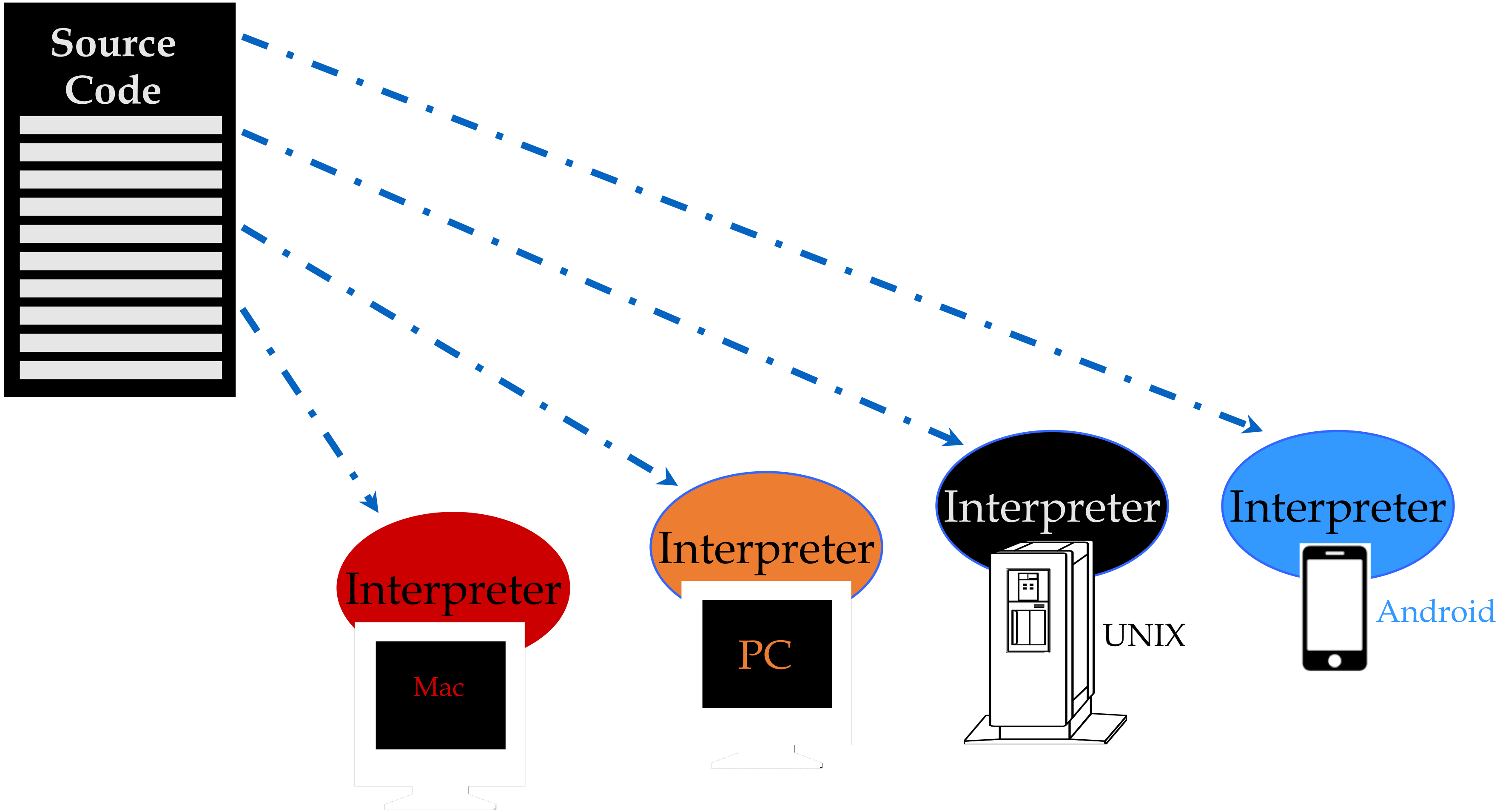
Java IDEs

- Intellij Idea (community edition is free)
- Android Studio (free)
- Eclipse (free)
- Others

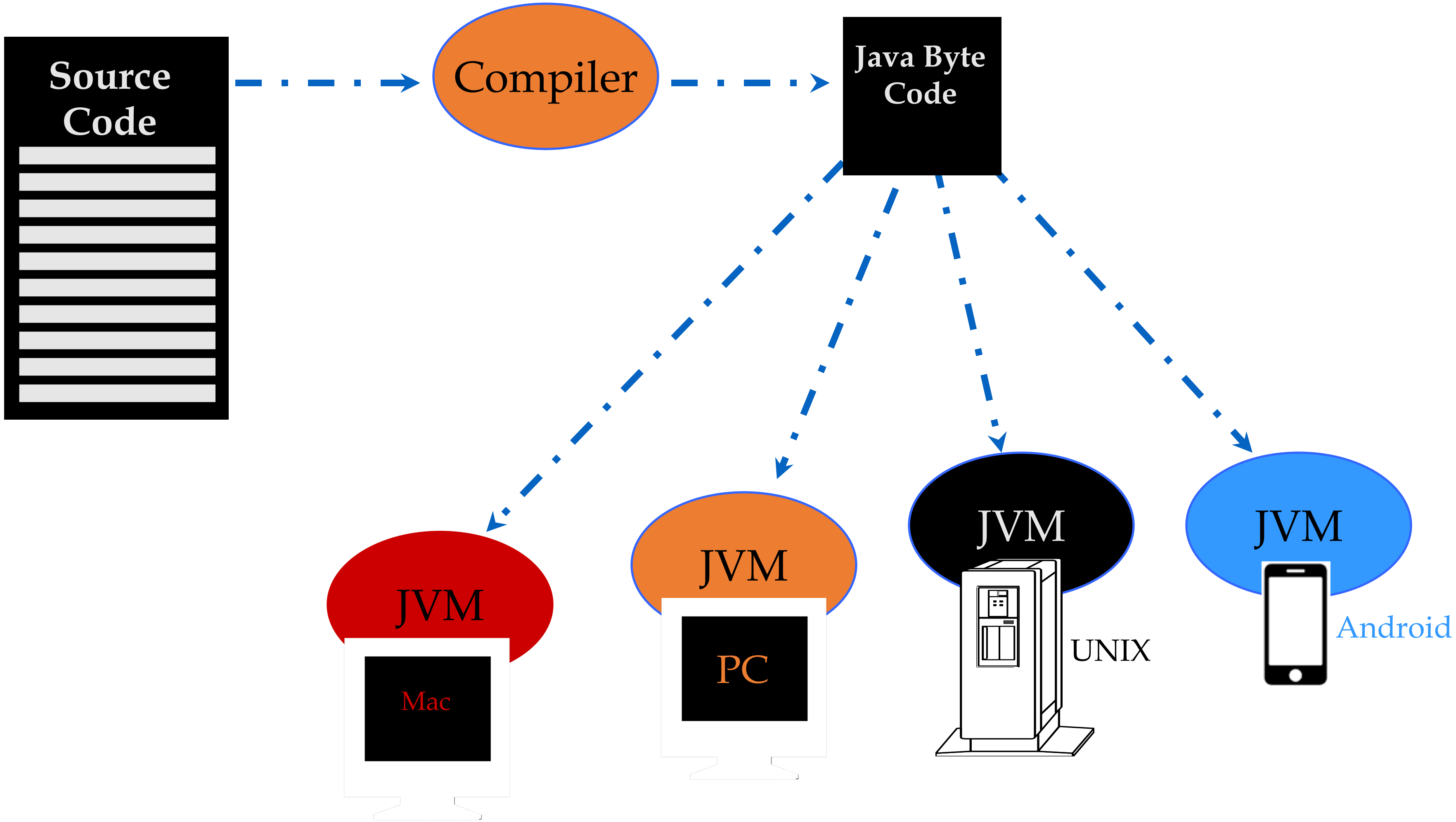
Compiled Code



Interpreted Code



Java Code


Source Code
Compiler
Java Byte Code
JVM
Mac
JVM
PC
JVM
UNIX
JVM
Android

Compiled vs. Interpreted Code

- Compiled = fast but not portable
  - Runs on bare hardware-–instructions are not interpreted at runtime
  - Recompile (and often re-code and then recompile) to run on different hardware
- Interpreted = slow but portable
  - Runs on a VM or interpreted that interprets and translates instructions at runtime
  - Runs on any platform with an interpreter without recompiling
- Java: seeks to have best of both (fast and portable)
  - Compiled to bytecode which runs on a virtual machine spec
  - Translation to actual machine language is minimal and fast
  - Runs on any platform with a JVM (which is most platforms)

JIT Compilation and The Hotspot Virtual Machine

- JIT = Just in Time Compilation
- Hotspot VM = Dynamically recompilation at runtime
- Provides new opportunities for performance improvement
- Causes programs to start and run faster than JIT compiled code
- Can optimize to the specific hardware architecture
- Uses a generational garbage collector

Java Files

- MyClass.java = source file
  - With a few exceptions, there is one Java class per .java file
  - The file name must match the class name
- MyClass.class = executable file (executable by the JVM)

- The main method
  - public static void main(String [] args)
  - public static void main(String…args)

Creating Java Classes

```
public class SimpleJavaClass {
  public static void main(String [] args) {
    System.out.println(“Hello BYU!”);
  }
}
```

Code Examples:

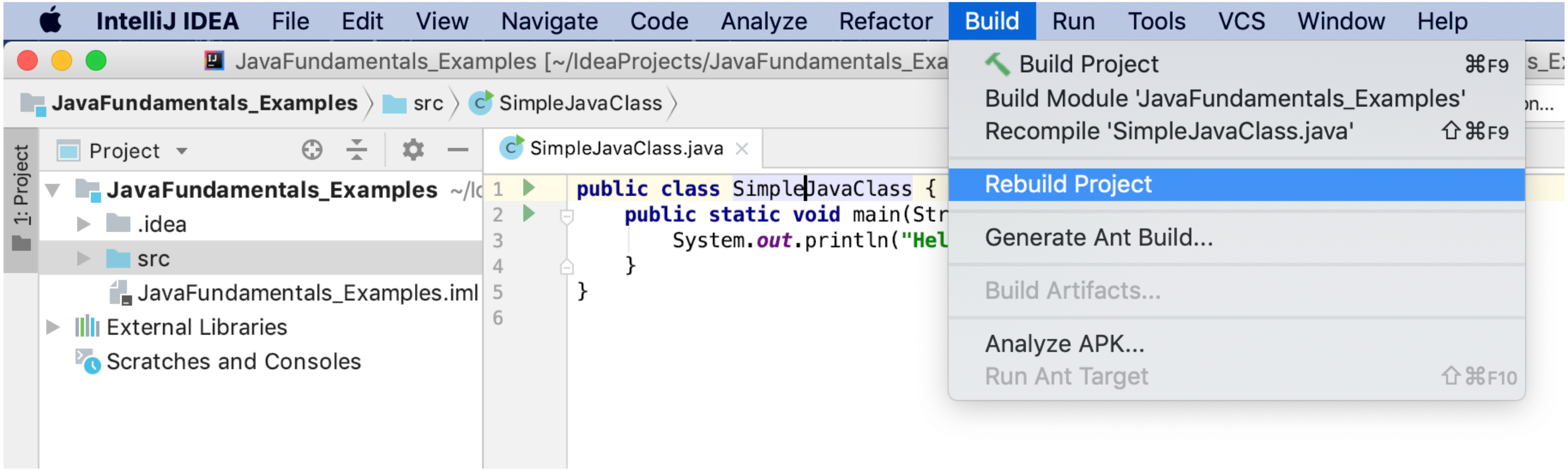
- SimpleJavaClass.java
- Point.java
- Rectangle.java
- PointAndRectangleUser.java

Compiling and Running Java Programs

- Compile
  - `javac SimpleJavaClass.java`
  - Produces SimpleJavaClass.class
  - For now, you must be in the directory that contains the .java file

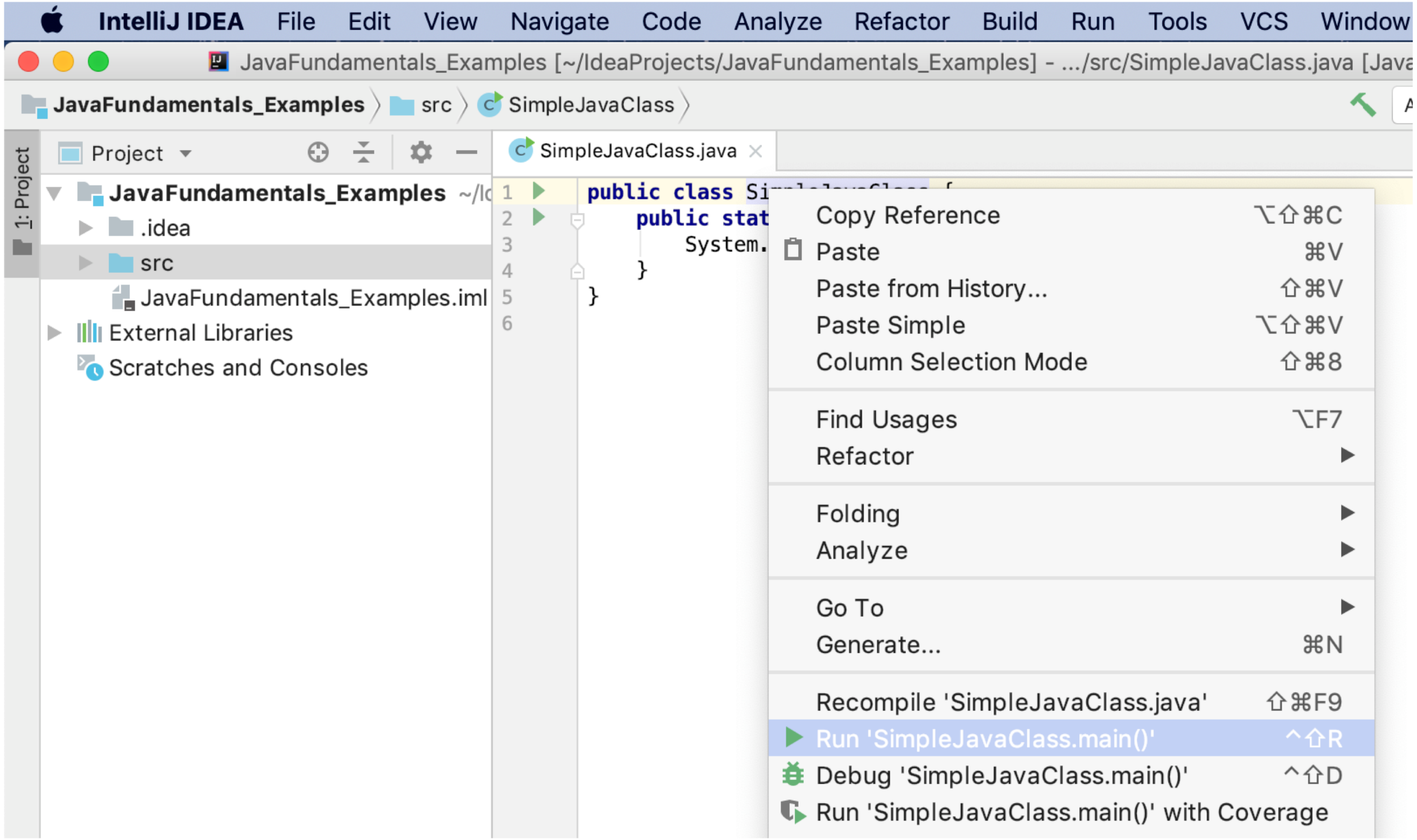
- Run
  - `java SimpleJavaClass`
  - No .class at the end
  - For now, you must be in the directory that contains the .class file

Compiling and Running in Intellij



```
public class SimpleJavaClass {
    public static void main(Str
        System.out.println("Hel
    }
}
```

Compiling and Running in Intellij



```
public class Si
    public stat
        System.
    }
}
```

IntelliJ IDEA
File
Edit
View
Navigate
Code
Analyze
Refactor
Build
Run
Tools
VCS
Window
JavaFundamentals_Examples [~/IdeaProjects/JavaFundamentals_Examples] - .../src/SimpleJavaClass.java [Java
JavaFundamentals_Examples
src
SimpleJavaClass
1: Project
Project
JavaFundamentals_Examples ~/Id
.idea
src
JavaFundamentals_Examples.iml
External Libraries
Scratches and Consoles
SimpleJavaClass.java
Copy Reference ⌥⇧⌘C
Paste ⌘V
Paste from History... ⇧⌘V
Paste Simple ⌥⇧⌘V
Column Selection Mode ⇧⌘8
Find Usages ⌥F7
Refactor
Folding
Analyze
Go To
Generate... ⌘N
Recompile 'SimpleJavaClass.java' ⇧⌘F9
Run 'SimpleJavaClass.main()' ^⇧R
Debug 'SimpleJavaClass.main()' ^⇧D
Run 'SimpleJavaClass.main()' with Coverage

Javadoc

- Documentation for the Java class library
- Generated from code and Javadoc comments in the code
- Download and install or access from Sun’s website with a Google search
    - Google search: Java 12 api
- Can generate for your own classes using the Javadoc tool that comes with the JDK

Primitive Datatypes

- byte
- short
- **int**
- long
- float
- **double**
- char
- **boolean**

- Code Example
  - PrimitiveDataTypes.java

Converting a String to an int

- The Integer Wrapper Class
  - **int Integer.parseInt(String value)**
  - Several other methods for parsing between Strings, ints and Integers
- Similar Methods in:
  - Byte
  - Short
  - Double
  - Long
  - Float
  - Double
  - Boolean

Strings

- String Declaration and Assignment
  - `String s = “Hello”;`
  - `String s = new String(“Hello”);`
- String concatenation
  - `String s1 = “Hello”;`
  - `String s2 = “BYU”;`
  - `String s3 = s1 + ” “ + s2;`
    - Strings are immutable (concatenation always creates a new String)
- String formatting
  - `String s1 = “Hello”;`
  - `String s2 = “BYU”;`
  - `String s3 = String.format(“%s %s”, s1, s2);`

- Code Example:
  - StringExamples1.java

Important String Methods

- `int length()`
- `char charAt()`
- `String trim()`
- `boolean startsWith(String)`
- `int indexOf(int)`
- `int indexOf(String)`
- `String substring(int)`
- `String substring(int, int)`

- Many others. See Javadoc.
- Remember: Strings are immutable, none of these methods change the String

- Code Example
  - StringExamples2.java

Special Characters

- \n (newline)
- \t (tab)
- \” (double quote)
- \’ (single quote)
- \\ (backslash)
- \b (backspace)
- \uXXXX (insert the Unicode character represented by XXXX)
- \r (carriage return—return to the beginning of the current line—obsolete)
- \f (form feed—advance to the next line—obsolete)

- Code Example
  - SpecialCharacterExamples.java

Arrays

- See ArrayExample.java

Command-Line Arguments

```
public class CommandLineArgsExample {

    public static void main(String [] args) {

        for(int i = 0; i < args.length; i++) {
            String message = String.format(”Argument %d is %s", i, args[i]);
            System.out.println(message);
        }
    }
}
```

Specifying Command Line Arguments

- From the command line
  - `java CommandLineArgsExample abc 123 “Hello BYU”`

- From Intellij
  - Create a run configuration and specify arguments in the ”Program Arguments” field

Packages

- Packages provide a way to organize classes into logical groups
- Packages can have sub-packages (separated by . (dots))
- Specify the package for a class with a ’package’ statement at the top of the .java file
- Files (.java and .class) must be in a directory structure that matches the path structure
- The package name becomes part of the class name. Example: Java has two date classes:
  - java.util.Date
  - java.sql.Date
- You must refer to classes by their fully-qualified package name unless you use imports
- Code Examples:
  - Student.java
  - Student2.java

Import

- Import statements provide a shorthand for the fully-qualified package name (they allow you to just enter the class part of the name)
- They do not increase the size of your compiled .class files (unlike C/C++ includes)
- If used, they appear at the top of the file—before class declarations but after the package declaration (if a package declaration exists)
- The wildcard * imports all classes in the package, but not subpackages
    - Example: import java.util.*;
- You do not need an import in the following cases:
    - You choose to use fully-qualified package names (not normally recommended)
    - The class you are using is in the java.lang package (Object, String, and several others)
    - The class you would import is in the same package as the class that needs to use it

CLASSPATH

- An environment variable that contains a list of directories that contain .class files, package base directories, or other resources your application needs to access
    - Colon separated on Mac OS and Linux
    - Semicolon separated on Windows
- . (current directory) is implicitly on the CLASSPATH if you don't set a CLASSPATH
- Can use -classpath command line param
- IDEs like Intellij and Eclipse and Android Studio manage this for you

Input / Output (IO)

- Use a File object to represent a file in your program
- Use Readers and Writers to read and write text files
- Use InputStreams and OutputStreams to read and write binary files
- Readers and Writers, InputStreams and OutputStreams can be chained together to add functionality to your reads and writes
- Most file IO operations can result in IOExceptions being thrown
  - For now, just handle them by declaring that your method throws them:
    ```
    public void myMethod() throws IOException {
    ```
  - Will require you to import java.io.IOException (or use the fully-qualified name)
- Close your readers and writers when you are through (try-with-resources statements will do that for you)
  ```
  try(…) {
  ```
- Code Example
  - CopyFileExample.java

Another Way to Read a File: java.util.Scanner

```
public void processFile(File file) throws IOException {
    Scanner scanner = new Scanner(file);
    scanner.useDelimiter("((#[^\\n]*\\n)|(\\s+))+");

    while(scanner.hasNext()) {
        String str = scanner.next();
        // Do something with the String
    }
}
```

Another Way to Read A File: Files.readAllLines(Path)

```
public List<String> readFile(File file) throws IOException {
    Path path = Paths.get(file.getPath());
    List<String> fileContents = Files.readAllLines(path);
    return fileContents;
}
```