Object Oriented Programming in Java

1: Introduction
How to organize, compile and run simple examples

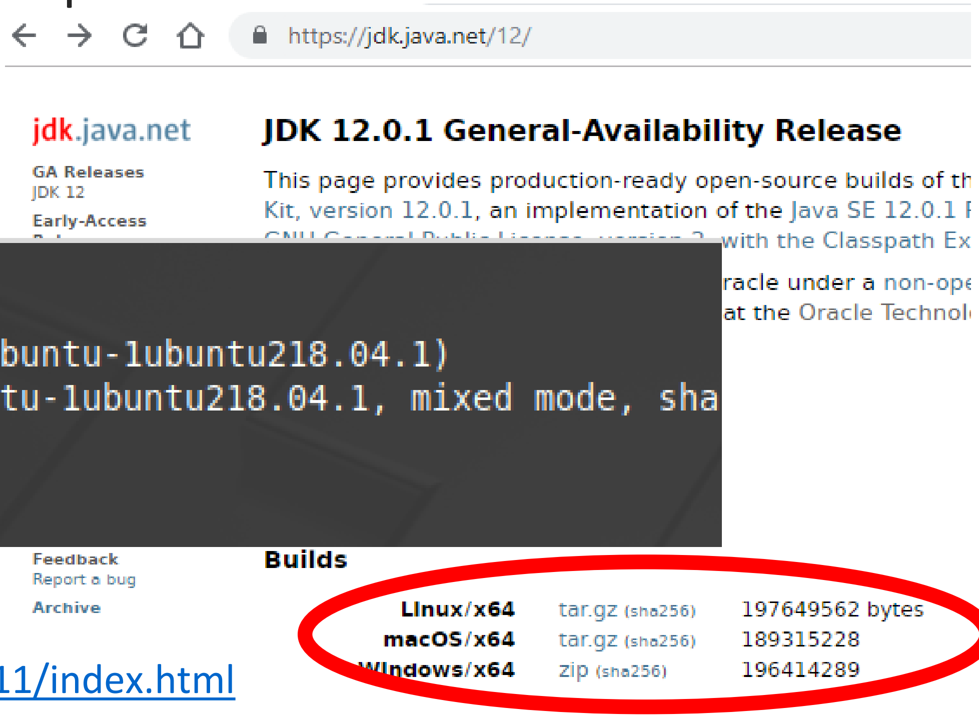
Licence

Java Development Kit

- Open JDK (Java Development Kit) ≥ 11
- Linux distributions based on Ubuntu 18.04

```
sudo apt install default-jdk
```

- Or download from http://jdk.java.net, unpack and set up path
  - Before moving on, check the output of these 2 commands
    - `java -version`
    - `javac -version`



https://jdk.java.net/12/

jdk.java.net

GA Releases
JDK 12
Early-Access

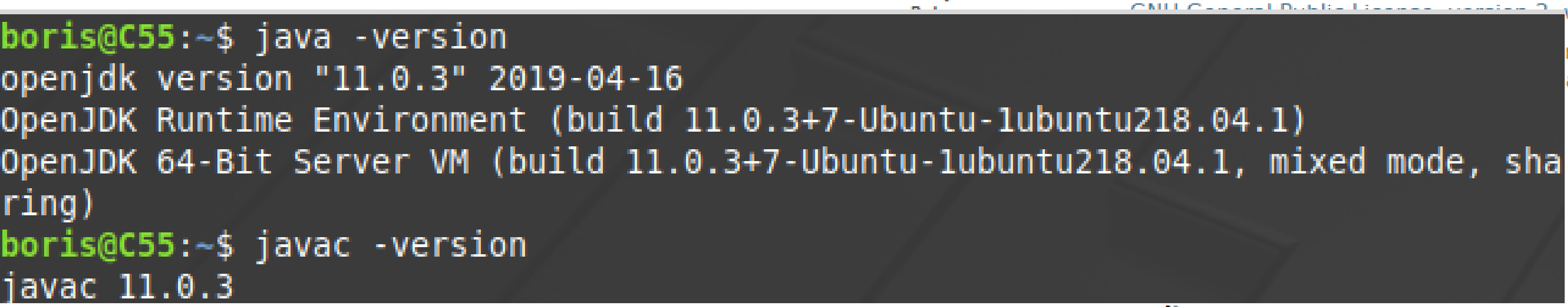
Feedback
Report a bug
Archive

JDK 12.0.1 General-Availability Release

This page provides production-ready open-source builds of th
Kit, version 12.0.1, an implementation of the Java SE 12.0.1 F
with the Classpath Ex
racle under a non-ope
at the Oracle Technol

Builds

```
Linux/x64     tar.gz (sha256)   197649562 bytes
macOS/x64     tar.gz (sha256)   189315228
Windows/x64   zip (sha256)      196414289
```

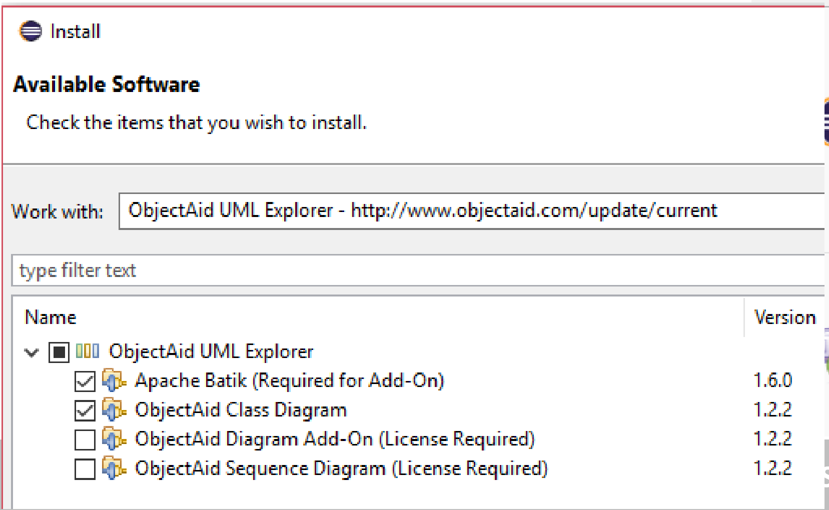


```
boris@C55:~$ java -version
openjdk version "11.0.3" 2019-04-16
OpenJDK Runtime Environment (build 11.0.3+7-Ubuntu-1ubuntu218.04.1)
OpenJDK 64-Bit Server VM (build 11.0.3+7-Ubuntu-1ubuntu218.04.1, mixed mode, sha
ring)
boris@C55:~$ javac -version
javac 11.0.3
```

- Documentation:
  - https://docs.oracle.com/en/java/javase/11/index.html

Integrated Development Environment (IDE)

- Many choices (Eclipse, Netbeans, IntelliJ, Visual Studio Code, …)
  - Recommended to use IDE after few introductory examples
- Eclipse: https://www.eclipse.org/downloads/packages/
  - Choose **Eclipse IDE for Java Developers,** download and unpack the archive
- Optionaly: ObjectAid UML Explorer (free for class diagrams)
  - https://www.objectaid.com/install-objectaid



Install

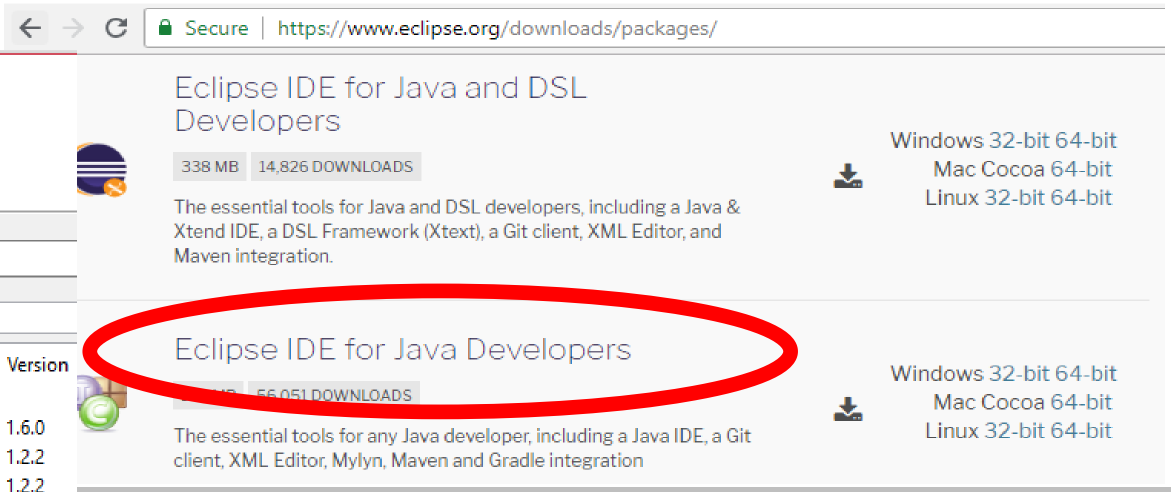
Available Software

Check the items that you wish to install.

Work with: ObjectAid UML Explorer - http://www.objectaid.com/update/current

type filter text

Name	Version
☐ ObjectAid UML Explorer	
☑ Apache Batik (Required for Add-On)	1.6.0
☑ ObjectAid Class Diagram	1.2.2
☐ ObjectAid Diagram Add-On (License Required)	1.2.2
☐ ObjectAid Sequence Diagram (License Required)	1.2.2



Secure | https://www.eclipse.org/downloads/packages/

Eclipse IDE for Java and DSL Developers

338 MB 14,826 DOWNLOADS

The essential tools for Java and DSL developers, including a Java & Xtend IDE, a DSL Framework (Xtext), a Git client, XML Editor, and Maven integration.

Windows 32-bit 64-bit
Mac Cocoa 64-bit
Linux 32-bit 64-bit

Eclipse IDE for Java Developers

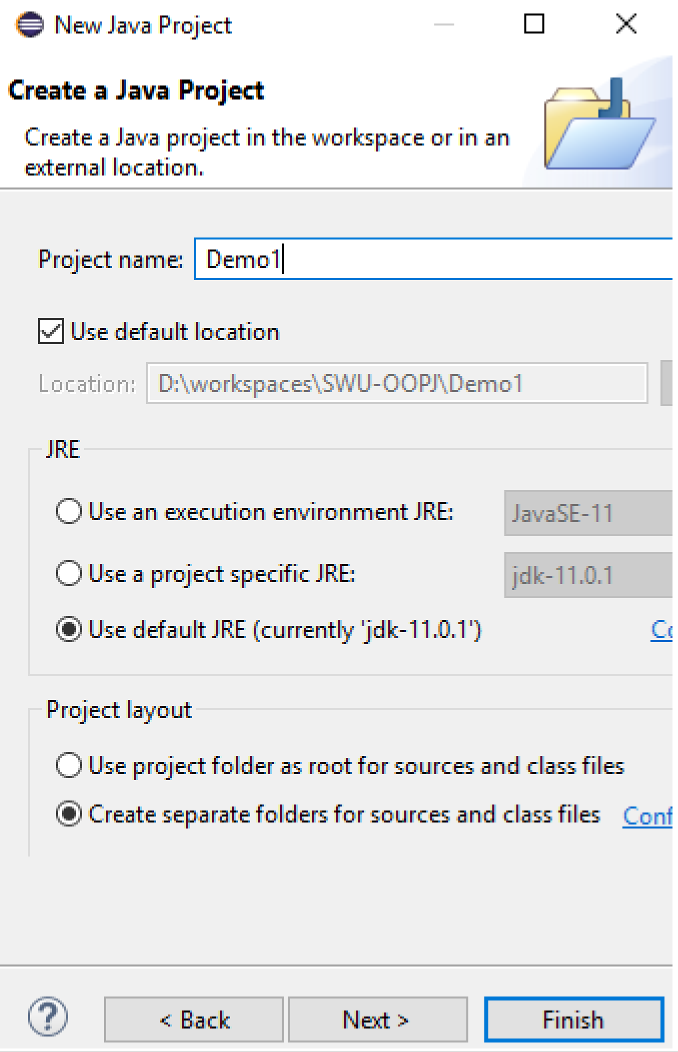
56,051 DOWNLOADS

The essential tools for any Java developer, including a Java IDE, a Git client, XML Editor, Mylyn, Maven and Gradle integration

Windows 32-bit 64-bit
Mac Cocoa 64-bit
Linux 32-bit 64-bit

Note for Eclipse users

- Skip the creation of module-info.java
  - Java 9 have introduced modules, but they would not be used during the course



New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: Demo1

☑ Use default location

Location: D:\workspaces\SWU-OOPJ\Demo1

JRE

○ Use an execution environment JRE: JavaSE-11

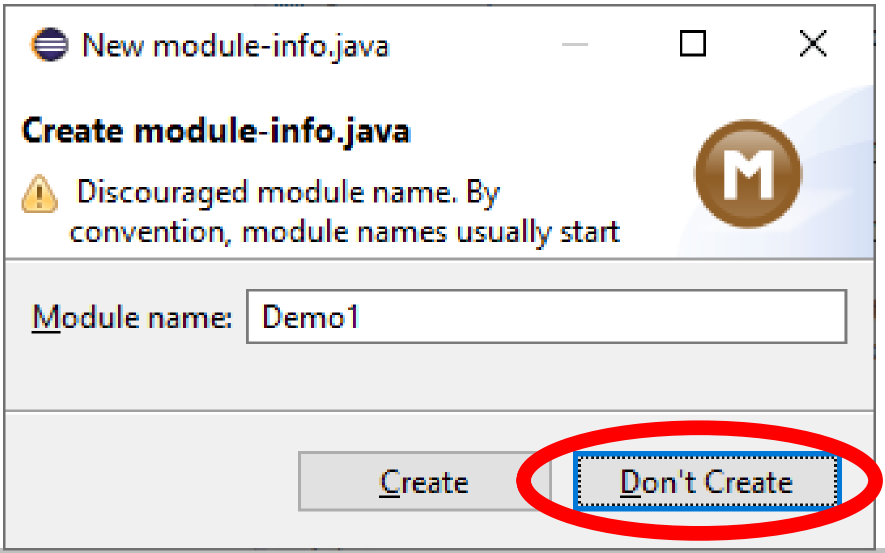
○ Use a project specific JRE: jdk-11.0.1

● Use default JRE (currently 'jdk-11.0.1')

Project layout

○ Use project folder as root for sources and class files

● Create separate folders for sources and class files



New module-info.java

Create module-info.java

Discouraged module name. By convention, module names usually start

Module name: Demo1

The first Java program – Hello World

- Create a new file and rename it to HelloWorld.java
  - If using Windows, take care about extension hiding in order to avoid file named *HelloWorld.java.txt*

```
public class HelloWorld {
  public static void main(String[] args){
    System.out.println("你好 重庆市");
  }
}
```

HelloWorld.java

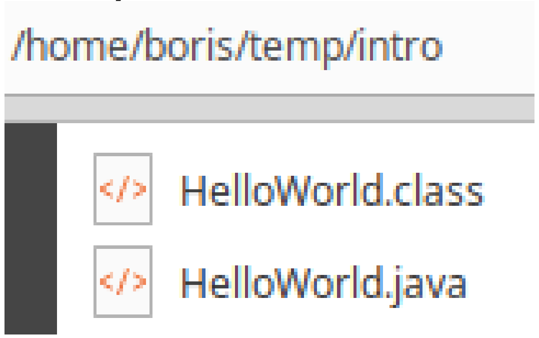
- Two basic rules
  1. The name after the keywords *public* and *class* must match the filename (the name preceding *.java*)
  2. only one *public class* per file allowed
     - direct consequence of the rule #1
  - Other rules and combinations discussed later

How to run a program – general concepts

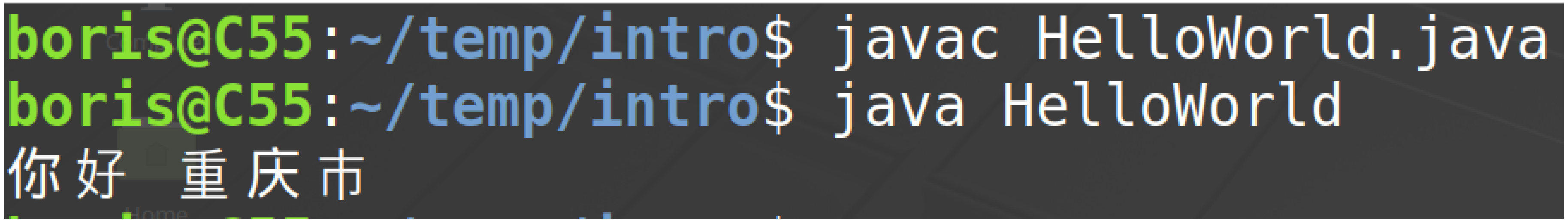
- Programmer writes source code - Computer needs machine code
  - Classic approach (e.g. C language)
    - (preprocessed) source code → (compiler compiles to) assembly code → (assembler creates) object code → (linker combines one or more object code to) → executable file or library
    - source code could be portable, other is platform dependent
  - Python, Perl, MathLab, …
    - interpreter for particular operating system interprets (and/or translate code to an efficient one) instructions from source code and runs them
    - source code could be portable
  - Java, C#
    - source code → (compiler compiles to) byte code (binary code with instruction intended for a virtual machine) → Virtual machine's Just-In-Time compiler translates bytecode to machine code and runs it
    - <u>Compiled code (byte code) is portable!</u>

How to write and run the first Java program

- Source code from .java file(s) compiled using Java compiler (*javac*) to byte code
  - intermediate language, language for virtual machine, …
- Compilation produces one or more class files
  - Not an executable file by itself
  - <u>Portable code </u>that needs Java virtual machine
    - Same class file can be copied and executed on Linux, Windows, MacOs, or any OS with appropriate Java Runtime Environment (JRE) installed
- *java* runs the code from class files
  - *.class* is omitted from command


/home/boris/temp/intro

HelloWorld.class
HelloWorld.java

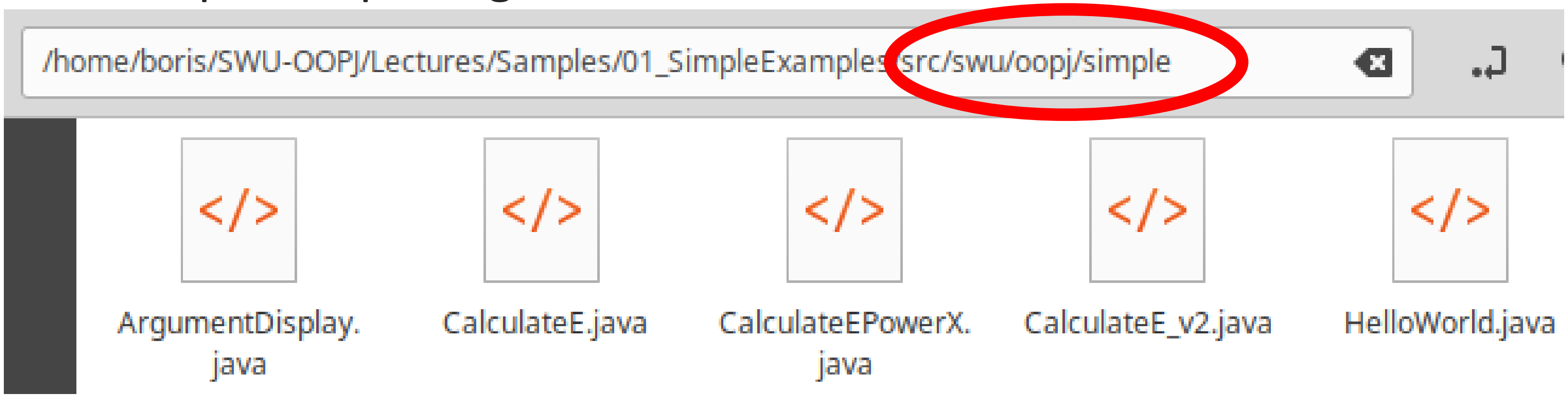

```
boris@C55:~/temp/intro$ javac HelloWorld.java
boris@C55:~/temp/intro$ java HelloWorld
你好 重庆市
```

Packages

- Source code and compiled (binary code, bytecode) should not be in the same folder
  - usually divided to *src* and *bin* folders (for source and binary code)
- Akin source code grouped into packages
  - ease maintenance and search
  - helps avoiding naming conflicts
    - e.g. what if we have several classes/files named *HelloWorld*
    - full name consists of package name + class name
- Convention for package names
  - use lower case
  - Institutions and companies usually use reversed Internet domain name (e.g. cn.edu.swu) + product name
    - For the sake of briefness we shall further name packages as swu.oopj.topicname

Folder organization when using packages

- Choose root folder as you want and create subfolders *src* and *bin*
- Each part of package name is one folder



/home/boris/SWU-OOPJ/Lectures/Samples/01_SimpleExamples/src/swu/oopj/simple

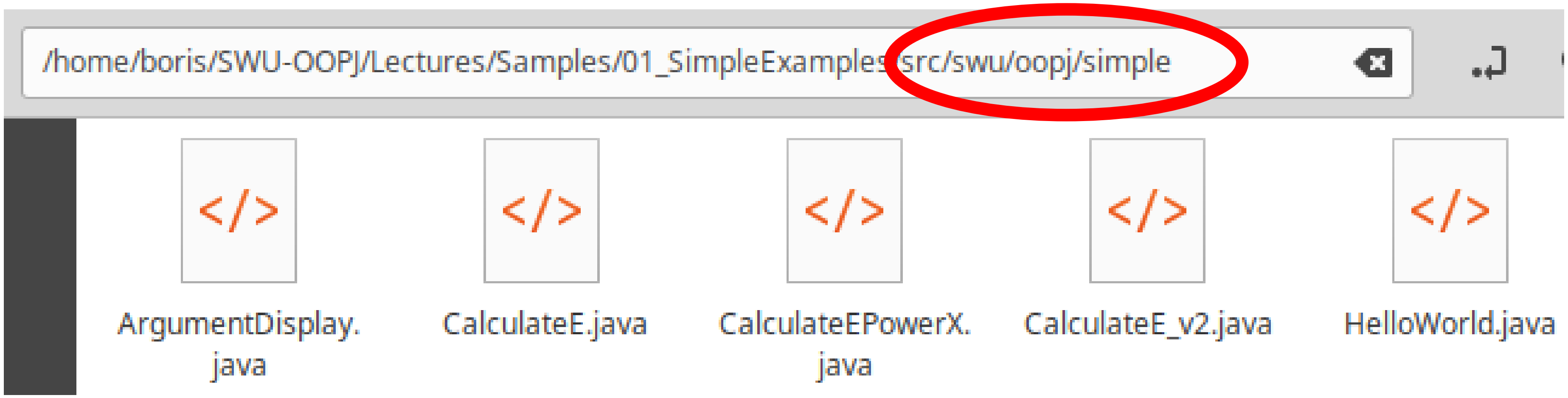
ArgumentDisplay.java CalculateE.java CalculateEPowerX.java CalculateE_v2.java HelloWorld.java

- Compiler should be run with flag –d bin in order to produce same structure for class files

Hello World – a variant with package name

- Use keyword package at the top of the file

```
package swu.oopj;
  public class HelloWorld {
   public static void main(String[] args){
      System.out.println("你好 重庆市");
   }
}
```



/home/boris/SWU-OOPJ/Lectures/Samples/01_SimpleExamples/src/swu/oopj/simple

Compilation with *src* and *bin* folders

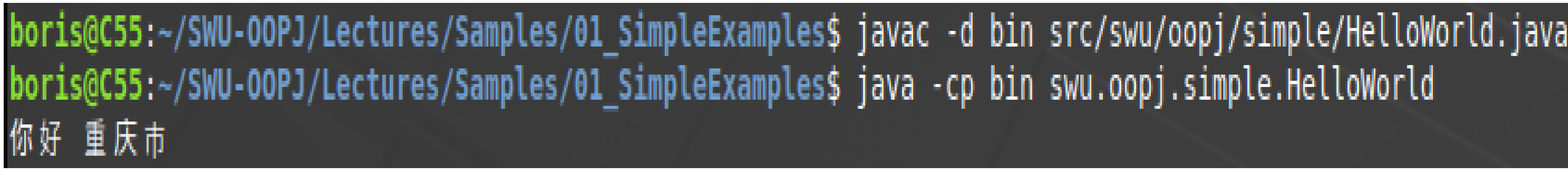
- Go to the root of the project and run

  ```
  javac -d bin src/swu/oopj/simple/HelloWorld.java
  ```

  - Parameter *–d bin* sets the destionation for compiled (binary) files
    - later we would combine it with –sourcepath parameter
- Successful compiling produces HelloWorld.class in folder …`bin/swu/oopj/simple`

- In order to run use parameter –cp to indicate where are the compiled class(es) you would like to run and provide full class name (<u>not a path to class file</u>)

  ```
  java -cp bin swu.oopj.simple.HelloWorld.java
  ```



```
boris@C55:~/SWU-OOPJ/Lectures/Samples/01_SimpleExamples$ javac -d bin src/swu/oopj/simple/HelloWorld.java
boris@C55:~/SWU-OOPJ/Lectures/Samples/01_SimpleExamples$ java -cp bin swu.oopj.simple.HelloWorld
你好 重庆市
```

Java language basics

- Java syntax style is similar to C language style
  - definition of variables
    - statically-typed (all variables must be declared before use)
    - variable naming
    - similar primitive types
  - blocks with curly braces
  - loops (for, while, do-while) and decision-making statements (if-else, switch)
    - an exception are logical conditions: separate Boolean type instead zero/non-zero for false and true
  - syntax of function definition
    - in Java term *method* is used instead of *function*
- If not familiar with C-like style, please read https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html

Primitive (Basic) Java Types

Type	Size in bytes	Default values	Range
byte	1	0	-128 to 128
short	2	0	-32768 to 32767
int	4	0	-2 147 483 648 to 2 147 483 647
long	8	0L	-9 223 372 036 854 775 808 to 9 223 372 036 854 775 807
char	2	'\u0000'	0 to 65 536 (unsigned) - UTF-8 encoding
boolean	?	false	true or false
float	4	0.0f	approximately ±3.40282347E+38F (6-7 significant decimal digits) Java implements IEEE 754 standard
double	8	0.0d	approximately ±1.79769313486231570E+308 (15 significant decimal digits)

An example: Calculate Euler's number *e*

e = 2.7182818284590452353602874713527…

- Can be approximated by taking first n elements of Taylor series sum

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

for *x* = 1

An example: Calculate Euler's number *e*

- Calculation moved to the separate method $e = \sum_{i=0}^{\infty} \frac{1}{i!}$

```
package swu.oopj.simple;
public class CalculateE {
  public static void main(String[] args) {
       double sum = ePowerX(1);
       System.out.printf("e = %.6f%n", sum);
  }
  public static double ePowerX(double x) {
       double power = 1.0;   double factorial = 1.0;
       double sum = 1.0;
       for(int i = 1; i < 10; i++) {
              power = power * x;
              factorial = factorial * i;
              sum += power/factorial;
       }
       return sum;
  }
}
```

CalculateE.java

Moving parts of the code to another files

- *ePowerX* could be useful for some future programs
  - Let’s move it to a new file that belongs to swu.oopj.*util* package

```
package swu.oopj.util;
public class Taylor {
  public static double ePowerX(double x) {
        double power = 1.0;
        double factorial = 1.0;
        double sum = 1.0;
        for(int i = 1; i < 10; i++) {
               power = power * x;
               factorial = factorial * i;
               sum += power/factorial;
        }
        return sum;
  }
}
```

…01_SimpleExamples/src/swu/oopj/util/Taylor.java

How it affects our previous program

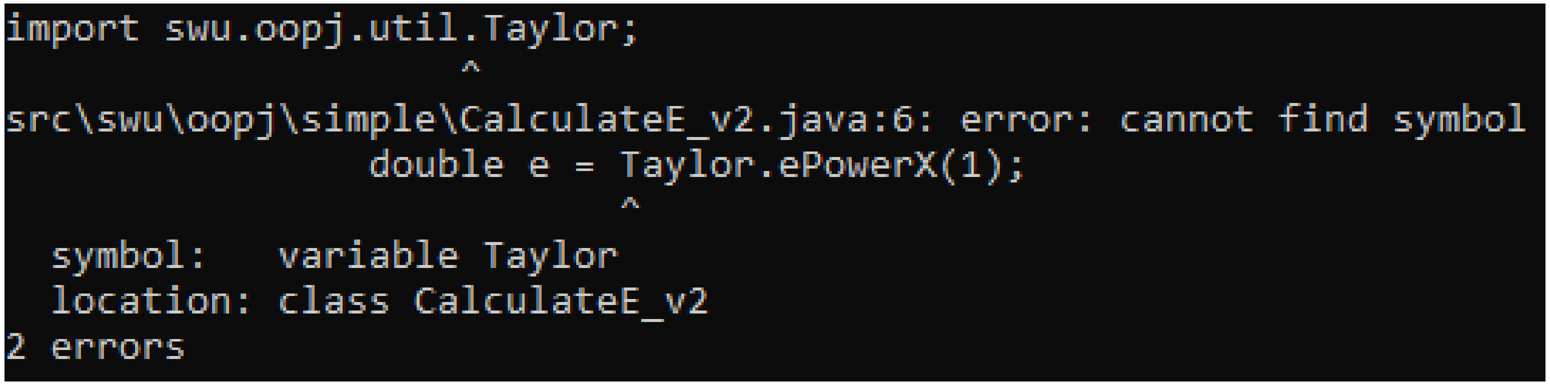
- Method ePowerX is not anymore in the same file
  - Now it belongs to class with a name *Taylor*
  - Class *Taylor* is not in the same package
    - Need to import it (tells compiler that we would like to use it)
  - Same applies to mathematical functions and constants. They belong to class *Math* (package *java.lang* that does not have to be imported)

```
package swu.oopj.simple;
import swu.oopj.util.Taylor;
public class CalculateE_v2 {
  public static void main(String[] args) {
       double e = Taylor.ePowerX(1);
       System.out.printf("e = %.6f%n", e);
       double diff = Math.abs(Math.E - e);
       System.out.printf("diff = %g%n", diff);
  }
}
```

…01_SimpleExamples/src/swu/oopj/simple/CalculateE_v2.java

Compile code from several source code files

- Command `javac -d bin src/swu/oopj/simple/CalculateE_v2.java` produces an error



```
import swu.oopj.util.Taylor;
                    ^
src\swu\oopj\simple\CalculateE_v2.java:6: error: cannot find symbol
                double e = Taylor.ePowerX(1);
                           ^
  symbol:   variable Taylor
  location: class CalculateE_v2
2 errors
```

- Taylor.java is in another package (different folder)
  - We need to use parameter *sourcepath*

  javac ***-sourcepath src*** *-d bin src/swu/oopj/simple/CalculateE_v2.java*

- Note: It does not change run command

  `java -cp bin swu.oopj.simple.CalculateE_v2`

Comments and documentation

- Comment code to help others
  - reading and reviewing
    - Simple comments
      - /* multi-line comments */
      - // one-line comment
  - reusing code
    - Special JavaDoc comments for classes and methods
      - /** *comments with text and special tags* */
- Some of JavaDoc tags
  - @author, @version, @param, @return, …

An example of JavaDoc

```
package swu.oopj.util;

public class Taylor {
  /**
   * Calculates e^x for Taylor series, according to formula:
   * e^x=1+x+(x^2/(2!))+(x^3/(3!))+(x^4/(4!))+...
   * @param x argument of function e^x
   * @return e^x calculated as sum of first 10 numbers in Taylor
series.
 */
   public static double ePowerX(double x) {
       double sum = 0.0;
       ...
```

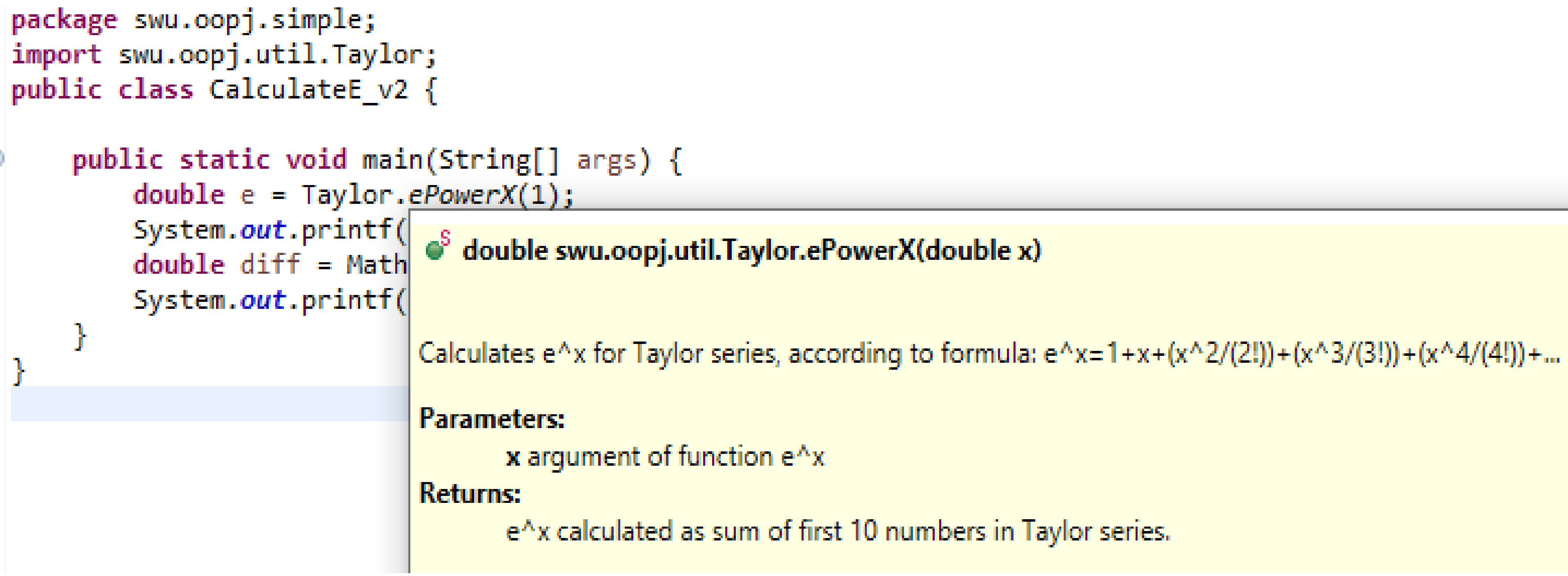
- Command

  `javadoc -sourcepath src swu.oopj.util -d docs`

  creates HTML files with Java documentation of our classes

Using custom JavaDoc inside an IDE

- JavaDoc comments helps writing code inside an IDE by showing methods (classes, parameters, ...) descriptions in the same manner as for built-in classes



```
package swu.oopj.simple;
import swu.oopj.util.Taylor;
public class CalculateE_v2 {

    public static void main(String[] args) {
        double e = Taylor.ePowerX(1);
        System.out.printf(
        double diff = Math
        System.out.printf(
    }
}
```

double swu.oopj.util.Taylor.ePowerX(double x)

Calculates e^x for Taylor series, according to formula: e^x=1+x+(x^2/(2!))+(x^3/(3!))+(x^4/(4!))+...

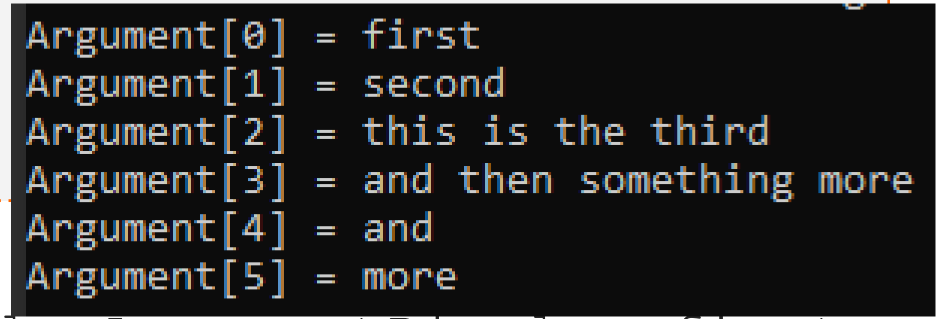
Parameters:
　　x argument of function e^x

Returns:
　　e^x calculated as sum of first 10 numbers in Taylor series.

Program arguments

- Arguments stored as array of Strings
  - Valid indices for arrays are from 0 to array length - 1
  - String is (in general) sequence of characters

```
package swu.oopj.simple;
public class ArgumentDisplay {
   public static void main(String[] args) {
      int argCount = args.length;
      for(int i = 0; i < argCount; i++) {
         System.out.printf("Argument[%d] = %s%n", i, args[i]);
      }
    }
}
```



```
Argument[0] = first
Argument[1] = second
Argument[2] = this is the third
Argument[3] = and then something more
Argument[4] = and
Argument[5] = more
```

```
java -cp bin swu.oopj.simple.ArgumentDisplay first
second "this is the third" "and then something
more" "and" "more"
```

Extracting number from a string

- If a string contains only digits (and decimal point) as characters, then it can be *parsed* in order to get a number stored inside
  - Some typical examples

    `Integer.parseInt("1232")` - returns an *int* with value 1232

    `Double.parseDouble("3.14")` – returns *double* with value 3.14
  - If the string content cannot be parsed it would produce an error
    - Precisely it would produce an exception that breaks the normal execution of the program
      - it would be explained in topic T7 – Exceptions
    - E.g. *Integer.parseInt("12w")* breaks the normal program execution
- Parsing does not change a string!

Using program argument for *x* in e^x

```
package swu.oopj.simple;
import swu.oopj.util.Taylor;
public class CalculateEPowerX {
  public static void main(String[] args) {
    if (args.length != 1) {
       System.out.println("The program needs an integer value x to
calculate e^x.");
       System.exit(1); //exit program with error code 1
    }
    int x = Integer.parseInt(args[0]);
    double result = Taylor.ePowerX(x);
    System.out.printf("e^%d = %.6f%n", x, result);
    double diff = Math.abs(Math.pow(Math.E, x) - result);
    System.out.printf("diff = %g%n", diff);
  }
}
```