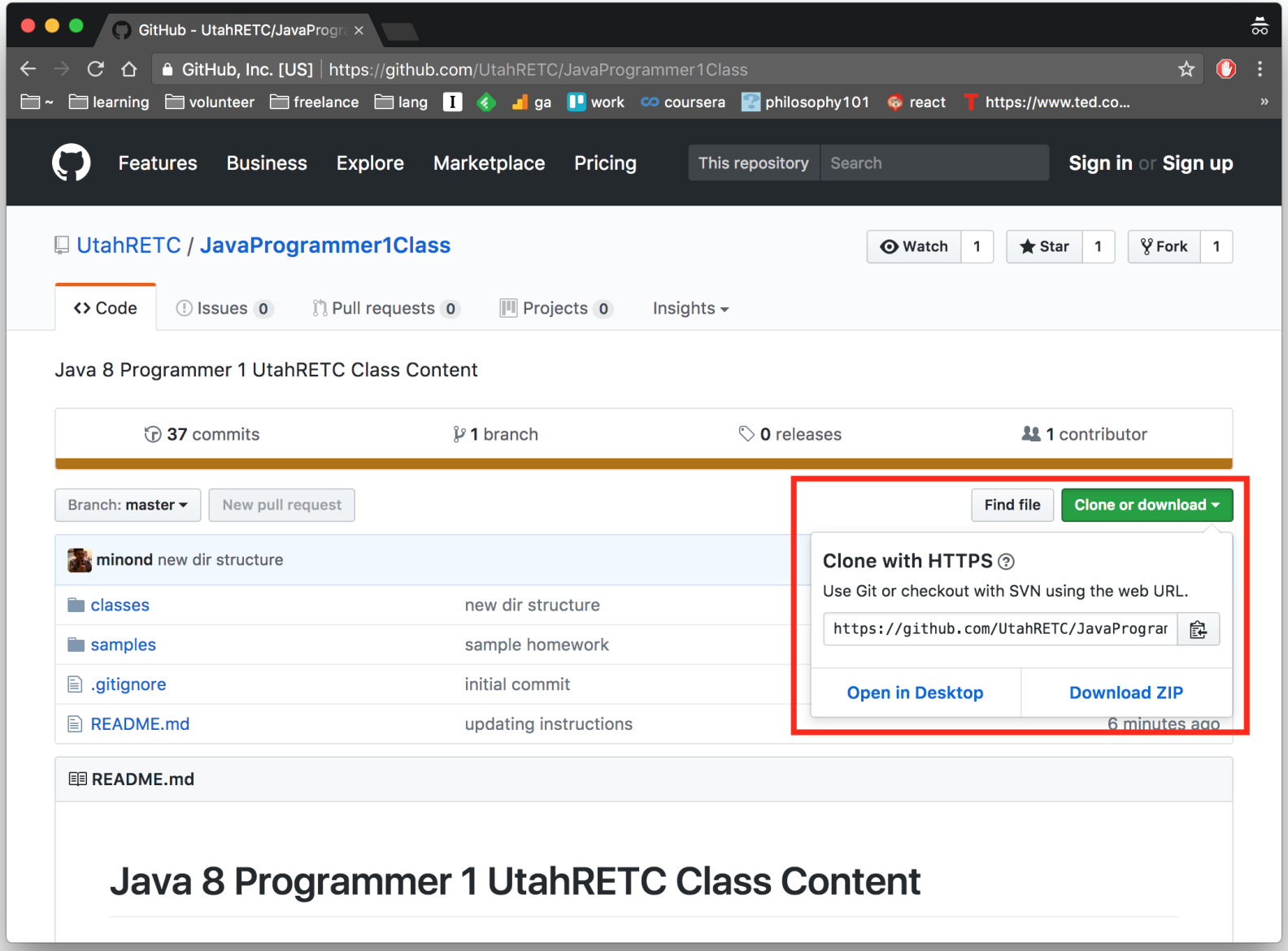
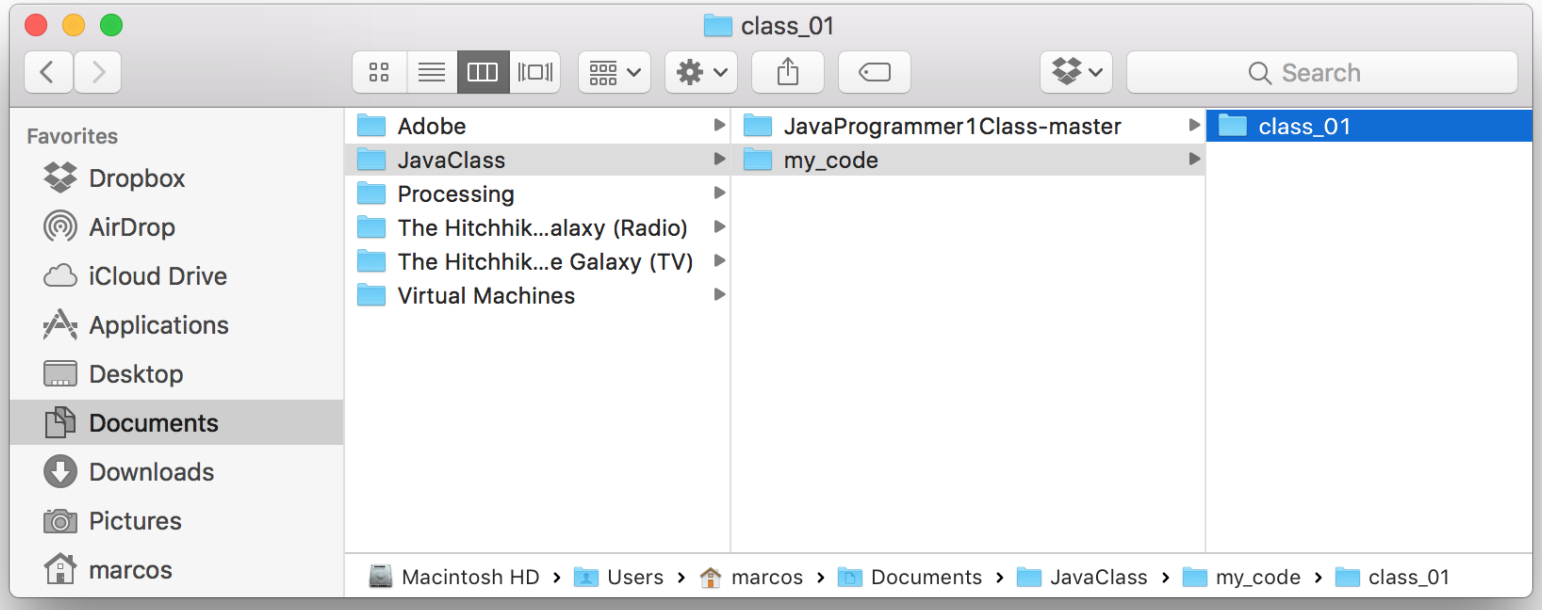
Class 1

- Downloading class content
- Terminology and a few commands
- How you should read these slides
- Hello, World
- Introduction to variables and data types
- Operators
- User input
- Dealing with problematic code

Class content can be downloaded by going to tiny.cc/urect-java

Download the files by clicking on "Clone or download" then click "Download ZIP".


GitHub - UtahRETC/JavaProgr
GitHub, Inc. [US] | https://github.com/UtahRETC/JavaProgrammer1Class
learning volunteer freelance lang ga work coursera philosophy101 react https://www.ted.co...
Features Business Explore Marketplace Pricing
This repository Search
Sign in or Sign up
UtahRETC / JavaProgrammer1Class
Watch 1 Star 1 Fork 1
Code Issues 0 Pull requests 0 Projects 0 Insights
Java 8 Programmer 1 UtahRETC Class Content
37 commits 1 branch 0 releases 1 contributor
Branch: master New pull request
Find file Clone or download
minond new dir structure
classes new dir structure
samples sample homework
.gitignore initial commit
README.md updating instructions 6 minutes ago
Clone with HTTPS
Use Git or checkout with SVN using the web URL.
https://github.com/UtahRETC/JavaPrograr
Open in Desktop
Download ZIP
README.md
Java 8 Programmer 1 UtahRETC Class Content



You can do this however you want to, but the way I did it was I created a `JavaClass` folder in my `Documents` folder and in there I put the extracted content I downloaded from Github and created a `my_code` folder as well. I also created a `class_01` where I will put all of the code I write for this class.

Terminology and commands

Before we start writing code, let's cover a few terms, commands, and other useful information that you will need to know to effectively continue with the next sections.

Code editor: this is a program used to write anything, including code. Examples include Nodepad, Notepad++, Sublime Text, Atom, IntelliJ. If you're unsure about white editor to use, start with Atom.

What ever you use, make sure you disable syntax highlighting.

Terminal/Command Prompt: this is a program used to *run* other programs. Your computer already has this program installed.

- To open the command prompt in Windows, click on the Start menu and search for "*command*".
- To open the terminal in Macos, open Spotlight `Cmd + Space` and search for "*terminal*".

Command: commands are programs you run in your terminal.

"Running a command": when the phrase "*running a command*" or "*run this command*" is used, the intention is that you open your command prompt, carefully type in the example command, and press the Enter key to run it.

Compile: or "*compiling*" is the act of taking code that you wrote and preparing it so that your computer can run it. Not all programming languages need to be compiled, but Java does.

Commands

- `cd` : lets you move from one folder to another folder. Remember the phrase "*change directory*" (change = c, directory = d)
- `dir` (Windows) or `ls` (Macos/Linux): this command is used to list all of the files and folders in the folder you are currently in.
- `pwd` (Macos/Linux): this command will tell you what folder you are on right now. Useful when you want to `cd` into another folder.
- Let's practice.

Commands

- `javac` : the Java compiler. For example, if you had a file named "HelloWorld.java" and you wanted to compiled it, you would do so by running `javac HelloWorld.java` in your terminal/command prompt.
- `java` : another Java related command. This one is used to actually run your compiled program. If you wanted to run the HelloWorld program from the example above, you would run `java HelloWorld` .

Reading these slides: code

Below is an example of how code is represented in these slides. Notice the gray background color. Let's practice writing this code in your text editor.

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

Reading these slides: commands

Commands will look a lot like code with the exeption that they will have a ">" at the beginnig of the line. Let's practice running these commands.

```
> javac HelloWorld.java
> java HelloWorld
```


You're a Java Programmer!!!

Reading these slides: inline examples

Sometimes you'll see text formatted `like this`. This can either be a command or a line of code. If you're unsure about which one it is, just ask.

Reading these slides: references

If you see a number surrounded by square brackets (like this [123]) it's a reference. The last slide in this in this slide deck will have a list of all of the resources used in the slide and a reference number as well. Sometimes references will go into more detail than the slides or teacher does, so it can be useful to check them out.

Understanding Hello, World

Let's revisit the code that we wrote earlier in depth.

```java
/**
 * A HelloWorld example class that prints          // (1)
 * out a message to the user.
 */
public class HelloWorld {                          // (2)
  public static void main(String[] args) {         // (3)
    // print out Hello, World                      // (4)
    System.out.println("Hello, World!");           // (5)
  }
}
```

Change something about your code and see what happens. What happens if you change the string *"Hello, World"*? Do you see your updates? What happens if you remove the word void? Did you get an error? Make any change you'd like. Below is a reference to the original code if you want to undo any changes you make.

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

`HelloWorld.java` and `HelloWorld.class`

List the contents in the directory (`ls` in Macos and `dir` for Windows). You should see `HelloWorld.class`. Notice there are two files here, `HelloWord.java` and `HelloWorld.class`. `HelloWorld.java` is the file you wrote. `HelloWorld.class` is the result of running the `javac` command on that file. It's contents are called "bytecode".

Java code is for humans and bytecode is for computers. You won't have to know much about bytecode to be a really great Java programmer or to do well in the exam.

Bytecode for HelloWorld.java example

```
public class HelloWorld {
  public HelloWorld();
    Code:
       0: aload_0
       1: invokespecial #1 // Method java/lang/Object."<i
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2 // Field java/lang/System.out:
       3: ldc           #3 // String Hello, World!
       5: invokevirtual #4 // Method java/io/PrintStream.
       8: return
}
```

Workflow for writing and running Java.

1. Write your Java code in `FileName.java`
2. Compile using `javac FileName.java`
3. Run your program using `java FileName`

Variables

> A variable is a symbolic name for (or reference to) information. The variable's name represents what information the variable contains. They are called variables because the represented information can change but the operations on the variable remain the same. [1]

Three things are required to create a new variable in Java: the data type, a name, and an optional value.

A valid variable name can be made up of letters, numbers, dollar signs `$` and underscores `_`. They should always begin with letters, however (`$` and `_` are valid characters at the beginning of variable names, but this should be avoided.)

```
String activityName = "Java programming";
int age = 47;
double pi = 3.14159;
int favoriteNumber;
```

Additional terminology

Declaration: as previously mentioned, a value is optional when creating a variable. When a variable is created and it does not have a value, you are *declaring* the variable, meaning you are telling your program that this variable exists and it is information that should be tracked, but that you do not yet know what the value is and will set it after the program starts running. Keep in mind that the variable's data type is only required when it is being *declared*.

```
int x;
```

Definition: a variable is *defined* when it is given a value. As shown in earlier examples, a variable can be declared and defined in one step, but that is not required. The first example below declares and then defines `x` and the one below that does the same but in one step:

```
int x;
x = 7;
```

```
int x = 7;
```

Data types

What's a type?

Remember the code from earlier?

```
String activityName = "Java programming";
int age = 47;
double pi = 3.14159;
```

- `byte`, `short`, `int`, and `long` are for integers, such as `1`, `99`, and `32423325`
- `float` and `double` are for floating-point numbers, such as `3.14`, `3.1`, and `89.12e32`
- `char` is for a single character, such as `a`, `B`, and `\n`
- `boolean` is for storing either `true` or `false` (think of this as on/off or yes/no)
- `String` is for a sequence of characters, or a string. For example, `Hello, World!`

- `byte` = 1 Byte, signed, -128 to 127
- `short` = 2 Bytes, signed, -32,768 to 32,767
- `int` = 4 Bytes, signed, -2^{31} to $2^{31}-1$
- `long` = 8 Bytes, signed, -2^{63} to $2^{63}-1$
- `float` = 4 Bytes, signed
- `double` = 8 Bytes, signed
- `char` = 2 Bytes, unsigned
- `boolean` = 1 Byte, unsigned

Arrays

Arrays are lists. In order to use an Array, you must declare the type as an Array and you must declare the type of the values the array will hold.

```
public class ArraySample {
  public static void main(String[] args) {
    String[] myFriends;

    myFriends = new String[3];

    myFriends[0] = "Ryan";
    myFriends[1] = "Andi";

    System.out.println(myFriends[0] + " is my friend.");
    System.out.println(myFriends[1] + " is my friend.");
  }
}
```

What happens if I use the wrong type?

```
public class BigByte {
  public static void main(String[] args) {
    byte oneTwentySeven = 127;
    byte oneTwentyEight = 128;

    System.out.println(oneTwentySeven);
    System.out.println(oneTwentyEight);
  }
}
```

Create a new file, write this program, and compile it. What happens?

Why do Data Types exists?

You should use the smallest amount of data possible to accomplish a task, so that your computer doesn't run out of memory.

Operators

You're already using operators

```
String variableName = "string variable value";
```

Assigned Operator

- `=`

```
int age = 92;

age = 93;
age = age + 1;
```

Arithmetic Operators

- `*`
- `/`
- `%`
- `+`
- `-`

Arithmetic Operators

```
int a = 37;
int b = 21;

int multiplication = a * b;
int division = a / b;
int modulos = a % b;
int addition = a + b;
int subtraction = a - b;
```

Compound Operators

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`

```
int age = 92;

age += 1;
```

Unary Operators

- `++`
- `--`

```
int age = 92;

age++;
```

Pre-unary vs. post-unary: Pre-unary

```
public class PreUnaryOperators {
  public static void main(String[] args) {
    int age = 92;
    int nextAge = ++age;

    System.out.println("Age = " + age);
    System.out.println("Next Age = " + nextAge);
  }
}
```

```
Age = 93
Next Age = 93
```

Pre-unary vs. post-unary: Post-unary

```
public class PostUnaryOperators {
  public static void main(String[] args) {
    int age = 92;
    int nextAge = age++;

    System.out.println("Age = " + age);
    System.out.println("Next Age = " + nextAge);
  }
}
```

```
Age = 93
Next Age = 92
```

Relational Operators

- `<`
- `>`
- `<=`
- `>=`

```java
public class RelationalOperators {
  public static void main(String[] args) {
    boolean is5LessThan5 = 5 < 5;
    boolean is5LessThanOrEqualTo5 = 5 <= 5;

    System.out.println("Is 5 less than 5? "
      + is5LessThan5);

    System.out.println("Is 5 less than or equal to 5? "
      + is5LessThanOrEqualTo5);
  }
}
```

```
Is 5 less than 5? false
Is 5 less than or equal to 5? true
```

Equality Operators

- `==`
- `!=`
- `.equal()`

```java
public class EqualityOperators {
  public static void main(String[] args) {
    String stringThree = "3";
    int intThree = 3;
    double doubleThree = 3;

    boolean isStringThreeEqualToIntThree =
      stringThree.equals(intThree);

    boolean isInt3EqualToDouble3 =
      intThree == doubleThree;

    System.out.println("Is String 3 equal to Int 3? "
      + isStringThreeEqualToIntThree);

    System.out.println("Is Int 3 equal to Double 3? "
      + isInt3EqualToDouble3);
  }
}
```

```
System.out.println("Is String 3 equal to Int 3? "
  + "three".equals(3));

System.out.println("Is Int 3 equal to Double 3? "
  + (3 == 3.0));
```

```
Is String 3 equal to Int 3? false
Is Int 3 equal to Double 3? true
```

Operator Precedence [2]

From left to right:

1. Parentheses `()`
2. Multiplicative `* / %`
3. Additive and string concatenation `+ -`
4. Relational `< <= > >=`
5. Equality `== !=`
6. Assignment `= += -= *= /= %=`

Operator Precedence Examples

```
5 + 4 * 3 = 17
```

```
"abc" + 1 + 2 = "abc12"
```

```
1 + 2 + "abc" = "3abc"
```

User Input

Now that we have written a program which has output and know much more about variables, let's write some code to work with user input.

Commands can take arguments, so let's use an argument to make our Hello World program have the ability to say "Hello" to more than just "World" depending on if we run our program with an argument or not.

New Terminology!

Command arguments/parameters: sometimes you want to pass custom information to a command. This information is placed after the command name and is refered to as an argument or a parameter. For example, the command to compile a Java file is `javac` and this command takes a file name as one of its arguments.

So if I wanted to compile a file named "HelloWorld.java" I would run `javac HelloWorld.java` where `javac` is the command and `HelloWorld.java` is the argument.

Input and output: there are different forms of input and output, but for the next sections, we mean input to be custom arguments you pass to your Java programs and output to be the text that you see after running a command.

In your text editor, create a new file and write the program below. Name this file "*HelloWorldWithArguments.java*" and save it in your "*class_01*" folder.

```
public class HelloWorldWithArguments {
  public static void main(String[] args) {
    String name = "World";

    if (args.length > 0) {
      name = args[0];
    }

    System.out.println("Hello, " + name + "!");
  }
}
```

Notice some new things?

```
public class HelloWorldWithArguments {
  public static void main(String[] args) {
    String name = "World";

    if (args.length > 0) {
      name = args[0];
    }

    System.out.println("Hello, " + name + "!");
  }
}
```

Go to your terminal and run the following commands (make sure you're in the right folder)

```
> javac HelloWorldWithArguments.java
> java HelloWorldWithArguments Programmer
```

If everything worked you should see `Hello, Programmer!` as the output of running the last command. Try running the last command using your name!

Dealing with problematic code

You're going to make mistakes. This is OK.

Understanding compiler output

The compiler should become your friend. Learning how to understand what it is telling you will help accomplish this. When your code fails to compile, most of the time you will be given a very explicit reason why.

For example, try compiling `CompileError.java`. You should see something like this:

```
CompileError.java:3: error: cannot find symbol

    System.out.prntln("This is a string");
               ^
  symbol:   method prntln(String)
  location: variable out of type PrintStream

1 error
```

What can we get out of this?!?!?!

1. The file that the compiler had problems in: `CompileError.java`
2. The line number in that file: `3`
3. The actual "thing" that is causing problems:

```
    System.out.prntln("This is a string");
               ^
  symbol:   method prntln(String)
  location: variable out of type PrintStream
```

4. Finally, the number of errors it found: `1 error`. Keep in mind that as you fix errors other may arise.

Understanding runtime errors

The Java compiler cannot predict every error in code.

Compile and execute `RuntimeError.java` and let's talk about what's happening.

Running `javac RuntimeError.java` results in no errors, but when you run `java RuntimeError` you should see something like this:

```
Exception in thread "main" java.lang.NumberFormatExceptio
        at java.lang.NumberFormatException.forInputString
        at java.lang.Integer.parseInt(Integer.java:580)
        at java.lang.Integer.parseInt(Integer.java:615)
        at RuntimeError.main(RuntimeError.java:3)
```

What can we get out of this?!?!?!

1. The file with the problem: `RuntimeError.java`
2. The line number in that file: `3`
3. The exception that was thrown `NumberFormatException`
4. The stack trace:

```
at java.lang.Integer.parseInt(Integer.java:580)
at java.lang.Integer.parseInt(Integer.java:615)
at RuntimeError.main(RuntimeError.java:3)
```

New Terminology!

Exception: we'll cover this topic in much more detail in a later class, but for the time being think of exceptions as your code telling you that it ran into an issue. Different exceptions mean different issues, so this will help you get to the bottom of the problem.

Stack trace: stack traces are related to Exceptions and they include information about the files/lines that your code ran through before it had into an error. Stack traces will help you find the exact location (file and line number) of the error.

Additional resources

- *Introduction and Setting up Your Environment* by Jim Wilson
  https://app.pluralsight.com/player?course=java-fundamentals-language&author=jim-wilson&name=java-fundamentals-language-m1&clip=0
- *Variables, Data Types, and Math Operators* by Jim Wilson
  https://app.pluralsight.com/player?course=java-fundamentals-language&author=jim-wilson&name=java-fundamentals-language-m3&clip=0

Reference list

1. Germain, https://www.cs.utah.edu/~germain/PPS/Topics/variables.html
2. Operator Precedence in Java, http://introcs.cs.princeton.edu/java/11precedence/