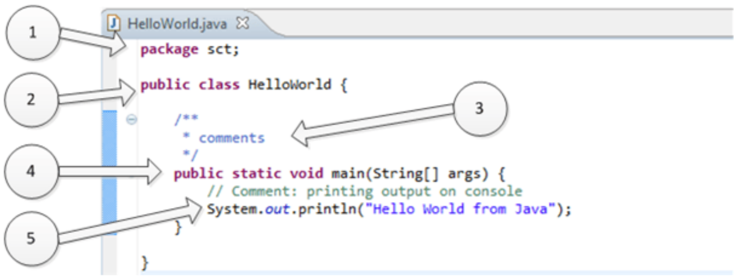
CS20B: Data structures with Java

prof. dr. Irma Ravkic

1 Activity 1: Hello World

In this activity we will first practice how to compile and run Java programs on your operating system. To do that, we are going to use Google!

1. Identify your operating system (either in the classroom or your laptop).
2. Go to Google and write an appropriate query for running Java on your computer.
3. Now as you see, you need an actual Java program in order to see if it works. We will make a very simple program called "HelloWorld.java" and you just make it print "Hello, World!" into the console. See Figure 1 to remind yourselves how to do it. Write the Java code in a text editor of your liking and call it "HelloWorld.java".



```java
package sct;

public class HelloWorld {

    /**
     * comments
     */
    public static void main(String[] args) {
        // Comment: printing output on console
        System.out.println("Hello World from Java");
    }

}
```

Figure 1: Hello World program in Java.

4. Now run your Java program as it is specified in your Google tutorial. It probably tells you to go to Terminal and run these commands:
   - javac HelloWorld.java (this will compile your java file)
   - java HelloWorld (this will run your program)

   (make sure you put all your Java code in appropriate folders, don't just pile them up in a single folder like Downloads, Pictures and similar). Ideally your path would be something like this: Home//Documents//SMC//CS20B// Class_activity//Activity_1//HelloWorld// (don't put spaces in the names of your folders/files)

2 Activity 2: getting familiar with a Java IDE

Well, writing Java code in a text editor is not so fun since you don't have syntax highlighting. IDE stands for (Integrated Development Environment), and can give you a more compact and visual development environment. Java, like other programming languages, has a couple of the favorites ones. In this activity we are going to identify them, install them, and simply use them.

1. Write in Google an appropriate query to see which IDEs are listed for Java.
2. I will focus on Eclipse. So go ahead and Google how you can download it on your laptop. Otherwise, you're free to use another IDE. If you have one, skip this step, and go to the next one.
3. Open Eclipse, and open the file. Can you just compile it and run it? Nope. In Java you need to create a project in order to compile files. So let's do it!
4. Create new project in Java. File → New → Java Project. Choose some reasonable path and folder name, give your project a meaningful name like "HelloWorld" and there you are. Go to that folder and see what was created.
5. How would you now copy your HelloWorld.java code we already compiled and ran in terminal, and make it appear in your Eclipse? Do it. Once you see it in your Eclipse file directory on the left, click on it and run it. It should print *Hello, World!* in the console below. If it doesn't work, or you're stuck, let me know.

If you need more help with Eclipse, check this out:
https://www.youtube.com/watch?v=23tAK5zdQ9c

3 Activity 3: loops and conditionals

Write another program in your IDE (not in your HelloWorld.java) that takes an array/list of numbers and counts how many even numbers there are in the list. You can also try to compile/run it using the command line.

4 Activity 4: remember the classes?

Not every class needs to have a main function! You can write classes without a main function, and those classes and their functionalities can be used in other **application** or **driver** classes that have a main function. We will practice that in this activity.

A class defines the structure of an object, or a set of objects. So, it's not surprising then that we will be using classes to implement data structures in this course. In other programming languages, such as Python, classes are also used to accomplish this. Simply, classes are data types that allow for complex and structured data. Compare this to, for example, integers, double or character types. They are not structured, and we call them "primitive" data types. An integer representing number 3 is just 3 (well it's more complicated than that on the lower/machine lewel). Now think about **complex numbers**. Are they a primitive data type? Why yes/not?

1. Create a new project called **Activity1** (again placed in a meaningful folder). Then create a package called **numbers**. Then create a Java file representing your Complex class in the **numbers** package. I chose a simple **Complex.java** name (remember: the convention is to write class names with first letter uppercase, and the names should be meaningful).

2. Attributes of your class: What kind of class members do you need? What are their types? What is their visibility (public, private, protected)? Ask yourself this question: what is the main **attribute** of Complex numbers? What are the properties of a number that makes it a complex number? Look at the definition in Wikipedia if needed. If you cannot think about any property, then think of some similar classes: Fractions, Integer, Line,... What are their main properties? Remember the visibility of class members (see Figure 2).

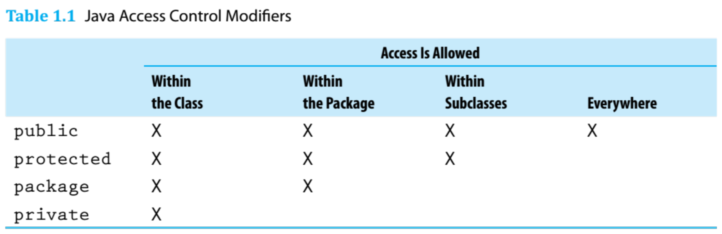
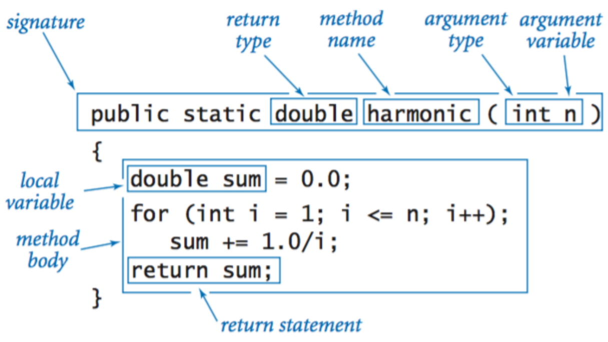


Table 1.1 Java Access Control Modifiers

	Access Is Allowed			
	Within the Class	Within the Package	Within Subclasses	Everywhere
`public`	X	X	X	X
`protected`	X	X	X	
`package`	X	X		
`private`	X			

Figure 2: Possible visibility / access modifiers of class members

3. Functionality of your class: What can I do with one, two or more complex numbers? Ask yourself, what is the behavior of complex numbers? What do we do with them? What are the allowed operations and what is their outcome? (think of Integers) Implement two operations of your choice for Complex numbers. Remember how method signature looks like in Java (Figure 3).



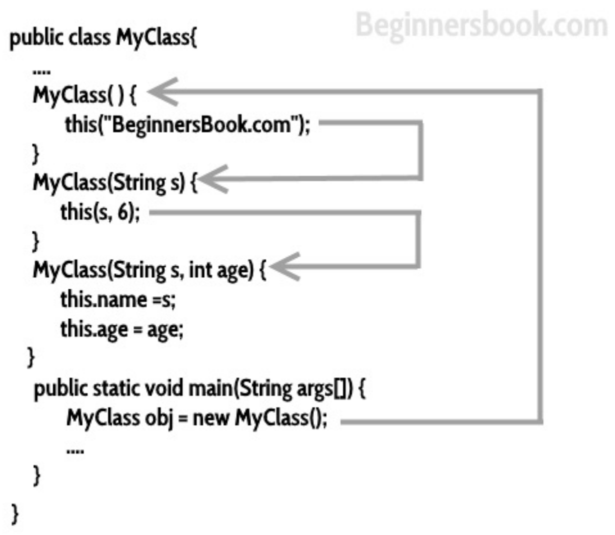
```
public static double harmonic ( int n )
{
   double sum = 0.0;
   for (int i = 1; i <= n; i++);
      sum += 1.0/i;
   return sum;
}
```

Figure 3: Visualizing the components of functions.

4. By now you should have a skeleton of your class. Ok, now let us put it to use. Create a new class called **ComplexDriver.java** and put it in a new package called **apps**. Because we're going to actually run this code, it needs a main method (your program needs a main method to run).

5. Create one object of your Complex number. Hmm...? how do I create objects of my class?

   You need a **constructor** in your Complex class to tell it how to make its own objects! Take a look at the example in Figure 4. As you see,



```
public class MyClass{
   ....
   MyClass( ) {
       this("BeginnersBook.com");
   }
   MyClass(String s) {
       this(s, 6);
   }
   MyClass(String s, int age) {
       this.name =s;
       this.age = age;
   }
   public static void main(String args[]) {
       MyClass obj = new MyClass();
       ....
   }
}
```

Figure 4: Visualizing the components of functions.

   you can have multiple ways of creating your objects. What information do you need to create your complex numbers? Make your constructor(s), and then create two complex number objects in your driver class.

6. What happens when you try to print your Complex number out onto the console? You don't get what you expect, right? In order to do that, we need to use a mechanism called method **overriding**. Remember that all classes in Java inherit from the "mothership" of classes: **Object** class. So when you do *System.out.println(myObject)*, the toString() method is automatically invoked in the class that has the definition of *toString()* method. Does your Complex class have it? No. In that case the compiler goes one level up and checks if there's *toString()* method in the **superclass** of *Complex*. Your class doesn't inherit from any class you made, which means it only inherits from *Object* class. Since Object class doesn't know anything about what your Complex number needs to look like, you need to override toString() method in your Complex class. Do the overriding in such a way that when you print a complex number it should print it in the mathematical format (check Wikipedia). Cool, ha?

7. The icing on the cake! Now test your Complex class in your driver class by envoking the operations you defined and checking if the result is correct and what you expected.

8. **OPTIONAL:** Test your Complex class by using **JUnit** testing in Eclipse. If you don't have, JUnit download it (again Google, I will help you if you're stuck). You can find a nice tutorial here:

   https://www.youtube.com/watch?v=8xEi5Pp0h9Q

   Test all of your methods except from *toString()*. In short you should create a new **Source Folder** in your project called **tests**, and create a new package (use the same name in which you put your Complex class). Then right click on your package and select to create a new *JUnit test case*. Follow the instructions from there. To test your methods use **assert**. For example, **assert(1 == 1)** will succeed and **assert(0 == 1)** will fail.

5 More advanced activity: inheritance is logical

Inheritance is a very powerful concept in object-oriented programming and with this exercise we will remind ourselves of it.

The goal of the activity is to model *digital circuits* in Java using the concepts of *logic gates*. Logical gates are switches based on Boolean algebra. You might have multiple inputs to the logical gates, but only one output (0 or 1, False or True, No or Yes).We will use three logic gates: AND, OR and NOT. AND gate has two input lines (either can be 0 or 1), and the output is 1 only in case when both inputs are 1, in all other cases it's equal to 0. OR gate also has two inputs, and its output is 0 only when both inputs are 0, in all other cases it's equal to 1. NOT gate has only one input, and it's output is the negation of the input. If the input is 0 the output of NOT will be 1. You can see it nicely visualized in Figure 5.

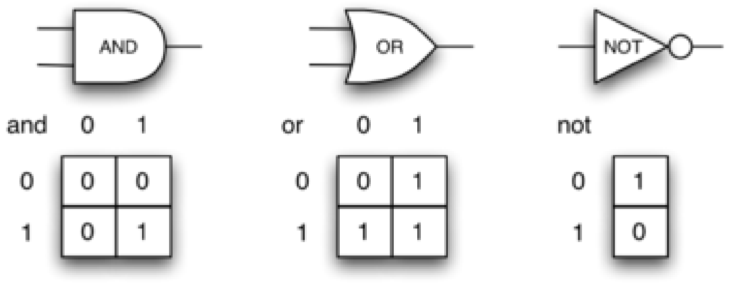


Figure 5: Truth tables for our three logic gates.

Now, having these core components, you can build more complex circuits. You can see one in Figure 6.

Now obviously we can recognize that we have two kinds of logic gates: AND and OR taking two inputs (binary), and NOT taking a single input (unary). From this follows that we obviously have two types of objects that are of similar class (namely, they are all logic gates), but they have different behaviour and properties. When modelling these kind of problems, we use inheritance.

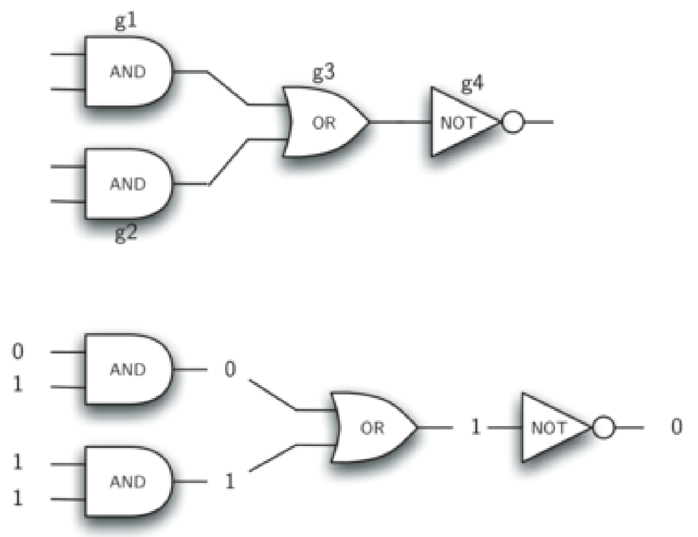


Figure 6: Circuit example.

1. Now create a new project and use the hierarchy from Figure 7 to implement this inheritance hierarchy. (Hint: you will need to put each class in a separate .java file, the keyword for inheritance in Java is **extends** as in *public class Dog extends Animal*).

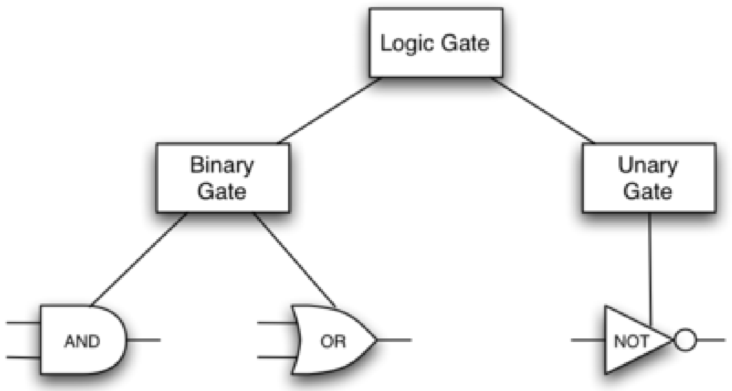


Figure 7: Inheritance hierarchy for logic gates.

2. After that implement behaviour for all the classes (AND, OR and NOT gate). You should write it in such a way that when you test your classes with the code in Listing 1 it should work.

Listing 1: Main class in Java

```java
  public static void main(String[] args){
    AndGate a1 = new AndGate(0, 1);
    assert a1.getValue() == 0 : "awkwaaard";
  }
```