Imperial College London – Department of Computing

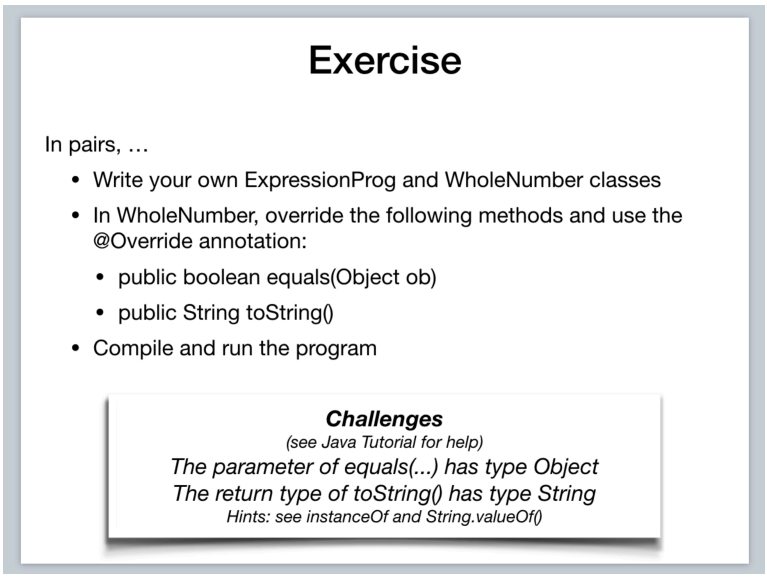
MSc in Computing Science

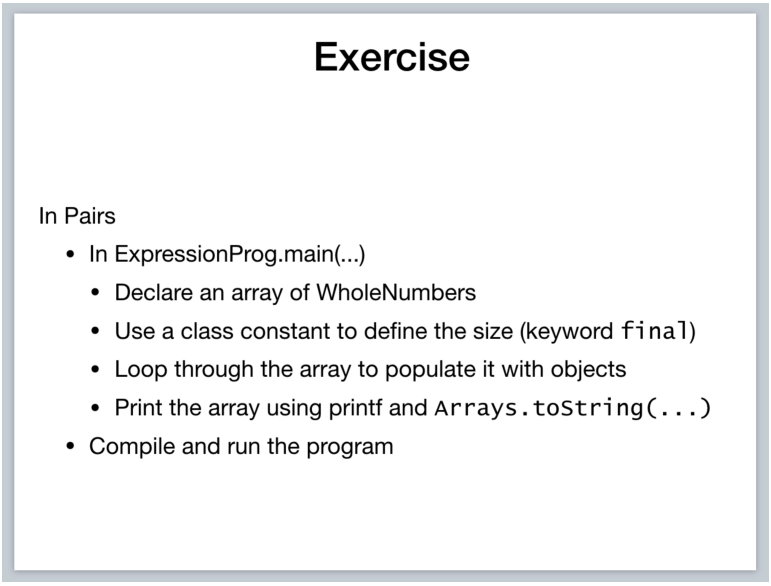
536: Introduction to Java and Concurrency
Tutorial 1 — Arithmetic Expressions

(Updated Wed, 10 January 2018, 14:30)

O. Before We Start

In yesterday's lecture we skipped two exercises that you should do BEFORE starting this tutorial. I am attaching screenshots of the Exercise slides for your convenience.





In this tutorial you will continue to implement Java types representing simple arithmetic expressions. You can either use the code you have written yourself so far, or start from the code provided.

1. Aims

This tutorial will help you understand how to:

- Write simple Java classes
- Implement inheritance in Java
- Use abstract classes in Java
- Implement interfaces in Java
- Make use of classes from the Java API including collection classes and iterators
- Use exceptions in Java

2. Your Development Environment

Something you might want to think about at this point is the sort of development environment you want to use. In the lecture I have been using command line tools and if you like things like vim and make, then feel free to stick to that. For Java programming, however, an integrated development environment (IDE) is a very popular alternative. An IDE is a sophisticated graphical tool that is designed to help:

- Organise your code, including via version control
- Compile and run your code
- Debug and correct your code
- Refactor your code
- Build and test your application

Three of the most popular IDEs are Idea (IntelliJ), Eclipse and NetBeans. These are all available on the lab machines (under Programming on the Applications menu). For what it's worth, I use the Eclipse IDE, certainly for bigger projects. Although, I am told that the most recent versions of IntelliJ are thought to be better.

In the long run you will definitely be using an IDE. Note that these tools can hide away some of the complexities regarding classpaths, package and directory structures. This is great news when the IDE gets it right. When it does not and you have to manually configure them, you need to understand how packages, directories and classpaths work.

My recommendation is that if you are you want to try using an IDE then, once you have finished the tutorial, try to compile the code again from the command line. Just to make sure you know what is going on. If you are going the IDE route, look for a tutorial online. There will be an initial learning period, but once you get the hang of it you will speed up no end.

3. Obtain Provided Files

Download the provided code for this tutorial from CATE into a suitable directory and unzip the file.

4. What To Do

4.1 Implement Remaining Expressions

Your first task is to implement two other types of expression: a sum (addition) and a product (multiplication). These should be implemented as the classes `Sum` and `Product` as follows:

- Both classes should be in the same package as `WholeNumber`
- Both classes should have a constructor that accepts two `Expressions`. In each case, the first argument is the left operand and the second is the right operand.
- Both classes should implement the `Expression` interface.
- Both classes should override the `toString()` method of the superclass `Object`.

4.2 Refactor Expression Types

The two classes you have just written share various properties:

- • They have the same member variables
- • Their constructors are identical

So, you can extract this part of the classes, the part that defines their state, into a common parent class `BinaryExpression`. `BinaryExpression` should declare that it implements `Expression`, so that all `BinaryExpressions` are automatically `Expressions`. This might seem a bad choice, because you cannot define a single way to evaluate all binary expressions. To find out how to solve this problem, and how to implement subclasses in Java you will need to go to the Java Tutorial and doing a bit of reading.

The main section you need is on inheritance, under “Interfaces and Inheritance”:

`https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html`

There are several subsections under that page (see the navigation on the left). The important ones to look at now are “Using the Keyword super”, so you can see how to call the constructor of a superclass from a subclass, and “Abstract Methods And Classes”. Make sure you read the very last section about abstract classes, when an abstract class implements an interface. Finally, you will need to think about the visibility of the fields of `BinaryExpression`. This is covered in the section on “Controlling Access to Members of a Class”:

`https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html`

You should limit the visibility of the fields as much as possible while still allowing the access you need.

4.3 Sorting Expressions

You have seen how to define and implement your own interface type in Java. The Java API also contains “off the shelf” interfaces that your classes can implement. Doing this bestows on the class some useful ability. One such interface is `java.lang.Comparable`. You can read its documentation online.

In a nutshell, objects of any class implementing Comparable can be sorted using `Arrays.sort(...)` and other similar methods.

Your next task is to implement `Comparable` for all `Expression` types, so that they are sortable. Firstly, make `Expression` a subtype of `Comparable`. Interfaces can inherit from one another like classes, using the same notation. So, change the declaration of `Expression` to be:

- `public interface Expression extends Comparable<Expression>…`

The `<Expression>` is necessary because `Comparable` is generic. If we have not talked about it in the lectures, don't worry about this now. For now you need to decide how you are going to implement `Comparable<Expression>`. The interface has a single method:

- `public int compareTo(Expression other)`

which must be defined for all Expressions. Decide in which class or classes this method should be defined. A call `this.compareTo(that)` returns:

- a positive int if this is "greater than" that
- a negative int if this is "less than" that
- zero if this is "equal to" that

You should use the value of the expression (the integer value obtained by evaluating the expression) to determine the order. Test your implementation by running `ExpressionProg`.

4.4 Managing an Unbounded Number of Expressions

The program `ExpressionProg` creates a fixed array of expressions to work on. Sometimes the number of elements to process is unknown a priori. In these cases linked lists can be useful. Rather than implementing a linked list from scratch, we will use a class from Java class library. For this task you will need to do the following:

- Write a method in `ExpressionProg` that builds and returns a random `Expression`. The method has one parameter that bounds the maximum depth (of nested expressions) of the expression to be built. For instance the expression `((1+2)+3)` has depth `3`. This is easy to see if the expression is represented as a tree. You can use `java.util.Random` and its method `nextInt()` to support random number generation.

  ```
  static Expression randomExpression(int maxDepth)
  ```

- Change the main program in `ExpressionProg` so that instead of a fixed number of expressions are stored in an array, a random number of expressions are stored in a `LinkedList`. Use the following snippet of code:

  ```
  List<Expression> nums = new
        do {
              nums.add(randomExpression(4));
        } while (Math.random() < 0.95);
  ```

  Use a `ListIterator` to apply the following statement to all expressions:

  ```
  System.out.println(e + " = " + e.evaluate());
  ```

- Add to the end of the main program in `ExpressionProg` code that classifies the expressions of the list based on the last digit (e.g., all expressions that evaluate to numbers ending in "9" should go together). To produce the classification use an implementation of the Java Map interface. The keys of the map shall be Integer objects representing digits 0 to 9. The values of the map corresponding to key "9" shall

be the expressions that satisfy `(e.evaluate() % 10) == 9`
Print out the contents of the map once the expressions have been classified. Use `for (Integer key: m.keySet())` to range over the different keys.

As an example, if there were only 3 expressions: `5+2`, `(10*3)+7`, and `4`, the output would be something like:

```
Expressions with value ending in 4:
4
Expressions with value ending in 7:
5+2
(10*3)+7
```

4.5 Handling Exceptions

You are now going to add create a new class called division that works similarly to sum and product in that it is built from two expressions. However, in this case the constructor must check if the right hand side operand is a divisor of the left hand side. It must also check that there will be no division by 0. If conditions are not met, an exception `WholeDivisionUndefined` must be raised. For this task you must:

- Write class `WholeDivisionUndefined` that extends Exception
- Write class `Division`
- Extend your function `randomExpression` to catch these kinds of exceptions and resolve them in some way you find suitable.