CSC207
http://www.teach.cs.toronto.edu/~csc207h/fall/index.shtml
Instructors: Paul Gries, Lindsey Shorser
Fall 2017

An introduction to software design and development concepts, methods, and tools using a statically-typed object-oriented programming language such as Java. Topics from: version control, unit testing, refactoring, object-oriented design and development, design patterns, advanced IDE usage, regular expressions, and reflection. Representation of floating-point numbers and introduction to numerical computation.

You …

… know the CSC108, CSC148, and (usually) the CSC165 material: lists, dictionaries, functions, classes, stacks, queues, trees, recursion, unit testing, logical notation and reasoning, the basics of computational complexity (big-Oh), and an approach to developing functions (the function design recipe).

… presumably want to do very well in this course.

… should expect to spend 8–10 hours a week on each of your courses (including lectures and labs).

… might think about this course as training for a software internship.

8–10 hours before the end of the day next Wednesday

• Attend lecture (2 hours)
• Attend lab next week (1 hour)
• Log into the Teaching Labs and run IntelliJ IDEA (1/2 hour)
• Install Git, Java, and IntelliJ on your own computer (1 hour)
• Work through Quest 1 on the PCRS and practice in IntelliJ (4 hours)
• Lab next week will involve Java code, so you should try to get through as much as you can before next Tuesday

Learning goal: Object-oriented programming in a statically-typed language

Strong typing
Lots of inheritance
Unit testing
A memory model for Java
File handling
Exception handling
Floating-point issues
Learning goal:
Fundamental code development techniques used professionally

How to think about and plan a large program
How to analyze requirements
How to safely refactor code
Design patterns
Version control (using git)
Aspects of team dynamics
An Integrated Development Environment (IntelliJ)

How we’re going to teach

2 lecture hours / week
8 (ish) 1-hour labs
1 45-minute midterm in week 5
2 individual assignments
A two-phase project in the second half of the course
1 final exam

Resources

Course website (readings, links)
Discussion board
Office hours
Lectures and labs!
Anonymous feedback (if you don’t want to email us or post on the boards): please give us constructive suggestions!

Java Reference Materials

Course website (readings, lecture notes, links)
This reference is particularly useful:
http://docs.oracle.com/javase/tutorial/java/TOC.html
Java PCRS
This website does a nice job walking you through Java if the PCRS isn’t enough:
https://www.sololearn.com/Course/Java/
Email registration is required. (Have you heard about disposable email addresses? Here’s a top-15 article about the topic.)
What does it mean to run a program?

What is a program?

A set of instructions for a computer to follow.

To run a program, it must be translated from a high-level programming language to a low-level machine language whose instructions can be executed.

Roughly, two flavours of translation:

- Interpretation
- Compilation

Interpreted vs. Compiled

Interpreted (like Python)

Translate and execute one statement at a time

Compiled (like C)

Compile the entire program (once), then execute (any number of times)

Hybrid (like Java)

Compile to something intermediate (in Java, bytecode)

The Java Virtual Machine (JVM) runs this intermediate code

Compiling Java

If using command line, you need to do this manually.

First, compile using “javac”:

```
pgries@laptop$ javac HelloWorld.java
```

This produces file “HelloWorld.class”:

```
pgries@laptop$ ls
HelloWorld.class  HelloWorld.java
```

Now, run the program using “java”:

```
pgries@laptop$ java HelloWorld
Hello world!
```

Most modern IDEs offer to do this for you (IntelliJ does).

But you should know what's happening under the hood!

SOLID: five basic principles of object-oriented design

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle
Single Responsibility Principle

Every class should have a single responsibility

Responsibility should be entirely encapsulated by the class

All class services should be aligned with that responsibility

Why?

- makes the class more robust
- makes the class more reusable

Open/Closed Principle (simplified)

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Add new features not by modifying the original class, but rather by extending it and adding new behaviours.

The derived class may or may not have the same interface as the original class.

Example:

area calculates the area of all Rectangles in the input.

What if we need to add more shapes?

<table>
<thead>
<tr>
<th>Rectangle</th>
<th>AreaCalculator</th>
</tr>
</thead>
<tbody>
<tr>
<td>- width: double</td>
<td>+ area(shapes: Rectangle []): double</td>
</tr>
<tr>
<td>- height: double</td>
<td>+ getHeight(): double</td>
</tr>
<tr>
<td>+ getWidth(): double</td>
<td></td>
</tr>
<tr>
<td>+ setWidth(w: double): void</td>
<td></td>
</tr>
<tr>
<td>+ getHeight(): double</td>
<td></td>
</tr>
<tr>
<td>+ setHeight(h: double): void</td>
<td></td>
</tr>
</tbody>
</table>

What if we need to add even more shapes?
Open/Closed Principle (simplified)

With this design, we can add any number of shapes (open for extension) and we don't need to re-write the AreaCalculator class (closed for modification).

Liskov Substitution Principle (simplified)

If S is a subtype of T, then objects of type S may be substituted for objects of type T, without altering any of the desired properties of the program.

“S is a subtype of T”?

In Java, S is a child class of T, or S implements interface T.

For example, if C is a child class of P, then we should be able to substitute C for P in our code without breaking it.

Liskov Substitution Principle (simplified)

A classic example of breaking this principle:

In OO programming and design, unlike in math, it is not the case that a Square is a Rectangle!

This is because a Rectangle has more behaviours than a Square, not less.

The LSP is related to the Open/Closed principle: the subclasses should only extend (add behaviours), not modify or remove them.
Interface Segregation Principle

No client should be forced to depend on methods it doesn't use.

Better to have lots of small, specific interfaces than fewer larger ones.

Easier to extend and modify the design.

Dependency inversion principle

When building a complex system, we may be tempted to define the "low-level" classes first and then build the "higher-level" classes that use the low-level classes directly.

But this approach is not flexible! What if we need to replace a low-level class? The logic in the high-level class will need to be replaced.

To avoid such problems, we can introduce an abstraction layer between low-level classes and high-level classes.

Dependency inversion principle

To make Manager work with SuperWorker, we would need to rewrite the code in Manager.

Manager does not know anything about Worker, nor about SuperWorker. It can work with any IWorker, the code in Manager does not need rewriting.
Dependency inversion principle

Two aspects:

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend upon details. Details should depend upon abstractions.

When building a complex system, we may be tempted to define the "low-level" classes first and then build the "higher-level" classes that use the low-level classes directly.

But this approach is not flexible! What if we need to replace a low-level class? The logic in the high-level class will need to be replaced.

To avoid such problems, we can introduce an abstraction layer between low-level classes and high-level classes.