CSC207

http://www.teach.cs.toronto.edu/~csc207h/summer/

Instructors: Lindsey Shorser, David Jorjani
Email: csc20718y@cs.toronto.edu
Summer 2018

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.

— The Arts and Science Calendar

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.

Learning goal:

Object-oriented programming in a statically-typed language

- Strong typing
- Lots of inheritance
- Unit testing
- File handling
- A memory model for Java
- 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
- 10 (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**

Coursework Overview

Work	Weight	Comment
Labs (8)	8%	1% each, best 8 out of 10
A1	5%	Individual
A2	10%	Individual
Project: Phase 1	10%	Group: team of four from the same lecture section
Project: Phase 2	17%	
Test	10%	during lab time (bring your TCard)
Final Exam	40%	You must get ≥ 40% on the final exam to pass CSC108!

Resources

- **Portal/Blackboard**: Announcements, Grades
- **Course website**: Readings, Links
  - www.teach.cs.toronto.edu/~csc207h/summer/
- **Discussion board**
- **Office hours**
  - BA3201: Tuesdays 5:30 pm -6:30 pm
- **Lectures** and **labs**!
- **Anonymous feedback** (if you don't want to email us or post on the boards): please give us constructive suggestions!
- **Help Centre** in BA2230 every weekday 4-6 pm, except holidays
- **PCRS and On-Line Tutorials** (see course website)

General Resources

- Lectures and Labs
- Office Hours
- Discussion Forum
- Help Centre
- On-Line Communities?
- PCRS and on-line tutorials
- Oracle Website

Java Reference Materials

- Course website (readings, lecture notes, links)
- This reference is particularly useful:
  - http://docs.oracle.com/javase/tutorial/java/TOC.htm
    l
- 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? https://en.wikipedia.org/wiki/Disposable_email_address
      - Here's a top-15 article about the topic: www.updateland.com/15-best-fake-email-address-generator-online/

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

- 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 Thursday

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”:

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

This produces file “HelloWord.class”:

```
jsin@laptop$ ls

HelloWorld.class  HelloWorld.java
```

Now, run the program using “java”:

```
jsin@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 Principles of Object-Oriented Design

- How do we make decisions about what is better and what is worse design?
- Principles to aim for instead of rules.
    - For example, there is no maximum number of class you should have in your program, nor a minimum. But if the number of classes violates a generally accepted principle, you should reconsider your class structure.
- The SOLID principles are useful and cover most major situations you are likely to encounter.

Software design goals

- A major goal when programming is to write an easy-to-read, hard-to-break, maintainable, efficient program.
- Software design has you use a set of principles and techniques that help you do this. This lecture is an introduction to the tools and techniques we'll see in this course.
- We'll cover each of these in more detail later in the course.

Fundamental OOP concepts

- **Abstraction** — the process of distilling a concept to a set of essential characteristics.
- **Encapsulation** — the process of binding together data with methods that manipulate that data, and hiding the internal representation.
- The result of applying abstraction and encapsulation is (often) a class with instance variables and methods that together model a concept from the real world. (Further reading: what's the difference between Abstraction, Encapsulation, and Information hiding?)
- **Inheritance** — the concept that when a subclass is defined in terms another class, the features of that other class are inherited by the subclass.
- **Polymorphism** ("many forms") — the ability of an expression (such as a method call) to be applied to objects of different types.

Fundamental OOD goals: low coupling, high cohesion

- **Coupling** — how much a class is directly linked to another class.
- High coupling means that changes to one class may lead to changes in several other classes.
- Low coupling is, therefore a desired goal.
- **Cohesion** — how much the features of a class belong together.
- Low cohesion means that methods in a class operate on unrelated tasks. This means the class does jobs that are unrelated.
- High cohesion means that the methods have strongly-related functionality.

Fundamental OOD principles

SOLID: five basic principles of object-oriented (Developed by Robert C. Martin, affectionately known as “Uncle Bob”.)

- 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.

Another way to view this is that a class should only have one reason to change.

But who causes the change?

"*This principle is about people.* … When you write a software module, you want to make sure that when changes are requested, those changes can only originate from a single person, or rather, a single tightly coupled group of people representing a single narrowly defined business function. You want to isolate your modules from the complexities of the organization as a whole, and design your systems such that each module is responsible (responds to) the needs of just that one business function." [Uncle Bob, The Single Responsibility Principle]

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, or by adding *plugin* capabilities.

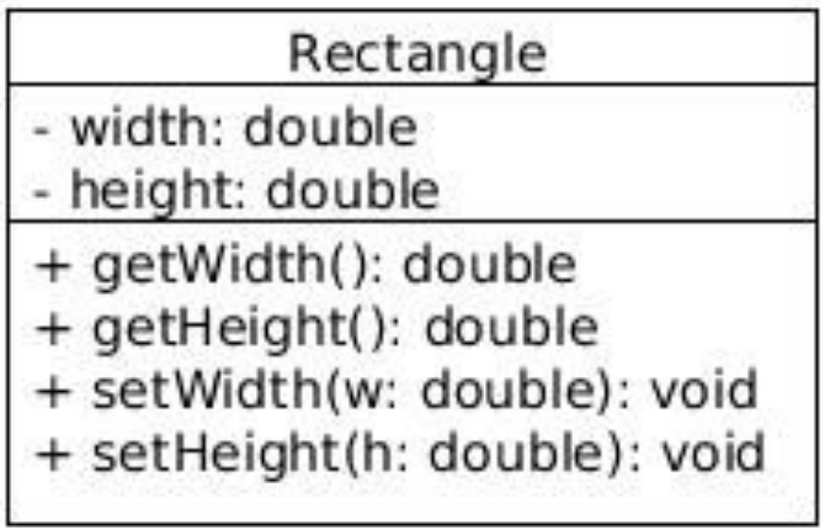
“I’ve heard it said that the OCP is wrong, unworkable, impractical, and not for real programmers with real work to do. The rise of plugin architectures makes it plain that these views are utter nonsense. On the contrary, a strong plugin architecture is likely to be the most important aspect of future software systems.” [Uncle Bob, The Open Closed Principle]

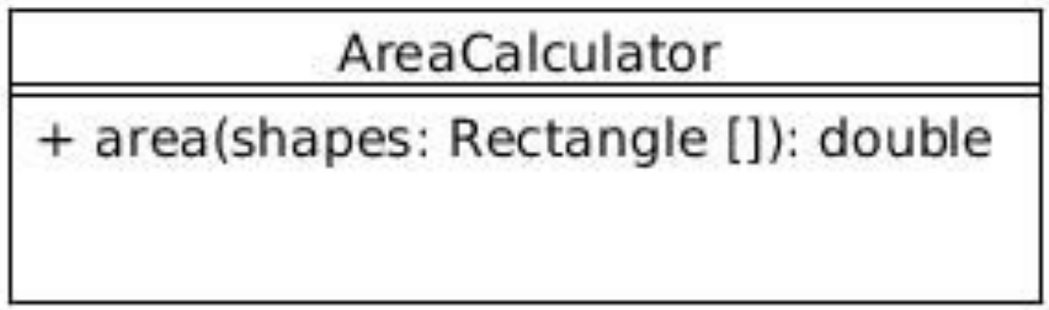
Open/Closed Principle (simplified)

An example, using inheritance:

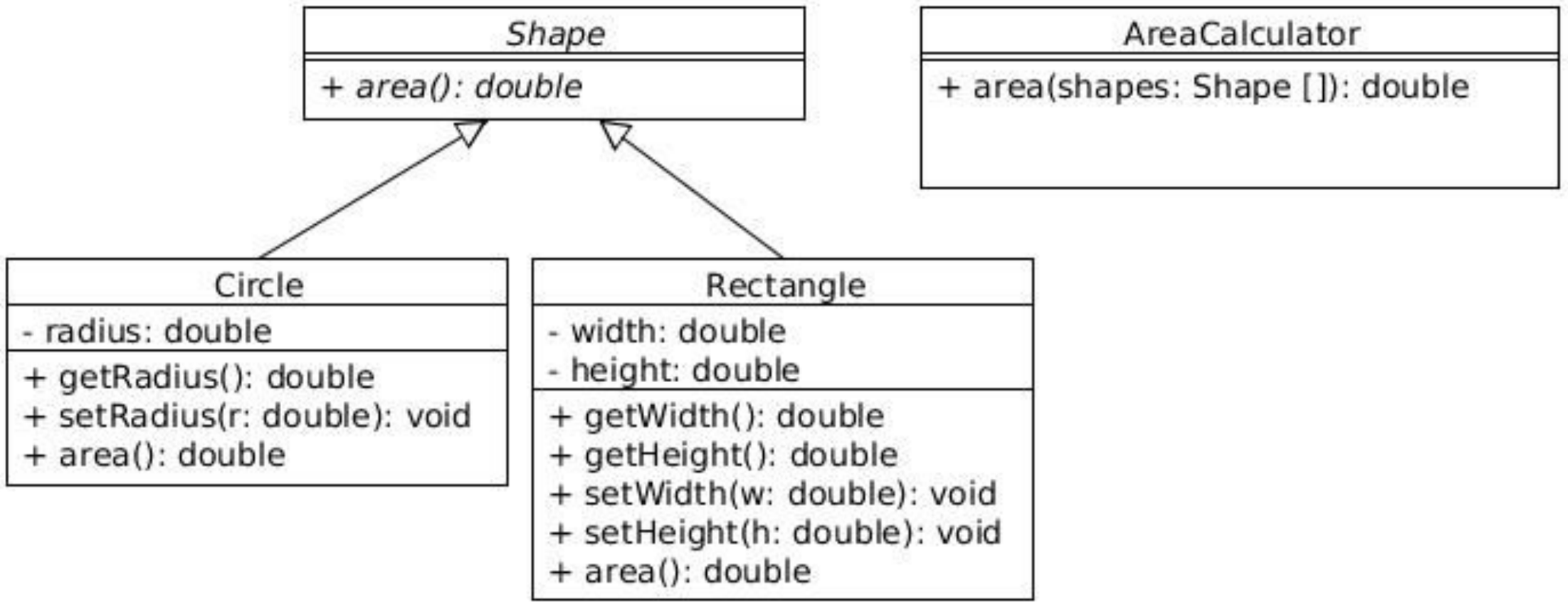
`area` calculates the area of all `Rectangle`s in the input.

What if we need to add 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.

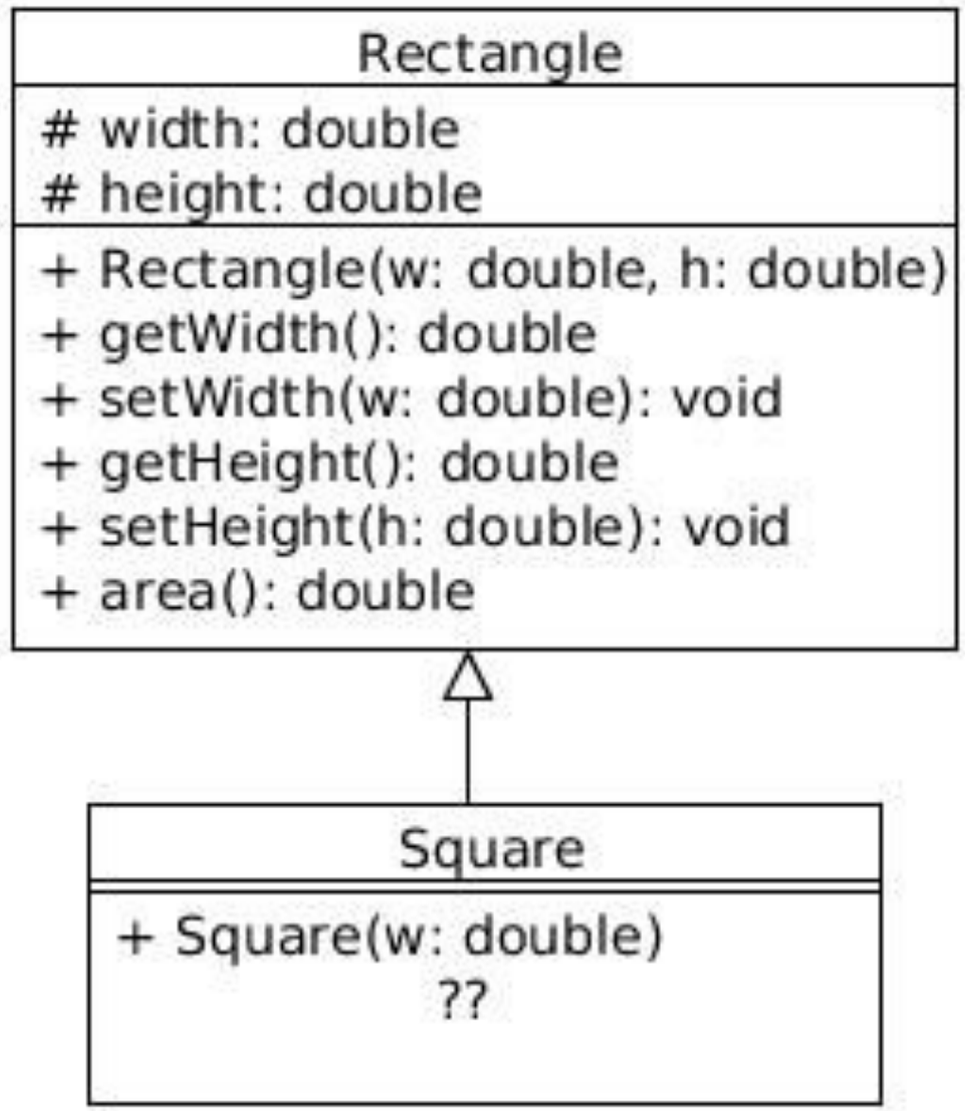
“S is a subtype of T”?

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:



Liskov Substitution Principle (simplified)

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

Here, *interface* means the public methods in a class. (In Java, these are often specified using a Java `interface`, which you'll learn about soon.)

Context: a class that provides a service for other "client" programmers usually requires that the clients write code that has a particular set of features. The service provider says "your code needs to have this interface".

No client should be forced to implement irrelevant methods of an interface. Better to have lots of small, specific interfaces than fewer larger ones: easier to extend and modify the design.

(Uh oh: "The interface keyword is harmful." [Uncle Bob, 'Interface' Considered Harmful])

Dependency inversion principle

When building a complex system, programmers are often tempted to define “low-level” classes first and then build “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 — an indication of high coupling.

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

Dependency inversion principle

Goal:

You want to decouple your system so that you can change individual pieces without having to change anything more than the individual piece.

Two aspects to the dependency inversion principle:

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.

Dependency inversion principle (example from Dependency Inversion Principle on OODesign)

Example: you have a large system, and part of it has Managers manage Workers. Let's say that the company is restructuring and introducing new kinds of workers, and wants the code updated to reflect this.

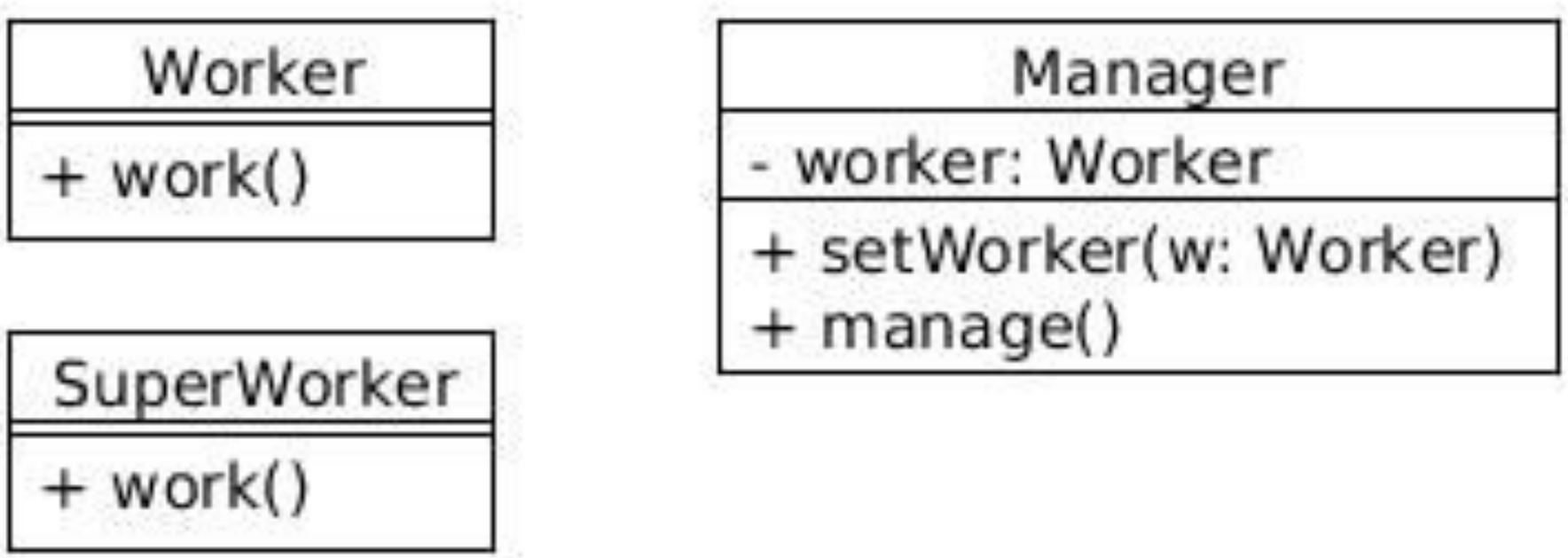
Your code current has a Manager class and a Worker class, and the Manager class has several methods that have Worker parameters.

Now there's a new kind of worker called SuperWorker, and their behaviour and features are separate from regular Workers.

Oh dear …

Dependency inversion principle (example from Dependency Inversion Principle on OODesign)

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



Solution: create an IWorker interface and have Manager use it.

Dependency inversion principle (example from Dependency Inversion Principle on OODesign)

In this design, Manager does not know anything about Worker, nor about SuperWorker. It can work with any IWorker, the code in Manager does not need rewriting.