# Class Techniques - Part One

## Overview

This lab focuses on the various class techniques discussed in the chapter - nullable types, exceptions, and generics.

## Roadmap

There are 4 exercises in this lab, of which the last exercise is "if time permits". Here's a brief summary of the tasks you'll perform in each exercise; more detailed instructions follow later in this lab doc:

1. Define a class with nullable properties
2. Throw and catch exceptions
3. Define and use a generic class
4. (If Time Permits) Additional suggestions

## Exercise 1: Define a class with nullable properties

Define a class named Employee with a couple of properties in the primary constructor:

* The name of the employee (fixed)
* The employee's salary (can change)

Inside the class, define another property named manager; define it as a nullable Employee reference, indicating the employee's manager. Set this property to null initially, to indicate the employee doesn't have a manager yet.

Also inside the class, define a computed property named hasManager. Define a getter that returns true or false, indicating whether the employee currently has a manager (i.e. is the manager property null or not).

Finally inside the class, override toString(). Return a string containing the employee's name, salary, and whether he/she currently has a manager.

Now write some client code as follows:

* Create some Employee instances and display their details on the console.
* For one employee, print the name of his/her manager (you'll need to use the ?. operator). This should display null, because the employee doesn't have a manager yet.
* For the same employee, set the manager property to one of the other employees.
* For the same employee, print the name of his/her manager again. This should now display the manager's name.

## Exercise 2: Throw and catch exceptions

In this exercise, you'll tighten up the rules for assigning a manager to an employee. Specifically, if an employee already has a manager, you can't set the manager again. In this case, you'll throw an exception to indicate the employee already has a manager.

The first step is to define a custom exception class to represent "manager-related errors". As in Java, exception classes in Kotlin should follow these rules:

* Inherit from Exception
* Take a string parameter in the constructor, and pass it to the superclass constructor

This is how to do this in Kotlin (we'll discuss the details about inheritance later in the course):

class ManagerException(msg: String) : Exception(msg)

The next step is to define a custom setter for the manager property, to tighten up the rules for setting the manager for an employee. Follow these steps:

* If the manager property is null, it means the employee doesn't have a manager at the moment. In this case, assign the manager to the employee now.
* Otherwise, it means the employee already has a manager, so it's not allowed to assign another manager. In this case, throw a ManagerException.

The final step is to tweak your client code to try out this new functionality. Try assigning a manager to an employee who already has a manager; this should cause an exception, so you'll need to wrap the code in a try/catch block.

## Exercise 3: Define and use a generic class

Applications deal with the concept of IDs a lot. Some IDs are numbers, some are strings, some are UUIDs, and so on. In this exercise you'll define a generic ID<T> class to represent any type of ID. You could then use it in scenarios such as the following:

* In a relational database, each table has a primary key. The primary key is typically just a number. You could create an ID<Long> to represent a primary key.
* In a network, each server has a unique IP address. You could create an ID<String> to represent a server IP address.
* In a company, each employee has an employee ID (e.g. a unique random identifier). You could create an ID<UUID> to represent an employee ID.

So define an ID<T> class as follows:

* The primary constructor should have a T property, i.e. the actual "value" of the ID.
* Define another property named timestamp and set it to the current date/time.
* Override toString() to return a string containing the timestamp and the ID value.

Now enhance your Employee class so that an employee has an ID<UUID>. Tweak the toString() method to display the ID. Also tweak your client code, so that whenever you create an Employee instance, you pass in a suitable ID<UUID> value.

## Exercise 4 (If Time Permits): Additional suggestions

Take another look at the miscellaneous techniques covered at the end of the chapter, and think about how you might use some of these techniques in your code.