COMP0004 Object-Oriented Programming
Programming Exercises 5

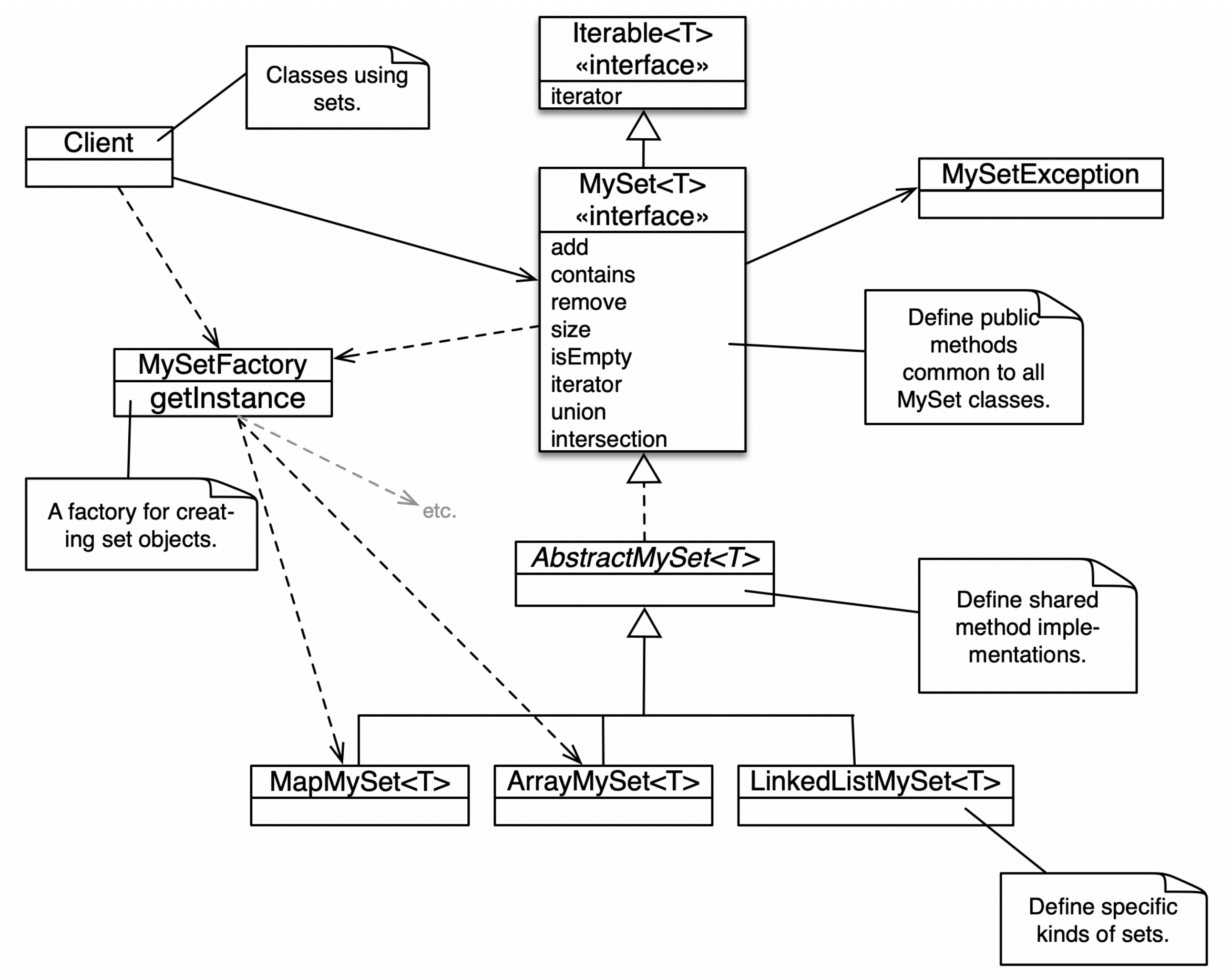
Purpose: A more complex Java programming task to help you understand interfaces, abstract classes, generics, nested classes, inheritance, overriding methods and using exceptions. There are a number of important concepts covered here, make sure you understand them all.

Complete as many of the exercise questions as you can. If you are keeping up, you need to do at least the core questions. The challenge questions are quite demanding and are designed to stretch the more confident programmers. If you can't do them now, be prepared to come back and try them later on. You should aim to at least complete the core questions and get them reviewed in a lab session.

Feedback: It is important that you get feedback on your exercise answers so that you know they are correct, that you are not making common mistakes, that the program code is properly presented and that you are confident you have solved the problem properly. To do this, get your answers reviewed by a TA during a lab session.

Writing a Set Class

A set data structure represents a mathematical set, which stores a collection of values with no duplicates. This coursework is about implementing set classes, making use of interfaces, generics, an abstract class, inheritance, overridden methods and exceptions. The Java language features needed to implement these classes were covered in the lectures and other material on Moodle, but it would be a good idea to read up on them first.



The goal is to set up a framework that will support several classes providing different concrete set implementations. The framework structure is shown above.

This framework is based on the principles used by the data structure classes in the standard Java libraries but is intended to illustrate a range of programming concepts rather than be a fully practical design for a real framework.

The `Iterable` interface is defined in the standard Java class library and declares the `iterator` method, so that each value stored in a data structure can be accessed, or iterated, in sequence. Implementing the `Iterable` interface also allows the values stored in a data-structure object to be accessed in sequence using the enhanced for loop (foreach).

The `MySet` interface defines the common methods for all set classes. It is named `MySet` to avoid confusion with the `Set` class in the Java class Library, which you do not need to use at all. `MySet` extends the `Iterable` interface, remembering that one interface can extend another. A variable of type `MySet` can reference any set object of a class that implements `MySet` (or inherits the implements relationship). This enables 'Programming to an Interface', allowing code to be written using the `MySet` type without needing to be concerned about the exact type of set of object it is working with at runtime.

For these exercises you are given the code of a basic version of the `MySet` framework, including one example set implementation called `ArrayMySet`. You will find this code in a GitHub repository at:

https://github.com/UCLComputerScience/COMP0004MySet

Clone this repo into your working directory on your local machine and then load it into IDEA by creating a new project, or create a new project directly by cloning. Note that you cannot push to the repo, but you can connect it to your own GitHub repo if you create one and push to that.

The `MySet` code uses generics so that the classes and interfaces are parameterised over the type of value held in a set. Hence, when declaring a variable of type `MySet` you would have:

```
MySet<Integer> set; // Reference to a set of Integers.
```

When you look at the `MySet` interface you will see that it declares the type parameter like this:

```
public interface MySet<T extends Comparable<T>> extends Iterable<T>
```

The `<T extends Comparable<T>>` means that the type `T` is constrained to be `Comparable` or a subtype of `Comparable`. `Comparable` is an interface that defines the `compareTo` method to compare the values of two objects (see the Javadoc for a full explanation). This constraint is necessary as the values held in a set must be comparable to ensure that there are no duplicates in the set. A set can only hold values of types that implement the `Comparable` interface, which include `Integer`, `Double`, `String`, or any class you write that implements `Comparable`.

Some `MySet` methods are declared as throwing exceptions of type `MySetException`, and a default maximum size is imposed on all sets of `1000` values. A different maximum size can be specified when a set object is created. If an attempt is made to to create a set with size greater than the maximum size, or less than `1`, an exception should be thrown. In addition, if adding a value to a set would cause it to exceed the maximum size an exception should be thrown.

The `AbstractMySet` class is an abstract class that provides common methods to be inherited by concrete subclasses. This avoids duplicating the methods in the subclasses as they can just be defined once and inherited. In the code you get from GitHub you will see that a number of methods are included but are not complete. You will need to add the missing code. For example, the `union` method is intended to find the set union between the set the method is called on and the set passed as a parameter. A new set containing the union is returned, with the new set object created using a *Factory*. A factory is a class that specialises in creating objects. The use of a factory avoids code having to name a specific concrete set class and decouples object creation from class naming. Factory is an example of a Design Pattern.

Class `MySetFactory` implements a basic factory. When a program using sets is started, the `MySetFactory` is given the name of the set class (a String) that it will create objects of. The `MySetFactory` also implements the Singleton Pattern, meaning that only one instance object of the class can be created when the program is run (see the example code in the Main class included with the framework code).

A complete example set implementation class, `ArrayMySet`, is provided that uses an `ArrayList` as the private internal data structure used to store the values in the set. A private nested (member) class is used to implement the `Iterator` interface to enable iteration of the set object. Don't confuse `Iterable` with `Iterator`. `Iterable` defines the `iterator` method, while `Iterator` defines the `next` and `hasNext` methods.

Note that class `ArrayMySet` throws exceptions *but does not catch any of its own exceptions*. A set throws exceptions, while code using a set catches any exceptions using try/catch blocks. Make sure you understand the point being made here.

Core Questions

The initial code can be obtained from the GitHub repo. This code will compile and run but won't do anything useful until the missing method code is implemented. There are no guarantees that the code does not contain errors - check carefully!

Do not change any of the public method names in the MySet interface or MySetFactory, or any of the parameter or return types. Also use the class names specified in the questions.

The GitHub content includes a `Main` class that is intended to contain code that uses the classes and code that you write, to demonstrate that the code works correctly. As you answer the questions below you should extend the `Main` class with more methods to confirm the answers work. It will be important to keep the code in the `Main` class well-organised and properly written, don't let it become a mess. If you feel that your `Main` class is getting too large and unfocussed, then split it into two or more smaller classes. For example, the code to confirm that Q5 works could be put into another class.

Q1. Complete the missing method bodies in class `AbstractMySet`, so that the code will compile and you can create and use `ArrayMySet` objects. The `union`, `intersection` and `difference` methods must all be implemented in the abstract class and inherited by the concrete subclasses. You should not implement the methods in the subclasses, or copy and paste the code into other classes.

Q2. Add a method with the signature `String toString()` to the `AbstractMySet` class, that returns a string representation of the contents of a set in this style:

```
{ value1, value2, ... }
```

Remember that `toString()` is a method declared in class `Object` (the superclass of all other classes), and can be overridden by any of your classes to specialise the way it represents an object's value as a string. No code in any concrete subclasses should be changed, they can inherit the method you add to `AbstractMySet`. The example code in the `Main` class will be helpful for this question.

Q3. Implement a concrete class `LinkedListMySet` that uses a chain of linked list elements as its data structure. Do *not* use any linked list classes from the standard Java class libraries, you should implement the list code yourself. A linked list element class might look like this:

```
private static class Element<E>
{
  public E value;
  public Element<E> next;
```

```
    public Element(E value, Element<E> next)
    {
      this.value = value;
      this.next = next;
    }
  }
```

This should be a nested class inside the `LinkedListMySet` class. As it is a private class providing infrastructure for the `LinkedListMySet` class the instance variables are public (think of a C struct). You might want to make them private and provide setters and getters.

Q4. Implement a concrete class `MapMySet` that uses a `HashMap` (dictionary) as the internal data structure. You can use `Map` and `HashMap` from the Java class library to do this.

Q5. (Harder) Add the ability to save the contents of a set to a file using a text representation, and to read the file and re-create the set object. Get this working for common types such as `Integer`, `String`, `Double`, etc. Getting file load/save working for any type you can put in a set is more complex - see if you can get something working.

Challenge Questions

Q6. Add a powerSet method to generate the Power Set (set of subsets) of a set. The use of generics makes this a bit tricky to solve.
Hint, can you have a `MySet` of `MySets`, such as
`MySet<MySet<T extends Comparable<T>>>`
or will you have to be more creative to get this working?
Don't forget about searching online tutorials and resources for additional sources of information about generics.

Q7. Modify your code to implement the `Comparator` interface for comparing values, rather than `Comparable`, allowing lambdas to be used to specify how comparison is done for the values you store in a set. Can you get this working for examples such as MySet<ArrayList<Integer>> or MySet<MySet<String>>?

You will need to read up about the `Comparator` interface and why it is useful. The major advantage of Comparator is that you can compare values of any type and are no longer limited to subclasses of Comparable.