Lab 8. Maps and Sets in JCF

Theme. In this lab, you will:

- use Map and Set in JCF to model a software problem
- explore the difference between different implementations of Map and Set in JCF
- define the natural order for a class of object
- use Map as a means to index records

Key concepts: maps, sets, JCF, `Comparable`, hash codes

Required file(s): `lab8.zip`

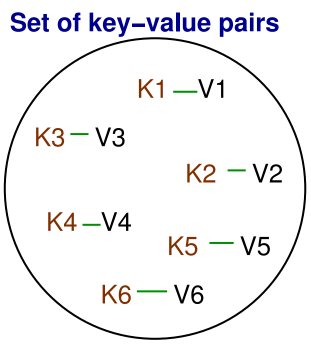
1 Getting started...

1. In a web browser, download the archive `lab8.zip` from Blackboard and extract its contents into your default `Eclipse` workspace for this module.
2. Start up `Eclipse`.
3. Making use of the contents of your extracted archive, create a new Java project named **phonebook_maps_and_sets** in your `Eclipse` workspace using the contents of your extracted archive.

2 Generic Phone Book

Suppose we would like to develop an application for modelling a phone book. Different clients may have different data requirements for a record within a phone book, e.g. some may wish to include the URL of the organisation, others may or may not wish to include addresses. One thing we are certain of is that each record in a phone book must contain one or more phone numbers. Using generics, we can define a collection type that would work like a generic phone book whose record type may vary depending on the client's requirements.

To facilitate record retrieval, we keep the records in a map. A map is an Abstract Data Type (ADT) which *associates keys to values*. Each key is mapped to *exactly* one value, e.g.:



Name (key)	Phone (value)
Tony Beaumont	0121 2043447
Hai Wang	0121 2043457
Nick Powell	0121 2045101
Sylvia Wong	0121 2043473

We can use the name of each contact as the key, with a phone number mapped to it. Such an implementation would work only if we assume that each person in the phone book has only **one** phone number. If a person were to have two or more phone numbers (e.g. work phone, home phone, mobile phone, etc.), the application would fail to handle the situation. Furthermore, such an implementation also would not allow us to include address or other contact information as part of the value. One way to resolve this problem is to create a suitable object to model one contact/correspondence record in the phone book. Each record may comprise information such as nickname, name, a set of phone numbers and an address. To facilitate a mapping between the record and its key for look up purposes, we can make the object aware of its associated key. A key may be the value of a field in the object, but it may also be composed of values in more than one field, e.g. a nickname and a name, (similar to the idea of a *composite key* in database terminology).

The Java Collections Framework (JCF) includes two specifications of **map**: `Map` and `SortedMap`. Values in a `SortedMap` are sorted by their associated **keys**. Using a `SortedMap` to keep records in a phone book will facilitate the listing of all phone book data and make it easier for the user to search for the required information, especially when the data is in printed form.

Suppose a client also needs to find out to whom a phone number belongs. We can define an additional **map**, i.e. one that associates phone numbers to records.

This set of practical exercises will help you gain more practical experience on using generic types, `Map` and `Set` in JCF.

The archive `lab8.zip` contains five classes and one interface:

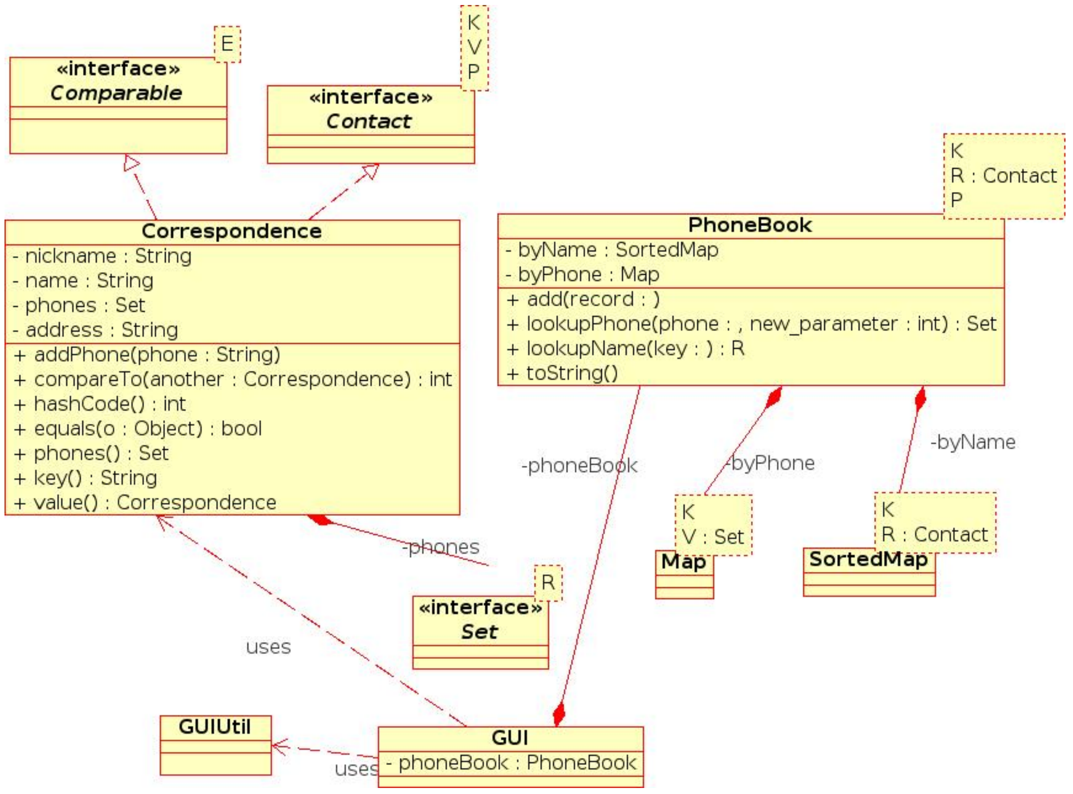


Figure 1: Class Diagram

- `PhoneBook`: a class to model a generic phone book. Every record in the phone book is unique. The uniqueness is defined by the key of that value.
- `Correspondence`: a class to model a contact detail.
- `GUI`: a GUI for handling the input/output (I/O) of a phone book.
- `Contact`: an interface which specifies three methods `key`, `value` and `phones`. A class that implements this interface can be used within the context of a map because every object of such a class knows the key that it is meant to be associated with.

- `GUIUtil`: a class containing three utility methods setting up a suitable font for a GUI. This class contains **static methods**[1] only. No instance of this class is expected to be created.

Hint: The locations where you are expected to pay particular attention on the given Java code have also been annotated. Look out for **block comments** that include a sequence of *four* plus signs, i.e.:

```
/* ++++ ...     ...   */
```

The rough locations where you are expected to add your Java code and relevant hints for accomplishing the tasks have been marked throughout the given Java programs. Look out for **block comments** that include a sequence of *four* exclamation marks, i.e.:

```
/* !!!!  ...     ...   */
```

Your Tasks

1. There exists an **inherent** order amongst all `Correspondence` objects. However, the given class definition has not explicitly made this fact known to the Java Virtual Machine (JVM). Modify the header of class `Correspondence` to remove this error.

Hint: You can define the inherent order of a class of objects by attaching a **comparator** to the class. One way to do this is to make the class implements the interface `java.lang.Comparable`.

2. A `Correspondence` object is expected to be used whenever a `Contact` "object" is required. Modify the header of class `Correspondence` to enable this usage.

3. Complete the implementation for method `key` in class `Correspondence`.

4. Complete the implementation for method `compareTo` in class `Correspondence`.

5. Complete the implementation for method `hashCode` in class `Correspondence`.

6. Complete the implementation for method `add` in class `PhoneBook`.

7. Complete the implementation for method `lookupPhone` in class `PhoneBook`.

Hint: The identifier `P` is a type variable which denotes the data type of a phone number.
In the current implementation, a phone number is modelled by a `String` object. In another implementation, it may be modelled by a `Phone` object which has fields for modelling a phone number and its type, e.g. home phone, work phone, mobile phone, etc. Using the type variable `P` improves the flexibility of this application.

8. Complete the definition of field `phoneBook` in class `GUI` by specifying its data type.

Hint: To which piece of data in this application does *each* of the type parameters refer?

[1]Static methods are also known as **class methods**, as they belong to a class, not an instance of a class. You do not need to create an object of the class in order to invoke its methods. Simply invoke the required class method by referencing the class name, e.g. `Maths.random()`.

3 Testing

Now test your implementation to see if it meets the above requirements.

To test your GUI application, use the correspondence data in the given file `phone.txt`.

Hint:

1. Display the content of the file `phone.txt` using a web browser
2. Copy and paste each phone record in `phone.txt` to the appropriate text fields in the GUI and press the `Add` button.
3. Continue adding more phone records using the GUI.

1. Can you look up a given phone number from the phone book correctly?
2. Can you look up a given nickname and name from the phone book correctly?

Hint: The retrieval of phone number(s) requires *both* the nickname and the name of the correspondence.

3. Are the entries in the phone book displayed in the correct order?

4 Further Challenges

1. Class `PhoneBook` uses both `SortedMap` and `Map`. What are the differences between `SortedMap` and `Map` in terms of:
   (a) the order of elements kept in each map, and
   (b) the overall speed for retrieving an element when the size of the set has grown to > 1000000.
2. Making use of a diagram, briefly explain what is meant by the Abstract Data Type (ADT) **map**.
3. **Array**, **linear linked structure** and **hash table** are three types of data structures that can be used to implement an ADT. Which *one* of these data structures is best-suited for implementing a **bounded map**? Making reference to the characteristics of these data structures and the time efficiency of typical **bounded map** operations, justify your answer.

5 Further Programming Exercises

Implement the following functions for the given phone book application.

N.B.: If you do not wish to incorporate your implemented feature into the given GUI, simply output the result on the console using `System.out.print` statements.

1. Display all contact records

2. Remove a phone number from the phone book

Hint: This function will cause a given phone number to be removed from all records in the phone book. This will lead to changes in both `byName` and `byPhone` maps.

(a) Given a phone number, you will need to locate all correspondence records that are associated with it from `byPhone`.

(b) For each correspondence records in the result set, you will need to remove the given phone number from the `Set` object `phones`.

(c) You will also need to implement a new method in `Correspondence` for removing one phone number from `phones`.

(d) Finally, remove the phone and correspondence records mapping from `byPhone`.

3. Remove a correspondence record from the phone book

Hint: This function will cause a given correspondence to be removed from the phone book. This will lead to changes in both `byName` and `byPhone` maps.
The implementation of this function should be similar to removing a phone number from the phone book.