UNBC Laboratory Assignment
Computer Science 101—Winter 2023

Time Classes

Purpose

To consolidate your understanding of methods, pre-conditions, post-conditions, and packaging in the context of a simple `Time` class; to explore the difference between attributes and representation; and to gain experience with interfaces and inheritance.

Outcomes

After completing this assignment, you should

- be able to code with `package` statements;
- explain the effect of adding a `public String toString()` method to a class;
- explain the effect of adding `implements Comparable<...>` to a class declaration;
- be able to give an example of two classes with the same attributes and behaviours, but differing state representations.

Due Date

The completed lab assignment is due Friday, 2023-01-27 *by the beginning of lecture*.

Hand-In format

In this laboratory assignment, there are multiple places where correctly completing the lab means creating code that does *not* compile. Please be sure to capture the results of each failure, and submit them with the rest of your assignment.

If the failures are compile-time failures, you should be able to capture the failure text from the compiler window in your IDE, or redirect `javac` output to a text (`.txt`) file.

If the failures are run-time failures, you should be able to capture the failure text from the run window in your IDE, or redirect `java` output to a text file.

Put the error outputs in the top-level of the `.jar` or `zip`-file that you submit. You may also create an `answers.txt` file if that helps communicate what you are doing.

There are multiple `packages` that you need to create for this assignment, as described below. Figure 3 on page 5 summarizes them.

Time Class — version 1

Write a simple `Time` class with whose state consists of three private member variables representing the hours, the minutes, and the seconds. Put your `Time` class in a package called `version1`.

It should have the methods specified in Figure 1 on the next page.

⇒ Write a test class that uses the various methods of the `Time` class to show that they work. When testing this version, the test code and the time class code should be in different directories, and the test code should contain an "`import version1.Time;`" statement.[1] Be sure to test setting hours, minutes, or seconds outside of the usual range to see what happens.

Show that you can convert a `Time` to a `String` *without* writing additional code: for instance "`System.out.println("The time is"+t)`" should work for a `Time t`.

Time Class — version 2

Package this version in a package called "`version2`".

This version should have identical public method signatures and testing, but each `Time` object should have a single member variable that represents the number of seconds since midnight.

⇒ In the code comment before this `Time` class, comment on which methods are easier, and which methods are more difficult for this version.

⇒ Again, write a test class that uses the various methods of the `Time` class to show that they work.

Theory

Be sure that you have read the chapter about interfaces of the *Big Java*: *Early Objects* text.

Sorting Experiments (`version1b` and `verion2b`)

Using your version 1 and version 2 `Time` classes, attempt to sort a `Time []` array using `java.util.Arrays.sort`. This will produce an error.

⇒ Capture the precise error message. Does the error occur at compile or run time?

Now create new packages `version1b` and `version2b` that are the same as `version1` and `version2` except that the class starts with

```
public class Time implements Comparable<Time> { ...
```

[1]You may choose to use deeper packaging, for instance, `import lab3.version1.Time;`. The same remark applies for all of the versions.

The various classes should all have the following public methods unless otherwise specified.

- Constructors
  - `Time()` (creates midnight),
  - `Time(h,s,m)`, and
  - `Time(Time t)` (initialize from another `Time` object).
- Accessor methods
  - `getHour`,
  - `getMinute`, and
  - `getSecond`

  that return the corresponding value from the object. The hours should be between 0 and 23, and the minutes and seconds should be between 0 and 59.
- Mutator methods
  - `setHour`,
  - `setMinute`, and
  - `setSecond`

  to set the corresponding attributes of a `Time` object. These should ensure that the resulting time is legitimate. Decide and document what happens when you, say set the number of seconds to 75.
- A mutator method
  - `public void advanceBy(int seconds) { ... }`

  that changes the time by a given number of seconds.
- A method
  - `public String toString() { ... }`

  that produces a string like "`22:03:12`". The hours should be between 0 and 23, and the minutes and seconds should be between 0 and 59.
- A method
  - `public int compareTo(Time t) { ... }`

  that produces the number of seconds from `t` to `this`. That is, `t.advanceBy(this.compareTo(t))` should set `t` to the same time as `this`.
- A method
  - `public boolean equals(Time another) { ... }`

  that returns `true` if and only if the times have the same value.

Figure 1: Time class features

```
public interface TimeInterface
{
  int getHour() ;
  int getMinute() ;
  int getSecond() ;
}
```

Figure 2: Time interface specification

Repeat the sorting experiment. (It should now work).

⇒ Explain your test results.

Implementing your own interface (`version1c` and `verion2c`)

Create an interface `TimeInterface` that looks like Figure 2 in a separate `.java` file. Write test code to determine something like

```
TimeInterface ti = new version2.Time(12,30,0) ;
```

works "out of the box". (It shouldn't.)

⇒ Capture the error message.

⇒ Create new packages `version1c` and `version2c` with `Time` classes that explicitly "`implements TimeInterface`". Now test code like

```
TimeInterface ti = new version2c.Time(12,30,0) ;
```

works. (It should.)

⇒ Explain your results.

⇒ What follows is a sequence of questions about how your code works. For each question, ensure that there is output in your script file that shows the answer to the question. Comment on why you get the results that you do.

- Can you create a `TimeInterface` array that contains a mixture of `version1c.Time` and `version2c.Time` objects?
- What happens with code like?

```
TimeInterface ti = new version2c.Time(12,30,0) ;
System.out.println(ti.getSecond()) ;
```

- What happens with code like?

```
TimeInterface ti = new version2c.Time(12,30,0) ;
ti.setSecond(12) ;
System.out.println(ti.getSecond()) ;
```

- What happens with code like?

```
TimeInterface ti = new version2c.Time(12,30,0) ;
System.out.println(ti) ; // Do you expect a hex address?? Why?
```

⇒ Explain your results.

Here is a list summarizing the packages to create:

version1 A version with separate variables for hours, minutes, seconds.

version2 A version with the same attributes and behaviours as `version1`, but which uses a single variable tracking the time since midnight.

version1b `version1` with an added "`implements Comparable<Time>`".

version1c `version1` with an added "`implements TimeInterface`".

version2b `version2` with an added "`implements Comparable<Time>`".

version2c `version2` with an added "`implements TimeInterface`".

package(s) for test drivers All of the `public static void main(...)`'s should be in a package separate from the list above. Having the various test classes in the same package is acceptable. It is also acceptable to put `TimeInterface.java` in the appropriate test package(s).

All of the package names *may* be nested deeper, for instance, `lab3.version2c`.

Figure 3: Packages to create in Lab 3