

Section 6: Inheritance and Interfaces

Pair up with anyone who is agreeable to pairing up with you, pick the first driver, and get to work!

PART I: Creating a Ticket Interface

You've probably bought a raffle ticket at least once – one of those paper tickets with a several-digit number on it. This week, you and your partner will be writing classes that support the creation of such tickets. Because there are many similar kinds of tickets with common behaviors, an interface that specifies the common operations is a good place to start.


1. Open DrJava. There's nothing to load; we'll be creating our interface from scratch.
2. Using the example from class last week as a guide (see the slide titled “User-Defined Interfaces” from <http://www.cs.arizona.edu/classes/cs127b/fall15/slides/20150923-.html>), create an interface named `TicketGeneratable.java` containing the following method headers:

- (a) `public String issueTicket()`
- (b) `public int qtyIssued()`
- (c) `public int firstIssued()`
- (d) `public int lastIssued()`

(The example in class included a constant; our interface doesn't have one.)

3. Save your interface using the filename `TicketGeneratable.java`.

(Yes, that's it for this checkpoint. Might be the easiest one all semester!)

 **CHECKPOINT 1** Raise your hand. Your SL will come over and make sure you've included everything your interface requires.


PART II: Creating the TicketGenerator Class

A basic ticket generator only supports the methods of the `TicketGeneratable` interface.

1. Open a new file in DrJava by clicking on the “New” icon in the upper left.
2. Start by typing the first line of a new public class named `TicketGenerator` that implements the `TicketGeneratable` interface.
3. A `TicketGenerator` object is going to need two variables and one constant. Declare these at the top of the class:
 - (a) An public integer constant `NONE_ISSUED` representing the value `-1`.
 - (b) A boolean variable `anyIssued` initialized to false.
 - (c) An integer variable named `nextNumber`.

(Continued ...)

4. Next, we need a no-argument constructor that sets `nextNumber` to zero. Write it!
5. Because `TicketGenerator` implements `TicketGeneratable`, it must have those four methods. Create all four of them, such they they each do what we need them to do:
 - (a) `issueTicket()`: Use `String`'s `format(<format>, <args>)` method with a format string of `"%06d"` to create a six-digit string from `nextNumber`'s current value. Then increment the `nextNumber` counter, change `anyIssued` to true (because we're issuing a ticket), and return `nextNumber`'s six-character string representation.
 - (b) `qtyIssued()`: Return the number of tickets this object has issued. (Hint: It's closely related to the number of the next ticket.)
 - (c) `firstIssued()`: This method returns the number of the first ticket this object issued. For `TicketGenerator`, that's always zero . . . unless no tickets have been issued, in which case the method returns the value of `NONE_ISSUED`.
 - (d) `lastIssued()`: Similar to `firstIssued()`, this method returns the number of the last (most-recently-generated) ticket the object issued. Again, if no tickets have been issued, the method returns the value of `NONE_ISSUED`.
6. Visit the class web page, find the `Section6.java` program, and load it into DrJava. This program is a testing program for the classes you're writing in today's section.
7. Compile and run `Section6.java`. If all of your methods are coded correctly, the output should end with the message: `==> Congratulations! All tests passed!`

 **CHECKPOINT 2** Raise your hand. Your SL will come over and verify that your `TicketGenerator` class is passing the tests.

PART III: Creating `RaffleTicketGenerator` from `TicketGenerator`

With `TicketGenerator` done, we can turn our attention to raffle tickets. These need a little more functionality than do basic tickets, while retaining the same basic behaviors. Specifically, we need to be able to generate ticket numbers in new ranges, so that people with numbers from previous raffles don't try to re-use their numbers. And, of course, we need to draw a winner of the raffle. Extra functionality means extra methods, but to retain the functionality that `TicketGenerator` provides, we need . . . inheritance!

1. In `Section6.java` are two comment lines with lots of equal signs and the text "Remove this line at the start of Part III!" Do it; delete both of those lines, but only those two lines.
2. On the class web page is the file `RaffleTicketGenerator.java`. Load it into DrJava.
3. The first line of the `RaffleTicketGenerator` class is missing. Create it, keeping in mind that this class inherits from `TicketGenerator` and also inherits from `TicketGeneratable`.
4. `RaffleTicketGenerator` needs two constructors. The no-argument constructor is very nearly the same as the no-argument constructor for `TicketGenerator`; we've supplied it for you. The second constructor accepts the starting number for the sequence of raffle numbers, and uses it to initialize `nextNumber` and `startNumber` (the new instance variable in `RaffleTicketGenerator`). Using the no-argument constructor as a guide, write this second constructor.

(Continued . . .)

5. Thanks to inheritance, we don't need to re-write any of the methods from `TicketGenerator` that don't change. Specifically, `issueTicket()` and `lastIssued()` can be reused. We do need to 'replace' `qtyIssued()` and `firstIssued()`, because they depend on the sequence starting number, which in this class doesn't need to be zero. **Question:** Is this 'replacement' of these two methods an example of overriding or overloading? Make note of your answer; your SL will be asking — and will want an explanation!
6. Using the `TicketGenerator` version as a starting point, write the new version of `firstIssued()`; we've given you the new version of `qtyIssued()`.
7. `RaffleTicketGenerator` needs two new methods. Create both of them, such they they each do what we need them to do:
 - (a) `reset()` is a void method that accepts a new starting number for a raffle ticket sequence and resets the state of the current object so that it appears to be a brand-new `RaffleTicketGenerator` object, ready to issue the given starting number as the first number of the sequence. Thus, `reset()` is *almost* the same as the second constructor; the difference is that `reset()` does not create a new object; instead, it changes the state of the current object.
 - (b) `drawWinner()` randomly selects and returns one number from the range of ticket numbers that have been issued by this object. If no numbers have yet been issued, this method returns the value of `NON_ISSUED`.
8. Time to test the `RaffleTicketGenerator` methods! In `Section6.java`, we've provided a start on the testing of `RaffleTicketGenerator`, but you need to complete it. Near the bottom is a comment that says:


```

/* Part III: Your job is to add the assertion-based testing
 * for the second constructor. Fortunately, you have the
 * above tests as guides. Suggestion: Copy-n-paste the
 * code that tests Constructor 1 of RaffleTicketGenerator,
 * and adjust it to test Constructor 2.
 */

```


Do it!

9. Compile and run `Section6.java`. When you no longer have syntax and logic errors — that is, when you see the congratulations message — you're done.

 **CHECKPOINT 3** Raise your hand. Your SL will come over and ask about your answer to the overriding/overloading question and see that your testing was a success.

PART IV: Clean Up!

1. Log out of your computer.
2. Pick up your papers, writing implements, cell phones, trash, etc.
3. Push in your chair(s).

 **CHECKPOINT 4** Raise your hand. Your SL will come over and count how many sparkles of cleanliness your workspace emits per unit of time.

You're free to go! But, if you have time, we recommend that you use it to work some more on the programming assignment (individually, of course), if you're not already done.