

1 Objectives

- Use Java exceptions and generics together.
- Use inheritance and polymorphism to make a special purpose subclass.
- Create your own exceptions.
- Use Javadoc.

2 Preliminaries

Make a new project for today's lab. Import the files for this lab from

```
/home/accounts/COURSES/csci204/2009-fall/student/labs/lab05
```

3 Exercise 1: Streams and Exceptions

An important ADT in computer science is the stream. A *stream* carries an ordered sequence of data of undetermined length. For flexibility, Java has many streams. Streams are used to read and write files as well as to communicate over a network.

By now, you are familiar with the `Scanner` class. It uses a stream to input text and interprets it as integers, booleans, string, etc as requested. Up till now, you have had to assume that a user entered an integer when your program asked for one and not some other type such as a boolean. Now you can use exceptions to take charge and handle bad user input.

Create a Java class named `StubbornInput` that will prompt for user input. Internally, it will use a `Scanner` to actually read the input. The `StubbornInput` class will also contain methods similar to the `Scanner` class but these methods will stubbornly ask the user for input again and again until data of the correct type is entered.

This class needs methods for

```
public int nextInt(String prompt, String errorMessage);  
public boolean nextBool(String prompt, String errorMessage);
```

The first parameter of each method is a string to print when requesting information from the user. The second parameter is a string to print when yelling at an unruly user. Using the parameters, each method should repeatedly ask the user for input until the correct type of input is given. You will need to catch and handle the exceptions that are generated when bad user input is given. Question: what is the easiest way to find out what exception will be thrown?

There are two ways to setup your exception handling:

- You can use recursion and have your catch block call your function again.
- You can use iteration via a loop and have the try-catch block inside your loop with a boolean for “am I done yet”. Simply return the data from inside the loop when you finally get it. Question: which form of loop is appropriate for a situation where the number of repetitions is unknown but the loop body will run at least once?

In both cases, you may need to put a call to the `nextLine` of `Scanner` in your catch block. This tells the `Scanner` to move along to the next possible input in case it threw the exception mid-input. If you run your program on one bad input and it goes into an infinite loop, you may need this line. However, it is possible to solve this problem without it.

Run `TestStubbornInput` to test your class.

4 Exercise 2: A Simple Way to Use Streams: Readers

4.1 Learn to read the ASCII chart

Characters stored and displayed on computers are, as you already know, really just stored as numbers. A number of standards have been developed over the years that provide a mapping from numbers to characters. One of the most familiar and widely used is the ASCII chart. Familiarize yourself with this set of codes for common characters.

<http://www.unicode.org/charts/PDF/U0000.pdf>

The chart itself is in hexadecimal. To convert from hexadecimal to decimal, you need to multiply the number in a character’s corresponding top row cell by sixteen and add the number in the corresponding first column cell. (Note: The letters A-F represent the numbers ten to fifteen)

4.2 The Reader class

The `Reader` class, much like the `Scanner` class, is a simplified interface to an input stream. However, a `Reader` generally has a much simpler interface, intended to convert input streams into characters or strings. Look up the `Reader` in the standard Java API Javadoc and familiarize yourself with the abstract `Reader` class and its subclasses.

<http://java.sun.com/javase/6/docs/api/java/io/Reader.html>

4.3 Testing a Reader

In your imported files, there are two classes `ReaderTest` and `MinValueReader`. In `MinValueReader`, you need to initialize the constants `FIRST_NON_WHITESPACE_CHAR_VALUE` and `LAST_NON_WHITESPACE_CHAR_VALUE` to the correct ASCII values for the characters they describe. When they are set correctly, run the `ReaderTest` a few times to get a feel for what `MinValueReader` does. Try the inputs "abcabc" and "123123". Note that the `read()` method will pull a new character from the input stream, so even whitespace characters will be considered as input to the reader. What happens when you input a character that is lower on the ASCII chart than '0' like a newline?

Try changing around the constants in `ReaderTest` and using various inputs. Notice that both classes are missing their documentation? Write documentation for both classes.

The `read()` method in `MinValueReader` is probably responsible for any of the `NumberFormatException`s you saw in when running `ReaderTest`. Use a try-catch block to catch the exception in `read()` and return `-1` in the case that an exception is thrown. Test out your updated class and modify your comments as necessary.

5 Exercise 3: Create your own exception

For this exercise you are to create your very own exception. When you create a new exception, the convention is to end the name with "Exception". Let's name your new class `MinValueException`. For each new exception you need to create a new class which extends (inherits from) an existing exception. In general, you need to consult the Java exception hierarchy (see page 504 in Big Java) to determine which exception your new exception should inherit. See chapter 11.6 for instructions on how to create an exception. Make sure it is a checked exception.

Next, make a new copy of the `MinValueReader` class (you can use copy and paste in Eclipse) and name it `EnhancedMinValueReader`. Have your `EnhancedMinValueReader` throw your new exception instead of returning `number.intValue()` in the `read()` method. Instead, catch the generated exception and return the intended value at that point. Update `ReaderTest` to use `EnhancedMinValueReader` instead of `MinValueReader`. Run the tests a few times and update your Javadoc comments.

6 Exercise 4: Using Java generics

Generics are not only useful for allowing you to remove casts when you get objects out of a class, but they can be useful error detection tools, as well!

Change your `EnhancedMinValueReader` class to take a single type parameter that is a subtype of `Number`. Correspondingly, change the `number` field to be generic.

Do you notice any errors in `ReaderTest` as a result of the change? No errors, but there are new warnings in `ReaderTest`. Those warnings say you are using a generic class without providing a type parameter. As a result, the type of the unparameterized class is called a *raw* type. Java did not actually have generics until release 5. As a result, it allows you to use raw types so that code written prior to the release of Java 5 is still valid. However, using raw types in post Java 5 code is generally considered bad practice.

Add an `Integer` type parameter to this use of `EnhancedMinValueReader` so that it is no longer a raw type. Now what happened?

By changing the type parameter, you are now claiming that you only want to use integers in this instance of `EnhancedMinValueReader`. So, when the `number` is changed, you are no longer allowed to use other incompatible types (such as doubles) where the parameterized type is expected. Fix the error without changing the type parameter (there are multiple possible ways to do this).

Update all of your Javadoc comments to take into account your changes.

7 Exercise 4: Javadoc

Look at the Java API (linked off the course website). These pages are a form of documentation called a Javadoc. They show the public and protected portions of the classes in the Java libraries. In this format, they are part of the User's Manual for Java.

It is also possible to have Javadoc show the private member data and methods for classes. That would be part of the technical specification.

In this section, you will learn how to use Javadoc. First, let's have eclipse make some Javadoc. Select `Program` and `Generate Javadoc` from the Eclipse menus. The source should be your project for this lab. The destination should be `lab05/doc`. Note that you have options to include public, protected, and private information. The default is public, which is good for publishing APIs, but you should select private in order to use Javadoc for creating technical specifications. Choose private for this lab.

Then hit finish. Many black lines of text will zip by in the console. You should not see any red ones.

In your new doc directory, you will be able to find the index.html file. Double click on it. If you get an error message, say ok and wait a minute. Either a new window will appear or double click it again to get the new window. It is looking for firefox in the wrong place. Erase the line with firefox in it and just write firefox and hit ok. It should bring up your Javadoc from then on. You should not have to do this again.

Look at the “A quick example of things Javadoc can do and the Java file.” link on the course website. Put a few embedded HTML tags in your Javadoc (such as `` or `<pre></pre>`) to enhance your comments. Embedded HTML will allow you to include almost any type of formatting in your Javadoc. This can be useful for embedding images or tables for your projects. In this way, you can include CRC cards in your Javadoc without having to maintain them separately.

Regenerate your Javadoc with the embedded HTML.

8 Upon Completion

Commit your files to your repository.