# 12 Java Exceptions, Linked Lists, Queues, Stacks

## Activities

- Use exceptions to detect and handle errors
- Learn to use and implement doubly linked list, stack, and queue.

## Lab Preparation

Download the provided zip file and unzip it. Create a new Eclipse project named **Lab12**. Add the given code to the project and link the external JAR jpt.jar to the project. You should have the following Java files:

- **interface** *ISame* that should be implemented by any class that needs to be tested for equality.
- **class** *Examples* that contains the test suite. It extends *SimpleTestHarness*.
- **class** *SimpleTestHarness* that manages the test evaluation and reporting.
- **class** *Interactions* that provides a framework for user interactions.
- **class** *MyLinkedLlist* that you will complete.
- **class** *MyStack* that you will implement.
- **class** *MyQueue* that you will implement
- **class** *Person* to provide data for our data structures

## Activity 1: Using Exceptions

Exceptions provide a general method for dealing with errors that occur during the execution of a program. When an error occurs, an exception object is created. This exception can then be detected and handled smoothly, instead of causing the entire program to fail.

### Catching Exceptions

Exceptions are either automatically created by Java when certain types of errors occur or can be created manually by creating a new instance of an exception class and throwing that exception.

As an example, consider the case of detecting when a divide by zero error occurs. Look at the divide method in the *Interactions* class. This method takes two int's, x and y, and returns x/y. What happens if y is 0? In this case, Java throws an *ArithmeticException* that represents the error. We use a try-catch to handle the occurrence of this error in a smooth way. The format of a try-catch is as follows:

```
try {
  // some code that could result in an exception being thrown
} catch (ExceptionType1 e) {
  // handle exceptions of Type1
} catch (ExceptionType2 e) {
  // handle exceptions of Type2
}
```

There is one *t*ry-block, which contains code that can possibly create an exception. This is followed by one or more *c*atch-blocks, each of which deals with a specific type of exception. In the case of the divide method there is only one type of exception we are interested in, so we have only one catch-block.

Try running the project and executing the divide method with different values for $x$ and $y$. What happens when you enter 0 for $y$?

### Manually Creating and Throwing Exceptions

You can manually create and throw an exception when an abnormal situation occurs. In this case, what situations are abnormal are defined by the programmer. For example, we may only want to allow our program to work with int's less than 1000. See the multiply method in the *Interactions* class. This method takes two int's, $x$ and $y$, and returns x*y. If $x * y > 1000$ an exception is created and immediately caught. Try running this method with various inputs and observe the results.

### Defining a Method to Throw Exceptions

In the case of multiply, we created and immediately caught an exception. Sometimes we may not know how to immediately handle the exception and want to pass the exception back to where the method was called and handle the error there. We define methods to throw errors in the following way:

```
void foo(...) throws ExceptionType1, ExceptionType 2 {
  // some code that can throw ExceptionType1 and ExceptionType2
}
```

Any method that calls *foo* should place that call of *foo* in a try-catch block to handle the error or should itself be declared to **throw** those exceptions to a higher level.

As an example, see *multiplyAndThrow* in *Interactions*. This method is similar to multiply but the exceptions are not immediately handled. Try running this method and causing an exception to be thrown. Notice that the exception is printed to the console in red. This signifies that the exception was never caught and was reported by Java as a fatal error.

The method *multiplyAndCatch* calls *multiplyAndThrow* and handles the exceptions that it can throw with a try-catch statement.

**Exercise:**

Add the methods *subtractAndThrow* and *subtractAndCatch* to the *Interactions* class. These methods should be similar to *subtractAndThrow* and *subtractAndCatch* described above. Given two int's, $x$ and $y$, they should return $x - y$ and should produce an error if the result is negative.

## Activity 2: Defining Exceptions

The programmer can define new Exceptions, if necessary. In our case, the *ArithmeticException* does not convey the fact that we exceeded our own bound for the size of the result. We can instead define a class *BigNumException* that extends *Exception*:

```
class BigNumException extends Exception{
   BigNumException(String message){
      super(message);
   }
}
```

1. Add this class to your project and change the exceptions in the methods that do multiplication to use the *BigNumException*.

2. Add tests for the methods that do multiplications.

## Activity 3: Using Collections

The Collection interface is the root of a hierarchy of interfaces that represent groups of objects. There are three main types of collections, each with a corresponding interface:

- **List** — an ordered collection of objects. Each object in a list has a specific position in the list. Lists can contain multiple occurrences of the same object. The lists you have been designing and using so far are all examples of this kind of collection. We will use only list collections in this lab, but you already used other types of collections in earllier labs and assignments.

- **Set** — an unordered collection of objects that can not contain duplicates. The set collections correspond closely to the mathematical notion of sets.

- **Map** — an unordered collection that relates keys to values. An example of this type of data is a dictionary. The words in the dictionary are keys that are mapped to their definitions, the values.

### Linked Lists

Find the *LinkedList* class in the Java API. Read the Javadocs to see what methods it provides. Notice that it implements both the *Stack* and *Queue* interface. Look up the Javadocs for both of those interfaces as well.

We will first learn how to test this implementation of linked lists, then build our own *MyLinkedList*. Finally, we will use our implementation of a *d*oubly likked list to implement our version of the *MyStack* and *MyQueue*.

We will use this list class as the basis of two new classes, *MyStack* and *MyQueue*.

### Testing LinkedList Class

Methods *testStandardLinkedList* and *testListIteratorOfStandard* are examples of how to design tests for some of the methods implemented by the *LinkedList*

class. We will use variants of these methods to test our implementation of a linked list data structure.

### Exercise: Implement Linked List Iterator

The class *MyLinkedList* provides an implementation of a linked list where each element is an instance of a *Node* class:

```
class ListNode<E>{
  E data;
  ListNode<E> prev;
  ListNode<E> next;
  ListNode(E data, ListNode<E> prev, ListNode<E> next){
    this.data = data;
    this.prev = prev;
    this.next = next;
  }
}
```

Draw a picture that illustrates how the new elements are added to this linked list. Once you undserstand the implementation, complete the class definition of the *MyLLIterator* inner class. Run the tests in the *Examples* class.

### Exercise: Define Stack Methods

Look up the Javadocs for the interface *Stack*. Complete the implementation of the class *MyStack* using the class *MyLinkedList*. As an example, see the *testStack* method in the *Examples* class.

Methods to add to *MyStack*:

- push: add an element to the top of the stack
- pop: return the element at the top of the stack and remove it from the stack
- peek: return the element at the top of the stack without removing it
- empty: return true if the stack is empty, return false otherwise

### Exercise: Define Queue Methods

Look up the Javadocs for the interface *Queue*. Complete the implementation of the class *MyQueue* using the class *MyLinkedList*.

Methods to add to *MyQueue*:

- element: return the element front of the queue without removing it
- offer: add an element to the back of the queue
- poll: return the element at the front of the queue and remove it from the queue
- peek: return the element front of the queue without removing it, return null, if the queue is empty
- remove: return the element at the front of the queue and remove it from the queue
- empty: return true if the queue is empty, return false otherwise