Herbert Schildt's *Java: A Beginner's Guide*, 6th ed.

Chapter 9 Self Test Answers

1. **What class is at the top of the exception hierarchy?**

   The class that is at the top of the exception hierarchy is known as *Throwable*. Throwable has two direct subclasses, *Exception* and *Error*. *Exception*s are the responsibility of programmers, *Error*s are generated in the Java Virtual Machine (JVM) and are beyond the control of the normal programmer.

2. **Briefly explain how to use *try* and *catch*.**

   Using *try* and *catch* are the basic exception handling syntax of Java. The programmer simply surrounds any code which could possibly generate an exception with a *try* code block. Immediately following the *try* code block is a *catch* block, which informs the program how to deal with any exception that is thrown by the code within the *try* block. For example:

   ```
   try {
     // insert any code which could generate an exception
   }

   catch(SomeJavaException exc){
     // How to handle the SomeJavaException
   }
   ```

3. **What is wrong with this fragment?**

   ```
   // ...
   vals[18] = 10;
   catch (ArrayIndexOutOfBoundsException exc){
     // handle error
   }
   ```

   This code does not contain a *try* code block in which the exception could be generated. Exceptions will only be caught by *catch* blocks if they are generated within a *try* block.

4. **What happens if an exception is not caught?**

   If an exception is not caught within the code, then it will cause a run-time error when the program runs. This results in a technical output by the program which is a very unpleasant experience for the end user.

5. **What is wrong with this fragment?**

   ```
   class A extends Exception { ...

   class B extends A { ...

   // ...

   try {
     // ...
   }

   catch (A exc) {...}
   catch (B exc) {...}
   ```

   The problem with the above code fragment is that the superclass (*A* catch comes before the subclass (*B*) catch. The superclass catch will catch every exception this code generates, and therefore there will never be a B exception. There is no way to reach the last line of this code, so no matter how the programmer of the subclass wants the exception to be handled, it will instead follow the directions laid out in the superclass.

6. **Can an inner *catch* rethrow an exception to an outer *catch*?**

   In Java it is possible to rethrow an exception. So yes, an inner *catch* statement can rethrow an exception that will be caught by an outer *catch* statement.

7. **The *finally* block is the last bit of code executed before your program ends. True or False?**

   The above statement is *False*. The *finally* block will be executed whenever execution leaves a *try*/*catch* block, no matter what conditions cause the exit. It is not code at the end of the program, but rather at the end of a *try*/*catch* code block.

8. **What type of exceptions must be explicitly declared in a *throws* clause of a method?**

   Exceptions that are subclasses of *Error* or *RuntimeException* don't need to be specified in a *throws* list. All other types of exceptions must be declared - failure to do so causes a compile-time error. Specifically this includes those custom exceptions created by the user as well as Java's built-in exception classes.

9. **What is wrong with this fragment?**

```
class MyClass { // ...}
// ...
throw new MyClass();
```

   The problem with the above fragment is that it does not extend *Throwable*, the superclass of all exception handling. It is not possible for *MyClass* to throw any exception until it has extended *Throwable* (or one of its subclasses).

10. **In Question 3 of the Chapter 6 Self Test, you created a *Stack* class. Add custom exceptions to your class to report stack full and stack empty conditions.**

    Below is the code supplied in the Answers section of this book - please note that the code does not work (even after careful review and debugging). When combined with the StackDemo class that was created in Chapter 6, this code generates 4 errors as seen below. It seems that the *push* and *pop* methods have been rendered inoperable:

```
StackDemo.java:103: error: unreported exception StackFullException;
  must be caught or declared to be thrown
    stack1.push((char)('1' + i));
               ^

StackDemo.java:114: error: unreported exception StackEmptyException;
  must be caught or declared to be thrown
    ch = stack1.pop();
                ^

StackDemo.java:122: error: unreported exception StackEmptyException;
  must be caught or declared to be thrown
    ch = stack2.pop();
                ^

StackDemo.java:130: error: unreported exception StackEmptyException;
  must be caught or declared to be thrown
    ch = stack3.pop();
                ^

4 errors
```

    Here is the code, in its non-working state:

```
/*
* StackDemo - a demonstration of how a stack works
* Remember - a stack is last-in, last-out (LILO)
* Throws exception when full or empty
* From Herbert Schildt's Java: A Beginner's Guide, 6th ed.
* Page 628-629, 637-638
-*/

// An exception for stack-full errors
class StackFullException extends Exception {
  int size;

  StackFullException(int s){
    size = s;
  }

  public String toString(){
    return "\nStack is full.  Maximum size is " + size;
  }
}

// An exception for stack-empty errors
class StackEmptyException extends Exception {
  public String toString(){
    return "\nStack is empty.";
  }
}

// A stack class for characters
class Stack {
  private char stack[]; // this array holds the stack
```

```
  private int tos; // top of stack

  // Construct an empty Stack given its size
  Stack(int size){
    stack = new char[size]; // allocate memory for stack
    tos = 0;
  }

  // Construct a Stack from a Stack
  Stack (Stack ob){
    tos = ob.tos;
    stack = new char [ob.stack.length];

    // copy elements
    for (int i=0; i < tos; i++){
      stack[i] = ob.stack[i];
    }
  }

  // Construct a stack with initial values
  Stack(char a[]){
    stack = new char[a.length];

    for (int i = 0; i < a.length; i++){
      try {
        push(a[i]);
      }
      catch(StackFullException exc){
        System.out.println(exc);
      }
    }
  }

  // Push characters onto the stack
  void push(char ch) throws StackFullException{
    if (tos == stack.length){
      throw new StackFullException(stack.length);
    }

    stack[tos] = ch;
    tos++;
  }

  // Pop a character from the stack
  char pop() throws StackEmptyException{
    if (tos == 0){
      throw new StackEmptyException();
    }

    tos--;
    return stack[tos];
  }
} // end Stack

// Demonstrate the Stack Class
public class StackDemo {
  public static void main(String args[]){

      // construct a 10-element empty stack
      Stack stack1 = new Stack(10);

      char name[] = {'B', 'r', 'a', 'd'};

      // construct stack from array
      Stack stack2 = new Stack(name);

      char ch = 0;
      int i;

      // put some characters into stack1
      for (i=0; i < 10; i++){
        stack1.push((char)('1' + i));
```

```
        }

        // construct stack from another stack
        Stack stack3 = new Stack(stack1);

        System.out.println();

        // show the stacks
        System.out.printf("Contents of stack1: ");
        for (i=0; i < 10; i++){
          ch = stack1.pop();
          System.out.print(ch);
        }

        System.out.println("\n");

        System.out.printf("Contents of stack2: ");
        for (i=0; i< 4; i++){
          ch = stack2.pop();
          System.out.print(ch);
        }

        System.out.println("\n");

        System.out.printf("Contents of stack3: ");
        for(i=0; i < 10; i++){
          ch = stack3.pop();
          System.out.print(ch);
        }

        System.out.println("\n");
    } // end main
} // end StackDemo
```

11. **What are the three ways that an exception can be generated?**

    There are three ways that an exception can be generated. The first is an error in the Java Virtual Machine (JVM), something that is generally beyond the scope the average programmer will ever have to debug. The second, much more common way that an exception can be generated is an error in the programmer's code. The third, and most common way is for an exception to be explicitly thrown by the code itself - that is, an error is expected by the programmer and a way to handle it is built into the program.

12. **What are the two direct subclasses of *Throwable*?**

    The two direct subclasses of *Throwable* are *Error* and *Exception*. The *Error* class is only used when the Java Virtual Machine is creating an error, so there is no need for the average programmer to account for it in his/her code.

13. **What is the multi-catch feature?**

    Using the multi-catch feature, one catch statement can catch more than one exception (there is no limit of how many exceptions can be caught!).

14. **Should your code typically catch exceptions of type *Error*?**

    There is no need to try to catch exceptions of the type *Error* as these come exclusively from the Java Virtual Machine and are not part of the normal programmer's scope of problem-solving.