TASK

Defensive Programming

Visit our website

Introduction

Welcome to the Defensive Programming Task!

In this task you will be introduced to defensive programming and how to write code that can respond gracefully to such things as invalid data, events that you might assume would never happen and the mistakes of other programmers.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact your mentor to get support on any aspect of your course.

The best way to get help is to login to **www.hyperiondev.com/portal** to start a chat with your mentor. You can also schedule a call or get support via email.

Your mentor is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

What is Defensive Programming?

The idea of defensive programming is based on defensive driving. With defensive driving, you have the mindset that you can never predict what other divers will do while on the road. Therefore, you do not assume anything and make sure that if they do something dangerous, you won't be hurt. In that way, even if it might be the other driver's fault, you take charge of ensuring no harm comes to you. This mindset is very similar to the one adopted in defensive programming. Defensive programming is the recognition that programs will have problems and modifications, and that a smart programmer will develop code accordingly.

The whole point of defensive programming is guarding against errors you don't expect.
-Steve McConnell, Code Complete

Protecting Your Program From Invalid Inputs

In the software development world there is a popular expression: "Garbage in, garbage out." This expression means that incorrect or poor-quality input will produce faulty or "garbage" output.

As you can probably imagine, having your program generate invalid output is never ideal. A good program should never output garbage, even if it takes in garbage. A good program should always use either, "garbage in, nothing out," "garbage in, error message out," or "no garbage allowed in" instead.

Below are three ways to handle garbage in:

- **Check the values of all data from external sources:**
  There are many ways in which data can be input into a program. We can get data from files, users, the network, or other external interfaces. When getting data from one of these sources, we must check to be sure that the data falls within the allowable range. Numeric values should be within tolerances and strings should be short enough to handle. If a string is used to represent a restricted range of values (such as an employee ID for example), you must be sure that the string is valid for its intended purpose. If it is not, it should be rejected.

- **Check the values of all routine input parameters:**
  This is essentially the same as checking data from an external source.

However, the data comes from another routine rather than from an external interface.

- **Decide how to handle bad inputs:**
  Once an invalid input is detected, you might choose any of a dozen different approaches to handle it. These approaches are discussed in more detail later on in this task.

Assertions

An assertion is code, used during development that allows a program to check itself as it runs. When everything is operating as expected, the assertion is true. When the assertion is false, it means it has detected an unexpected error in the code. For example, if a system assumes that a customer information file will never have more than 50 000 records, the program might contain an assertion that the number of records is less than or equal to 50 000. As long as the number of records is less than or equal to 50 000, the assertion will be silent. However, if it encounters more than 50 000 records, it will "assert" that an error has occurred.

Assertions are useful in large, complicated programs and high-reliability programs. They enable programmers to flush out mismatched interface assumptions quickly — errors that creep in when code is modified.

An assertion takes two arguments: a boolean expression that describes the assumption that's supposed to be true, and a message to display if it isn't.

The example below shows what a Java assertion would look like if the variable denominator were expected to be nonzero:

```
assert denominator != 0 : "denominator is unexpectedly equal to 0.";
```

This assertion ensures that the denominator is not equal to 0. The first argument, `denominator != 0`, is a boolean expression that evaluates to *true* or *false*. The second argument is a message to print if the first argument is false (i.e. if the assertion is false).

Assertions are used to document assumptions made in the code and to flush out unexpected conditions.

Assertions can be used to check assumptions like the following:

- That an input (or output) value falls within its expected range.
- That a file or stream is open (or closed) when a routine begins (or ends) executing.
- That a file or stream is at the beginning (or end) when a routine begins (or ends) executing.
- That a file or stream is open for read-only, write-only, or both read and write.
- That a routine does not change the value of an input-only variable.
- That a pointer is non-null
- That an array or other container passed into a routine can contain at least X number of data elements.
- That a table has been initialised to contain real values.

You don't normally want users to see assertion messages in production code. Assertions are primarily for use during development and maintenance. Assertions are normally compiled into the code at development time and compiled out of the code for production. During development, assertions flush out contradictory assumptions, unexpected conditions, bad values passed to routines, and so on. During production, they can be compiled out of the code so that the assertions don't degrade system performance.

Error-Handling Techniques

Assertions are used to handle errors that should never occur in the code, but how do you handle errors that you do expect to occur? Depending on the circumstances, you might want to do one of the following:

- **Return a neutral value**
  It is often best to respond to bad data by simply continuing operation and returning a value that's known to be harmless. For example, a numeric computation might return 0 or a string operation might return an empty string.

- **Substitute the next piece of valid data**
  When processing a stream of data, some circumstances call for simply returning the next valid data. For example, if you're reading records from a database and encounter a corrupted record, you might continue reading until you find a valid record.

- **Return the same answer as the previous time**
  Sometimes you can return some data that you used previously. For

example, if a thermometer-reading software that takes 100 readings per second doesn't get a reading one time, it might return the same value read previously. This is because the temperature is not very likely to change much in 1/100th of a second.

- **Substitute the closest legal value**
  You might choose to return the closest legal value. This is a reasonable approach when taking readings from a calibrated instrument. For example, a thermometer might be calibrated between 0 and 100 degrees Celsius. If you detect a reading less than 0, you can substitute 0, which is the closest legal value and, if you detect a value greater than 100, you can substitute 100.

- **Log a warning message to a file**
  You might also choose to log a warning message to a file when bad data is detected and then continue. This approach can be used in conjunction with other techniques described above, like substituting the closest legal value or substituting the next piece of valid data.

- **Return an error code**
  You could decide that only certain parts of a system will handle errors while other parts will report that an error has been detected and trust that some other routine higher up in the calling hierarchy will handle the error.

- **Call an error-processing routine/object**
  You could also centralise error handling in a global error-handling routine or error-handling object. An advantage of doing this is that error-processing responsibility is centralised, which can make debugging easier. However, a disadvantage is that the whole program will know about this central capability and will be coupled to it. Therefore, if you want to reuse code from one system in another system, you will have to drag the error-handling machinery along with the code you decide to reuse.

- **Display an error message wherever the error is encountered**
  This approach minimises the overhead of error-handling. However, it does have the potential to spread user interface messages through the entire application. This can create challenges when you need to create a consistent user interface, when you try to separate the UI from the rest of the system clearly, or when you try to localise the software into a different language.

- **Handle the error in whatever way works best locally**
  Some designs call for handling all errors locally. The decision of which specific error-handling method to use is left to the programmer designing and implementing the part of the system that encounters the error. This can provide individual developers with great flexibility. Depending on how developers end up handling specific errors, however, this approach also has the potential to spread user interface code throughout the system, which exposes the program to all the problems associated with displaying error messages.

- **Shut down**
  Some systems shut down whenever they detect an error. This approach is very useful in safety-critical applications. For example, if the software that controls radiation equipment for treating cancer patients receives bad input data for the radiation dosage, shutting down is the best option. It is better to reboot the machine than to run the risk of delivering the wrong dosage.

Exceptions

Runtime errors occur if the Java virtual machine (JVM) detects an operation that is impossible to carry out while a program is running. For example, if you enter a double value when your program expects an integer, you will get a runtime error with an *InputMismatchException*.

Runtime errors are thrown as exceptions in Java. An exception is an object that represents an error or a condition that prevents execution from proceeding normally. If the exception is not handled, the program will terminate abnormally. However, you can handle these exceptions so that the program can continue to run, or terminate gracefully.

The example program below reads in two integers and displays their quotient.

```
import java.util.Scanner;

public class Quotient {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();
```

```
    System.out.println(number1 + " / " + number2 + " is " + (number1 /
number2));
    }
}
```

If you enter a 0 for the second number, a runtime error would occur, because you cannot divide an integer by 0. A simple way to fix this error is to add an *if statement* to test the second number, as shown below. We can also rewrite the code quotient using a method.

```
import java.util.Scanner;

public class QuotientWithMethod {

  public static int quotient(int number1, int number2) {
    if (number2 == 0) {
      System.out.println("Divisor cannot be zero");
      System.exit(1);
    }
    return number1 / number2;
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    int result = quotient(number1,number2);
    System.out.println(number1 + " / " + number2 + " is  " + result);
  }
}
```

The method quotient returns the quotient of two integers. If *number2* is 0, it cannot return a value, so the program is terminated. This is a problem as you should not let the method terminate the program. The caller should always decide whether to terminate the program.

Java enables a method to throw an exception that can be caught and handled by the caller, therefore notifying its caller that an exception has occurred. The program can be rewritten, as follows:

```
import java.util.Scanner;

public class QuotientWithException {

  public static int quotient(int number1, int number2) {
    if (number2 == 0)
      throw new ArithmeticException("Divisor cannot be zero");

    return number1 / number2;
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    try {
      int result = quotient(number1, number2);
      System.out.println(number1 + " / " + number2 + " is  " + result);
   }
   catch (ArithmeticException ex) {
     System.out.println("Exception: an integer " + "cannot be divided by zero
");
  }

  System.out.println("Execution continues ...");
 }
}
```

If *number2* is 0, the method throws an exception executing `throw new ArithmeticException("Divisor cannot be zero");`

The value thrown, in this case, *new ArithmeticException("Divisor cannot be zero")*, is called an exception. The execution of a *throw* statement is called 'throwing an exception'. An exception is an object created from an exception class. In this case, the exception class is *java.lang.ArithmeticException*. The constructor *ArithmeticException(str)* is invoked to construct an exception object, where *str* is a message that describes the exception.

When an exception is thrown, the normal execution flow is interrupted. To 'throw an exception' is to pass the exception from one place to another. The statement for invoking the method is contained in a *try* block and a *catch* block. The *try* block

contains the code that is executed in normal circumstances. The *catch* block catches the exception. The code in the *catch* block is executed to handle the exception. The statement after the *catch* block is executed afterwards.

The *throw* statement is similar to a method call, but instead of calling a method, it calls a *catch* block. Therefore, a *catch* block can be thought of as a method definition with a parameter that matches the type of the value being thrown. However, unlike a method, after the *catch* block is executed, the program control does not return to the *throw* statement. Instead, it executes the next statement after the *catch* block.

The identifier, *ex*, in the catch-block header catch (*ArithmeticException ex*) acts as a parameter in a method and is referred to as a catch-block parameter. The type (e.g., *ArithmeticException*) before *ex* specifies what kind of exception the *catch* block can catch. Once the exception is caught, you can access the thrown value from this parameter in the body of a catch block.

Below is a basic syntax of a *try-catch* block:

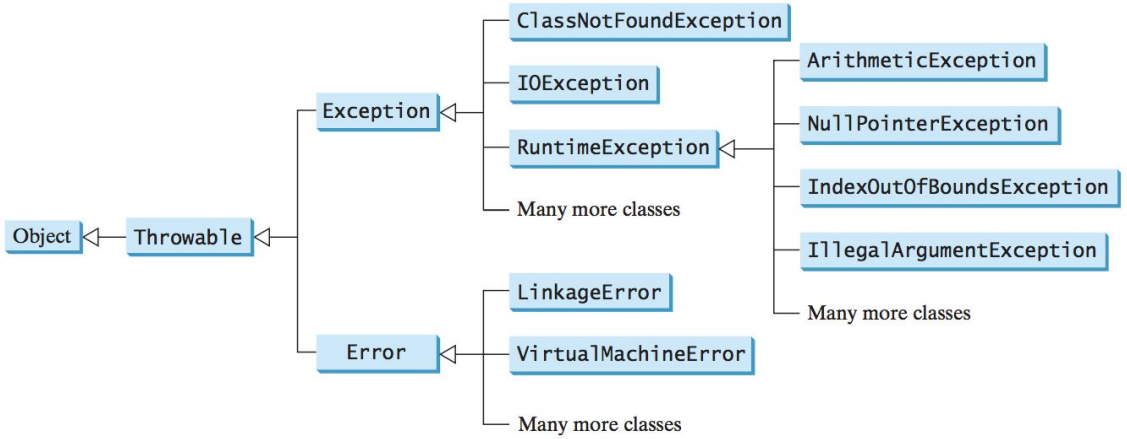
```
try {
  Code to run;
  A statement or a method that may throw an exception; More code to run;
}
catch (type ex) {
  Code to process the exception;
}
```

An exception may be thrown directly by using a *throw* statement in a *try* block, or by invoking a method that may throw an exception. In our *QuotientWithException* program, the main method invokes quotient. If the quotient method executes normally, it returns a value to the caller. If the quotient method encounters an exception, it throws the exception back to its caller. The caller's *catch* block then handles the exception.

The advantage of using exception handling is separation of the detection of an error (done in a *called* method) from the handling of an error (done in the *calling* method).

Exception Types

So far we have only discussed the classes *ArithmeticException* and *InputMismatchException*. However, there are many other types of exceptions than you can use. The diagram below shows some many predefined exception classes in the Java API.



The *Throwable* class is the root of exception classes. All Java exception classes inherit directly or indirectly from *Throwable*. You can create your own exception classes by extending *Exception* or a subclass of *Exception*.

The exception classes can be classified into three major types:

- **System errors** are thrown by the JVM and are represented in the *Error* class. The *Error* class describes internal system errors, though such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

- **Exceptions** are represented in the *Exception* class, which describes errors caused by your program and by external circumstances. These errors can be caught and handled by your program.

- **Runtime exceptions** are represented in the *RuntimeException* class, which describes programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors. The JVM generally throws runtime exceptions.

RuntimeException, *Error*, and their subclasses are known as unchecked exceptions. All other exceptions are known as checked exceptions, meaning that the compiler forces the programmer to check and deal with them in a try-catch block

Unchecked exceptions normally reflect programming logic errors that are unrecoverable and should be corrected in the program. For example:

- a *NullPointerException* is thrown if you access an object through a reference variable before an object is assigned to it; or
- an *IndexOutOfBoundsException* is thrown if you access an element in an array outside the bounds of the array.

Unchecked exceptions can occur anywhere in a program. To avoid overuse of *try-catch* blocks, you should not write code to catch or declare unchecked exceptions.

You should now have some understanding of what exception handling is. We shall now provide an in-depth discussion of exception handling.

Java's exception-handling model is based on three operations:

1. declaring an exception
2. throwing an exception
3. catching an exception

Declaring Exceptions

In Java, the statement currently being executed belongs to a method. The Java interpreter invokes the main method to start executing a program. Every method must state the types of checked exceptions it might throw. This is known as declaring exceptions.

To declare an exception in a method, we use the *throws* keyword in the method header:

```
public void myMethod() throws IOException
```

The *throws* keyword indicates that *myMethod* might throw an *IOException*. If the method throws multiple exceptions, the exceptions can be listed by separating them by commas after throws:

```
public void myMethod()
  throws Exception1, Exception2, ..., ExceptionN
```

Throwing Exceptions

When a program detects an error, it can create an instance of an appropriate exception type and throw it. This is known as throwing an exception. For example, suppose the program detects that an argument passed to the method violates the method contract; the program can create an instance of *IllegalArgumentException* and throw it, as follows:

```
throw new IllegalArgumentException("Wrong Argument");
```

Catching Exceptions

When an exception is thrown, it can be caught and handled in a *try-catch* block, as follows:

```
try {
statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
    handler for exception1;
  }
catch (Exception2 exVar2) { handler for exception2;
}
...
catch (ExceptionN exVar3) {
    handler for exceptionN;
}
```

If there are no exceptions that arise during the execution of the *try* block, the *catch* blocks are skipped. However, if one of the statements inside the *try* block throws an exception, Java skips the remaining statements in the *try* block and starts finding the code to handle the exception.

The code that handles the exception is called the exception handler. It is found by propagating the exception backwards through a chain of method calls, starting from the current method. Each catch block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the catch block. If so, the exception object is assigned to the variable declared, and the code in the *catch* block is executed. If no handler is found, Java exits this method, passes the exception to the method that invoked the method, and continues the same process to find a handler. If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console. The process of finding a handler is called catching an exception.

Compulsory Task

Follow these steps:

- Create a file called **ArrayIndexOutOfBounds.java**

- Write a program that:
  - Creates an array with 100 randomly chosen integers.
  - Prompts the user to enter the index of the array, then displays the corresponding element value.
  - If the specified index is out of bounds, display the message “Out of Bounds”.

Compulsory Task 2

Follow these steps:

- The following example implements the *hexToDecimal(String hexString)* method, which converts a hex string into a decimal number.

```
import java.util.Scanner;

public class HexToDecimalConversion {

  /** Main method */
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

   // Prompt the user to enter a string
   System.out.print("Enter a hex number: ");
   String hex = input.nextLine();

   System.out.println("The decimal value for hex number " + hex + "
is " + hexToDecimal(hex.toUpperCase()));
  }

  public static int hexToDecimal(String hex) {
    int decimalValue = 0;
    for (int i = 0; i < hex.length(); i++) {
      char hexChar = hex.charAt(i);
```

```
      decimalValue = decimalValue * 16 + hexCharToDecimal(hexChar);
    }

   return decimalValue;
  }

  public static int hexCharToDecimal(char ch) {
    if (ch >= 'A' && ch <= 'F')
      return 10 + ch - 'A';
    else // ch is '0', '1', ..., or '9'
      return ch - '0';
    }
}
```

- Implement the *hexToDecimal* method to throw a *NumberFormatException* if the string is not a hex string.

Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.