Exception Handling

by Sophia

WHAT'S COVERED

In this lesson, you will learn how to catch exceptions using the try and except statements. Specifically, this lesson covers:

1. Catching Exceptions Using try and catch Blocks

A main goal, when writing programs, is to ensure that they run perfectly all of the time. However, there are times in a real-world scenario that a program can stop. It may be due to the program itself. Or, it could be due to something that the person using the program did incorrectly. Handling errors and exceptions is all about anticipating these errors and directing the user on how to fix the issue, or helping the program to end gracefully.

THINK ABOUT IT

Consider the following example of error handling:

```
import java.util.Scanner;

class ScannerException {
  /* This program asks the user to enter his or her age & then
     prints the entry back out. */
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Please enter your age in years: ");
    // Scanner expectes the input to be a valid integer
    int age = input.nextInt();
    System.out.println("You are " + age + " years old.");
  }
}
```

The `Scanner` object (input) expects that the user will enter a value that can be parsed (that is, interpreted) as a valid integer (such as 58, 102, or -1, though this last entry would not make logical sense as an age). A valid integer can't have any non-digit characters or a decimal point.

If the user enters "private" rather than a valid number, the program will end unexpectedly with a message that is not likely to mean much to the user:

```
Please enter your age in years: private
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at ScannerException.main(ScannerException.java:10)
```

This rather extended error message is known as a **"stack trace"** . It is an attempt to provide information about the chain of unfortunate events that occurred when the problem arose. This stack trace indicates that the Scanner ran into problems, and ultimately the problems can be traced back to line 10 in the ScannerException.java program.

HINT

It may be difficult to interpret a message, which is not unexpected.

What is clear is that an exception is involved. An **exception** is an erroneous or anomalous condition that arises while a program is running. In this case, the exception is external to the logic of the program, an invalid user input. In other cases, logical errors in the code itself, such as division by 0 or attempting to write to a file that doesn't exist, may lead to an exception being thrown. When a problem like this one arises during execution of the program, it is described that an exception has been thrown. Ideally, a program should be written so that it can catch exceptions that are thrown and do something to correct or lessen the problem.

Java uses blocks of code called **try and catch statements** to deal with problems at runtime. Code that may run into a problem and throw an exception is placed in a `try` block. The code that can react to the error and do something about it is in a `catch` block that comes right after the try block. When an exception is thrown, the program's control flow, the sequence of statements executed, is redirected to the relevant `catch` block.

Given that a call to the Scanner's `nextInt()` method can throw an exception, this piece of the code needs to go in a `try` block.

A `try` block starts with the keyword try and is followed by curly brackets that mark the block of code where the exception may be thrown:

EXAMPLE

```
try {
 int age = input.nextInt();
}
```

Since the `try` block can complicate access to the variable age, if it is declared in the block, it is not uncommon to declare such a variable before the start of the `try` block and then assign it the needed value in the body of the `try` block.

The following demonstrates complicating access to the variable `age`:

EXAMPLE

```
// Declare & initialize variable
int age = 0;
try {
 // Get input value using method that may throw an exception
 age = input.nextInt();
}
```

This code is incomplete, though, because for every `try` block, there needs to be at least one `catch` block that is invoked if an exception is thrown. There can be multiple `catch` blocks corresponding to a single `try` block. Each `catch` block responds to a different type of exception. For now, though, we will use just a single `catch` block that catches the most general type of exception, which is of the type `Exception`.

The complete design looks like this:

```
import java.util.Scanner;

class ScannerException {
  /* This program asks the user to enter his or her age & then
     prints the entry back out. */
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Please enter your age in years: ");
    int age = 0;
    try {
      age = input.nextInt();
    }
    catch(Exception ex) {
      System.out.println("You did not enter a valid integer.");
      System.out.println("Please run again & enter a valid age.");
    }
    System.out.println("You are " + age + " years old.");
  }
}
```

Now when the user makes an invalid entry, the result looks like this:

```
Please enter your age in years: private
You did not enter a valid integer.
Please run again & enter a valid age.
You are 0 years old.
```

The invalid user input throws an exception, so the `catch` block is executed (and no value is assigned to age, so it is still set to the initial value, 0). After the `catch` block runs, the program control flow continues with the statement after the end of the `catch` block. If we want to keep this output from appearing, it could be added to the `try` block.

Even though the `System.out.println()` won't throw an exception, putting it in the `try`block after the call to `nextInt()` means that it will only execute if the `Scanner` reads the user's input successfully:

```
import java.util.Scanner;

class ScannerException {
  /* This program asks the user to enter his or her age & then
     prints the entry back out. */
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Please enter your age in years: ");
    int age = 0;
    try {
      age = input.nextInt();
      // Next line only executes if call to nextInt() works correctly
      System.out.println("You are " + age + " years old.");
    }
    catch(Exception ex) {
      System.out.println("You did not enter a valid integer.");
      System.out.println("Please run again & enter a valid age.");
    }
  }
}
```

The output should look like this:

```
Please enter your age in years: private
```

```
You did not enter a valid integer.
Please run again & enter a valid age.
```

THINK ABOUT IT

To break this down, Java starts by executing the sequence of statements in the `try` block. If all goes well, it skips the `catch` block and proceeds with the code after the `catch` block. If an exception occurs in the `try` block, Java jumps out of the `try` block and executes the sequence of statements in the `catch` block.

KEY CONCEPT

It's important to note that the catching of the exception, as we have done here, hides all of the errors—even those that may not be expected. Although this is useful for this scenario, it's not an ideal situation, since there may be other errors that could potentially occur. We also can't tell what the error was specifically, either.

TERMS TO KNOW

Stack Trace
A stack trace is a list of Java objects and methods that are involved in an error at runtime and are printed to the screen when an exception occurs.

Exceptions
Errors detected during execution (running of the code) are called exceptions.

try and catch Statements
The idea of the `try` and `catch` statements is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the catch block) are ignored if there is no error.

SUMMARY

In this lesson, you learned about **catching exceptions using try and catch blocks**. You used these statements and associated blocks of code to help the program deal with errors at runtime—or at least to help it fail gracefully, rather than filling the screen with long stack traces that may mean nothing to the user.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from "Python for Everybody" By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Exceptions
Errors detected during execution (running of the code) are called exceptions.

Stack Trace

A stack trace is a list of Java objects and methods that are involved in an error at runtime and are printed to the screen when an exception occurs.

try and catch Statements

The idea of the `try` and `catch` statements is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the catch block) are ignored if there is no error.