Chapter 20 Error Handling and Runtime Exceptions

Java AP

Error Handling

- Although simple programs may not have much potential for causing errors when coded correctly, many programs rely on unknown, unpredictable elements such as user input and files created or modified by users.
- Error handling refers to code written specifically to negate or deal with problems caused by data outside our direct control.
- What sorts of errors could occur with the following code?

Example (pseudo-code)	
	```
	public void printFile(String filename)
	{
	    open file
	    determine file size
	    allocate memory for file
	    read file into memory
	    print file contents
	    close file
	}
	```

Explanation	Nearly every operation in the above pseudo-code has the potential to fail. We need to be able to properly handle the following errors: 1. The file could not be opened. 2. The file could not be read. 3. The size of the file could not be determined. 4. The amount of memory needed could not be allocated. 5. The file could not be closed. If the program fails due to one of these errors, it should do so gracefully and have a way to tell us what went wrong.

Simple error handling with error codes

- The concept of *error codes* refers to the use of values returned by a method to provide basic information about what went wrong, if anything.
- A return value of 0 typically means the method or program was successful, while nonzero values indicate errors of varying kinds.

- For our file printing example, we could define the following error codes:

Return Value		Meaning
0	SUCCESS	No error
-1	ERROR_OPEN	Could not open file
-2	ERROR_READ	Could not read file
-3	ERROR_SIZE	Could not determine file size
-4	ERROR_MEMORY	Could not allocate enough memory
-5	ERROR_CLOSE	Could not close file

- **Note**: Rather than just using the literals 0, -1, etc., we would define public static final members of the class and give them those values while also giving them readable names like ERROR_OPEN.

Example (pseudo-code)

```
public void printFile(String filename)
{
    int error = SUCCESS;
    open file
    if(file is open)
    {
        determine file size
        if(size == -1) // Size could not be determined
        {
            error = ERROR_SIZE;
        }
        else
        {
            allocate memory for file
            if(memory could not be allocated)
            {
                error = ERROR_MEMORY;
            }
            else
            {
                read file into memory
                if(file could not be read)
                {
                    error = ERROR_READ;
                }
                else
                {
                     print file contents
                }
            }
        }
    }
    else
    {
        error = ERROR_OPEN;
    }

    close file
    if(file could not be closed)
    {
        error = ERROR_CLOSE;
    }

    return error;
}
```

Explanation	Our relatively simple file printing method has become much more complex due to the addition of error handling. However, now anyone who uses the method can use its return value to determine when something goes wrong and where the error occurred.

- Advantage of error codes: Easy to implement. Simply change the return type of methods that need error checking to *int* and return specific numbers when errors occur.
- Disadvantages:
    - Can't return more than one thing from a method, so methods that originally returned something else will need to be restructured.
    - No way to force other programmers using the method to actually check the error code and handle it appropriately.
    - Difficult to propagate errors. In other words, to be able to return an error code up through several methods, all of the methods will have to be rewritten to return an error code.

- Below is an example of real code made safer through the use of error codes.

Example

```
public static void main(String[] args)
{
    int input = 0;
    Scanner s = new Scanner(System.in);
    do
    {
        System.out.println("Enter a number (-1 to quit)");
        String next = s.next();
        Input = Integer.parseInt(next);
        System.out.println("Unsafe: " + input);

    }while(input != -1);

    s.close();
}
```

Explanation

Java has a method for converting String objects to integers, **Integer.parseInt()**. However, when given a String that contains non-integral characters, it will crash. To be able to handle this case, we will need to change our code so that it detects when non-integral characters are present in the String and returns an error.

Solution

```
public static int safeStringToInt(String input, Integer
output)
{
    if(output == null)
    {
        return ERROR_INVALID_OUTPUT;
    }

    if(input == null)
    {
        return ERROR_INVALID_STRING;
    }

    for(int i = 0; i < input.length(); ++i)
    {
        if(!Character.isDigit(input.charAt(i))
                && !(i == 0 && input.charAt(i) == '-'))
        {
            return ERROR_INVALID_STRING;
        }
    }
    output = Integer.parseInt(input);

    return SUCCESS;
}

public static void main(String[] args)
{
    int input = 0;
    Scanner s = new Scanner(System.in);
    Integer outputSafe = new Integer("");
    do
    {
        System.out.println("Enter a number (-1 to quit)");
        String next = s.next();

        int error = safeStringToInt(next, outputSafe);
        if(error != SUCCESS)
        {
            if(error == ERROR_INVALID_STRING)
            {
                System.out.println("Invalid Input!");
            }

            if(error == ERROR_INVALID_OUTPUT)
            {
                System.out.println("Output passed to "
                        + "safeStringToInt is invalid");
            }
        }
        else
        {
            input = outputSafe;
            System.out.println("Safe: " + outputSafe);
        }

    }while(input != -1);

    s.close();
}
```

Explanation

Since adding all of our error handling code in the main method would make it extremely long and we may want to use our safe int-to-String conversion elsewhere, we've created a separate method to handle the conversion.

Our method **safeStringToInt()** takes two parameters, the String object we want to convert and an Integer in which to place the output. The necessity of the second argument stems from the fact that we need to be able to return an error code as well as the integer result of the conversion. Since we can't return more than one thing, we need the user to provide an additional modifiable argument for the real output of the method.

Note: Using a regular *int* data type for the output argument would not work since it is passed by copy, not by reference.

The first thing the conversion method does is check to make sure the output and input are valid objects. If either is null (i.e. no memory has been allocated for it), it is unusable and the appropriate error message is returned.

The method then traverses the String and uses the method **Character.isDigit()** to check that each character is a numeral. If a character is not a digit, the String is considered invalid, unless it's the first character (`i == 0`) and is the minus sign (`input.charAt(i) == '-'`). These additional conditions will ensure that String representations of negative numbers are considered valid even though they each have a non-numeric character at the beginning.

If the method hasn't returned by this point, no errors have been found with the input or output and the String can be safely converted to an integer using **Integer.parseInt()**. After the String is parsed, the code representing no errors is returned.

The main method now calls our safe conversion method instead of directly parsing the String and is responsible for providing the proper output object to the method as well as handling the error codes that it returns. Note that error messages are printed when an error is returned. It's good practice to tell the user why their input is incorrect so that they can correct what they're doing wrong.

Runtime Exceptions

- Exceptions are a built-in, extensible feature of Java that can be used to provide more robust error handling.
- An *exception* refers to an error caused by a particularly exceptional or unusual event that typically requires some kind of special handling.
- Exceptions are objects, so they can contain more data than a simple error code can provide.
- Exceptions are handled using a programming construct called a **try/catch** block:

```
try
{
    code that may generate an error
}
catch(exceptionType e)
{
    process error e
}
```

- **Note**: Additional catch blocks may be added to handle multiple exception types.

Example	
	```
	public static void main(String[] args)
	{
	    int input = 0;
	    Scanner s = new Scanner(System.in);
	    do
	    {
	        System.out.println("Enter a number (-1 to quit)");
	        String next = s.next();

	        try
	        {
	            input = Integer.parseInt(next);
	            System.out.println("User input: " + input);
	        }
	        catch(NumberFormatException e)
	        {
	            input = outputSafe;
	            System.out.println("Invalid Input");
	        }

	    }while(input != -1);
	    s.close();
	}
	```

Explanation	As it turns out, **Integer.parseInt()** is written so that it “throws” an exception if we try to give it invalid input. We just needed to know which exception(s) it throws in order to be able to catch and handle them. Our code is much easier to read now that we’re relying on the built-in error checking functionality in the **parseInt()** method. Keep in mind that while most methods in the official Java libraries will properly detect errors and throw exceptions, code from other sources may not, so be ready to write additional error handling code.

Classifying exceptions

- In Java, exceptions fall into one of two major categories – **checked** and **unchecked.**
- **Checked** exceptions occur due to errors that should be easy to anticipate and recover from.
    - Example: A program that reads data from files asks the user to enter a filename. The user enters a nonexistent file, causing Java’s file reader to throw a *FileNotFoundException.* The program should catch this exception, inform the user of the mistake and allow them to correct the filename.
    - In order to a compile the program, checked exceptions must be caught by the programmer or the method that causes the exception must specify that it throws that exception.
- **Unchecked** exceptions occur due to errors that are not generally feasible to recover from.
    - Example: A program tries to read data from a file, but fails due to a hardware or operating system malfunction and throws an *IOError*. The program is not expected to be able to proceed as normal in this circumstance.
    - Example: Due to a programming logic error, a *null* reference is passed to Java’s file reader, causing a *NullPointerException* to be thrown. While the programmer could write code to catch this exception, the better course is to fix the code so that this has no chance of occurring.
    - Handling unchecked exceptions is optional.

- <u>**Note**</u>: Some exceptions that might seem like they should be checked are, in fact, unchecked. If the *NumberFormatException* in the String conversion example was checked, we would have had to catch it in order to compile the program.

Handling checked exceptions – Catch or Specify

Example

```java
public static void printFile(String filename)
{
    File file = new File(filename);
    Scanner s = new Scanner(file);

    while(s.hasNextLine())
    {
        System.out.println(s.nextLine());
    }
}
```

Explanation

This code will not compile due to unhandled exception types. In this case, the constructor of Scanner is the culprit. The constructor used here, which takes a File object as an argument, throws *FileNotFoundException*, which is a checked exception. We can fix this by adding a try/catch block.

Solution 1

```java
public static void printFile(String filename)
{
    File file = new File(filename);

    try
    {
        Scanner s = new Scanner(file);
        while(s.hasNextLine())
        {
            System.out.println(s.nextLine());
        }
    }
    catch(FileNotFoundException e)
    {
        System.out.println(e.getMessage());
    }
}

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

    while(true)
    {
        System.out.println("Enter a filename (quit to "
                + "exit)");
        input = s.next();

        if(input.equals("quit"))
        {
            break;
        }
        else
        {
            System.out.println("Displaying contents of "
                    + input + ":");
            printFile(input);
            System.out.println("");
        }
    }
    s.close();
}
```

Explanation

Now that the code catches the *FileNotFoundException*, it will compile. As of right now, the only line in our catch block is a print statement: Exception objects have messages that contain information about what caused the exception to be thrown. While printing the message doesn't solve the problem, it at least lets the user of the program know that an error occurred, altering the program's normal sequence of operations.

Having caught the exception in this method, we can now use it elsewhere without having to do any error checking and we don't have to return any extra information from the method.

Solution 2

```java
public static void printFile(String filename)
throws FileNotFoundException
{
    File file = new File(filename);
    Scanner s = new Scanner(file);

    while(s.hasNextLine())
    {
        System.out.println(s.nextLine());
    }
}

public static void main(String[] args)
{
    String input;
    Scanner s = new Scanner(System.in);
    while(true)
    {
        System.out.println("Enter a filename (quit to "
                + "exit)");
        input = s.next();

        if(input.equals("quit"))
        {
            break;
        }
        else
        {
            try
            {
                System.out.println("Displaying contents of "
                        + input + ":");
                printFile(input);
                System.out.println("");
            }
            catch(FileNotFoundException e)
            {
                System.out.println(e.getMessage());
            }
        }
    }
    s.close();
}
```

Explanation

It is also possible to specify that a method throws an exception if you want to avoid having to handle it directly. However, this means any code that uses the method will have to catch the exception.

Notice that the main method in the above solution is nearly the same as the one from Solution 1, but has been altered to catch the exception that is now thrown by our **printFile()** method. Without this try/catch block, it would not compile.

Throwing exceptions

- To help prevent other programmers from misusing methods that you've written, it's possible to manually throw exceptions.
- The syntax for throwing an exception is as follows:

```
throw new ExceptionClassName("Message text");
```

- Simply replace *ExceptionClassName* with the name of the actual exception class that is appropriate to the situation, and provide a helpful error message.

Example	see code below

```
public static void printFile(String filename)
{
    File file = new File(filename);

    if(filename == null)
    {
        throw new IllegalArgumentException("filename cannot "
                + "be null");
    }

    if(!filename.endsWith(".txt"))
    {
        throw new IllegalArgumentException(filename
                + " is not a .txt file");
    }

    try
    {
        Scanner s = new Scanner(file);
        while(s.hasNextLine())
        {
            System.out.println(s.nextLine());
        }
    }
    catch(FileNotFoundException e)
    {
        System.out.println(e.getMessage());
    }
}

public static void main(String[] args)
{
    try
    {
        printFile("Main.java");
    }
    catch(IllegalArgumentException e)
    {
        System.out.println(e.getMessage());
    }

    try
    {
        printFile(null);
    }
    catch(IllegalArgumentException e)
    {
        System.out.println(e.getMessage());
    }
}
```

Output

```
Main.java is not a .txt file.
filename cannot be null
```

Explanation

To prevent users of our **printFile()** method from providing files that we can't print, we've added additional error handling to the method. If the filename String passed to the method is *null* or has the wrong file extension, an *IllegalArgumentException* is thrown.

IllegalArgumentException is an unchecked exception, so throwing it won't force users to catch it, but it will at least provide information about why our method failed if they do.

Note: If we had specified that printFile() throws a *FileNotFoundException*, forcing us to handle in the main method, we would have needed multiple *catch* blocks for each *try*:

```
    try
    {
        printFile("Main.java");
    }
    catch(IllegalArgumentException e)
    {
        System.out.println(e.getMessage());
    }
    catch(FileNotFoundException e)
    {
        System.out.println(e.getMessage());
    }
```

This is completely legal and often necessary since many methods have the

	potential to throw one of several different kinds of exceptions. If you have two or more catches for similar exceptions of varying levels of specificity (one is a subclass of the other), the most specific must come first.

Custom exception classes

- While Java has a wide array of built-in exception classes, there may be instances in which you want to handle an error not specifically covered by any of the existing classes. In these cases, you can create your own exception class that inherits from one of Java's more generic exception classes.
- Example: Suppose we have a GameObject class which has an Initialize() method that is expected to be called before the object can be used. What should happen when Initialize() is not called first?
    - We could write an exception class for this and throw the exception when a GameObject is used without first calling Initialize().
    - Since this is an exception that would likely be caused by logic errors in the programmer's code, it makes sense for it to be a subclass of an unchecked exception. One of the most basic of these is *RuntimeException.*

Example	
	```
	public class UninitializedGameObjectException extends RuntimeException
	{
	   // It is standard practice to give a default constructor
	   public UninitializedGameObjectException()
	   {
	   }

	   public  UninitializedGameObjectException(String message)
	   {
	      super(message);
	   }
	}
	```

Explanation	The name of our new exception is UnitializedGameObjectException. While adding the word Exception at the end of the class name for an exception may seem redundant, it is considered good practice to do so. This makes it completely clear that the class is an Exception without having to look up any details on it. In the constructor that takes a message, we can accept whatever information is given when the exception is created and augment it with anything else we think might be helpful to know for the person receiving the error.

- **Note:** There are many exception classes already included in the standard Java libraries. Be sure the error you want to provide an exception for is not already covered before creating your own class. It may be more efficient to simply provide a very specific error message when throwing an existing exception.

The finally keyword

Example (pseudo-code)	
	```
	try
	{
	    code that may generate an error
	}
	catch(exceptionType e)
	{
	    process error e
	}
	finally
	{
	    code that always executes even if no exception is found
	}
	```

Explanation	The **finally** portion of a try/catch statement is an optional addition that is often used to clean up resources that may have been allocated during the *try* block. This could include closing opened files and other similar tasks. Any code in the *finally* block is guaranteed to run after *try* exits regardless of whether an exception occurs. If an exception occurs that is not handled by any of the *catch* blocks, the code in *finally* will be executed before the exception is passed back to the method that called the offending code.

Additional information

- More information on the different classifications of exceptions: http://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html
- Documentation on the *Exception* base class and its subclasses: http://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html