Contents

0 Introduction

The next major component of the Java language will allow us to write programs that handle errors gracefully. Java provides a mechanism called *exception handling* that provides a parallel track of return from functions so that you can avoid cluttering the ordinary execution of code with endless error-handling routines.

Exceptions are objects that are "thrown" by various methods or actions. In this chapter we will learn how to handle (catch) an exception. By so doing we allow our program to recover and continue to work. Failure to catch and exception results in a flood of nasty red text from Java (a so-called "exploding heart"). Crashes such as these should be extremely rare in production-quality software. We can use exceptions as a means to protect our program from such dangers as user abuse and from such misfortunes as crashing whilst attempting

to gain access to an nonexistent or prohibited resource. Many of these hazards are beyond both user and programmer control.

When you program with files or with socket connections, the handling of exceptions will be mandatory; hence the need for this chapter before we begin handling files.

1 The `Throwable` Subtree

Go to the Java API guide and pull up the class `Exception`. The family tree is is as follows.

```
java.lang.Object
java.lang.Throwable
java.lang.Exception
```

The class name `Throwable` is a bit strange; one would initially think it were an interface. It is, however, a class. The class `java.lang.Exception` has a sibling class `java.lang.Error`.

When objects of type `Error` are thrown, it is not reasonable to try to recover. These things come from problems in the Java Virtual Machine, bad memory problems, or problems from the underlying OS. We just accept the fact that they cause program death. Continuing to proceed would just lead to a chain of ever-escalating problems.

Objects of type `Exception` are thrown for more minor problems, such as an attempt to open a non-existent file for reading, trying to convert an unparseable string to an integer, or trying to access an entry of a string, array or array list that is out of bounds.

Let us show this mechanism at work. For example, if you attempt to execute the code

```
int foo = Integer.parseInt("gobbledegook");
```

you will be rewarded with an exception. To see what happens, create this program `MakeException.java`.

```
public class MakeException
{
    public static void main(String[] args)
    {
        int foo = Integer.parseInt("gobbledegook");
    }
}
```

This program compiles happily. You will see that the infraction we have here is a run-time error, as is any exception.

When you run the program you will see this in the interactions pane.

```
java.lang.NumberFormatException: For input string: "gobbledegook"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:492)
    at java.lang.Integer.parseInt(Integer.java:527)
    at MakeException.main(MakeException.java:5)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:601)
    at edu.rice.cs.drjava.model.compiler.JavacCompiler.runCommand(JavacCompiler.java:272)
```

The exploding heart you see here shows a *stack trace*. This shows how the exception propagates through the various calls the program makes. To learn what you did wrong, you must look in this list for your file. You will see the offending line here.

```
at MakeException.main(MakeException.java:5)
```

You are being told that the scene of the crime is on line 5, smack in the middle of your `main` method. The stack trace can yield valuable clues in tracking down and extirpating a run-time problem such as this one.

We have seen reference to "throw" and "throws" before. Go into the API guide and bring up the class `String`. Now scroll down to the method summary and click on the link for the familiar method `charAt()`. You will see this notation.

Throws:

> `IndexOutOfBoundsException` - if the `index` argument is negative or not less than the length of the string.

Let us now look up this `IndexOutOfBoundsException`. The family tree reveals that this class extends the `RuntimeException` class. The purpose of this exception is to create an opportunity to gracefully get out of an exceptional situation and to avoid having your program simply crash. Incidentally, this an an opportunity we will not always avail ourselves of, especially when it arises because of a programmer error.

2 Checked and Run-Time Exceptions

There are two types of exceptions that exist: `RuntimeExceptions` and all others, which are called *checked exceptions*. Generally a run-time exception is caused by *programmer* error. Programmers should know better, and it is probably good for them to have their programs using your class die with an exploding heart as a reward for writing rotten code. Such programmers can read the stack trace and lick their wounds. How do you know if an exception is a `RuntimeException`? Just look up its family tree and see if it is a descendant of `RuntimeException`. So far in our study of Java, we have only seen runtime exceptions.

Checked exceptions, on the other hand, are usually caused by situations beyond programmer control. Suppose a user tries to get a program to open a file that does not exist, or a file for which he lacks appropriate permissions. Another similar situation is that of attempting to create a *socket*, or a connection to another computer. That computer may disallow such connections, it could be down, or it could even be nonexistent. These situations are not necessarily the user's or programmer's fault.

Checked exceptions must be *handled*; this process entails creating code to tell your program what to do in the face of these exceptions being thrown. It is entirely optional to handle a runtime exception.

Sometimes a runtime exception will be caused by user error; in these cases it is appropriate to use exception handling to fix the problem. For example if a user is supposed to enter a number into a `JOptionPane` dialog and enters a string that is not numeric, your program might try to use `Integer.parseInt` to convert it into an integer. Here we see a problem created by an end-user. This user should be protected and this error should be handled gracefully so that (bumbling) user can go about his business. You always want to protect the end-user from exceptions if it is at all feasible or reasonable.

2.1 Catching It

Java provides a parallel track of execution for handling exceptions gracefully. Suppose you are writing a program that displays a color in response to a hex code entered by a (very dumb) end-user of Your Shining Program. The user enters something like `ffgg00`; this is a situation that you, the programmer do not control. The Java exception mechanism would allow you to cleanly punt and reset your color calculator to some state, such as white, and display the appropriate hex code, `0xFFFFFF`.

Some resourceful hackish readers might think, "Here is a new and useful way to get unwedged from a bad situation." This is a mistake. Only use exception handling for error situations beyond programmer control. Do not use them for the ordinary execution of your programs.

3 A Simple Case Study

Let us write a simple color calculator. Our application is to have three graphical elements. It will have a color panel to display the color sample which will occupy most of the frame. On the top of the frame we will place a `JButton` and a `JTextField`. The user types into the `JTextField` and hits enter or hits the button and the color is shown.

We shall immediately bring on our existing `ColorPanel` class and recycle it shamelessly. This should give you the idea that you want to design plenty of reusable classes that can be helpful in a variety of situations.

```
import javax.swing.JPanel;
import java.awt.Color;
import java.awt.Graphics;
public class ColorPanel extends JPanel
{
    private Color color;
    public ColorPanel()
    {
        super();
        color = Color.white;
    }
    public void setColor(Color _color)
    {
        color = _color;
    }
    public Color getColor()
    {
        return color;
    }
    public void paintComponent(Graphics g)
    {
        g.setColor(color);
        g.fillRect(0,0,getWidth(), getHeight());
    }
}
```

Now let us begin by building the frame. We block in the three graphical elements as state variables

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import java.awt.Color;
```

```
import java.awt.Container;
public class SimpleColorCalc extends JFrame implements Runnable
{
    final ColorPanel cp;
    final JButton show;
    final JTextField hexCode;

    public SimpleColorCalc()
    {
        super("Simple Color Calculator");
        cp = new ColorPanel();
    }
    public static void main(String[] args)
    {
        SimpleColorCalc scc = new SimpleColorCalc();
        javax.swing.SwingUtilities.invokeLater(scc);
    }
    public void run()
    {
        setSize(400,300);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container c = getContentPane();
        setVisible(true);
    }

}
```

Now we shall write the balance of the constructor During this process, we establish the basic properties of the graphical widgets. The color panel will be white. The text field will show the hex code for white, `0xffffff`. The `JButton` gets created and labeled. We also set fonts for the button and the text field to enhance the appearance of the app.

```
public SimpleColorCalc()
{
    super("Simple Color Calculator");
    cp = new ColorPanel();
    show = new JButton("Show the Color");
    hexCode = new JTextField("0xffffff");
    hexCode.setFont(new Font("Monospaced", Font.BOLD, 12));
    hexCode.setHorizontalAlignment(JTextField.RIGHT);
    show.setFont(new Font("Monospaced", Font.BOLD, 12));
}
```

Enter this code and compile. Add the necessary imports.

Next we add these lines to the `run()` method. Here we create a `JPanel` and pop the button and text field into it. We then add the panel to the top of the frame. We next place the color panel in the center, where it will occupy the biggest space, as we had planned.

```
JPanel top = new JPanel();
c.add(BorderLayout.NORTH, top);
top.setLayout(new GridLayout(1,2));
top.add(show);
top.add(hexCode);
c.add(BorderLayout.CENTER, cp);
```

Make sure you put the `setVisible(true)` call last. Run this code and you will see a button in the upper-left, a text field with `"ffffff"` emblazoned on it in the upper right, and a color panel filled with white.

The next logical step is to make the button and text fields live. When the button is pushed or enter is hit in the text field , we want the following to happen.

1. Get the text from the `JTextField`. It comes in as a string.
2. Turn it into a hex code.
3. Get the color for the hex code
4. Have the color panel paint itself that color.

To this end, we will create an action listener. Since the text field will issue an action event when enter is hit in it, we can use the same listener class for the text field and the button.

Begin with a shell we place inside of our class.

```
class ColorChangeListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
    }
}
```

Go to the API guide and look up `JTextField`. Look in the constructor summary and see the constructor we used. Now do a search for `getText()`. This is a method inherited from the parent class `javax.swing.JTextComponent`. It returns the text, as a string, residing in the `JTextField`. Let us now test this using our action listener. Make sure you do the includes for the action listener and the action event.

```
class ColorChangeListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String text = hexCode.getText();
        System.out.println(text);
    }
}
```

Compile and run and you should see that when you click on the button, the text in the `JTextField` is put to `stdout`. Thusfar, we have waded in and learned about a new widget. How about an exception here?

What is beyond programmer control? The user just might do something stupid like enter `cowpie` in the `JTextField`. As it stands now, our program will happily put that to stdout. However, we want to convert this value to a hex code. To perform the conversion, we use the `Integer` wrapper class. Bring it up in your API guide. Now find the method `parseInt(String s, int radix)` and click on its link. The word *radix* is just another word for number base. Since we are trafficking in hex codes here, we shall use base 16. You can also see that it throws a `NumberFormatException`. Go to the API page for this exception; you can do so by clicking on the link shown.

This is a runtime exception, as you can see by looking up the family tree. However, it is triggered by an end-user's blunder, so we shall deal with it gracefully. We will just reset our color calculator to its original white.

The tasks confronting us here are: get the hex code, convert it into a hex number, but if an illegal value is entered, reset everything to white. First here is the naked call to the static method `Integer.parseInt()`. We know this method is dangerous: it throws an exception.

```
class ColorChangeListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String text = hexCode.getText();
        int colorcode = Integer.parseInt(text, 16);
    }
}
```

Imagine you are about to get into a tub full of water. Do you just plunge in? This is not the recommended course of action if you desire continued and comfortable existence. You first dip a finger or toe in the water. If it's too hot or too cold, you throw a `BadTubwaterTemperatureException`. You can recover from such an error. If the tub is too cold, add hot water until the desired

temperature is reached. If the tub is too hot, let it cool or add cold water until the water is at a suitable temperature.

Java provides a mechanism called the the `try-catch` sequence to handle the exception. We now insert this and explain it.

```
class ColorChangeListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String text = hexCode.getText();
        try
        {
            int colorcode = Integer.parseInt(text, 16);
        }
        catch(NumberFormatException ex)
        {
            hexCode.setText("ffffff");
            cp.setColor(Color.white);
            repaint();
        }
    }
}
```

You place the "dangerous code" you are using inside of the `try` block. In this case, the danger is generated by a number format exception triggered by the abuse of our innocent program. If the user enters a legal hex code, the `catch` block is ignored; if not, the `catch` block executes. This precludes the occurrence of an exploding heart caused by an end–user abusing `Integer.parseInt()`. Once we have circumnavigated the danger, we can go about our business of obtaining a color and coloring the panel. Notice that all of the code using the integer variable `colorCode` is placed in the `try` block. This is because an exception will cause the `try` block to abort immediately, and the variable you wish to use will never be created during an error state.

```
class ColorChangeListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String text = hexCode.getText();
        try
        {
            int colorCode = Integer.parseInt(text, 16);
            Color enteredColor = new Color(colorCode);
            cp.setColor(enteredColor);
            repaint();
```

```
        }
        catch(NumberFormatException ex)
        {
            hexCode.setText("ffffff");
            cp.setColor(Color.white);
            repaint();
        }
    }
}
```

Psssssst.... Confidentially..... Duplicate code! We see that `repaint()` should occur regardless of whether an exception is thrown or not. The `try–catch` apparatus has one more item, the `finally` block. This block is carried out whether an exception occurs or not. Here we see it at work.

```
class ColorChangeListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String text = hexCode.getText();
        try
        {
            int colorCode = Integer.parseInt(text, 16);
            Color enteredColor = new Color(colorCode);
            cp.setColor(enteredColor);
        }
        catch(NumberFormatException ex)
        {
            hexCode.setText("ffffff");
            cp.setColor(Color.white);
        }
        finally
        {
            repaint();
        }
    }
}
```

Now finish by attaching this listener to the button and text field. Go into the run method and add these lines just before the `setVisible` line.

```
show.addActionListener(new ColorChangeListener());
hexCode.addActionListener(new ColorChangeListener());
```

This is a ready–for–prime–time program that functions robustly. Exception handling gives it fault-tolerance that makes it deployable in a realistic situation.

4 All Code Shown

Here is the complete program `SimpleColorCalc.java`. Make sure you have the `ColorPanel` in the same directory when compiling.

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JButton;
import java.awt.Color;
import java.awt.Font;
import java.awt.Container;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class SimpleColorCalc extends JFrame implements Runnable
{
    final ColorPanel cp;
    final JButton show;
    final JTextField hexCode;
    public SimpleColorCalc()
    {
        super("Simple Color Calculator");
        cp = new ColorPanel();
        show = new JButton("Show the Color");
        hexCode = new JTextField("0xffffff");
        hexCode.setFont(new Font("Monospaced", Font.BOLD, 12));
        hexCode.setHorizontalAlignment(JTextField.RIGHT);
        show.setFont(new Font("Monospaced", Font.BOLD, 12));
    }
    public static void main(String[] args)
    {
        SimpleColorCalc scc = new SimpleColorCalc();
        javax.swing.SwingUtilities.invokeLater(scc);
    }
    private static String presentHexCode(String s)
    {
        s = s.toLowerCase();
        if(! s.substring(0,2).equals("0x"))
```

```
            s = "0x" + s;
        return s;
    }
    public void run()
    {
        setSize(400,300);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container c = getContentPane();
        //Make a space for the text field and button.
        JPanel top = new JPanel();
        top.setLayout(new GridLayout(1,2));
        top.add(show);
        top.add(hexCode);
        //add the top and the color panel to the content pane
        c.add(BorderLayout.NORTH, top);
        c.add(BorderLayout.CENTER, cp);
        show.addActionListener(new ColorChangeListener());
        hexCode.addActionListener(new ColorChangeListener());
        setVisible(true);
    }
    class ColorChangeListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            String text = hexCode.getText();
            try
            {
                int colorCode = Integer.parseInt(text, 16);
                Color enteredColor = new Color(colorCode);
                hexCode.setText(presentHexCode(text));
                cp.setColor(enteredColor);
            }
            catch(NumberFormatException ex)
            {
                hexCode.setText("0xffffff");
                cp.setColor(Color.white);
            }
            finally
            {
                repaint();
            }
        }
    }
}
```

5 Exception Handling, In General

Now that we have seen a simple case study, we can begin to understand the whole business of handling exceptions. You will have already noticed some things.

Exceptions are objects; they are instances of classes. The class `Throwable` is the root class. The class `Throwable` has two children, `Exception` and `Error`. As we said before, we shall concentrate on exceptions here, so you may actually treat `Exception` as the root class.

5.1 Can you have several `catch` blocks?

The answer to this is yes. A `try` block must be followed immediately by at least one `catch` block. A `finally` block is an optional block that occurs after a `try` block and one or more `catch` blocks; this block executes whether or not an exception is thrown. Do not place other code between the succeeding `try`, `catch` and `finally` blocks. Each `catch` block can catch a different type of exception.

The general syntax looks like this.

```
try
   {
   //some dangerous code that throws various exceptions
   }
catch(FairlySpecificException e)
   {
   //code to handle a fairly specific exception.
   }
catch(LessSpecificExceptoin e)
   {
   //code to handle a less specific exception
   }
.
.
.
catch(LeastSpecificException e)
   {
   //code to handle the most general possible exception
   }
finally
   {
   //do this no matter what
   }
```

5.2 The Bucket Principle

It is possible that code in a `try` block might throw several types of exceptions. Think of the catch blocks as being buckets and the exceptions as "dropping out" from your code. The most general type of exception is `Exception`; if you do the following, you will be in for a surprise.

```
try
   {
   //code that throws a NumberFormatException
   //and some other exceptions
   }
catch(Exception e)
   {
   //handle an exception
   }
catch(NumberFormatExcepton e)
   {
   //handle a number format exception
   }
```

The second catch block is dead code! Think of the catch blocks as being buckets for catching exceptions. A catch block with a more general type of exception is a bigger bucket. Once a bucket catches the exception, it is handled and execution skips to the `finally` block if one exists. If not, you continue execution at the end of the `try`–`catch` progression. Were a number format exception to be thrown in our code here, the big bucket at the top, the `Exception` bucket, would catch any number format exception. The moral of this tale is: Place more specialized exceptions in earlier catch blocks and the more general ones at the bottom. If two types of exception are independent, they represent independent buckets. An example of such exceptions is `NumberFormatException` and `FileNotFoundException`. Catch blocks for these would represent independent, nonoverlapping buckets. Look in the API guide to see that they are unrelated, and their lowest common ancestor is the root exception class `Exception`.

Reasoning by analogy is dangerous, but we plunge ahead insouciantly nonetheless. The behavior if `try`–`catch` progression is much like an `if--else if--else` progression in that the first active block executes. The analogy, however, is incomplete because the `finally` block executes in any event, unlike the `else` block.

Arrange your buckets so you get the desired action in recovering from your error. Also engineer your buckets to make them leak–proof: remember, an uncaught exception will crash your program. You should aim to catch as narrowly as possible.

Check the methods you are using in the `try` block for the types of exceptions they can throw. Then aim to catch just these.

6 Mr. Truman, We Must Pass the Buck!

Go into the API guide to `Integer` and get the method detail for `parseInt()`. Here is its method header.

```
public static int parseInt(String s) throws NumberFormatException
```

You see a new keyword, `throws`. This declaration in the function header is a tocsin to the client programmer: *Beware, this method can generate, or throw, a* `NumberFormatException`. The creator of this method is "passing the buck" and forcing the client to handle this exception. The penalty for failing to do so is the possibility of an uncaught exception and ugly program death.

6.1 Must I?

We have said that, when an exception is caused by programmer error, you probably should not catch it, unless there is a compelling reason of cost or operational practicality. Handling runtime exceptions is *optional*. Handling checked exceptions is mandatory, unless you pass the buck by using the `throws` keyword in the method header. In this case, you force the caller to handle the exception.

Let us look at such an example. Go into the API guide and bring up the `FileReader` class. The method detail for the first constructor reads as follows.

```
public FileReader(String fileName) throws FileNotFoundException
```

Click on the link for `FileNotFoundException`. Since `RuntimeException` is not in the family tree of `FileNotFoundException`, we see that `FileNotFoundException` is a checked exception. This would be appropriate since there is no programmer control over the file system where the program is being used. You will see plenty of examples of checked exception in the next chapter on fileIO.

Reading the preamble, we see this exception is thrown if you attempt to open a nonexistent file for reading or you try to open a file for which you do not have read permission. If you are going to instantiate a `FileReader` in a class method, you have two choices. You can handle the exception in the method, or you can add `throws FileNotFoundException` to the method header. Ultimately, any client code invoking this method must either handle the exception or pass the buck.

Here is how we handle it.

```
public void processFile(String filename)
{
    try
    {
        FileReader f = new FileReader(fileName);
        //code that processes the file
    }
    catch(FileNotFoundException e)
    {
        //code to bail out of the wild goose chase
    }
}
```

Here is how to pass the buck.

```
public void processFile(String filename) throws FileNotFoundException
{
    FileReader f = new FileReader(fileName);
    //code that processes the file
}
```

Passing the buck forces the caller to handle the exception. You should also note that you must add the import statement

```
import java.io.FileNotFoundException
```

to use a `FileNotFoundException` because it is not part of the `java.lang` pacakage.

7 Can I Throw an Exception?

In short, the answer is "yes." Let us show an example. Suppose you are writing a fraction class.

```
public class Fraction
{
    int num;
    int denom;
    public Fraction(int _num, int _denom)
    {
        num = _num;
```

```
        denom = _denom;
        //more code
    }
    public Fraction()
    {
        this(0,1);
    }
    //more code
}
```

Your program should get annoyed if some foolish client does something like this.

```
Fraction f = new Fraction(3,0);
```

You do not want people abusing your code by programming with zero-denonomiator functions. Modify your code as follows to punish the offender. We shall throw an `IllegalArgumentException`.

```
public class Fraction
{
    int num;
    int denom;
    public Fraction(int _num, int _denom)
        throws IllegalArgumentException
    {
        if(_denom == 0)
             throw new IllegalArgumentException("Zero Denominator");
        num = _num;
        denom = _denom;
        //more code
    }
    public Fraction()
    {
        this(0,1);
    }
```

The `IllegalArgumentException` is a runtime exception (check its family tree), so it is optional for the caller to check it. You can leave the issue to the caller's judgment. As soon as the exception is triggered, the execution of the constructor is called to a halt. The exception is thrown to the caller; if the caller does not handle it, the caller is rewarded with an exploding heart.

7.1 Can I make my own exceptions?

Yes, all you need do is extend an existing exception class. For example we might want to have a special exception for our `Fraction` class. We create it as follows.

```
public class ZeroDenominatorException extends RuntimeException
{
}
```

We extended the `RuntimeException`, so our exception is a runtime exception. You may now throw `ZeroDenominatorExceptions`. You could choose to extend `IllegalArgumentException`, since this exception results from passing illegal arguments to a constructor.

However, you should strive to use standard exceptions wherever possible. The use of the `IllegalArgumentException` in this constructor was probably the best call.

8 Summary

Java provides an exception handling apparatus that allows you to handle various common error states without cluttering up the main line of execution of your code by error handling routines. You can create new types of exceptions by extending existing ones. Exceptions propagate through the stack via the buck–passing mechanism. If they are unhandled the the program crashes. You should not use exceptions for the ordinary course of your code as an alternative branching mechanism. This is an abuse of exception handling.

In the next chapter on File IO, you will see we make frequent use of exception handling.