An Introduction to Computer Science with Java

Community College of Philadelphia Computer Science

Introduction to
Computer Science with Java

Chapter 12 – Java Exceptions

This short chapter discusses exceptions and exception handling in Java.

Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

- Describe what an exception is in Java programming, and three things programmers do to address the possibility of exceptions in their code.
- Describe the three different kinds of exceptions listed in the JLS, and the difference between checked and unchecked exceptions.
- Describe how to throw an exception, how to catch an exception, and why novice programmers tend to throw exceptions while more experienced programmers tend to catch exceptions.
- Write java code that throws exceptions, and code that uses try-catch blocks to handle exceptions.

11.1 What is an Exception?

An **exception** is an event generated by the Java Virtual Machine "*when a program violates the semantic constraints of the Java programming language.*"[1] This means that the JVM will generate an exception whenever it encounters instructions that it cannot execute, either while preparing to execute code or while executing code.

Generally, exceptions can be thought of as errors that occur when a program runs, as opposed to things like syntax errors in code that a compiler will detect. Exceptions occur when we attempt to execute a program, not when we compile a program, although as you will see, Java compilers do check to see of our code addresses certain classes of exceptions.

Technically exceptions are generated by the JVM itself under some circumstances , and by the JVM in cooperation with the operating system in other circumstances, but the details about how this actually works are beyond the scope of this course. For now, we simply need to know that exceptions are events generated by errors that occur when we try to run a program.

Remember that an event has an *event trigger*, an *event listener*, and an *event handler*. The *event trigger* for an exception is the error condition that occurs, such as trying to divide by zero, or trying to read a file that does not exist. Changes in a system that trigger exceptions are called **exceptional events**.

[1] from the Java Language Specification, Chapter 11, version 7, 2013

The *event listener* for all exceptions is the JVM itself. The *event handler* can be any code that handles the exception, such as code in a *catch* block in our software, which we see later in this chapter.

When an exception is generated, the JVM alerts the current method and sends it an exception object with information about the exception.

A Java method can do one of three things when the JVM detects an exceptional event and sends an exception object back to the method:

- It can **handle the exception**. For this to work, an exception handler must be built into the code. We put exception handlers in methods by using **try-catch-finally blocks**, described later in this chapter.
- It can **throw the exception**. This means that the current method will send the exception back to the method that invoked the current method. The invoking method could then handle the exception itself or throw it farther back to the method that invoked that method. Theoretically, this could keep going all the way back to the operating system, as we did earlier in this course. Throwing exceptions is discussed later in this chapter. Generally, it is better to handle exceptions than to throw them. Novice programmers tend to throw exceptions, while more experienced programmers tend to write co to try to avoid escpetions and to handle then when they do occur.
- A method can simply **ignore the exception**. Most likely, the JVM will simply halt any program that generates an exception which is ignored. Some exceptions, known as checked exceptions, cannot be ignored, as we will see in the next section.

11.2 Checked Exceptions, Run-Time Exceptions and Errors

The Java Language Specification describes three different kinds of exceptions:

- **checked exceptions** – These are exceptions for which a Java compiler will check source code to ensure that methods which could cause the exception address the exception. Methods that contain code which could cause a checked exception must address the exception, either by throwing the exception, or by handling it with a catch block. Otherwise, the code will not compile.

  The classical example of a checked exception is a *FileNotFound* exception. Any method that reads or writes to a data file must somehow address the possibility of a *FileNotFound* exception.

  Checked exceptions are typically events that could occur even if code is well written and the system is working properly, such as a missing file. A file could be missing or inaccessible even if the code was well written and the computer system is working properly.

  Another example of a checked exception is an *SQLexception*, which could happen when a Java program attempts to access a database on a database server. Dealing with checked exceptions may seem like extra work for programmers, but it results in code that more robust – meaning it is less likely to crash – more safe, and more secure. This is one of the reasons Java is so widely used, especially for large, complex, and mission critical software projects.

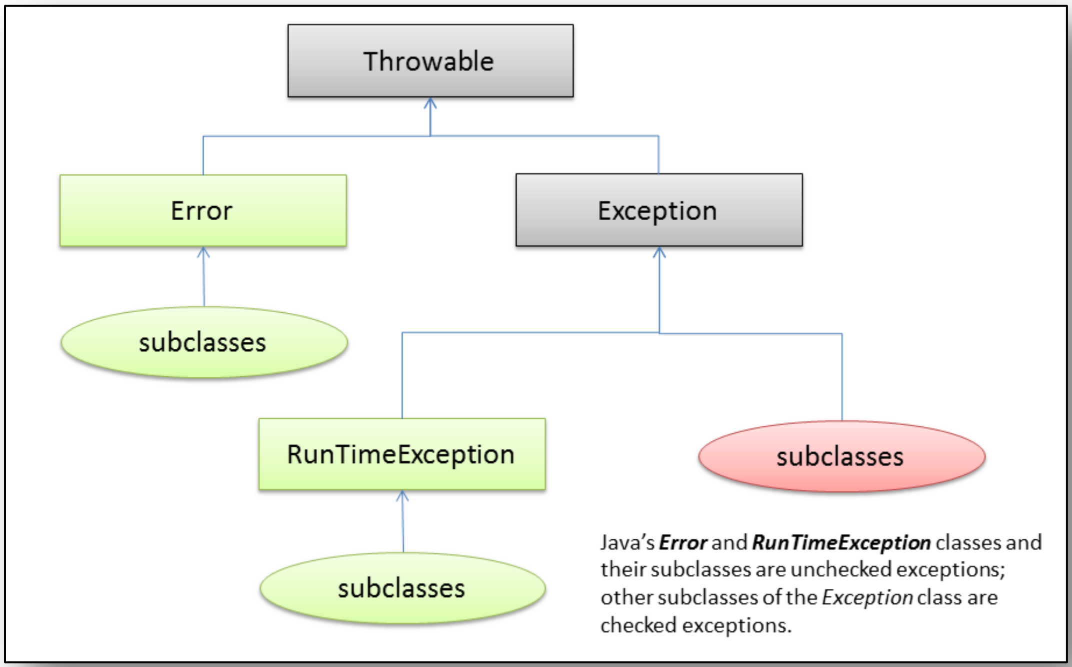
Checked exceptions are usually detected by the JVM before it actually tries to execute code that will not work because of the exception. For example, the JVM will determine that a file does not exist and generate an exception before it tries to read from a non-existent file.

- **run-time exceptions** –these are events that are generated when a program runs, but which can probably be avoided by very careful programing. For example, a *DivisionByZero* exception is a run time exception that could be avoided by careful programming. Java compilers do not check for these exceptions because it would be too cumbersome to do so, such as requiring programmers to account for every possible place in the code where division occurs to avoid *DivisionByZero* exceptions. It is up to programmers to either write their code to avoid these exceptions, or to address the possibility of such exceptions occurring in the same way that checked exceptions are addressed.

- **errors** - In the context of Java exceptions, an error is an external event in the computer system that interferes with the execution of a program, such as and *OutOfMemory* error. It is not the responsibility of the programmer to build code to address these errors, but Java programs could address certain system errors in critical software, in the same way that checked exceptions are addressed.

To summarize, checked exceptions must be addressed in Java methods or the method will not compile. Run-time exceptions and system errors are **unchecked exceptions** that Java methods do not need to address, but methods may address them if the programmer desires to do so.

Java has an extensive set of classes for exceptions and programmers may also create their own classes of exceptions. Exceptions are covered in more detail in CSCI 112. The diagram below shows how they are generally organized with unchecked exceptions in green and checked exceptions in red.


Throwable
Error
Exception
subclasses
RunTimeException
subclasses
subclasses
Java's ***Error*** and ***RunTimeException*** classes and their subclasses are unchecked exceptions; other subclasses of the *Exception* class are checked exceptions.

11.3 Throwing Exceptions

The simplest, and often the least effective, way to handle exceptions is to throw them. To **throw an exception** means that the current method sends the exception object back to the method that invoked the current method. As mentioned earlier in this chapter, the invoking method could then handle the exception itself or throw it farther back to the method that invoked that method, and this could keep going all the way back to the operating system.

There are two ways to throw exceptions: by ***including a throws clause with the method header***, or by ***using a throw statement within a method***. (Throwing exceptions from within a method is covered in CSCI 112.) In either, case, we need to specify the name of the class or classes of exceptions we are throwing.

Here are some examples:

Example: Throwing all Exceptions

```
public static void loadArray(BoardSquare[] square) throws Exception
    {
     … // the rest of the method follows
```

By using the superclass *Exception*, all exceptions will be thrown. This allows code with checked exceptions to compile, but doesn't really address possible problems.

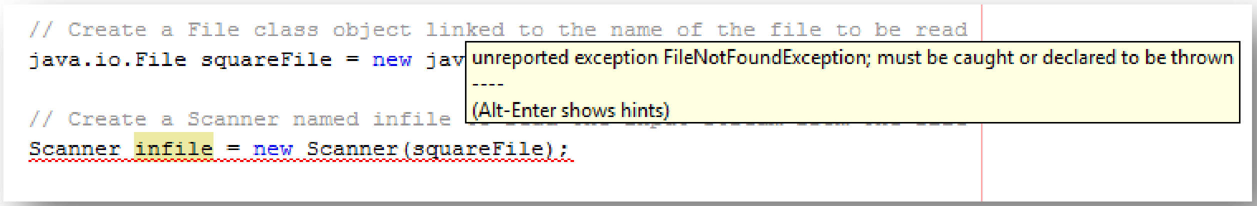
Example: Throwing a Specific Exception

```
public static void loadArray(BoardSquare[] square) throws FileNotFoundException
    {
     … // the rest of the method follows
```

In this case, only a *FileNotFoundException* will be thrown.

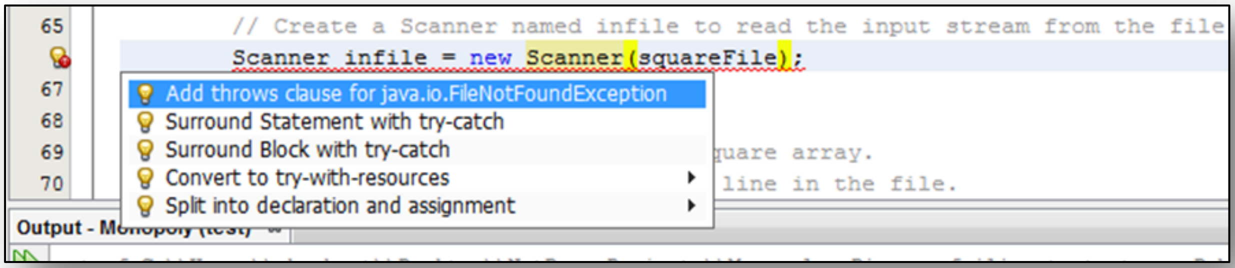
Determining which checked exceptions need to be addressed

How do we know which checked exceptions need to be addressed by our code? The answer is that a good compiler or IDE, such as NetBeans, will tell us. If we do not address checked exceptions in our code, then when we try to compile the code, NetBeans will give us an error message identifying the exception class, as in this error message, generated for a method that attempts to read a file without addressing whether or not the file exists:



```
// Create a File class object linked to the name of the file to be read
java.io.File squareFile = new jav
unreported exception FileNotFoundException; must be caught or declared to be thrown
----
(Alt-Enter shows hints)
// Create a Scanner named infile
Scanner infile = new Scanner(squareFile);
```

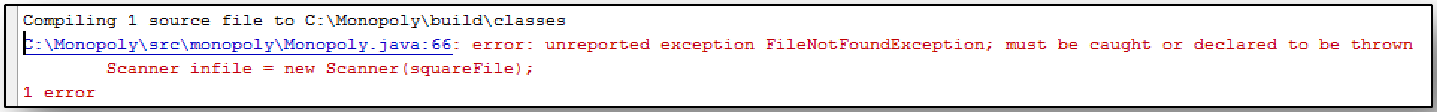
The error message in the code reads "***unreported exception FileNotFoundException; must be caught or declared to be thrown***". Here is what the NetBeans tip looks like for this error.



```
    // Create a Scanner named infile to read the input stream from the file
    Scanner infile = new Scanner(squareFile);
```

Add throws clause for java.io.FileNotFoundException
Surround Statement with try-catch
Surround Block with try-catch
Convert to try-with-resources
Split into declaration and assignment
Output - Monopoly (test)

It lists several suggestions, the first of which is to add a throws clause for the checked exception. Here is what the error message looks like if we attempt to compile the code:



```
Compiling 1 source file to C:\Monopoly\build\classes
C:\Monopoly\src\monopoly\Monopoly.java:66: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
        Scanner infile = new Scanner(squareFile);
1 error
```

The error message says "*FileNotFoundException; must be caught or declared to be thrown*". As we can see, if we fail to address checked exceptions, a good compiler or IDE will indeed tell us what class of checked exception our method must address.

In section 6.5 of chapter 6 we learned how to avoid dealing with file I/O exceptions using the "*Hot Potato*" technique of throwing "*the exception like a hot potato back to the main method, which will in turn throw it back to the operating system.*" This will allow our program to run, but it's not really the best way to handle exceptions. In the next section we will see how to handle exceptions locally within our own code, but for now, any checked exceptions that we don't know how to address can be thrown back to the main() method, then back to the operating system.

Of course, throwing exceptions rather than handling them means our software may be more prone to crash when an exception occurs. Robust, safe, and secure software should attempt to avoid or handle as many exceptions as is reasonably feasible. Again, this will be discussed in more detail in CSCI 112.

11.4 Handling Exceptions with *try*, *catch* and *finally* Blocks

Quoting from the JLS: "*Every exception is represented by an instance of the class Throwable or one of its subclasses. Such an object can be used to carry information from the point at which an exception occurs to the handler that catches it. Handlers are established by catch clauses of try statements.*"

We can handle java exceptions by including ***catch*** blocks associated with ***try*** blocks in our code. The *catch* construct in java is a clause of the *try* statement in the same way that *else* is a clause of an *if* statement; a catch block can only exist following a *try* block.

To **catch** an exception, we enclose the code that might generate the exception in a *try* block, which is immediately followed by a ***catch*** block containing the code to be executed if the exception does occur. If the exception occurs within the try block, then the catch block is executed . If no exception occurs, then the catch block is ignored. Here is an example:

Example: A Try and Catch Block related to File I/O

```
        // Create a File class object linked to the name of the file to be read
        File squareFile = new File( fileNameString);

        try {
        // Create a Scanner named infile to read the input stream from the file
        Scanner infile = new Scanner(squareFile);

        while ( infile.hasNextLine() )
        {
            // read a line and put it in an array element
            lines[count] = infile.nextLine();
            count ++;  // increment the number of array elements with data
        } // end while

        infile.close();

        } catch (FileNotFoundException e) { // e is the instance of the exception
                 System.out.println(("File not found error for file:" + fileNameString
)        } // end catch
```

Notice that all of the code to access the file – from the declaration of the File class object *squarefile* to the statement that closes the file – are enclosed in the try block. if any of these instructions cannot access the file, an exception will be generated. Also, remember the scope of local variables – they only exist in the block of code in which they are enclosed. The variable *infile,* which refers to the I/O stream Scanner for the file, is declared in the *try* block, so it only exists and can only be used within the *try* block.

The JVM does not actually attempt to locate the file until the I/O stream Scanner is established. That is why the declaration of the File variable *squareFile* does not need to be in the try block, but the declaration of the Scanner variable does.

A catch block contains the code that is the exception handler. In this case, it simply prints an error message with the file name.

However, we could do more. when an exception occurs , an instance of an exception object class will automatically be generated by the JVM and passed back to the method; it is a parameter of the catch block. One of the properties of an exception object is an error message, name *message*. We could simply print that error message using the *getMessage()* method, as in the following example:

```
        try {
          // code the same as in the above example goes here.
        } catch (FileNotFoundException e)  // e is the instance of the exception
        {
         System.err.println("FileNotFoundException: " + e.getMessage());
        } // end catch
```

We could also write an exception handler that would log the error and trace the system stack for the error, but these things are beyond the scope of the chapter and will be discussed in CSCI 112.

The two examples above both show local try-catch blocks in the method with the code that could cause an exception. We could also let the method throw the exception, then handle the exception in the invoking method . We do this by enclosing the code that invokes a method in a try block, followed by a catch block with the method handler. The following code shows an example of this.

The writeLines method write data to a data file

```
public static void main(String[] args) throws Exception
{
        String[] tutorials = new String[100]; // an array to hold a list of tutorials
        int count;      // the number of elements in the that are used

        // read data into tutorials[] line by line and return count
        count = readLines(tutorials);

        // print the array on the screen
        System.out.println("The original array:");
        displayLines(tutorials, count);

        // sort the array
        sortStringArray(tutorials, count);

        // print the array on the screen  line by line
        System.out.println("\nThe sorted array:");
        displayLines(tutorials, count);

        try {

        // write the array to a data file line by line
        writeLines(tutorials, count);

        } catch (IOException e) {

            // error message dialog box with custom title and the error icon
             System.err.println("FileNotFoundException: " + e.getMessage());

        } // end catch

    } // end main()
/**************************************************/
    public static void  writeLines(String[] lines, int count) throws IOException
    {
            // create a File class object and give the file the name tutorials.txt
            java.io.File tut  = new java.io.File("tutorials.txt");

            // Create a PrintWriter text output stream and link it to the file x
            java.io.PrintWriter outfile  = new java.io.PrintWriter(tut);

            for ( int i=0; i < count; i++)
                outfile.println(lines[i]);

            outfile.close();
    }  // end writeLines()
```

The try block encloses the statement that invokes the writeLines() method. It has a catch clause to handle the exception.

The *writLines()* method throws the exception back to the *main()* method.

The code on the page above three different ways to handle exceptions:

1. a **chain of throw clauses** implementing a "hot potato" approach that sends the exception back to the JVM and the operating system. This is an easy approach for novice programmers to implement, but it will cause methods to crash and should generally be avoided by experienced programmers.
2. A **try-catch block around the statement invoking the method** in which the error could occur. For this to work, the exception must be thrown back to the invoking method.
3. **local try and catch blocks** closest to the code that could be the source of an exception .

CSCI 112 has more detailed coverage of exception handling, including logging and tracing exceptions. For now is it enough to know how to use try-catch blocks in your code for checked exceptions.

```
    void main() throws Exception
    {
        // invoke  methodA
        methodA();
    } // end main()

    // *************************************

    void methodA() throws Exception
    {
        //invoke  methodB
        methodB(;
    } // end methodA()

    //*************************************

    void methodB() throws Exception
    {
        // method B accesses a data file

    }  // end methodB()
```

Chained Throws Statements

1. methodB() throws to methodA()
2. methodB() throws to main()
3. main() throws to the JVM and Operating System

```
void main()
    {
        try {

        // invoke  methodA
        methodA();

        } catch (IOException e) {

            // error message
             System.err.println("FileNotFoundException: " + e.getMessage());
        } // end catch

    } // end main()

    // *************************************

    void methodA() throws Exception
    {
        // method A accesses a data file

    }  // end methodA()
```

The Invoking Method has a Handler

4. methodA() throws to main()
5. main() handles the exception

```
void main()
    {
        // invoke  methodA
        methodA();
    } // end main()

    // ***************************************

    void methodA()
    {
        try {

        // method A accesses a data file

        }  catch (IOException e) {

            // error message
             System.err.println("FileNotFoundException: " + e.getMessage());
        } // end catch

    }  // end methodA()
```

The Method has a local Handler

A *catch* block will only catch errors of the type specified in the catch blocks parameter. We could have multiple *catch* blocks following a *try* block, checking for different classes of exceptions, such as:

```
try
   { code here}
catch (FileNotFoundException e)
   { code to handle FileNotFoundException here }
catch (IOException e)
   { code to handle IOException here }
```

Code with multiple *catch* blocks needs to be written so a sub-class is checked before its super class is checked. In the example above, *FileNotFoundException* is a sub-class of *IOException*, so *FileNotFoundException* is checked first. This approach will catch any *IOExceptions* other than just the *FileNotFoundException*, and will allow us to print a specific message for *FileNotFoundExceptions* and a more general message for any other *IOExceptions*. Of course, There could be other differences in the way specific exceptions are handled.

Generally, a program will continue after a *catch* block unless something other than the caught exception causes the program to crash. If we want to stop the program we can put a *return* statement in the *catch* block.

A **finally** block may follow a *catch* block or set of *catch* blocks. The code in the *catch* block will only execute if an exception of the caught type occurs in the *try* block. The code in the *finally* block will always be executed, unless we purposely terminate the program in the *catch* block, or some fatal error halts the program. *Finally* blocks are very useful for cleanup, such as closing open Internet sockets in our program or tuning off connections to databases. These things are beyond the scope of this course, but you should know what a *finally* block is.

Chapter Review

Section 1 describes what an extension is, and the three different things a program can do when it detects an event.

Section 2 discusses checked exceptions, run-time exceptions and errors. It tells us that run-time exceptions and errors are unchecked exceptions.

Section 3 describes throwing exceptions.

Section 4 discusses handling exceptions with *try* and *catch* blocks. Using multiple *catch* blocks and the use of *finally* blocks are briefly mentioned at the end of the section.

Chapter Questions

1. What is an exception?
2. What is the event listener for an exception?
3. What is a checked exception?
4. Which exceptions are unchecked?
5. Apart from looking up a comprehensive list of java Exceptions, how can we tell which checked exceptions a method must address?
6. How can we throw an exception?
7. How can we establish exception handlers on our java Code?
8. What goes in a try block if we are setting up an exception handler?
9. Where can a catch block exist in Java code?
10. How can we print the error message property of an exception object?

Chapter Exercise

Choose any program from this semester that accesses a file but that does not use *try-catch* blocks to handle file not found exceptions. The program may be one you wrote or a sample provided by you instructor. Add a try and catch block to handle the exception. You can decide what the handler does. It could simply print a message – such as in a JOptionPane error message dialog box (see page 6 in chapter 7) – or it could allow the user to enter the correct file name.

In place of a lab report, add comments to the beginning of the project code describing what you did and how it works.