**What Is an Exception?**

The term *exception* is shorthand for the phrase "exceptional event."

**Definition:** An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

## **Exception Hierarchy**

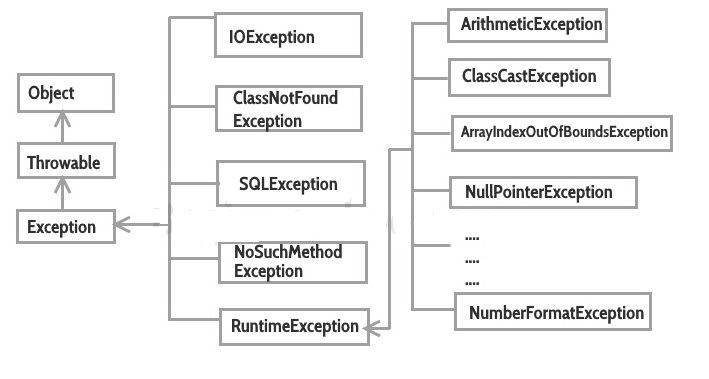
Ultimately, exceptions are just Java objects with all of them extending from Throwable:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ---> Throwable <---                |    (checked)     |                |                  |                |                  |        ---> Exception           Error        |    (checked)        (unchecked)        |  RuntimeException    (unchecked) |

There are three main categories of exceptional conditions:

* Checked exceptions
* Unchecked exceptions / Runtime exceptions
* Errors

**Runtime and unchecked exceptions refer to the same thing. We can often use them interchangeably.**



### 3.1. **Checked Exceptions**

Checked exceptions are exceptions that the Java compiler requires us to handle. We have to either declaratively throw the exception up the call stack, or we have to handle it ourselves. More on both of these in a moment.

[Oracle’s documentation](https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html) tells us to use checked exceptions when we can reasonably expect the caller of our method to be able to recover.

A couple of examples of checked exceptions are IOException and ServletException.

### **3.2. Unchecked Exceptions**

Unchecked exceptions are exceptions that the Java compiler does not require us to handle.

Simply put, if we create an exception that extends RuntimeException, it will be unchecked; otherwise, it will be checked.

And while this sounds convenient, [Oracle’s documentation](https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html) tells us that there are good reasons for both concepts, like differentiating between a situational error (checked) and a usage error (unchecked).

Some examples of unchecked exceptions are NullPointerException, IllegalArgumentException, and SecurityException.

### **3.3. Errors**

Errors represent serious and usually irrecoverable conditions like a library incompatibility, infinite recursion, or memory leaks.

And even though they don’t extend RuntimeException, they are also unchecked.

In most cases, it’d be weird for us to handle, instantiate or extend Errors. Usually, we want these to propagate all the way up.

A couple of examples of errors are a StackOverflowError and OutOfMemoryError.

## **4. Handling Exceptions**

In the Java API, there are plenty of places where things can go wrong, and some of these places are marked with exceptions, either in the signature or the Javadoc:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /\*\*   \* @exception FileNotFoundException ...   \*/  public Scanner(String fileName) throws FileNotFoundException {     // ...  } |

As stated a little bit earlier, when we call these “risky” methods, we must handle the checked exceptions, and we may handle the unchecked ones. Java gives us several ways to do this:

### **4.1.**throws

The simplest way to “handle” an exception is to rethrow it:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public int getPlayerScore(String playerFile)    throws FileNotFoundException {        Scanner contents = new Scanner(new File(playerFile));      return Integer.parseInt(contents.nextLine());  } |

Because FileNotFoundException is a checked exception, this is the simplest way to satisfy the compiler, but **it does mean that anyone that calls our method now needs to handle it too!**

parseInt can throw a NumberFormatException, but because it is unchecked, we aren’t required to handle it.

### **4.2.**try**–**catch

If we want to try and handle the exception ourselves, we can use a try-catch block. We can handle it by rethrowing our exception:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public int getPlayerScore(String playerFile) {      try {          Scanner contents = new Scanner(new File(playerFile));          return Integer.parseInt(contents.nextLine());      } catch (FileNotFoundException noFile) {          throw new IllegalArgumentException("File not found");      }  } |

Or by performing recovery steps:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public int getPlayerScore(String playerFile) {      try {          Scanner contents = new Scanner(new File(playerFile));          return Integer.parseInt(contents.nextLine());      } catch ( FileNotFoundException noFile ) {          logger.warn("File not found, resetting score.");          return 0;      }  } |

### **4.3.**finally

Now, there are times when we have code that needs to execute regardless of whether an exception occurs, and this is where the finally keyword comes in.

In our examples so far, there ‘s been a nasty bug lurking in the shadows, which is that Java by default won’t return file handles to the operating system.

Certainly, whether we can read the file or not, we want to make sure that we do the appropriate cleanup!

Let’s try this the “lazy” way first:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public int getPlayerScore(String playerFile)    throws FileNotFoundException {      Scanner contents = null;      try {          contents = new Scanner(new File(playerFile));          return Integer.parseInt(contents.nextLine());      } finally {          if (contents != null) {              contents.close();          }      }  } |

Here, the finally block indicates what code we want Java to run regardless of what happens with trying to read the file.

Even if a FileNotFoundException is thrown up the call stack, Java will call the contents of finallybefore doing that.

We can also both handle the exception and make sure that our resources get closed:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public int getPlayerScore(String playerFile) {      Scanner contents;      try {          contents = new Scanner(new File(playerFile));          return Integer.parseInt(contents.nextLine());      } catch (FileNotFoundException noFile ) {          logger.warn("File not found, resetting score.");          return 0;      } finally {          try {              if (contents != null) {                  contents.close();              }          } catch (IOException io) {              logger.error("Couldn't close the reader!", io);          }      }  } |

**Because close is also a “risky” method, we also need to catch its exception!**

This may look pretty complicated, but we need each piece to handle each potential problem that can arise correctly.

### **4.4.**try**-with-resources**

Fortunately, as of Java 7, we can simplify the above syntax when working with things that extend AutoCloseable:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public int getPlayerScore(String playerFile) {      try (Scanner contents = new Scanner(new File(playerFile))) {        return Integer.parseInt(contents.nextLine());      } catch (FileNotFoundException e ) {        logger.warn("File not found, resetting score.");        return 0;      }  } |

When we place references that are AutoClosable in the try declaration, then we don’t need to close the resource ourselves.

We can still use a finally block, though, to do any other kind of cleanup we want.

Check out our article dedicated to [try-with-resources](https://www.baeldung.com/java-try-with-resources) to learn more.

### **4.5. Multiple**catch**Blocks**

Sometimes, the code can throw more than one exception, and we can have more than one catch block handle each individually:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public int getPlayerScore(String playerFile) {      try (Scanner contents = new Scanner(new File(playerFile))) {          return Integer.parseInt(contents.nextLine());      } catch (IOException e) {          logger.warn("Player file wouldn't load!", e);          return 0;      } catch (NumberFormatException e) {          logger.warn("Player file was corrupted!", e);          return 0;      }  } |

Multiple catches give us the chance to handle each exception differently, should the need arise.

Also note here that we didn’t catch FileNotFoundException, and that is because it extends IOException. Because we’re catching IOException, Java will consider any of its subclasses also handled.

Let’s say, though, that we need to treat FileNotFoundException differently from the more general IOException:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public int getPlayerScore(String playerFile) {      try (Scanner contents = new Scanner(new File(playerFile)) ) {          return Integer.parseInt(contents.nextLine());      } catch (FileNotFoundException e) {          logger.warn("Player file not found!", e);          return 0;      } catch (IOException e) {          logger.warn("Player file wouldn't load!", e);          return 0;      } catch (NumberFormatException e) {          logger.warn("Player file was corrupted!", e);          return 0;      }  } |

Java lets us handle subclass exceptions separately,**remember to place them higher in the list of catches.**

### **4.6. Union**catch**Blocks**

When we know that the way we handle errors is going to be the same, though, Java 7 introduced the ability to catch multiple exceptions in the same block:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public int getPlayerScore(String playerFile) {      try (Scanner contents = new Scanner(new File(playerFile))) {          return Integer.parseInt(contents.nextLine());      } catch (IOException | NumberFormatException e) {          logger.warn("Failed to load score!", e);          return 0;      }  } |

## **5. Throwing Exceptions**

If we don’t want to handle the exception ourselves or we want to generate our exceptions for others to handle, then we need to get familiar with the throw keyword.

Let’s say that we have the following checked exception we’ve created ourselves:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class TimeoutException extends Exception {      public TimeoutException(String message) {          super(message);      }  } |

and we have a method that could potentially take a long time to complete:

|  |  |
| --- | --- |
| 1  2  3 | public List<Player> loadAllPlayers(String playersFile) {      // ... potentially long operation  } |

### **5.1. Throwing a Checked Exception**

Like returning from a method, we can throw at any point.

Of course, we should throw when we are trying to indicate that something has gone wrong:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public List<Player> loadAllPlayers(String playersFile) throws TimeoutException {      while ( !tooLong ) {          // ... potentially long operation      }      throw new TimeoutException("This operation took too long");  } |

Because TimeoutException is checked, we also must use the throws keyword in the signature so that callers of our method will know to handle it.

### **5.2.**Throw**ing an Unchecked Exception**

If we want to do something like, say, validate input, we can use an unchecked exception instead:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public List<Player> loadAllPlayers(String playersFile) throws TimeoutException {      if(!isFilenameValid(playersFile)) {          throw new IllegalArgumentException("Filename isn't valid!");      }        // ...  } |

Because IllegalArgumentException is unchecked, we don’t have to mark the method, though we are welcome to.

Some mark the method anyway as a form of documentation.

### **5.3. Wrapping and Rethrowing**

We can also choose to rethrow an exception we’ve caught:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public List<Player> loadAllPlayers(String playersFile)    throws IOException {      try {          // ...      } catch (IOException io) {          throw io;      }  } |

Or do a wrap and rethrow:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public List<Player> loadAllPlayers(String playersFile)    throws PlayerLoadException {      try {          // ...      } catch (IOException io) {          throw new PlayerLoadException(io);      }  } |

This can be nice for consolidating many different exceptions into one.

### **5.4. Rethrowing**Throwable**or**Exception

Now for a special case.

If the only possible exceptions that a given block of code could raise are unchecked exceptions, then we can catch and rethrow Throwable or Exception without adding them to our method signature:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public List<Player> loadAllPlayers(String playersFile) {      try {          throw new NullPointerException();      } catch (Throwable t) {          throw t;      }  } |

While simple, the above code can’t throw a checked exception and because of that, even though we are rethrowing a checked exception, we don’t have to mark the signature with a throws clause.

**This is handy with proxy classes and methods.**More about this can be found [here](http://4comprehension.com/sneakily-throwing-exceptions-in-lambda-expressions-in-java/).

### 5.5. **Inheritance**

When we mark methods with a throws keyword, it impacts how subclasses can override our method.

In the circumstance where our method throws a checked exception:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Exceptions {      public List<Player> loadAllPlayers(String playersFile)        throws TimeoutException {          // ...      }  } |

A subclass can have a “less risky” signature:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class FewerExceptions extends Exceptions {      @Override      public List<Player> loadAllPlayers(String playersFile) {          // overridden      }  } |

But not a “more riskier” signature:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class MoreExceptions extends Exceptions {      @Override      public List<Player> loadAllPlayers(String playersFile) throws MyCheckedException {          // overridden      }  } |

This is because contracts are determined at compile time by the reference type. If I create an instance of MoreExceptions and save it to Exceptions:

|  |  |
| --- | --- |
| 1  2 | Exceptions exceptions = new MoreExceptions();  exceptions.loadAllPlayers("file"); |

Then the JVM will only tell me to catch the TimeoutException, which is wrong since I’ve said that MoreExceptions#loadAllPlayers throws a different exception.

**Simply put, subclasses can throw fewer checked exceptions than their superclass, but not more.**

## **7. Common Exceptions and Errors**

Here are some common exceptions and errors that we all run into from time to time:

### **7.1. Checked Exceptions**

* IOException – This exception is typically a way to say that something on the network, filesystem, or database failed.

### **7.2. RuntimeExceptions**

* ArrayIndexOutOfBoundsException – this exception means that we tried to access a non-existent array index, like when trying to get index 5 from an array of length 3.
* ClassCastException – this exception means that we tried to perform an illegal cast, like trying to convert a String into a List. We can usually avoid it by performing defensive instanceof checks before casting.
* IllegalArgumentException – this exception is a generic way for us to say that one of the provided method or constructor parameters is invalid.
* IllegalStateException – This exception is a generic way for us to say that our internal state, like the state of our object, is invalid.
* NullPointerException – This exception means we tried to reference a null object. We can usually avoid it by either performing defensive null checks or by using Optional.
* NumberFormatException – This exception means that we tried to convert a String into a number, but the string contained illegal characters, like trying to convert “5f3” into a number.

### **7.3. Errors**

* StackOverflowError – this exception means that the stack trace is too big. This can sometimes happen in massive applications; however, it usually means that we have some infinite recursion happening in our code.
* NoClassDefFoundError – this exception means that a class failed to load either due to not being on the classpath or due to failure in static initialization.
* OutOfMemoryError –  this exception means that the JVM doesn’t have any more memory available to allocate for more objects. Sometimes, this is due to a memory leak.

**Summary**:

* An **Exception is a run-time error** which interrupts the normal flow of program execution.Disruption during the execution of the program is referred as error or exception.
* Errors are classified into two categories
  + Compile time errors – Syntax errors, Semantic errors
  + Runtime errors- Exception
* A **robust program should handle all exceptions** and continue with its normal flow of program execution. Java provides an inbuilt exceptional handling method
* Exception Handler is a set of code that **handles an exception**. Exceptions can be handled in Java using try & catch.
* **Try block**: Normal code goes on this block.
* **Catch block**: If there is error in normal code, then it will go into this block

# Questions and Exercises

## Questions

1. Is the following code legal?
2. try {
4. } finally {
6. }
7. What exception types can be caught by the following handler?
8. catch (Exception e) {
10. }

What is wrong with using this type of exception handler?

1. Is there anything wrong with the following exception handler as written? Will this code compile?
2. try {
3. } catch (Exception e) {
5. } catch (ArithmeticException a) {
7. }
8. Match each situation in the first list with an item in the second list.
   1. int[] A;  
      A[0] = 0;
   2. The JVM starts running your program, but the JVM can't find the Java platform classes. (The Java platform classes reside in classes.zip or rt.jar.)
   3. A program is reading a stream and reaches the end of stream marker.
   4. Before closing the stream and after reaching the end of stream marker, a program tries to read the stream again.
   5. \_\_error
   6. \_\_checked exception
   7. \_\_compile error
   8. \_\_no exception

## Exercises

* 1. Add a readList method to [ListOfNumbers.java](https://docs.oracle.com/javase/tutorial/essential/exceptions/examples/ListOfNumbers.java). This method should read in int values from a file, print each value, and append them to the end of the vector. You should catch all appropriate errors. You will also need a text file containing numbers to read in.
  2. Modify the following cat method so that it will compile.
  3. public static void cat(File file) {
  4. RandomAccessFile input = null;
  5. String line = null;
  6. try {
  7. input = new RandomAccessFile(file, "r");
  8. while ((line = input.readLine()) != null) {
  9. System.out.println(line);
  10. }
  11. return;
  12. } finally {
  13. if (input != null) {
  14. input.close();
  15. }
  16. }

}