***Exceptions:***

* *A program can fail for just about one reason. Here are just few possibilities:*
* *The code tried to connect to a website, the Internet connection is down*
* *You made a coding mistake and tried to access an invalid index in an array*
* *One method calls another with a value that the method doesn’t support.*
* *As you can see, some of these coding mistakes. Others are completely beyond your control. Your program cant help it if the Internet connection foes down.*
* *An exception is Java’s way of saying, “I give up I don’t know what to do right now. You deal with it.” When you write a method, you can either deal with the exception or make it the calling code’s problem.*

*public class Zoo {*

*public static void main(String[] args) {*

*System.out.println(args[0]);*

*System.out.println(args[1]);*

*}}*

*Then you tried to call with enough arguments:*

*javac Zoo.java*

*java Zoo Zoo*

*Java realized there’s only one element in the array and index 1 is not allowed. Java threw up its hands in defeat and threw an exception. It dint try to handle the exception.*

* *Exceptions can and do occur all the time even in the solid program code. When you write more advanced programs, you will need to deal with failures in accessing files, networks and outside services.*

*Return codes vs Exceptions:*

* *Exceptions are used when “something goes wrong”. The following code returns -1 instea of throwing an exception of no match found:*

*public int indexOf(String[] names, String name) {*

*for(int i-0;i<names.length;i++){*

*if(names[i].equals(name)) { return i; }*

*}*

*return -1;*

*}*

*This approach is common when writing a method that does a search. For example, imagine being asked to find the name Joe in the array. It is perfectly reasonable that Joe might not appear in the array. When this happens, a special value is returned. An exception should be reserved for exceptional conditions like names being null.*

* *In general, try to avoid return codes. Return codes are commonly used in searches, so programmers are expecting them. In other methods, you will return a special value. An exception forces the program to deal with the problem or end with the exception if left unhandled whereas a return code could be accidentally ignored and cause problems later in the code. A return value could confuse the real data.*

*Understanding Exception Types:*

* *An exception is an event that alters program flow. Java has a throwable superclass for all objects that represent these events.*

*Java.lang.Object 🡨 java.lang.Throwable 🡨 java.lang.Exception, 🡨 java.lang.Error 🡨 java,lang,RuntimeException*

* *Error means something went wrong so horribly that your program should not make an attempt to recover from it. For example, disk drive disappeared, or the program ran out of the memory. These are abnormal conditions that you aren’t likely to encounter and cannot recover from.*
* *Throwable is the parent class of all exception including the Error class. While you can handle Throwable and Error exceptions it is not recommended you do so in your application code.*

*Checked Exceptions:*

* *A checked exception is an exception that must be declared or handled by the application code where it is thrown. In Java, checked exceptions all inherit Exception but not RuntimeException. Checked Exceptions tend to be more anticipated- for example, trying to read a file that doesn’t exist.*
* *Checked Exceptions also include any class that inherits Throwable but not Error or RuntimeException. For example, a class that directly extends Throwable would be a checked exception.*
* *Java has a rule called the handle or declare rule. The handle or declare rule means that all checked exceptions that could be thrown within a method are either wrapped in compatible try and catch blocks or declared in the method signature.*
* *Because checked exceptions tend to be anticipated. Java enforces the rule that the programmer must do something to show the exception was thought about.*
* *While only checked exceptions must be declared and handled in Java, unchecked exceptions may be handled or declared. The distinction is that checked exceptions must be handled or declared, while unchecked exceptions can be optionally handled or declared.*
* *The following fall() method declares that it might throw an IOException which is checked exception:*

*void fall(int distance) throws IOException {*

*if(distance > 10) {*

*throw new IOException();*

*}}*

* *Notice that you are using two different keywords here. The throw keyword tells Java that you want to throw an Exception, while the throws keyword simply declares that the method might throw an Exception. The following fall() method handles the exception*

*Void fall(int distance) {*

*try {*

*if(distance>10) {*

*throw new IOException();*

*}}catch(Exception e) {*

*e.printStackTrace();*

*}}*

*Notice that the catch statements use Exception, not IOException. Since IOException is a subclass of Exception, the catch block is allowed to catch it.*

*Unchecked Exceptions:*

* *An unchecked exception is any exception that doesn’t need to be declared or handled by the application code where it is thrown. Unchecked exceptions are often referred to as runtime exceptions although in Java, unchecked exceptions include nay class that inherits RuntimeException or Error.*
* *A runtime exception is defined as the RuntimeException class and its subclasses. Runtime exceptions tend to be unexpected but not necessarily fatal. For example, accessing an invalid array index is unexpected. Even though they do inherit the Exception class, they are not checked exceptions.*

*Runtime vs at the time of the program is run:*

* *A runtime(unchecked) exception is a specific type of exception. All exceptions occur at the time of the program is run. The alternative is compile time, which would be a compiler error. People don’t refer to them as runtime exceptions because that would be too easy to confuse with runtime!!. When you see runtime, it means unchecked.*
* *An unchecked exception can often occur on nearly any line of code, as it is not required to be handled or declared. For example, a NullPointerException can be thrown in the body of the following method if the input reference is null.*

*Void fall( String input ) {*

*System.out.println(input.toLowerCase());*

*}*

* *We work with objects in Java so frequently, a NullPointerException can happen almost anywhere. If you had to declare unchecked exceptions everywhere, every single method would have that clutter!! The code will compile if you declare an unchecked exception. However it is redundant.*

*Checked vs Unchecked(Runtime) Exceptions:*

* *According to Oracle, they are intended for issues a programmer might reasonably be expected to recover from. Then developers started writing code where a chain of methods kept declaring the same exception and nobody actually handled it. Some libraries started using unchecked exceptions for issues a programmer might reasonably be expected to recover from.*

*Throwing an Exception:*

* *Any Java code can throw an exception. This includes code we write.*

*String[] animals = new String[0];*

*System.out.println(animals[0]);*

*This code throws an ArrayIndexOutOfBoundsException since the array has no elements. Pay attention to code that calls a method on a null reference or that references on invalid array or List index. If you spot this the correct answer is that the code throws an exception at runtime.*

* *The second way for code to result in an exception is to explicitly request Java to throw one.*

*throw new Exception();*

*throw new Exception(“Ow! I fell…”);*

*throw new Exception();*

*throw new RuntimeException();*

*throw new RuntimeException(“Ow!! I fell…”);*

* *The throw keyword tells Java you want some other part of the code to deal with the exception.*

*Throw vs Throws:*

* *The throw keyword is used as a statement inside a code block to throw a new exception or rethrow an existing exception while the throws keyword is used only at the end of a method declaration to indicate what exceptions it supports.*
* *When creating an exception, you can usually pass a String parameter with message, or you can pass no parameters and use the defaults. We say usually because this is a convention.*
* *Someone could create an exception class that doesn’t have a constructor that takes a message. The first two examples create a new object or type Exception and throw it. The last two show that the code looks the same regardless of which type of exception you throw.*
* *Additionally, point to be noted is that an Exception is an Object. This means you can store in a variable and this is legal*

*Exception e = new RuntimeException();*

*throw e;*

* *The code instantiates an exception on one line and then throws on the next; The exception can come from anywhere, even passed into a method. As long as it is a valid exception it can be thrown.*

*throw RuntimeException(); //Doesn’t compile*

*If your answer is that there is a missing keyword you are absolutely right; The exception is never instantiated with the new keyword.*

*try{*

*throw new RuntimeException();*

*throw new ArrayIndexOutOfBoundsException(); //Doesn’t compile*

*}catch(Exception e){*

*}*

*Since line 2 throws an exception. Line 3 can never be reached during runtime. The compiler recognizes this and reports an unreachable code error.*

|  |  |  |  |
| --- | --- | --- | --- |
| *Type* | *How to recognize* | *Okay for program to catch* | *Is program required to handle or declare?* |
| *RuntimeException* | *Subclass of RuntimeException* | *Yes* | *No* |
| *CheckedException* | *Subclass of Exception but not subclass of RuntimeException* | *Yes* | *Yes* |
| *Error* | *Subclass of Error* | *No* | *No* |

***Recognizing Exception classes:***

* *You need to recognize three groups of exception classes- RuntimeException, checked exception and error. For some exceptions, you also need to know which are inherited from one another.*

*RuntimeException classes:*

* *RuntimeException and its subclasses are unchecked exceptions that don’t have to be handled or declared. They can be thrown by the programmer or by the JVM. Common RuntimeException classes include the following:*

***ArithmeticException****: thrown when code attempts to divide by zero.*

***ArrayIndexOutOfBoundsException:*** *Thrown when code uses an illegal index to access an array.*

***ClassCastException:*** *Thrown when an attempt is made to cast an object to a class of which it is not an instance.*

***NullPointerException:*** *Thrown when there is a null reference where an object is required.*

***IllegalArgumentException:*** *Thrown by the programmer to indicate that a method has been passed an illegal or inappropriate argument.*

***NumberFormatException:*** *Subclass of IllegalArgumentException thrown when an attempt is made to convert a string to a numeric type but the string doesn’t have an appropriate format.*

***ArithmeticException:***

*Trying to divide an int by zero gives an undefined result. When this occurs, the JVM will throw an ArithmeticException:*

*int answer = 11/0;*

*Running this code results in the following output:*

*Exception in thread “main” java.lanf.ArithmeticException: / by zero*

*Java doesn’t spell out the word divide. That’s okay, though, because we know that / is the division operator and that Java is trying to tell you division by zero occurred.*

*The thread “main” is telling you the code was called directly or indirectly from a program with a main method. Next comes the name of the exception, followed by extra information that goes with the exception.*

*ArrayIndexOutOfBoundsException:*

* *Array indexes start with 0 and go upto 1 less than the length of the Array- which means this code will throw an ArrayIndexOutOfBoundsException:*

*Int[] countsOfMoose = new int[3];*

*System.out.println(countsOfMosse(-1));*

*This is a problem because there is no such thing as a negative array index. Running this code yields the following output:*

*Exception in thread “main”*

*Java.lang.ArrayIndexOutOfBoundsException:*

*Index -1 out of bounds for length 3*

*Atleast Java tells us at what index was invalid.*

*int total = 0;*

*int[] countsOfMoose = new int[3];*

*for(int i=0; i<=countsOfMoose.length; i++){*

*total += countsOfMoose[i];*

* *The problem is that the for loop should have < instead of <=. On the final iteration of the loop, Java tries to call countsOfMoose[3], which is invalid. The array includes only three elements, making 2 the largest possible index.*

*ClassCastException:*

* *Java tries to protect you from impossible casts. This code doesn’t compile because integer is not a subclass of String*
* *String type = “moose”;*

*Integer number = (Integer) type; //Doesn’t compile*

*More complicated code thwarts Java’s attempts to protect you. When the cast fails at runtime, Java will throw a ClassCastException:*

*String type = “moose”;*

*Object obj = type;*

*Integer number = (Integer) obj;*

*The compiler sees a cast from Object to Integer. This could be okay. The compiler doesn’t realize there is a string in that Object. When this code runs it yields the following exception*

*Exception in thread “main” java.lang.ClassCastException*

*Java tells you both types that were involved in the problem making it apparent whats wrong.*

*NullPointerException:*

* *Instance variables and methods must be called on a non-null reference. If the reference is null, the JVM will throw a NullPointerException. Its usually subtle, such as in the following example, which checks whether you remember instance variable references default to null.*

*String name;*

*public void printLength() {*

*System.out.println(name.length());*

*}*

*Running this code results in NullPointerException*

*IllegalArgumentException:*

* *IllegalArgumentException is a way for your program to protect itself.*

*public void setNumberEggs(int numberEggs) { //setter*

*if(numberEggs >= 0) //guard condition*

*this.numberEggs = numberEggs;*

*}*

*This code works, but you don’t really want to ignore the callers request when they tell you a Swan has -2 eggs. You want to tell the caller that something is wrong- preferably in an obvious way that the caller cant ignore so that the programmer will fix the problem. Exceptions are an efficient way to do this. Seeing the code end with an exception is great reminder that something is wrong*

*public void setNumberEggs(int numberEggs) {*

*if(numberEggs < 0)*

*throw new IllegalArgumentException(“# eggs must not be negative”);*

*this.numberEggs = numberEggs;*

*}*

*The program throws an exception when it is not happy with the parameter values. The output gives an IllegalArgumentException*

*NumberFormatException*

* *Java provides methods to convert Strings to numbers. When these are passed an invalid value, they throw a NumberFormatException. Since, this is a common problem, Java gives it a separate class. In fact, NumberFormatException is a subclass of IllegalArgumentException.*

*Integer.parseInt(“abc”);*

*The output states NumberFormatException*

*Checked Exception classes:*

* *Checked Exceptions have Exception in their hierarchy but not RuntimeException. They must be handled or declared. Common Checked Exceptions include following:*
* *IOException: Thrown programmatically when there is a problem reading or writing a file*
* *FileNotFoundException: Subclass of IOException thrown programmatically when code tries to reference a file that doesn’t exist.*

*Error Classes:*

* *Errors are unchecked exceptions that extend the Error class. They are thrown by the JVM and should not be handled or declared. Errors are rare, but you might see this:*

*ExceptionInInitializeError: Thrown when a static initializer throws an exception and doesn’t handle it.*

*StackOverflowError: Thrown when a method calls itself too many times. This is called as Infinite Recursion because the method typically calls itself without end.*

*NoClassDefFoundError: Thrown when a class that the code uses is available at compile time but not runtime.*

***ExceptionInitializerError****:*

* *Java runs static initializers the first time a class is used. If one of the static initializers throws a Exception, Java cant start using the class. It declares defeat by throwing an ExceptionInitializerError. This code throws an ArrayIndexOutOfBounds in a static initializer.*

*static {*

*int[] countsOfMoose = new int[3];*

*int num = countsOfMoose[-1];*

*}*

*public static void main(String… args) {}*

*This code yields information about the error and the underlying exception*

*Exception in thread “main”*

*Java.lang.ExceptionInitializerError*

*Caused by: java.lang.ArrayIndexOutOfBoundsException: -1 out of bounds for length 3*

* *When executed, you get an ExceptionInitializerError because the error happened in a static initializer. That information alone wouldn’t be particularly useful in fixing the problem. Therefore, Java also tells you the original cause of the problem: the ArrayIndexOutOfBoundsException that you need to fix.*
* *The ExceptionInInitializerError is an error because Java failed to load the whole class. This failure prevents Java from continuing.*

***StackOverflowError****:*

* *When Java calls methods, it puts parameters and local variables on the stack. After doing this a very large number of times, the stack runs out of the room and overflows. This is called a StackOverflowError. Most of the times this error occurs when a method calls itself.*
* *public static void doNotCodeThis(int num) {*

*doNotCodeThis(1);*

*}*

*This output contains: Exception in thread “main” java.lang.StackOverflowError*

* *Since the method calls itself, it will never end. Eventually, Java runs out of the room on the stack and throws this error. This is called infinite recursion. It is better than an infinite loop because atleast Java will catch it and throw the error. With an infinite loop, Java just uses all your CPU until you can kill the program.*

*NoClassDefFoundError:*

* *A NoClassDefFoundError occurs when Java cant find the class at runtime. Generally, this means a library available when the code was compiled and is not available when the code is executed.*

*Handling Exception using TRY and CATCH statements:*

* *Java uses a try statement to separate the logic that might know an exception from the logic to handle that exception. The code in the try block is run normally. If any of the statements throw an exception that can be caught in the exception type listed in the catch block, then the try block stops running and execution goes to the catch statement. If none of the statements in the try block throws an exception that can be caught, the catch clause will not run. You probably notices the words block and clause used interchangeably. Both are correct. Block is correct because there are braces present. Clause is correct because they are part of the try statement.*

*void explore() {*

*try {*

*fall();*

*System.out.println(“never get here”);*

*}catch(RuntimeException e) {*

*getUp();*

*}*

*seeAnimals();*

*}*

*void fall() { throw new RuntimeException(); }*

*First falls() method is called. Line 12 throws an exception. This means Java jumps straight to the catch block. Now the try statement is over, and execution proceeds normally.*

* *The try statements are like methods in that the curly braces are required even if there is only one statement inside the code blocks, while the if statements and loops are special and allow you to omit the curly braces.*

*try {*

*fall();*

*}*

*This code doesn’t compile because the try block doesn’t have anything after it. Remember, the point of a try statement is for something to happen if an exception is thrown. There is a special type of try statement that includes an implicit finally block, although the syntax for this quite different.*

***Chaining Catch Blocks:***

* *So far, you have been catching only one type of exception. First, you have to recognize if the exception is a checked or unchecked exception. Second, you need to determine whether any of the exceptions are subclasses of the others.*

*Class AnimalsOutForAWalk extends RuntimeException {}*

*Class ExhibitClosed extends RuntimeException {}*

*Class ExhibitClosedForLunch extends ExhibitClosed {}*

*In this example, there are three custom exceptions. All are unchecked exceptions because they are directly or indirectly extend RuntimeException. Now we chain both types of exceptions with two catch blocks and handle them by printing out the appropriate message:*

*public void visitPorcupine {*

*try {*

*seeAnimal();*

*} catch(AnimalsOutForWalk e) { //First Catch Block*

*System.out.print(“try back later”);*

*}catch(ExhibitClosed e) { //second catch block*

*System.out.print(“Not today”);*

*}*

*}*

* *There are three possibilities for when this code is run. If seeAnimal() doesn’t throw an exception, noting is printed out. If the animal is out for a walk, only the first catch block runs. If the exhibit is closed,, only the second catch block runs. It is not possible for both catch blocks to be executed when chained together like this.*
* *A rule exists for the order of catch blocks. Java looks at them in order they appear. If it is impossible for one of the catch blocks to be executed, a compiler error about unreachable code occurs. For this example, this happens when a superclass catch block appears before a subclass catch block.*
* *public void visitSnakes() {*

*try {*

*}catch(IllegalArgumentException e) {*

*}catch(NumberFormatException e) { //Doesn’t compile*

*}}*

*NumberFormatException is a subclass of IllegalArgumentException. Since NumberFormatException is a subclass it always be caught by the first catch block making the second code block unreachable and that doesn’t compile. Similarly, FileNotFoundException is a subclass of IOException and cannot be used in similar manner.*

* *To review multiple catch blocks, remember that at most ne catch block will run, and it wil be the first catch block that can handle it. Also, remember that an exception defined by the catch statement is only in scope for that catch block.*

*public void visitManatees() {*

*try {*

*}catch(NumberFormatException e) {*

*System.out.println(e1);*

*}catch(IllegalArgumentException e2){*

*System.out.println(e2); //Doesn’t compile*

*}}*

*Applying a multi-catch block:*

* *Oftentimes, we want the result of an exception being thrown to be the same regardless of which particular exception is thrown.*

*public static void main(String args[]) {*

*try {*

*System.out.println(Integer.parseInt(args[1]));*

*}catch(ArrayIndexOutOfBoundsException e){*

*System.out.println(“Missing or invalid input”);*

*} catch(NumberFormatException e) {*

*System.out.println(“Missing or invalid input”);*

*}}*

*Notice that we have the same println() statement for two different catch blocks. Reduce duplicate code? One way is to have the related exception classes all inherit the same interface or extend the same class. You can have a single catch block that just catches Exception. This will catch everything and anything. Another way is to move the println statements into a separate method and have every related catch block call that method.*

* *While these solutions are valid, Java provides another structure to handle this more gracefully called a multi catch block. A multi catch block allows multiple exception types to be caught be the same catch block.*

*public static void main(String args[]) {*

*try {*

*System.out.println(Integer.parseInt(args[1]));*

*}catch(ArrayIndexOutOfBoundsException | NumberFormatException e) {*

*System.out.println(“Missing or invalid input…”);*

*}}*

*This is much better. There is no duplicate code, the common logic is all in one place, ad the logic is exactly where you would expect to find it. If you wanted, you could still have a second catch block for Exception in case you want to handle all other types of exceptions differently.*

* *Its like a regular catch clause, except two or more exception types are specified separated by a pipe. The pipe(|) is also used as the “or” operator making it easy to remember that you can use either/or the exception types. Notice that there is only one variable name in the catch clause. Java is saying that the variable named e can be of type Exception1 or Exception2.*
* *Remember that the exceptions can be listed in any order within the catch clause.*

*catch(Exception1 e | Exception2 e | Exception3 e ) //Doesn’t compile*

*catch(Exception1 e1 | Exception2 e2 | Exception3 e3) // Doesn’t compile*

*catch(Exception1 | Exception2 | Exception3 e)*

*The first line is incorrect because the variable name appears three times. Just because it happens to be the same variable name doesn’t make it okay. The second line is incorrect because the variable name again appears three times. Using different variable names doesn’t make it any better. The third line does compile. It shows the correct syntax for specifying three exceptions.*

* *Java intends multi-catch t be used for exceptions that aren’t related and it prevents you from specifying redundant types in a multi catch.*

*try {*

*throw new IOException();*

*}catch(FileNotFoundException | IOException p ) // Doesn’t compile*

*}*

*Specifying it in the multicatch is redundant, and the compiler gives a message*

* *Since FileNotFoundException is a subclass of IOException, this code will not compile. A multi catch block follows similar rules as chaining catch blocks together. For example, both trigger compiler errors when they encounter unreachable code or duplicate exceptions being caught. The one difference between multi-catch blocks and chaining catch blocks is that order doesn’t matter for a multi-catch block within a single expression.*

*try{*

*throw new IOException();*

*}catch(IOException e) {}*

*To review multi-catch, see how many errors you can find in this try statement.*

*public void doesNotCompile() //Method doesn’t compile*

*try {*

*mightThrow();*

*}catch(FileNotFoundException | IllegalStateException e) {*

*}catch(InputMismatchException e | MissingResourceException e) {*

*}catch(FileNotFoundException | IllegalStateException e) {*

*} catch(Exception e){*

*} catch(IOException e) {*

*}}*

*private void mightThrow() throws DateTimeParseException, IOException {}*

*This code is just swimming with errors. In fact, same errors hide others, so you might not see them all in the compiler. The second catch statement has an extra variable name. Remember that there can be only one exception variable per catch block. Third catch cannot catch FileNotFoundException because that exception was already caught on first catch. You cant list the same exception type more than once in the same try statement just like regular catch blocks. Fourth and fifth catch blocks are reversed. The more general superclasses must be caught after their subclasses. While this doesn’t have anything to do with multi-catch you will see regular catch block problems mixed in with multi-catch.*

***Adding a Finally block:***

* *The try statement also lets you run code at the end with a finally clause regardless of whether an exception is thrown. There are two paths through code with a catch and a finally. If an exception is thrown, the finally block is run after the catch block. If no exception is thrown, the finally block is run after the try block completes.*

*void explore() {*

*try {*

*seeAnimals();*

*fall();*

*}catch(Exception e) {}*

*finally { seeMoreAnimals(); }*

*goHome();*

*The girl falls.If she gets up by herself the the code goes on to the finally block and then runs seeMoreAnimals(). Then the try statement is over and the code goes to last line. If the girl doesn’t get up by herself, she throws an exception which is handled by catch. The catch block runs and then she gets a hug with that she is ready to seeMoreAnimals. Either way, ending is the same. The finally block is executed, and execution continues after the try statement.*

*try{ //Doesn’t compile*

*fall();*

*}finally {*

*System.out.println(“All better….”);*

*}catch(Exception e){*

*System.out.println(“get up…”);*

*}*

*try{ //Doesn’t compile*

*fall();*

*}*

*try {*

*fall();*

*} finally{*

*System.out.println(‘All better….”);*

*}*

*The first example doesn’t compile because catch and finally blocks are in wrong order. The second example doesn’t compile because there must be a catch or finally block. The third example is just fine. The catch block isn’t required if finally is present.*

*A finally block is typically used to close resources such as files or databases.*

*public static void main(String[] args) {*

*StringBuilder sb = new StringBuilder();*

*try{*

*sb.append(“t”);*

*}catch(Exception e) {*

*sb.append(“c”);*

*}finally{*

*sb.append(“f”);*

*}*

*sb.append(“a”);*

*System.out.print(sb.toString());*

*}*

*The answer is tfa. The try block is executed. Since no exception is thrown, Java goes straight to the finally block. Then the code after the try block is run.*

* *There is one additional rule you should know for finally blocks. If a try statement with a finally block is entered, then the finally block will always be executed, regardless of whether the code completes successfully.*

*int goHome() {*

*try {*

*//Optionally throw an exception here*

*System.out.print(“1”);*

*Return -1;*

*} catch(Exception e) {*

*System.out.print(“2”);*

*Return -2;*

*}finally{*

*System.out.print(“3”);*

*Return -3*

*}}*

*If an exception is thrown, 1 will be printed. Before the method returns though the finally block is executed printing 3. If an exception is thrown, 2 will be printed followed by 3 from the finally block. While the first value printed may differ, the method always prints 3 last since its in the finally block.*

*The return value is always -3. Because the finally block is executed shortly before the method completes, it interrupts the return statement from inside both the try and catch blocks.*

*Finally block always executes, that said it may not complete successfully.*

*}finally{*

*Info.printDetails();*

*System.out.print(“Exiting…”);*

*Return “Zoo”;*

*}*

*If Info is null then the finally block would be executed but it would stop at second line and throw a NullPointerException. Next lines will not be executed. Here you can see finally block is always executed but it may not finish.*

*SYSTEM.EXIT():*

* *There is one exception to the finally block always be executed rule. Java defines a method that you call as System.exit(). It takes an integer parameter that represents the error code that gets returned.*

*try{ System.exit(0);*

*}finally{*

*System.out.print(“Never going o be here”); //Not printed*

*}*

*System.exit() tells Java “Stop End the program right now.*

*Finally Closing Resources:*

* *Oftentimes, your application works with files, databases and various connection objects. Commonly, these external data sources are referred to as resources. In many cases, you open a connection to the resource, whether its over the network or within the file system. You then read/write the data you want. Finally, you close the resource to indicate you are done with it. What happens if you don’t close a resource when you are done with it?*
* *In short, a lot of bad things could happen. If you are connecting to a database, you could use up all available connections meaning no one talk to the database until you release your connections. Although you commonly hear about memory leaks as causing program fails to release its connections to a resource, resulting in the resource becoming inaccessible.*
* *Writing code that simplifies closing resources is what this section is about.*

*public void readFile(String file) {*

*FileInputStream is = null;*

*try{*

*is = new FileInputStream(“myFile.txt”);*

*//Read file data*

*}catch(IOException e){*

*e.printStackTrace();*

*}finally{*

*If(is!=null) {*

*try {*

*is.close();*

*}catch(IOException e2){*

*E2.printStackTrace();*

*}}}}*

*Why do we have two try and catch blocks? Well, the code on lines 7 and 14 both include checked IOException calls, so they both need to be caught in the method or rethrown by the method. Half the lines of code in this method are just closing the resource. And the more resources you have the longer code like this becomes. You may have multiple resources and they need to be closed in a particular order. You also don’t want an exception from closing one resource to prevent the closing of another resource.*

*To solve this, Java includes the try-with-resources statement to automatically close all resources opened in a try clause. This feature is also known as automatic resource management, because Java automatically takes care of the closing.*

*public void readFile(String file) {*

*try(FileInputStream is = new FileInputStream(“myFile.txt”)) {*

*}catch(IOException e){*

*e.printStackTrace();*

*}}*

*Functionally, they are both quite similar, but our new version hs half as many lines. More importantly though by using a try-with-resources statement, we guarantee that as soon as a connection passes out of scope, java will attempt to close it within the same method.*

*Implicit Finally Blocks:*

* *Behind the scenes, the compiler replaces a try-with-reosurces block with a try and finally block. We refer to this hidden finally block as an implicit finally block since it is created and used by the compiler automatically. You can still create a programmer defined finally lock when using a try-with-reosurces statement*

*Basics of Try-With-Resources:*

* *Notice that one or more resources can be opened in the try clause. When there are multiple resources opened, they are closed in the reverse order from which they are created. Also, notice that parenthese are used to list those resources, and semicolons used to separate the declarations. This works just like declaring multiple indexes in a for loop.*
* *Well it turns out that a catch block is optional with a try-with-resources statement. We learnt earlier that a try statement must have one or more catch blocks or a finally block. This is till true. The finally clause exists implicitly. You just don’t have to type it.*
* *Remember that only a try-with-resources statement is permitted to omit both the catch and finally blocks. A traditional try statement must have either or both. You can easily distinguish between the two by the presence of parentheses, () after the try keyword.*
* *Try-with-resources is still allowed to have catch and/or finally blocks. In fact, if the code within the try block allows a checked exception not declared by the method in which it is defined or handled by another try/catch block, then it will need to be handled by the catch block. also, the catch and finally blocks are run in addition to the implicit one that close the resources. Implicit finally block runs before any programmer coded ones.*

*Legal vs Illegal Configurations with a traditional try statement:*

|  |  |  |  |
| --- | --- | --- | --- |
|  | *0 finally blocks* | *1 finally block* | *2 or more finally blocks* |
| *0 catch blocks* | *Not Legal* | *Legal* | *Not Legal* |
| *1 or more catch* | *Legal* | *Legal* | *Not Legal* |

*Legal vs Illegal Configurations with a try-with-resources statement:*

|  |  |  |  |
| --- | --- | --- | --- |
|  | *0 finally blocks* | *1 finally block* | *2 or more finally* |
| *0 catch blocks* | *Legal* | *Legal* | *Not Legal* |
| *1 or more catch* | *Legal* | *Legal* | *Not Legal* |

* *You can see that for both of these try statements, two or more programmer defined finally blocks are not allowed. Remember that the implicit finally block defined by the compiler is not counted here.*

*AutoCloseable:*

* *You cant just put any random class in a try-with-resources statement. Java requires classes used in a try-with-resources statement. Java requires classes used in a try-with-resources implement the AutoCloseable interface, which includes a void close() method..*

*Declaring Resources:*

* *While try-with-resources does support declaring multiple variables, each variable must be declared in a separate statement.*

*Try(MyFileClass is = newFileClass(1), os = newFileClass(2)) { //Doesn’t compile*

*}*

*Try(MyFileClass ab = new FileClass(1), MyFileClass cd = new MyFileClass(2)) { //Doesn’t compile*

*}*

*A try-with-resources statement doesn’t support multiple variable declarations. The first example doesn’t compile because it is missing the data type and it uses a comma instead of semicolon. The second example doesn’t compile because it also uses a comma instead of semicolon. Each resource must include the data type and be separated by a semicolon.*

*You can declare a resource using var as the data type in a try-with-resources statement, since resources are local variables.*

*Try(var f = new BufferedInputStream(new FileInputStream(“it.txt”))){*

*//Process file*

*}*

*Declaring resources is a common situation where var is quite helpful as it shortens the already long line of code.*

*Scope of try-with-resources:*

* *The resources created in the try class are in scope only within the try block. This is another way to remember that the implicit finally runs before any catch/finally blocks that you code yourself. The implicit code has run already, and the resource is no longer available.*

*try ( Scanner s = new Scanner(System.in)) {*

*s.nextLine();*

*}catch(Exception e) {*

*s.nextLine(); //Doesn’t compile*

*} finally {*

*s.nextInt() ;//Doesn’t compile*

*}*

*The problem is that Scanner has gone out of scope at the end of the try clause.. This is actually a nice feature. You cant accidentally use an object that has been closed. In a traditional try statement, the variable has to be declared before the try statement so that both the try and finally blocks can access it, which has the unpleasant side effect of marking the variable in scope for the rest of the method.*

*Following order of operation:*

* *Resources are closed after the try clause ends and before any catch/finally clauses.*
* *Resources are closed in the reverse order from which they were created.*

*Lets review these principles with a more complex example. First, we define a custom class that you can use with try-with-resources statement as it implements AutoCloseable.*

*public class MyFileClass implements AutoCloseable {*

*private final int num;*

*public MyFileClass(int num) { this.num=num; }*

*public void close() {*

*System.out.println(“Closing:” + num);*

*}}*

*This is pretty simple class that prints the number, set by the constructor, when a resource is closed. Based on these rules*

*public static void main(String.. xyz) {*

*try(MyFileClass al = new MyFileClass(1); MyFileClass a2=new MyFileClass(2)) {*

*throw new RuntimeException();*

*}catch(Exception e){*

*System.out.println(“ex”);*

*}finally{*

*System.out.println(“finally”);*

*}}*

*Since the resources are closed in the reverse order from which they were opened, we have closing:2 and then closing:1, After that there is a catch block and finally block are run- just as they are in regular try statement. The output is as follows:*

*Closing:2*

*Closing:1*

*Ex*

*Finally*

*Does a try-with-resources statement guarantee a resource will be closed? The answer is “No”.*

*The try-with-resources statement guarantees only the close() method will be closed. If the close() method encounters an exception of its own or the method is implemented poorly, a resource leak can still occur.*

*Throwing Additional Exceptions:*

* *A catch or finally block can have any valid Java code in it- including another try statement. What happens when an exception is thrown inside of a catch or a finally block?*

*public static void main(String[] args) {*

*FileReader reader = null;*

*try{*

*reader= read();*

*}catch(IOException e){*

*Try{*

*If(reader!=null) reader.close();*

*}catch(IOException inner) {}*

*}}*

*private static FileReader read() throws IOException {//Code goes here}*

*The easiest way is if first try statement doesn’t throw an exception. Then the entire catch statement is skipped. Next consider, if first try statement throws an NullPointerException and not an IOException so the catch block will still be skipped resulting in the main() method terminating easily. If first try throws an IOException, the catch block runs and tries to close the reader. If that goes well the code completes and the main() method ends normally. If the close method does throw an exception, Java looks for more catch blocks. The exception is caught regardless the exception is handled, a different exception might be thrown.*

*try{*

*throw new RuntimeException();*

*}catch(RuntimeException e) {*

*throw new RuntimeException();*

*}finally{*

*throw new Exception();*

*}*

*Try block throws an exception which is caught in first catch statement. If there were no finally block the exception from catch would be thrown. However, the finally block runs after the catch block and since the finally block throws an exception of its own this one gets thrown. Th exception from catch is forgotten. This is why you often see another try/catch inside a finally block to make sure it doesn’t mask the exception from catch block.*

*public String exceptions() {*

*StringBuilder result = new StringBuilder();*

*String v=null;*

*try{*

*try{*

*result.append(“before\_”);*

*v.length();*

*result.append(“after\_”);*

*}catch(NullPointerException e) {*

*result.append(“catch\_”);*

*throw new RuntimeException();*

*}finally{*

*result.append(“finally\_”);*

*throw new Exception();*

*}*

*}catch(Exception e){*

*result.append(“done”);*

*}*

*return result.toString();*

*}*

*The correct answer is before\_catch\_finally\_done. First before\_ is added. Then a NullPointerException is throw. Java goes straight to catch block. Catch\_ is added. A RuntimeException is thrown. The finally block runs after the catch regardless of whether an exception is thrown, it adds finally\_ to the result. At this point, we have completed the inner try statement that ran. The outer catch block then sees and exception was thrown and catches it and adds done to the result.*

*Calling Methods that throw Exceptions:*

* *When you are calling a method that throws an exception, the rules are the same within a method.*

*class NoMoreCarrotsException extends Exception {}*

*public class Bunny{*

*public static void main(String args[]){*

*eatCarrot(); //Doesn’t compile*

*}*

*private static void eatCarrot() throws NoMoreCarrotsException{}*

*}*

*The problem is that NoMoreCarrotsException is a checked exception. Checked Exceptions must be handled or declared. The code would compile if you changed the main() method*

*public static void main(String[] args) throws NoMoreCarrotsException {*

*eatCarrot();*

*}*

*public static void main(String[] args) {*

*try{*

*eatCarrot();*

*}catch(NoMoreCarrotsException e) { //handle exception*

*System.out.print(“sad rabbit”);*

*}}*

*You might have noticed that the eatCarrot() dint actually throw an exception it just declared that it could. This is enough for the compiler to require the caller to handle or declare the exception. The compiler is still on the lookout for unreachable code. Declaring an unused exception isn’t considered unreachable code. It gives the method the option to change the implementation to throw that exception.*

*public void bad() {*

*try {*

*eatCarrot();*

*}catch(NoMoreCarrotsException e){*

*System.out.print(“sad rabbit”);*

*}}*

*Public void god() throws NoMoreCarrotsException{*

*eatCarrot();*

*}*

*public void eatCarrot() {}*

*Java knows that the eatCarrot() cant throw a checked exception which means there is no way for the catch block in bad() to be reached.*

*When you see a checked exception declared inside a catch block check and make sure the code is associated try block is capable of throwing the exception or a subclass of the exception. If not, the code is unreachable and doesn’t compile. This rule doesn’t extend to unchecked exceptions declared in the method signature.*

*Declaring and Overriding the methods with Exceptions*

* *When a class overrides a method from a superclass or implements a method from an interface its not allowed to add new checked exceptions to the method signature.*

*class CanNotHopException extends Exception {}*

*class Hopper{*

*public void hop() {}*

*}*

*Class Bunny extends Hopper {*

*public void hop() throws CanNotHopException {} //Doesn’t compile*

*}*

*Java knows hop() isn’t allowed to throw any checked exceptions because hop method in the super class Hopper doesn’t declare any. Imagine what would happen if the subclasses version of the method could add checked exceptions- you could write code that calls Hoppers hop() method and not handle any exceptions.*

*An overridden method in a subclass is allowed to declare fewer exceptions than the superclass or interface. This is legal because callers are already handling them.*

*class Hopper{*

*public void hop() throws CanNotHopException {}*

*}*

*Class Bunny extends Hopper{*

*public void hop() {}*

*}*

*An overridden method not declaring one of the exceptions thrown by the parent method is similar to the method declaring it throws an exception that it never actually throws.. This is perfectly legal.*

*Similarly a class is allowed to declare a subclass of an exception type. The idea is the same. The superclass or interface has already taken care of a broader type.*

*class Hopper{*

*public void Hop() throws Exception {}*

*}*

*Class Bunny extends Hopper {*

*public void hop() throws CanNotHopException {}*

*}*

*Bunny could declare that it throws Exception directly or it could declare that it throws a more specific type of Exception. It could even declare that it throws nothing at all.*

*This rule applies only to checked exceptions. The following code is legal because it has an unchecked exception in the subclass version:*

*class Hopper {*

*public void hop() {}*

*}*

*class Bunny extends Hopper {*

*public void hop() throws IllegalStateException {}*

*}*

*The reason that its okay to declare new unchecked exceptions in a subclass method is that the declaration is redundant. Methods are free to throw any unchecked exceptions they want without mentioning them in the method declaration.*

*Printing an Exception:*

* *There are three ways to print an exception. You can let Java print it out, print just the message, or print where the stack trace comes from. This example shows all three approaches:*

*public static void main(String[] args) {*

*try{*

*hop();*

*}catch(Exception e) {*

*System.out.println(e);*

*System.out.println(e.getMessage());*

*e.printStackTrace();*

*}}*

*private static void hop() {*

*throw new RuntimeException(“cannot hop”);*

*}*

*The first line shows what Java prints out by default: the exception and the message. The second line shows just the message. The rest shows a stack trace.*

*The stack trace is usually the most helpful one because it is a picture in time the moment the exception is thrown. it shows the hierarchy of method calls that were made to reach the line that threw the exception.*

*The stacktrace shows all the methods on the stack. Everytime you call a amethod, Java adds it to the stack until it completes. When an exception is thrown, it goes through the stack until it finds a method that can handle it or it runs out of the stack.*

*Why swallowing exceptions is bad?*

* *Because checked exceptions require you to handle or declare them, there is a temptation to catch them so they go away. But doing so can cause problems.*

*public static void main(String… p) {*

*String textInFile = null;*

*try{*

*textInFile = readInFile();*

*}catch(IoException e) {//ignore exception }*

*……*

*System.out.println(textInFile.replace(“ “,””);*

*}*

*private static Strign readInFile() throws IOException {*

*throw new IOException();*

*}*

*The code results in a NullPointerException. Java doesn’t tell you anything about the original IOException because it was handled. Granted, it was handled poorly but it was handled.*