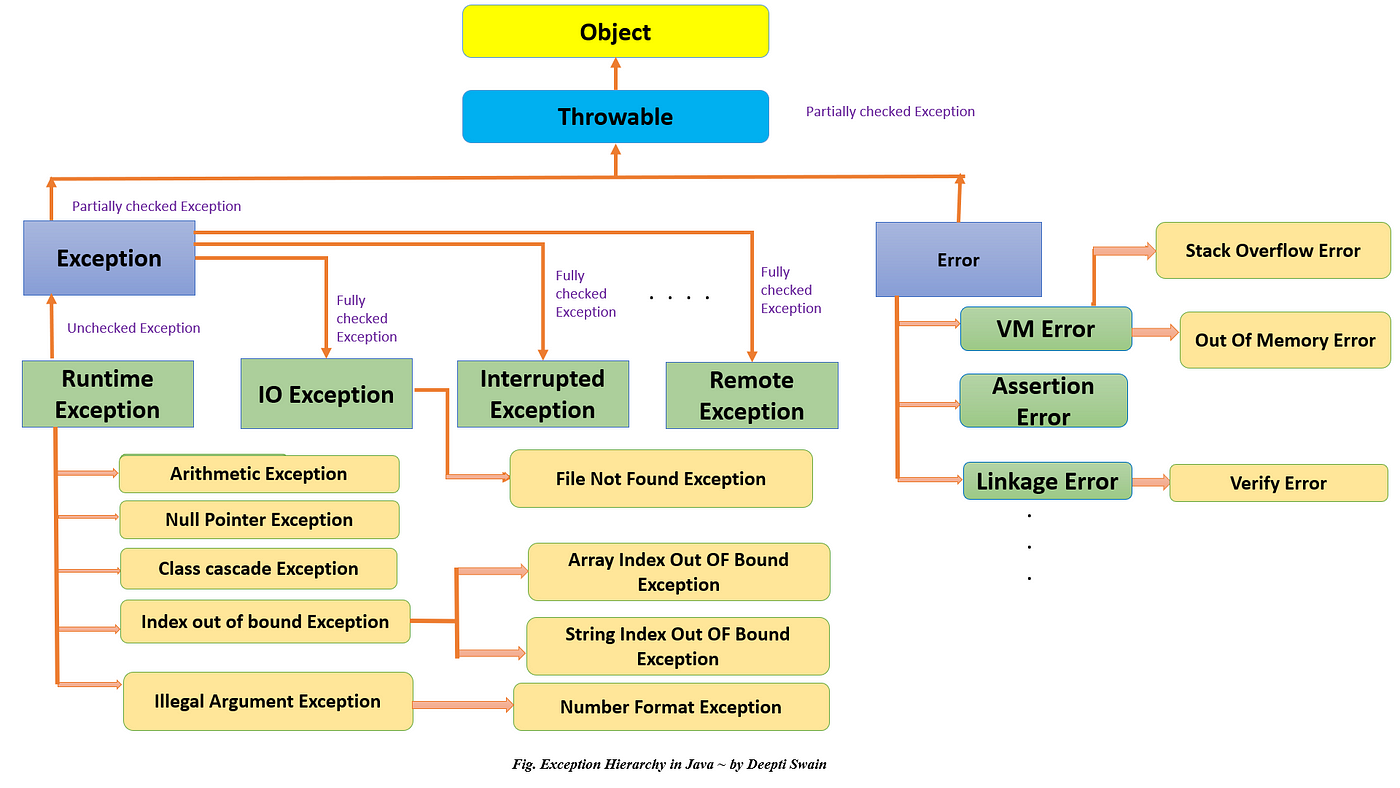
**Exception Hierarchy:**

* Abnormal termination of program is as **Exception.**



**Checked/Compile Time vs Un-Checked/Runtime Exceptions:**

**Checked Exceptions:**

* There are some exceptions which needs to be take care of exception handling must and should because we know that there is a chance of occurring exception.
* Because of exception handling must and should Java Sun micro system people forces to check/handle these exceptions at compile time.
* For Checked Exception it is recommend to use **Throws** because

**Examples:** Network related, Database connections, File related and Threads communicate related etc.

**Un-checked Exceptions:**

There are some exceptions which we don’t needs to be take care of exception handling because we don’t know that there is a chance of occurring exception.

Because of exception handling is not mandatory and java people won’t make forces to handle/check the exceptions either compile/run time. These kind of exceptions are called as “Run Time Exceptions”

**Examples**: Logical related during execution of code etc.

**Exception Handling:**

* Generally, to handle the exception in two ways:

1. Try-Catch
2. Throws

* **Throw** is just to create the exception manually and it will look for handling either try-catch or throws
* Both checked and unchecked exceptions can be thrown using the throw keyword. When an exception is thrown using the throw keyword, the flow of execution of the program is stopped and the control is transferred to the nearest enclosing try-catch block that matches the type of exception thrown.

Which way is recommended to handle the exceptions in java:

* We can’t decide the which is the best way to handle because it’s purely depends on the requirement base.
* When you want to come out/step out the when you arrive the abnormal condition during execution of code recommend to use Throws to handle the exceptions. Such kind of exceptions could be Checked Exceptions (Network/File/Resource Database connections, Threads communications).

**What are the legal combinations of try catch and finally blocks?**

1. **Simple try catch : possible combinations of try catch finally**

|  |
| --- |
| try {    } catch (Exception e) {  // TODO: handle exception  } |

1. **try with multiple catch : possible combinations of try catch finally:**

|  |
| --- |
| try {  // Code that may throw an exception  } catch (ExceptionType1 ex1) {  // Handle ExceptionType1  } catch (ExceptionType2 ex2) {  // Handle ExceptionType2  }  } catch (ExceptionType2 ex2) {  // Handle ExceptionType2  } |

1. **try catch and finally : possible combinations of try catch finally :**

|  |
| --- |
| try {  // Code that may throw an exception  } catch (Exception e) {  // Handle the exception  System.err.println("An exception occurred: " + e.getMessage());  // Additional error handling logic  } finally {  // Cleanup operations or resource release  System.out.println("Finally block executed");  // Additional finally block logic  } |

1. **two try-catch blocks one after the other : possible combinations of try catch finally**

|  |
| --- |
| try {  // Code that may throw an exception  } catch (Exception e) {  // Handle the exception from the first try block  System.err.println("An exception occurred in the first try block: " + e.getMessage());  // Additional error handling logic for the first try block  }  try {  // Code that may throw an exception  } catch (Exception e) {  // Handle the exception from the second try block  System.err.println("An exception occurred in the second try block: " + e.getMessage());  // Additional error handling logic for the second try block  } |

1. **try with finally : possible combinations of try catch finally**

|  |
| --- |
| **t**ry {  // Code that may throw an exception  } finally {  // Cleanup operations or resource release  System.out.println("Finally block executed");  // Additional finally block logic  } |

1. **try-catch-finally block with multiple catch blocks : possible combinations of try catch finally**

|  |
| --- |
| try {  // Code that may throw an exception  } catch (ArithmeticException e) {  // Handle ArithmeticException  System.err.println("ArithmeticException occurred: " + e.getMessage());  // Additional error handling logic for ArithmeticException  } catch (Exception e) {  // Handle other exceptions  System.err.println("Exception occurred: " + e.getMessage());  // Additional error handling logic for other exceptions  } finally {  // Cleanup operations or resource release  System.out.println("Finally block executed");  // Additional finally block logic  } |

1. **nested try-catch blocks : possible combinations of try catch finally**

|  |
| --- |
| try {  // Outer try block  try {  // Inner try block  // Code that may throw an exception  } catch (Exception e) {  // Handle the exception from the inner try block  System.err.println("An exception occurred in the inner try block: " + e.getMessage());  // Additional error handling logic for the inner try block  }  } catch (Exception e) {  // Handle the exception from the outer try block  System.err.println("An exception occurred in the outer try block: " + e.getMessage());  // Additional error handling logic for the outer try block  } |

1. **try-catch block with a nested try-catch block in catch block : possible combinations of try catch finally**

|  |
| --- |
| try {  // Outer try block  // Code that may throw an exception  } catch (Exception e) {  // Handle the exception from the outer try block  System.err.println("An exception occurred in the outer try block: " + e.getMessage());  // Additional error handling logic for the outer try block    try {  // Nested try block  // Code that may throw an exception  } catch (Exception y) {  // Handle the exception from the nested try block  System.err.println("An exception occurred in the nested try block: " + y.getMessage());  // Additional error handling logic for the nested try block  }  } |

1. **try-catch block followed by a finally block, which itself contains a nested try-catch block : possible combinations of try catch finally**

|  |
| --- |
| try {  // Code that may throw an exception  } catch (Exception e) {  // Handle the exception from the outer try block  System.err.println("An exception occurred in the outer try block: " + e.getMessage());  // Additional error handling logic for the outer try block  } finally {  // Cleanup operations or resource release  System.out.println("Finally block executed");  try {  // Nested try block within the finally block  // Code that may throw an exception  } catch (Exception e) {  // Handle the exception from the nested try block  System.err.println("An exception occurred in the nested try block within the finally block: " + e.getMessage());  // Additional error handling logic for the nested try block within the finally block  }  } |

**Custom Exceptions:**

**What is custom exception?**

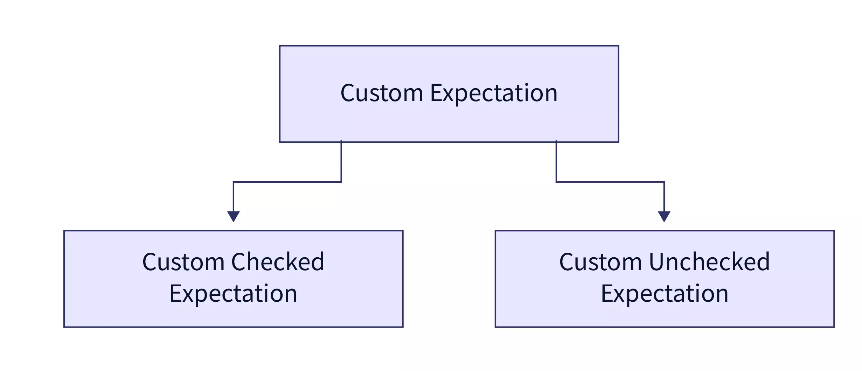
Custom or User-defined Exceptions are designed as per the requirement of the program.

**Why custom exception?**

Using custom Exception, we can have our own Exception and a meaningful message explaining the cause of the exception. We can create an exception by extending the **Exception or RuntimeException** class in our own/user Exception class.

Types of Custom Exception in Java, custom Exceptions are of two types.

* Custom Checked Exceptions
* Custom Unchecked Exceptions



**Custom Checked Exceptions:**

* Checked Exceptions are the exceptions that are detected at the time of compilation. These must be handled explicitly by the developers. The code won't compile only if these exceptions are not taken care of in other words if we ignore these exceptions (not handled with try/catch or throw the exception) then a compilation error occurs. These are caused by unexpected conditions outside the control of code like wrong input, file I/O error, etc.

|  |
| --- |
| import java.io.FileInputStream;  import java.io.FileNotFoundException;  // Exception class  class FileException extends Exception{  FileException(String message, Throwable th){  super(message,th);  }  }  public class FileJava {  public static void main(String[] main){  FileInputStream myFile;  //trying to open file  try {  myFile = new FileInputStream("E://myFolder//tempFile.txt");  }catch (FileNotFoundException e) {  //if not present, throw custom exception  try{  throw new FileException("PLease check File Name or path", e);  }catch(Exception fileE){  //getting message and stack Trace for exception  fileE.getMessage();  fileE.printStackTrace();  }  }  }  } |

* FileException extends the Exception class. Its constructor method takes the message and Throwable as arguments. And we call super with both these parameters, thus we can print a custom message and also share the actual Exception FileNotFoundException if the file is missing. Class FileJava has the main method that has FileInputStream myFile. Now we try to give a path to Stream for the file in the try-catch block. If the file exists, try block will execute. In case the file is not present, inside the catch block we throw a custom exception that makes the program more readable. We print StackTrace and message in the output.
* It is a checked exception as it identifies if the file is present or not during compilation only.
* In the output, we observe FielException and its message. It also prints StackTrace, that at line 16 we are trying to initialize myFile with the path that doesn't exist. Thus, we throw FileException which is caused due to FileNotFoundException.

**Custom Unchecked Exception**

* Unchecked exceptions are exceptions that cannot be detected during compilation. The compiler checks the code in advance for the presence of any potential exceptions. These exceptions occur at runtime and can only be detected at runtime.
* Let us take an example for Custom Unchecked Exception. Let us assume that there is a list of products from which the consumer is trying to fetch some item using the product key. Now if the product key is greater than the number of items in the product's list then we throw the ProductNotFound Exception. This also explains Consumer the cause of the exception.

|  |
| --- |
| // ProductNotFound Exception Class  class ProductNotFound extends Exception{  static String notFound = "Product Key Invalid, Try some other Key.";  public ProductNotFound(){  super(notFound);  }  }  public class Products {  // list of products  static String[] ProductsList = {"Phone", "Washing Machine", "Geyser", "Television", "Laptop"};  static int validKey = ProductsList.length;  // bring method  // if product key is valid it returns the product  public static void bring(int productNo) throws ProductNotFound {  if(productNo>=validKey || productNo<0)  // throws exception if the key is invalid  throw new ProductNotFound();  else System.out.println(ProductsList[productNo-1]);  }  // fetching products from products list  public static void main(String[] args) throws ProductNotFound {  System.out.println("Fetching Some Products...");  bring(2);  bring(6);  }  } |

* **ProductNotFound** is a class that extends the Exception class. notFound String explains the reason that the product key is invalid. The Constructor method takes no parameter. We pass the notFound String to the superclass.
* Superclass prints the custom message at the time of the exception. Products class defines ProductsList which contains various products.bring() method simply returns Products at Index ProductNo-1 also called Product-key from the ProductsList array if the product-key is in the range. bring() method throws a ProductNotFound Exception if it is not valid. This is called a Checked Exception as we can only know during runtime if we have a product at a particular product-key index. In the main method, we try to fetch some random products.

**Conclusion**:

* Custom or User-defined Exceptions are designed as per the requirement of the program. Using custom Exception we can have our own Exception and a meaningful message explaining the cause of the exception.
* We can create an exception by extending the **Exception or RuntimeException** class in our own Exception class. We can use the Exception.getMessage() method to understand the cause of the exception and Exception.printStackTrace() to know the root of the exception.
* Exceptions are of two types, one that can be detected during compilation known as Checked Exceptions, and the other that can be detected only during runtime called Unchecked Exceptions.

**Custom Checked vs Custom Unchecked:**

If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the **Exception** class. If you want to write a runtime exception, you need to extend the **RuntimeException** class

Reference Link :

<https://www.scaler.com/topics/custom-exception-in-java/>

<https://medium.com/@aedemirsen/spring-boot-global-exception-handler-842d7143cf2a#:~:text=When%20an%20exception%20occurs%2C%20the,error%20messages%20to%20the%20clients>.

**Why SpringBootExceptions are popular than Java Exceptions and why?**

Spring AOP will help you to navigate to Global Exception handler class (@ControllerAdvice) with help of aspect oriented programming.

**Why Spring boot exception Handling is recommended than writing custom exceptions?**

Reference: <https://medium.com/@AlexanderObregon/the-art-of-handling-exceptions-with-exceptionhandler-in-spring-mvc-dfb956d84aef>

* **Errors** and **exceptions** are an **inevitable** part of software development.
* Exception handling is one of the critical aspects of ensuring that your application gracefully recovers from unexpected conditions and provides a user-friendly experience. In the world of Spring MVC, the @**ExceptionHandler** annotation provides a seamless way to manage such unwanted scenarios.
* This article dives deep into how you can leverage the power of @**ExceptionHandler** to handle exceptions efficiently in your Spring MVC application.

**Understanding the Need for Exception Handling:**

* Imagine this: Your web application has a form, and users can input data and submit it.
* However, if the user inputs invalid data, the application crashes, presenting the user with an unsightly error page.
* This not only disrupts the user experience but may also cause a loss of trust in your application.

**That’s where exception handling comes into play.**

* Instead of letting the application crash, you can catch the error and redirect the user to a custom error page, maybe even provide them with suggestions on how to correct their mistake.
* Instead of letting the application crash, you can catch the error and redirect the user to a custom error page, maybe even provide them with suggestions on how to correct their mistake.

**@ExceptionHandler**

* Spring MVC provides a convenient annotation called @ExceptionHandler that you can use to define methods to handle specific exceptions. These methods act as centralized handlers for exceptions thrown within your controller.

**How to Use @ExceptionHandler**

Here’s a basic example. Let’s assume we want to handle a NullPointerException in our controller.

|  |
| --- |
| @Controller  public class MyController {  @RequestMapping("/someEndpoint")  public String someEndpoint() {  // Some logic that might throw a NullPointerException  return "viewName";  }  **@ExceptionHandler(NullPointerException.class)**  public ModelAndView handleNullPointerException(Exception ex) {  ModelAndView model = new ModelAndView("error");  model.addObject("exception", ex);  return model;  }  } |

In the above code, if a **NullPointerException** is thrown within the controller, the handleNullPointerException method will be invoked. We can then redirect the user to an error page and display the relevant error message.

**Handling Multiple Exceptions:**

* Sometimes, you might want to handle multiple exceptions in a similar fashion.
* You can do this by providing multiple exception types to the @ExceptionHandler annotation:

|  |
| --- |
| @ExceptionHandler(**{ArithmeticException.class, NullPointerException.class})**  public ModelAndView handleMultipleExceptions(Exception ex) {  // Handle the exception and return the model and view  } |

**Global Exception Handling**

* In larger applications, it’s common to have multiple controllers. Instead of defining the same exception handlers in each controller, you can use @ControllerAdvice to handle exceptions globally.

|  |
| --- |
| @ControllerAdvice  public class GlobalExceptionHandler {  @ExceptionHandler(NullPointerException.class)  public ModelAndView handleNullPointerException(Exception ex) {  // Handle the exception globally for all controllers  }  } |

* With @ControllerAdvice, any exception thrown in any controller that matches the exception types specified in the @ExceptionHandler will be caught and handled.

**Customizing the Response:**

* With the combination of @ExceptionHandler and @ResponseStatus, you can also customize the HTTP status code of the response when an exception occurs.

|  |
| --- |
| @ExceptionHandler(ResourceNotFoundException.class)  **@ResponseStatus(HttpStatus.NOT\_FOUND)**  public ModelAndView handleResourceNotFound(Exception ex) {  // Return custom 404 Not Found page  } |

In the above code, if a **ResourceNotFoundException** is thrown, the response will have a **404 Not Found HTTP status.**

**Benefits of Using @ExceptionHandler:**

* **Centralized Exception Handling:** With @ExceptionHandler, you have a centralized place to handle exceptions, which makes your code cleaner and more maintainable.
* **Custom Error Responses:** You can provide custom error pages or messages, enhancing the user experience.
* **Enhanced Application Robustness**: By handling exceptions gracefully, you ensure that minor errors don’t crash your entire application.

**Exception handling** is a crucial aspect of building robust and user-friendly applications. Spring MVC’s @ExceptionHandler provides developers with a powerful and flexible tool to handle exceptions seamlessly. Whether you're catching exceptions at the controller level or globally with @ControllerAdvice, @ExceptionHandler ensures that your application can recover gracefully from unexpected situations. Embrace the art of exception handling, and elevate your Spring MVC application's resilience and user experience.

**Advantages of using Spring Exception Handling:**

* Centralized Exception Handling: With @ExceptionHandler, you have a centralized place to handle exceptions, which makes your code cleaner and more maintainable.
* @ExceptionHandler works at the Controller level and it is only active for that particular Controller, not globally for the entire application.
* One of the main advantages of this approach is that it centralizes the error handling logic and promotes code reusability. Developers don’t have to add try-catch blocks in every method, and they can reuse the same exception handling logic across multiple controllers.
* HandlerExceptionResolver:This will resolve any exception thrown by the application. It is used to resolve standard Spring exceptions to their corresponding HTTP Status Codes. It does not have control over the body of the response, means it does not set anything to the body of the Response.It does map the status code on the response but the body is null.
* @ControllerAdvice used for global error handling in the Spring MVC application.It also has full control over the body of the response and the status code.