**Exception Handling**

**What is exception handling?**

Exception handling in java is a mechanism to handle unwanted interruption in program to that that normal flow if the program execution continues.

In java we can use try catch blocks and use keywords like finally, throws, throw to handle exceptions.

jvm by default handles exceptions, when an exception is raised it will halt the execution of the program and throw the occurred exception.

**What are Java Exceptions?**

In Java, Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program’s instructions and terminate the program abruptly.

There are various types of interruptions while executing any program like errors, exceptions, and bugs. These interruptions can be due to programming mistakes or due to system issues. Depending on the conditions they can be classified as errors and exceptions.

Java has classes that are used to handle built-in exceptions and provision to create user-defined exceptions.

Errors & Exceptions

**1. Exception**

* As mentioned earlier exceptions are unwanted conditions that disrupt the flow of the program.
* Exceptions usually occur due to the code and can be recovered.
* Exceptions can be of both checked(exceptions that are checked by the compiler) and unchecked(exceptions that cannot be checked by the compiler) type.
* They can occur at both run time and compile time.
* In Java, exceptions belong to java.lang.Exception class.

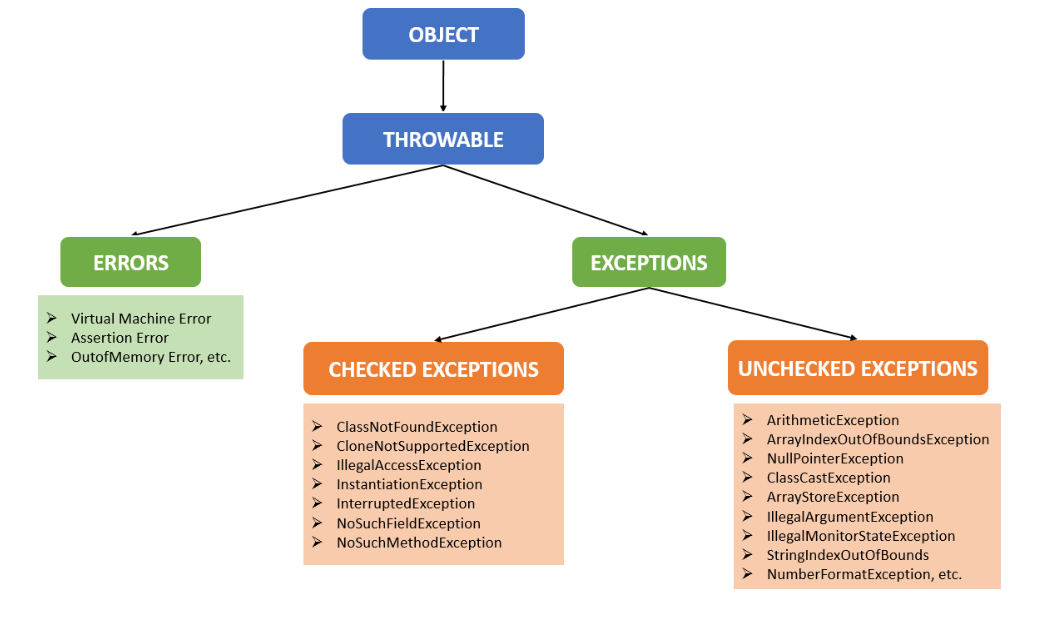
**2. Error**

* An error is also an unwanted condition but it is caused due to lack of resources and indicates a serious problem.
* Errors are irrecoverable, they cannot be handled by the programmers.
* Errors are of unchecked type only.
* They can occur only at run time.
* In java, errors belong to java.lang.error class.
* Eg: OutOfMemmoryError.

**Exception Hierarchy**

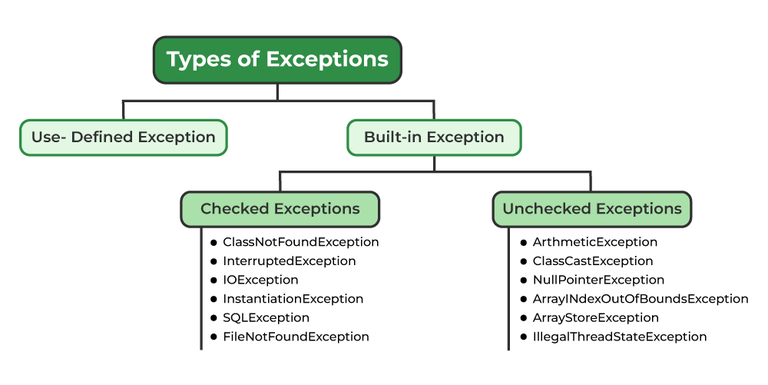
All the exceptions are the subclass of Throwable class and they are present in java.lang package.

Throwable is the base of exception handling hierarchy in java. This class is further extended in different classes like exception, error, and so on.



**Types of Exceptions**

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



[**Built-in Exceptions**](https://www.geeksforgeeks.org/built-exceptions-java-examples/)**:**

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations

**Checked Exception**

* These are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using the [throws keyword](https://www.geeksforgeeks.org/throw-throws-java/).
* In checked exception, there are two types: fully checked and partially checked exceptions. A fully checked exception is a checked exception where all its child classes are also checked, like IOException, InterruptedException.
* A partially checked exception is a checked exception where some of its child classes are unchecked, like Exception.

**Unchecked Exceptions**

* These are the exceptions that are not checked at compile time. In Java, exceptions under Error and RuntimeException classes are unchecked exceptions.

**Difference between checked and unchecked exception**

|  |  |
| --- | --- |
| **Checked/Compile time Exception** | **Unchecked/runtime Exception** |
| Checked exception are the exception which are handle at the compile time | Unchecked exception are the exception that are not able to checked at compiled time |
| Program gives compilation error if method throws a checked exception | Program compile fine because compiler is not able to check the exception |
| If some code within the method throws a checked exception then the method must specify the exception using throws keyword | Method is not forced by compiler to declare the unchecked exception thrown by its implementation generally, such methods almost do not declare them. |
| Checked exception are occur when chances of failure are too high | Unchecked exception are occur mostly due to programming mistake. |
| They are direct subclass of exception class and extends Exception class | Direct subclass of RuntimeException and extends RunTimeException class. |

**How JVM handle exception**

Whenever an exception occurs at particular line then method associated with that line will create object that object is called exception object.

JVM asked to main method have you handle exception if no then JVM pass exception to Default Exception Handler

And it prints error and terminate main method abnormally.

method abnormally.

Note:

There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of the methods is called Call Stack

The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an Exception handler.

The run-time system starts searching from the method in which the exception occurred, and proceeds through the call stack in the reverse order in which methods were called.

If it finds an appropriate handler, then it passes the occurred exception to it. An appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.

If the run-time system searches all the methods on the call stack and couldn’t have found the appropriate handler, then the run-time system handover the Exception Object to the default exception handler, which is part of the run-time system. This handler prints the exception information in the following format and terminates the program abnormally.

**Methods of Throwable class**

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public String getMessage() | Returns the detail message of the exception as provided during the exception creation |
| public String getLocalizedMessage() | Returns a localized description of the exception if localization is provided for that exception. If no localized message is available, it defaults to the result of getMessage(). |
| public void printStackTrace() | Prints this exception and backtraces it to the standard error stream. |
| public StackTraceElement[] getStackTrace() | Provides programmatic access to the stack trace information printed by printStackTrace(). |
| public String toString() | Returns a short description of exception |

**How Programmer handle an Exception**

By using below blocks and keywords we can handle exception in program.

* try
* catch
* finally
* throw
* throws

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| try { |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| // here we write risky code which may throw exception | | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |
| }catch (ExceptionType ex){ | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| // this will only execute if any exception occurs. | | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |
| }catch (GenericExcpetion ex) | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| // this will only execute if any exception occurs. | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |
| finally { | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| // weather exception occur or this block execute by default. it is the block which always execute, we write clean up code in this block | | | | | | | | | | | | | | | | | | | |
| eg: db connection  } | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Eg:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| public class FileHandlingExample { | | | | |  |  |  |  |  |  |  |  |  |
| public static void main(String[] args) { | | | | | | |  |  |  |  |  |  |  |
| FileReader reader = null; | | | | |  |  |  |  |  |  |  |  |  |
| try { | |  |  |  |  |  |  |  |  |  |  |  |  |
| File file = new File("example.txt"); | | | | | | |  |  |  |  |  |  |  |
| reader = new FileReader(file); | | | | | |  |  |  |  |  |  |  |  |
| int character; | | | |  |  |  |  |  |  |  |  |  |  |
| while ((character = reader.read())!=-1){ | | | | | | | |  |  |  |  |  |  |
| System.*out*.print((char) character); | | | | | | | |  |  |  |  |  |  |
| } | |  |  |  |  |  |  |  |  |  |  |  |  |
| } catch (IOException e) { | | | | |  |  |  |  |  |  |  |  |  |
| System.*out*.println("An error occurred while reading the file :: "+ e.getMessage()); | | | | | | | | | | | | | |
| } catch (Exception ex){ | | | | |  |  |  |  |  |  |  |  |  |
| System.*out*.println("An error occurred in program :: "+ex.getMessage()); | | | | | | | | | | | |  |  |
| } | |  |  |  |  |  |  |  |  |  |  |  |  |
| finally { | | |  |  |  |  |  |  |  |  |  |  |  |
| if(reader != null){ | | | | |  |  |  |  |  |  |  |  |  |
| try { | | |  |  |  |  |  |  |  |  |  |  |  |
| reader.close(); | | | | |  |  |  |  |  |  |  |  |  |
| } catch (IOException e) { | | | | | |  |  |  |  |  |  |  |  |
| System.*out*.println("Failed to close the file reader :: " + e.getMessage()); | | | | | | | | | | | | | |
| } | | |  |  |  |  |  |  |  |  |  |  |  |
| } | |  |  |  |  |  |  |  |  |  |  |  |  |
| } | |  |  |  |  |  |  |  |  |  |  |  |  |
| } |  |  |  |  |  |  |  |  |  |  |  |  |  |
| } |  |  |  |  |  |  |  |  |  |  |  |  |  |

**Difference between final, finally and finalise()**

final is a keyword, which is use with variable, method, class.

***final***

* Variable declare with final we cannot change value of that variable, method declare with final cannot be overridden and class declare with final cannot be subclass.

***finally***

finally is a block which is use with exception handling, is mainly use for writing clean up code, like closing file and db resources, finally block always executes irrespective of exception occur or not.

finally block does not executes in below situations.

* If the program terminates using System.exit() before reaching the finally block, the finally block will not execute. This is because System.exit() halts the entire JVM.
* If a thread running the try-catch-finally block is **forcefully killed** (e.g., using Thread.stop()), or it encounters a critical system-level failure like power failure, the finally block might not be executed. However, Thread.stop() is deprecated due to its unsafe behavior.
* If the JVM encounters a fatal error like StackOverflowError, OutOfMemoryError, or any critical system failure, the finally block may not execute because the JVM cannot recover from such errors.
* If the program is terminated forcefully (e.g., by killing the process or closing the console window), the finally block won't execute because the JVM itself is abruptly stopped.

***finalise()***

it is a method which is executed just before garbage collector, the object which are going to delete the resources related to that objects are closed inside finalize method

Executed just before GC.

**Try-with-resource**

In Java, the **try-with-resources** statement is a feature that automatically closes resources when they are no longer needed. It's an improvement over the traditional try-catch-finally block, specifically for managing resources like file streams, which must be closed to prevent resource leaks.

The resource which implements AutoClosable interface are written inside try block, resource closed automatically when they are no longer use.

Eg: FileReader, BufferReader.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| public class TryWithResourcesExample { | | | | | |  |  |  |  |  |  |  |  |
| public static void main(String[] args) { | | | | | | |  |  |  |  |  |  |  |
| try(FileReader reader = new FileReader("example.txt")){ | | | | | | | | |  |  |  |  |  |
| int character; | | | |  |  |  |  |  |  |  |  |  |  |
| while ((character = reader.read())!=-1){ | | | | | | | |  |  |  |  |  |  |
| System.*out*.print((char) character); | | | | | | | |  |  |  |  |  |  |
| } | |  |  |  |  |  |  |  |  |  |  |  |  |
| } catch (IOException e) { | | | | |  |  |  |  |  |  |  |  |  |
| System.*out*.println("An error occurred while reading the file :: "+ e.getMessage()); | | | | | | | | | | | | | |
| } | |  |  |  |  |  |  |  |  |  |  |  |  |
| } |  |  |  |  |  |  |  |  |  |  |  |  |  |
| } |  |  |  |  |  |  |  |  |  |  |  |  |  |

throw keyword

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either [checked or unchecked exception](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/). The throw keyword is mainly used to throw custom exceptions.

* here throw keyword create exception object and that object pass to JVM and default exception handler print error
* we can throw either checked or unchecked exception but throw is best for customized exception
* we cannot write any statement after throw, otherwise it will provide unreachable statement error.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| public class ThrowExample { | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| public static void main(String[] args) { | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |
| /\*try { | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.out.println(validateAge(17)); | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |
| } | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| catch (ValidateAgeException e){ | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.out.println(e.getMessage()); | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |
| } | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.out.println("Next statement!");\*/ | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| try { | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.*out*.println(*validateAge*(19)); | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |
| }catch (ValidateAgeRunTimeException ex){ | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |
| System.*out*.println(ex.getMessage()); | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |
| } | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| } |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| /\*public static String validateAge(int age) throws ValidateAgeException { // if we  are using custom exception by extending Exception class | | | | | | | | | | | | | | | | | | | |
| if(age<18){ | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| throw new ValidateAgeException("You are not eligible to vote!"); | | | | | | | | | | |  |  |  |  |  |  |  |  |  |
| }else { | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| return "Eligible to vote!"; | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| } | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| }\*/ | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| public static String validateAge(int age){ | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |
| if(age<18){ | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| throw new ValidateAgeRunTimeException();// if we are using custom exception by  extending RuntimeException class | | | | | | | | | | | | | | | | | |  |  |
| }else { | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| return "Eligible to vote!"; | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| } | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| } |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| } |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

**Throws keyword**

throws is a keyword in Java that is used in the signature of a method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| public class ThrowsExample { | | | |  |  |  |  |  |  |
| public static void main(String[] args) { | | | | | | |  |  |  |
| try(FileReader reader = new FileReader("example.txt")) { | | | | | | | | |  |
| *readFile*(reader); | | | | |  |  |  |  |  |
| }catch (IOException ex){ | | | | |  |  |  |  |  |
| System.*out*.println("An error occurred: " + ex.getMessage()); | | | | | | | | | |
| } | |  |  |  |  |  |  |  |  |
| System.*out*.println("Next statement!"); | | | | | | |  |  |  |
| } |  |  |  |  |  |  |  |  |  |
| public static void readFile(FileReader reader) throws IOException{ | | | | | | | | | |
| int character; | | | |  |  |  |  |  |  |
| while ((character = reader.read())!=-1){ | | | | | | |  |  |  |
| System.*out*.print((char) character); | | | | | | |  |  |  |
| } | |  |  |  |  |  |  |  |  |
| } |  |  |  |  |  |  |  |  |  |
| } |  |  |  |  |  |  |  |  |  |

**Difference between throw and throws**

|  |  |
| --- | --- |
| **throw** | **throws** |
| It is use to throw an exception explicitly. | It is use to declare that a method might throw an exception. |
| it use inside method body | it use with method signature |
| it can throw only one exception | it can throw multiple exception |
| throw new ValidateAgeRunTimeException() | throws IOException |

**Custom Exception**

The exception which are user defined are known as custom exception, they are created by extending Exception class for checked and RuntimeException for unchecked exceptions.

|  |  |
| --- | --- |
| public class ValidateAgeException extends Exception{ | |
| public ValidateAgeException(){ |  |
| super("Not eligible to vote!"); |  |
| } |  |
| public ValidateAgeException(String message){ | |
| super(message); |  |
| } |  |
| } |  |
|  |  |
| public class ValidateAgeRunTimeException extends RuntimeException{ | |
| public ValidateAgeRunTimeException(){ | |
| super("Not eligible!"); |  |
| } |  |
| public ValidateAgeRunTimeException(String message){ | |
| super("Not eligible"); |  |
| } |  |
| } |  |

**Spring boot Exception Handling**

**Exception handling in Spring Boot**helps to deal with errors and exceptions present in APIs, delivering a robust enterprise application.

key approaches to exception handling in Spring Boot:

* Default exception handling by Spring Boot
* Using @ExceptionHandler annotation
* Using @ControllerAdvice for global exception handling

**Using @ExceptionHandler Annotation**

* **@ExceptionHandler**annotation provided by Spring Boot can be used to handle exceptions in particular Handler classes or Handler methods.
* Any method annotated with this is automatically recognized by Spring Configuration as an Exception Handler Method.
* An Exception Handler method handles all exceptions and their subclasses passed in the argument.
* It can also be configured to return a specific error response to the user.
* Here we add handler methods inside controller and spring take care if any defined exception occurs.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | @Data  public class ErrorResponse { | | private int statusCode; | | private String message; | | private LocalDateTime timestamp; | | private String messageDescription; | |  | | public ErrorResponse(String message){ | | super(); | | this.message = message; | | } | | } | | } |   @ExceptionHandler(CustomerAlreadyExistsException.class) |
| @ResponseStatus(HttpStatus.*CONFLICT*) |
| public ErrorResponse handleCustomerAlreadyExistsException(CustomerAlreadyExistsException ex, WebRequest request) { |
| return new ErrorResponse(HttpStatus.*CONFLICT*.value(), |
| ex.getMessage(), LocalDateTime.*now*(), request.getDescription(false)); |
| } |
| @ExceptionHandler(NoSuchCustomerExistsException.class) |
| @ResponseStatus(HttpStatus.*NOT\_FOUND*) |
| public ErrorResponse handleNoSuchCustomerExistsException(NoSuchCustomerExistsException ex, WebRequest request) { |
| return new ErrorResponse(HttpStatus.*NOT\_FOUND*.value(), |
| ex.getMessage(), LocalDateTime.*now*(), request.getDescription(false)); |

**Using @ControllerAdvice for Global Exception Handling**

The **@ExceptionHandler**annotated method can only handle exceptions thrown by that particular class. However, if we want to handle any exception thrown throughout the application, we can define a global exception handler class and annotate it with **@ControllerAdvice**. This annotation helps to integrate multiple exception handlers into a single global unit.

|  |
| --- |
| @Data  public class ErrorResponse { |
| private int statusCode; |
| private String message; |
| private LocalDateTime timestamp; |
| private String messageDescription; |
|  |
| public ErrorResponse(String message){ |
| super(); |
| this.message = message; |
| } |
| } |
| } |
|  |
| @RestControllerAdvice |
| public class GlobalExceptionHandler { |
|  |
| @ExceptionHandler(NoSuchCustomerExistsException.class) |
| public ResponseEntity<?> handleException(NoSuchCustomerExistsException ex, WebRequest request){ |
| ErrorResponse errorResponse = new ErrorResponse(HttpStatus.*NOT\_FOUND*.value(), |
| ex.getMessage(), LocalDateTime.*now*(),request.getDescription(false)); |
| return new ResponseEntity<>(errorResponse,HttpStatus.*NOT\_FOUND*); |
| } |
|  |
| @ExceptionHandler(CustomerAlreadyExistsException.class) |
| public ResponseEntity<?> handleException(CustomerAlreadyExistsException ex,WebRequest request){ |
| ErrorResponse errorResponse = new ErrorResponse(HttpStatus.*CONFLICT*.value(), |
| ex.getMessage(), LocalDateTime.*now*(),request.getDescription(false)); |
| return new ResponseEntity<>(errorResponse,HttpStatus.*CONFLICT*); |
| } |
| } |