Fundamentals of Java

Session: 12

Exceptions

Objectives

- Describe exceptions
- Explain types of errors and exceptions
- Describe the Exception class
- Describe exception handling
- Explain try-catch block
- Explain finally block
- Explain execution flow of exceptions
- Explain guidelines to exception handling

Introduction

- Java is a very robust and efficient programming language.
- Features such as classes, objects, inheritance, and so on make Java a strong, versatile, and secure language.
- However, no matter how well a code is written, it is prone to failure or behaves erroneously in certain conditions.
- These situations may be expected or unexpected.
- In either case, the user would be nonplussed or confused with such unexpected behavior of code.
- To avoid such a situation, Java provides the concept of exception handling using which, a programmer can display appropriate message to the user in case such unexpected behavior of code occurs.

Exceptions 1-4

An exception is an event or an abnormal condition in a program occurring during execution of a program that leads to disruption of normal flow of program instructions.

- An exception can occur for different reasons such as:
  - when the user enters invalid data
  - a file that needs to be opened cannot be found
  - a network connection has been lost in the middle of communications
  - the JVM has run out of memory
- When an error occurs inside a method, it creates an exception object and passes it to the runtime system.
- This object holds information about the type of error and state of the program when the error occurred.

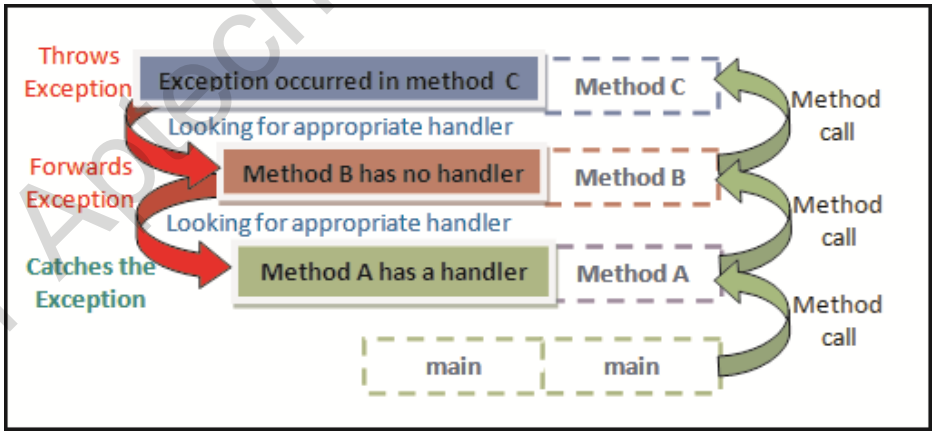
Exceptions 2-4

The process of creating an exception object and passing it to the runtime system is termed as throwing an exception.

- After an exception is thrown by a method, the runtime system tries to find some code block to handle it.
- The possible code blocks where the exception can be handled are a series of methods that were invoked in order to reach the method where the error has actually occurred.

This list or series of methods is called the call stack. The stack trace shows the sequence of method invocations that led up to the exception.

- Following figure shows an example of method call stack:



- The figure shows the method call from **main → Method A → Method B → Method C**.

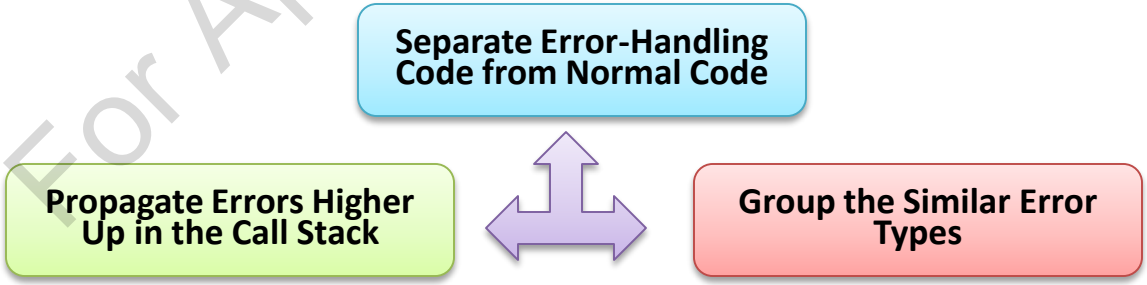
Exceptions 3-4

- When an exception occurs in method C, it throws the exception object to the runtime environment.
- The runtime environment then searches the entire call stack for a method that consists of a code block that can handle the exception.
- This block of code is called an exception handler.
- The runtime environment first searches the method in which the error occurred.
- If a handler is not found, it proceeds through the call stack in the reverse order in which the methods were invoked.
- When an appropriate handler is found, the runtime environment passes the exception to the handler.
- An appropriate exception handler is one that handles the same type of exception as the one thrown by the method.
- In this case, the exception handler is said to 'catch' the exception.

- If while searching the call stack, the runtime environment fails to find an appropriate exception handler, the runtime environment will consequently terminate the program.

Exceptions 4-4

- An exception is thrown for the following reasons:
  - A throw statement within a method was executed.
  - An abnormality in execution was detected by the JVM, such as:
    - Violation of normal semantics of Java while evaluating an expression such as an integer divided by zero.
    - Error occurring while linking, loading, or initializing part of the program that will throw an instance of a subclass of `LinkageError`.
    - The JVM is prevented from executing the code due to an internal error or resource limitation that will throw an instance of a subclass of `VirtualMachineError`.
  - An asynchronous exception occurred.
- The use of exceptions to handle errors offers some advantages as follows:



Types of Errors and Exceptions 1-5

- Java provides the following two types of exceptions:

Checked Exceptions

- These are exceptions that a well-written application must anticipate and provide methods to recover from.
- For example, suppose an application prompts the user to specify the name of a file to be opened and the user specifies the name of a nonexistent file.
- In such a case, the `java.io.FileNotFoundException` is thrown.
- However, a well-written program will have the code block to catch this exception and inform the user of the mistake by displaying an appropriate message.
- In Java, all exceptions are checked exceptions, except those indicated by `Error`, `RuntimeException`, and their subclasses.

Types of Errors and Exceptions 2-5

Unchecked Exceptions

- The unchecked exceptions are as follows:

Error

- These are exceptions that are external to the application.
- The application usually cannot anticipate or recover from errors.
- For example, suppose the user specified correct file name for the file to be opened and the file exists on the system.
- However, the runtime fails to read the file due to some hardware or system malfunction.
- Such a condition of unsuccessful read throws the `java.io.IOError` exception.
- In this case, the application may catch this exception and display an appropriate message to the user or leave it to the program to print a stack trace and exit.
- Errors are exceptions generated by `Error` class and its subclasses.

Types of Errors and Exceptions 3-5

Runtime Exception

- These exceptions are internal to the application and usually the application cannot anticipate or recover from such exceptions.
- These exceptions usually indicate programming errors, such as logical errors or improper use of an API.
- For example, suppose a user specified the file name of the file to be opened.
- However, due to some logical error a `null` is passed to the application, then the application will throw a `NullPointerException`.
- The application can choose to catch this exception and display appropriate message to the user or eliminate the error that caused the exception to occur.
- Runtime exceptions are indicated by `RuntimeException` class and its subclasses.

Errors and runtime exceptions are collectively known as unchecked exceptions.

In Java, `Object` class is the base class of the entire class hierarchy.

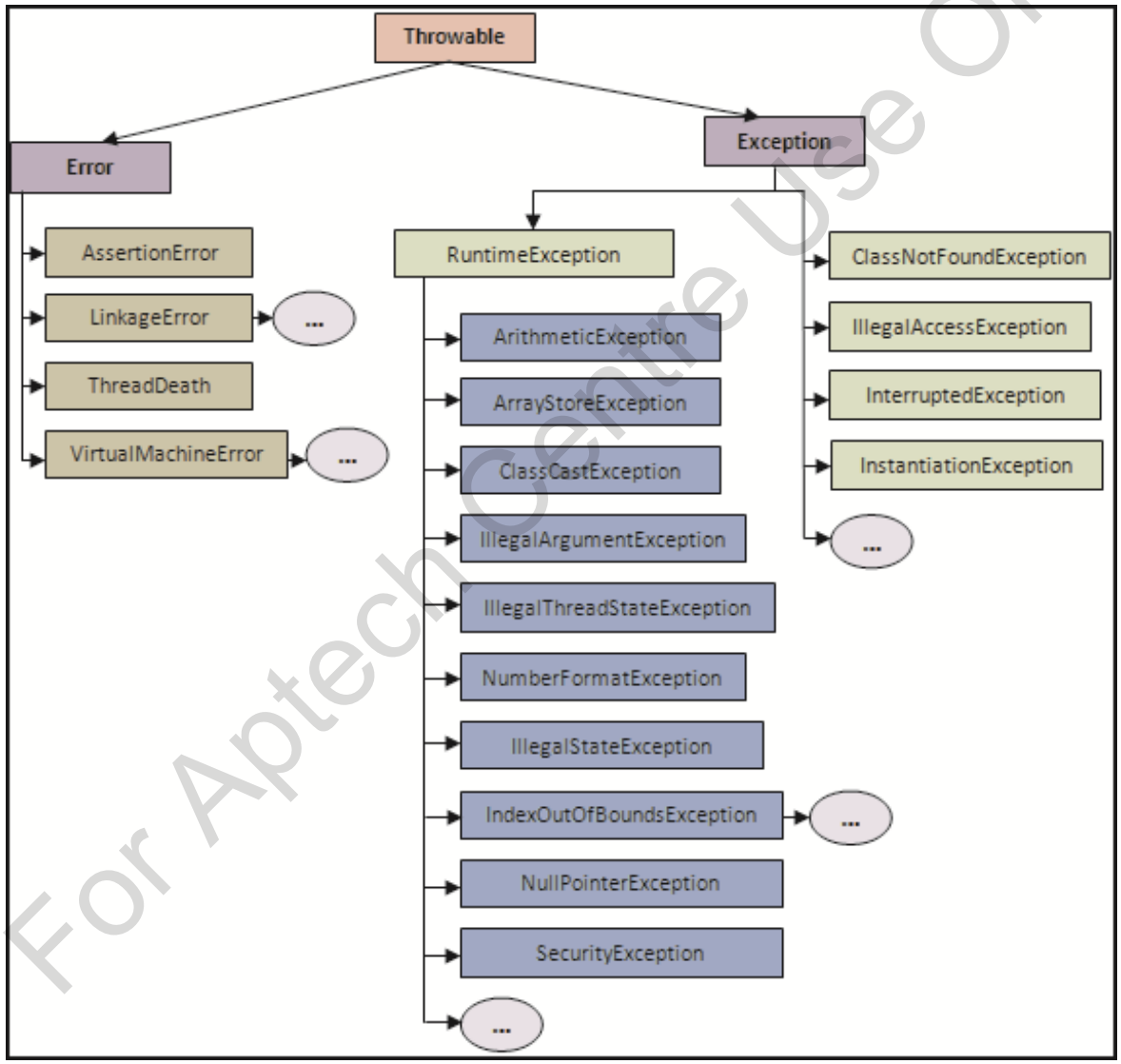
`Throwable` class is the base class of all the exception classes.

`Object` class is the base class of `Throwable`.

`Throwable` class has two direct subclasses namely, `Exception` and `Error`.

Types of Errors and Exceptions 4-5

- The `Throwable` class hierarchy is shown in the following figure:



Types of Errors and Exceptions 5-5

- Following table lists some of the checked exceptions:

Exception	Description
`InstantiationException`	Occurs upon an attempt to create instance of an `abstract` class.
`InterruptedException`	Occurs when a thread is interrupted.
`NoSuchMethodException`	Occurs when JVM is unable to resolve which method to be invoked.

- Following table lists some of the commonly observed unchecked exceptions:

Exception	Description
`ArithmeticException`	Indicates an arithmetic error condition.
`ArrayIndexOutOfBoundsException`	Occurs if an array index is less than zero or greater than the actual size of the array.
`IllegalArgumentException`	Occurs if method receives an illegal argument.
`NegativeArraySizeException`	Occurs if array size is less than zero.
`NullPointerException`	Occurs on access to a `null` object member.
`NumberFormatException`	Occurs if unable to convert the string to a number.
`StringIndexOutOfBoundsException`	Occurs if index is negative or greater than the size of the string.

Exception Class 1-3

The class `Exception` and its subclasses indicate conditions that an application might attempt to handle.

The `Exception` class and all its subclasses except `RuntimeException` and its subclasses, are checked exceptions.

- The checked exceptions must be declared in a method or constructor's `throws` clause if the method or constructor is liable to throw the exception during its execution and propagate it further in the call stack.
- Following code snippet displays the structure of the `Exception` class:

```
public class Exception extends Throwable
{
   …
}
```

Exception Class 2-3

- Following table lists the constructors of `Exception` class:

Exception Class Constructor	Description
`Exception()`	Constructs a new exception with error message set to `null`.
`Exception(String message)`	Constructs a new exception with error message set to the specified string `message`.
`Exception(String message, Throwable cause)`	Constructs a new exception with error message set to the specified strings `message` and `cause`.
`Exception(Throwable cause)`	Constructs a new exception with the specified `cause`. The error message is set as per the evaluation of `cause == null?null:cause.toString()`. That is, if `cause` is `null`, it will return `null`, else it will return the `String` representation of the message. The message is usually the class name and detail message of `cause`.

Exception Class 3-3

- `Exception` class provides several methods to get the details of an exception.
- Following table lists some of the methods of `Exception` class:

Exception Class Method	Description
`public String getMessage()`	Returns the details about the exception that has occurred.
`public Throwable getCause()`	Returns the cause of the exception that is represented by a `Throwable` object.
`public String toString()`	If the `Throwable` object is created with a message string that is not `null`, it returns the result of `getMessage()` along with the name of the exception class concatenated to it. If the `Throwable` object is created with a `null` message string, it returns the name of the actual class of the object.
`public void printStackTrace()`	Prints the result of the method, `toString()` and the stack trace to `System.err`, that is, the error output stream.
`public StackTraceElement [] getStackTrace()`	Returns an array where each element contains a frame of the stack trace. The index 0 represents the method at the top of the call stack and the last element represents the method at the bottom of the call stack.
`public Throwable fillInStackTrace()`	Fills the stack trace of this `Throwable` object with the current stack trace, adding to any previous information in the stack trace.

Handling Exceptions in Java 1-2

Any exception that a method is liable to throw is considered to be as much a part of that method's programming interface as its parameters and return value.

The code that calls a method must be aware about the exceptions that a method may throw.

This helps the caller to decide how to handle them if and when they occur.

More than one runtime exceptions can occur anywhere in a program.

Having to add code to handle runtime exceptions in every method declaration may reduce a program's clarity.

Thus, the compiler does not require that a user must catch or specify runtime exceptions, although it does not object it either.

Handling Exceptions in Java 2-2

- A common situation where a user can throw a `RuntimeException` is when the user calls a method incorrectly.
- For example, a method can check beforehand if one of its arguments is incorrectly specified as `null`.
- In that case, the method may throw a `NullPointerException`, which is an unchecked exception.
- Thus, if a client is capable of reasonably recovering from an exception, make it a checked exception.
- If a client cannot do anything to recover from the exception, make it an unchecked exception.

try-catch Block 1-5

- The first step in creating an exception handler is to identify the code that may throw an exception and enclose it within the `try` block.
- The syntax for declaring a `try` block is as follows:

Syntax

```
try{
   // statement 1
   // statement 2
}
```

- The statements within the `try` block may throw an exception.
- Now, when the exception occurs, it is trapped by the `try` block and the runtime looks for a suitable handler to handle the exception.
- To handle the exception, the user must specify a `catch` block within the method that raised the exception or somewhere higher in the method call stack.

try-catch Block 2-5

- The syntax for declaring a `try-catch` block is as follows:

Syntax

```
try{
   // statements that may raise exception
   // statement 1
   // statement 2
}
catch(<exception-type> <object-name>){
   // handling exception
   // error message
}
```

where,

`exception-type`: Indicates the type of exception that can be handled.

`object-name`: Object representing the type of exception.

- The `catch` block handles exceptions derived from `Throwable` class.

try-catch Block 3-5

- Following code snippet demonstrates an example of `try` with a single `catch` block:

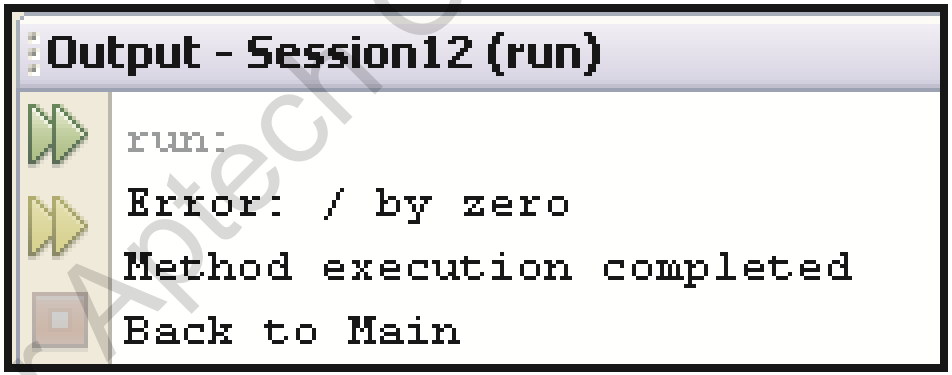
```
package session12;
class Mathematics {
  /**
   * Divides two integers
   * @param num1 an integer variable storing value of first number
   * @param num2 an integer variable storing value of second number
   * @return void
   */
  public void divide(int num1, int num2) {
    // Create the try block
    try {
      // Statement that can cause exception
      System.out.println("Division is: " + (num1/num2));
    }
    catch(ArithmeticException e){ //catch block for ArithmeticException
      // Display an error message to the user
      System.out.println("Error: "+ e.getMessage());
    }
    // Rest of the method
    System.out.println("Method execution completed");
  } }
```

try-catch Block 4-5

```
/**
 * Define the TestMath.java class
 */
public class TestMath {
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args)
  {
    // Check the number of command line arguments
    if(args.length==2) {
      // Instantiate the Mathematics class
      Mathematics objMath = new Mathematics();
      // Invoke the divide(int,int) method
      objMath.divide(Integer.parseInt(args[0]),
       Integer.parseInt(args[1]));
    }
    else {
      System.out.println("Usage: java Mathematics <number1> <number2>");
    }
  }
}
```

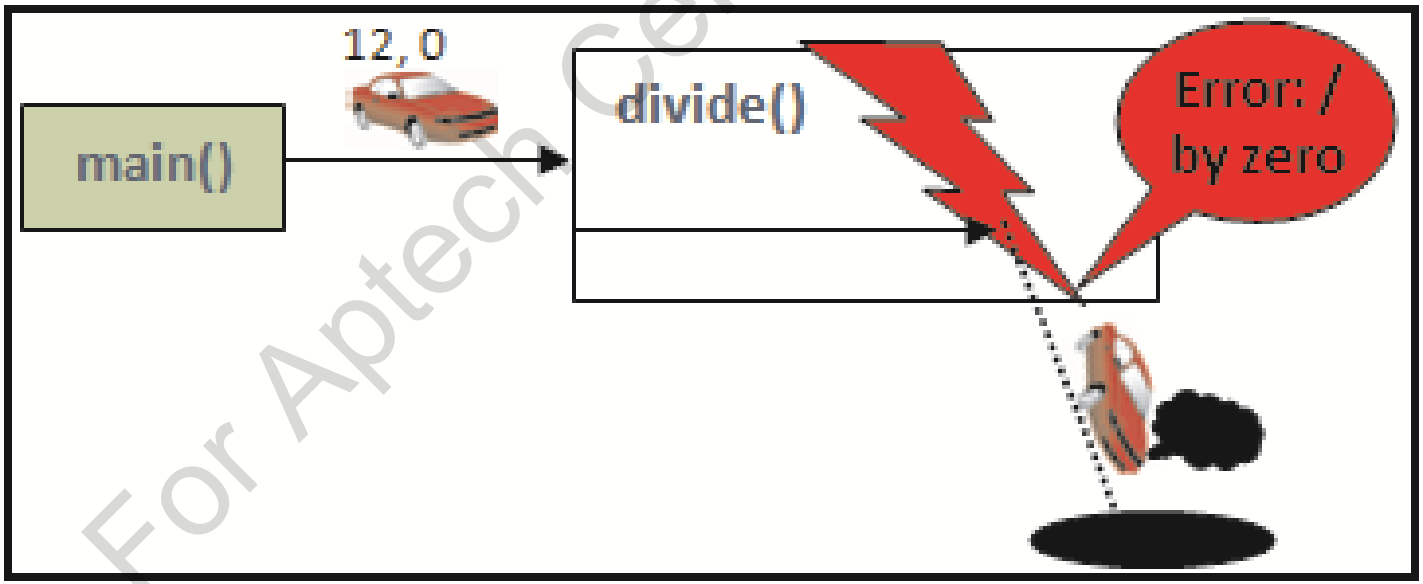
try-catch Block 5-5

- It is clear that the statement **num1/num2** might raise an error if the user specifies zero for the denominator **num2**.
- Therefore, the statement is enclosed within the `try` block.
- Division being an arithmetic operation, the user can create an appropriate `catch` block with `ArithmeticException` class object.
- Within the `catch` block, the `ArithmeticException` class object **e** is used to invoke the `getMessage()` method that will print the detail about the error.
- Following figure shows the output of the program when user specifies **12** as numerator and **0** as denominator:



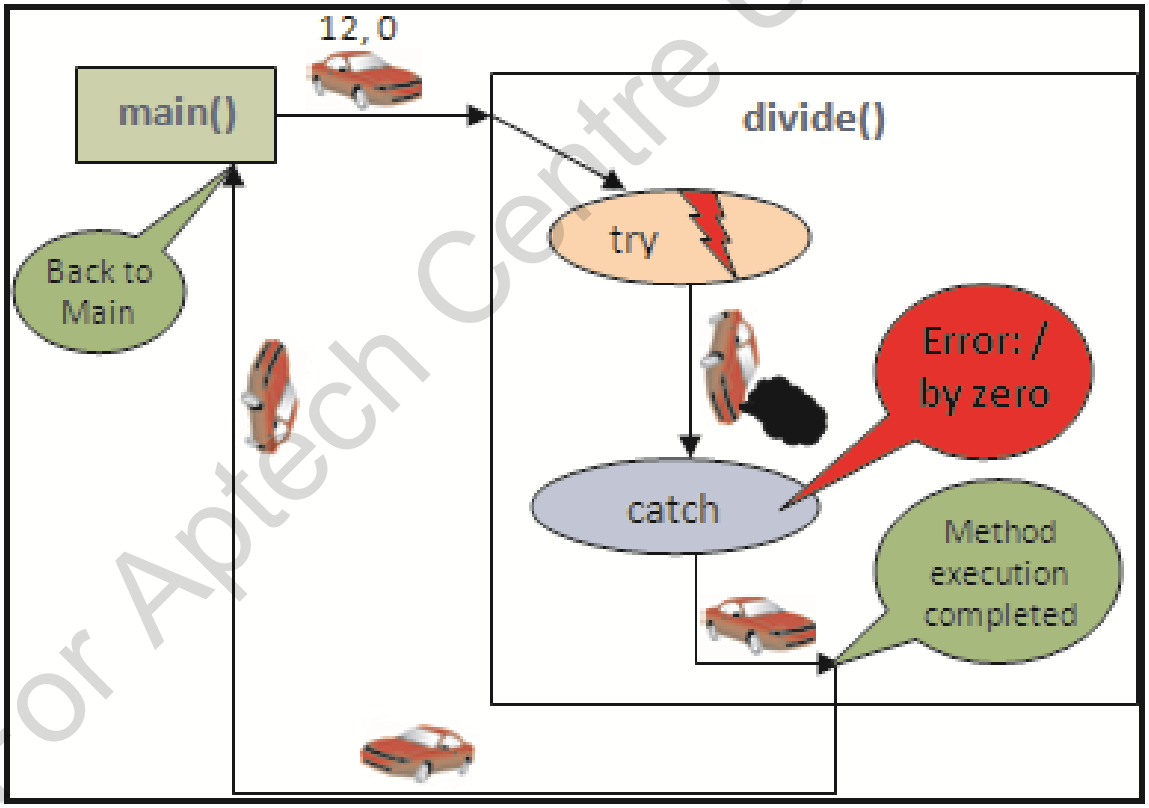
Execution Flow of Exceptions 1-2

- In the code, divide-by-zero exception occurs on execution of the statement `num1/num2`.
- If `try-catch` block is not provided, any code after this statement is not executed as an exception object is automatically created.
- Since, no `try-catch` block is present, JVM handles the exception, prints the stack trace, and the program is terminated.
- Following figure shows the execution of the code when `try-catch` block is not provided:



Execution Flow of Exceptions 2-2

- When the try-catch block is provided, the divide-by-zero exception occurring in the code is handled by the try-catch block and an exception message is displayed.
- Also, the rest of the code gets executed normally.
- Following figure shows the execution of the code when `try-catch` block is provided:



‘throw’ and ‘throws’ Keywords 1-5

- Java provides the `throw` and `throws` keywords to explicitly raise an exception in the `main()` method.
- The `throw` keyword throws the exception in a method.
- The `throws` keyword indicates the exception that a method may throw.
- The `throw` clause requires an argument of `Throwable` instance and raises checked or unchecked exceptions in a method.
- Following code snippet demonstrates the modified class **`Mathematics`** now using `throw` and `throws` keywords for handling exceptions:

```
package session12;
class Mathematics {

/**
 * Divides two integers, throws ArithmeticException
 * @param num1 an integer variable storing value of first number
 * @param num2 an integer variable storing value of second number
 * @return void
 */
public void divide(int num1, int num2) throws ArithmeticException {
```

‘throw’ and ‘throws’ Keywords 2-5

```
    // Check the value of num2
    if(num2==0) {
      // Throw the exception
      throw new ArithmeticException("/ by zero");
    }
    else {
      System.out.println("Division is: " + (num1/num2));
    }

    // Rest of the method
    System.out.println("Method execution completed");
    }
}

/**
 * Define the TestMath.java class
 */
public class TestMath {
    /**
    * @param args the command line arguments
    */
    public static void main(String[] args) {
```

‘throw’ and ‘throws’ Keywords 3-5

```
    // Check the number of command line arguments
    if(args.length==2) {
      // Instantiate the Mathematics class
      Mathematics objMath = new Mathematics();
      try {
        // Invoke the divide(int,int) method
        objMath.divide(Integer.parseInt(args[0]),
        Integer.parseInt(args[1]));
      }
      catch(ArithmeticException e) {
        // Display an error message to the user
        System.out.println("Error: "+ e.getMessage());
      }
    }
    else{
      System.out.println("Usage: java Mathematics <number1> <number2>");
    }
    System.out.println("Back to Main");
    }
}
```

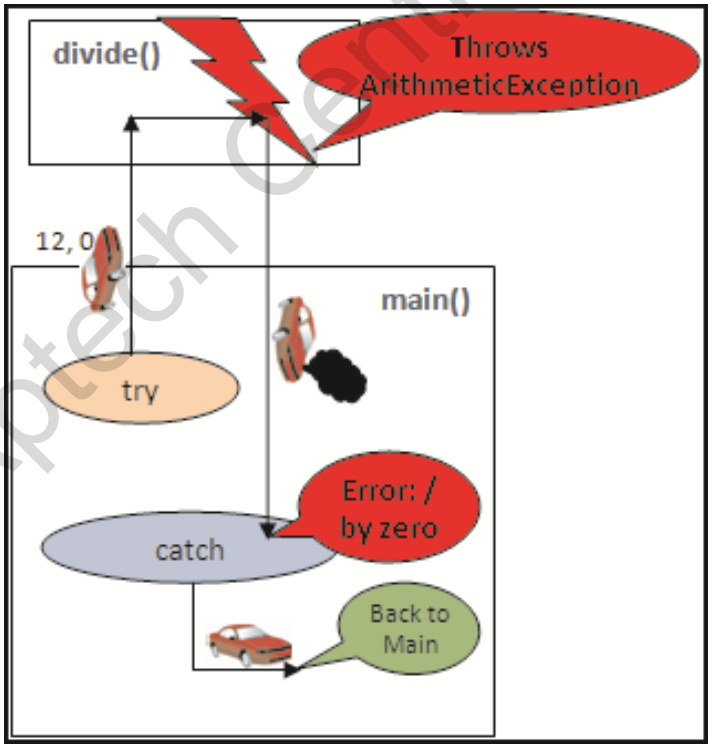
‘throw’ and ‘throws’ Keywords 4-5

- Within `divide(int,int)`, the code checks for the value of **num2**.
- If it is equal to zero, it creates an instance of `ArithmeticException` using the `new` keyword with the error message as an argument.
- The `throw` keyword throws the instance to the caller.
- Within the `main()` method, an instance, **objMath** is used to invoke the **divide(int,int)** method.
- However, this time, the code is written within the `try` block since `divide(int,int)` may throw an `ArithmeticException` that the `main()` method will have to handle within its `catch` block.
- Following figure shows the output of the program when user specifies **12** as numerator and **0** as denominator:

- The figure shows that upon execution of the code, the `if` condition becomes true and it throws the `ArithmeticException`.

‘throw’ and ‘throws’ Keywords 5-5

- The control returns back to the caller, that is, the `main()` method where it is finally handled.
- The `catch` block was executed and the result of `getMessage()` is displayed to the user.
- Notice, that the remaining statement of the **`divide(int,int)`** method is not executed in this case.
- Following figure shows the execution of the code when `throw` and `throws` clauses are used:



Multiple 'catch' Blocks 1-4

- The user can associate multiple exception handlers with a `try` block by providing more than one `catch` blocks directly after the `try` block.
- The syntax for declaring a `try` block with multiple `catch` blocks is as follows:

Syntax

```
try
{…}
catch (<exception-type> <object-name>)
{…}
catch (<exception-type> <object-name>)
{…}
```

- In this case, each `catch` block is an exception handler that handles a specific type of exception indicated by its argument exception-type.
- The runtime system invokes the handler in the call stack whose exception-type matches the type of the exception thrown.

Multiple 'catch' Blocks 2-4

- Following code snippet demonstrates the use of multiple `catch` blocks:

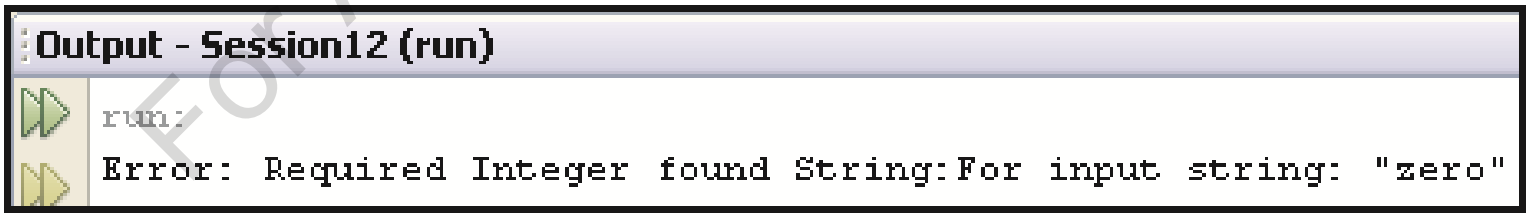
```
package session12;
public class Calculate {
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    // Check the number of command line arguments
    if (args.length == 2){
      try {
        // Perform the division operation
        int num3 = Integer.parseInt(args[0]) / Integer.parseInt(args[1]);
        System.out.println("Division is: "+num3);
      }
      catch (ArithmeticException e) { // Catch the ArithmeticException
        System.out.println("Error: " + e.getMessage());
      }
      catch (NumberFormatException e){ // Catch the NumberFormatException
        System.out.println("Error: Required Integer found String:" +
         e.getMessage());
      }
```

Multiple 'catch' Blocks 3-4

```
        catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
        }
    }
    else {
        System.out.println("Usage: java Calculate <number1> <number2>");
    }
  }
}
```

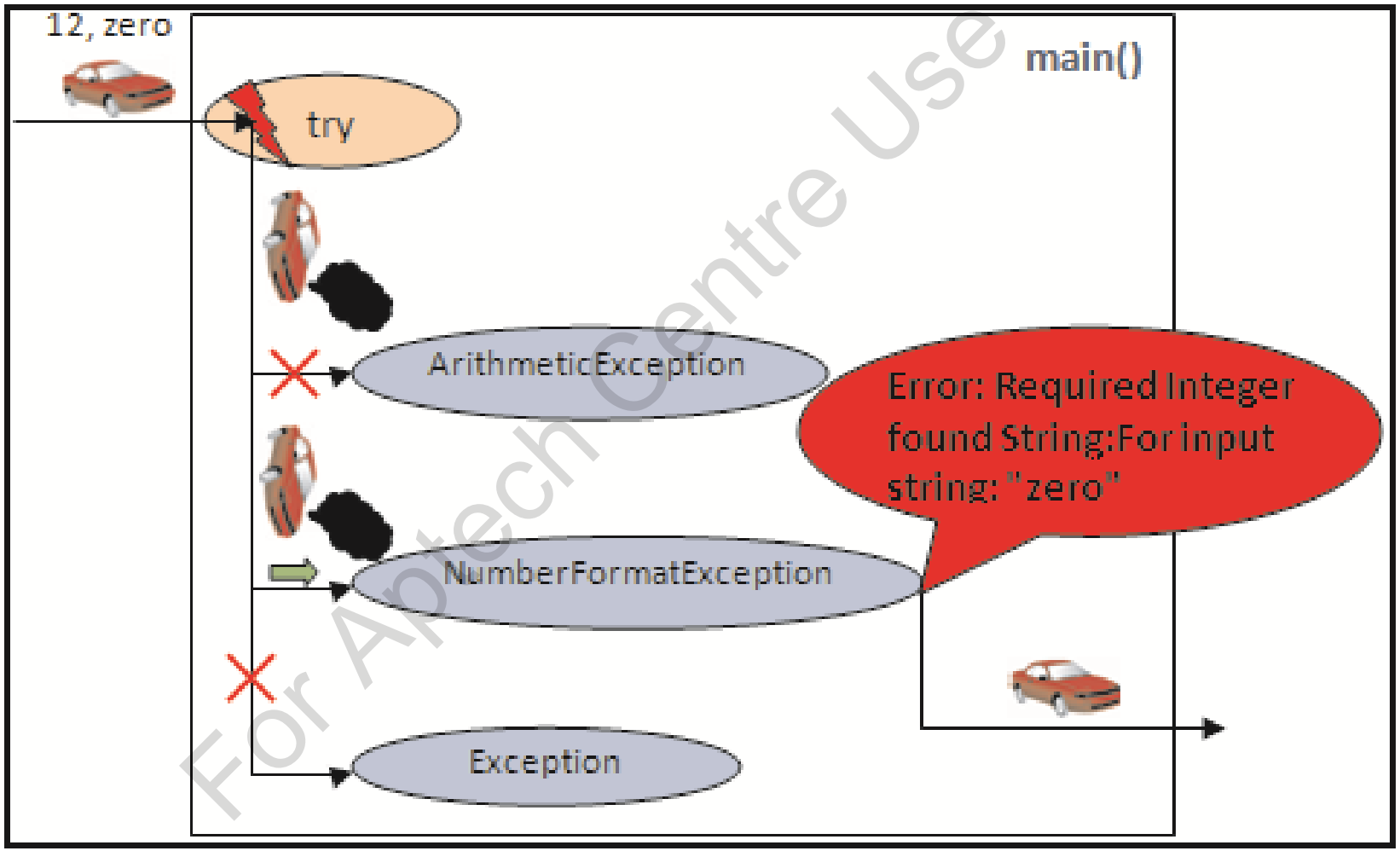
- Following figure shows the output of the code when user specifies **12** and **0** as arguments:

- Following figure shows the output of the code when user specifies **12** and '**zero**' as arguments:



Multiple 'catch' Blocks 4-4

- Following figure shows the execution of the code when multiple `catch` blocks are used:



‘finally’ Block 1-5

Java provides the `finally` block to ensure execution of certain statements even when an exception occurs.

The `finally` block is always executed irrespective of whether or not an exception occurs in the try block.

This ensures that the cleanup code is not accidentally bypassed by a `return`, `break`, or `continue` statement.

The `finally` block is mainly used as a tool to prevent resource leaks.

- Tasks such as closing a file and network connection, closing input-output streams, or recovering resources, must be done in a `finally` block to ensure that a resource is recovered even if an exception occurs.
- However, if due to some reason, the JVM exits while executing the `try` or `catch` block, then the `finally` block may not execute.
- Similarly, if a thread executing the `try` or `catch` block gets interrupted or killed, the `finally` block may not execute even though the application continues to execute.

'finally' Block 2-5

- The syntax for declaring `try-catch` blocks with a `finally` block is as follows:

Syntax

```
try
{
   // statements that may raise exception
   // statement 1
   // statement 2
}
catch(<exception-type> <object-name>)
{
   // handling exception
   // error message
}
finally
{
   // clean-up code
   // statement 1
   // statement 2
}
```

‘finally’ Block 3-5

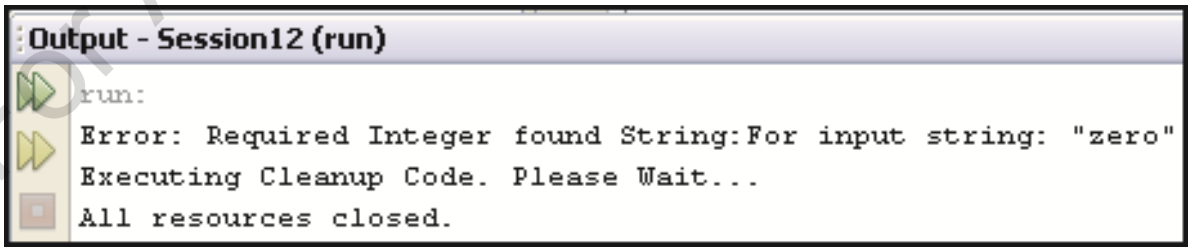
- Following code snippet demonstrates the modified class **`Calculate`** using the `finally` block:

```
package session12;
public class Calculate {
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    // Check the number of command line arguments
    if (args.length == 2) {
      try {
        int num3 = Integer.parseInt(args[0])/ Integer.parseInt(args[1]);
        System.out.println("Division is: "+num3);
      }
      catch (ArithmeticException e) {
        System.out.println("Error: " + e.getMessage());
      }
      catch (NumberFormatException e) {
        System.out.println("Error: Required Integer found String:" +
         e.getMessage());
      }
```

‘finally’ Block 4-5

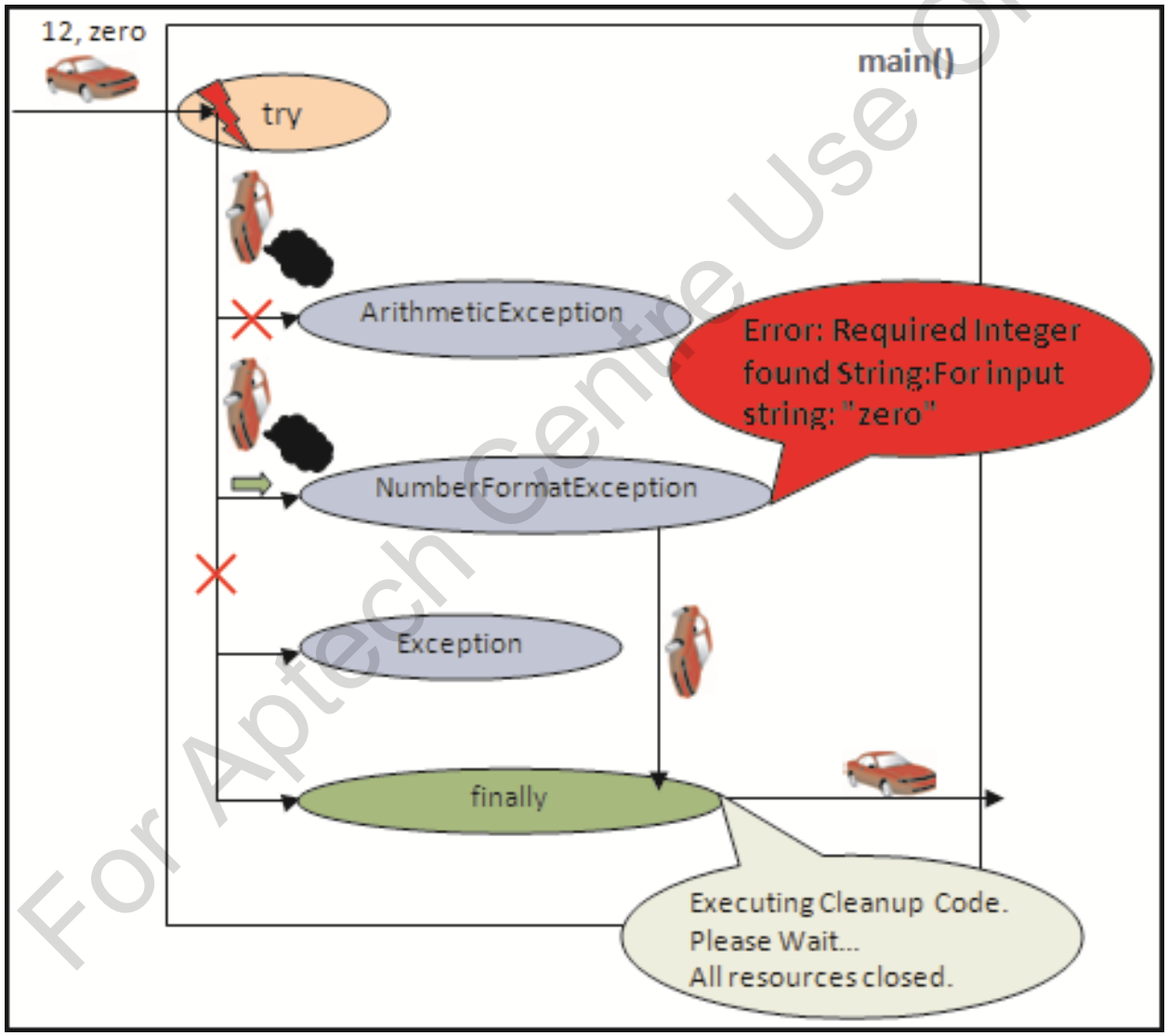
```
        catch (Exception e) {
          System.out.println("Error: " + e.getMessage());
        }
        finally {
          // Write the clean-up code for closing files, streams, and
           network connections
          System.out.println("Executing Cleanup Code. Please Wait...");
          System.out.println("All resources closed.");
        }
      }
      else {
        System.out.println("Usage: java Calculate <number1> <number2>");
      }
    }
}
```

- Following figure shows the output of the code after using `finally` block when user passes **12** and ‘**zero**’ as command line arguments:



‘finally’ Block 5-5

- Following figure shows the execution of code when `finally` block is used:



Guidelines for Handling Exceptions 1-3

- **The `try` statement must be followed by at least one `catch` or a `finally` block.**
- **Use the `throw` statement to throw an exception that a method does not handle by itself along with the `throws` clause in the method declaration.**
- **The `finally` block must be used to write clean up code.**
- **The `Exception` subclasses should be used when the caller of the method is expected to handle the exception.**
  - The compiler will raise an error message if the caller does not handle the exception.
- **Subclasses of `RuntimeException` class can be used to indicate programming errors such as `IllegalArgumentException`, `UnsupportedOperationException`, and so on.**

Guidelines for Handling Exceptions 2-3

Avoid using the `java.lang.Exception` or `java.lang.Throwable` class to catch exceptions that cannot be handled.

- Since, `Error` and `Exception` class can catch all exception of its subclasses including `RuntimeException`, the runtime behavior of such a code often becomes vague when global exception classes are caught.
- For example, one would not want to catch the `OutOfMemoryError`.
- How can one possible handle such an exception?

Provide appropriate message along with the default message when an exception occurs.

- All necessary data must be passed to the constructor of the exception class which can be helpful to understand and solve the problem.

Try to handle the exception as near to the source code as possible.

- If the caller can perform the corrective action, the condition must be rectified there itself.
- Propagating the exception further away from the source leads to difficulty in tracing the source of the exception.

Guidelines for Handling Exceptions 3-3

Exceptions should not be used to indicate normal branching conditions that may alter the flow of code invocation.

- For example, a method that is designed to return a zero, one, or an object can be modified to return `null` instead of raising an exception when it does not return any of the specified values.
- However, a disconnected database is a critical situation for which no alternative can be provided.
- In such a case, exception must be raised.

Repeated re-throwing of the same exception must be avoided as it may slow down programs that are known for frequently raising exceptions.

Avoid writing an empty `catch` block as it will not inform anything to the user and it gives the impression that the program failed for unknown reasons.

Summary 1-2

- An exception is an event or an abnormal condition in a program occurring during execution of a program that leads to disruption of the normal flow of the program instructions.
- The process of creating an exception object and passing it to the runtime system is termed as throwing an exception.
- An appropriate exception handler is one that handles the same type of exception as the one thrown by the method.
- Checked exceptions are exceptions that a well-written application must anticipate and provide methods to recover from.
- Errors are exceptions that are external to the application and the application usually cannot anticipate or recover from errors.
- Runtime Exceptions are exceptions that are internal to the application from which the application usually cannot anticipate or recover from.

Summary 2-2

- Throwable class is the base class of all the exception classes and has two direct subclasses namely, Exception and Error.
- The try block is a block of code which might raise an exception and catch block is a block of code used to handle a particular type of exception.
- The user can associate multiple exception handlers with a try block by providing more than one catch blocks directly after the try block.
- Java provides the throw and throws keywords to explicitly raise an exception in the main() method.
- Java provides the finally block to ensure execution of cleanup code even when an exception occurs.