Exceptions

Exceptions

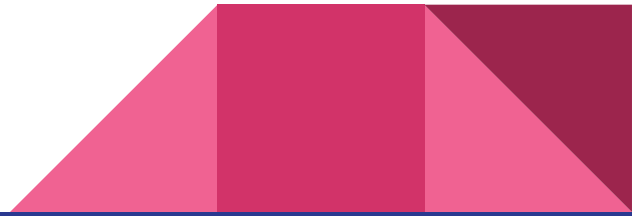
An exception is an **event that occurs during the execution of a program** that disrupts the normal flow of instructions.

When an exception occurs within a class or method, the method/class creates an **exception object** and hands the results to the runtime system (JVM).

What are some possible exceptions we can have in our code?

Some built-in exceptions in Java

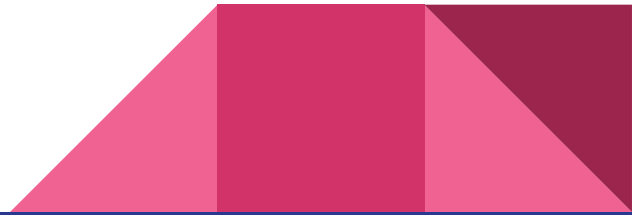
- Exception: This is the parent class of other exceptions
- IndexOutOfBoundsException
- IllegalArgumentException
- StringIndexOutOfBoundsException
- ArithmeticException
- ClassNotFoundException
- FileNotFoundException
- IOException
- SQLException
- NullPointerException
- NoSuchMethodException



Exception Handling

Exception Handling in Java is an **effective way to handle the runtime errors** so that the regular flow of the application can be preserved.

Exceptions can be caught and handled by the program. When an exception occurs within a method, it **creates an object**. This object is called the **exception object.** It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.



How can Java catch exceptions?

Try and Catch Statements

a. **Try**: Defines a block of code to be tested for error when it runs.
b. **Catch**: It defines a block of code that is executed if there is an error in the try block.

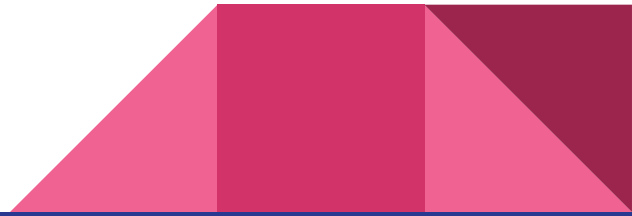
You can use the **try/catch** block to handle exceptions:

```
try{
    // block of code that could throw an exception.
}catch(ExceptionType variable){
    // block of code executed when there is an ExceptionType
}
```

Example

Let’s write a Java program where you will have three integer variables (a=10, b=0, and c=a/b) and see what happens.

```
try{
        int a = 10;
        int b = 0;
        int c = a / b;
 }catch(ArithmeticException e){
        System.out.println("Error division by zero");
 }
```



Try/Catch Statements

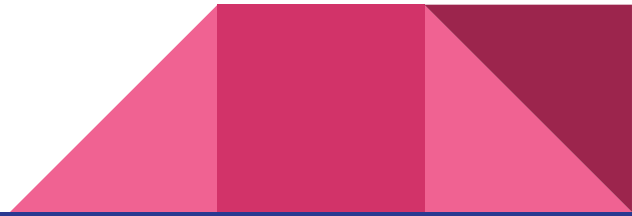
You may chain multiple catch statement if your code throws more that one exception:

```
try{
    // block of code that could throw an exception.
}catch(ExceptionType variable){
    // block of code executed when there is an ExceptionType
}catch(AnotherExceptionType variable){
    // block of code executed when there is an AnotherExceptionType
}
```

Use **try/catch** to print messages that any user can understand (not only programmers)

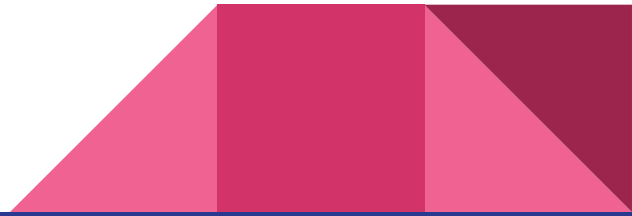
try/catch/finally

```
try{
        // block of code that could throw an exception.
}catch(ExceptionType variable){
        // block of code executed when there is an ExceptionType
}finally{
        // It is optional. The finally block always gets executed,
        // whether an exception occurred or not.
}
```



Example

```
int[] arr = new int[10];
try{
  System.out.println(arr[-1]);
}catch(ArrayIndexOutOfBoundsException e){
  System.out.println("Your index is wrong!");
}
```

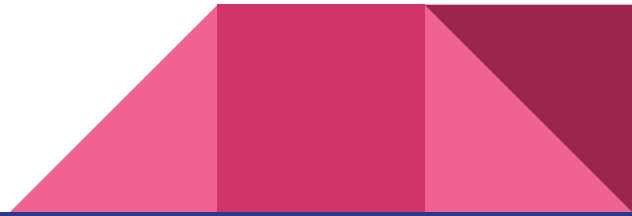


Example

```
int[] arr = {1, 2, 3, 4, 5};
try{
  for(int i = -1; i < 10; i++){
    System.out.println(arr[i]);
  }
}catch(ArrayIndexOutOfBoundsException e){
  System.out.println("Your index is wrong!");
}
```

What is the output?

Your index is wrong!



Example

```
int[] arr = {1, 2, 3, 4, 5};
for(int i = -1; i < 10; i++){
  try{
    System.out.println(arr[i]);
  }catch(ArrayIndexOutOfBoundsException e){
    System.out.println("Your index is wrong!");
  }
}
```

What is the output?

```
Your index is wrong!
1
2
3
4
5
Your index is wrong!
Your index is wrong!
Your index is wrong!
Your index is wrong!
Your index is wrong!
```

Example

```
int[] arr = {1, 2, 3, 4, 5};
for(int i = -1; i < 10; i++){
  try{
    System.out.println(arr[i]);
  }catch(ArrayIndexOutOfBoundsException e){
    System.out.println("Your index is wrong!");
    System.exit(1);
  }
}
```

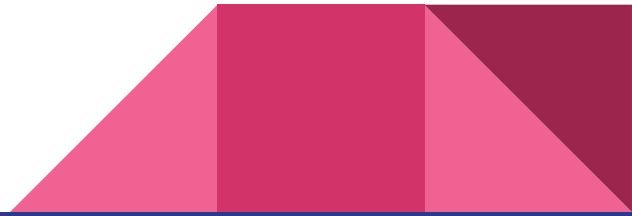
What is the output?

```
Your index is wrong!
|  State engine terminated.
|  Restore definitions with: /reload
-restore
```

It allows you to stop the program when an exception occurs. Until you identify and fix the underlying issue

Print the stack trace

```
int[] arr = {1, 2, 3, 4, 5};
for(int i = -1; i < 10; i++){
  try{
    System.out.println(arr[i]);
  }catch(ArrayIndexOutOfBoundsException e){
    System.out.println("Your index is wrong!");
    e.printStackTrace();
  }
}
```



Exceptions Advantages

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. For example, consider the pseudocode method here that reads an entire file into memory.

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

** See file

Cause an exception - Throw new exception

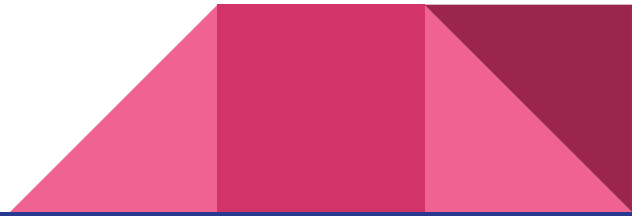
So far we have been printing errors when something in our code could cause an issue.

Instead of printing, we can create custom exceptions using the keyword **throw** in Java.

You then specify the Exception object you wish to throw.

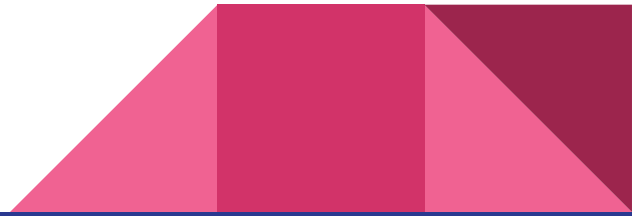
Every Exception includes a message which is a human-readable error description.

throw new Exception("Your custom message here");



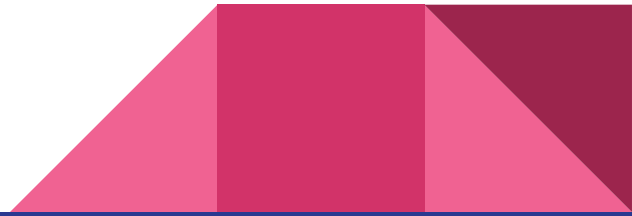
Example

```
class SafeDivider {
  public static double divide(int numerator, int denominator) {
    if (denominator == 0) {
      throw new IllegalArgumentException(
        "Denominator cannot be zero. Received: " + denominator);
    } else {
      double result = (double) numerator / denominator;
      System.out.println("Division result: " + result);
      return result;
    }
  }
}
```



Example

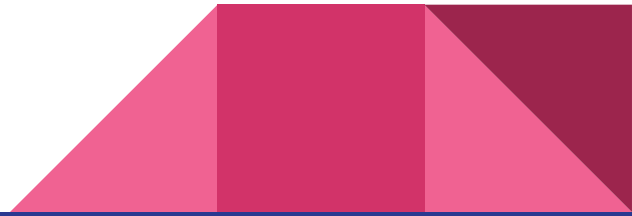
```
class ListAccessor {

  public static int getElement(int[] numbers, int index) {
    if (index < 0 || index >= numbers.length) {
      throw new IndexOutOfBoundsException(
        "Index " + index + " is out of bounds for length " +
         numbers.length);
    } else {
      System.out.println("Accessed element: " + numbers[index]);
      return numbers[index];
    }
  }
}
```



Handle an exception

Use a try/catch block in the function that calls a method which throws an exception.

For example, if your SuperArray class throws an exception, the driver (main program) should handle it using a try/catch block.



Example

SuperArray.java

```
public SuperArray(int initialCapacity) {
    if (initialCapacity < 0) {
        throw new IllegalArgumentException(
                "SuperArray cannot store a negative number
                 of elements.");
    }
    data = new String[initialCapacity];
    }
```

Driver.java

```
try {
      SuperArray a = new SuperArray(-1);
} catch (IllegalArgumentException e) {
       System.out.println(
           "Error creating a SuperArray object");
       e.printStackTrace();
}
```

Why Use Exceptions Instead of Printing Error Messages

1. Enforces proper error handling: Exceptions stop the program flow when something goes wrong, forcing you to handle the problem rather than ignore it.

2. Makes code more reliable: A printed message just *notifies* the user, but the program continues and may produce incorrect results or crash later.

3. Easier debugging: Exceptions include detailed information (type, message, stack trace) that helps locate the exact cause and location of the error.

4. Better design and reusability: Methods that throw exceptions can be reused safely in other programs or classes without worrying about hidden print statements.

5. Clear separation of concerns: Exception handling separates *normal logic* from *error-handling logic*, making code cleaner and easier to maintain.

Summary

1. **throw** – We know that if an error occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometimes we might want to generate exceptions explicitly in our code. The `throw` keyword is used to throw exceptions to the runtime to handle it.

2. **try-catch** – We use the `try-catch` block for exception handling in our code. `try` is the start of the block and `catch` is at the end of the `try` block to handle the exceptions. We can have multiple `catch` blocks with a `try` block. The `try-catch` block can be nested too. The `catch` block requires a parameter that should be of type `Exception`.

3. **finally** – the `finally` block is optional and can be used only with a `try-catch` block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use the `finally` block. The `finally` block always gets executed, whether an exception occurred or not.