
Java Exception

— www.andrew-programming.com —

What Is Exception?

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



Exception

Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords.

A try/catch block is placed around the code that might generate an exception.

Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following

Syntax

```
try {  
    // Protected code  
} catch (ExceptionName e1) {  
    // Catch block  
}
```

The Finally Block

- The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.
- Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

Syntax

```
try {  
    // Protected code  
} catch (ExceptionType1 e1) {  
    // Catch block  
} catch (ExceptionType2 e2) {  
    // Catch block  
} catch (ExceptionType3 e3) {  
    // Catch block  
}finally {  
    // The finally block always executes.  
}
```

The Try-Catch-Finally Block

```
1 public class ExceptTest {
2
3     public static void main(String args[]) {
4         int a[] = new int[2];
5         try {
6             System.out.println("Access element three : " + a[3]);
7         } catch (ArrayIndexOutOfBoundsException e) {
8             System.out.println("Exception thrown : " + e);
9         } finally {
10            a[0] = 6;
11            System.out.println("First element value: " + a[0]);
12            System.out.println("The finally statement is executed");
13        }
14    }
15 }
```

Output

```
Exception thrown : java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Note the following –

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

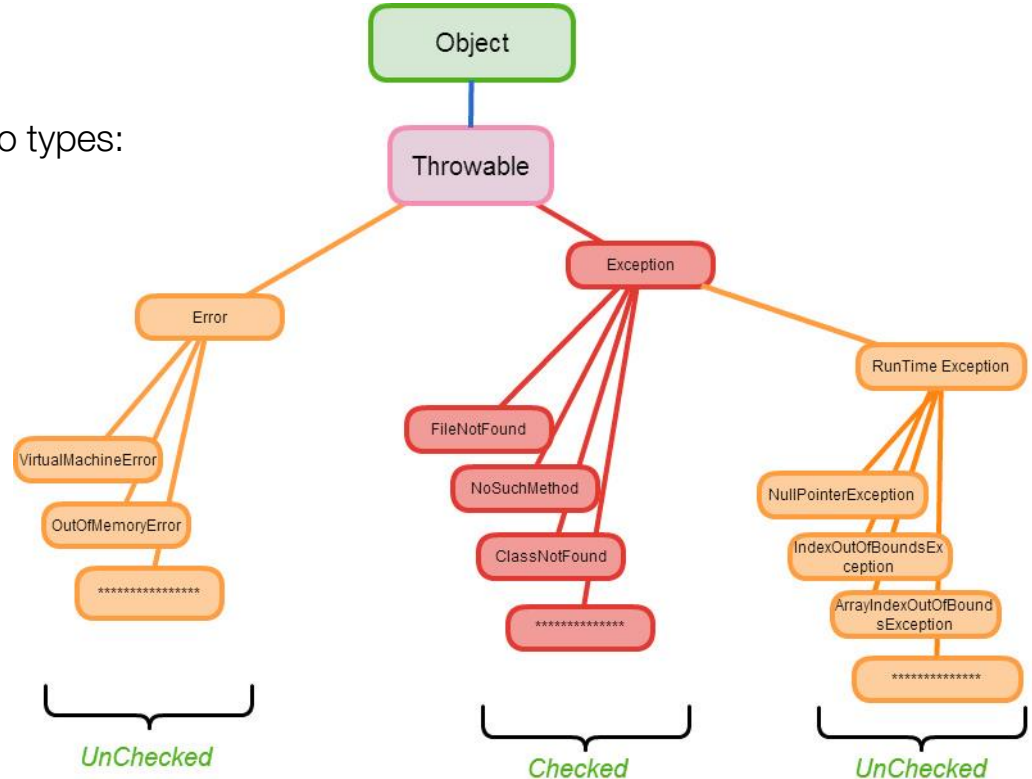
Java Exception Type

Java's exceptions can be categorized into two types:

- Checked exceptions
- Unchecked exceptions

Unchecked exceptions come in two types:

- Errors
- Runtime exceptions



Checked Exception

Checked exceptions are the type that programmers should anticipate and from which programs should be able to recover. All Java exceptions are checked exceptions except those of the Error and RuntimeException classes and their subclasses.

A checked exception is an exception which the Java source code must deal with, either by catching it or declaring it to be thrown. Checked exceptions are generally caused by faults outside of the code itself - missing resources, networking errors, and problems with threads come to mind. These could include subclasses of FileNotFoundException,

Name	Description
IOException	While using file input/output stream related exception
SQLException.	While executing queries on database related to SQL syntax
DataAccessException	Exception related to accessing data/database
ClassNotFoundException	Thrown when the JVM can't find a class it needs, because of a command-line error, a classpath issue, or a missing .class file
InstantiationException	Attempt to create an object of an abstract class or interface.

Checked Example Program

```
package exception.trycatchresourcedemo.checkedexception;

import java.io.*;

class Main {
    public static void main(String[] args) {
        FileReader file = null;

        file = new FileReader( fileName: "A.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());
        fileInput.close();
    }
}
```



```
package exception.trycatchresourcedemo.checkedexception;

import java.io.*;

class Main {
    public static void main(String[] args) {
        FileReader file = null;
        try {
            file = new FileReader( fileName: "A.txt");
            BufferedReader fileInput = new BufferedReader(file);

            // Print first 3 lines of file "C:\test\a.txt"
            for (int counter = 0; counter < 3; counter++)
                System.out.println(fileInput.readLine());
            fileInput.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```


Checked Example Program

```
package exception.trycatchresourcedemo.checkedexception;

import java.io.*;

class Main {
    public static void main(String[] args) {
        FileReader file = null;

        file = new FileReader( fileName: "A.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());
        fileInput.close();
    }
}
```



```
package exception.trycatchresourcedemo.checkedexception;

import java.io.*;

class Main {
    public static void main(String[] args) throws Exception {
        FileReader file = null;

        file = new FileReader( fileName: "A.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());
        fileInput.close();
    }
}
```

Unchecked Exception

Unchecked exceptions inherit from the Error class or the RuntimeException class. Many programmers feel that you should not handle these exceptions in your programs because they represent the type of errors from which programs cannot reasonably be expected to recover while the program is running.

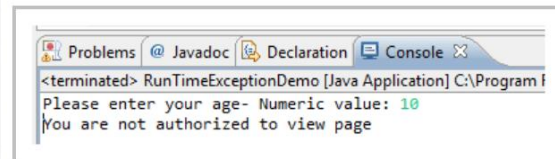
When an unchecked exception is thrown, it is usually caused by a misuse of code - passing a null or otherwise incorrect argument.

Name	Description
<code>NullPointerException</code>	Thrown when attempting to access an object with a reference variable whose current value is null
<code>ArrayIndexOutOfBoundsException</code>	Thrown when attempting to access an array with an invalid index value (either negative or beyond the length of the array)
<code>IllegalArgumentException.</code>	Thrown when a method receives an argument formatted differently than the method expects.
<code>IllegalStateException</code>	Thrown when the state of the environment doesn't match the operation being attempted, e.g., using a Scanner that's been closed.
<code>NumberFormatException</code>	Thrown when a method that converts a String to a number receives a String that it cannot convert.
<code>ArithmeticException</code>	Arithmetic error, such as divide-by-zero.

Unchecked Exceptions Demo

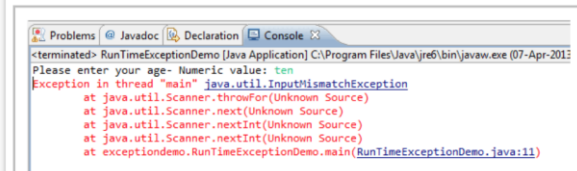
```
1 import java.util.Scanner;
2 public class RunTimeExceptionDemo {
3     public static void main(String[] args) {
4         //Reading user input
5         Scanner inputDevice = new Scanner(System.in);
6         System.out.print("Please enter your age- Numeric value: ");
7         int age = inputDevice.nextInt();
8         if (age>18){
9             System.out.println("You are authorized to view the page");
10            //Other business logic
11        }else {
12            System.out.println("You are not authorized to view page");
13            //Other code related to logout
14        }
15    }
16 }
17 }
```

Output:



```
Problems @ Javadoc Declaration Console X
<terminated> RunTimeExceptionDemo [Java Application] C:\Program f
Please enter your age- Numeric value: 10
You are not authorized to view page
```

If User enters non-numeric value, program ends in error/exceptional condition.



```
Problems @ Javadoc Declaration Console X
<terminated> RunTimeExceptionDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (07-Apr-2011)
Please enter your age- Numeric value: ten
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Unknown Source)
    at java.util.Scanner.next(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at exceptiondemo.RunTimeExceptionDemo.main(RunTimeExceptionDemo.java:11)
```

Catching Exception Demo

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
1 // File Name : ExceptTest.java
2 import java.io.*;
3
4 public class ExceptTest {
5
6     public static void main(String args[]) {
7         try {
8             int a[] = new int[2];
9             System.out.println("Access element three :" + a[3]);
10        } catch (ArrayIndexOutOfBoundsException e) {
11            System.out.println("Exception thrown :" + e);
12        }
13        System.out.println("Out of the block");
14    }
15 }
```

Output

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple Catch Blocks

you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list.

If the data type of the exception thrown matches `ExceptionType1`, it gets caught there. If not, the exception passes down to the second catch statement.

This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Syntax

```
try {  
    // Protected code  
} catch (ExceptionType1 e1) {  
    // Catch block  
} catch (ExceptionType2 e2) {  
    // Catch block  
} catch (ExceptionType3 e3) {  
    // Catch block  
}
```

Multiple Catch Blocks Demo

```
try {
    file = new FileInputStream(fileName);
    x = (byte) file.read();
} catch (IOException i) {
    i.printStackTrace();
    return -1;
} catch (FileNotFoundException f) // Not valid! {
    f.printStackTrace();
    return -1;
}
```

Catching Multiple Type of Exceptions

Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code.

```
catch (IOException|FileNotFoundException ex) {  
    logger.log(ex);  
    throw ex;  
}
```

The Throws/Throw Keywords

- If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.
- You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.
- Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The Throws/Throw Keywords Demo

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

```
import java.io.*;
public class className {

    public void deposit(double amount) throws RemoteException {
        // Method implementation
        throw new RemoteException();
    }
    // Remainder of class definition
}
```

The Throws/Throw Keywords Demo

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas.

For example, the following method declares that it throws a `RemoteException` and an `InsufficientFundsException` –

Example

```
import java.io.*;
public class className {

    public void withdraw(double amount) throws RemoteException,
        InsufficientFundsException {
        // Method implementation
    }
    // Remainder of class definition
}
```

The Finally Block

- The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.
- Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

Syntax

```
try {  
    // Protected code  
} catch (ExceptionType1 e1) {  
    // Catch block  
} catch (ExceptionType2 e2) {  
    // Catch block  
} catch (ExceptionType3 e3) {  
    // Catch block  
}finally {  
    // The finally block always executes.  
}
```

The Finally Block

```
1 public class ExceptTest {
2
3     public static void main(String args[]) {
4         int a[] = new int[2];
5         try {
6             System.out.println("Access element three : " + a[3]);
7         } catch (ArrayIndexOutOfBoundsException e) {
8             System.out.println("Exception thrown : " + e);
9         } finally {
10            a[0] = 6;
11            System.out.println("First element value: " + a[0]);
12            System.out.println("The finally statement is executed");
13        }
14    }
15 }
```

Output

```
Exception thrown : java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Note the following –

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

User-defined Exceptions

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes –

- All exceptions must be a child of Throwable.
- 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.

We can define our own Exception class as below –

```
class MyException extends Exception {  
}
```

User-defined Exceptions

Example

```
// File Name InsufficientFundsException.java
import java.io.*;

public class InsufficientFundsException extends Exception {
    private double amount;

    public InsufficientFundsException(double amount) {
        this.amount = amount;
    }

    public double getAmount() {
        return amount;
    }
}
```

```
// File Name CheckingAccount.java
import java.io.*;

public class CheckingAccount {
    private double balance;
    private int number;

    public CheckingAccount(int number) {
        this.number = number;
    }

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) throws InsufficientFundsException {
        if (amount <= balance) {
            balance -= amount;
        } else {
            double needs = amount - balance;
            throw new InsufficientFundsException(needs);
        }
    }

    public double getBalance() {
        return balance;
    }

    public int getNumber() {
        return number;
    }
}
```

User-defined Exceptions

```
// File Name BankDemo.java
public class BankDemo {

    public static void main(String [] args) {
        CheckingAccount c = new CheckingAccount(101);
        System.out.println("Depositing $500...");
        c.deposit(500.00);

        try {
            System.out.println("\nWithdrawing $100...");
            c.withdraw(100.00);
            System.out.println("\nWithdrawing $600...");
            c.withdraw(600.00);
        } catch (InsufficientFundsException e) {
            System.out.println("Sorry, but you are short $" + e.getAmount());
            e.printStackTrace();
        }
    }
}
```

Output

```
Depositing $500...

Withdrawing $100...

Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
    at CheckingAccount.withdraw(CheckingAccount.java:25)
    at BankDemo.main(BankDemo.java:13)
```