Java Throw Exception

24 June 2025 20:59

In Java, the **throw** keyword is used to explicitly throw an exception within a method or block of code. It allows developers to signal that an error has occurred and needs to be handled. The throw exception must be an instance of **Throwable**, or one of its subclasses, such as **Exception** or **RuntimeException**.

Java throw Keyword

The Java throw keyword is used to throw an exception explicitly from a method or block of code. It can be used to throw both checked and unchecked exceptions. It is mainly used to throw a custom exception. It provides developer with a way to trigger an exception whenever a particular undesirable situation occurs manually.

We specify the exception object that is to be thrown. The exception has a message with it that provides the error description. These exceptions may be related to the user input, server hardware malfunctions, invalid computations or many other conditions, depending on the program's requirements.

We can also define our own set of conditions and throw an exception explicitly using the **throw** keyword. For example, we can throw an ArithmeticException if we divide a number by another number. Here we just need to set the condition and throw an exception using the throw keyword.

Syntax:

```
throw new exception_class("Error message");
```

When instance refers to an object of a class that must extend throwable. (which includes the exception class and its subclasses).

```
throw new IOException("Sorry! Device error");
```

Here:

- new is used to create an instance(object) of the exception class.
- The error message provides context about the reason for the exception.

When the instance must be of type throwable or a subclass of throwable, for example, exception is a subclass of Throwable and the user defined exceptions usually extend the exception class.

Key requirements

- The object we throw must be of type Throwable or one of its subclasses.
- For example, Exception is a subclass of Throwable.
- User-defined exception (also known as custom exceptions) usually extend the Exception class.
- If you throw a checked exception and do not handle it inside the method body, you must declare it using the **throws** keyword in the method signature.

Why use throw in Java?

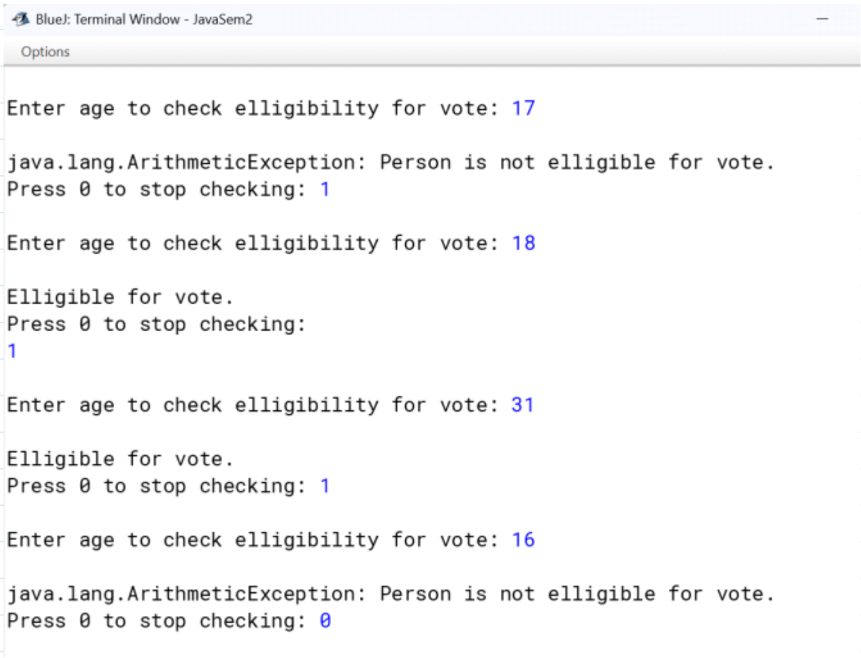
Manually throwing exceptions allows programmers to:

- Validate Inputs (for example, throwing an exception if the user input is invalid).
- Handle specific conditions (For example, throwing exceptions if a requested resource is not found).
- Create more meaningful error messages that are specific to the application's context.
- Improved debugging by providing custom error information.

Java throw Keyword Example:

```
package ExceptionHandling;
import java.util.*;

/**
 * Write a description of class ThrowException here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class ThrowException
{
    public static void validate(int age){
        if(age < 18){
            throw new ArithmeticException("Person is not elligible for vote.");
        }
        else{
            System.out.print("\nElligible for vote.");
        }
    }

    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        System.out.print("\f");
        //int ch;
        while(true){
            try{
                System.out.print("\nEnter age to check elligibility for vote: ");
                validate(sc.nextInt());
            }
            catch(ArithmeticException ae){
                System.out.print("\n"+ae);
            }
            System.out.print("\nPress 0 to stop checking: ");
            if(sc.nextInt() == 0){
                break;
            }
        }
    }
}
```

Output:



```
Enter age to check elligibility for vote: 17

java.lang.ArithmeticException: Person is not elligible for vote.
Press 0 to stop checking: 1

Enter age to check elligibility for vote: 18

Elligible for vote.
Press 0 to stop checking:
1

Enter age to check elligibility for vote: 31

Elligible for vote.
Press 0 to stop checking: 1

Enter age to check elligibility for vote: 16

java.lang.ArithmeticException: Person is not elligible for vote.
Press 0 to stop checking: 0
```

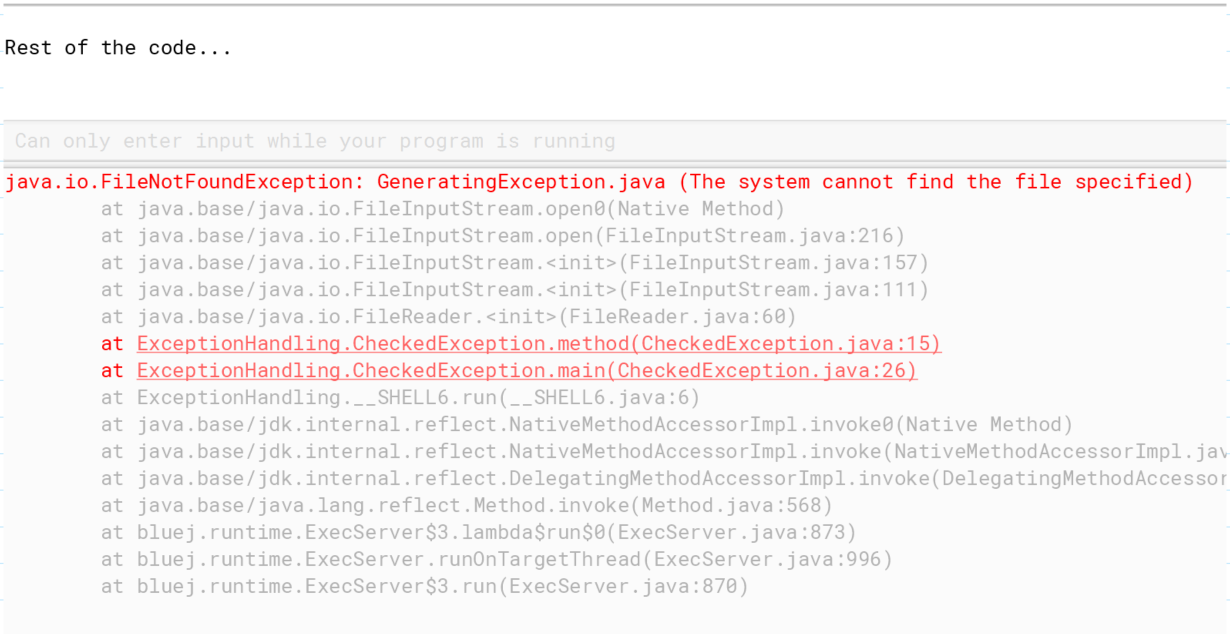
Example: Throwing a Checked Exception

In Java checked exceptions are the exceptions that the compiler requires you to either handle using a try-catch block or declare using **throws** keyword in the method signature. These exceptions typically arise from external sources such as file operations, network connections or database access operations that are prone to failure.

Unlike checked exceptions (which are subclasses of RuntimeException) Check the exceptions are subclasses of Exception (but not of RuntimeException).

```
package ExceptionHandling;
import java.util.*;
import java.io.*;


/**
 * Write a description of class CheckedException here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class CheckedException
{
   public static void method()throws FileNotFoundException{
      FileReader fr = new FileReader("GeneratingException.java");
      BufferedReader br = new BufferedReader(fr);
      //Explicitly throwing a checked exception
      throw new FileNotFoundException();
   }
   public static void main(String args[]){
      Scanner sc = new Scanner(System.in);
      System.out.print("\f");

      try
      {
         method();
      }
      catch (FileNotFoundException fnfe)
      {
         fnfe.printStackTrace();
      }
      System.out.print("\nRest of the code...");
   }
}
```

Output:



```
Rest of the code...

Can only enter input while your program is running

java.io.FileNotFoundException: GeneratingException.java (The system cannot find the file specified)
	at java.base/java.io.FileInputStream.open0(Native Method)
	at java.base/java.io.FileInputStream.open(FileInputStream.java:216)
	at java.base/java.io.FileInputStream.<init>(FileInputStream.java:157)
	at java.base/java.io.FileInputStream.<init>(FileInputStream.java:111)
	at java.base/java.io.FileReader.<init>(FileReader.java:60)
	at ExceptionHandling.CheckedException.method(CheckedException.java:15)
	at ExceptionHandling.CheckedException.main(CheckedException.java:26)
	at ExceptionHandling.__SHELL6.run(__SHELL6.java:6)
	at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
	at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.jav
	at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessor
	at java.base/java.lang.reflect.Method.invoke(Method.java:568)
	at bluej.runtime.ExecServer$3.lambda$run$0(ExecServer.java:873)
	at bluej.runtime.ExecServer.runOnTargetThread(ExecServer.java:996)
	at bluej.runtime.ExecServer$3.run(ExecServer.java:870)
```

Throwing User-defined Exception

The exception is everything else under the **Throwable** class. Java allows developers to create their own custom exceptions by extending the **Exception** class or one of its subclasses. They are typically used when you want to enforce application specific rules and constraints that are not covered by Java's built-in exceptions.

In this example, we will define a custom exception called UserDefinedException and throw it manually using the throw keyword. We will also catch and handle it using a try-catch block, just like we would with any other checked exceptions.

```
public class UserDefinedException extends Exception
{
   public UserDefinedException(String msg){
      //Callling the constructor of super class or parent class
      super(msg);
   }
}
```

sending 'msg' to Exception class which is a parent.

```
import java.util.*;
public class TestUserDefinedException
{
   public static void main(String args[]){
      Scanner sc = new Scanner(System.in);
      System.out.print("\f");
      try{
         throw new UserDefinedException("This is user defined exception.");
      }
      catch(UserDefinedException ude){
         System.out.print("\nCaught the Exception");
         System.out.print("\n"+ude.getMessage());
      }
   }
}
```

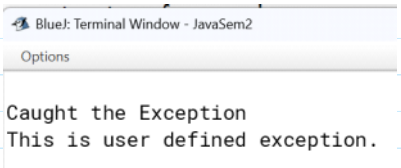
This is a constructor

Message.

Throwing the object created by new keyword

Catching the object.

OUTPUT:



```
Caught the Exception
This is user defined exception.
```

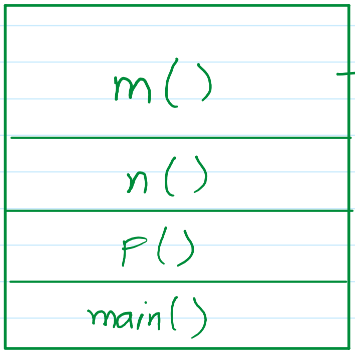
Exception Propagation in Java

An exception is a first thrown from the top of the stack, and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Note: By default, unchecked exceptions are forward in calling chain (propagated).

```
package ExceptionHandling;


/**
 * Write a description of class ExceptionPropagating here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class ExceptionPropagation
{
    public void m(){
        int data = 100/0;
    }

    public void n(){
        m();
    }

    public void p(){
        try{
            n();
        }
        catch(Exception e){
            System.out.print("\nException handled");
        }
    }

    public static void main(String args[]){
        System.out.print("\f");
        ExceptionPropagation obj = new ExceptionPropagation();
        obj.p();
        System.out.print("\nNormal flow...");
    }
}
```

→ exception not handled, so it is propogated to the previous method n(), where it is not handled, again it is propagted to p() method. where exception is handled.

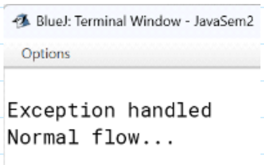


call stack.

→ exception ocured.

Exception can be handled in any method in call stack either in main(), p(), n() or m() method.

Output:



```
Exception handled
Normal flow...
```

Java throws Keyword

The Java throws keyword is used in method declarations. It specifies the exceptions that a method might throw during execution. Therefore, the programmer should provide exception handling code or propagate the exception to maintain the normal flow of the program.

Exception handling is mainly used to handle the directions. If there occurs any unchecked exception occurs, such as **NullPointerException** it is the programmer's fault that he is not checking the code before it is used.

Syntax:

```
        Return_type method_name() throws exception_class_name{
            //method code
        }
```

Advantages of the throws keyword

1. **Simplifies exception handling:** Instead of managing exceptions with the method, throws allow the caller to handle them, making the code cleaner.
2. **Encourages Code Reusability:** Methods can be reused without integrating exception handling logic, which enhances flexibility.
3. **Useful for Checked Exceptions:** It ensures that check the exceptions are properly or appropriately handled or propagated, preventing unexpected runtime failures.
4. **Supports Multiple Exception Declaration:** A method can declare multiple exceptions using commas, which improves clarity.

For example:

```
package ExceptionHandling;
import java.io.*;


/**
 * Write a description of class ThrowsKeywordDemo here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class ThrowsKeywordDemo
{
  public void method() throws IOException{
    throw new IOException("Device Error");
  }
```

case - 2

If we declare an exception and it does not occur, the code will execute correctly.

If we declare the exception and it does occur, it will be thrown at runtime, as the throws keyword does not manage the exception.

```
  public void anotherMethod() throws IOException {
    method();
  }

  public void nextMethod(){
    try
    {
      anotherMethod();
```

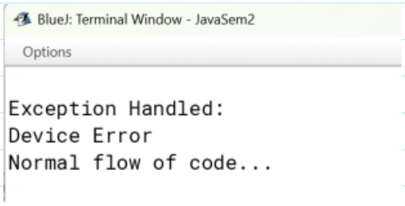
case-1

Exception.

```
        try
        {
            anotherMethod();
        }
        catch (IOException ioe)
        {
            System.out.print("\nException Handled: ");
            System.out.print("\n"+ioe.getMessage());
        }
    }

    public static void main(String args[]){
        System.out.print("\f");
        ThrowsKeywordDemo ob = new ThrowsKeywordDemo();
        ob.nextMethod();
        System.out.print("\nNormal flow of code...");
    }
}
```

Throwabe
↑Exception
↑IOException

Output:



```
Exception Handled:
Device Error
Normal flow of code...
```

Rule: If we are calling a method that declares an exception, we must either catch or declare the exception.
There are the following two cases:
Case one: we have got the exception. That is, we have handled the exception using a try /catch block.
Case two: We have declared the exception that. is specified through keyword with the method.

When to use throws keyword?
The throws keywords should be used when a method might result in a checked exception, but it but is not responsible for handling it directly. It is appropriate when you want to inform the caller about potential exceptions, allowing them to decide how to handle them.

You can write methods that deal with file handling, database connections or network operations, often declare exceptions using throws. This approach maintains cleaner code by pushing the responsibility of handling the exception to higher levels in the program's call stack.

Common Mistake while using throws Keyword
A common mistake is using throws for unchecked exception NullPointerException, which do not require a declaration. Another error is assuming that declaration and exception with throws means it is handled. it only notifies the caller. Also over using throws can lead to poor exception management, making the code harder to maintain. developers must ensure that exceptions are properly caught at an appropriate level to prevent unexpected crashes.

Important points to remember

1. Notice that the throws keyword is used to define exceptions in the header of a method in question, which. notifies the caller of some exceptions that can be thrown.
2. It is only appropriate to use throws with checked exceptions, such as IOException or SQLException and not with unchecked exceptions like NullPointerException or ArithmeticException.
3. A method that calls another method that declares a checked exception should either try to handle it with try - catch block or redeclare it with a throws clause.

4. Using tools to define an exception does not mean that the exception is being handled. It is just begin handed over to the parent level.
5. An exception that goes uncaught will be thrown at runtime, causing an unexpected and uncontrolled crash of the application.
6. Classes of Error like OutOfMemory error and StackOverflowError. Do not need to be caught with defined throws because such errors are not meant to be controlled by the program.

Conclusion:

The throes feature in Java programming language is very important in the context of exception handling, specifically, checked exceptions. It enables the code to pass the burden of dealing with exception upwards to the calling method. This enhances the organization of the code as well as modularity property. Proper handling of exceptions improves the program's robustness and ease of debugging, since the errors location and its types are clearly marked.