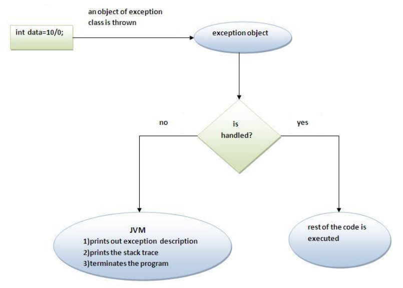
Java

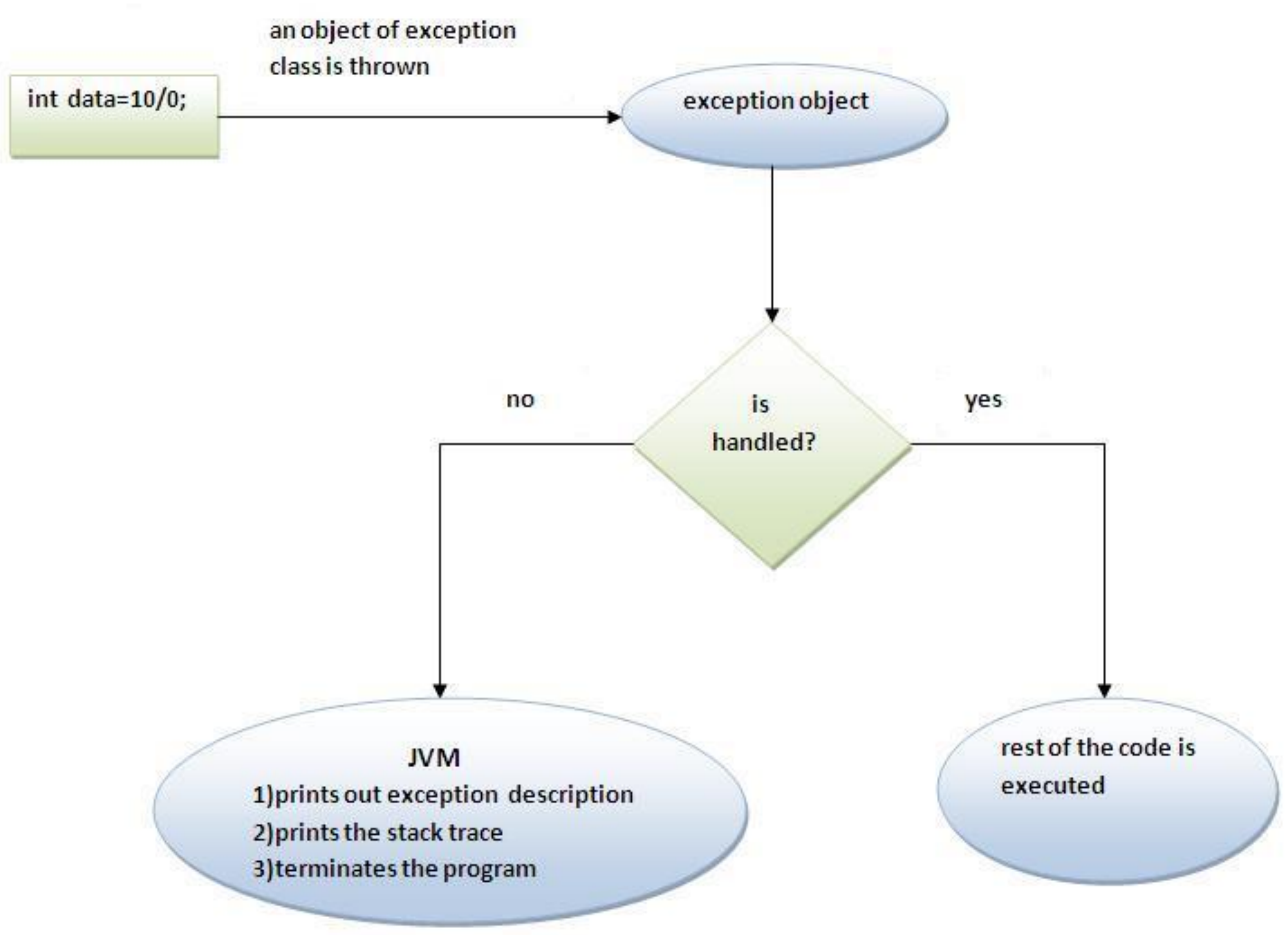
Exceptions


an object of exception class is thrown
int data=10/0;
exception object
is handled?
no
yes
JVM
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

Exceptions

An **Exception** (or *exceptional* event) is a problem that arises during the execution of a program.

When an **Exception** occurs, the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended. Therefore, these *Exceptions* need to be handled.


int data=10/0;
an object of exception class is thrown
exception object
is handled?
no
yes
JVM
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

Why do programs throw Exceptions?

An Exception can occur due to many different reasons. Here are some scenarios, where an Exception occurs.

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications.
- JVM has run out of memory.

Some of these Exceptions are caused by user error, others by programmer mistakes, and others by physical resources that have failed in some manner.

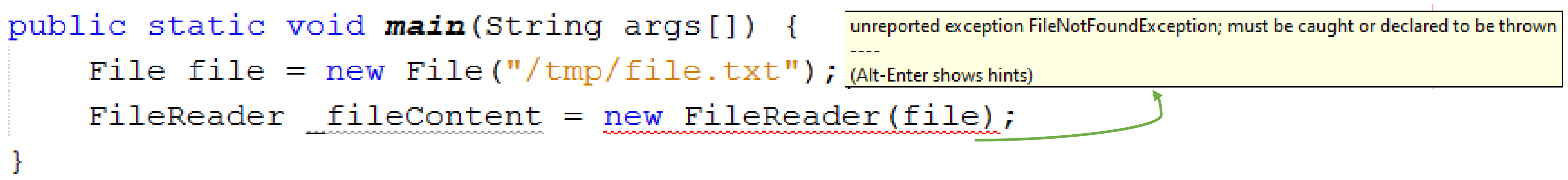
Types of Exceptions

We have three categories of Exceptions. You need to understand them to know how Exception handling works in Java.

- *Checked exceptions*
- *Unchecked exceptions*
- *Errors*

Checked Exceptions - Example

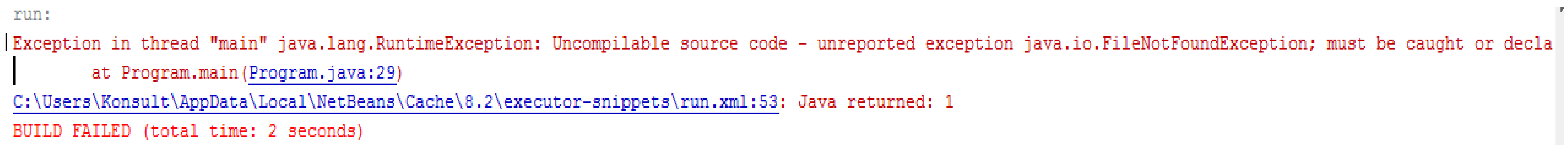
A *checked* Exception is an Exception that occurs at *compile time*. These are also called *compile time* Exceptions. Such Exceptions cannot simply be ignored at the time of compilation. The programmer should take care of (handle) these Exceptions.



```
public static void main(String args[]) {
    File file = new File("/tmp/file.txt");
    FileReader fileContent = new FileReader(file);
}
```

```
unreported exception FileNotFoundException; must be caught or declared to be thrown
----
(Alt-Enter shows hints)
```

For example, if you use **FileReader** class in your program to read data from a file, and the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the Exception. If you try to compile the above program, you will get the following Exception:



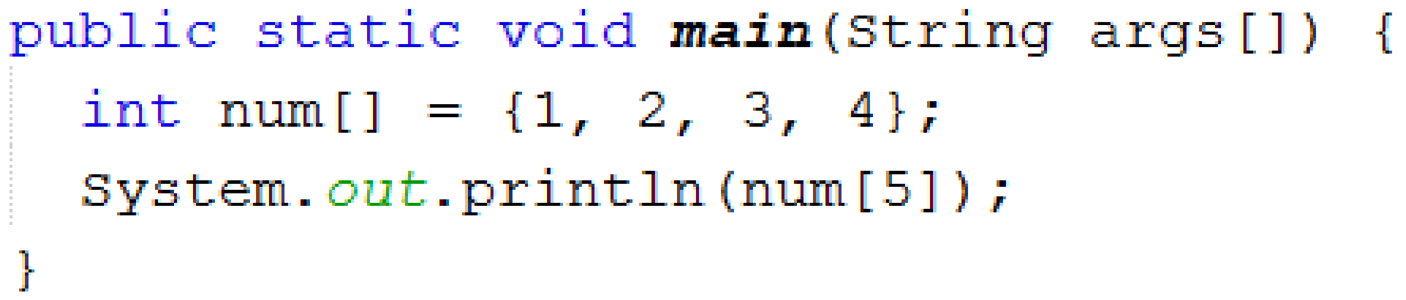
```
run:
Exception in thread "main" java.lang.RuntimeException: Uncompilable source code - unreported exception java.io.FileNotFoundException; must be caught or decla
	at Program.main(Program.java:29)
C:\Users\Konsult\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 2 seconds)
```

Unchecked Exceptions - Example

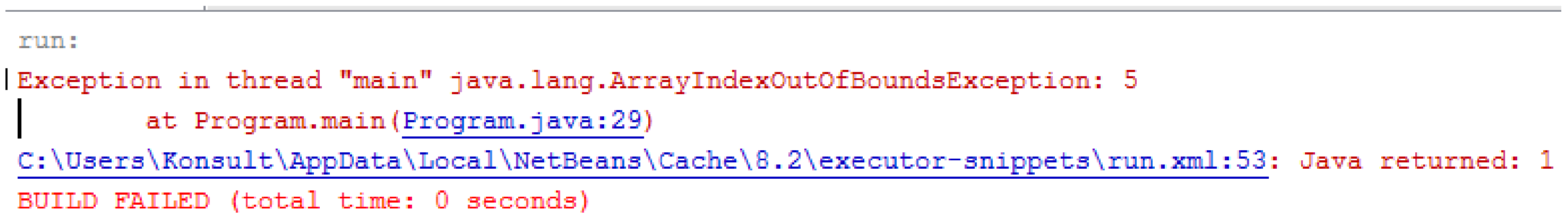
An unchecked exception is an exception that occurs at the time of execution. These are also called as *Runtime Exceptions*. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 4 in your program, and try to access the 6th element of the array, then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

If you compile and execute this program, you will get the following exception.



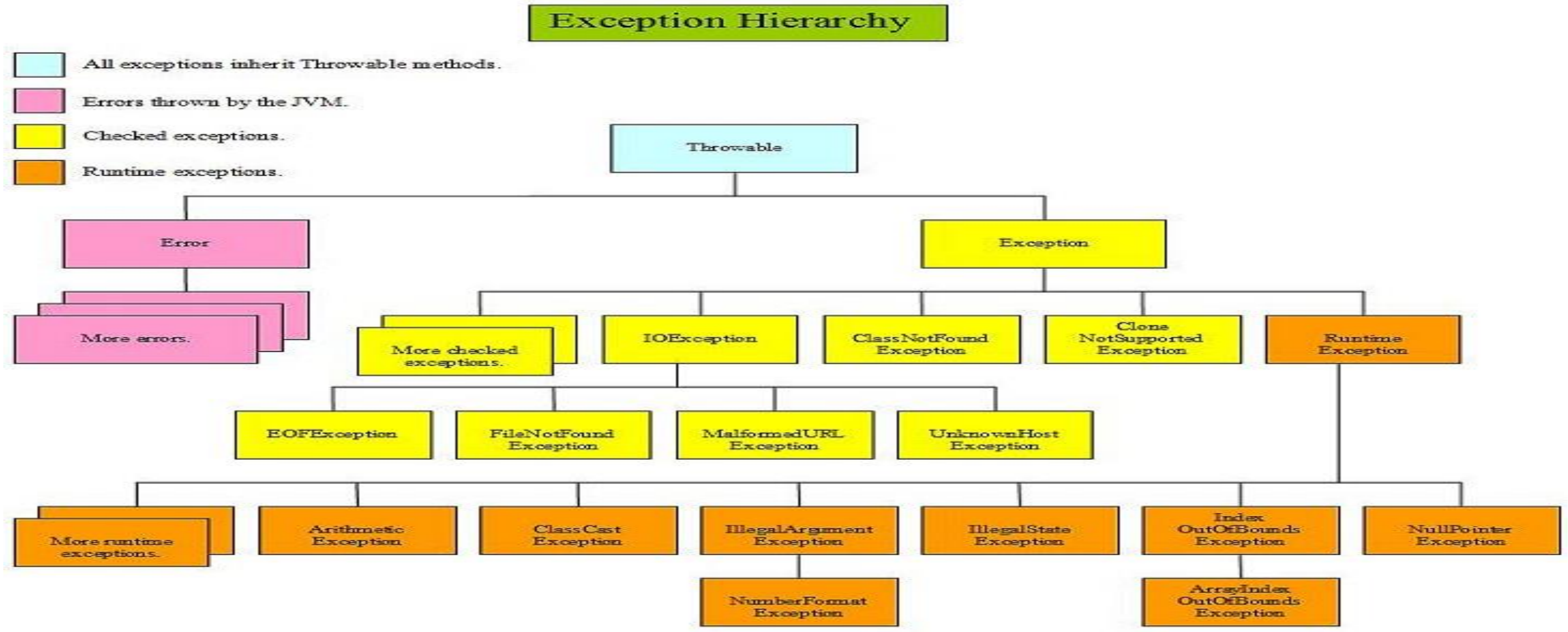
```
public static void main(String args[]) {
    int num[] = {1, 2, 3, 4};
    System.out.println(num[5]);
}
```



```
run:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Program.main(Program.java:29)
C:\Users\Konsult\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

Exception Hierarchy

All exception *classes* are subtypes of the java.lang.Exception *class*. The exception *class* is a subclass of the Throwable *class*. Other than the exception *class*, there is another subclass called Error, which is derived from the Throwable *class*. Errors are abnormal conditions that happen in case of critical failures. These are not handled by the Java program. Example: JVM is out of memory. Normally, programs cannot recover from errors. The Exception *class* has two main subclasses: IOException *class* and RuntimeException *class*.

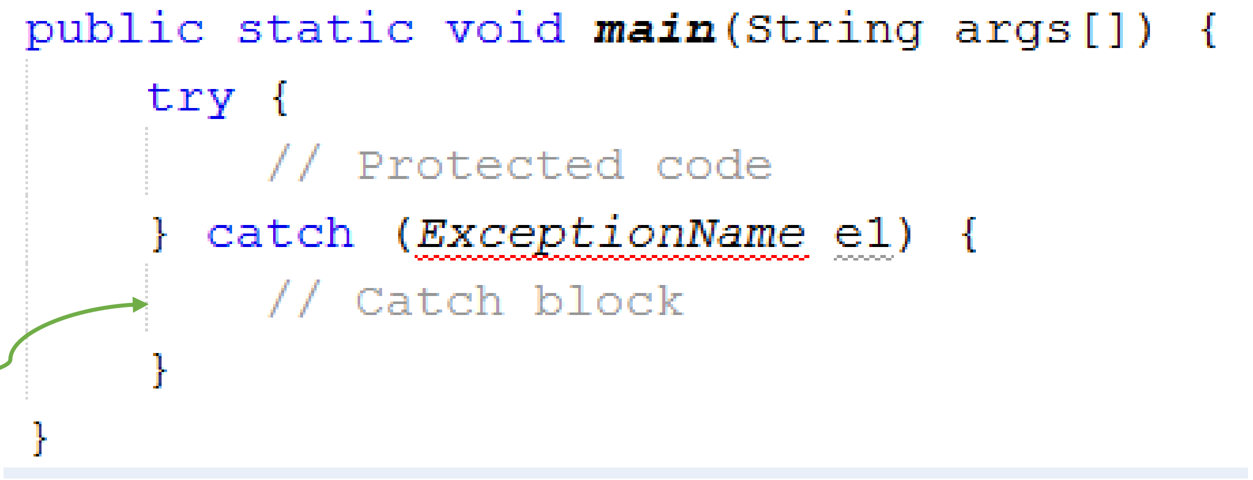


Exception Methods

No.	Method & Description
1	**public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString()** Returns the name of the class concatenated with the result of getMessage().
4	**public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	**public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**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.

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



```
public static void main(String args[]) {
    try {
        // Protected code
    } catch (ExceptionName e1) {
        // Catch block
    }
}
```

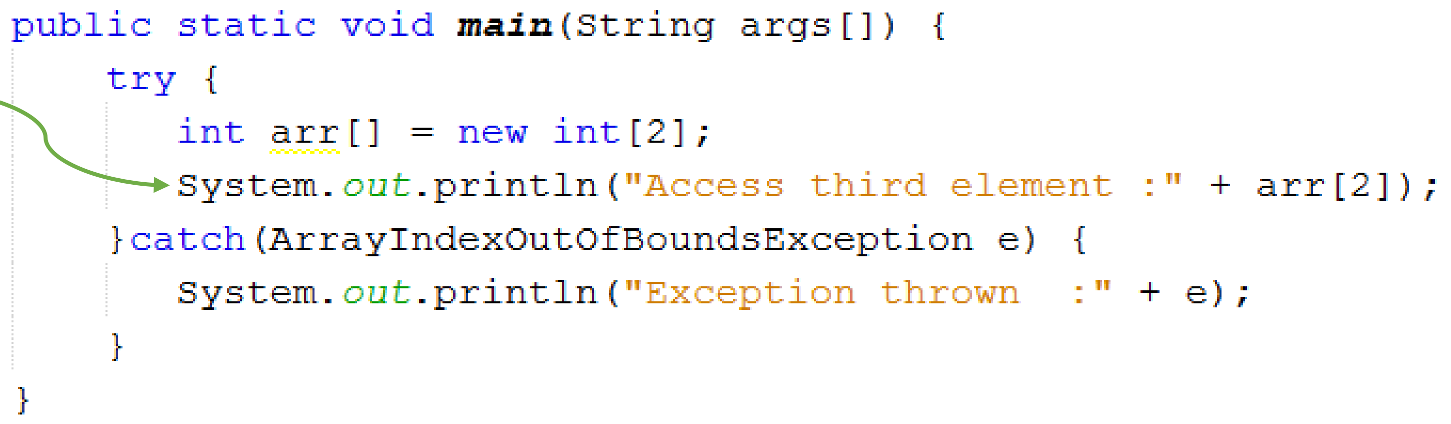
The code which is prone to exceptions is placed in the *try* block. When an exception occurs, the exception that occurred is handled by the *catch* block associated with it. Every *try* block should be immediately followed either by a *catch* block or a *finally* block.

A *catch* statement involves declaring the type of exception that you are trying to *catch*. If an exception occurs in protected code, the *catch* block (or blocks) that follows the *try* is checked. If the type of exception that occurred is listed in a *catch* block, the exception is passed to the *catch* block in the same way as an argument is passed as a method parameter.

Catching Exceptions - Example

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

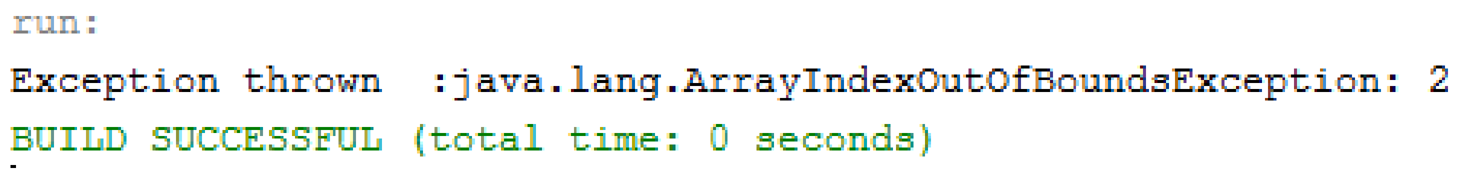
Example



```
public static void main(String args[]) {
    try {
        int arr[] = new int[2];
        System.out.println("Access third element :" + arr[2]);
    }catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Exception thrown  :" + e);
    }
}
```

This will produce the following result

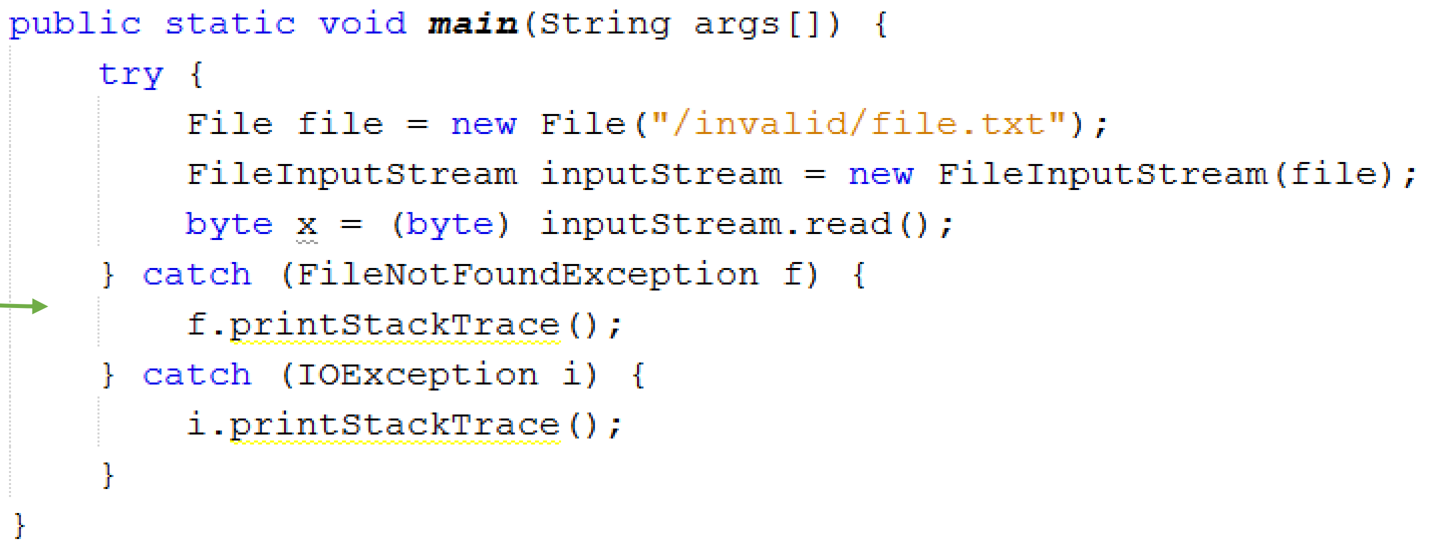
Result



```
run:
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 2
BUILD SUCCESSFUL (total time: 0 seconds)
```

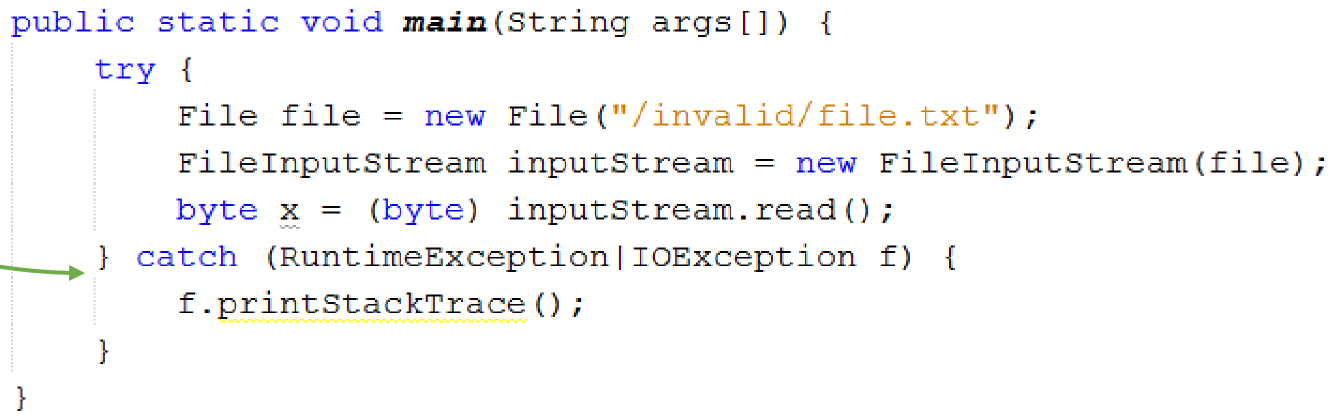
Catching Multiple Exceptions

Here is a code segment showing how to use multiple *try*/*catch* statements.



```
public static void main(String args[]) {
    try {
        File file = new File("/invalid/file.txt");
        FileInputStream inputStream = new FileInputStream(file);
        byte x = (byte) inputStream.read();
    } catch (FileNotFoundException f) {
        f.printStackTrace();
    } catch (IOException i) {
        i.printStackTrace();
    }
}
```

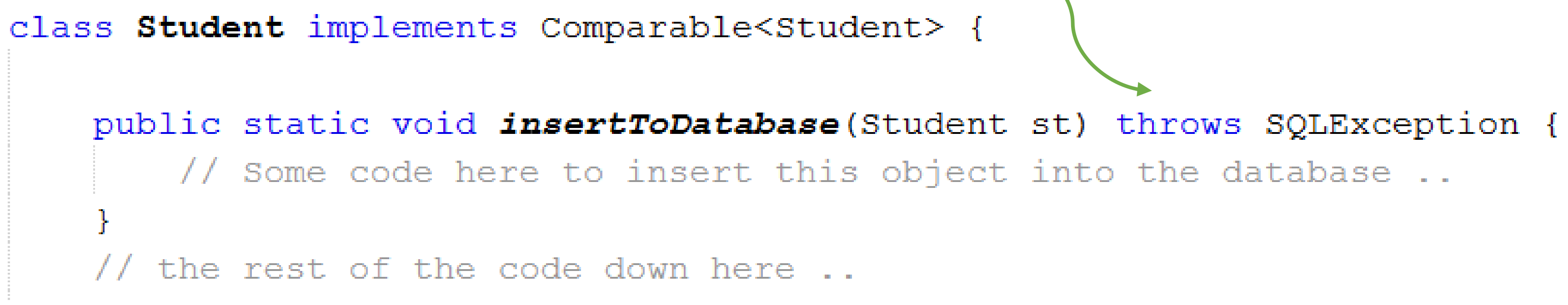
Since Java 7, you can handle more than one exception using a single *catch* block. This feature simplifies the code. Here is how you would do it



```
public static void main(String args[]) {
    try {
        File file = new File("/invalid/file.txt");
        FileInputStream inputStream = new FileInputStream(file);
        byte x = (byte) inputStream.read();
    } catch (RuntimeException|IOException f) {
        f.printStackTrace();
    }
}
```

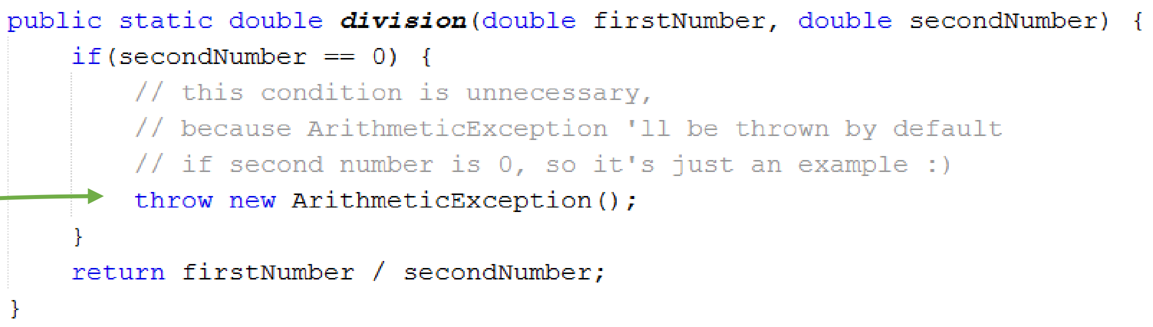
The Throws VS Throw Keywords

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.



```
class Student implements Comparable<Student> {

    public static void insertToDatabase(Student st) throws SQLException {
        // Some code here to insert this object into the database ..
    }
    // the rest of the code down here ..
```

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



```
public static double division(double firstNumber, double secondNumber) {
    if(secondNumber == 0) {
        // this condition is unnecessary,
        // because ArithmeticException 'll be thrown by default
        // if second number is 0, so it's just an example :)
        throw new ArithmeticException();
    }
    return firstNumber / secondNumber;
}
```

Try to understand the difference between throws and throw *keywords*. The throws is used to postpone the handling of a checked exception, and throw is used to invoke an exception explicitly.

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.

A *finally* block appears at the end of the *catch* blocks, and has the following syntax.

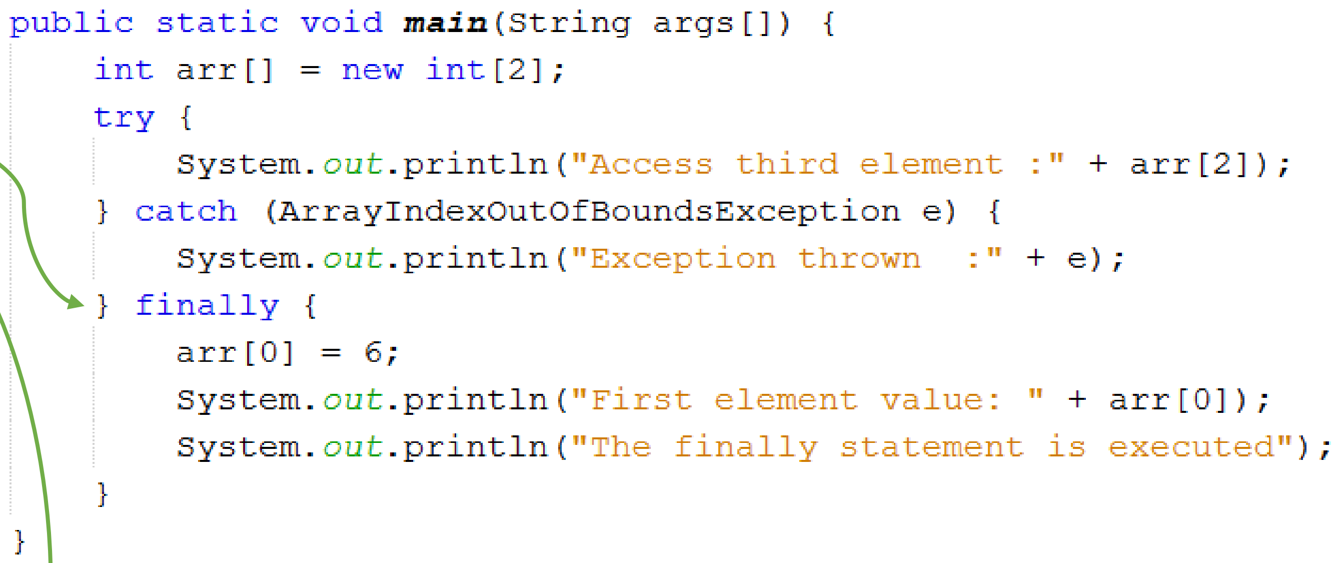
This will produce the following result

A catch clause cannot exist without a try statement.

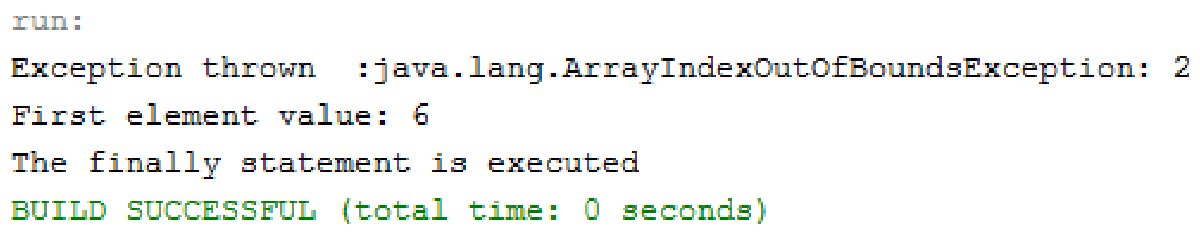
It is not compulsory to have finally clauses whenever a try/catch block is present.

The try block cannot be present without either catch clause or finally clause.

No code can be present in between the try, catch, finally blocks.



```java
public static void main(String args[]) {
    int arr[] = new int[2];
    try {
        System.out.println("Access third element :" + arr[2]);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Exception thrown  :" + e);
    } finally {
        arr[0] = 6;
        System.out.println("First element value: " + arr[0]);
        System.out.println("The finally statement is executed");
    }
}
```

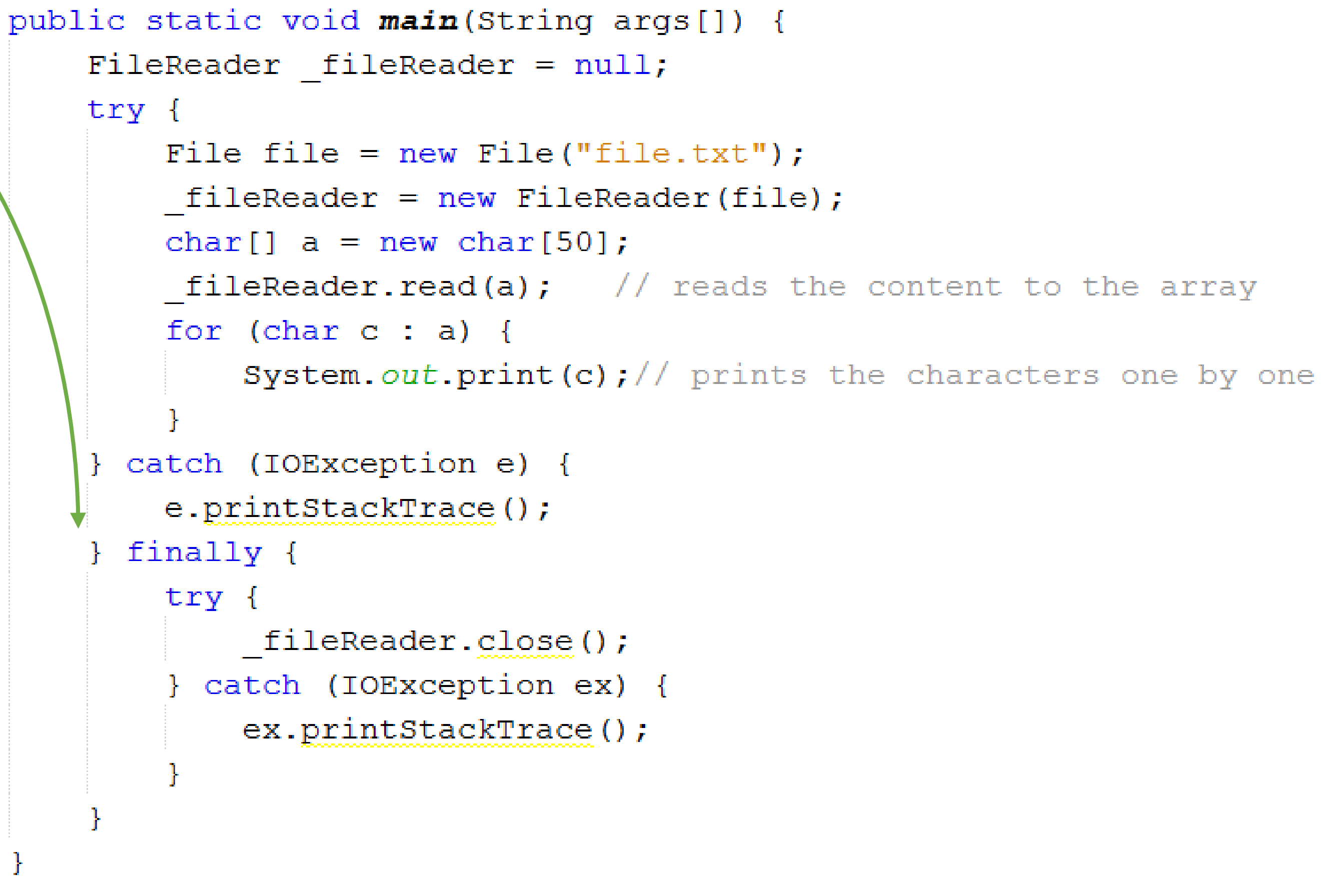


```
run:
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 2
First element value: 6
The finally statement is executed
BUILD SUCCESSFUL (total time: 0 seconds)
```

The try-with-resources

Generally, when we use any resources like streams, connections, etc, we have to close them explicitly using a *finally* block. In the following program, we are reading data from a file using **FileReader** and we are closing it using a *finally* block.

try-with-resources, also referred to as automatic resource management, is a new exception handling mechanism that was introduced in Java 7, which automatically closes the resources used within the *try catch* block. To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block. Following is the syntax of *try-with-resources* statement.

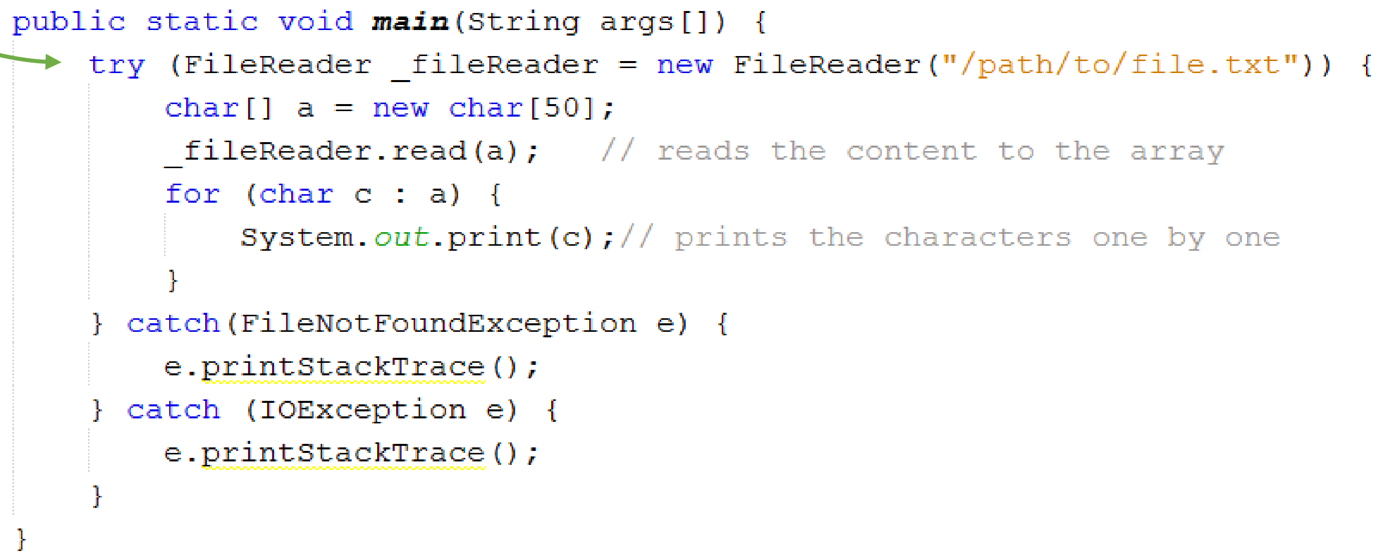


```
public static void main(String args[]) {
    FileReader _fileReader = null;
    try {
        File file = new File("file.txt");
        _fileReader = new FileReader(file);
        char[] a = new char[50];
        _fileReader.read(a);   // reads the content to the array
        for (char c : a) {
            System.out.print(c);// prints the characters one by one
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            _fileReader.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

The try-with-resources

Generally, when we use any resources like streams, connections, etc, we have to close them explicitly using a *finally* block. In the following program, we are reading data from a file using **FileReader** and we are closing it using a *finally* block.

try-with-resources, also referred to as automatic resource management, is a new exception handling mechanism that was introduced in Java 7, which automatically closes the resources used within the *try catch* block. To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block. Following is the syntax of *try-with-resources* statement.



```
public static void main(String args[]) {
    try (FileReader _fileReader = new FileReader("/path/to/file.txt")) {
        char[] a = new char[50];
        _fileReader.read(a);    // reads the content to the array
        for (char c : a) {
            System.out.print(c);// prints the characters one by one
        }
    } catch(FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

The try-with-resources

The following points are to be kept in mind while working with *try-with-resources* statement.

- In order to use a *class* with *try-with-resources* statement, it should implement **AutoCloseable** interface and the **close()** method, it gets invoked automatically at runtime.
- You can declare more than one *class* in *try-with-resources* statement.
- While you declare multiple *classes* in the try block of *try-with-resources* statement, these *classes* are closed in reverse order.
- Except for the declaration of resources within the parenthesis, everything is the same as in a normal *try*/*catch* block.
- The resource declared in *try* gets instantiated just before the start of the *try* block.
- The resource declared at the *try* block is implicitly declared as final.

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 children 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.

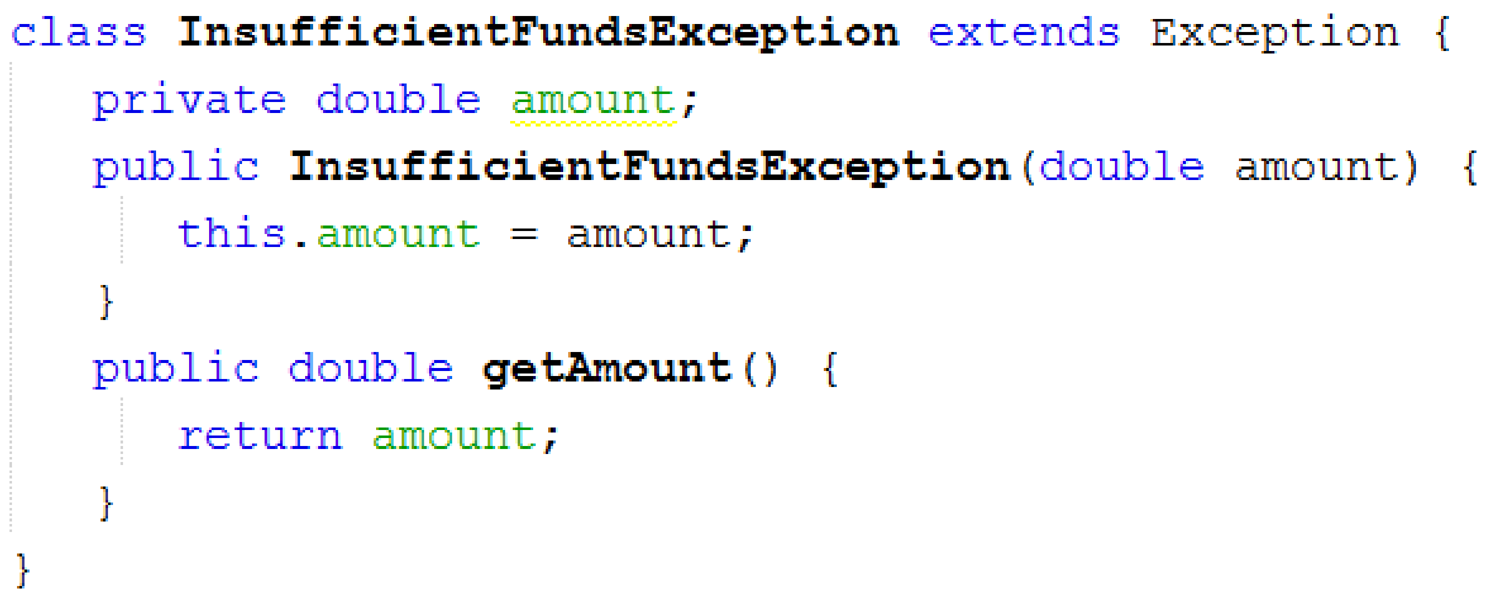
We can define our own Exception class as the following

```
class MyException extends Exception {

}
```

You just need to extend the predefined Exception class to create your own Exception. These are considered to be checked exceptions.

User-defined Exceptions - Example

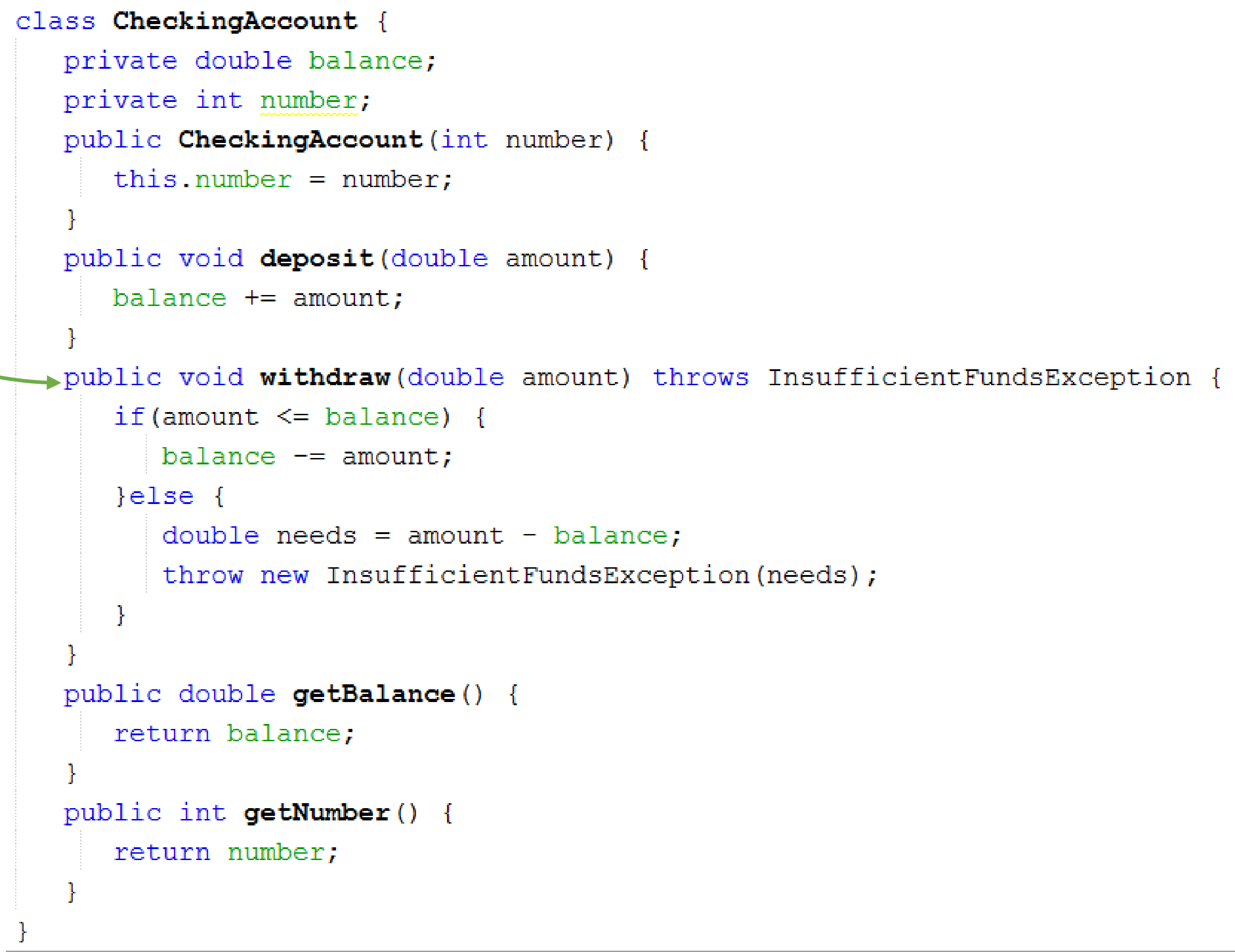
The following InsufficientFundsException class is a *user-defined* exception that extends the Exception class, making it a checked exception.
An exception class is like any other class, containing useful fields and methods.



```
class InsufficientFundsException extends Exception {
    private double amount;
    public InsufficientFundsException(double amount) {
        this.amount = amount;
    }
    public double getAmount() {
        return amount;
    }
}
```

User-defined Exceptions - Example

In order to demonstrate the use of user-defined exception, the following **CheckingAccount** class contains a **withdraw()** method that throws an InsufficientFundsException

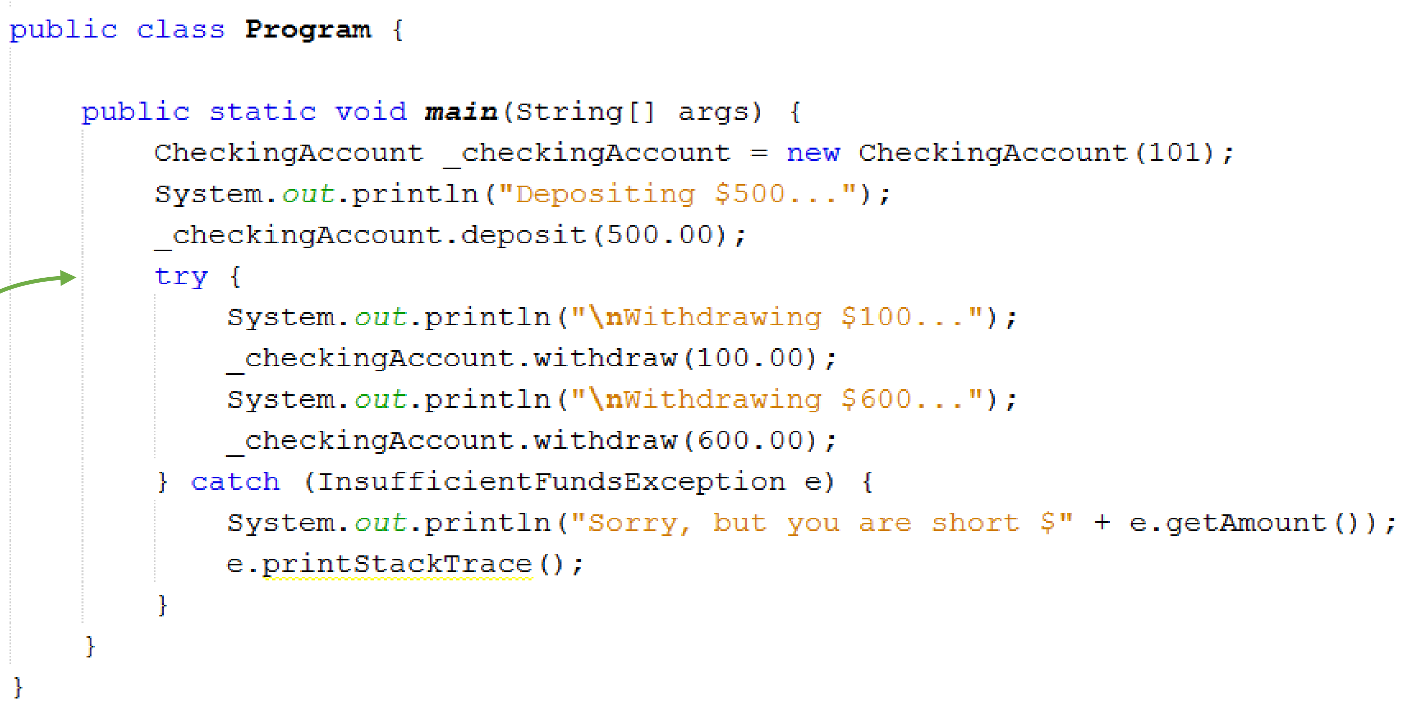


```java
class CheckingAccount {
    private double balance;
    private int number;
    public CheckingAccount(int number) {
        this.number = number;
    }
    public void deposit(double amount) {
        balance += amount;
    }
    public void withdraw(double amount) throws InsufficientFundsException {
        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 - Example

In order to demonstrate the use of user-defined exception, the following **CheckingAccount** class contains a **withdraw()** method that throws an InsufficientFundsException

The following ***BankDemo*** program demonstrates invoking the **deposit()** and **withdraw()** methods of **CheckingAccount**.



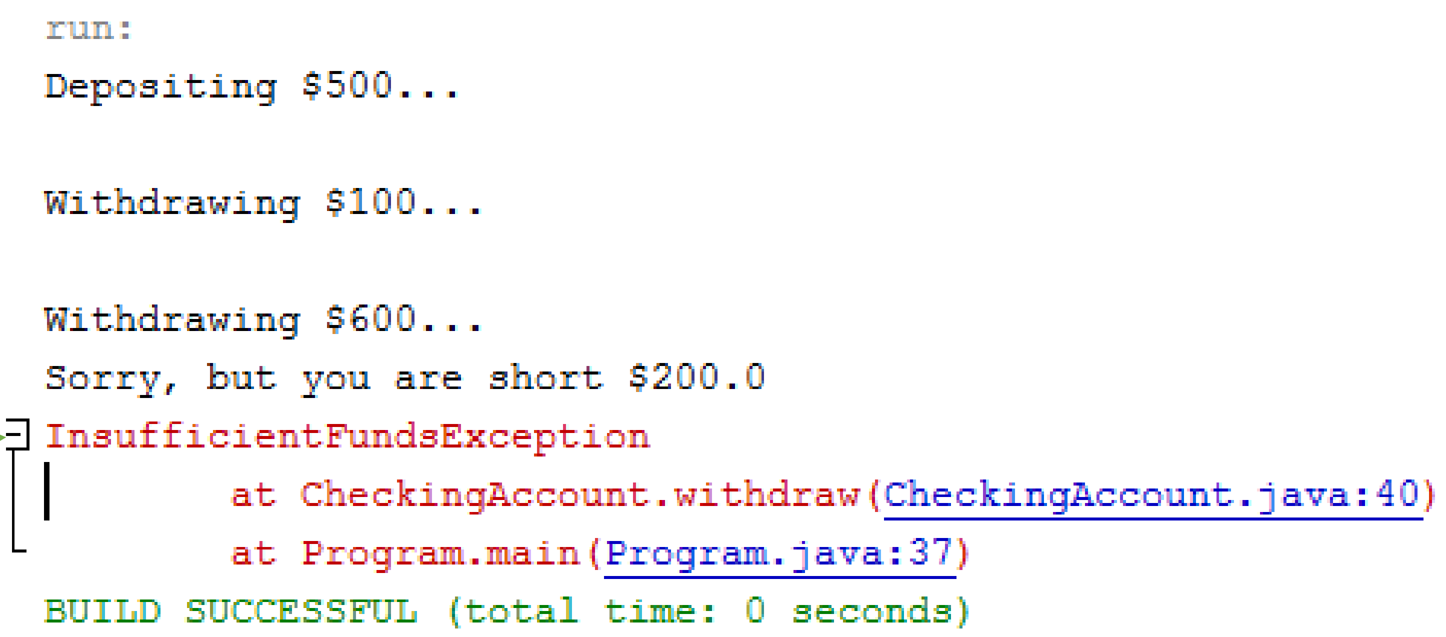
```
public class Program {

    public static void main(String[] args) {
        CheckingAccount _checkingAccount = new CheckingAccount(101);
        System.out.println("Depositing $500...");
        _checkingAccount.deposit(500.00);
        try {
            System.out.println("\nWithdrawing $100...");
            _checkingAccount.withdraw(100.00);
            System.out.println("\nWithdrawing $600...");
            _checkingAccount.withdraw(600.00);
        } catch (InsufficientFundsException e) {
            System.out.println("Sorry, but you are short $" + e.getAmount());
            e.printStackTrace();
        }
    }
}
```

User-defined Exceptions - Example

In order to demonstrate the use of user-defined exception, the following **CheckingAccount** class contains a **withdraw()** method that throws an InsufficientFundsException

The following ***BankDemo*** program demonstrates invoking the **deposit()** and **withdraw()** methods of **CheckingAccount**.

Compile all the above three files and run *BankDemo*. This will produce the following result



```
run:
Depositing $500...

Withdrawing $100...

Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
        at CheckingAccount.withdraw(CheckingAccount.java:40)
        at Program.main(Program.java:37)
BUILD SUCCESSFUL (total time: 0 seconds)
```

Questions?