Java Programming

Exception Handling and File IO

By Kevin 2022/2023

Exception

- Java provides a way to handle certain kinds of special conditions
- An **exception** is an object that signals the occurrence of an unusual event during the execution of a program
- When a piece code of, usually methods, encounters error, it **throws an exception**.

Exception

```
int noOfBiscuits = 30;
System.out.println("Enter number of people: ");
int people = scanner.nextInt();

System.out.printf("Each gets %d biscuits with %d left",
 noOfBiscuits / people, noOfBiscuits % people);
```

```
Enter number of people:
9
Each gets 3 biscuits with 3 left
```

- This work just fine for good user.

Exception (con't)

- What if the user enter unpredictable inputs?

```
Enter number of people:
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.main(Test.java:10)
```

- This code **throws an exception**!
- This exception is called `ArithmeticException`.
- We can patch this piece of code like

```
if (noOfPeople <= 0)  {
    System.out.println("Invalid input!");
} else ...
```

- In general, we never know when error[1] happens!

[1] Error is not the same as exception, we will talk about that later.

Two ways of handling exceptions

img: memes

Method 1 - Throws the exception

- The simplest way is to propagate the problem to your caller method
- The caller should then handle the problem, e.g. warning the user, retry the action, abort the action, etc..
- To do it, add `throws Exception` after your method

```
void riskyMethod() throws Exception {
    Scanner scanner = new Scanner(System.in);
    int noOfBiscuits = 30;
    System.out.println("Enter number of people: ");
    int people = scanner.nextInt();

    System.out.printf("Each gets %d biscuits with %d left",
    noOfBiscuits / people, noOfBiscuits % people);
}
void caller() {
    ...  //caller must handle the exception
    riskyMethod();
}
```

Method 1 - Throws the exception

```
void riskyMethod() throws Exception {
    ...
}
```

- A method labelled `throws Exception` tells the caller that this method may throws Exception
- The caller must either:
  1. Handle the problem
  2. Propagate the problem (label itself with `throws Exception` too)

Method 2 - Handle with Try-catch

- An exception can be handled by a `try-catch` block.
- Risky code is placed inside the `try` block and the response to the exception is placed inside the `catch` block.

```
try {
    System.out.println("Inside the try block.");
    int a = 10 / 0;
    System.out.println("This will not shown");
} catch (Exception e) {
    System.out.println("Inside the catch block.");
}
```

```
Inside the try block.
Inside the catch block.
```

Method 2 - Handle with Try-catch

```
void riskyMethod() throws Exception {
    Scanner scanner = new Scanner(System.in);
    int noOfBiscuits = 30;
    System.out.println("Enter number of people: ");
    int people = scanner.nextInt();

    System.out.printf("Each gets %d biscuits with %d left",
    noOfBiscuits / people, noOfBiscuits % people);
}
void caller() {
    //caller must handle the exception
    try {
        riskyMethod();
    } catch (Exception e) {
        System.out.println("The riskyMethod throws me an exception!");
    }
}
```

Another Example of Exception

```
System.out.println("Enter an integer: ");
Scanner scanner = new Scanner(System.in);
int x = scanner.nextInt();
```

```
Enter an integer:
 abc
Exception in thread "main" java.util.InputMismatchException
```

- Scanner does not expect the user to enter a non-integer input.
- When it encounter the error, it will keep the token in the buffer.
- It returns that token in the next `next()` or `nextInt()`.

Handling the Exception with a loop

```
Scanner scanner = new Scanner(System.in);
boolean error = false;
int x;
do {
  try {
    System.out.println("Enter an integer: ");
    x = scanner.nextInt();
    error = false;
  } catch (Exception e) {
    System.out.println("Input error! Not an integer!");
    error = true;
    scanner.next(); //the token is still in the scanner. Skip it!
  }

} while (error);
```

Javadoc of nextInt

According to the doc of `Scanner.nextInt()` it is possible to throw InputMismatchException

nextInt

`public int nextInt()`

Scans the next token of the input as an `int`.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Returns:

the int scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

Create an exception

- Some methods like `Scanner.nextInt()` will throw exceptions
- You can also create methods that throw exceptions and let the caller to handle that for you.
- Reason behind: only the caller knows what to do when exception happen!
- Done by the statement `throw new Exception("Error Message");`
- When an exception is thrown, it must either be thrown the its caller (method 1) or be caught in a try-catch block (method 2).

Create an exception

```
/**
 * Match the name from the list and return the corresponding value
 */
int findValue(String[] nameList, int[] values, String name) {
    for (int i = 0; i < nameList.length; i++)
        if (nameList[i].equals(name))
            return values[i];
    return ??;
}
```

- What should ?? be if the name is not found from the input?
- 0? -1? What if 0 or -1 are also possible values from the value list?

Create an exception

```
/**
 * Match the name from the list and return the corresponding value
 */
int findValue(String[] nameList, int[] values, String name) throws Exception {
  for (int i = 0; i < nameList.length; i++)
      if (nameList[i].equals(name))
          return values[i];
  throw new Exception(name + " is not found!");
}
```

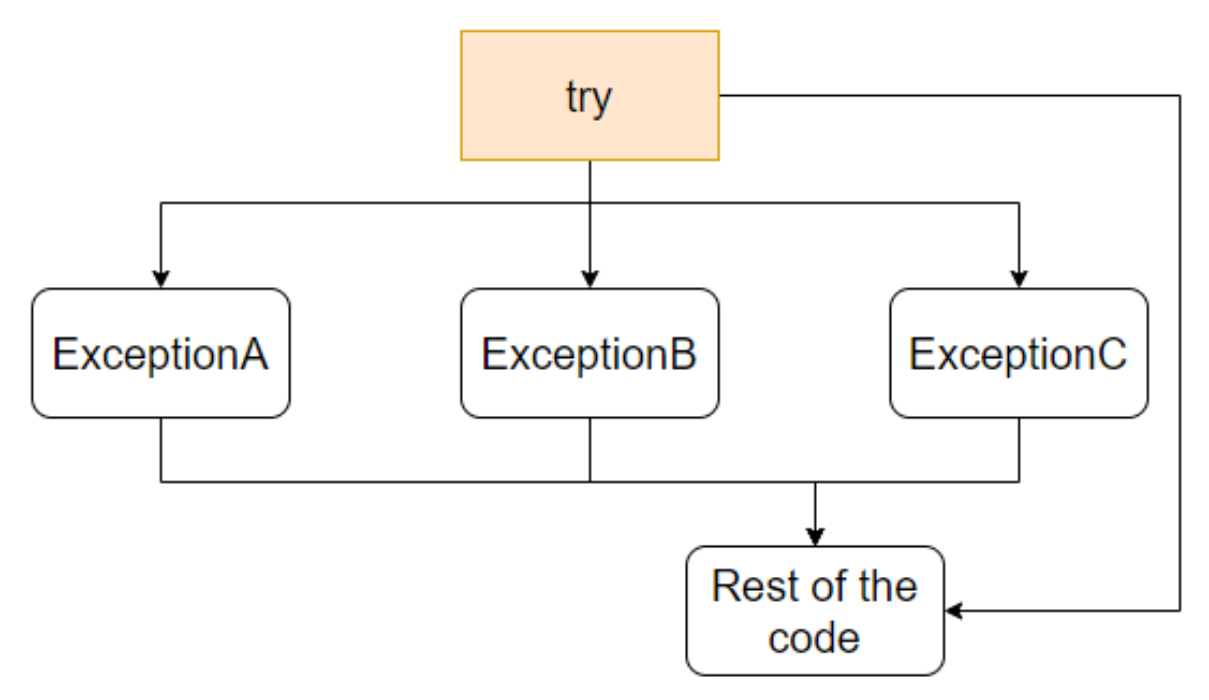
Mixing Method 1 and 2?

- It is possible to throw an exception and do try catch at the same time?
- Once an exception is caught in a try-catch block, it will not be thrown

```
void method() throws Exception {
    try {
        riskyMethod();
    } catch (Exception e) {
        ...
    }
}
void caller() {
    try {
        method();
    } catch (Exception e) {
        System.out.println("Never thrown");
    }
}
```

Multiple Catch

- We saw different types of exceptions earlier, e.g.: `InputMismatchException`, `ArithmeticException`
- We might want to **handle different exceptions differently**
- We can have multiple catch blocks after a try block.
- Each catch block declares a specific exception that you want to handle.



Multiple Catch

```
Scanner scanner = new Scanner(System.in);
try {
    System.out.println("Enter a number");
    int num = scanner.nextInt();
    System.out.printf("100 / %d = %d", num, 100/num);

} catch (ArithmeticException e) {
    System.out.println("0 can't be used as divisor.");

} catch (InputMismatchException e) {
    System.out.println("This is not a integer");

}
```

Multiple Catch

```
void riskyMethod() throws InputMismatchException {
    Scanner scanner = new Scanner(System.in);
    try {
        System.out.println("Enter a number");
        int num = scanner.nextInt();
        System.out.printf("100 / %d = %d", num, 100/num);

    } catch (ArithmeticException e) {
        System.out.println("0 can't be used as divisor.");

    // } catch (InputMismatchException e) {
    //     System.out.println("This is not a integer");

    }
}
```

- Mixing method 1 and 2 may make sense if you want the caller to handle a specific exception.

Finally Block

- A `finally` block is **optionally** added after a catch block.
- The `finally` block **always executes** when the try block exits, even when:
  - Exception is thrown; or
  - A return statement is executed
- Provide a good way for programmer to clean up the resource.

Syntax

```
try {
    //open some resource
} catch (Exception e) {

} finally {
    //clean up the resource
}
```

Finally Block - All print Finally

```
try {
    System.out.println("Try");
    throw new Exception("-");
} catch (Exception e) {
    System.out.println("Catch");
} finally {
    System.out.println("Finally");
}
```

```
try {
    System.out.println("Try");
    return;
} catch (Exception e) {
    System.out.println("Catch");
} finally {
    System.out.println("Finally");
}
```

```
try {
    System.out.println("Try");
} catch (Exception e) {
    System.out.println("Catch");
} finally {
    System.out.println("Finally");
}
```

```
try {
    System.out.println("Try");
    throw Exception("-");
} catch (Exception e) {
    System.out.println("Catch" + 1 / 0);
} finally {
    System.out.println("Finally");
}
```

Different Type of Exceptions

- Not all exceptions are required to be caught.
- There are three different categories of exceptions:
  - Checked Exception
  - Error
  - Unchecked Exception

Different Type of Exceptions

Checked Exception

- A well-written application **must** anticipate and recover from
- If a method would throw a checked exception, the caller must handle it
- e.g. `FileReader` throws `FileNotFoundException` that must be handled.

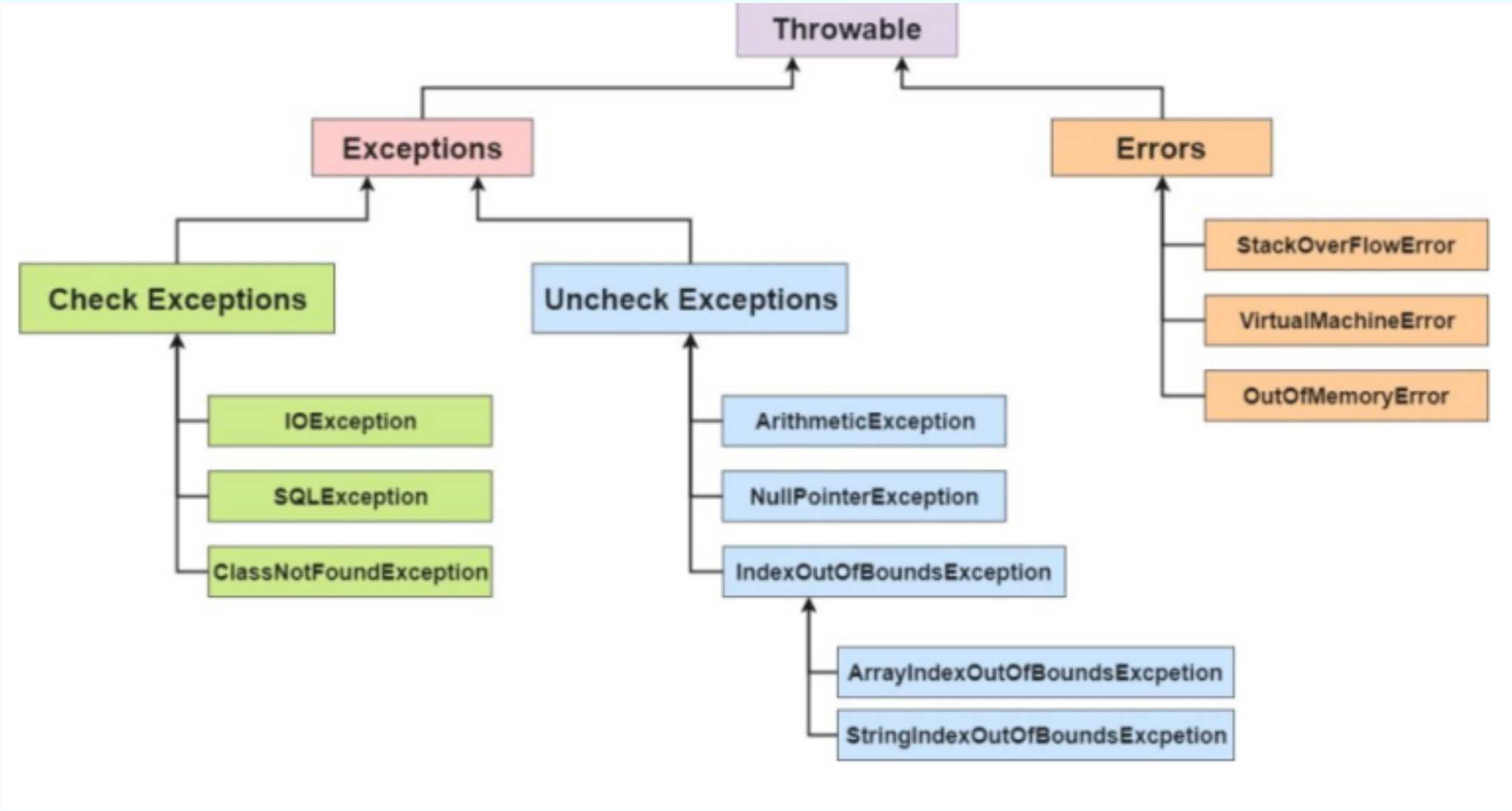
Error

- Exceptional conditions that are external to the application
- Not expect to handle it
- e.g. `OutOfMemoryError` when there isn't enough memory

Unchecked Exception (a.k.a. RuntimeException):

- Exceptional conditions that are internal to the application
- Indicate programming bugs, logic errors or improper use of API
- e.g. `ArrayOutOfBoundException`: `array[-1]`, `ArithmeticException`: `1/0`

Exceptions Hierarchy



img src: https://sematext.com/blog/java-exceptions/

Checked Exception Must be Caught

```
27          File inputFile = new File( pathname: "inputFile.txt");
28          Scanner scanner = new Scanner(inputFile);
29          int i = 0;
30          while (scanner.hasNext()) {
31              String token = scanner.next();
32              System.out.println((++i) + ":" + token);
33          }
34      }
35
```

```
ns  C:\Users\kevinw\IdeaProjects\lecture\tmp\src\Test.java:28:27
    java: unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown
```

Javadoc of nextInt

- `InputMismatchException`, `NoSuchElementException`, and `IllegalStateException` are all **uncheck** exceptions.
- No need to use try/catch when use `nextInt()`

nextInt

```
public int nextInt()
```

Scans the next token of the input as an `int`.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Returns:

the `int` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

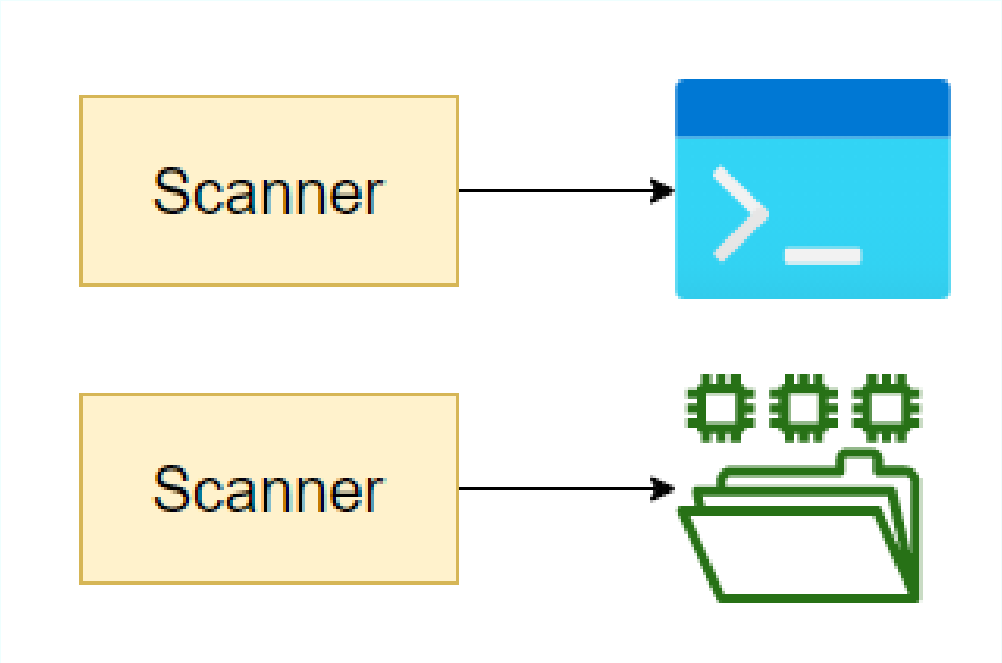
`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

File I/O

Reading from Files

- We use `Scanner` to read inputs from console.
- Using `next()`, `nextInt()`, `nextDouble()` to read data.
- We can also use `Scanner` to read inputs from **files**.
- A scanner acts like **an adaptor** to connect different sources.



File Object

- A file in Java is represented by a File Object.
- A file object is created by

```
File inputFile = new File("InputFile.txt");
```

- The system will try to locate the file from the project folder.
- Having a File object allows us to read or write a local file through a media of a reader or a writer.

File Object

- Under linux/unix system directory separator is `/`
- Under windows system directory separator is `\`
- A file placed under another directory can be reached using linux/unix convention(even on Windows!):

```
File inputFile = new File("directory/InputFile.txt");
```

- Note: don't use the character `\`! This is an escape character.

Scanner Object

- A scanner can be used in reading a file:

```
File inputFile = new File("inputfile.txt");
Scanner scanner = new Scanner(inputFile);
```

- The **Scanner object** reads text from a file instead of `System.in`.
- We can use the Scanner methods to read data from the input file
- `nextInt()`, `nextDouble()`, `next()`, `nextLine()`, etc.
- The statement `Scanner scanner = new Scanner(inputFile);` throws `FileNotFoundException` which is a checked exception.

Scanner Object

```
try {
    File inputFile = new File("inputFile.txt");
    Scanner scanner = new Scanner(inputFile);
    String firstLine = scanner.nextLine();
    System.out.println("The first line is: " + firstLine);
    ...
} catch (Exception e) {
    System.out.println("No such file found.");
}
...
```

```
The first line is: abc
```

- The statement `s.nextLine()` throws `NoSuchElementException` if there is nothing left to read.
- Unlike reading user inputs, it does not stall the program and wait for new content written in the file!

Loop until reaching the end

- We can use the methods `scanner.hasNext()` or `scanner.hasNextLine()` to check if there is another token (or line) in the file for reading.

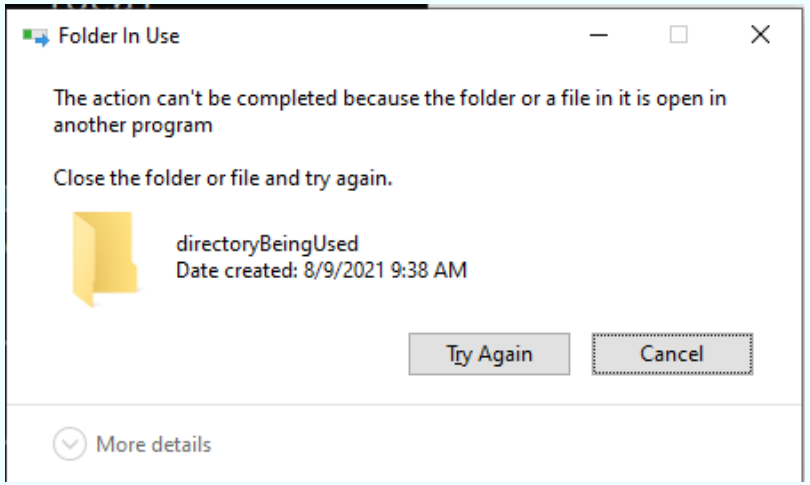
```
try {
    File inputFile = new File("inputFile.txt");
    Scanner scanner = new Scanner(inputFile);
    int i = 0;
    while (scanner.hasNext()) {
        String token = scanner.next();
        System.out.println( (++i) + ":" + token);
    }
    ...
} catch (Exception e) {
    System.out.println("No such file found.");
}
```

```
inputFile.txt:
abc def
hij
```

```
1:abc
2:def
3:hij
```

Close the opened file scanner.

- A file is locked when a program is reading it.
- You need to close the file scanner when you have finished.
- Close it by `scanner.close()`



All together

```
Scanner scanner = null;

try {
    File inputFile = new File("inputFile.txt");
    scanner = new Scanner(inputFile);
    int i = 0;
    while (scanner.hasNext()) {
        String token = scanner.next();
        System.out.println( (++i) + ":" + token);
    }
} catch (Exception e) {
    System.out.println("No such file found.");
} finally {
    if (scanner != null)
        scanner.close();
}
```

Replaced by Try-resource block

- Java provide a short hand for all *closeable* resource - a resource that needs to be close after use
- We call this a Try-resource block, syntax below

```
File inputFile = new File("inputFile.txt");
try (Scanner scanner = new Scanner(inputFile) ) {
    ...
} catch (Exception) {

} //close the scanner automatically
```

Reading file code 2

```
File inputFile = new File("inputFile.txt");
try (Scanner scanner = new Scanner(inputFile)) {

    int i = 0;
    while (scanner.hasNext()) {
        String token = scanner.next();
        System.out.println( (++i) + ":" + token);
    }
} catch (Exception e) {
    System.out.println("No such file found.");
}
```

Data on a file

- Create a `PrintWriter` object to open a file for writing

```
PrintWriter out = new PrintWriter("Output.txt");
```

- Write data to the file by `print` or `println` method

```
out.print("Java ");
out.println("Programming");
```

Data on a file

- JVM will decide when the data will be written to the file for optimization
- Do **flush** the content if to force the data to be written to the file

```
out.flush();
```

- Close the `PrintWriter` after using it

```
out.close();
```

- This will **overwrite** the content of the existing file!

Overwrite Data on a file

```
try (PrintWriter out = new PrintWriter("outputFile.txt")) {
  out.print("This ");
  out.print("is ");
  out.println("line 1. ");
  out.println("and line 2 ");
  out.flush();
} catch (Exception e) {
  System.out.println("Cannot write a file");
}
```

Appending Data on a file

- Create a `FileWriter` object as follows:

```
FileWriter writer = new FileWriter("outputFile.txt", true);
```

- Writing on this FileWriter will append data in the file.
- Connect it with a `PrintWriter` as follows:

```
PrintWriter out = new PrintWriter(writer);
```

- Remember to flush and close

```
out.print("Hello ");
out.print("World");
out.flush();
out.close();
writer.close();
```

Appending on a file

```
try (FileWriter writer = new FileWriter("outputFile.txt", true);
  PrintWriter out = new PrintWriter(writer)) {
  out.println("appending line");
  out.flush(); // flush the output as a good practice
} catch (Exception e) {
  System.out.println("Cannot append on the file");
}
```

- Two resources declared in the try-resource block will be close automatically!

Summary

Exception Handling

- Throws
- Try-catch
- Finally
- Try-resource
- Different types of Exceptions

File I/O

- File Object
- Scanner
- PrintWriter