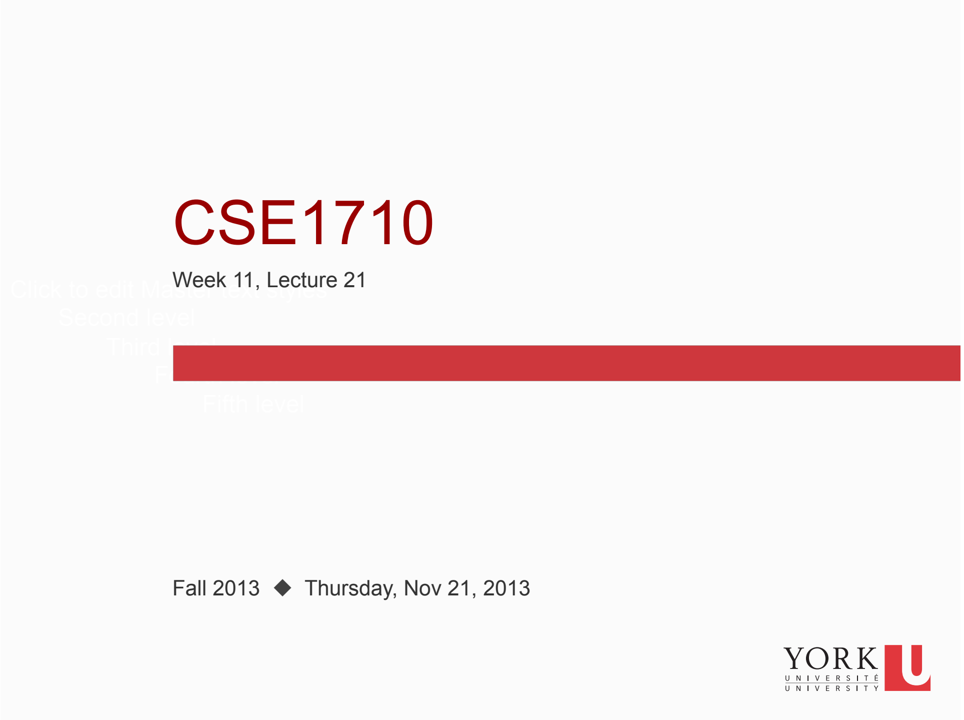
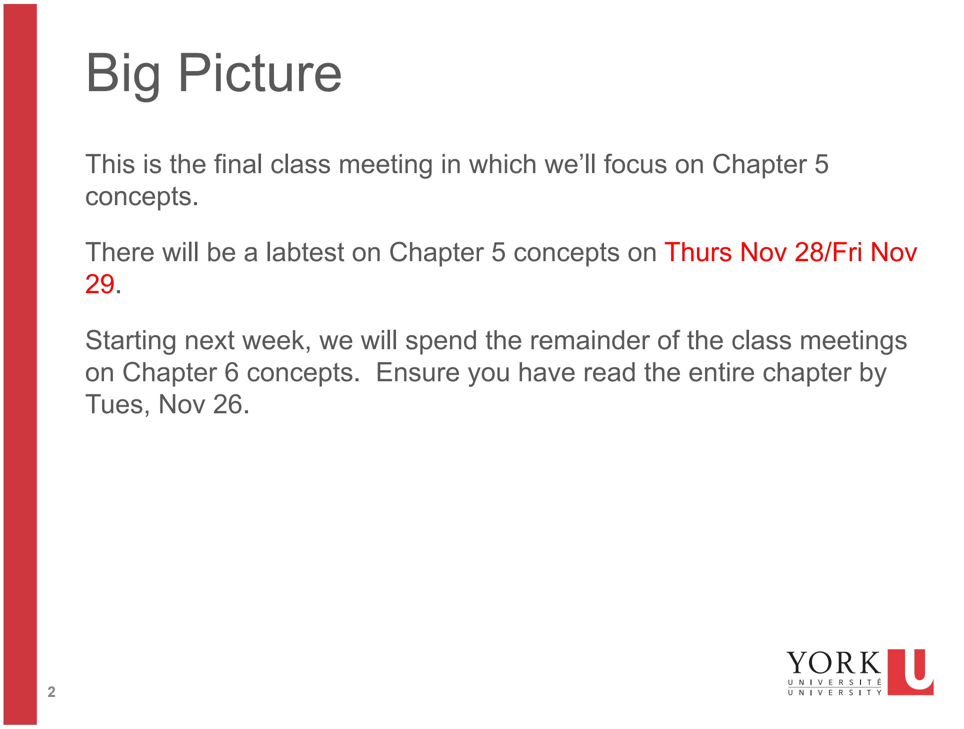


CSE1710

Week 11, Lecture 21

Fall 2013 ◆ Thursday, Nov 21, 2013



Big Picture

This is the final class meeting in which we'll focus on Chapter 5 concepts.

There will be a labtest on Chapter 5 concepts on Thurs Nov 28/Fri Nov 29.

Starting next week, we will spend the remainder of the class meetings on Chapter 6 concepts. Ensure you have read the entire chapter by Tues, Nov 26.

5.3.1 Input Validation

Validate means *to check that something meets* ***criteria*** *for being acceptable*

For instance:

- an input is a positive, non-zero integer
- an input is a string consisting of two, space-delimited tokens
- an input is a string that conforms to a particular format, such as: `letter-number-letter-number-letter-number`
  - or some other such format (foreshadowing: regular expressions in Ch 6)

It should be possible to instantiate the **CRITERIA** in a `boolean` expression

- if not, this is a clue that your criteria are **not sufficiently precise**

Motivation for Input Validation?

Because user inputs are often used subsequently as **values** that are…

- used in arithmetic calculations and other derivations
- passed as parameters to method and/or constructor invocations

You want to be certain that the values meet the pre-conditions of any services that are later used

Remember: if you, the client, **do not** meet the pre-condition of a service, then the provider is under no obligation at all to **follow the contract.**

This all relates to establishing the **correctness** of an application

Three Methods for Input Validation

- **Make the app crash**
  - use the `crash` service of `ToolBox` or the built-in functionality of services, such as `Scanner`’s `nextInt`
  - the app terminates with an exception
  - pretty rudimentary and basic, but better than no validation at all
- **Terminate the app (nicely, not with a crash)**
  - use an `if-else` construct. If the input fails the validation criteria, then skip the rest of the app
  - better than crashing, but still rudimentary
- **Provide feedback and allow retries**
  - use iteration. Use the validation criteria in the for loop’s condition.
  - Best option of the three.

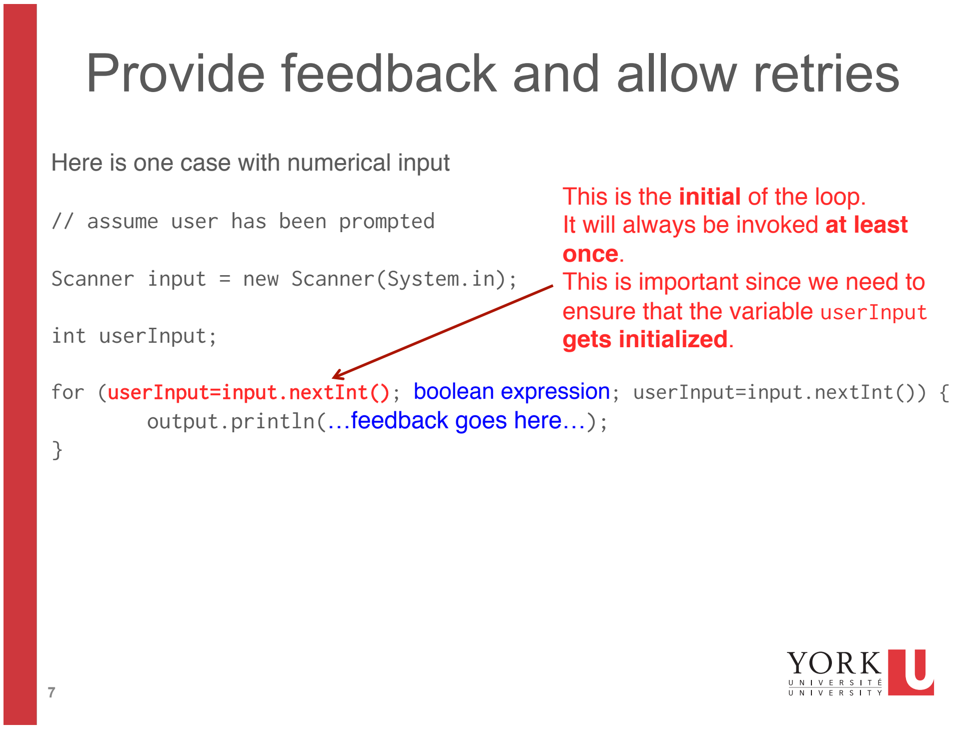
Provide feedback and allow retries

Here is one case with numerical input

```
// assume user has been prompted

Scanner input = new Scanner(System.in);

int userInput;

for (userInput=input.nextInt(); boolean expression; userInput=input.nextInt()) {
        output.println(...feedback goes here...);
}
```

Examples of boolean expressions

- `userInput > 0`
- `userInput >= 0 && userInput <= 10`
- `userInput % 2 == 0`
- `userInput % 2 != 0`
- `Math.abs(userInput – 100) <= 5`

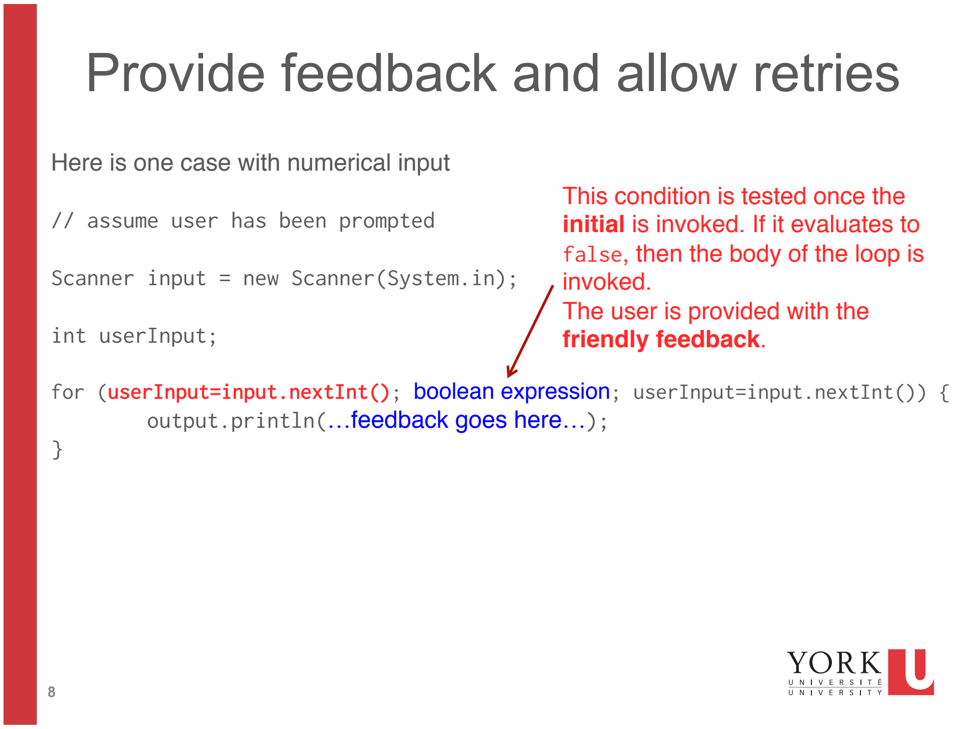


Provide feedback and allow retries

Here is one case with numerical input

```
// assume user has been prompted

Scanner input = new Scanner(System.in);

int userInput;

for (userInput=input.nextInt(); boolean expression; userInput=input.nextInt()) {
        output.println(…feedback goes here…);
}
```

This is the **initial** of the loop.
It will always be invoked **at least once**.
This is important since we need to ensure that the variable `userInput` **gets initialized**.



Provide feedback and allow retries

Here is one case with numerical input

```
// assume user has been prompted

Scanner input = new Scanner(System.in);

int userInput;

for (userInput=input.nextInt(); boolean expression; userInput=input.nextInt()) {
        output.println(…feedback goes here…);
}
```

This condition is tested once the **initial** is invoked. If it evaluates to `false`, then the body of the loop is invoked.
The user is provided with the **friendly feedback**.

Provide feedback and allow retries

Here is one case with numerical input

Once the user is provided with the friendly feedback, the **bottom** of the loop is invoked.
The bottom involves the `nextInt()` method. This method causes the program thread **to block** until the user types the next input and presses ‘enter’.

```
// assume user has been prompted

Scanner input = new Scanner(System.in);

int userInput;

for (userInput=input.nextInt(); boolean expression; userInput=input.nextInt()) {
        output.println(…feedback goes here…);
}
```

Provide feedback and allow retries

Here is one case with numerical input

Then the condition is tested once again…
and so on until the condition evaluates to `true`

```
// assume user has been prompted

Scanner input = new Scanner(System.in);

int userInput;

for (userInput=input.nextInt(); boolean expression; userInput=input.nextInt()) {
        output.println(…feedback goes here…);
}
```

Input Validation – Exception-Based Approach

```
boolean cond = amount < 0;

…

String msg = “The inputted amount was negative”;

…

ToolBox.crash(cond, msg);
```

Input Validation – Message-Based Approach

```
boolean cond = amount < 0;
…
String msg = “The inputted amount was negative”;
…
if (cond) {
      output.println(msg);
      }
else {
      //rest of program
}
```

- Now shifting topics away from input validation to File I/O

Abstraction of Output

By now you’ve typed the following statement a million times…

```
PrintStream output = System.out;
```

…and then you use the variable `output` like so…

```
output.printf(“Here are my weighty words.%n”);

output.println(“and some more words”);
```

Abstraction of Output

By now you've typed the following statement a million times…

```
PrintStream output = System.out;
```

here is a PrintStream variable

…and then you use the variable output like so…

here is the PrintStream variable in use

```
output.printf("Here are my weighty words.%n");

output.println("and some more words");
```

Abstraction of Output

Even though you could just as easily do this…

```
System.out.printf("Here are my weighty words.%n");

System.out.println("and some more words");
```

Abstraction of Output

Even though you could just as easily do this…

```
System.out.printf(“Here are my weighty words.%n”);

System.out.println(“and some more words”);
```

here is a specific `PrintStream` instance being used, namely the one that is assigned to the static field of the `System` class.

Abstraction of Output

… we coached you **NOT to use** the specific `PrintStream` instance

…and we coached you **to use** a `PrintStream` variable instead

the rationale is for the sake of abstraction…

NOW is finally the time to demonstrate WHY

Abstraction of Output

Suppose you want your output to go to a file instead of to the console.

If you abstracted your output using a `PrintStream` variable, then the change is SUPER EASY!

Instead of this:

```
PrintStream output = System.out;
```

Do this:

```
PrintStream output = new PrintStream(“file.txt”);
```

Abstraction of Output

Suppose you want your output to go to a file instead of to the console.

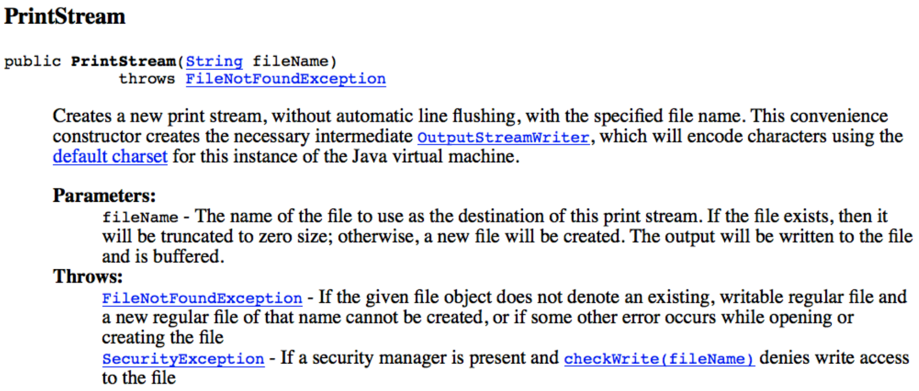
If you didn’t abstract your output using a `PrintStream` variable and instead used `System.out.println(…)` everywhere, then you need to go and change **each and every single statement**.

About the `PrintStream` constructor

But wait! Is it really so easy?

```
PrintStream output = new PrintStream(“file.txt”);
```

This statement is causing a compiler error. What gives?



PrintStream

```
public PrintStream(String fileName)
            throws FileNotFoundException
```

Creates a new print stream, without automatic line flushing, with the specified file name. This convenience constructor creates the necessary intermediate `OutputStreamWriter`, which will encode characters using the default charset for this instance of the Java virtual machine.

Parameters:

`fileName` - The name of the file to use as the destination of this print stream. If the file exists, then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.

Throws:

`FileNotFoundException` - If the given file object does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file

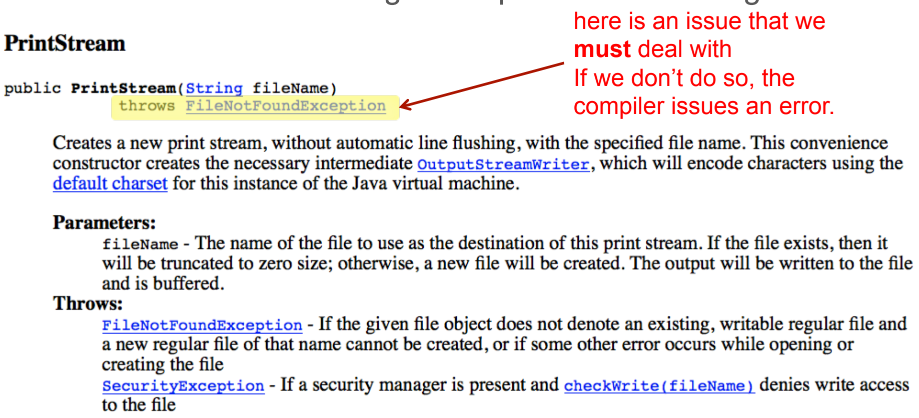
`SecurityException` - If a security manager is present and `checkWrite(fileName)` denies write access to the file

About the `PrintStream` constructor

But wait! Is it really so easy?

```
PrintStream output = new PrintStream(“file.txt”);
```

This statement is causing a compiler error. What gives?



PrintStream

```
public PrintStream(String fileName)
            throws FileNotFoundException
```

here is an issue that we **must** deal with
If we don’t do so, the compiler issues an error.

Creates a new print stream, without automatic line flushing, with the specified file name. This convenience constructor creates the necessary intermediate `OutputStreamWriter`, which will encode characters using the default charset for this instance of the Java virtual machine.

Parameters:

`fileName` - The name of the file to use as the destination of this print stream. If the file exists, then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.

Throws:

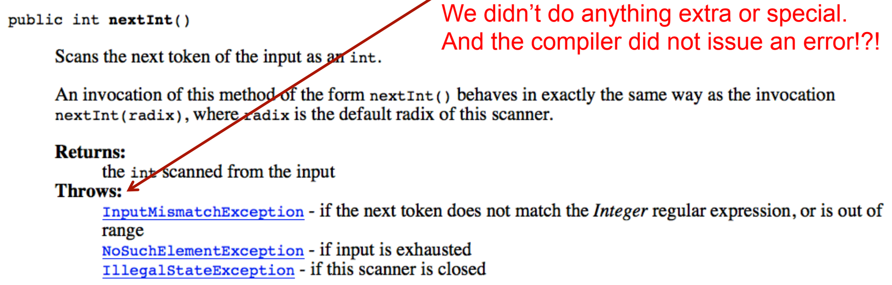
`FileNotFoundException` - If the given file object does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file

`SecurityException` - If a security manager is present and `checkWrite(fileName)` denies write access to the file

Services that (potentially) throw exceptions

So the constructor of `PrintStream` can potentially throw an exception. We've dealt with this sort of thing before, for instance when we use `Scanner`

The `nextInt()` method may potentially throw an exception.
We didn't do anything extra or special.
And the compiler did not issue an error!?!



```
public int nextInt()
```

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

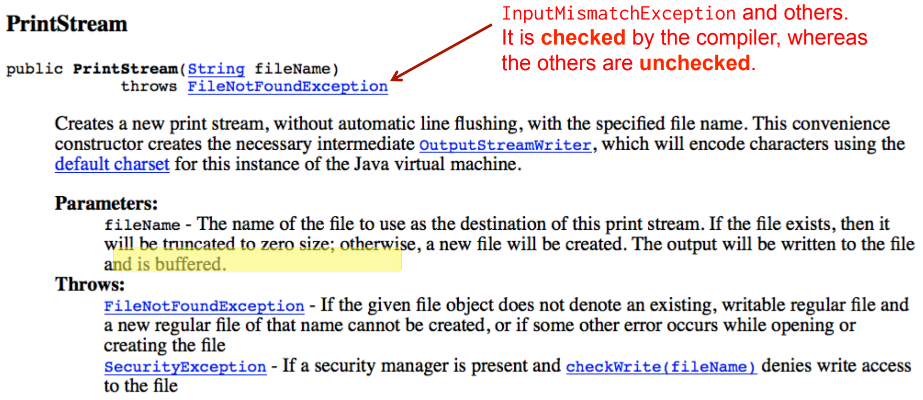
An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

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

About the `PrintStream` constructor

This type of exception is different from the `InputMismatchException` and others.
It is **checked** by the compiler, whereas the others are **unchecked**.



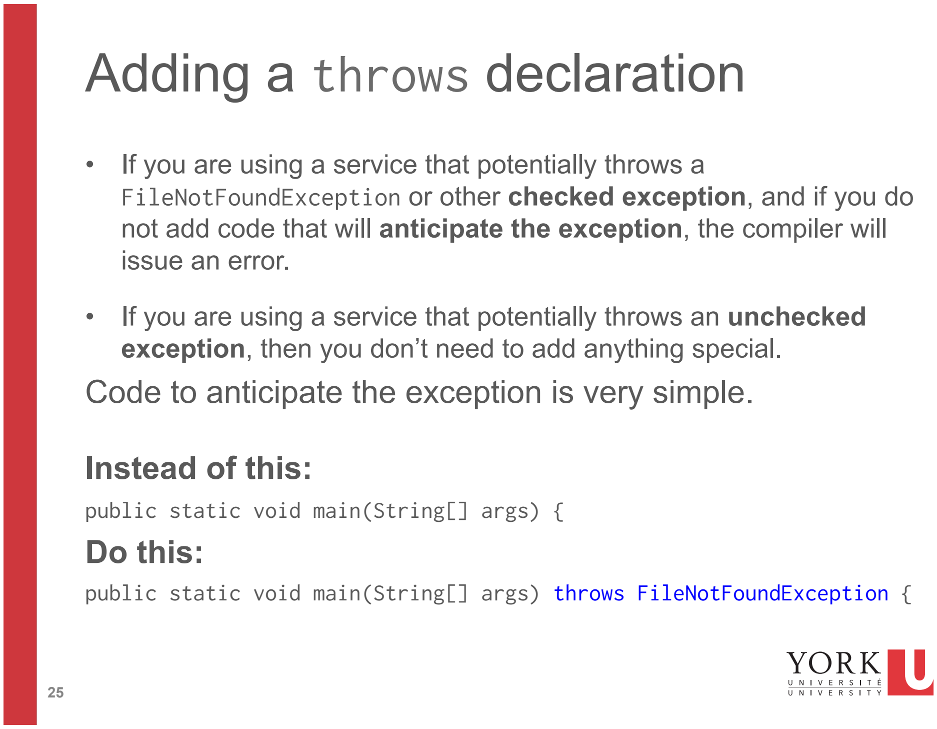
PrintStream

```
public PrintStream(String fileName)
            throws FileNotFoundException
```

Creates a new print stream, without automatic line flushing, with the specified file name. This convenience constructor creates the necessary intermediate `OutputStreamWriter`, which will encode characters using the default charset for this instance of the Java virtual machine.

Parameters:
`fileName` - The name of the file to use as the destination of this print stream. If the file exists, then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.

Throws:
`FileNotFoundException` - If the given file object does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file
`SecurityException` - If a security manager is present and `checkWrite(fileName)` denies write access to the file



Adding a `throws` declaration

- If you are using a service that potentially throws a `FileNotFoundException` or other **checked exception**, and if you do not add code that will **anticipate the exception**, the compiler will issue an error.
- If you are using a service that potentially throws an **unchecked exception**, then you don't need to add anything special.

Code to anticipate the exception is very simple.

Instead of this:

```
public static void main(String[] args) {
```

Do this:

```
public static void main(String[] args) throws FileNotFoundException {
```