COSC 121: Computer Programming II

Dr. Bowen Hui
University of British Columbia
Okanagan

A1 Feedback

- Marks posted last Friday
- Common mistakes Q1:
  - Incorrect visibility modifiers
  - Not initializing attributes
  - Not using constants consistently
  - Submitting only .class files and no .java files
- Common mistakes Q2:
  - Not using super.method() to call parent methods
  - Children class attributes declared unnecessarily protected

Quick Review

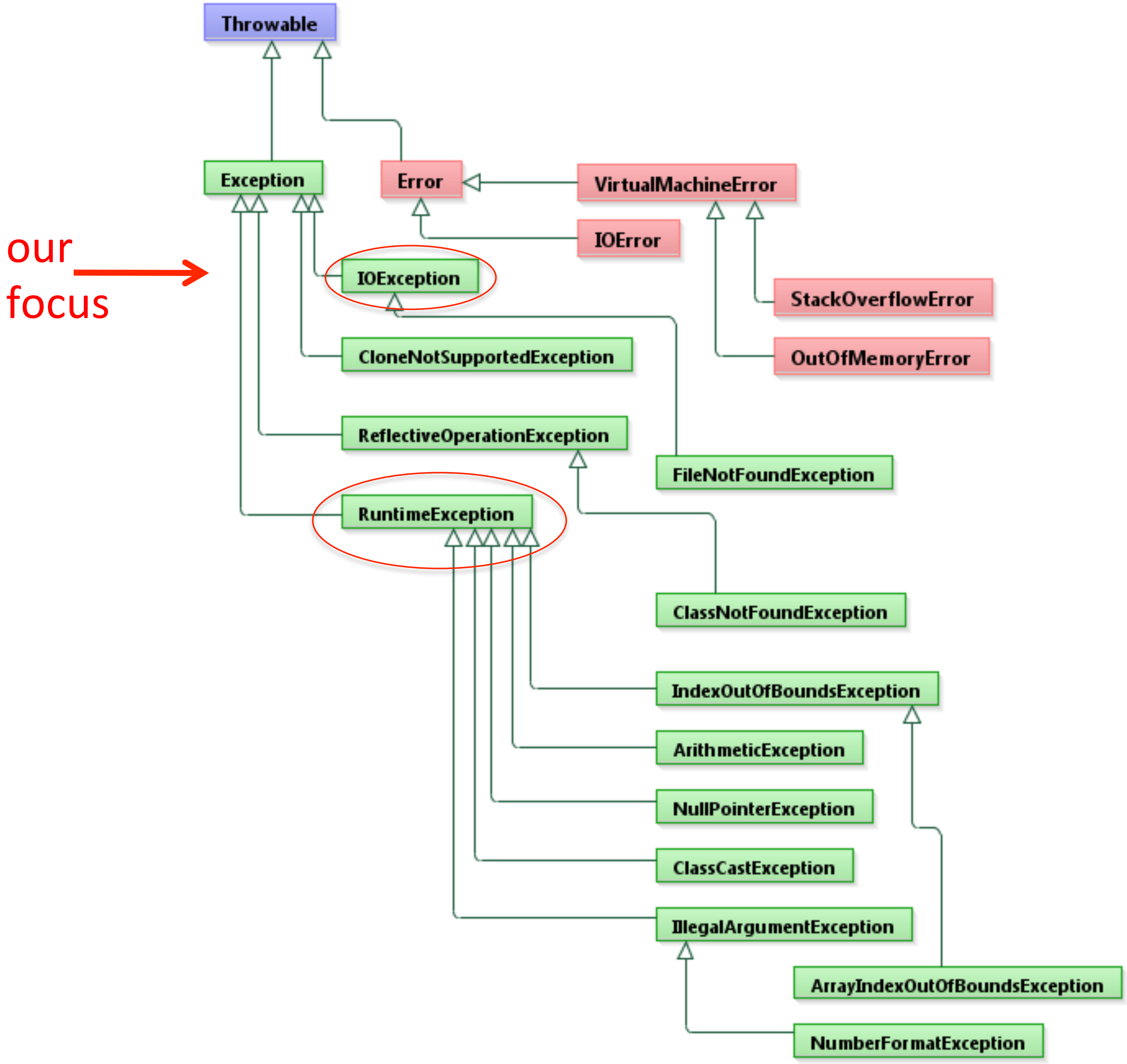
- Potential problems need to be anticipated
- Code should handle exceptions
  - Immediately
  - Handle elsewhere
- Custom exception classes can be created
- Today:
  - Files and associated exceptions

Checked vs. Unchecked Exceptions

- An exception is either checked or unchecked
- Checked exceptions:
  - Generally indicate invalid condition outside program, which can be detected at compile time
  - Requires programmer to explicitly handle it
  - Must be either:
    - Caught via `try` statement, or
    - Be listed in the `throws` clause of any method that may throw or propagate it
  - Otherwise, the compiler will issue an error

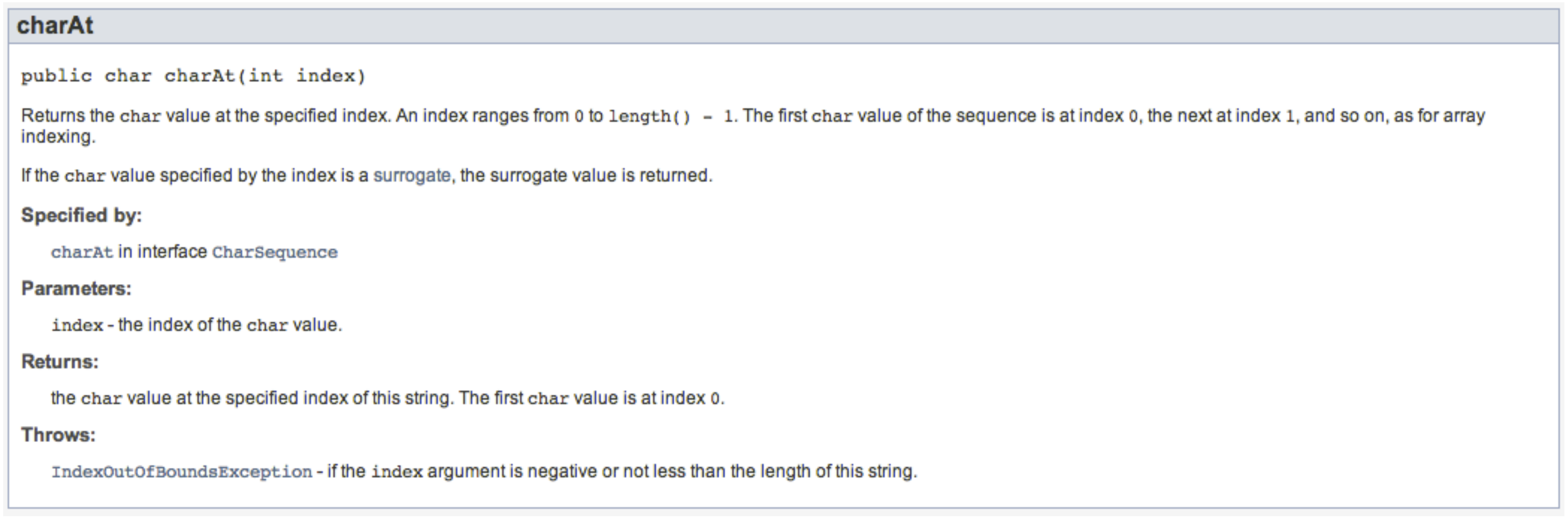
Checked vs. Unchecked Exceptions

- An exception is either checked or unchecked
- Unchecked exceptions:
  - Generally indicate program logic error that happens at run-time
  - Does not require explicit handling
  - Only unchecked exceptions in Java are objects of type `RuntimeException` or any of its descendants


Throwable
Exception
Error
VirtualMachineError
IOError
our focus
IOException
StackOverflowError
OutOfMemoryError
CloneNotSupportedException
ReflectiveOperationException
FileNotFoundException
RuntimeException
ClassNotFoundException
IndexOutOfBoundsException
ArithmeticException
NullPointerException
ClassCastException
IllegalArgumentException
ArrayIndexOutOfBoundsException
NumberFormatException

Java API Documentation

- Search online "Java String class" or other classes
- Read Oracle documentation
  - E.g., http://docs.oracle.com/javase/7/docs/api/java/lang/String.html
- Know which exceptions are thrown under different circumstances



charAt

```
public char charAt(int index)
```

Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`. The first `char` value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

If the `char` value specified by the index is a surrogate, the surrogate value is returned.

Specified by:

`charAt` in interface `CharSequence`

Parameters:

`index` - the index of the `char` value.

Returns:

the `char` value at the specified index of this string. The first `char` value is at index `0`.

Throws:

`IndexOutOfBoundsException` - if the `index` argument is negative or not less than the length of this string.

Working with Files

- Files may be various types (note: file extensions)
- Two main operations on files:
  - Read (when the file is used as input)
  - Write (when the file is used as output)
- Reading involves:
  - Open file, get info from file, close file
- Writing involves:
  - May involve reading operations
  - Open file, add info to file, close file
  - Create new file, add info to file, close file

IOException Class

- Operations performed by some I/O classes may throw an `IOException`
  - A file might not exist
  - Even if the file exists, a program may not be able to find it
  - The file exists but the program does not have the right access to open it
  - The file might not contain the kind of data we expect
- An `IOException` is a checked exception

I/O Streams

- A stream is a sequence of bytes that flow from a source to a destination
- In a program:
  - Read information from an input stream
  - Write information to an output stream
- A program can manage multiple streams simultaneously
  - E.g., read from multiple files at the same time
  - E.g., read from one file, write to another file

Standard I/O

- There are three standard I/O streams:
  - Standard output – defined by `System.out`
  - Standard input – defined by `System.in`
  - Standard error – defined by `System.err`
- We use `System.out` when we execute `println()` statements
- `System.out` and `System.err` typically represent the console window
- `System.in` typically represents keyboard input, which we've used many times with `Scanner`
  - E.g., `Scanner sysin = new Scanner( System.in );`

Reading from Text File

- Use `File` and `Scanner` classes

Reading from Text File

- Use `File` and `Scanner` classes
- Create new `File` object with file name
  - E.g., `File readFrom = new File( “test.txt” );`

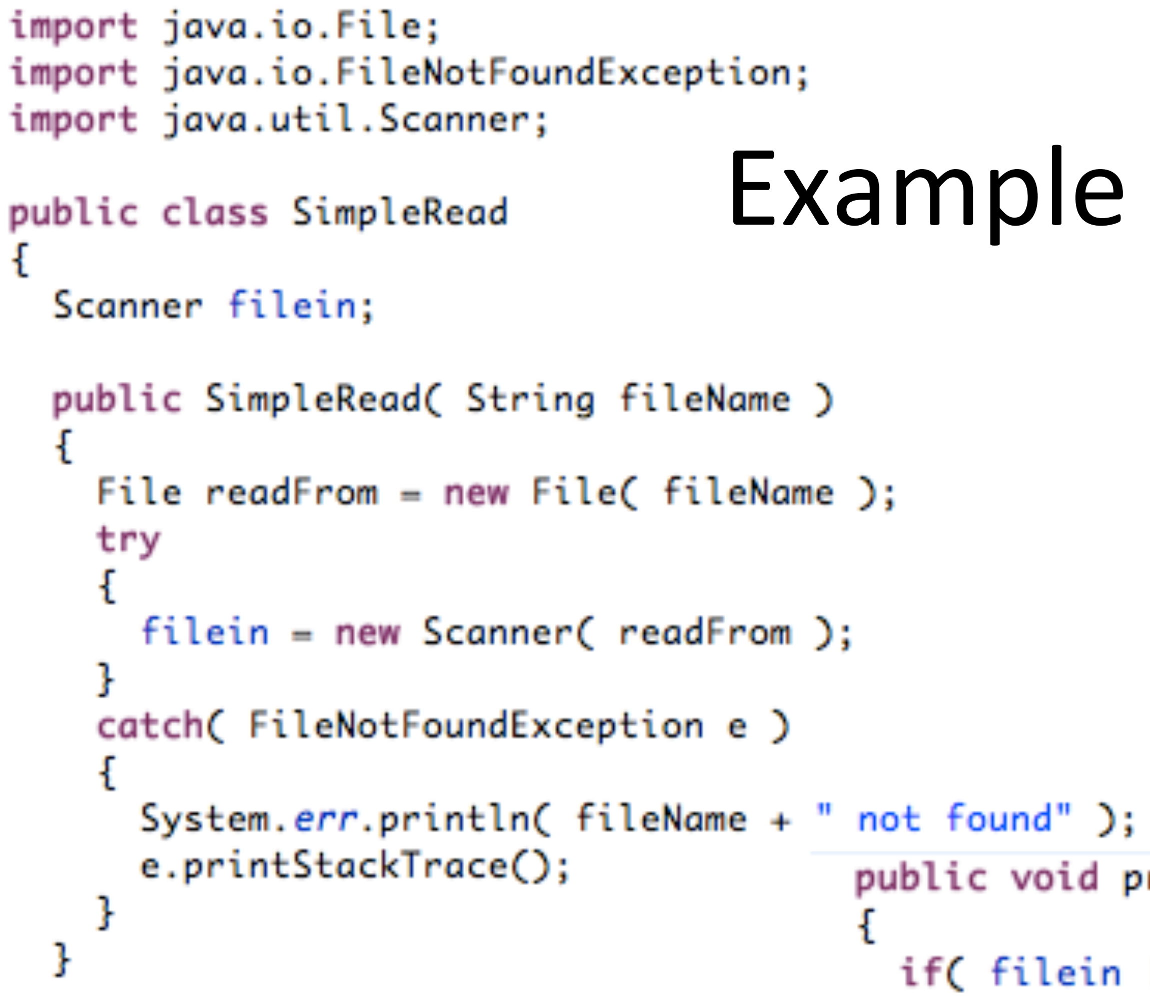
Reading from Text File

- Use `File` and `Scanner` classes
- Create new `File` object with file name
  - E.g., `File readFrom = new File( “test.txt” );`
- Pass `File` object to `Scanner` during creation
  - E.g., `Scanner inFile = new Scanner( readFrom );`
  - This creates new input file stream
  - Needs to handle `FileNotFoundException`

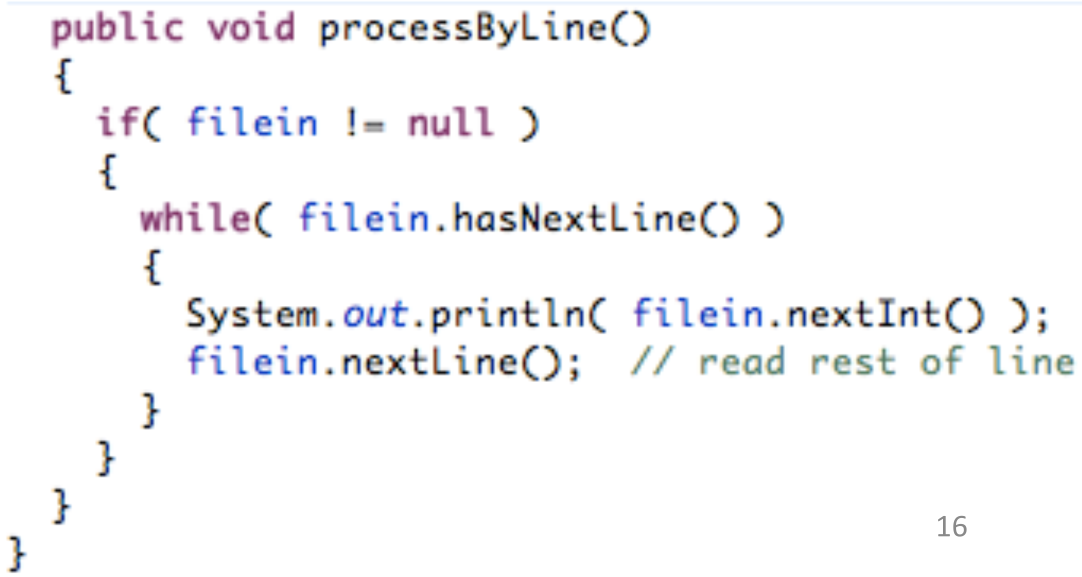
Reading from Text File

- Use `File` and `Scanner` classes
- Create new `File` object with file name
  - E.g., `File readFrom = new File( “test.txt” );`
- Pass `File` object to `Scanner` during creation
  - E.g., `Scanner inFile = new Scanner( readFrom );`
  - This creates new input file stream
  - Needs to handle `FileNotFoundException`
- Use `Scanner` object as usual

Example

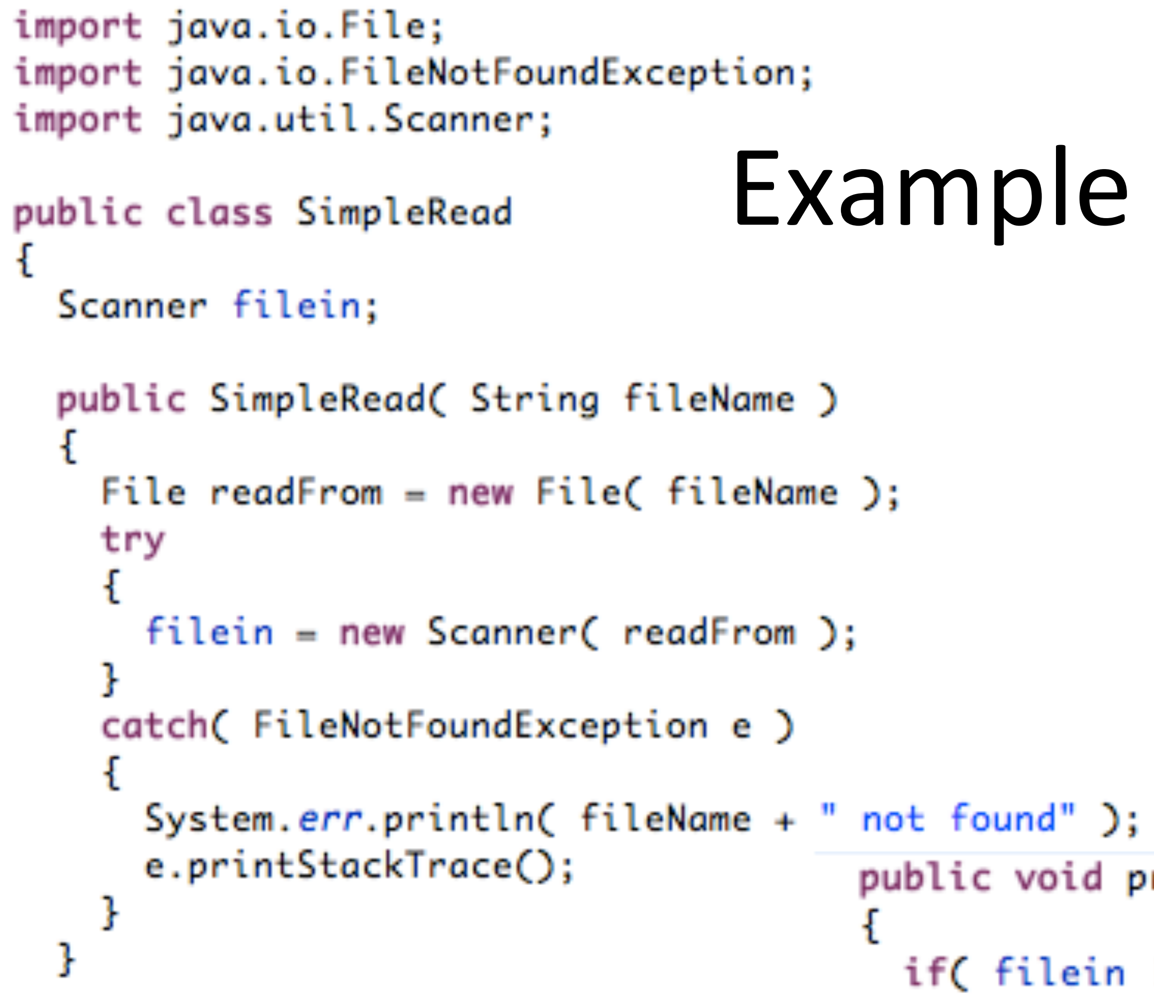


```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class SimpleRead
{
  Scanner filein;

  public SimpleRead( String fileName )
  {
    File readFrom = new File( fileName );
    try
    {
      filein = new Scanner( readFrom );
    }
    catch( FileNotFoundException e )
    {
      System.err.println( fileName + " not found" );
      e.printStackTrace();
    }
  }
```



```
  public void processByLine()
  {
    if( filein != null )
    {
      while( filein.hasNextLine() )
      {
        System.out.println( filein.nextInt() );
        filein.nextLine();  // read rest of line
      }
    }
  }
}
```

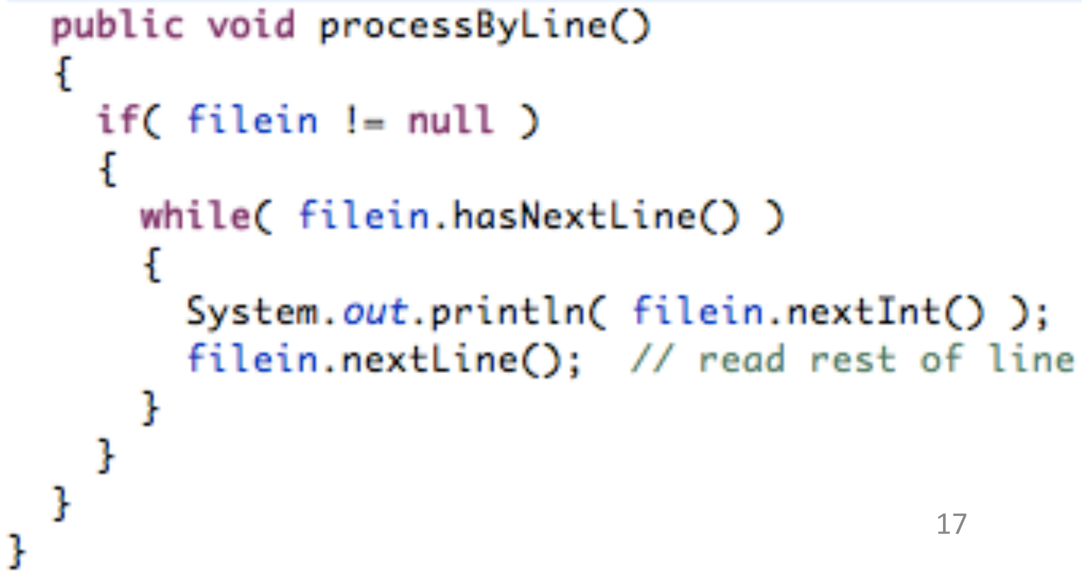
Example



```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class SimpleRead
{
  Scanner filein;

  public SimpleRead( String fileName )
  {
    File readFrom = new File( fileName );
    try
    {
      filein = new Scanner( readFrom );
    }
    catch( FileNotFoundException e )
    {
      System.err.println( fileName + " not found" );
      e.printStackTrace();
    }
  }
```

1. create File object

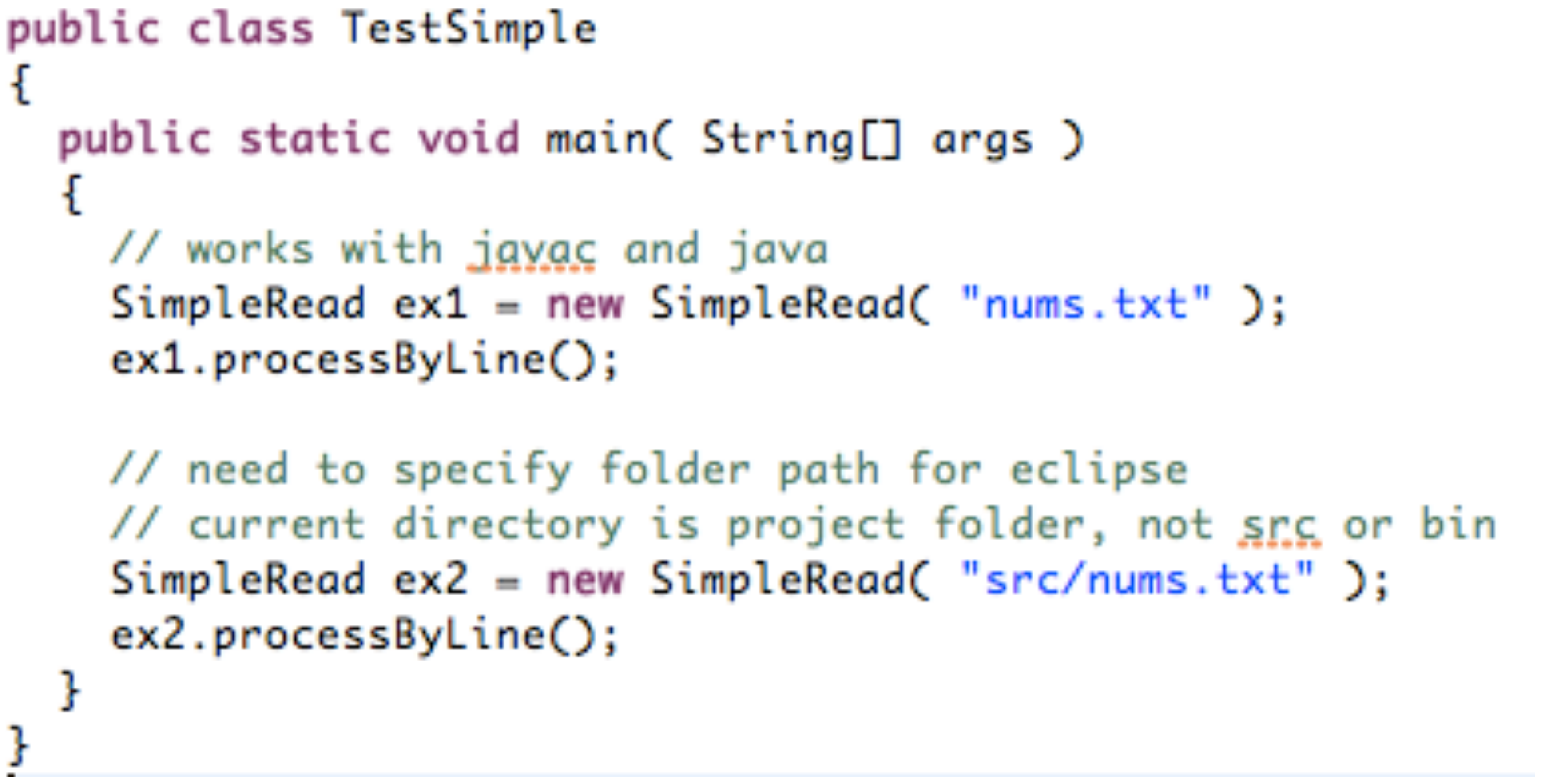
2. create Scanner object



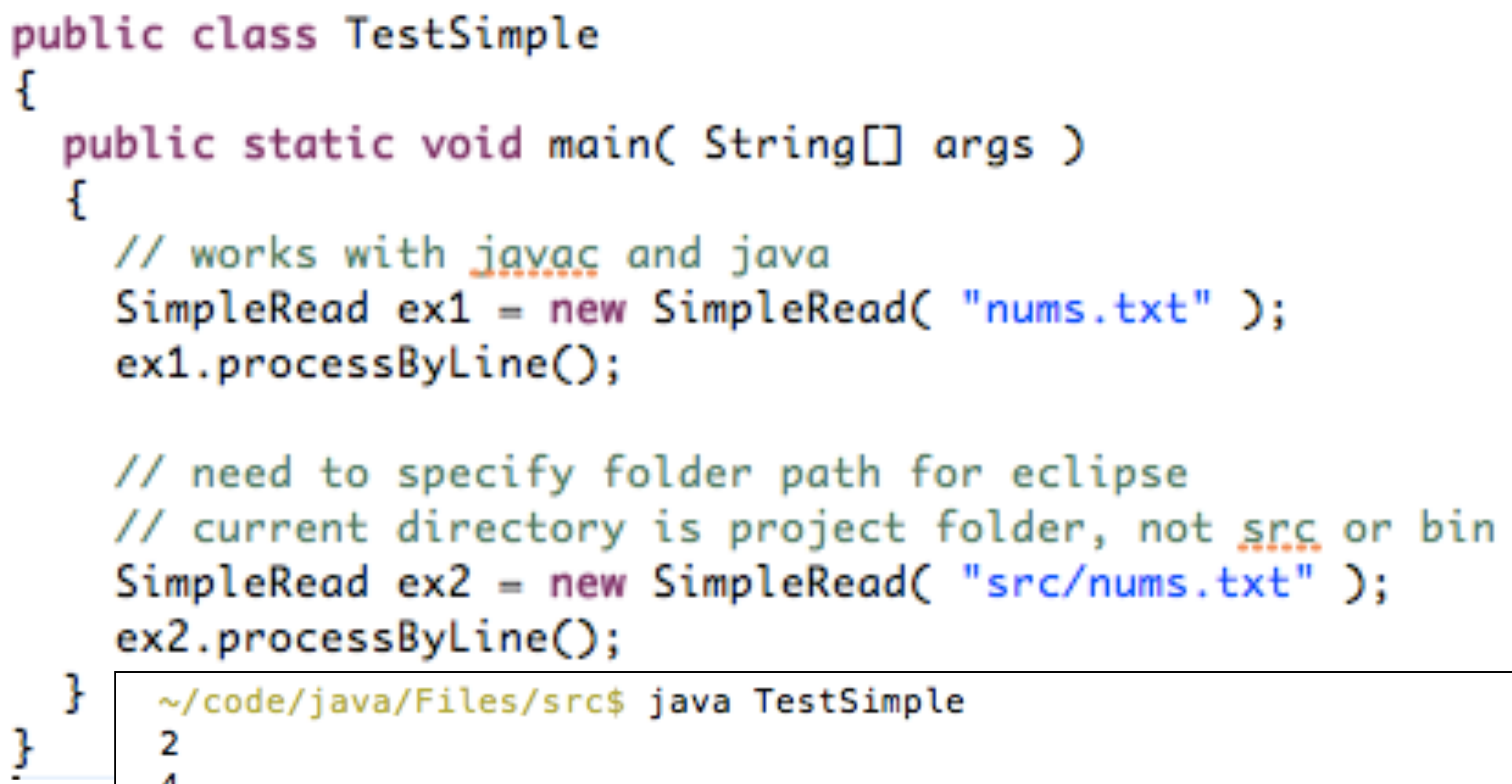
```java
  public void processByLine()
  {
    if( filein != null )
    {
      while( filein.hasNextLine() )
      {
        System.out.println( filein.nextInt() );
        filein.nextLine();  // read rest of line
      }
    }
  }
}
```

3. use Scanner object
just like before

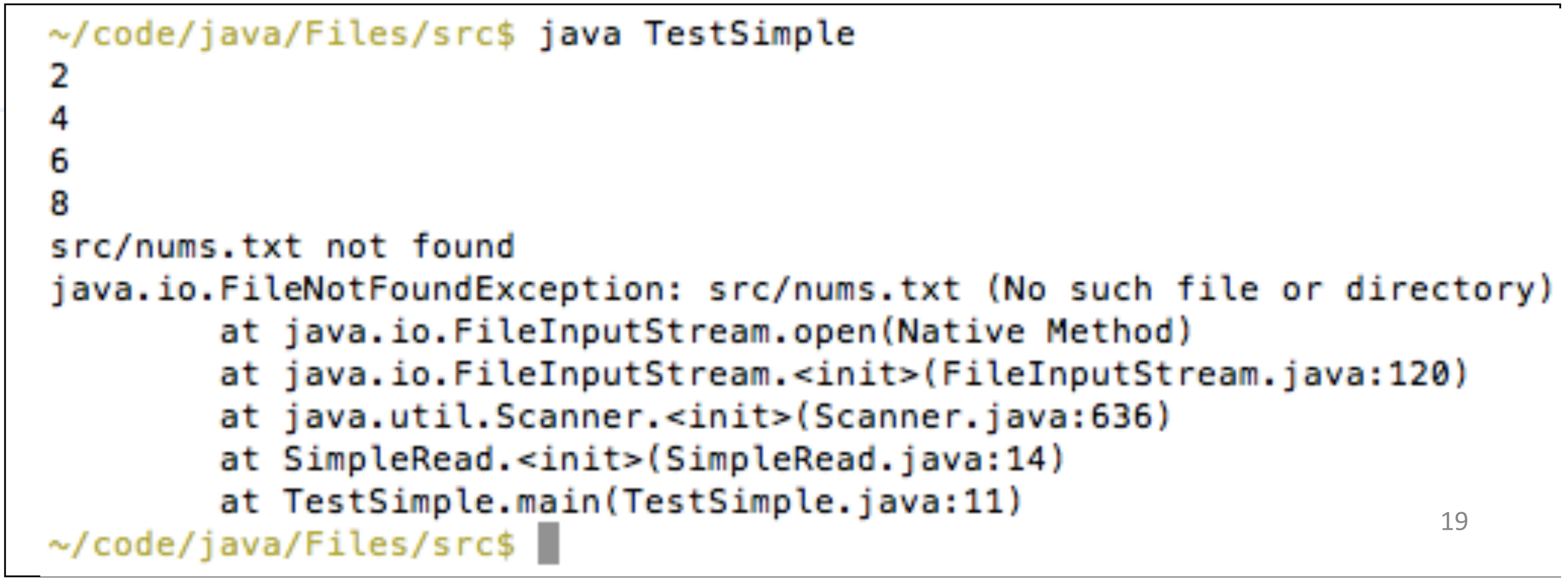
Specifying the File Name and Path



```java
public class TestSimple
{
  public static void main( String[] args )
  {
    // works with javac and java
    SimpleRead ex1 = new SimpleRead( "nums.txt" );
    ex1.processByLine();

    // need to specify folder path for eclipse
    // current directory is project folder, not src or bin
    SimpleRead ex2 = new SimpleRead( "src/nums.txt" );
    ex2.processByLine();
  }
}
```

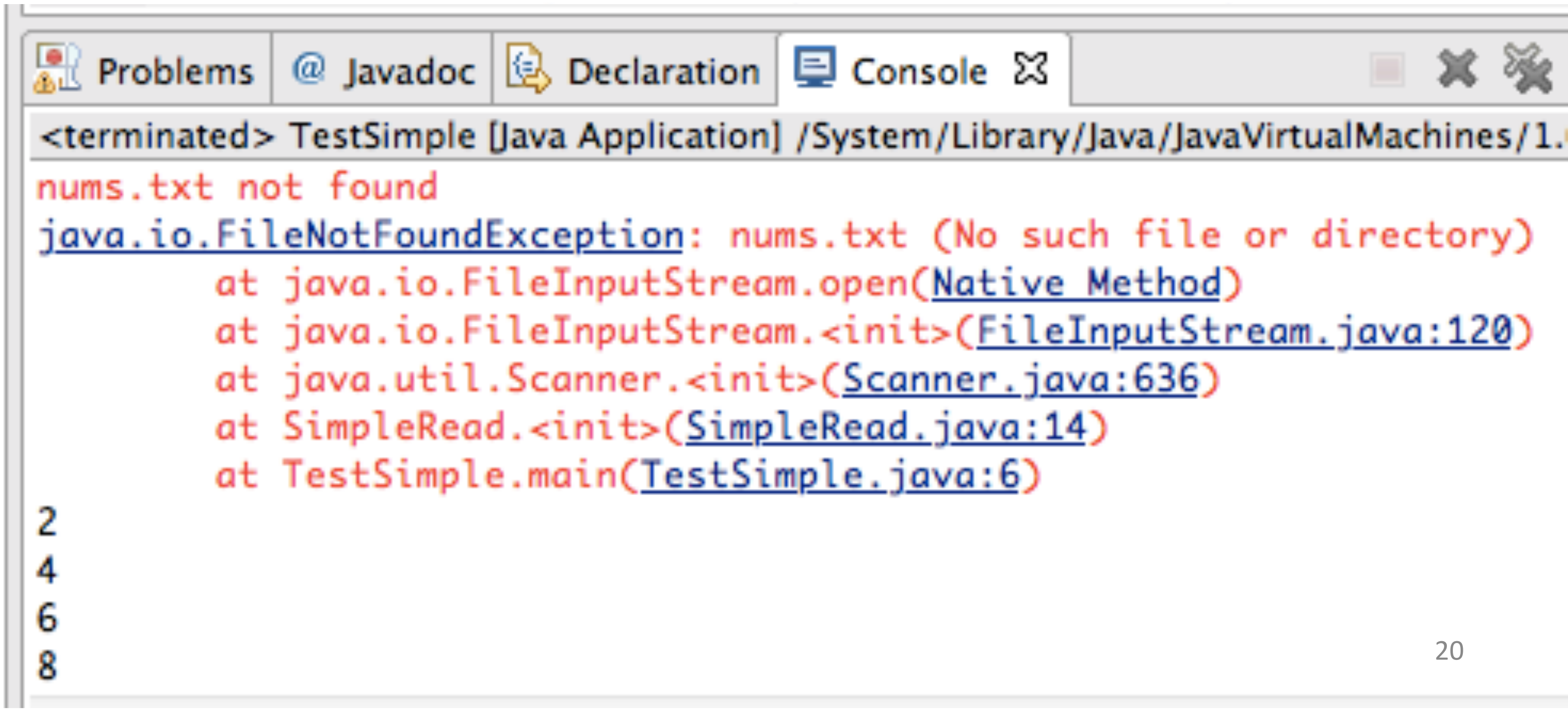


```java
public class TestSimple
{
  public static void main( String[] args )
  {
    // works with javac and java
    SimpleRead ex1 = new SimpleRead( "nums.txt" );
    ex1.processByLine();

    // need to specify folder path for eclipse
    // current directory is project folder, not src or bin
    SimpleRead ex2 = new SimpleRead( "src/nums.txt" );
    ex2.processByLine();
  }
}
```



```
~/code/java/Files/src$ java TestSimple
2
4
6
8
src/nums.txt not found
java.io.FileNotFoundException: src/nums.txt (No such file or directory)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:120)
        at java.util.Scanner.<init>(Scanner.java:636)
        at SimpleRead.<init>(SimpleRead.java:14)
        at TestSimple.main(TestSimple.java:11)
~/code/java/Files/src$ 
```

```java
public class TestSimple
{
  public static void main( String[] args )
  {
    // works with javac and java
    SimpleRead ex1 = new SimpleRead( "nums.txt" );
    ex1.processByLine();

    // need to specify folder path for eclipse
    // current directory is project folder, not src or bin
    Si
    e
  }
}
```



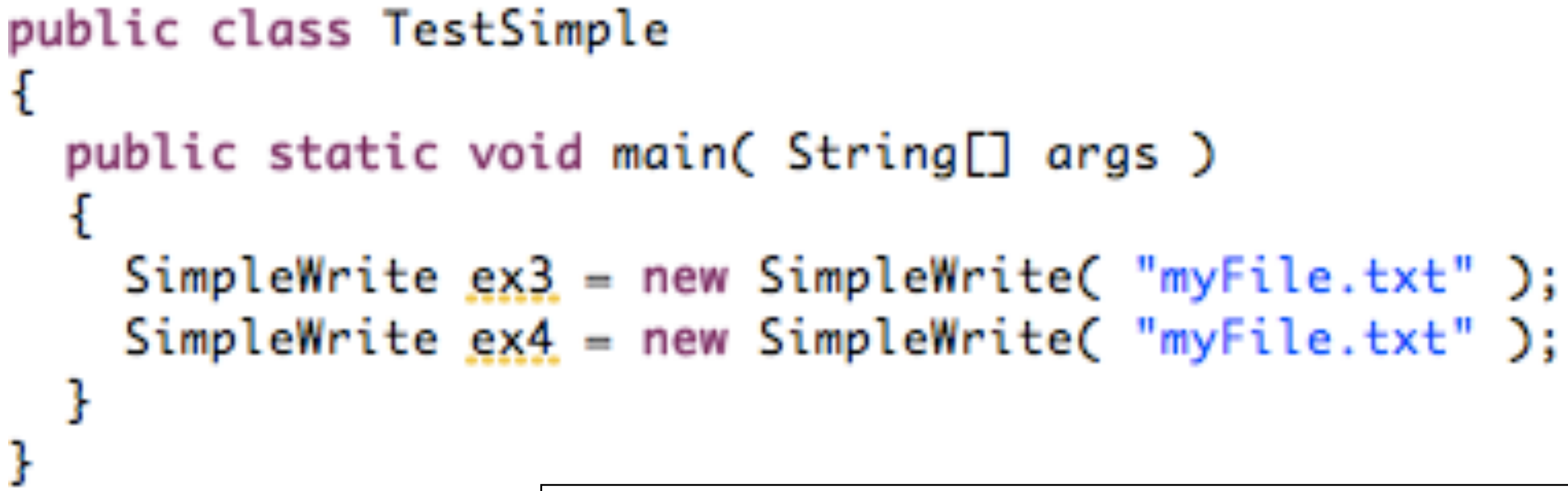
Problems | Javadoc | Declaration | Console

```
<terminated> TestSimple [Java Application] /System/Library/Java/JavaVirtualMachines/1.
nums.txt not found
java.io.FileNotFoundException: nums.txt (No such file or directory)
	at java.io.FileInputStream.open(Native Method)
	at java.io.FileInputStream.<init>(FileInputStream.java:120)
	at java.util.Scanner.<init>(Scanner.java:636)
	at SimpleRead.<init>(SimpleRead.java:14)
	at TestSimple.main(TestSimple.java:6)
2
4
6
8
```

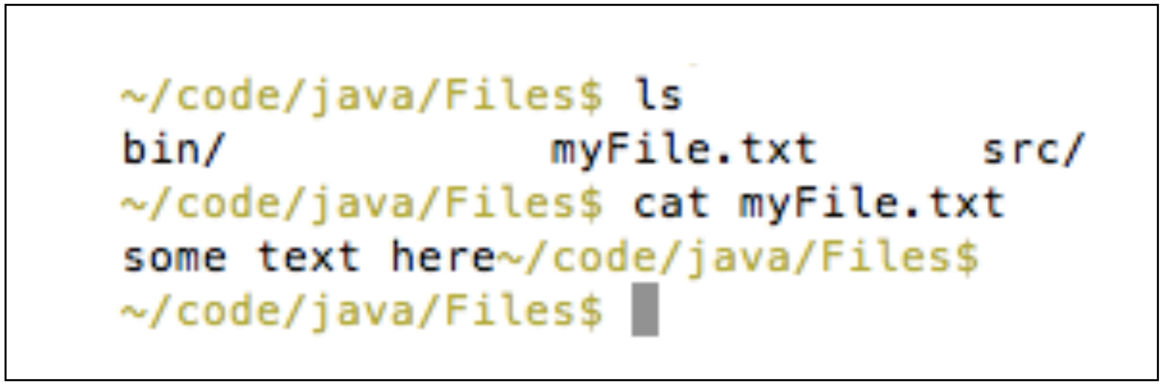
Writing to Text File

- Use `FileWriter` class
- Create new `FileWriter` object with file name
  - E.g., `FileWriter outFile = new FileWriter( “out.txt” );`
  - This creates new output file stream
  - Needs to handle `IOException`
- Use methods such as:
  - `write()`
  - `close()` – must do otherwise data won’t save

```
import java.io.FileWriter;
import java.io.IOException;

public class SimpleWrite
{
  FileWriter fileout;

  public SimpleWrite( String fileName )
  {
    try
    {
      fileout = new FileWriter( fileName );
      fileout.write( "some text here" );
    }
    catch( IOException e )
    {
      e.printStackTrace();
    }
    finally
    {
      try
      {
        fileout.close();
      }
      catch( IOException e )
      {
        System.err.println( fileout + " could not be closed" );
        e.printStackTrace();
      }
    }
  }
}
```



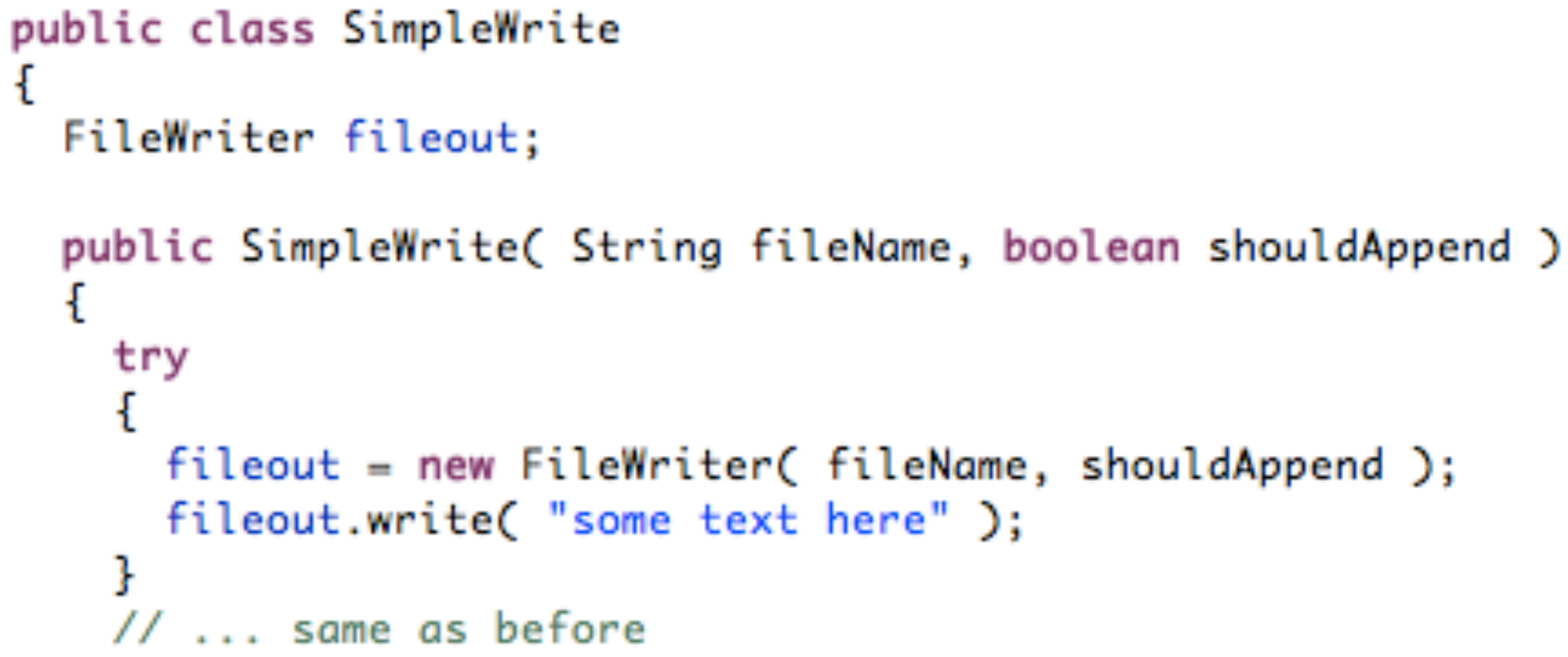
```java
public class TestSimple
{
  public static void main( String[] args )
  {
    SimpleWrite ex3 = new SimpleWrite( "myFile.txt" );
    SimpleWrite ex4 = new SimpleWrite( "myFile.txt" );
  }
}
```



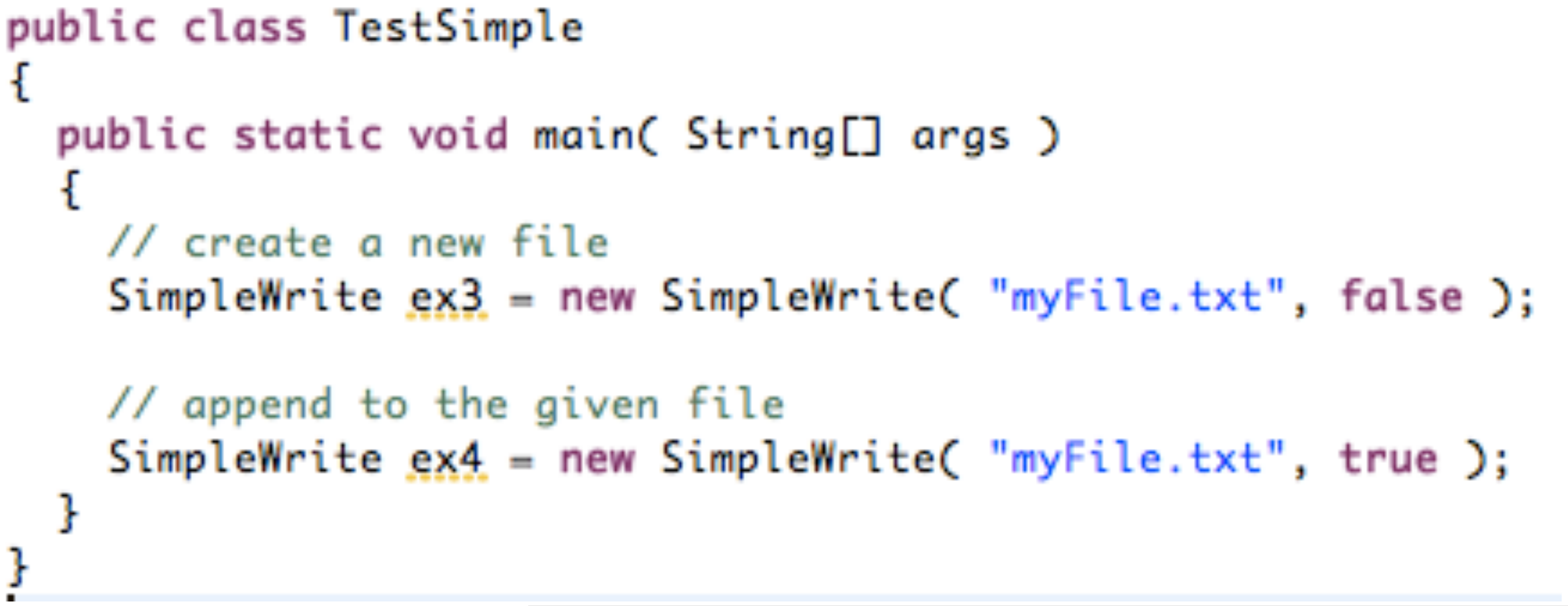
```
~/code/java/Files$ ls
bin/                myFile.txt      src/
~/code/java/Files$ cat myFile.txt
some text here~/code/java/Files$
~/code/java/Files$ 
```

Appending to File

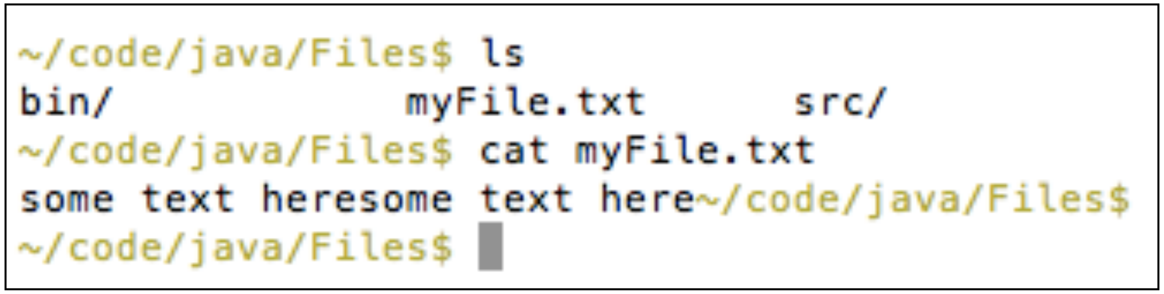
- Previously, new `FileWriter` object is created
  - E.g., `FileWriter outFile = new FileWriter( “out.txt” );`
- Even if “out.txt” exists, it overwrites content
- Appending means to add to end of existing file
- Change constructor statement with boolean flag:
  - E.g., `FileWriter outFile = new FileWriter( “out.txt”, true );`



```
public class SimpleWrite
{
  FileWriter fileout;

  public SimpleWrite( String fileName, boolean shouldAppend )
  {
    try
    {
      fileout = new FileWriter( fileName, shouldAppend );
      fileout.write( "some text here" );
    }
    // ... same as before
```



```
public class TestSimple
{
  public static void main( String[] args )
  {
    // create a new file
    SimpleWrite ex3 = new SimpleWrite( "myFile.txt", false );

    // append to the given file
    SimpleWrite ex4 = new SimpleWrite( "myFile.txt", true );
  }
}
```



```
~/code/java/Files$ ls
bin/               myFile.txt      src/
~/code/java/Files$ cat myFile.txt
some text heresome text here~/code/java/Files$
~/code/java/Files$
```

Alternative Writer Class

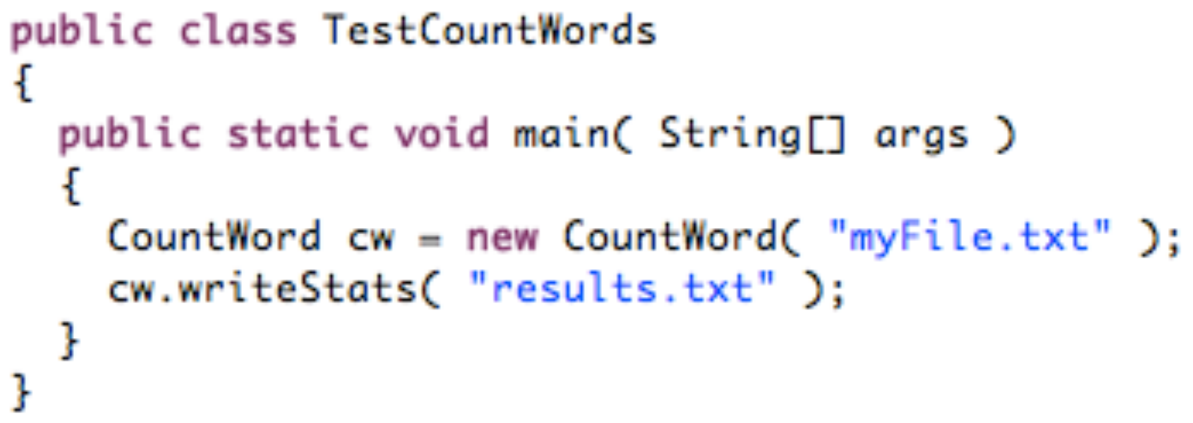
- Use `PrintWriter` class
- Create new `PrintWriter` object with file name
  - E.g., `PrintWriter outFile = new PrintWriter( "out.txt" );`
  - This creates new output file stream
- Also extension of `Writer` class, just like `FileWriter`
- Has different methods such as:
  - `write()`
  - `print()`
  - `println()`
  - `close()` – must do otherwise data won't save

Importing Classes

- Scanner
  - `import java.util.Scanner;`
- File
  - `import java.io.File;`
- FileWriter
  - `import java.io.FileWriter;`
- PrintWriter
  - `import java.io.PrintWriter;`
- Exceptions
  - `import java.io.IOException;`
  - `import java.io.FileNotFoundException;`
- Multiple libraries from the same path:
  - `import .java.io.*;`

Exercise

- Write a class called CountWord that has
  - A constructor
  - `private int countNumWords()` to open a file and return the number of words in that file
  - A `writeStats()` method that counts the number of words in a given file and adds the count to end of that file
  - Used in test class as follows:



```
public class TestCountWords
{
  public static void main( String[] args )
  {
    CountWord cw = new CountWord( "myFile.txt" );
    cw.writeStats( "results.txt" );
  }
}
```

Summary of File I/O

- Checked exceptions
  - Describe bad situation outside of program
  - Must be handled
- Unchecked exceptions
  - Describe bad situation when running the program usually logic error
  - Does not have to be explicitly handled

- File I/O involve checked exceptions
- New classes:
  - `File` (used with `Scanner`)
  - `FileWriter`, `PrintWriter`