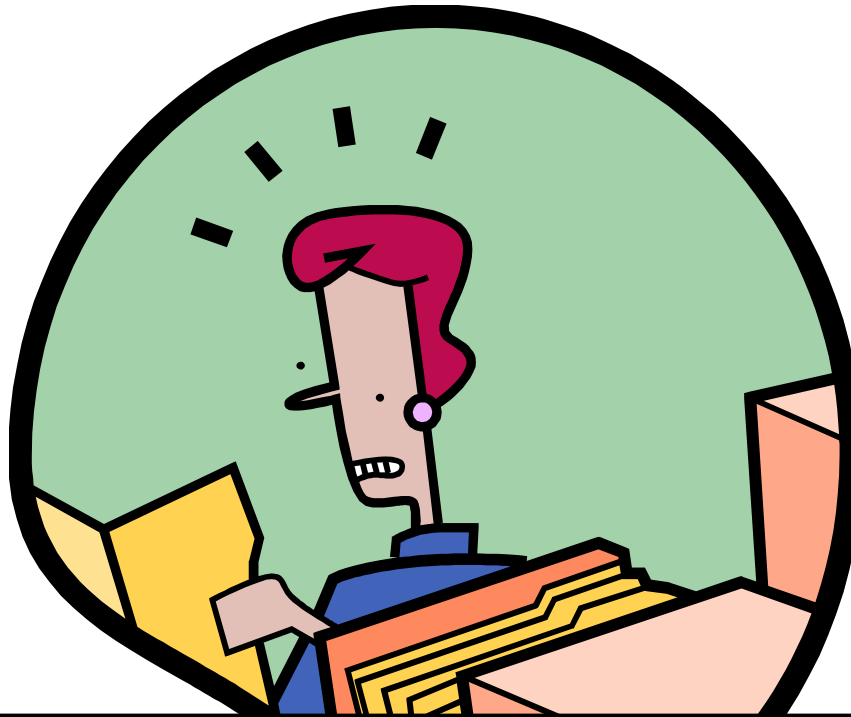


# Exceptions and file I/O



```
Exception in thread "main" java.lang.NumberFormatException: For input string: "3.5"  
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)  
    at java.lang.Integer.parseInt(Integer.java:458)  
    at java.lang.Integer.parseInt(Integer.java:499)  
    at AddNums.main(AddNums.java:8)
```

# Overview

- **Exceptions**
  - Handling **unexpected events**
    - e.g. File is missing
    - e.g. Trying to parse "\$56.89" as a double
- **File input/output**
  - **Output** to a text file
  - **Input** from a text file
    - Parsing the input using a Scanner
  - Frees us from the limits of standard input
    - Read from multiple files, read data twice, etc.

# Adding two numbers

- **Goal: Defend against all types of bad input**
  - Problem 1: Crashes if less than 2 arguments

```
public class AddNums
{
    public static void main(String [] args)
    {
        int num1 = Integer.parseInt(args[0]);
        int num2 = Integer.parseInt(args[1]);
        int sum = num1 + num2;
        System.out.println(num1 + " + " + num2 + " = " + sum);
    }
}
```

```
% java AddNums
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
    at AddNums.main(AddNums.java:5)
```

# Adding two numbers

- **Goal: Defend against all types of bad input**
  - Fix 1: Add conditional to check there are 2 args
  - Problem 2: Crashes if passed a non-integer arg

```
public class AddNums
{
    public static void main(String [] args)
    {
        if (args.length < 2)
        {
            System.out.println("AddNums <integer 1> <integer 2>");
            return;
        }
        int num1 = Integer.parseInt(args[0]);
        int num2 = Integer.parseInt(args[1]);
        int sum = num1 + num2;
        System.out.println(num1 + " + " + num2 + " = " + sum);
    }
}
```

```
% java AddNums 2 3.5
```

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "3.5"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:458)
    at java.lang.Integer.parseInt(Integer.java:499)
    at AddNums.main(AddNums.java:8)
```

# Adding two numbers

- How to check for invalid input to `parseInt`?
  - e.g. 1.0, 192.168.1.4, \$1, 123., one

```
public class AddNums
{
    public static void main(String [] args)
    {
        if (args.length < 2)
        {
            System.out.println("AddNums <integer 1> <integer 2>");
            return;
        }
        int num1 = Integer.parseInt(args[0]);
        int num2 = Integer.parseInt(args[1]);
        int sum = num1 + num2;
        System.out.println(num1 + " + " + num2 + " = " + sum);
    }
}
```

# Java exceptions

- When things go wrong:
  - Java throws an exception
  - An exception is an object that can be caught
  - You get to decide if program can recover or not
    - Rather than always crashing with a runtime error

```
% java AddNums 2 3.5
Exception in thread "main" java.lang.NumberFormatException: For input string: "3.5"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:458)
    at java.lang.Integer.parseInt(Integer.java:499)
    at AddNums.main(AddNums.java:8)
```

# try-catch block

```
try
{
    // Do some risky things

    // Do some more risky things
}
catch (Exception e)
{
    // Try and recover from the problem
}
```

If **something goes horribly wrong** on a line in the try block, flow of control **immediately jumps to the catch block**.

Details about the exception are passed as an object into the catch block.

Like a method's parameter list, e is just a name for the Exception object in the catch block.

The mother of all exception types.

All other types of exceptions inherit from this parent class.

# Add code to catch all exceptions

```
public class AddNums
{
    public static void main(String [] args)
    {
        try
        {
            int num1 = Integer.parseInt(args[0]);
            int num2 = Integer.parseInt(args[1]);
            int sum = num1 + num2;
            System.out.println(num1 + " + " + num2 + " = " + sum);
        }
        catch (Exception e)
        {
            System.out.println("Something went wrong!");
        }
        System.out.println("End of program");
    }
}
```

```
% java AddNums 2 3.5
Something went wrong!
End of program
```

```
% java AddNums
Something went wrong!
End of program
```

Not an ideal solution:

*How is the user suppose to know what to do differently?*



# A better solution

```
public static void main(String [] args)
{
    if (args.length < 2)
    {
        System.out.println("AddNums <integer 1> <integer 2>");
        return;
    }

    int num1, num2;
    try
    {
        num1 = Integer.parseInt(args[0]);
    }
    catch (NumberFormatException e)
    {
        System.out.println("1st argument invalid: " + args[0]);
        return;
    }

    try
    {
        num2 = Integer.parseInt(args[1]);
    }
    catch (NumberFormatException e)
    {
        System.out.println("2nd argument invalid: " + args[1]);
        return;
    }

    int sum = num1 + num2;
    System.out.println(num1 + " + " + num2 + " = " + sum);
}
```

## Principle 1:

Don't catch exceptions you can handle with logic such as staying in bounds of an array.

## {} scoping rules apply inside a try-block:

You'll need to declare variables outside if you need them in catch-block or later.

## Principle 2:

Don't use generic Exception class. Catch the specific type of exception you had in mind.

# Writing to a text file

- Java has many built in file I/O classes
  - **PrintWriter**, class that allows writing text to a file

Constructor Summary	
<a href="#">PrintWriter</a> ( <a href="#">File</a> file)	Creates a new <code>PrintWriter</code> , without automatic line flushing, with the specified file.
<a href="#">PrintWriter</a> ( <a href="#">File</a> file, <a href="#">String</a> csn)	Creates a new <code>PrintWriter</code> , without automatic line flushing, with the specified file and char
<a href="#">PrintWriter</a> ( <a href="#">OutputStream</a> out)	Creates a new <code>PrintWriter</code> , without automatic line flushing, from an existing <code>OutputStream</code>
<a href="#">PrintWriter</a> ( <a href="#">OutputStream</a> out, boolean autoFlush)	Creates a new <code>PrintWriter</code> from an existing <code>OutputStream</code> .
<a href="#">PrintWriter</a> ( <a href="#">String</a> fileName)	Creates a new <code>PrintWriter</code> , without automatic line flushing, with the specified file name.
<a href="#">PrintWriter</a> ( <a href="#">String</a> fileName, <a href="#">String</a> csn)	Creates a new <code>PrintWriter</code> , without automatic line flushing, with the specified file name an
<a href="#">PrintWriter</a> ( <a href="#">Writer</a> out)	Creates a new <code>PrintWriter</code> , without automatic line flushing.
<a href="#">PrintWriter</a> ( <a href="#">Writer</a> out, boolean autoFlush)	Creates a new <code>PrintWriter</code> .

Method Summary	
<a href="#">PrintWriter</a> <a href="#">append</a> (char c)	Appends the specified character to this writer.
<a href="#">PrintWriter</a> <a href="#">append</a> ( <a href="#">CharSequence</a> csq)	Appends the specified character sequence to this writer.
<a href="#">PrintWriter</a> <a href="#">append</a> ( <a href="#">CharSequence</a> csq, int start, int end)	Appends a subsequence of the specified character sequence to this writer

void	<a href="#">print</a> (double d) Prints a double-precision floating-point number.
void	<a href="#">print</a> (float f) Prints a floating-point number.
void	<a href="#">print</a> (int i) Prints an integer.
void	<a href="#">print</a> (long l) Prints a long integer.
void	<a href="#">print</a> ( <a href="#">Object</a> obj) Prints an object.
void	<a href="#">print</a> ( <a href="#">String</a> s) Prints a string.
<a href="#">PrintWriter</a>	<a href="#">printf</a> ( <a href="#">Locale</a> l, <a href="#">String</a> format, <a href="#">Object</a> ... args) A convenience method to write a formatted string to this writer using the specified for
<a href="#">PrintWriter</a>	<a href="#">printf</a> ( <a href="#">String</a> format, <a href="#">Object</a> ... args) A convenience method to write a formatted string to this writer using the specified for
void	<a href="#">println</a> () Terminates the current line by writing the line separator string.
void	<a href="#">println</a> (boolean x) Prints a boolean value and then terminates the line.
void	<a href="#">println</a> (char x) Prints a character and then terminates the line.
void	<a href="#">println</a> (char[] x) Prints an array of characters and then terminates the line.
void	<a href="#">println</a> (double x) Prints a double-precision floating-point number and then terminates the line.

# Perfect square writer

- Goal: Write perfect squares  $0^2 - 999^2$  to a file

```
import java.io.*;

public class PerfectSquareWriter
{
    public static void main(String [] args)
    {
        PrintWriter writer = new PrintWriter("squares.txt");
        for (int i = 0; i < 1000; i++)
            writer.println(i * i);
        writer.close();
    }
}
```

So Java can find the  
PrintWriter class.

PrintWriter constructor  
takes a string specifying the  
filename to write to.

```
% javac PerfectSquareWriter.java
PerfectSquareWriter.java:8: unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown
    PrintWriter writer = new PrintWriter("squares.txt");
                          ^
1 error
```

*Program fails to compile...*

# Perfect square writer

- Lesson: Some risky behaviors require try-catch

```
import java.io.*;

public class PerfectSquareWriter
{
    public static void main(String [] args)
    {
        try
        {
            PrintWriter writer = new PrintWriter("squares.txt");
            for (int i = 0; i < 1000; i++)
                writer.println(i * i);
            writer.close();
        }
        catch (FileNotFoundException e)
        {
            System.out.println("Failed to open file!");
        }
    }
}
```

This line had to be in a try-catch block catching a `FileNotFoundException` (or a parent thereof).

Not required, but good style to cleanup after you are done with a file.

```
% more squares.txt
0
1
4
9
16
...
```

# Reading a text file

- Need two Java classes:
  - **File class**, represents a filename
    - System independent abstraction of a file's path
    - Not actually used for reading or writing
  - **Scanner class**, parses out values
    - Very similar to `StdIn.read*` methods

# Handy methods: File class

Method	Description
<code>boolean canRead()</code>	Test if the program can read from the file.
<code>boolean canWrite()</code>	Test if the program can write to the file.
<code>boolean delete()</code>	Attempt to delete the file.
<code>boolean exists()</code>	See if the given file exists.
<code>long length()</code>	Get length of the file in bytes
<code>String getName()</code>	Returns the filename portion of the path
<code>String getPath()</code>	Returns the path without the filename
<code>boolean mkdir()</code>	Creates a directory specified by the path

```
File file = new File("c:\\workspace\\nums.txt");  
System.out.println(file.getName());  
System.out.println(file.getPath());
```

Windows paths need **escaping of the backslash** by using two of them.

```
nums.txt  
c:\workspace\nums.txt
```

# Handy methods: Scanner class

Method	Description
<code>String next()</code>	Returns the next string, separated via whitespace
<code>String nextLine()</code>	Returns the entire line up to the next line break
<code>int nextInt()</code>	Returns the next integer
<code>double nextDouble()</code>	Returns the next double
<code>boolean hasNext()</code>	Are there any more tokens available?
<code>boolean hasNextLine()</code>	Is there another line available?
<code>boolean hasNextInt()</code>	Can the next token be interpreted as an int?
<code>boolean hasNextDouble()</code>	Can the next token be interpreted as a double?
<code>void close()</code>	Free up resources

# Averaging numbers

```
public class AvgNums
{
    public static void main(String [] args)
    {
        double sum = 0.0;
        long count = 0;
        while (!StdIn.isEmpty())
        {
            sum += StdIn.readDouble();
            count++;
        }
        System.out.println(sum / count);
    }
}
```

Original program:  
reads from standard input.

```
% java AvgNums < nums.txt
8.614076923076924
```

```
public class AvgNumsFile
{
    public static void main(String [] args)
    {
        double sum = 0.0;
        long count = 0;
        if (args.length < 1)
        {
            System.out.println("AvgNumsFile <filename>");
            return;
        }
        try
        {
            Scanner scanner = new Scanner(new File(args[0]));
            while (scanner.hasNext())
            {
                sum += scanner.nextDouble();
                count++;
            }
            scanner.close();
            System.out.println(sum / count);
        }
        catch (FileNotFoundException e)
        {
            System.out.println("Failed to open file!");
        }
    }
}
```

New program: reads from filename given by args[0].

```
% java AvgNumsFile nums.txt
8.614076923076924
```



# Trying to break it

```
public class AvgNumsFile
{
    public static void main(String [] args)
    {
        double sum = 0.0;
        long count = 0;
        if (args.length < 1)
        {
            System.out.println("AvgNumsFile: No file specified");
            return;
        }
        try
        {
            Scanner scanner = new Scanner(new File(args[0]));
            while (scanner.hasNext())
            {
                sum += scanner.nextDouble();
                count++;
            }
            scanner.close();
            System.out.println(sum / count);
        }
        catch (FileNotFoundException e)
        {
            System.out.println("Failed to open file!");
        }
    }
}
```

```
% java AvgNumsFile noexist.txt
Failed to open file!
```

```
% java AvgNumsFile mobydick.txt
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:840)
    at java.util.Scanner.next(Scanner.java:1461)
    at java.util.Scanner.nextDouble(Scanner.java:2387)
    at AvgNumsFile.main(AvgNumsFile.java:21)
```

# Multiple catch blocks

```
...
double sum = 0.0;
long count = 0;
if (args.length < 1)
{
    System.out.println("AvgNumsFile <filename>");
    return;
}
try
{
    Scanner scanner = new Scanner(new File(args[0]));
    while (scanner.hasNext())
    {
        sum += scanner.nextDouble();
        count++;
    }
    scanner.close();
    System.out.println(sum / count);
}
catch (FileNotFoundException e)
{
    System.out.println("Failed to open file!");
}
catch (InputMismatchException e)
{
    System.out.println("Invalid data in file!");
}
...
```

```
% java AvgNumsFile mobydick.txt
Invalid data in file!
```

# Throwing exceptions

- How do exceptions start their life?

- Somebody throws them:

```
throw new NoCaffeineException();
```

- Whoever is using the method has to catch
- Or else that method can throw it
- What if nobody catches it?

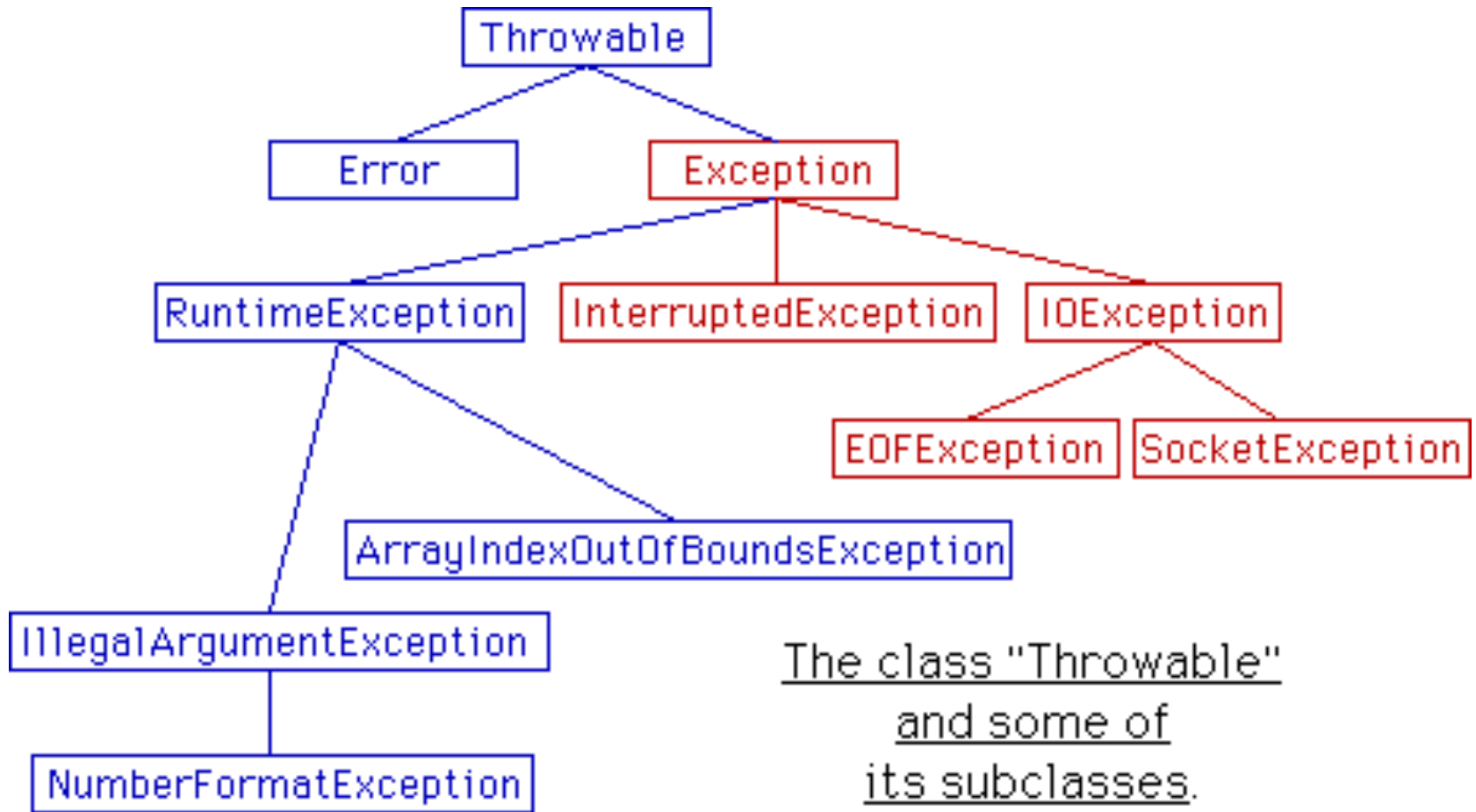
- Checked exceptions

- Causes compile error if exception is not caught

- Unchecked exceptions

- Will compile without a catch, catching is optional
- Usually a programming error
- e.g. `ArrayIndexOutOfBoundsException`, `NullPointerException`

# Java exception class hierarchy



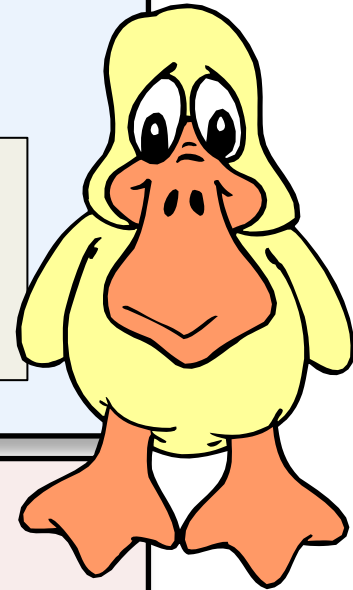
<http://www.faqs.org/docs/javap/c9/s3.html>

# Ducking an exception

```
public static double getFileAvg(String filename) throws FileNotFoundException
{
    double sum = 0.0;
    long count = 0;

    Scanner scanner = new Scanner(new File(filename));
    while (scanner.hasNext())
    {
        sum += scanner.nextDouble();
        count++;
    }
    scanner.close();
    return (sum / count);
}
```

Not our problem anymore, whoever calls getFileAvg() has to catch it now.  
Methods can throw 0 or more exception types.



```
public static void main(String [] args)
{
    if (args.length < 1)
    {
        System.out.println("AvgNumsFile <filename>");
        return;
    }
    System.out.println(getFileAvg(args[0]));
}
```

```
% javac AvgNumsFile.java
AvgNumsFile.java:26: unreported exception java.io.FileNotFoundException; must be
    caught or declared to be thrown
    double avg = getFileAvg(args[0]);
                   ^
1 error
```

# Finally, the finally block

- **Finally block executes no matter what**
  - If no exception, runs after try-block
  - If exception occurs, runs after catch-block
  - Useful for doing cleanup that is always needed

```
try
{
    turnOvenOn();
    x.bake();
}
catch (BakingException e)
{
    e.printStackTrace();
}
finally
{
    turnOvenOff();
}
```

Prints out the stack trace  
(like what you see when  
you get a runtime error).

Handy for debugging what  
line in the try-block is  
causing the trouble.

# Summary

- **Exceptions**
  - An important part of writing defensive code
  - **Many Java classes require exception handling**
    - e.g. File I/O, network communication, playing sounds, using threads
  - **Checked** versus **unchecked** exceptions
- **File I/O**
  - Text file input/output
  - **PrintWriter** class, writing text to file
  - **Scanner** and **File** classes, reading and parsing text