

## COSC 123 Computer Creativity

### I/O Streams and Exceptions

**Dr. Ramon Lawrence**  
University of British Columbia Okanagan  
ramon.lawrence@ubc.ca

COSC 123 - Dr. Ramon Lawrence

### Objectives

Explain the purpose of exceptions.

Examine the try-catch-finally statement for handling exceptions.

Show how to throw exceptions to other methods.

Identify I/O streams with specific focus on reading and writing text files and handling I/O exceptions.

Page 2



COSC 123 - Dr. Ramon Lawrence

### Exception Handling

An **exception** is an error situation that must be handled or the program will fail.

◆ **Exception handling** is a mechanism for communicating error conditions between methods of your program.

Examples:

- ◆ Attempting to divide by zero
- ◆ An array index that is out of bounds
- ◆ A specified file that could not be found
- ◆ A requested I/O operation that could not be completed normally
- ◆ Attempting to follow a null reference
- ◆ Attempting to execute an operation that violates some kind of security measure

Page 3

COSC 123 - Dr. Ramon Lawrence

### Uncaught Exceptions

If a program does not handle the exception, it will terminate abnormally and produce the message that describes the exception that occurred and where in the code it was produced.

Example: Exception in thread "main" java.lang.NullPointerException  
at Asteroids.printScores(Asteroids.java:86)  
at Asteroids.addScore(Asteroids.java:78)  
at Asteroids.main(Asteroids.java:14)

- ◆ The output is the call stack trace that indicates where the exception occurred.

Page 4



COSC 123 - Dr. Ramon Lawrence

### The try-catch Statement

The **try-catch statement** identifies a block of statements that may throw an exception.

A **catch clause** defines how a particular kind of exception is handled. Each catch clause is called an **exception handler**.

When the try-catch statement is executed, the statements in the **try** block are executed.

If an exception is thrown at any point during the execution of the **try** block, control is immediately transferred to the appropriate **catch** handler.

Page 5

COSC 123 - Dr. Ramon Lawrence

### Catching Exceptions Example Code

```
try
{
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter your age? ");
    int age = sc.nextInt();
    System.out.println("You are: "+age+" years old!!!");
}
catch (InputMismatchException e)
{
    System.out.println("Input was not a number.");
}
```

Page 6

## The finally Clause

A try-catch statement can have an optional **finally** clause which defines a section of code that is executed *no matter* how the try block is exited.

If no exception is generated, the statements in the **finally** clause are executed after the **try** block is complete.

If an exception is generated in the **try** block, control first transfers to the appropriate **catch** clause, then to **finally** clause.

## Finally Example

```
try
{
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter your age?");
    int age = sc.nextInt();
    System.out.println("You are: "+age+" years old!!!");
}
catch (InputMismatchException e)
{
    System.out.println("Input was not a number.");
}
finally
{
    System.out.println("We always go in here!");
}
```

## Throwing Exceptions

Your method has two ways of handling exceptions:

- ◆1) It can handle them inside the method using a try-catch-finally block.
- ◆2) It can throw the exception to the method that called it and force that method to handle it.

To throw an exception you must do two things:

- ◆1) List the type of exception that is thrown in the method header.
- ◆2) Not catch an exception (do not use try-catch block) or create a new exception and call **throw** to pass it to the caller.

When an exception is thrown, the **method exits immediately** similar to a return statement.

## Throwing Exceptions Example Code

```
public class ThrowException
{
    public static void main(String[] args)
    {
        System.out.println("This isn't smart...");
        doSomethingDumb();
    }

    public static int doSomethingDumb()
        throws ArithmeticException
    {
        int num1 = 5, num2 = 0;
        int result = num1/num2; // Divide by zero
        return result;
    }
}
```

## Checked and Unchecked Exceptions

**Checked exceptions** are exceptions that you must tell the compiler how your code is handling them.

- ◆A checked exception *must* be either caught or thrown.
  - ⇒ Checked exceptions are typically exceptions that are not your fault.
  - ⇒ e.g. `IOException` (and all its subclasses)

**Unchecked exceptions** are exceptions that the compiler does not force your program to handle.

- ◆An unchecked exception is automatically passed to the caller method if it is not handled by the method that generated the exception.
  - ⇒ Unchecked exceptions include `NumberFormatException`, `IllegalArgumentException`, and `NullPointerException`.
  - ⇒ Exceptions that are a subclass of `RuntimeException` are unchecked.

## Exceptions

**Question:** TRUE or FALSE: A good programmer can always avoid exceptions.

- A) TRUE
- B) FALSE

## Exceptions

**Question:** TRUE or FALSE: An uncaught exception may be passed through several methods before the program crashes.

- A) TRUE
- B) FALSE

## Exceptions

**Question:** What does this code output if the user enters "32"?

```
try
{
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter a number: ");
    int num = sc.nextInt();
    System.out.print(num+" ");
}
catch (InputMismatchException e)
{
    System.out.print("Input was not a number. ");
}
finally
{
    System.out.print("HELLO!");
}
```

- A) nothing
- B) 32
- C) Input was not a number.
- D) 32 HELLO!

## Exceptions

**Question:** What does this code output if the user enters "abc"?

```
try
{
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter a number: ");
    int num = sc.nextInt();
    System.out.print(num+" ");
}
catch (InputMismatchException e)
{
    System.out.print("Input was not a number. ");
}
finally
{
    System.out.print("HELLO!");
}
```

- A) abc
- B) Input was not a number.
- C) abc HELLO!
- D) Input was not a number. HELLO!

## Java File Input/Output

A **stream** is an ordered sequence of bytes.

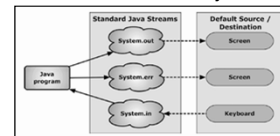
A stream may be either an input stream or an output stream.

An **input stream** is a stream from which information is read.

An **output stream** is a stream to which information is written.

Streams may generate exceptions such as `IOException`.

The `System` class contains three object reference variables:



## Reading and Writing Text Files

A file is opened as a stream for reading or writing.

Programmers need to know the contents of the file and how to translate it to a usable form.

If for some reason there is a problem finding or opening a file, the attempt to create a `File` object will throw an `IOException`.

- ◆ To put a backslash ("\") in a filename string, you must enter each backslash TWICE as backslash is an escape character.

⇒ e.g. `File in = new File("c:\\homework\\input.dat");`

Output file streams should be explicitly closed or they may not correctly retain the data written to them.

## ★ Read Text File with Scanner

```
Scanner sc = null;
try
{
    sc = new Scanner(new File("MyFile.txt"));
    while (sc.hasNextLine())
    {
        String st = sc.nextLine();
        System.out.println(st);
    }
}
catch (FileNotFoundException e)
{
    System.out.println("Did not find input file: "+e);
}
finally
{
    if (sc != null)
        sc.close();
}
```

Note: The `Scanner` class handles some exceptions for you.

## Write Text File with *PrintWriter*

```
PrintWriter out = null;
try
{
    out = new PrintWriter("output.txt");
    // Write the numbers 1 to 10 in the file
    for (int i=1; i <=10; i++)
        out.println(i);
}
catch (FileNotFoundException e)
{
    System.out.println("Could not create output file: "+e);
}
finally
{
    if (out != null)
        out.close();
}
```

## Streams and Exceptions Practice Question

1) Write a program that prompts the user for a filename then opens the text file and counts the number of lines in the file.

## Conclusions

An **exception** is an error situation that must be handled or the program will fail.

◆ **Exception handling** is a mechanism for communicating error conditions between methods of your program.

There are two ways for handling exceptions:

- ◆ 1) Instead method using a try-catch-finally block.
- ◆ 2) By throwing it to the caller method.
- ◆ Checked exceptions must always be handled.

A stream is a sequential sequence of bytes which can be used for input or output. Files are streams as is `System.out`.

Reading from text files can be done using `Scanner` class similar to reading from `System.in`.

Writing to text files is done using the `PrintWriter` class.

◆ Make sure to close all files!

## Objectives

Key terms:

- ◆ exceptions and exception handling

Java skills:

- ◆ exception handling using try-catch-finally statement
- ◆ uncaught exceptions and the call stack trace
- ◆ throwing exceptions (`throws` in method header)
- ◆ checked vs. unchecked exception
- ◆ streams and the standard I/O streams in the `System` class
- ◆ Reading from a text file using `Scanner`
- ◆ Writing to a text file using `PrintWriter`