

# CMSC 131: Chapter 22 (Supplement)

## Exceptions

### Exceptions

Handling run-time errors is an important part of programming.

**Arithmetic errors:** Divide by zero, ...

**Object/Array errors:** Using a null reference, illegal array index, ...

**File and I/O errors:** Nonexistent file, attempt to read past the end of the file, ...

**Application Specific:** Errors particular to your application. (E.g., attempt to remove a nonexistent customer from a database).

**Handling Errors:** When an error occurs what should happen?

- **Print message and abort?**
- **Handle the error** here, and fix things up? (May not be possible.)
- Return a **special error-flag** value and let calling program handle it?

### Exceptions

If you do nothing, an exception will cause your program to be aborted. Example:

```
public static void generateException() {
    int[] a = new int[20];
    System.out.println( "We got this far..." );
    a[32] = 5;        // We're askin' for trouble!
    System.out.println( "...but we never got here." );
}
```

Java aborts your program and automatically prints a **stack trace**.

```
We got this far...
java.lang.ArrayIndexOutOfBoundsException: 32
at Chapt22Snippets.generateException(Chapt22Snippets.java:13)
at Chapt22Snippets.main(Chapt22Snippets.java:7)
```

## Handling Exceptions

In some applications, aborting the program is not an option:

- Email processor:** Must restore file structure to a stable state.
- Commercial web/database server:** Must recover and carry on.
- Air traffic control system:** (Okay, everyone stop while we figure this out.)
- Guidance system on a cruise missile:** (Oops, my bad!)

Some Java terminology:

- Throw:** When an error is detected, an **exception is thrown**. It is possible for you to define and throw your own exceptions.
- Catch:** In order to avoid aborting, a program can catch an exception. This gives the program a chance to recover, and either resume or terminate gracefully.
- Try:** Executing some code that might throw an exception is called **trying**. For each block of tried code, there is an associated **catch**.

## Exception Types

There are different types of exceptions, depending on the error.

- ArithmeticException:** divide by zero.
- NullPointerException:** attempt to access an object with a null reference.
- IndexOutOfBoundsException:** array or string index out of range.
- ArrayStoreException:** attempting to store an object of type X in an array of type Y.
- EmptyStackException:** Attempt to pop an empty Stack (java.util).
- IOException:** Attempt to perform an illegal input/output operation (java.io)
- NumberFormatException:** Attempt to convert an invalid string into a number (e.g., when calling `Integer.parseInt( )`).
- ...
- Exception:** The most generic type of exception.

All of these are Objects in **Java's class library** (most in java.lang).

## Exception Objects

**Exception Object:** When an exception is thrown, a new **exception object is created**, which encodes information about the nature of the exception.

Every Exception object supports the following methods:

**Exception( String message ):** A constructor that is given an explanatory message string.

**String getMessage( ):** Returns the message.

**void printStackTrace( ):** Prints the contents of the Java call stack (a list of the methods that are executing) at the time of the exception.

You can define your own Exception objects, but we won't discuss this now.

## try-catch Blocks

To handle exceptions, Java provides **try-catch blocks**.

**Try block:** a block of code that might generate an exception.

**Catch block:** execution jumps here as soon as an exception is thrown.

**Syntax:**

```
try {
    ... ( this code might throw an exception ) ...
}
catch ( ExceptionType1 e1 ) {
    ... ( code to handle exceptions of type ExceptionType1 ) ...
}
catch ( ExceptionType2 e2 ) {
    ... ( code to handle exceptions of type ExceptionType2 ) ...
}
finally {
    ... ( this is executed no matter what ) ...
}
```

## Example: readDate1

Let us illustrate this with a little example.

**readDate1( )**: Inputs a date in the format "mm/dd/yyyy" and outputs the year as an integer. To do this, it calls **getDate( )**.

**getDate( )**: Given a date string "mm/dd/yyyy", extracts the "yyyy" as an int using the Java's built-in **substring( )** and **parseInt( )** methods.

**substring(b, e)**: Extracts a substring from position b through e-1. Generates an **IndexOutOfBoundsException** if b or e-1 is illegal.

**parseInt(s)**: converts string s to an int. Generates a **NumberFormatException** if s is not in valid integer format.

Our first example, **readDate1( )** reads a date. It catches **Exception**, which the most generic type of exception (and so catches both **IndexOutOfBoundsException** and **NumberFormatException**).

## Example: readDate1

```
public static int getYear( String d ) {
    String yearString = d.substring( 6, 10 );
    return Integer.parseInt( yearString );
}

public static void readDate1( ) {
    try {
        String d = JOptionPane.showInputDialog( "Enter date: (mm/dd/yyyy)" );
        int year = getYear( d );
        System.out.println( "The year is " + year );
    }
    catch ( Exception e ) {
        System.out.println( "An exception occurred:\n" + e.getMessage( ) );
    }
}
```

## Exception Propagation

**Exception Propagation:** An interesting thing to observe about the previous program is that the **exception took place in one method** (getDate) but was **processed in the calling method** (readDate1).

- When an exception occurs, Java pops **back up the call stack** to each of the calling methods to see whether the exception is being handled (by a try-catch block). This is **exception propagation**.
- The **first method** it finds that catches the exception will have its **catch block executed**. At this point the exception has been handled, and the **propagation stops** (no other catch blocks will be executed). **Execution resumes normally** after this catch block.
- If we get all the way back to main and **no method catches** this exception, Java catches it and **aborts** your program.

## Example: readDate2

Sometimes we want to take different action, based on the type of the exception. For example:

- **ArithmeticException:** ignore
- **IndexOutOfBoundsException:** print error message and exit
- **IOException:** ask user to enter a different file name

**readDate2( ):**

- Catches each of the two **specific types of exceptions** that might arise:
  - IndexOutOfBoundsException
  - NumberFormatException
- Adds a finally block, which is executed, no matter what.

## Example: readDate2

```
public static void readDate2() {
    String d = "";
    try {
        d = JOptionPane.showInputDialog( "Enter date: (mm/dd/yyyy)");
        int year = getYear( d );
        System.out.println( "The year is " + year );
    }
    catch ( IndexOutOfBoundsException e ) {
        System.out.println( "Index error" );
    }
    catch ( NumberFormatException e ) {
        System.out.println( "Number format exception" );
    }
    finally {
        System.out.println( "The original string: " + d );
    }
}
```