CSC 1051 Algorithms & Data Structures I

Exceptions

EXCEPTION HANDLING

Exceptions

An exception occurs when something unexpected happens while a program is **running**.

An exception is an object that represents such an unusual or erroneous situation, such as:

- Attempting to divide by zero
- Attempting to follow a null reference
- An array index that is out of bounds
- A specified file could not be found
- Lots more!

Exceptions

Catching an exception enables a program to respond gracefully. For example, when the following line of code is executed:

```
double result = 12345 / 0;
```

An ignored or uncaught exception causes a message like:

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
   at DivideByZero.main(DivideByZero.java:11)
```

Exceptions

Java uses exception handling to enable a program to catch exceptions and respond to them programmatically.

The Java API has a predefined a set of exceptions that can occur during execution, and a programmer can create more.

There are 3 ways a program can handle an exception:

1. Ignore it (error message!)
2. Handle it where it occurs (try-catch statement)
3. Handle it somewhere else in the program (exception propagation)

Exceptions

The try-catch statement is used in Java for exception handling.

```
try
{
    double result = 12345 / 0;
}
catch (Exception e)
{
    System.out.println("Something bad happened!");
}
```

The exception was handled **where it occurred**:

```
Something bad happened!
```

Exception propagation handles exceptions elsewhere.

Exceptions

The error message that results from an uncaught exception is called a stack trace. It identifies what happened and where.

```
Exception in thread "main"
java.lang.NullPointerException
    at PrereqMatrix.makePrereqMap(PrereqMatrix.java:97)
    at Scheduler.checkPrerequisites(Scheduler.java:248)
    at ScheduleManager.main(ScheduleManager.java:31)
```

A stack trace is read from the bottom up:

- On line 31 of ScheduleManager the main method calls
- checkPrerequisites in Scheduler which on line 248 calls
- makePrereqMap of PrereqMatrix where on line 97 it tries to follow a null reference causing a NullPointerException

The try-catch Statement

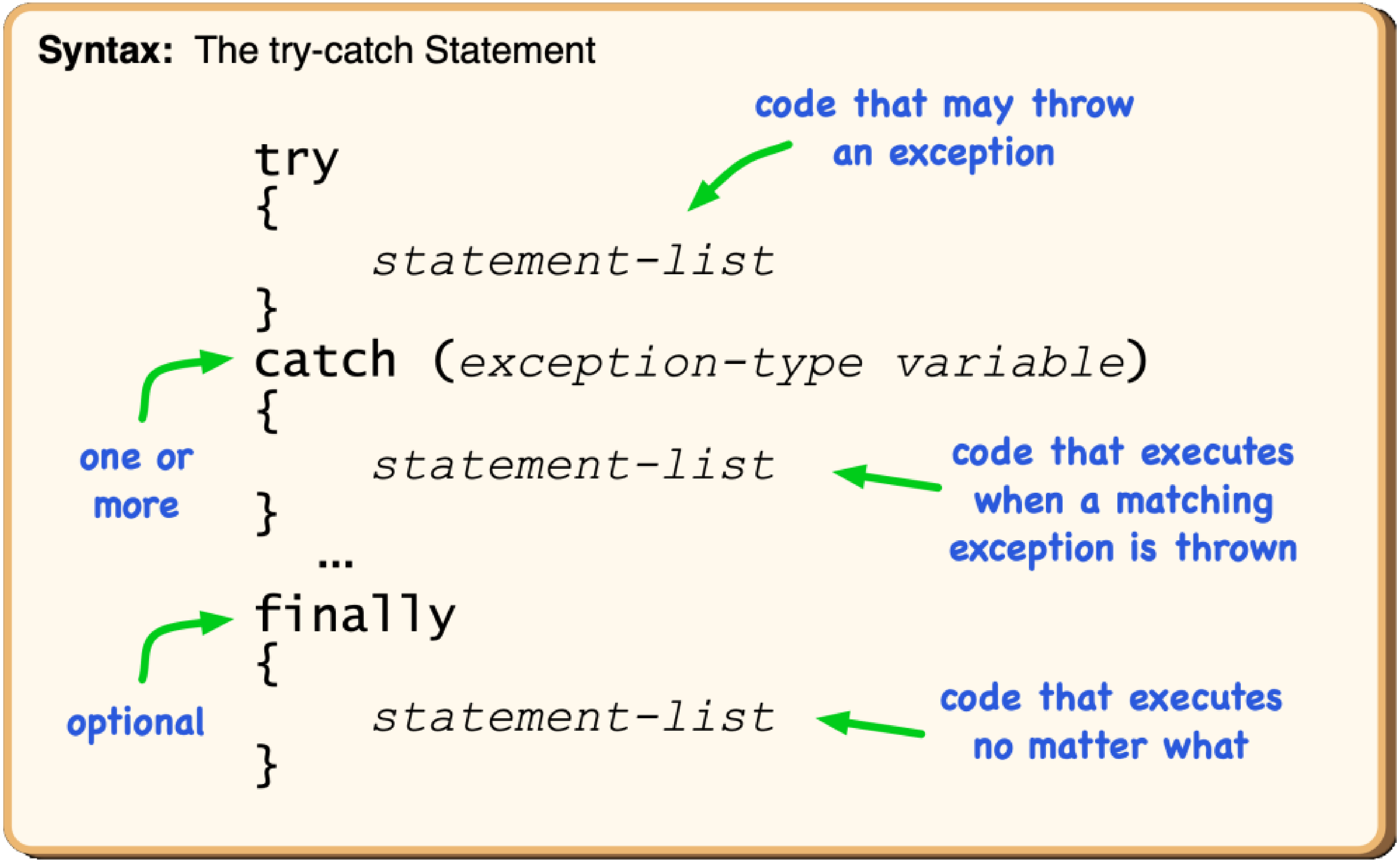
Catching an exception that is thrown due to a runtime error enables a program to respond rather than simply crashing.

In Java, this is done using the try-catch statement.

The 3 parts or blocks of a try-catch statement are:

1. try
2. catch
3. finally

The try-catch Statement



Syntax: The try-catch Statement

```
try
{
     statement-list
}
catch (exception-type variable)
{
     statement-list
}
  ...
finally
{
     statement-list
}
```

code that may throw an exception

one or more

code that executes when a matching exception is thrown

optional

code that executes no matter what

The try-catch Statement

This try-catch statement catches a NullPointerException.

```
String myString = null;
try
{
    System.out.println("Length is: " + myString.length());
}
catch (NullPointerException e)
{
    System.out.println("Hey, that's a null reference!");
}
System.out.println("We're past the try-catch now.");
```

```
Hey, that's a null reference!
We're past the try-catch now.
```

The try-catch Statement

The parenthetical expression after the catch keyword is how the caught exception is made available to be handled and printed.

```java
try
{
    int result = 45 / 0;
}
catch (ArithmeticException e)
{
    System.out.println("Hey, don't divide by zero!");
    System.out.println("Message: " + e.getMessage());
}
```

```
Hey, don't divide by zero!
Message: / by zero
```

The try-catch Statement

More information about the caught exception is available by printing its stack trace, showing where the exception occurred.

```
try
{
    int result = 45 / 0;
}
catch (ArithmeticException e)
{
    System.out.println("The stack trace is:");
    e.printStackTrace();
}
```

```
The stack trace is:
java.lang.ArithmeticException: / by zero
   at ExceptionMethods.main(ExceptionMethods.java:16)
```

The try-catch Statement

An optional finally block can come after a try-catch statement. Code in the finally block is always executed, no matter what.

```
try
{
    int result = 45 / 0;
}
catch (Exception e)
{
    System.out.println("Something horrible happened!");
}
finally
{
    System.out.println("But I'm ok with it now.");
}
```

```
Something horrible happened!
But I'm ok with it now.
```

The try-catch Statement

Because code in the finally block is always executed, it gives programmers a handy way to respond even when exceptions occur, like:

- Perform "wrap up" tasks
- Be sure an open file gets closed
- Make sure parts of an algorithm are always executed

It's a good idea to design your program to work under "normal" circumstances. For example, instead of using if statements to guard against rare problems, use exception handling as a more elegant and efficient technique.

Exception Propagation

If an exception is thrown but isn't caught in the method that threw it, it propagates up the call stack.

The exception may then be caught anywhere along the series of methods that were called to reach the point in the code where the exception occurred.

If the exception isn't caught anywhere in a try-catch statement, the program will crash… it will stop running with an error message describing the exception.

Exception Propagation

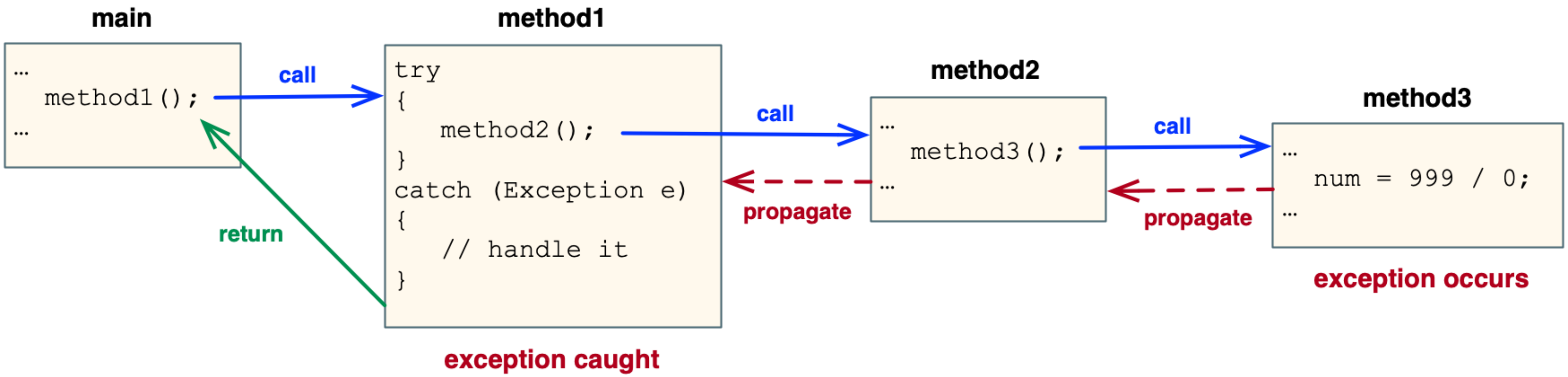
In this example, the exception throw by **method2** propagates to **method1**, where it is caught and handled.

```
public void method1()
{
    try
    {
        method2();
    }
    catch (Exception e)
    {
        System.out.println("Problem in method2!");
    }
}

public void method2()
{
    int nope = 5432 / 0;
}
```

Exception Propagation

A thrown exception affects flow of control in a way similar to a conditional statements (if and switch), repetition statements (for, for-each, and while), and method calls.



Rephactor

Exception Propagation

The two categories of exceptions in Java are:

- **Checked** – it must be caught in the method or use a throws clause to declare that might throw it.
- **Unchecked** – it doesn't need to be caught or a throws clause.

```
public void readFile() throws FileNotFoundException
{
    Scanner in = new Scanner(new File("data.txt"));
    // code to read the file
}
```

The throws clause is required since the Scanner constructor will throw a FileNotFoundException if "data.txt" does not exist.

The throw Statement

For exception handling, the try-catch statement is how runtime errors can be dealt with programmatically in Java.

You may also want to write code to raise or **throw** your own exception. This is done using the throw statement, like this:

```
if (whoWasPwned.equals("me"))
    throw new Exception("I was pwned big time!");
System.out.println("The person pwned was " + whoWasPwned);
```

If the value of whoWasPwned is "me", the exception is thrown. Otherwise, the output looks like:

```
The person pwned was Weird Al
```

The throw Statement

Exceptions are defined by the java.lang.Exception class, or can be derived via inheritance.

When you construct a new Exception, the constructor accepts a custom message to associate with the it:

```
if (true)
    throw new Exception("Custom message goes here");
```

The throw Statement

Define your own exception class, derive it from Exception.

```
public class MyException extends Exception
{
    public MyException(String message)
    {
        super(message);
    }

    // Other methods can be defined here.
}
```

The constructor can call super to specify the message for the exception (see the Inheritance topic for details).

Other Topics

- Example: Histogram
  - Encapsulation using array inside a class
- JavaFX for Graphical User Interfaces
  - Introduction to JavaFX
  - Mouse Events
  - Example: Aliens
  - FileChooserDemo example program