Chapter 18

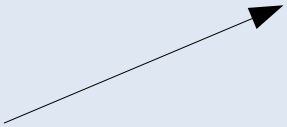
Exception Handling

Introduction

- Up to this point all error checking that we have done was with if statements and System.out.println() statements to warn the user.
- Java has an entire framework in place just for handling error checking and exceptions at runtime.
- We will be learning how to handle exceptions thrown by Java, throw pre-made exceptions, and make new exceptions of our own.

Exception Handling

```
import java.util.*;

public class Division{
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        int num1 = 0;
        int num2 = 1;

        System.out.println("Please enter two numbers to divide");

        if (input.hasNextInt()){
            num1 = input.nextInt();
        }

        if (input.hasNextInt()){
            num2 = input.nextInt();
        }

        System.out.println("The answer is: " + num1 / num2);
    }
}
```



This can cause a problem

Exception Handling

- If the user were to enter a zero for num2, then we would get an ArithmeticException runtime exception.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Division.main(Division.java:19)
```

Exception Handling

- We can try to take care of the possible problem by using another if statement

```
import java.util.*;

public class Division{
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        int num1 = 0;
        int num2 = 1;

        System.out.println("Please enter two numbers to divide");

        if (input.hasNextInt()){
            num1 = input.nextInt();
        }

        if (input.hasNextInt()){
            num2 = input.nextInt();
        }

        if (num2 == 0){
            System.err.println("You cannot divide by 0");
        }
        else{
            System.out.println("The answer is: " + num1 / num2);
        }
    }
}
```

Exception Handling

- Normally this may be a great approach because the problem we are preventing is simple and there is only one way to trigger it.
- There is an alternate to using an explicit if statement.
- We can use a try block, followed by a catch statement.

Exception Handling

```
import java.util.*;

public class Division{
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        int num1 = 0;
        int num2 = 1;

        System.out.println("Please enter two numbers to divide");

        if (input.hasNextInt()){
            num1 = input.nextInt();
        }

        if (input.hasNextInt()){
            num2 = input.nextInt();
        }

        try{
            System.out.println("The answer is: " + (num1 / num2));
        }catch(ArithmeticException ae){
            System.err.println("Exception: " + ae.toString());
        }

        System.out.println("After the try-catch...");
    }
}
```

Exception Handling

```
try {
    Code to try;
    Throw an exception explicitly or
    Call a method that throws an exception

    More code
}
catch( ExceptionType e){
    Code to deal with exception
}
```

Exception Handling

- The try block begins one or more lines of code that may throw an exception.
- Once an exception is thrown, The try block stops execution, and the execution in the appropriate catch block begins
- After the catch block has run (and if the catch block does not exit the program) then the code continues executing after the catch block

Exception Handling Advantages

- The advantage of using explicit exception handling really shows with methods.

```
public static int divide(int num1, int num2){
       if (num2 == 0){
          throw new ArithmeticException("Divisor cannot be zero");
       }

       return num1 / num2;
   }
```

- Here we are writing a method that explicitly throws and ArithmeticException if we specify 0 as the second number.

Exception Handling Advantages

```
public class Division{
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        int num1 = 0;
        int num2 = 1;

        System.out.println("Please enter two numbers to divide");

        if (input.hasNextInt()){
            num1 = input.nextInt();
        }

        if (input.hasNextInt()){
            num2 = input.nextInt();
        }

        try{
            int answer = divide(num1, num2);
            System.out.println(answer);

        }catch(ArithmeticException ae){
            System.err.println("Exception: " + ae.toString());
        }

        System.out.println("After the try-catch...");
    }
```

We are “trying” to execute this code that may cause an exception

Exception Handling Advantages

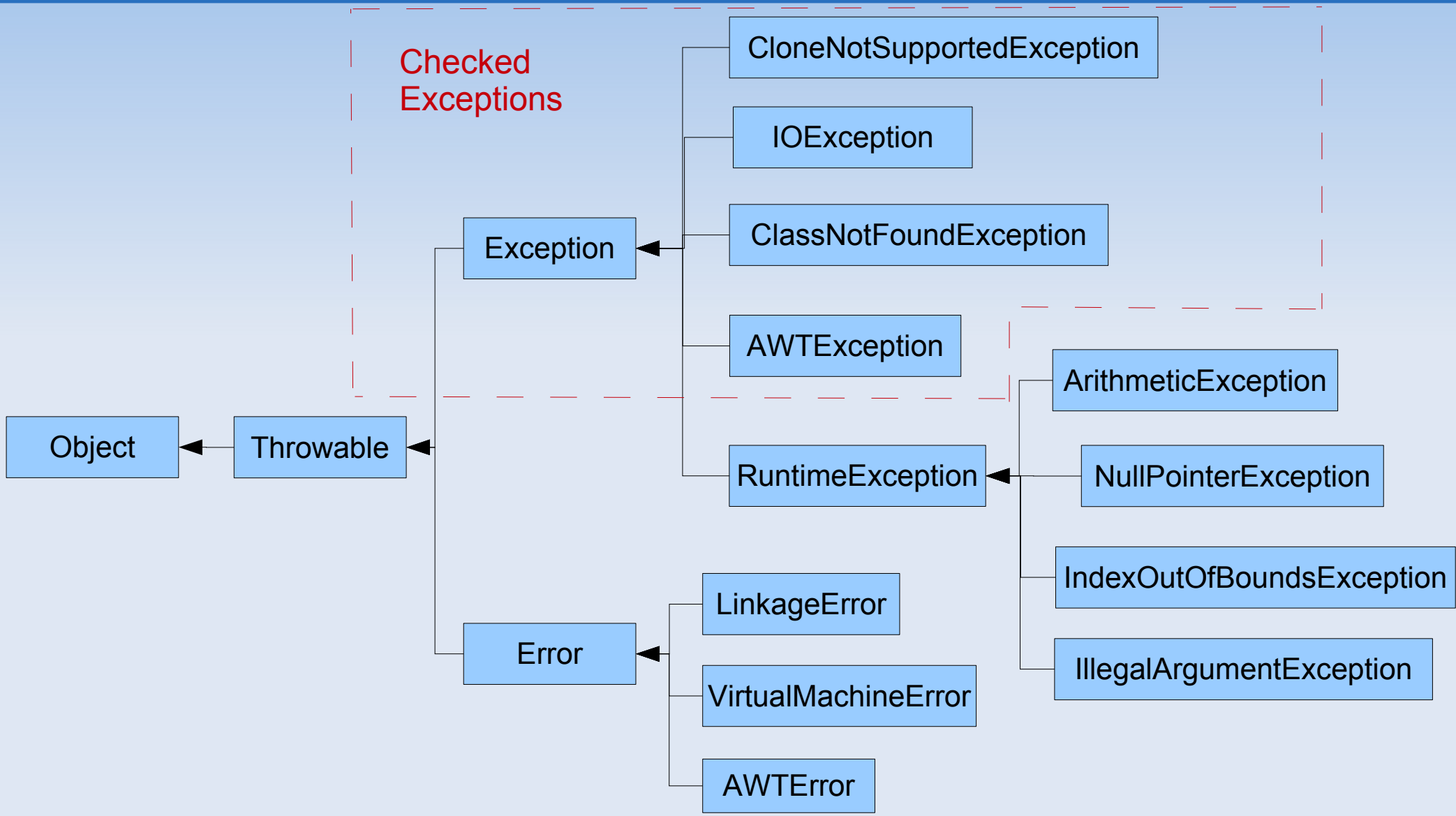
- We can see that the advantage of having exception handling is that we can stop running a block of code before our program crashed, and return control back to the calling method.
- Before exception handling, there would be no clean way to create the divide method and have it handle bad input.
- We would have had to violate good design and have the method print data to the screen (This is a no-no)

Types of exceptions

- There are three main types of exception in two groups. The two groups are
  1) Checked Exceptions
  2) Unchecked Exceptions

- The three main types of exceptions are
  1) Exceptions (checked)
  2) RuntimeExceptions (unchecked)
  3) Errors (unchecked)

Types of exceptions



Unchecked Exceptions

- Unchecked Exceptions – These are exceptions that can occur but do not have to be explicitly handled.
- Every time you access an array you are at risk of throwing an IndexOutOfBoundsException.
- You do not need to handle this in a try-catch block because there is really no way recover from such an exception.
- Generally, unchecked exceptions are exceptions that cannot be recovered from.
- RuntimeException, Error, and all of there subclasses make up the unchecked exceptions.

Checked Exceptions

- Checked Exceptions – These are exceptions that MUST be handled by the programmer.
- The rule for checked exceptions is “declare or handle” which means you MUST surround the exception throwing code with a try catch block, or you MUST add the throws statement to the method declaration

```
public String getStringFromFile(File f) throws IOException{
```

Handling Errors

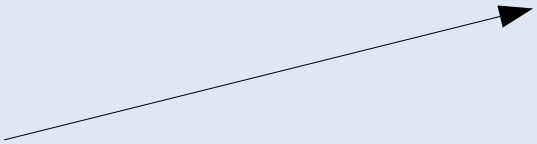
- You should never explicitly catch or handle Errors. Errors are usually low level system problems that occur.
- If you want to make your own unchecked exception, extend RuntimeException and not Error.
- Extending Error is bad practice unless you are describing an exception that is caused by some hardware, or low level software (like a class loader)

Declaring Exceptions

- You only need to declare checked Exceptions when you are throwing them.
- You can declare any number of checked or unchecked exceptions
- You declare an exception by declaring that a method throws an exception of that type

```
public String getStringFromFile(File f) throws IOException{
```

```
public int divide(int num1, int num2) throws ArithmeticException
```



Optional because this is an unchecked exception

Throwing Exceptions

- You can throw an exception in two ways
  1) Declare an instance of the exception type
  2) Throw an anonymous instance of the exception type

```
public String getStringFromFile(File f) throws IOException{
       IOException ioe = new IOException("Cannot open file");
       throw ioe;
   }
```

```
public String getStringFromFile(File f) throws IOException{
       throw new IOException("Cannot open file");
   }
```

NOTE: use `throws` to declare an exception, and `throw` to actually throw the exception

Throwing Exceptions

- You can throw as many exceptions from a method as you want
- Declare them in a comma separated list

```
public int DoStuff() throws ArithmeticException,
                            IOException,
                           IndexOutOfBoundsException{

      //Code here that throw those exceptions

   }
```

NOTE: You only need to declare the Exception if you are throwing it, if you are catching it and handling it you don't declare it

Catching Exceptions

- If you want to handle the exception caused by some code you can surround the exception causing code in a try block
- The try block MUST be immediately followed by one or more catch blocks.
- The order of the catch blocks matter!

Catching Exceptions

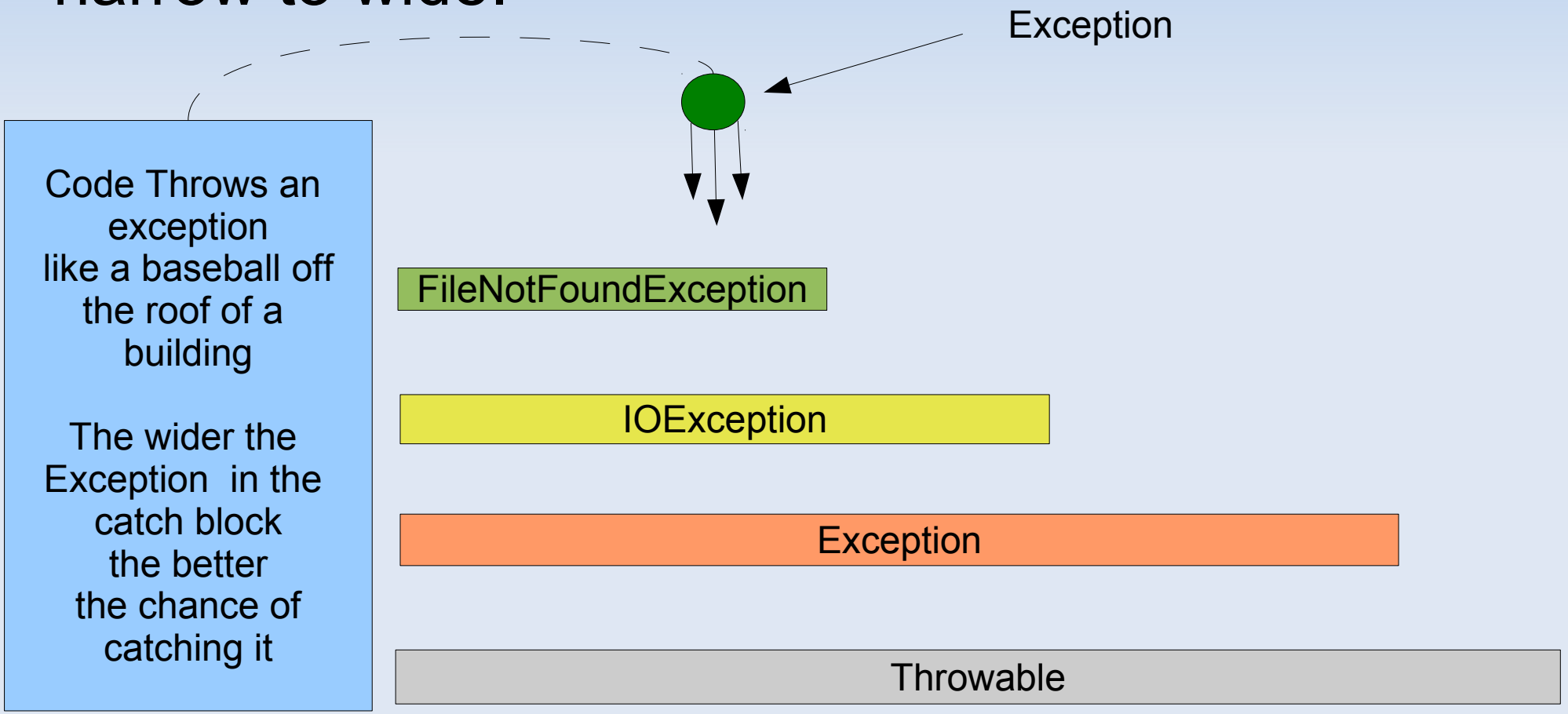
```
import java.io.*;
import java.util.*;
public class TestExceptions{

    public static void main(String[] args){
        File inputFile = new File(args[0]);
        Scanner input = null;

        try{
            input = new Scanner(inputFile);
            while(input.hasNextLine()){
                System.out.println(input.nextLine());
            }
        }
        catch (FileNotFoundException fnf){
            System.out.println(args[0] + " Was not found!");
        }
        catch(IOException ioe){
            System.out.println("An IO Exception has occured");
        }
        catch (Exception e){
            System.out.println("Some unexpected exception occured: " +
                                                        e.toString());
        }
    }

}
```

Catching Exceptions

- The scanner constructor throws a FileNotFoundException if the file being requested does not exits, or cannot be opened for reading.
- FileNotFoundException is a sub class of IOException, which is a subclass of Exception
- Because of this inheritance, FileNotFoundException is a checked Exception

Catching Exceptions

- The order in which I declared the catch statements is very important. You must catch narrow to wide.



Catching Exceptions

- If you have multiple catches in the wrong order, you will get a compiler error

```
catch (Exception e){
    System.out.println("Some unexpected exception occured: " +
                                                    e.toString());
}
catch (FileNotFoundException fnf){
    System.out.println(args[0] + " Was not found!");
}
catch(IOException ioe){
    System.out.println("An IO Exception has occured");
}
```

```
TestExceptions.java:19: exception java.io.FileNotFoundException has already been caught
        catch (FileNotFoundException fnf){
        ^
TestExceptions.java:22: exception java.io.IOException has already been caught
        catch(IOException ioe){
        ^
2 errors
```

Catching Exceptions

- You don't need to catch every possible exception type, only the type declared or a wider type.

```
public class TestExceptions{

    public static void main(String[] args){
        File inputFile = new File(args[0]);
        Scanner input = null;

        try{
            input = new Scanner(inputFile);
            while(input.hasNextLine()){
                System.out.println(input.nextLine());
            }
        }
        catch (IOException e){
            System.out.println("Some IOException occured: " +
                                                    e.toString());
        }

    }

}
```

Checked Exception Choices

- Remember that with checked exceptions you have the option of handle or declare

```
import java.io.*;
import java.util.*;

public class TestExceptions{

    public static void main(String[] args) throws IOException{
        File inputFile = new File(args[0]);
        Scanner input = null;

        input = new Scanner(inputFile);
        while(input.hasNextLine()){
            System.out.println(input.nextLine());
        }
    }

}
```

When should you handle Exceptions

- Generally you should only handle the exceptions in your main program (in the main method, or main class)
- If you are writing a new class that is to be used somewhere else, you should throw the exceptions that you encounter
- If you can handle an exception with 100% transparency, it is not incorrect to handle an exception inside of a non-main class.

Creating your own checked exception

- It is really easy to create an exception (checked or unchecked)
- If you create a checked exception, you must extend the Exception class.
- If you use a checked exception you are FORCING the programmer to declare or handle this exception.

Creating checked exceptions

```
public class ShapeException extends Exception{
    private String description;

    public ShapeException(){

    }

    public ShapeException(String description){
        super(description);
        this.description = description;
    }

    public ShapeException(String description, Throwable t){
        super(description, t);
        this.description = description;
    }

    public String getDescription(){
        return this.description;
    }
}
```

Creating checked exceptions

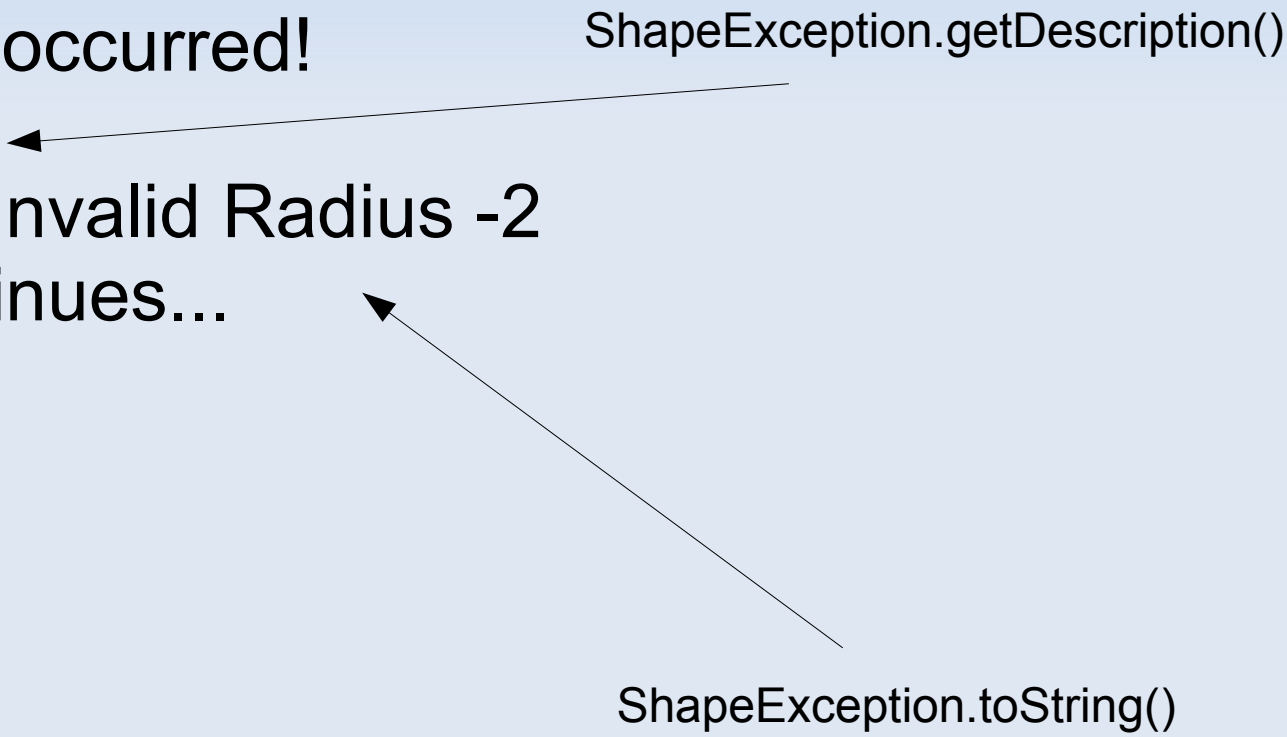
```
public class Circle{
    int radius;

    public Circle(int radius) throws ShapeException{
        if (radius <= 0){
            throw new ShapeException("Invalid Radius " + radius);
        }

        this.radius = radius;
    }

    public int getRadius(){
        return this.radius;
    }

    public void setRadius(int radius) throws ShapeException{
        if (radius <= 0){
            throw new ShapeException("Invalid Radius " + radius);
        }

        this.radius = radius;
    }
```

Creating checked exceptions

```
    public static void main(String[] args){
        Circle jamesCircle = null;
        Circle slocumCircle = null;

        try{
            jamesCircle = new Circle(5);
            slocumCircle = new Circle(-2);

            System.out.println("Circles Created");
        }
        catch(ShapeException ce){
            System.out.println("An Exception has occurred!");
            System.out.println(ce.getDescription());
            System.out.println(ce.toString());
        }

        System.out.println("The program continues...");
    }
}
```

Results

```
>java Circle
An Exception has occurred!
Invalid Radius -2
ShapeException: Invalid Radius -2
The program continues...
```



Checked Exceptions

- What happens if we don't use the try catch block?

```
Circle.java:28: unreported exception ShapeException; must be caught or declared to be thrown
        jamesCircle = new Circle(5);
                      ^
Circle.java:29: unreported exception ShapeException; must be caught or declared to be thrown
        slocumCircle = new Circle(-2);
                       ^
2 errors
```

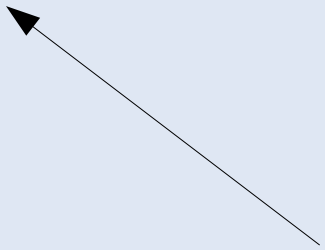
Alternatively we can declare that main throws ShapeException, or Exception

Exceptions and Inheritance

- When extending a class and overriding a method that throws checked exceptions you must follow some rules
  1) If you call a super() method that throws a checked exception, you must also declare to throw the same exception
  2) You cannot throw new checked exceptions!
  3) You can throw as many unchecked exceptions as you want
  4) You can throw a more narrow exception, but not a wider exception
  5) If you override a method that throws a checked exception, but your method does not throw any exceptions, you don't need to declare it.

Exceptions and Inheritance

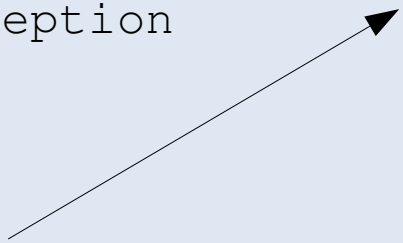
```
public class Sphere extends Circle{
   private int x, y, z;

   public Sphere(int radius) throws ShapeException{
      super(radius);
   }

   public void setRadius(int radius){
      //code doesn't throw a ShapeException, so
      //don't have to declare one.
   }
}
```



This is legal to not declare an exception that you don't throw

Exceptions and Inheritance

```
import java.io.*;

public class Sphere extends Circle{
    private int x, y, z;

    public Sphere(int radius) throws ShapeException{
        super(radius);
    }

    public void setRadius(int radius) throws IOException{
        //code that throws an IOException
    }
}
```

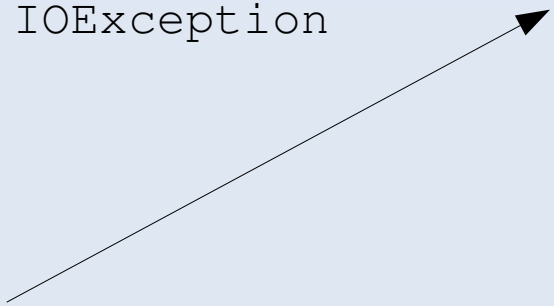


```
Sphere.java:10: setRadius(int) in Sphere cannot override setRadius(int) in Circle; overridden
method does not throw java.io.IOException
    public void setRadius(int radius) throws IOException{
                ^
1 error
```

Cannot throw a new exception!

Exceptions and Inheritance

```
public class Sphere extends Circle{
    private int x, y, z;

    public Sphere(int radius) throws ShapeException{
        super(radius);
    }

    public void setRadius(int radius) throws Exception{
        //code that throws an IOException
    }
}
```



Sphere.java:10: setRadius(int) in Sphere cannot override setRadius(int) in Circle; overridden method does not throw java.lang.Exception

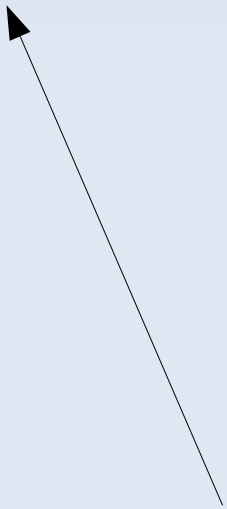
```
    public void setRadius(int radius) throws Exception{
                ^
```

1 error

Cannot throw a wider exception!

Exceptions and Inheritance

```
public class Sphere extends Circle{
    private int x, y, z;

    public Sphere(int radius) throws ShapeException{
        super(radius);
    }

    public void setRadius(int radius) throws SphereException{
        //code that throws a ShpereException
    }

    class SphereException extends ShapeException{
        //code here
    }
}
```



Perfectly fine to throw a narrower exception then the one declare in the parent class.

Chaining Exceptions

- Just because you have a catch block doesn't mean the exception has to be completely handled there.
- You can throw the exception again from the catch block to chain exceptions.
- By chaining exceptions, you can handle the exception in multiple ways, or find out more info about what caused the exception.

Chaining Exceptions

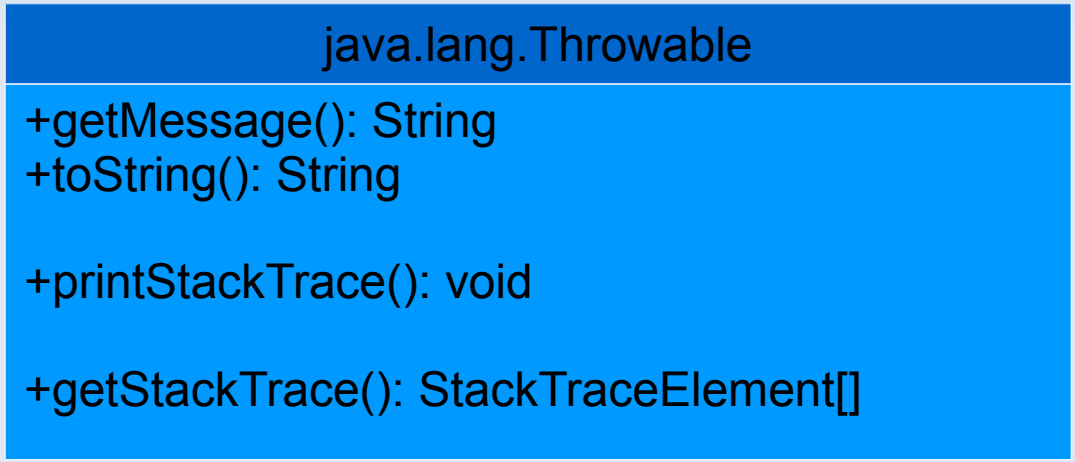
```
public class ChainedExceptionDemo {
  public static void main(String[] args) {
    try {
      method1();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public static void method1() throws Exception {
    try {
      method2();
    }
    catch (Exception ex) {
      throw new Exception("New info from method1", ex);
    }
  }

  public static void method2() throws Exception {
    throw new Exception("New info from method2");
  }
}
```

Result

```
java.lang.Exception: New info from method1
    at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:16)
    at ChainedExceptionDemo.main(ChainedExceptionDemo.java:4)
Caused by: java.lang.Exception: New info from method2
    at ChainedExceptionDemo.method2(ChainedExceptionDemo.java:21)
    at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:13)
    ... 1 more
```

Getting info from the exception

- There are a few methods to help you figure out where the program went wrong



The finally clause

- There is one more keyword when dealing with exceptions
- The keyword finally describes a block of code that executes NO MATTER WHAT
- The finally block will run if an exception is thrown, or if the try block was completed successfully
- This is where you can free system resources regardless of whether an exception has occurred or not

The finally clause

```
import java.io.*;
import java.util.*;

public class TestExceptions{

    public static void main(String[] args){
        File inputFile = new File(args[0]);
        Scanner input = null;

        try{
            input = new Scanner(inputFile);
            while(input.hasNextLine()){
                System.out.println(input.nextLine());
            }
        }
        catch(IOException ioe){
            System.out.println("An IO Exception has occured");
        }
        finally{
            if (input != null){
                input.close();
            }
        }
    }
}
```

The finally clause

- The finally block will run even if a return statement is encountered inside of the try statement.
- The only time a finally block will not run is if System.exit() is called.
- If you use a finally block, you don't need a catch statement, but this is bad practice and you should always use one.

Lab Assignment

Labs

- Page 621 #18.2 & 18.3

Homework

- Page 622 #18.8

Acknowledgments

Introduction to Java Programming by Y. Daniel Liang