

# Abstract class, Interface *again* & Exception handling

# Pre-Lecture Videos



## Abstract Classes

<https://goo.gl/VmNOhj> (03:33)

## Interfaces

<https://goo.gl/ZJynK9> (04:30)

## Exceptions

Understanding syntax errors vs. runtime exceptions:

<https://goo.gl/FEblu4> (05:33)

Handling exceptions with try/catch: <https://goo.gl/bxGyOo> (04:24)

Creating multiple catch blocks: <https://goo.gl/PXMa7F> (03:29)

Throwing custom exceptions: <https://goo.gl/KeWd9T> (03:19)



# Content



**Abstract class *again***

**Interface *again***

**Random numbers**

**Exceptions**

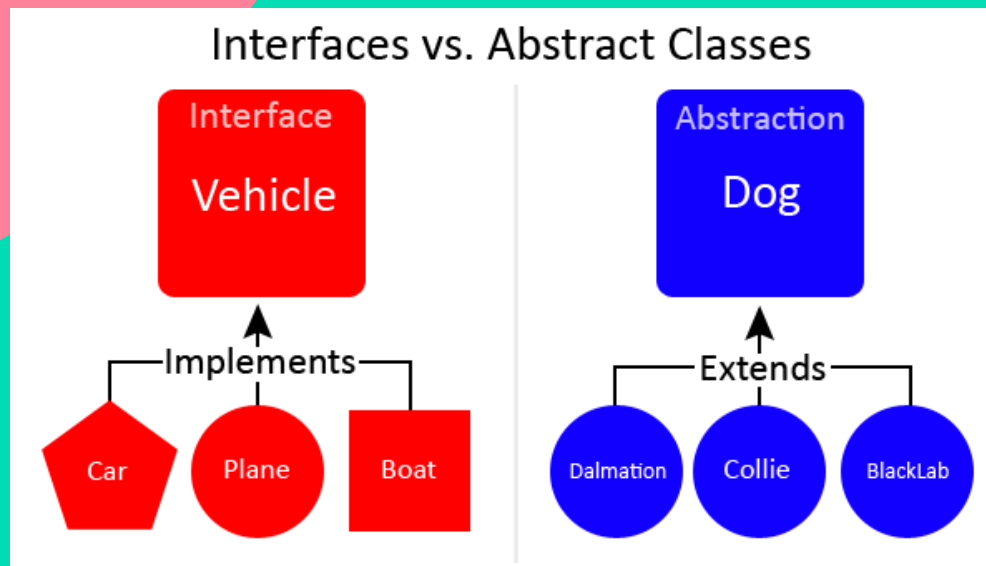


## Abstract Classes & Interfaces

Abstract classes and interfaces implement key aspects of object-oriented design.

Any programmer has to decide whether it is beneficial for their Java classes to use and implement an abstract class or an interface or often a combination of these.

There are trade-offs in using one or the other, and the final decision will be dictated by the context of the application you are building



# Abstract Classes & Interfaces

Both provide a level of abstraction that can improve a program's design and influence how classes are implemented.

**Abstraction** is a process of hiding the implementation details (how things work) from the user and other programmers.

In Java Abstraction is achieved using **abstract classes and/or interfaces**

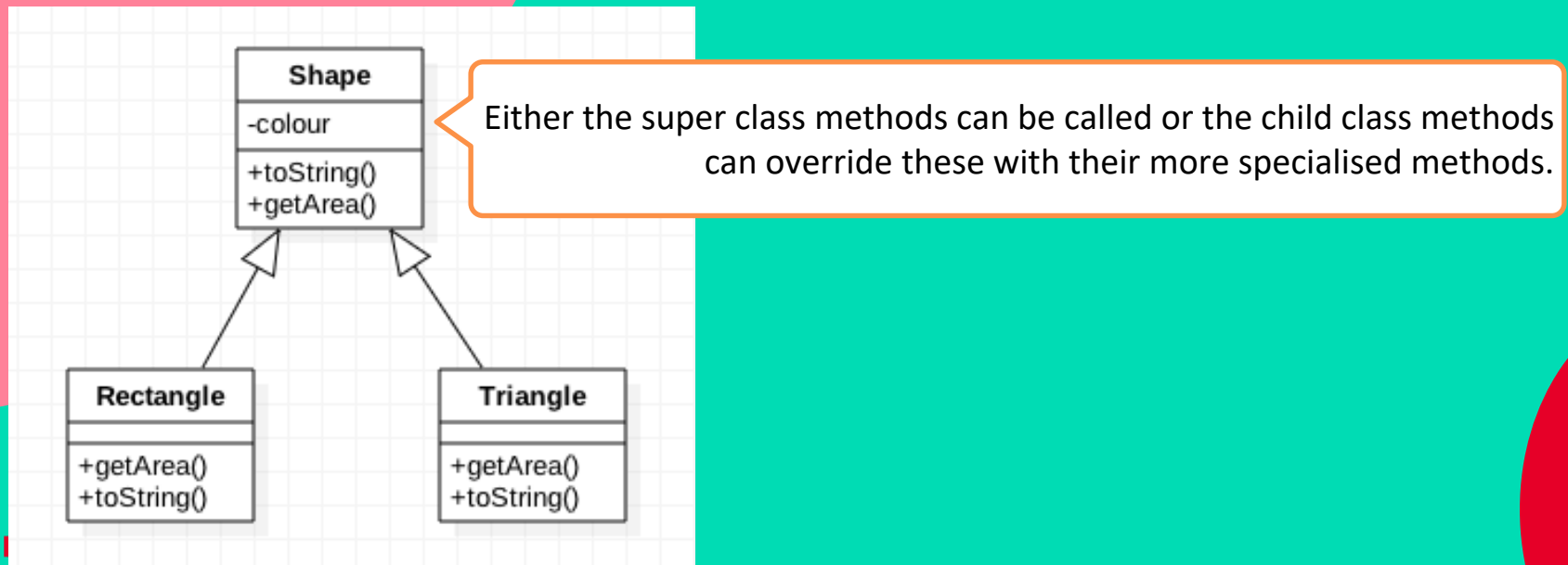
## Inheritance with an abstract class

Imagine a simple inherited class structure depicted in the class diagram below.

We can observe that two methods are inherited by the sub classes from the parent class **Shape**.

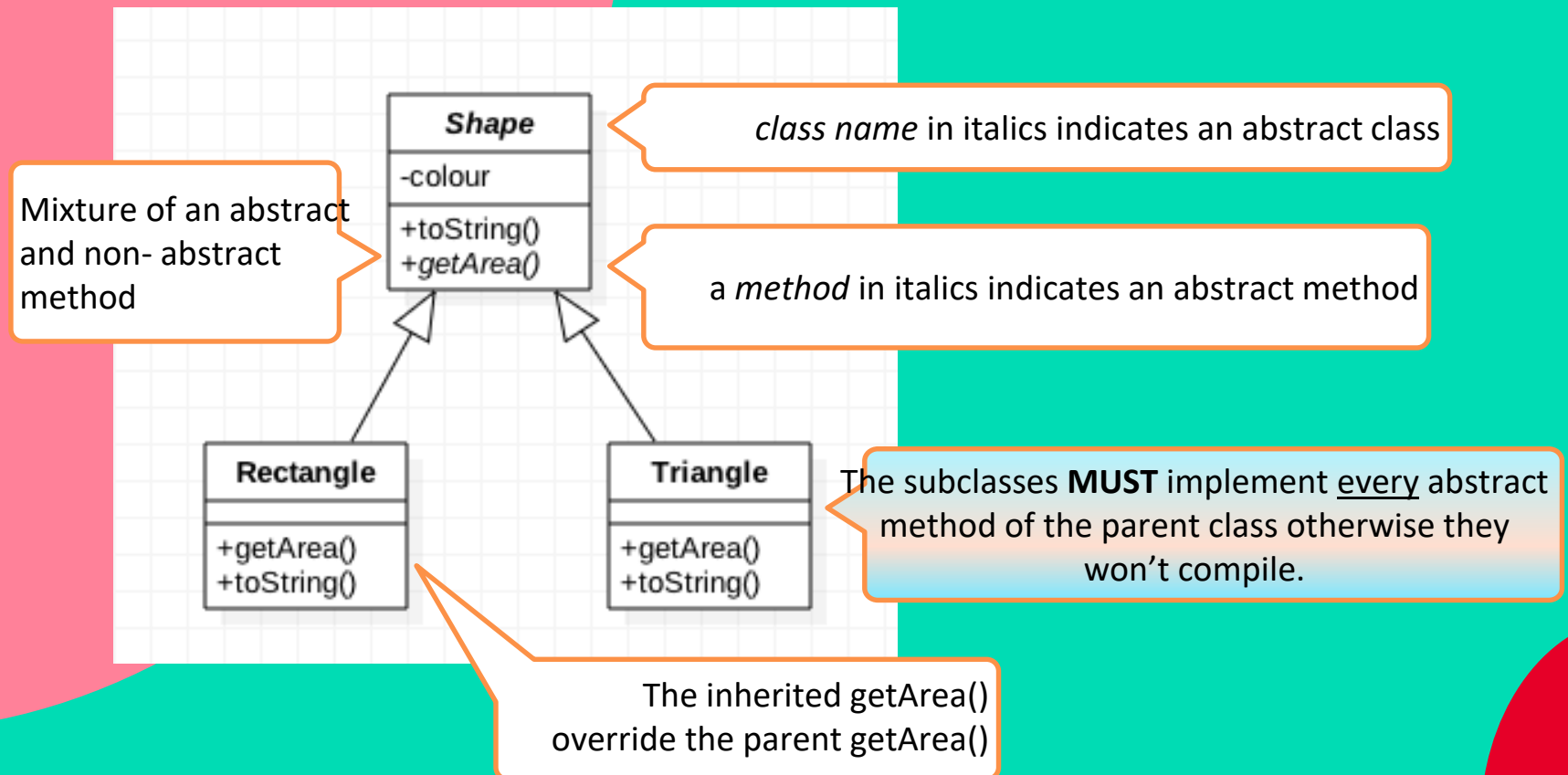
Possible to declare a class abstract even it has no abstract methods - preventing users instantiating

Abstract classes may have instance variables in their declaration



## Inheritance with an abstract class

**Shape** is an abstract class with **getArea()** an abstract method.



## So what do these changes mean?

```
public abstract class Shape  
{
```

abstract keyword used in declarations

```
    public abstract void getArea();  
}
```

The method contains no code & no { }

- Shape class can no longer be instantiated.
- The **getArea()** method in Shape class cannot be called. Without being abstract it still could.
- Inherited classes Rectangle & Triangle class MUST have a **getArea()** method. They can specify the code they want for this method.

By making **getArea()** abstract you are forcing all child classes to implement that method. It effectively creates a sort of template, a requirement on the inheritance structure. If any further child classes are added, they will have to implement this method, as well.



# Characteristics of an Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain abstract methods i.e., methods without code statements.
- If a class has at least one abstract method, then the class must be declared abstract.
- If a class is declared abstract it **cannot be instantiated**.
- If you inherit an abstract class you have to provide implementations for all the abstract methods in it.

This forces design constraints on any inherited classes.  
It is a means of specifying the minimum subset of methods  
that the child class must implement

## Abstract methods are optional

You do not need to implement an **abstract class**, however it helps defining a discipline and structure for your APIs to adhere to

- In particular, the **polymorphic** processing of child class objects becomes more precise and better structured
- Super class objects cannot be instantiated. Thus processing only has to deal with child objects
- A level of abstraction occurs because other programmers adding child classes in the future don't need to be concerned with the abstract methods. They only need to provide the implementations that are relevant to their codebase

# Java Interface

The purpose of an interface is similar to an abstract class and abstract methods.

It acts as a template that specifies methods that must be implemented into every class that you choose to connect to it via the reserved **implements**.

An Interface, any class can 'connect' by using the reserved word **implements** in their class declaration.

**Inheritance (class hierarchies) model an 'is a' relationship**

**Interfaces model a 'behaves like' relationship.**

**A class can implement many interfaces, but inherit from only one super class.**



# Random Numbers



## Random Numbers (concepts)

- Most computer programs do the same thing every time they run (they are **deterministic**)
- Generally, a good thing, since we expect the same calculation to yield the same result.
- Some applications however should produce less predictable behaviours
- Games are an obvious example, but there are many others, such as some scientific simulations
- Making a program nondeterministic is a hard problem, because it's impossible for a computer to generate truly random numbers.
- There are algorithms though that generate unpredictable sequences called pseudo-random numbers.
- For most applications, and certainly for our work in PT, they are as good as random.

## Random Numbers (examples)

```
public static int[] randomArray(int size)
{
    Random random = new Random();
    int[] a = new int[size];

    for (int i = 0; i < a.length; i++)
    {
        a[i] = random.nextInt(100);
    }

    return a;
}
```

## Random Numbers (examples)

```
private static int getRandomNumberInRange(int min, int max)
{
    if (min >= max)
    {
        throw new IllegalArgumentException("max > min, please!");
    }

    Random r = new Random();
    return r.nextInt((max - min) + 1) + min;
}
```

# Random Numbers (examples)



**Please refer to the [official Java docs](#) for more information on additional methods**

Generation of other variable types (e.g., float, double, Boolean, long, etc.)

Generation of “streams” e.g., Random.ints()

```
public static int getRandomNumberInts(int min, int max)
{
    Random random = new Random();

    return random.ints(min, (max+1)).findFirst().getAsInt();
}
```





# Exceptions

# What is an exception?

An Exception is triggered by an unexpected situation or event that occurs during the execution of a program and may lead to its termination.

An Exception in Java is an object which contains information about the runtime error that has occurred. These Exception objects are automatically created when an unexpected situation arises.

When an exception is generated, the Java Runtime Environment searches your code method by method looking for an appropriate handler to 'deal' with the error.

If an appropriate handler code is found then the exception is handled normally, otherwise the program terminates with a **crash!!!**

# Checked & Unchecked Exceptions

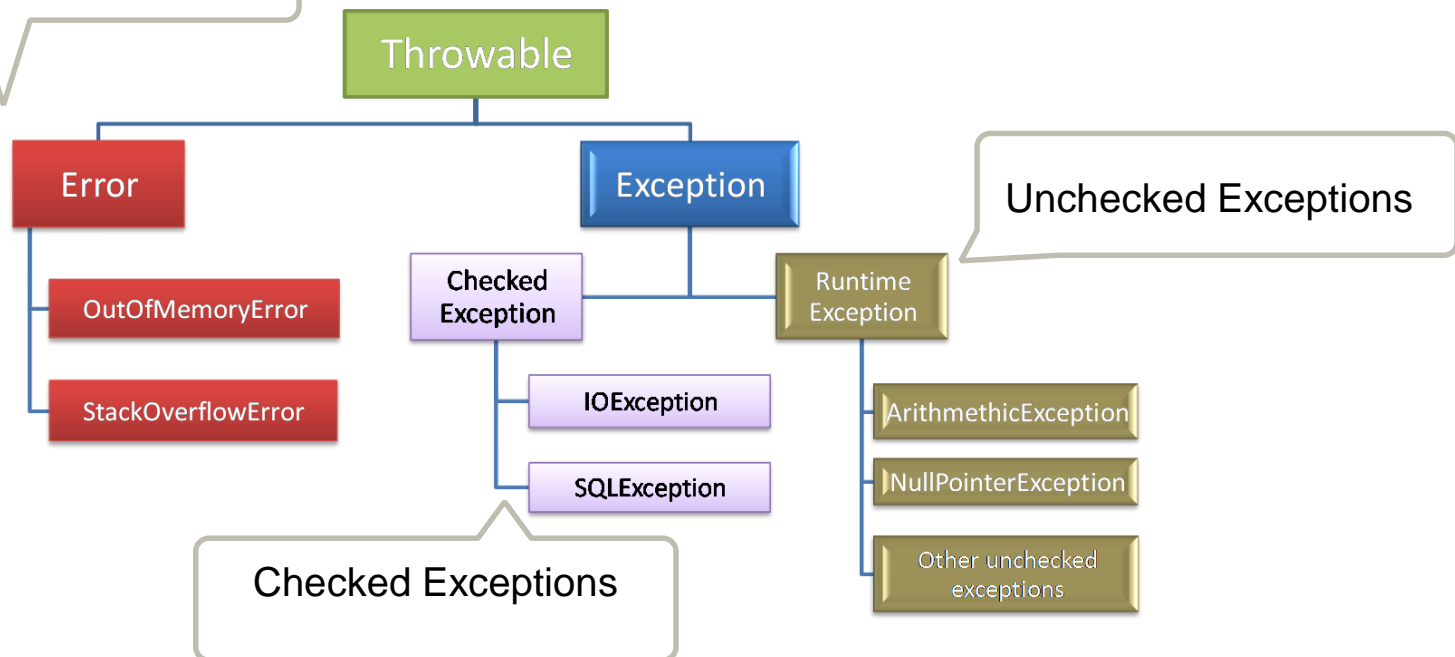
Java distinguishes between **checked** and **unchecked** exceptions.

- **Checked** exceptions are those Exceptions for which we need to provide exception handling because they are identified at compile time. Your program will not run until you have provided code to handle the exceptions.
- **Unchecked** exceptions are those for which providing exception handlers is optional as the compiler does not check for them. Hence your program might compile but you could have runtime errors.

<http://www.javawithus.com/tutorial/exception-hierarchy>

# Throwable class hierarchy

Errors are not Exceptions.  
Errors thrown by the Error class cannot be caught or 'handled'.



<http://www.javatutorialguide.com/core-java/interview-questions/java-exception-handling-interview-questions-answers.php>

## compile errors from example

```
Exception in thread "main" java.lang.Error: Unresolved  
compilation problems: Unhandled exception type  
FileNotFoundException  
Unhandled exception type IOException  
Unhandled exception type IOException
```

<http://beginnersbook.com/2013/04/java-checked-unchecked-exceptions-with-examples/>

## Method 1 - Handling the checked exceptions with....throws keyword

```
import java.io.*; class Example {  
    public static void main(String args[]) throws IOException {  
        FileInputStream fis = null;  
        fis = new FileInputStream("C:/myfile.txt");  
        int k;  
        while(( k = fis.read() ) != -1)  
            System.out.print((char)k);  
        fis.close(); }  
}
```

normally a throws reserved w  
is appended to a method  
declaration

*The throws statement only needs to mention IOException because this is a parent class of FileNotFoundException class. Alternatively, you could list the exceptions. See below.*






```
public static void main(String args[]) throws IOException, FileNotFoundException.
```

<http://beginnersbook.com/2013/04/java-checked-unchecked-exceptions-with-examples/>

## Method 2 - Handling exceptions with try - catch blocks

```
FileInputStream fis = null;  
  
try{  
    fis = new FileInputStream("B:/myfile.txt");  
}catch(FileNotFoundException fnfe){  
    System.out.println("The specified file is not " + "present at  
the given path");}  
  
int k;  
  
try{  
    while(( k = fis.read() ) != -1) {  
        System.out.print((char)k); }  
        fis.close();}  
catch(IOException ioe){  
    System.out.println("I/O error occurred: " + ioe); }
```



With this technique each checked exception is enclosed in a **try - catch block**.

This allows for better user feedback via more specific messages  
Should an exception be thrown

# Unchecked exceptions

## Unchecked Exceptions:

- Compiler doesn't check if they are handled in code.
- Caused mostly because of programmer error like NullPointerException
- Runtime Exceptions, when identified, should usually be prevented with code rather than just handling exceptions that maybe generated.

Often, these exception occur due to bad data provided by users during some interaction. It is up to the programmer to anticipate such conditions in advance and to handle them e.g., via data validation.

All Unchecked exceptions are direct sub classes of the RuntimeException class.

## Common examples:

**ArrayIndexOutOfBoundsException** ..... Array index is out-of-bounds.

**ArithmeticException**..... Arithmetic error, such as divide-by-zero

<http://beginnersbook.com/2013/04/java-checked-unchecked-exceptions-with-examples/>



# Unchecked exceptions


```
class Example {  
    public static void main(String args[ ])  
    {  
        int num1= 10;  
        int num2 = 0;  
  
        /*Throwing ArithmeticException because dividing by zero*/  
  
        int res = num1 / num2;  
        System.out.println(res);  
    }  
}
```

If you compile this code, it would compile successfully however when you will run it, it would throw `ArithmeticException`

**Q:** *What code could be written to prevent the exception from occurring?*

## Handling a non specific exception

```
try{  
  
    int arr[ ]={1,2,3,4,5};  
  
    System.out.println(arr[7]);  
  
}  
catch (Exception e) {  
    System.out.println("An error has occurred.");  
  
    System.out.println("The problem relates to: " + e.toString());  
  
}
```



very generic exception handling

Program output:

An error has occurred.

[The problem relates to: java.lang.ArrayIndexOutOfBoundsException: 7](#)

## Multiple catches

explicit exception throw

explicit exception throw

multiple catches for  
the throws

```
try
{
    System.out.println("Enter number of widgets produced:");
    Scanner keyboard = new Scanner(System.in);
    int widgets = keyboard.nextInt( );
    if (widgets < 0)
        throw new NegativeNumberException("widgets");

    System.out.println("How many were defective?");
    int defective = keyboard.nextInt( );
    if (defective < 0)
        throw new NegativeNumberException("defective widgets");

    double ratio = exceptionalDivision(widgets, defective);
    System.out.println("One in every " + ratio +
        " widgets is defective.");
}
catch(DivideByZeroException e)
{
    System.out.println("Congratulations! A perfect record!");
}
catch(NegativeNumberException e)
{
    System.out.println("Cannot have a negative number of " +
        e.getMessage( ));
}
System.out.println("End of program.");
}
```

## Order of multiple catch statements

When using multiple exception handling, place the more specific class higher up the hierarchy, as below.

As IOException is a parent of FileNotFoundException if it was placed first then all IO exceptions would be caught in that context. More specific exceptions would be ignored.

```
try{  
    //call some methods that throw IOException's  
} catch (FileNotFoundException e)  
{  
    //code statement  
}  
catch (IOException e)  
{  
    //code statement  
}
```

<http://tutorials.jenkov.com/java-exception-handling/exception-hierarchies.html>

## The finally block

The finally block **always** executes when the try block exits.

This ensures that the finally block is executed even if an unexpected exception occurs. But finally{ } is useful for more than just exception handling — it allows the programmer to cleanup code and perform ‘maintenance’ tasks such as closing files. It is good practice to include a finally block attached to your key try - catch especially, if they are accessing files or databases.

```
class Example {  
    public static void main(String args[]) {  
        try {  
            System.out.println("First statement of try block");  
            int num=45/0;  
            System.out.println(num);  
        }  
        catch(ArithmeticException e) {  
            System.out.println("ArithmeticException");  
        }  
        finally {  
            System.out.println("finally block");  
        }  
        System.out.println("Out of try-catch-finally block");  
    }  
}
```

## Writing your own exceptions (class)

The code below defines your own class. In their most basic form custom Exception subclass types only need to override two constructors, one that accepts an error message and one that doesn't.

```
public class myException extends RuntimeException
{
    public myException(String message)
    {
        //constructor with one parameter
        super(message);
    }
    public myException()
    {
        //constructor with no parameter
        super();
    }
}
//class
```