# ET0736

## Lesson 5

Exception Handling, Abstract class and Interface

# Topics

- Exceptions by examples
- Multiple catch
- try-catch-finally
- throw vs throws
- Checked/unchecked exceptions
- Try-with-reference
- Interface
- Abstract class

# Exception

- Is an unexpected event, caused by runtime errors.
- may terminate abnormally without codes handling exceptions
- is coded in ***try-catch*** structure or ***try-catch-finally*** structure
- There are many different types of exceptions, generated by processing of different methods.
- To know which exception to ‘catch’ (or pay attention to), always refer to API document.
- The ***try*** block houses the intended operational code
- The execution of the program will skip and jump to the ***catch*** block if an exception occurs while executing the ***try*** block. The rest of the code in the ***try*** block will be skipped.

# Arithmetic Exception

In the example below, x is divided by y, which is a zero, and because of this, an *ArithmeticException* is thrown.

```
try{
     System.out.println (“Trying …");
     int x=30, y=0;
     int z=x/y;
     System.out.println ("Answer: "+ z);
 }
 catch(ArithmeticException e){
     System.out.println ("Cannot be divide a number by zero");
 }
```

Output:
**Trying …**
**Cannot be divide a number by zero**

# ArrayIndexOutOfBounds Exception

```
try{
     System.out.println ("Trying …");
     int x[] = new int[10];
     x[10] =888;
     System.out.println ("Done!");
 }
 catch(ArrayIndexOutOfBoundsException e){
     System.out.println (“No such index in array");
 }
```

Output:
**Trying …**
**No such index in array**

# NumberFormat Exception

```
try{
     System.out.println ("Trying …");
     double x = Double.parseDouble ("SP123");
     System.out.println (“x: "+ x);
 }
 catch(NumberFormatException e){
     System.out.println (“Wrong format");
 }
```

Output:
**Trying …**
**Wrong format**

# Multiple Catch

```
try{
   int x[]=new int[7];
   x[4]=88/0;
   System.out.println("First print statement in try block");
 }
 catch(ArithmeticException e){
   System.out.println("Warning: ArithmeticException");
 }
 catch(ArrayIndexOutOfBoundsException e){
   System.out.println("Warning: ArrayIndexOutOfBoundsException");
 }
 catch(Exception e){
   System.out.println("Warning: Some Other exception");
 }

 System.out.println("Out of try-catch block...");
```

Output:

**Warning: ArithmeticException**

**Out of try-catch block…**

# try-catch-finally

The ***finally*** block will always execute even if there is no exception thrown from the ***try*** block.

```
    try{
          int x=888/0;
          System.out.println(x);
       }
       catch(ArithmeticException e){
         System.out.println("Number should not be divided by zero");
       }

       finally{
          System.out.println("This is finally block");
       }
       System.out.println("Out of try-catch-finally");
   }
```

Output:

**Number should not be divided by zero**
**This is finally block**
**Out of try-catch-finally**

# throw

- The ***throw*** keyword is used ***inside*** a method.
- It is used when it is ***required to throw*** an Exception explicitly.

```
public static void validate(int age) {
    if(age<18) {
        throw new ArithmeticException ("Person is not
        eligible for competition");
    }
    else {
        System.out.println("Person is eligible for
    competition");
    }
}

public class Test {
  public static void main(String[] args)  {
    validate (7);
  }
}
```

**Compiled successfully but the output is unpleasant**

For this example, a proper declaration of the ***validate()*** method should include the ***throws*** keyword and the caller program (i.e. in this case, the ***main()*** ) should try to handle the exception (by ***try-catch***) or propagate the exception to the next-higher level.

```
public static void validate(int age) {
    if(age<18) {
        throw new ArithmeticException ("Person is not eligible for
        competition");
    }
    else {
       System.out.println("Person is eligible for competition");
    }
}

public class Test {
  public static void main(String[] args)  {

    try {
       validate (7);
    }
    catch(ArithmeticException e) {
       System.out.println(e.getMessage());
    }
  }
}
```

# Checked/Unchecked Exceptions

Notice that in the last example, there will be no compilation error even if the ***throws ArithmeticException*** is omitted at the method signature.

This is because the exception expected is an ***ArithmeticException***, which belongs to the category of *unchecked* exceptions. The compiler will not check it during compile time. Other examples of *unchecked* exceptions include:

- NullPointerException
- ArrayIndexOutOfBoundsException
- IllegalArgumentException
- NumberFormatException

The other type of exception is *checked* exception which include:

- SQLException
- IOException
- ClassNotFoundException
- InvocationTargetException

# throws

- The throws keyword is used in the method signature.
- It is used when the method has some statements that can lead to exceptions.

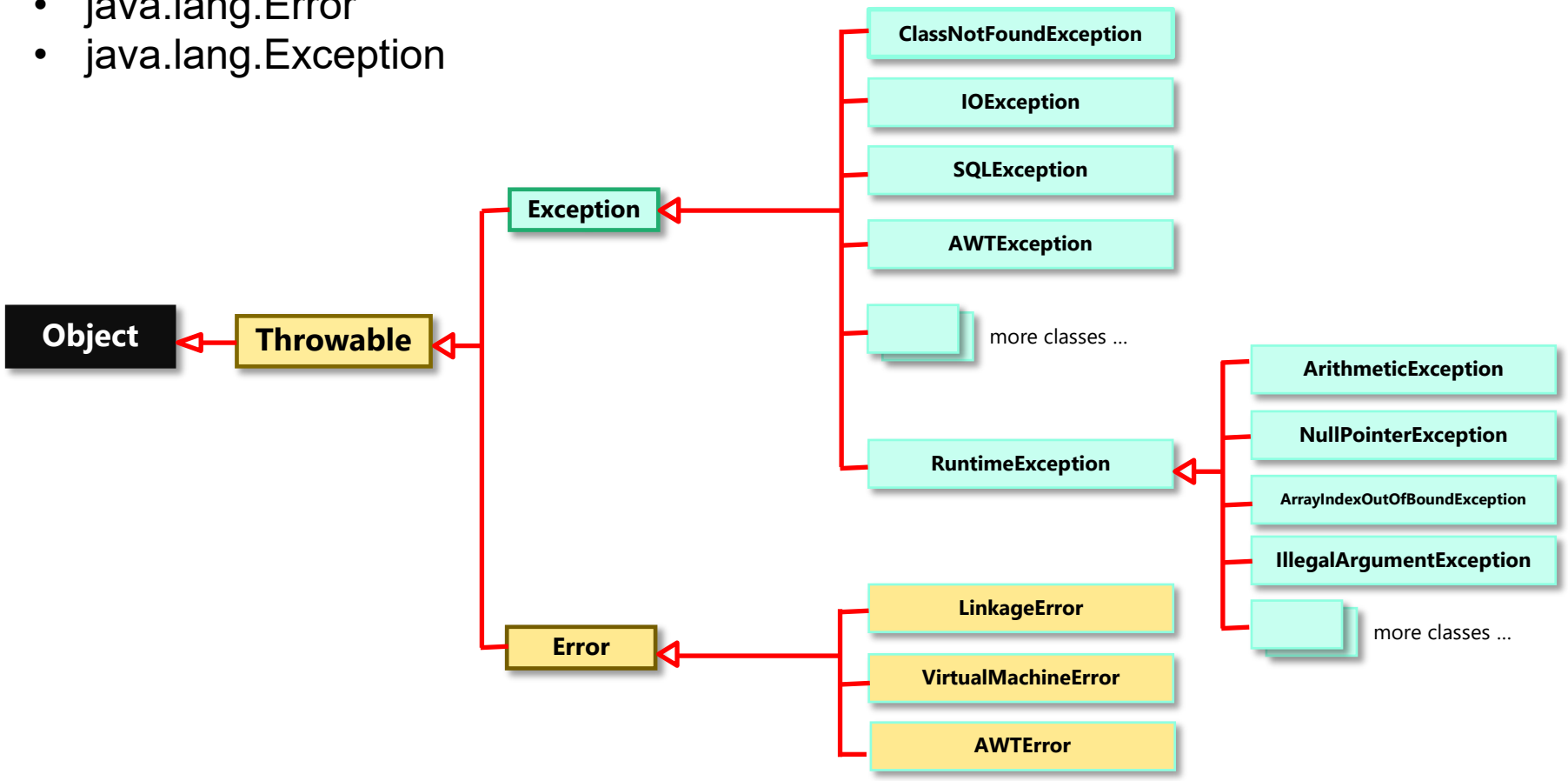
```
public static void writeToFile() throws IOException
{
   BufferedWriter bw = new BufferedWriter(
   new FileWriter("myFile.txt"));
   bw.write("Compiler enforces the throws keyword to be included");
   bw.close();
}

public class Test {
 public static void main(String[] args)  {
   try {  writeToFile();}
    catch (IOException e) {  System.out.println(e.getMessage()); }
  }
}
```

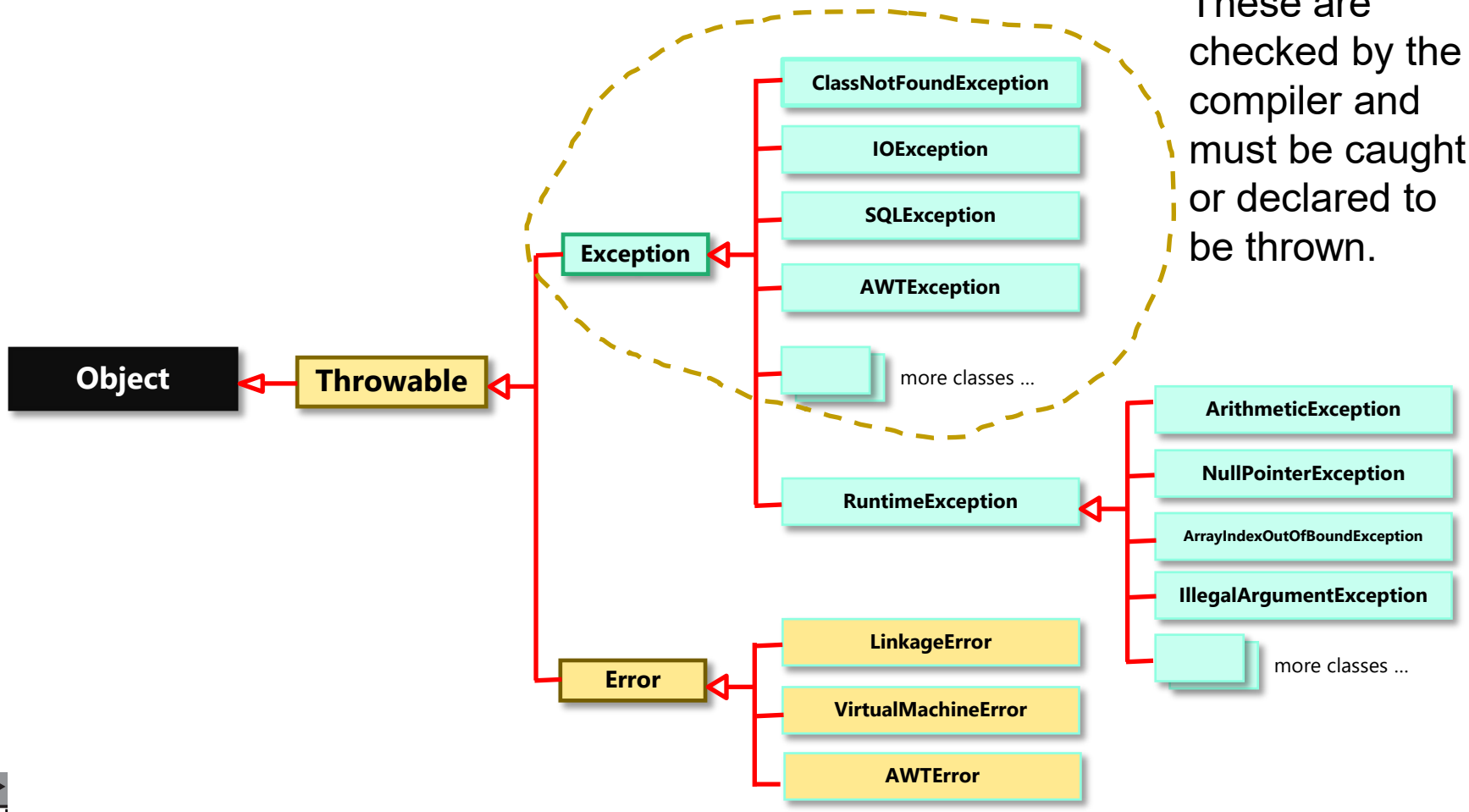
# Hierarchy of the Exception classes

2 subclasses extends from Throwable:

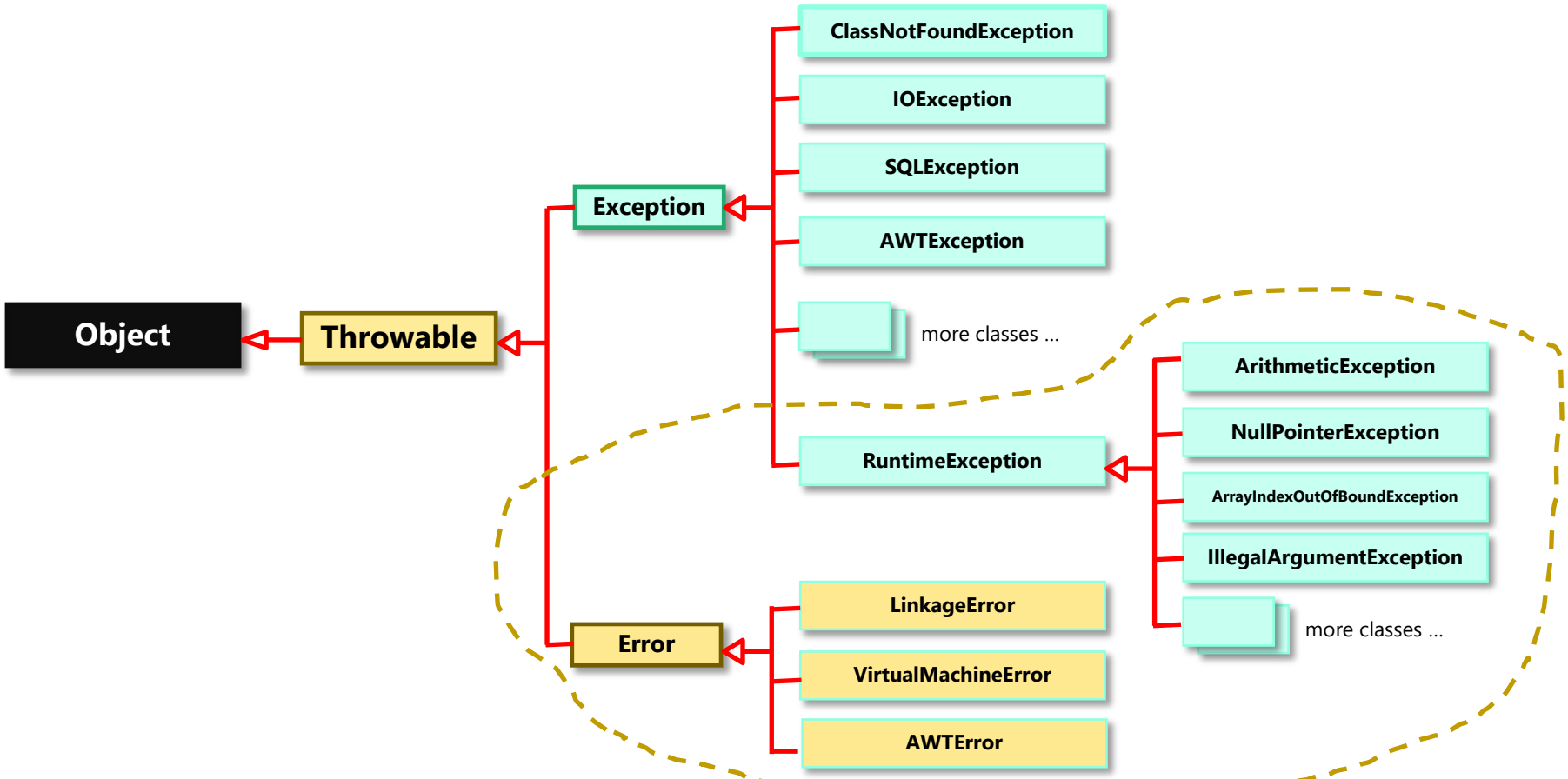
- java.lang.Error
- java.lang.Exception



# Checked Exceptions



# UnChecked Exceptions




Subclasses of Error and Runtime Exception are known as unchecked exceptions

These exceptions are not checked by the compiler, and hence, need not be caught or declared to be thrown in the program.

Below is the typical try-catch-finally (without reference).

```
public class Test {
 public static void main(String[] args) {
  Scanner scanner = null;
  File myFile = new File("test.txt");
  try {
     scanner = new Scanner(myFile);
     while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
     }
  }
  catch (FileNotFoundException e) {
    System.out.println(e.getMessage());   }
  finally {  if (scanner != null) {   scanner.close();   }  }
 }
}
```

# Try-with-references

Below is the try-with reference version.

```
public class Test {
 public static void main(String[] args) {
  File myFile = new File("test.txt");

    try (Scanner scanner= new Scanner(myFile) {
      while (scanner.hasNext()) {
         System.out.println(scanner.nextLine());
      }
    }

    catch (FileNotFoundException e) {
      System.out.println(e.getMessage());
    }
 }
}
```

# Interface

- Like a class, an *interface* can have methods and data
- Unlike a class, it can have only constant data but not variables.
- Methods declared in an interface are usually *abstract* (i.e. only method signature, no logic or no implementation. A method without an implementation is called an *abstract* method).
- It can also have *static/default* methods
- It can extends from another interface
- It cannot be instantiated
- Used across unrelated classes (“capable of -doing” relations)
- Can be used as the return type for methods as long as the object returned implement the interface

# Define, Use and Test Interface

```
public interface Investment {
   double getInterest() ;          // No details of method here
                                   // NOT even empty body{};
                                   // It is to be implemented in
                                   // class(es) that inherit
                                   //  (implements) the interface
}
```

**Define interface**

```
public class SavingAccount implements Investment {
   double balance;
   double iRate;
    public double getInterest() { // Details of method
      return ( balance*iRate);
    }
}
```

**implements interface**

```
public class Test {
   public static void main(String[] args) {
     SavingAccount a1 = new SavingAccount();
     a1.balance = 1000;
     a1.iRate = 0.005;
     System.out.println ( a1.getInterest() );
   }
}
```

**Test interface**

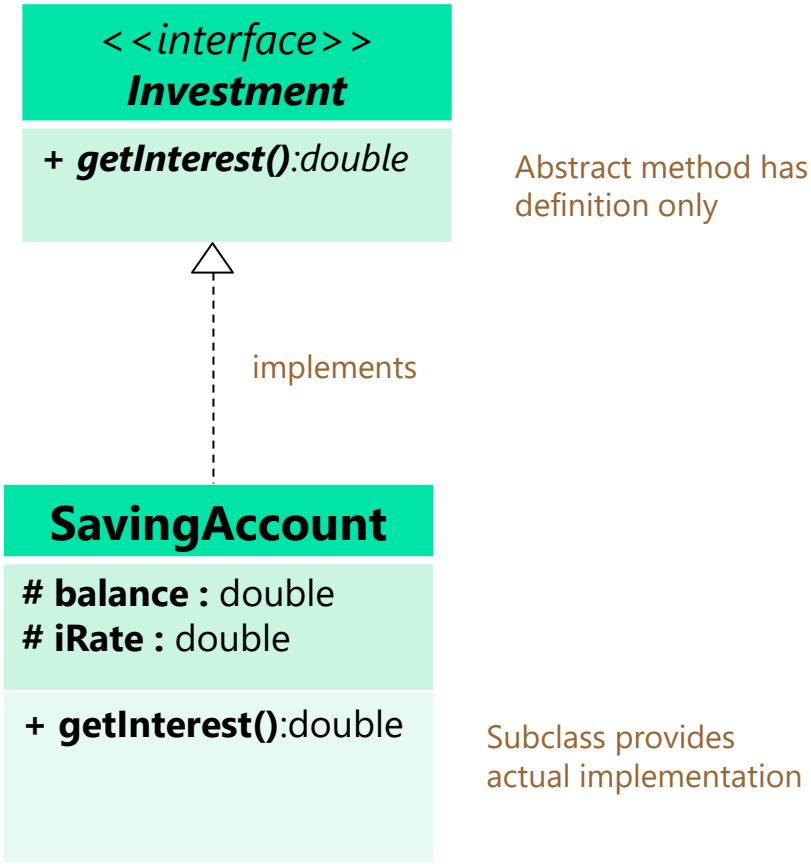
Output:

```
5.0
```

# UML

An interface is indicated in italics in UML Notation.

The implementation of interface is marked by a dash-arrow leading from the subclasses to the interface.



# Interface

- Once a class has *implements* an *interface*, the class must provide the implementation logic for all the abstract methods specified in the interface. Otherwise, there will be error.

```
public interface Investment {
   double getInterest() ;
}

public class SavingAccount implements Investement {
   double balance;
   double iRate;

}
```

**Error!**

class ***SavingAccount*** must provide the logic for ***getInterest( )***

- In a way, *interface* enforces "*consistency*" in software development

# Investment not restricted to banking

Class Property represents the return of investment in a property.

```
public interface Investment {
   double getInterest() ;
}

public class Property implements Investment {
   double cost;
   double rentalReturn;
   double tax;


   public double getInterest() {
      return ( cost*rentalReturn - tax);
   }

}
```

**Implement the abstract method**

# Interface with default method

```
public interface Investment {
   double getInterest() ;  // abstract method

   default void display() {
     System.out.println("This is an investment.");
   }

}
```

Classes implementing the interface can override the default method.

# Implements Multiple Interfaces

Unlike a class, an interface allows *multiple* inheritance (a class can only *extends* from 1 parent class)

```
interface IncomeGain {   double getInterest() ; }
interface CapitalGain { double getProfit() ; }

class SavingAccount  implements  IncomeGain, CapitalGain  {

   double balance, investAmount;
   double iRate, ivProfit;
   public double getInterest() {
      return ( (balance- investAmount)*iRate );
   }
   public double getProfit() {
      return ( ivProfit );
   }
}
```

# Interface extends another Interface

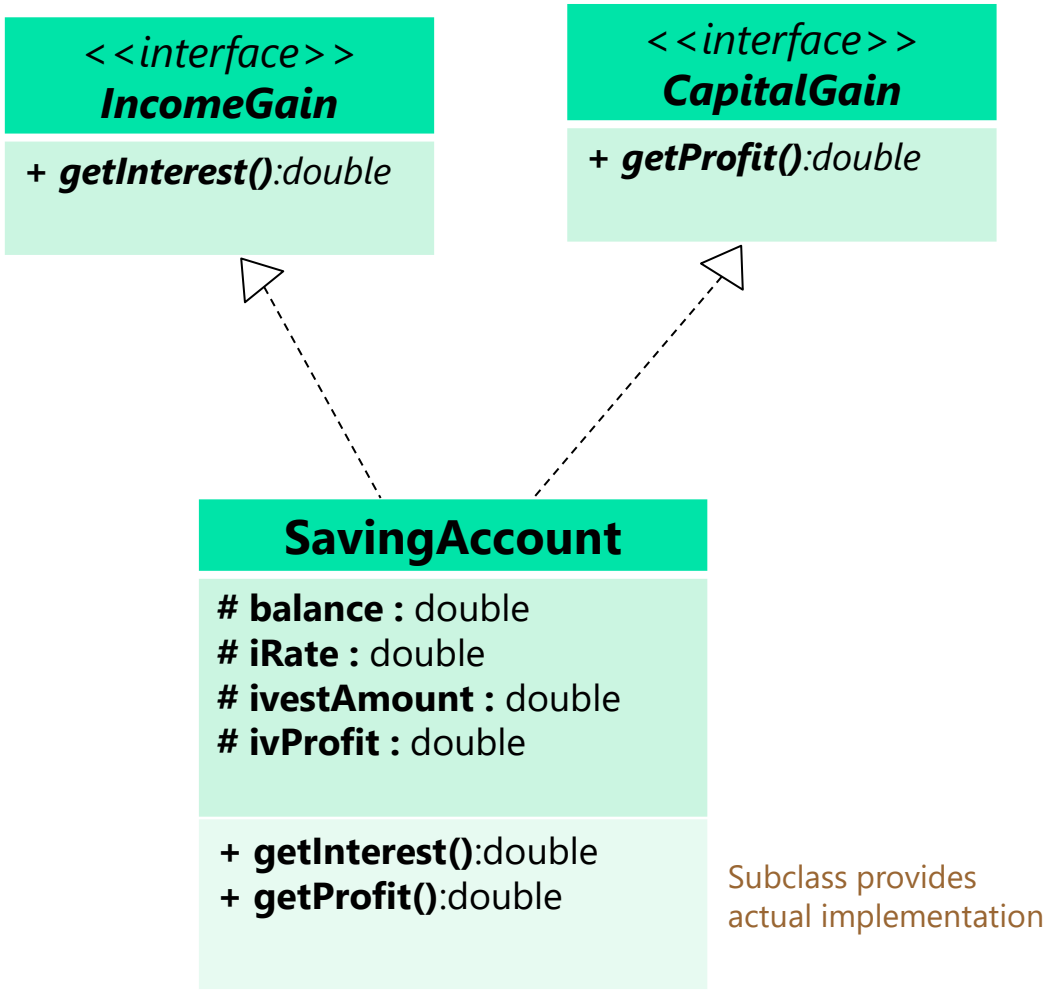
An interface can extend other interfaces, just as a class subclass or extend another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces.

```
interface Edible {
    void serve();
}

interface Soluble {
    void dissolve();
}

interface Seasoning extends Edible, Soluble {
    void mix();
    void marinate();
}
```

# Actual Proper UML



# Abstract Class

- Just like *normal class* but *abstract* class contains *at least* 1 abstract method
- *Abstract* class can be extended using keyword "*extends*" (*Interface* can be implemented using keyword "implements")
- *Abstract* class can provide both abstract and non-abstract methods
- An *Abstract* class can contain no abstract method
- However, a class with 1 or more abstract methods must be defined as an *Abstract* class
- *Abstract* class can have final, non-final, static and non-static variables. (*Interface* has only static and final variables)
- *Abstract* class and *interface* both cannot be instantiated
- Used in related classes ("is-a" relations)

# Abstract class

```
public  abstract class BankAccount {
  protected String acctNum;
  protected double balance = 0, iRate;

  public double getInterest();

  protected BankAccount(String a, double b) {
     acctNum = a;
     balance = b;
  }
}
```

**Abstract method**

# Extends from abstract class

```
public class SavingAccount extends BankAccount {
   SavingAccount(String a, double b, double iRate ) {
      super (a,b);
      this.iRate = iRate;
   }
   public double getInterest() {
     return ( balance*iRate);
   }

}
```

```
public class currentAccount extends BankAccount {
   CurrentAccount(String a, double b) {
      super (a,b);
   }

   public double getInterest() {
      return ( 0.0);
   }
}
```

# UML

An abstract class and its abstract methods are indicated in italics in UML Notation. A solid-arrow leading from the subclasses to the abstract class.