JAVA @11

Object as Method Argument and Return type

Objective

After completing this session you will be able to understand,

- Passing Object as method argument
- Returning Object from the method

A Problem Developer Faces

Scenario: Assume you are developer and you need to create a method which accept the user details which includes the Name , Address ,Age ,Employment details and several other details. The method should calculate and return the tax rate and tax amount based on the user's age and salary and store the employee and tax details in database. The method should return user name and the tax.

Question: How can you pass all the details into the method as an argument ?

Answer: We can pass it as arguments to the method like

Method(String name ,String address ,int age………………….)

Disadvantages:

•This makes the method complex as the number of arguments increases.

•Any new user details addition will result in a change in the method name.

•We cannot return more than one value from a method.

What is the solution?

Solution is passing objects as method arguments and returning objects as return value

Types of methods argument

- **Passing Primitive Data Type :** Passing the primitive type data as method argument. **Example:** int, double, float etc.
- **Passing objects are argument :** In this case we pass objects as arguments to the method. **Example:** UserVO, ArrayList, Map

In this session we are going to learn on how to pass objects to methods?

Passing objects as method argument

- Objects are passed as arguments to the method
- Can make change **only** to the state of the object
- Doesn’t make any change to the reference variable hence we say that this mechanism is not exactly same as “**Pass-By-Reference**” in C programming.
- Since the state changes are done in objects some people argue this indirectly achieves “**Pass by reference** ”.
- Can be used with constructors also.

Next slide explains how the state of objects changes when passed as arguments.

Passing Objects How it works?

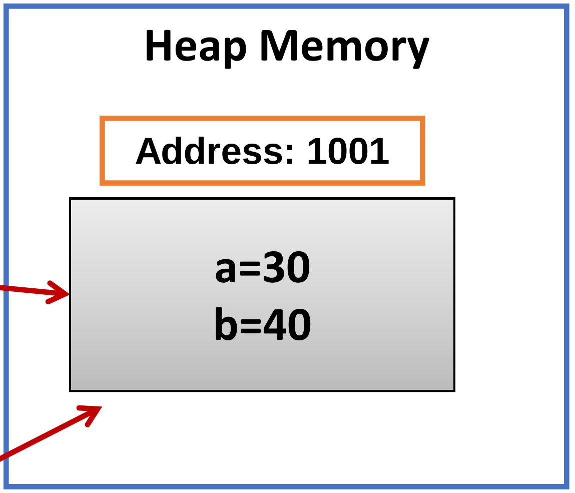
Consider a class named MyClass with two fields a and b.

```
MyClass obj=new MyClass();
aMethod(obj);
```

Object Created

The **obj** contains the object memory address **1001** as value which is the address of the newly created object.

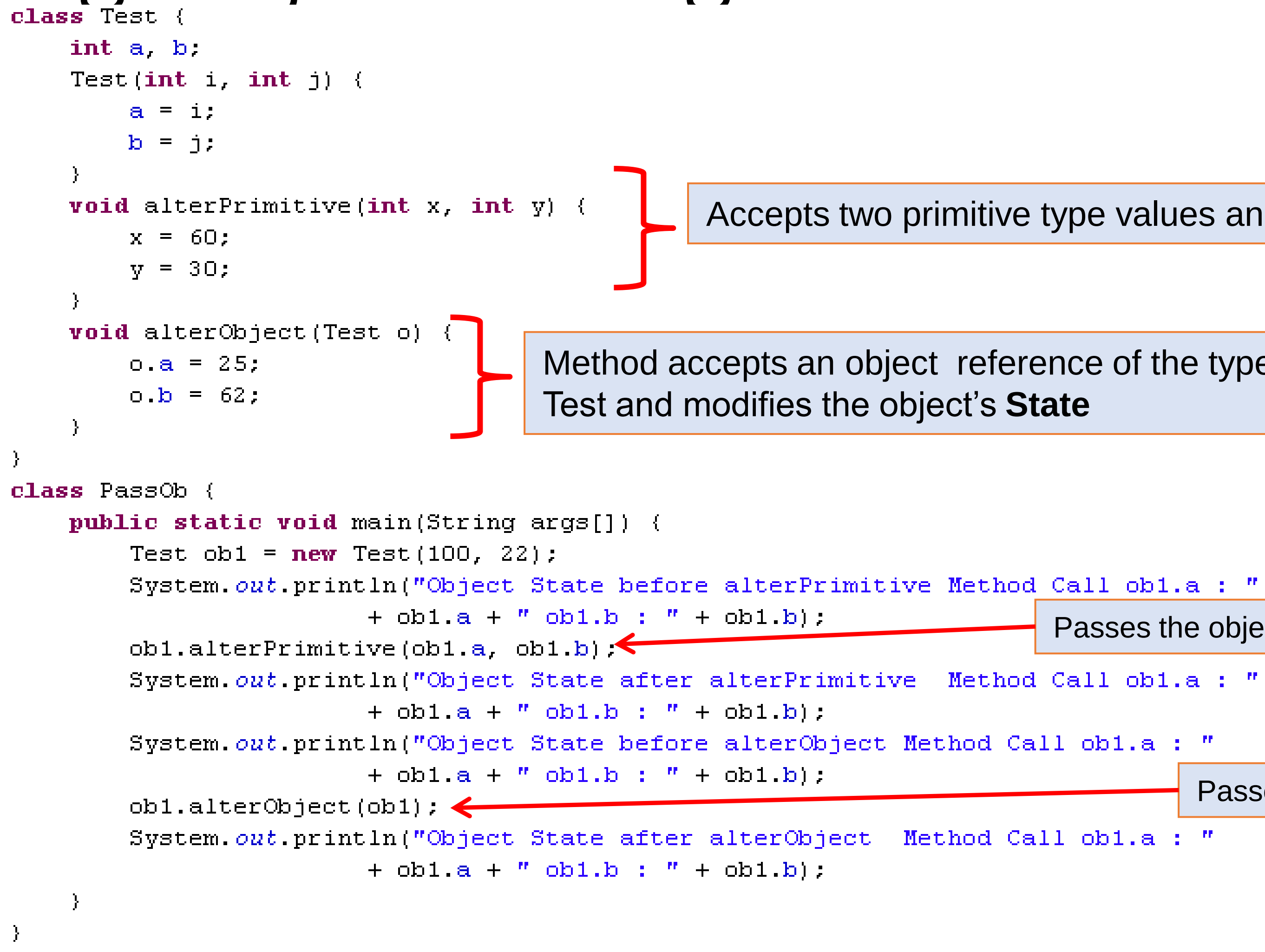
```
void aMethod(MyClass obj1)
{
 obj1.a=30;
obj1.b=40;
}
```



When the method **aMethod** is invoked by passing **obj** as argument. Now **obj1** in the invoked method contains the memory address value **1001** .

Since **obj1** contains the address **1001** , any changes in the state of this will also change the state of **obj** in the calling method.
So changes done to the obj1 state a=30 and b=40 will also change the state of **obj.**

Passing Object as Args.



```
class Test {
    int a, b;
    Test(int i, int j) {
        a = i;
        b = j;
    }
    void alterPrimitive(int x, int y) {
        x = 60;
        y = 30;
    }
    void alterObject(Test o) {
        o.a = 25;
        o.b = 62;
    }
}
class PassOb {
    public static void main(String args[]) {
        Test ob1 = new Test(100, 22);
        System.out.println("Object State before alterPrimitive Method Call ob1.a : "
                        + ob1.a + " ob1.b : " + ob1.b);
        ob1.alterPrimitive(ob1.a, ob1.b);
        System.out.println("Object State after alterPrimitive  Method Call ob1.a : "
                        + ob1.a + " ob1.b : " + ob1.b);
        System.out.println("Object State before alterObject Method Call ob1.a : "
                        + ob1.a + " ob1.b : " + ob1.b);
        ob1.alterObject(ob1);
        System.out.println("Object State after alterObject  Method Call ob1.a : "
                        + ob1.a + " ob1.b : " + ob1.b);
    }
}
```

Accepts two primitive type values and tries to modify it.

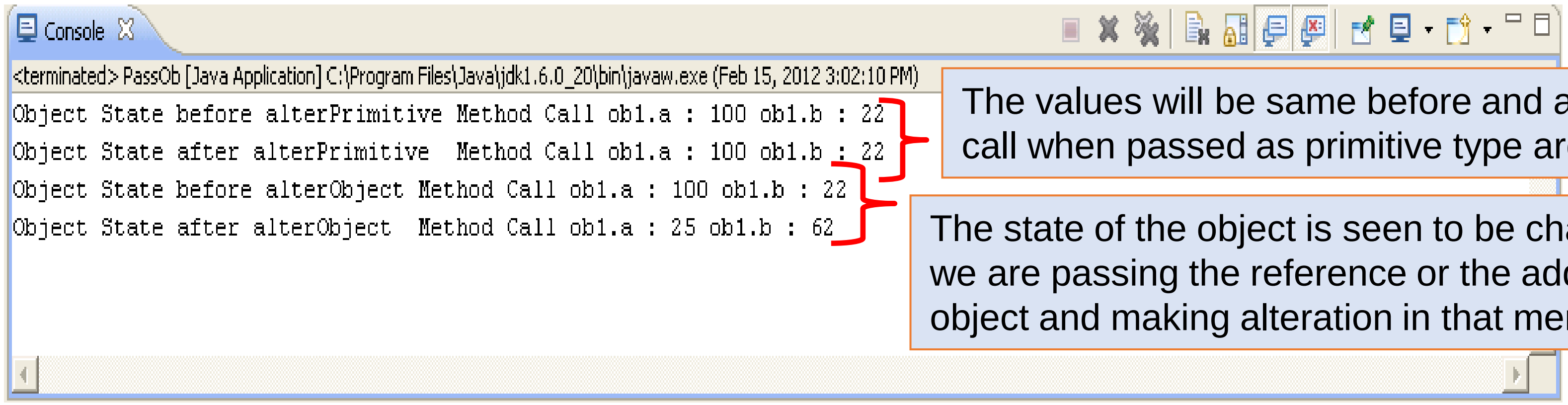
Method accepts an object reference of the type Test and modifies the object's **State**

Passes the object members as primitive data

Passes the object as an argument

We will see how object passing works with help of the following code.

What is the Output?



```
<terminated> PassOb [Java Application] C:\Program Files\Java\jdk1.6.0_20\bin\javaw.exe (Feb 15, 2012 3:02:10 PM)
Object State before alterPrimitive Method Call ob1.a : 100 ob1.b : 22
Object State after alterPrimitive  Method Call ob1.a : 100 ob1.b : 22
Object State before alterObject Method Call ob1.a : 100 ob1.b : 22
Object State after alterObject  Method Call ob1.a : 25 ob1.b : 62
```

The values will be same before and after the method call when passed as primitive type argument.

The state of the object is seen to be changed because we are passing the reference or the address of the object and making alteration in that memory area.

Returning Objects from a Method

- Instead of returning primitive data type from a method, we can return objects from a method.
- Works similar to methods with primitive return type.
- The return type should be either of class type or any of it’s super classes/implemented interfaces.
- The methods can also return Collection objects like ArrayList, Map etc which you will learn in the later sessions.

Advantage:

If you need to return multiple values the values can be defined in a value object and returned.

Object as Return type

```
public class Student {
String studentID;
int mark1;
int mark2;
}
```

```
public class Result {
String studentID;
String grade;
}
```

```
public class ResultCalculator {
    public Result calculateResult(Student student) {
        int total = student.mark1 + student.mark2;
        Result result = new Result();
        result.studentID = student.studentID;
        if ((total / 2) < 60) {
            result.grade = "Fail";
        } else {
            result.grade = "pass";
        }
        return result;
    }
}
```

Note the methods return type is Result Object

Method accepts a Student object , calculates the result based on the marks , loads the result into a Result object and returns it.

Object as Args & Return type

Objective: In this lend a hand we will familiarize how objects can be passed as an argument to a method and how objects can be returned by methods.

Scenario : We are asked to develop a method which accepts the details of an student, which includes the marks of three subjects. The method should calculate the total mark and should rate the student pass/fail based on the average marks

if average >40 { Pass }

else { Fail }

Components

1. Student Class : To hold the student details
2. Result Class : To hold the student results
3. ResultCalculator class : Contains method to calculate the total mark and result of the student
4. MainClass class: Drives the application

Student.java

```
public class Student {
    private String rollNo;
    private int mark1;
    private int mark2;
    private int mark3;
    public String getRollNo() {
        return rollNo;
    }
    public void setRollNo(String rollNo) {
        this.rollNo = rollNo;
    }
    public int getMark1() {
        return mark1;
    }
    public void setMark1(int mark1) {
        this.mark1 = mark1;
    }
    public int getMark2() {
        return mark2;
    }
    public void setMark2(int mark2) {
        this.mark2 = mark2;
    }
    public int getMark3() {
        return mark3;
    }
    public void setMark3(int mark3) {
        this.mark3 = mark3;
    }
}
```

Add the Accessor methods:

The methods starting with get and set are called ***accessor*** methods used for accessing the private members of the class.

The method starting with "***set***" is known as the setter used for setting the argument value to the member variable .

The method starting with "***get***" is known as the getter used for retrieving the value of the variable.

Result.java

```java
public class Result {
    private String rollNo;
    private String grade;

    public String getGrade() {
        return grade;
    }

    public void setGrade(String grade) {
        this.grade = grade;
    }

    public void setRollNo(String rollNo) {
        this.rollNo = rollNo;
    }

    public String getRollNo() {
        return rollNo;
    }
}
```

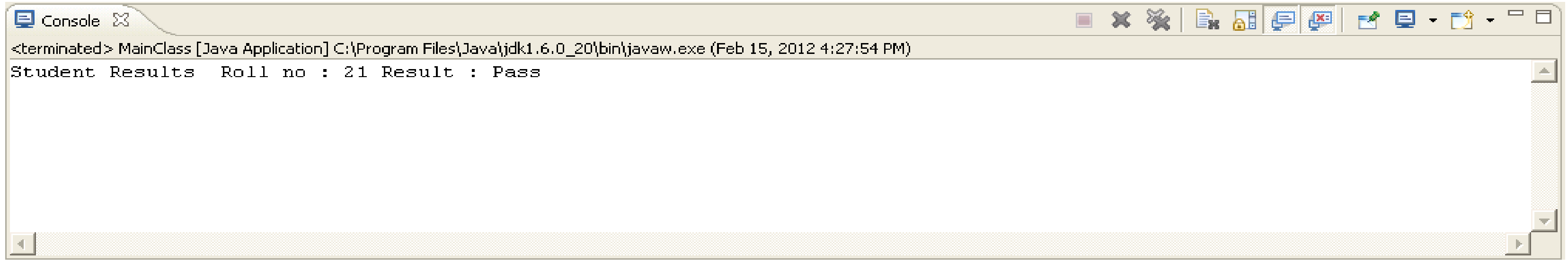
ResultCalculator.java

```
public class ResultCalculator {
    public Result calculateResult(Student student) {
        int total = student.getMark1() + student.getMark2()
                + student.getMark3();
        int average = total / 3;
        Result result = new Result();
        result.setRollNo(student.getRollNo());
        if (average > 40) {
            result.setGrade("Pass");
        } else {
            result.setGrade("Fail");
        }
        return result;
    }
}
```

MainClass.java

```
public class MainClass {
    public static void main(String args[]) {
        Student student = new Student();
        student.setRollNo("21");
        student.setMark1(65);
        student.setMark2(85);
        student.setMark3(40);
        ResultCalculator calculator = new ResultCalculator();
        Result result=calculator.calculateResult(student);
        System.out.println("Student Results  Roll no : "+result.getRollNo()+
                 " Result : " + result.getGrade());
    }
}
```

Output



```
<terminated> MainClass [Java Application] C:\Program Files\Java\jdk1.6.0_20\bin\javaw.exe (Feb 15, 2012 4:27:54 PM)
Student Results  Roll no : 21 Result : Pass
```

Thank you

You have successfully completed

Object as Args & Return type

JAVA @11

Inheritance

Objective

After completing this session you will be able to understand,

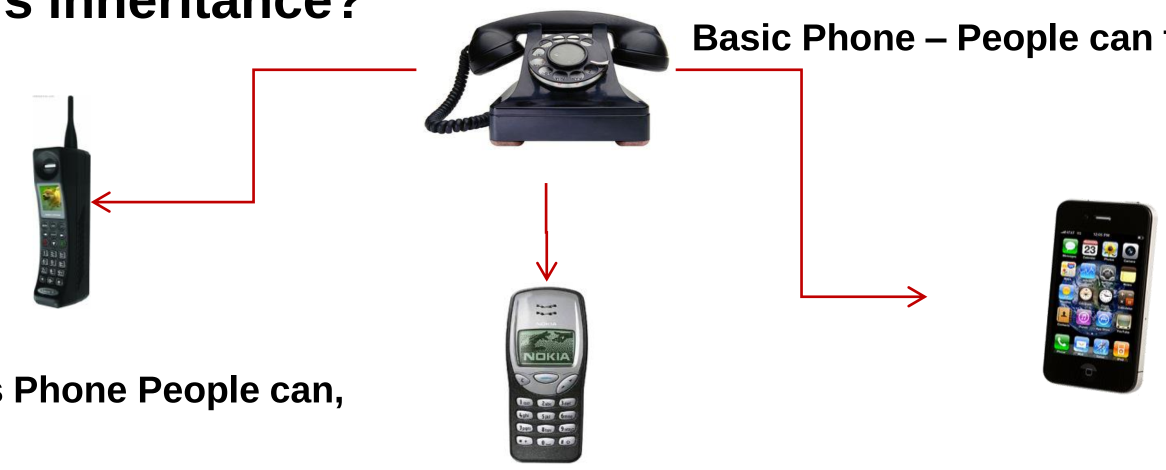
- Define Inheritance
- Super class and Sub class
- Super constructor
- Method overriding
- Run Time Polymorphism

Lets look at some model of phones that evolved in the past two decades. All the phones can be used to call friends(or anyone) and receive calls from them.
So how do they differ?

Inheritance

What is inheritance?

Basic Phone – People can talk



Cordless Phone People can,
- **Talk**

Mobile Phone People can,
- **Talk**
- **Send SMS**

Smart Phone People can,
- **Talk**
- **Send SMS**
- **Browse the Web**
- **Take Photos/Videos**

Each of them have INHERITED the common functionality ***talking*** from the ancestral basic phone and have some add-on features (SMS, Web browsing) of their own.
This is ***Inheritance***.

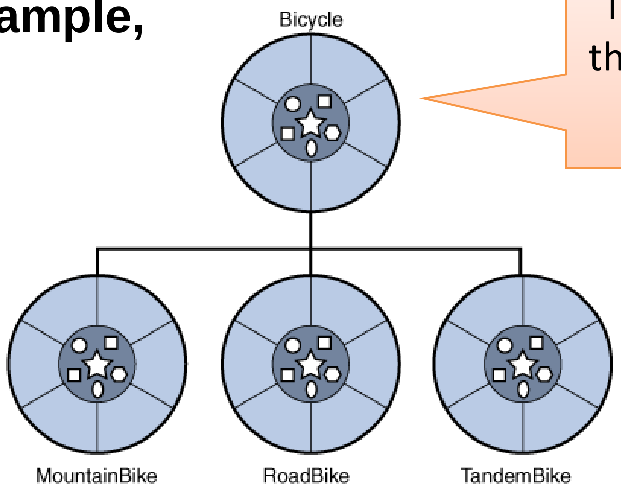
Inheritance

What is inheritance?

Inheritance is the concept of a **child** class (***sub class***) automatically **inheriting** the variables and methods defined in a **parent** class (***super class***).

It is one of the primary features of **O**bject **O**riented **P**rogramming

Here is another example,



The super Class bicycle will hold all the **common** characteristics namely apply brake, pedalling

Mountain bike has additional characteristics thick tyre.

Tandem bike has 2 seats & 2 handlebars.

Road bike has shock absorbers.

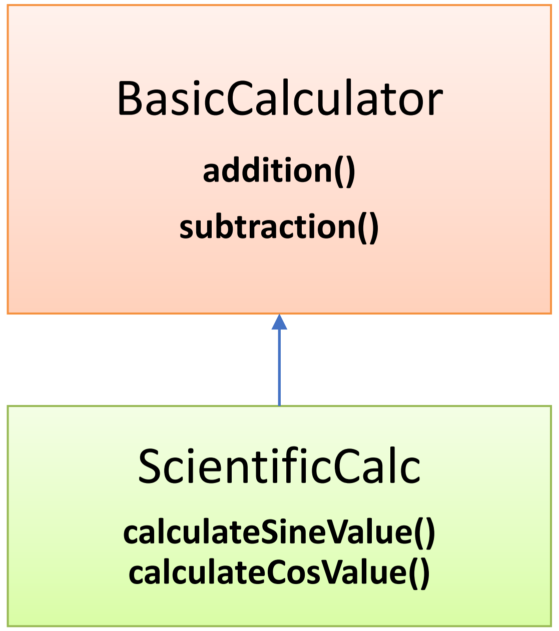
Benefits of Inheritance

Benefits of Inheritance:

The primary benefit of inheritance is ***reusability***.

Once a behavior is defined in a super class, that behavior is automatically inherited by all its subclasses and reused. So developers need not redevelop the logic again.

Example:



The ***ScientificCalc*** can calculate the sine and cos value and also perform addition and subtraction methods that it has inherited from ***BasicCalculator***

Implementation

How to derive a subclass from parent class?

To derive a sub class, you use the ***extends*** keyword.

Assume you have a parent class called ***BasicCalculator***

```
public class BasicCalculator {

    public int val1;
    public int val2;

    public int addition(){
        int sum = val1 + val2;
        return sum;
    }
    public int subtraction(){
        int diff = val1 - val2;
        return diff;
    }
}
```

Now you have to create another class called ***ScientificCalc*** which extends ***BasicCalculator*** so that you can inherit all methods of ***BasicCalculator***

```
public class ScientificCalc extends BasicCalculator {

    public double calculateSineValue(){
        double sineValue = Math.sin(val1);
        return sineValue;
    }
    public double calculateCosValue(){
        double cosValue = Math.cos(val1);
        return cosValue;
    }
}
```

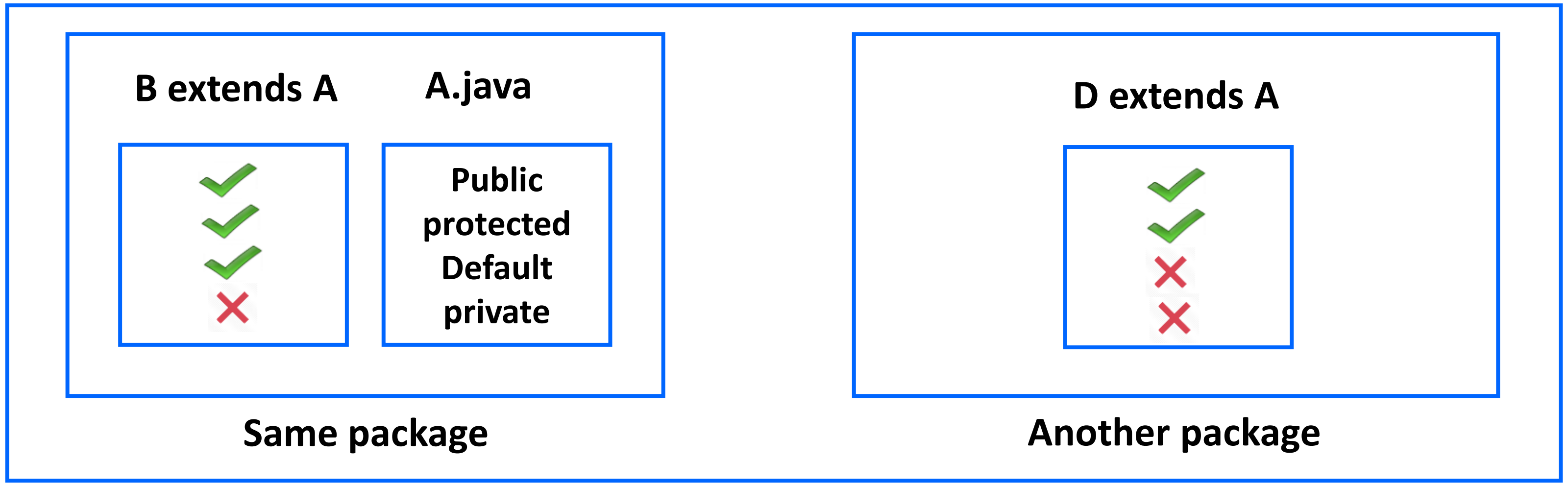
Features of Subclass

Features of the subclass:

▪ A subclass inherits all of the "**public**" and "**protected**" members of its parent, no matter what package the subclass is in.

▪ If the subclass is in the **same package**, then it also inherits the "**default**" access (members that do not have any access modifier) members of the parent

Assume A.java is the parent Class, B is a subclass present in same package and D is a subclass in a different package



Inheriting fields in Subclass

- The inherited fields can be accessed **directly**, just like any other fields again based on the access modifiers.
- You can declare **new** fields in the subclass that are **not** in the **super** class.
- You can declare a **field** in the subclass with the **same name** as the one in the super class, thus hiding the parent fields (not recommended).
- A subclass does **not** inherit the **private members** of its parent class. However, if the super class has public or protected methods for accessing the private fields, these can also be accessed by the subclass.

Inheriting methods in a Subclass

- The inherited methods can be accessed **directly** just like any other methods in a class again based on the access modifiers.

- You can write a **new** instance of the **method** in the subclass that has the same signature as the one in the super class, thus **overriding** it.

- You can write a **new static** method in the subclass that has the same signature as the one in the super class, thus hiding it. Here, the super class method should also be static.

- You can declare **new methods** in the subclass that are **not** in the super class.

java.lang.Object

Object class is mother of all classes, residing in “*java.lang*” package.

- In Java language, all classes are subclass (extended) of the Object super class.
- Object class does not have a parent class.
- Object class implements behavior common to all classes including the ones that you write.

Following are some of the important methods of ***Object*** class,

- **getClass()** – Returns the runtime class of an Object.
- **equals()** – Compares two objects to check if they are the equal.
- **toString()** – Returns a String representation of the object.

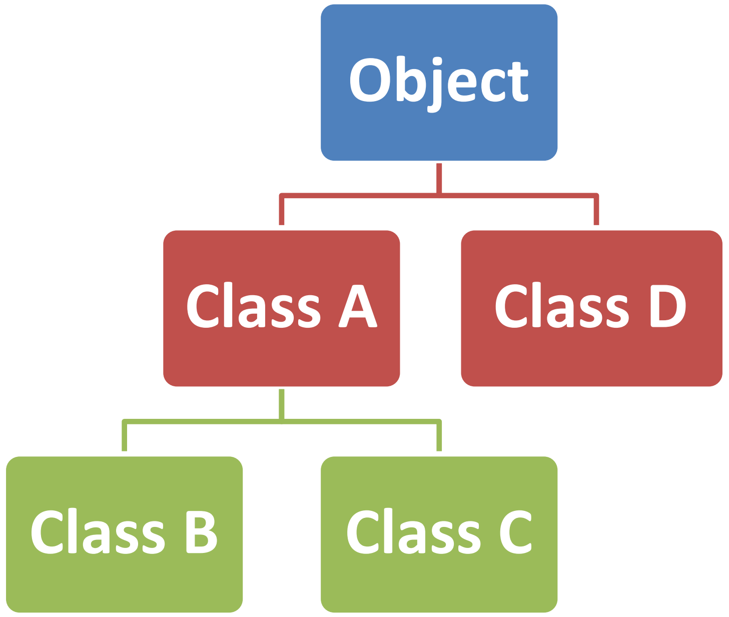
Super class & Sub class

What is a super class?

Any class **preceding** a specific class in the class hierarchy is the super class. It is also called as **parent** class.

What is a sub class?

Any class **following** a specific class in the class hierarchy.



Class A is the super class for Class B and Class C

Class B and Class C are the subclasses of Class A

Object Class is the super class for all classes

Super - keyword

When is the super keyword used?

• The super keyword is used to access the variables and methods of super class

• If your method overrides one of it’s super class methods, you can invoke the overridden method through the keyword super

Example:

```
public class SuperClass {

    public void printMethod() {
        System.out.println("Superclass.");
    }
}
```

```
public class SubClass extends SuperClass {

    // overrides printMethod in Superclass
    public void printMethod() {
        super.printMethod();
        System.out.println("Printed in Subclass");
    }
    public static void main(String[] args) {
        SubClass s = new SubClass();
        s.printMethod();
    }
}
```

Invokes the super method in the super class

super - method

When is the super constructor used?

• The super constructor call is used to call a constructor of its immediate super class

• Based on the argument passed, the corresponding constructor in the parent class is invoked

Few things to remember:

• The super() call must occur as the **first statement** in the constructor

• The super() call can **only** be used in **constructor** calls and **not method** calls.

Example – super constructor

Example: Create a main method and create an instance of TandemBike class to view output.

```java
 public class Bicycle {

     public int wheels = 2;
     public int currSpeed = 150;
     int gear = 0;

     public Bicycle() {
         System.out.println("Inside Bicycle Constructor");
     }
 }
```

Super Class with default constructor

```java
public class TandemBike extends Bicycle {

    public int seats = 2;
    public int handlebars = 2;

    public TandemBike(){
        super();
        super.gear = 2;
        System.out.println("inside Tandem Bike Constructor"+gear);
    }
}
```

Super keyword to call the constructor in super class

Example - Inheritance

Problem Statement:

1. Create a class ***BankAccount*** with the following methods

• ***depositMoney*** – Prints the depositAmount.

• ***withdrawMoney*** – Prints the withdrawalAmount and calculates balance as mentioned below,

balance = depositAmount – withdrawAmount

•The following instance variables need to be present in BankAccount

withdrawAmount, ***depositAmount***, ***interestRate*** (defaulted to 9.5) and ***balance***.

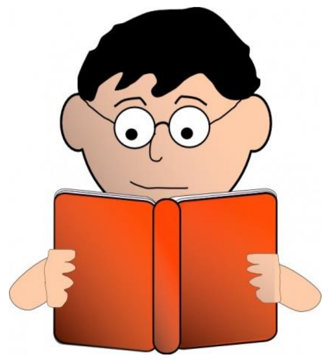
2. Create two subclasses ***NRIAccount*** and ***SeniorCitizen*** account which extends the ***BankAccount*** class

•Both the subclasses should have a method called ***applyFixedDeposit*** which should set the ***interestRate*** variable in the super class to 6.5 for ***NRIAccount*** and 10.5 for ***SeniorCitizen*** account and display the interest rate message.

3. Create a class ***InheritanceDemo*** with main method which creates instance of the NRIAccount and SeniorCitizen Account and invokes the methods ***depositMoney(), withdrawMoney(), applyFixedDeposit()***

Overriding

Father is the super class



Son is the subclass and he inherits the properties of the super class (father)

Father likes to read books. His favorite genre is medical fiction

Son also likes to read books. His favorite genre is modern fantasy comics

Here, the son has inherited the father's behavior of reading books, but with a difference in the type of book. Here the son is overriding the behavior of father.

Method Overriding

What is method overriding?

Creating a method with the same name, arguments and return type in both the parent and the child class is called ***method overriding***.

When do we override methods?

If a derived class (sub class) needs have a **different** method **implementation** from that of a super class, then that method can be overridden.

Example - Method Overriding

Parent Class:

```
public class Person{
        public String getName(){
                System.out.println("Parent:
getName");
                return name;
        }
}
```

getName method in the super class, Person

Now, when you invoke **getName()** method of an object of subclass (**Student)** , output would be,
Student:getName

Child Class:

```
public class Student extends Person{
        public String getName(){

        System.out.println("Student:getName");
                return name;
        }
}
```

getName method is overridden in subclass Student

Run Time Polymorphism

What is polymorphism?

Polymorphism is the capability of a method to do different things based on the object used for invoking the method.

What is run time polymorphism?

Method overriding is an example of runtime polymorphism. Here, the JVM determines the method invocation at the runtime and not at the compile time. The method being invoked is based on the object which on which the method is triggered.

From the previous Example,

```
Person p = new Person();
Person s = new Student();
p.getName()// will print “Parent getName”
s.getName()// will print “Student getName”
```

Based on the object on which the method is invoked the appropriate **getName()** method is called.

Example – Run Time Polymorphism

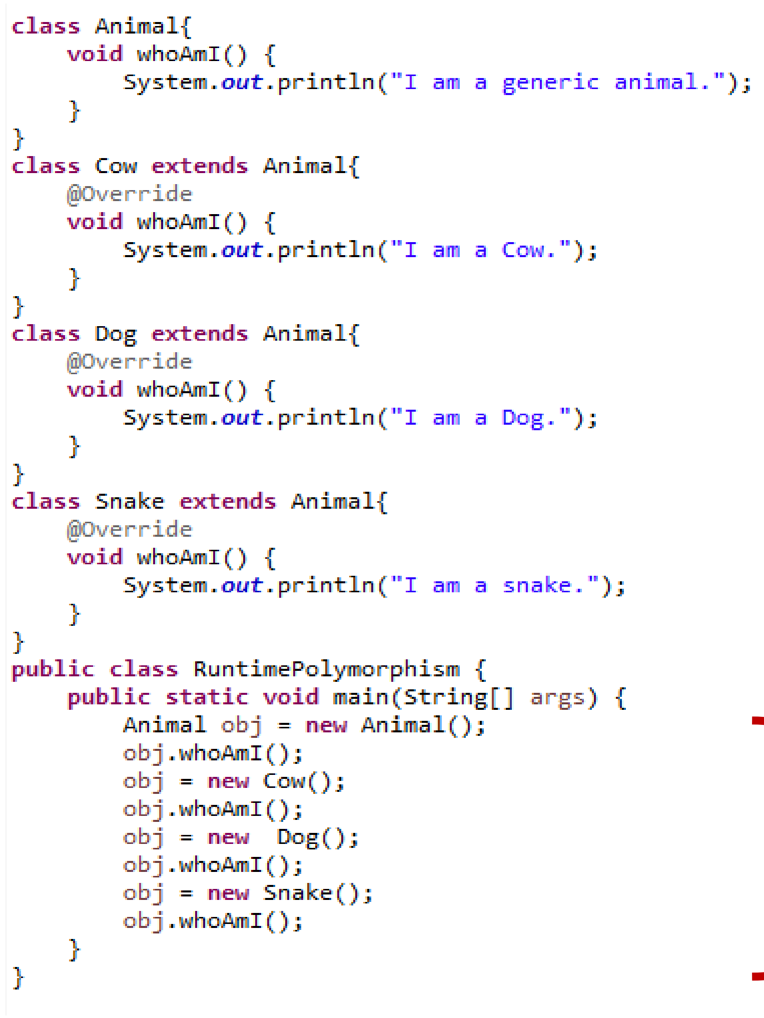
Let us all understand run time polymorphism with an example !!

- Let us create a parent class, *Animal* and three subclasses, *Dog*, *Cow* and *Snake* which extends Animal

- All the classes should have a method, *whoAmI* which prints the appropriate class name as "I am a Dog" (or) "I am a Cow" etc.

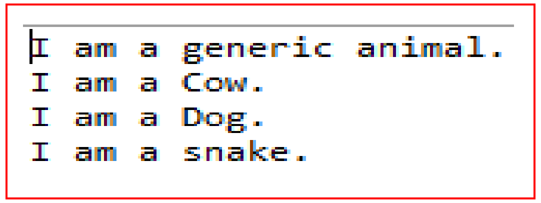
- A new class, *RunTimePolymorphism* is created with a main method which creates an instance of all the classes created above and calls the *whoAmI* method for each instance.

RuntimePolimorphism.java



```java
class Animal{
    void whoAmI() {
        System.out.println("I am a generic animal.");
    }
}
class Cow extends Animal{
    @Override
    void whoAmI() {
        System.out.println("I am a Cow.");
    }
}
class Dog extends Animal{
    @Override
    void whoAmI() {
        System.out.println("I am a Dog.");
    }
}
class Snake extends Animal{
    @Override
    void whoAmI() {
        System.out.println("I am a snake.");
    }
}
public class RuntimePolymorphism {
    public static void main(String[] args) {
        Animal obj = new Animal();
        obj.whoAmI();
        obj = new Cow();
        obj.whoAmI();
        obj = new  Dog();
        obj.whoAmI();
        obj = new Snake();
        obj.whoAmI();
    }
}
```

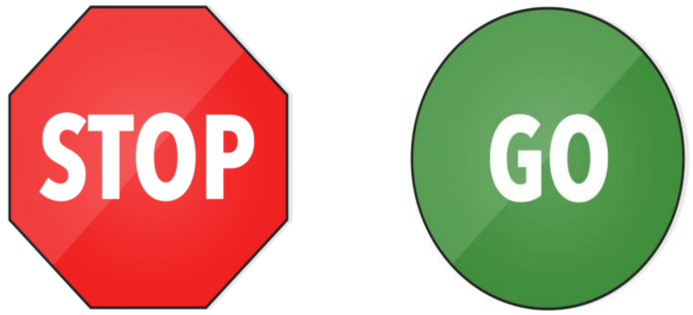
Console Output



```
I am a generic animal.
I am a Cow.
I am a Dog.
I am a snake.
```

There are 1 variable of type Animal. Only obj refers to instance of Animal class, all other refers to instances of subclass of Animal. From the output's result, we can see that the version of the method invoked is based on the actual object type

Time To Reflect



Trainees to reflect the following topics before proceeding.

- What is Inheritance?
- How to create a subclass?
- What is the super class for all classes?
- How to access the fields of the super class?
- What is overriding?
- What is run time polymorphism?

Thank you

You have successfully completed

Inheritance

JAVA @11

Abstract Classes

Objective

After completing this session you will be able to understand,

- Introduction to Abstract Classes
- When to use abstract classes

The word - Abstract

Dictionary Meaning of Abstract:

a. *Conceptual*

b. *Theoretical*

c. *Existing in thought or as an idea but not having a physical or concrete existence.*

Abstract Class

Abstract Class in Java

An abstract class is a class that contains one or more abstract methods.

Example:

```
abstract class Calculator{
    public void display() {
    }
    public abstract void calculateVolume();
}
```

- Abstract class should use the keyword "*abstract*"
- An abstract class can contain non abstract methods also.
- If a class has at least one abstract method, then that class should be declared abstract.

Abstract Method

Abstract Method in Java:

Methods that do not have any implementation(body) are called abstract methods

Example:

Abstract Method:

Public abstract void add();

Concrete Method:

```
public void add(){
    int x = a + b;
}
```

To *create* an *abstract method*, just write the method declaration without the body and use the keyword "*abstract*"

DO NOT USE CURLY BRACES.

Extending an Abstract class

Assume a parent class as LivingThing

```java
public abstract class LivingThing {

    public abstract void walk();
}
```

Abstract method - walk() in parent class

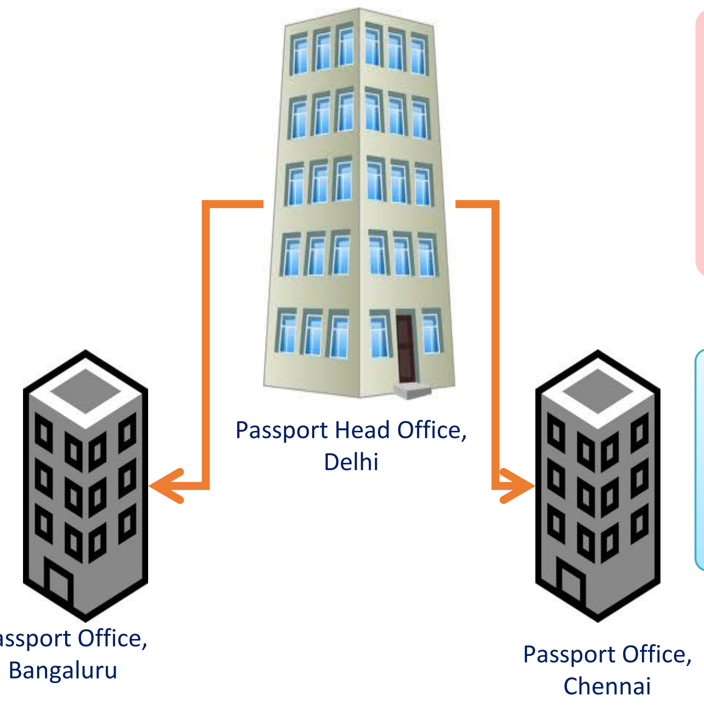
Human subclass of LivingThing

```java
public class Human extends LivingThing {

    @Override
    public void walk() {
        System.out.println("Human walks with two legs");
    }
}
```

Implementation of abstract method - walk() in subclass

Abstract in real world

Scenario: Lets take a passport issuance process, assume that the passport head office is located in ***Delhi*** and corresponding local offices are in ***Chennai/Bangaluru***. The passport issuance is a two step process, ***verification of photo identity*** and ***final approval***. In Chennai and Bangalore the verification document for photo identity are different. Say in Bangalore they verify using ration card and in Chennai they use drivers license. So the head office at Delhi decides to define a protocol and delegate the photo identity verification process but the final approval will be done by the head office.



Delhi Office,
- Defines a protocol, that all local passport office should verify the photo identity.
- HO verifies and approves the final document.

Bangaluru,
- Adheres to the protocol and performs photo identification using ration card.
- Sends the document to HO for approval.

Chennai,
- Adheres to the protocol and performs photo identification using driver license.
- Sends the document to HO for approval.

Abstract class & Passport office

Passport Head Office	Abstract Class
Local passport Offices	Sub Class
Photo Identity Verification	Protocol: Abstract Methods
Final Approval	Common concrete method in abstract class reused by sub classes.

NOTE: No people can directly reach Delhi head office for passport application rather they should always go via the local offices in appropriate cities. Similarly abstract Class cannot be instantiated only the subclasses can be instantiated and methods in abstract class invoked.

When to use Abstract classes?

When do we use abstract classes and Methods?

- Abstract class (*Passport Head Office*) is one way of dictating a strict protocol for the subclasses (*Local passport offices*) to follow, that is implement the abstract methods (*verify photo identity*) declared in parent class.

NOTE: Defining methods to be implemented (protocol) cannot be done in case of normal class being extended.

- Abstract methods are usually declared where two or more subclasses are expected to fulfill a similar role (abstract methods) in different ways

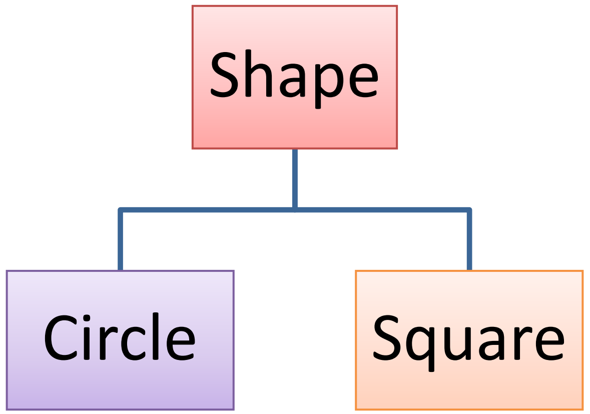
These subclasses extend the same abstract class and provide different implementations for the abstract methods.

- You can use abstract classes to implement common methods (passport final approval) and use the subclasses to implement the abstract methods specific to the sub class.

When to use Abstract classes?

When do we use abstract classes and Methods? – Software Example

Assume Shape is the parent class, Circle and Square are subclasses that extends Shape. All the Shapes should have red color and they should expose a method to calculate area based on their appropriate shape.



Abstract class ***Shape***
- Implements the ***common*** method ***setColor***.
- Declares a abstract method (protocol) ***calculateArea*** for sub class to implement.

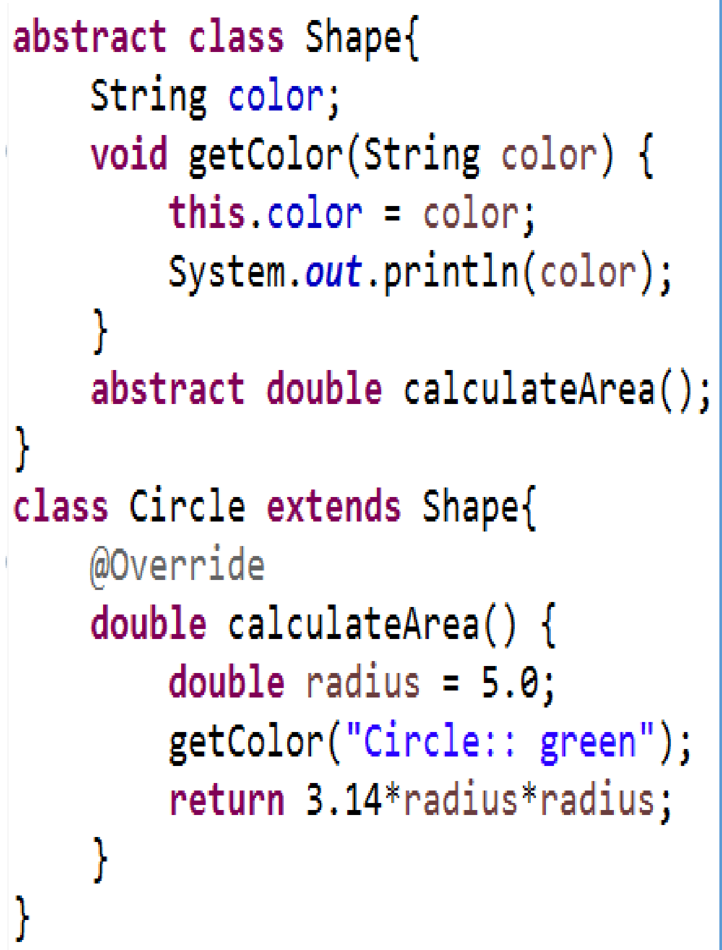
Classes ***Square*** & ***Circle*** extend ***Shape*** and
- Inherit ***setColor*** method.
- Implement abstract method ***calculateArea*** (adhere the protocol).

Example – Abstract classes

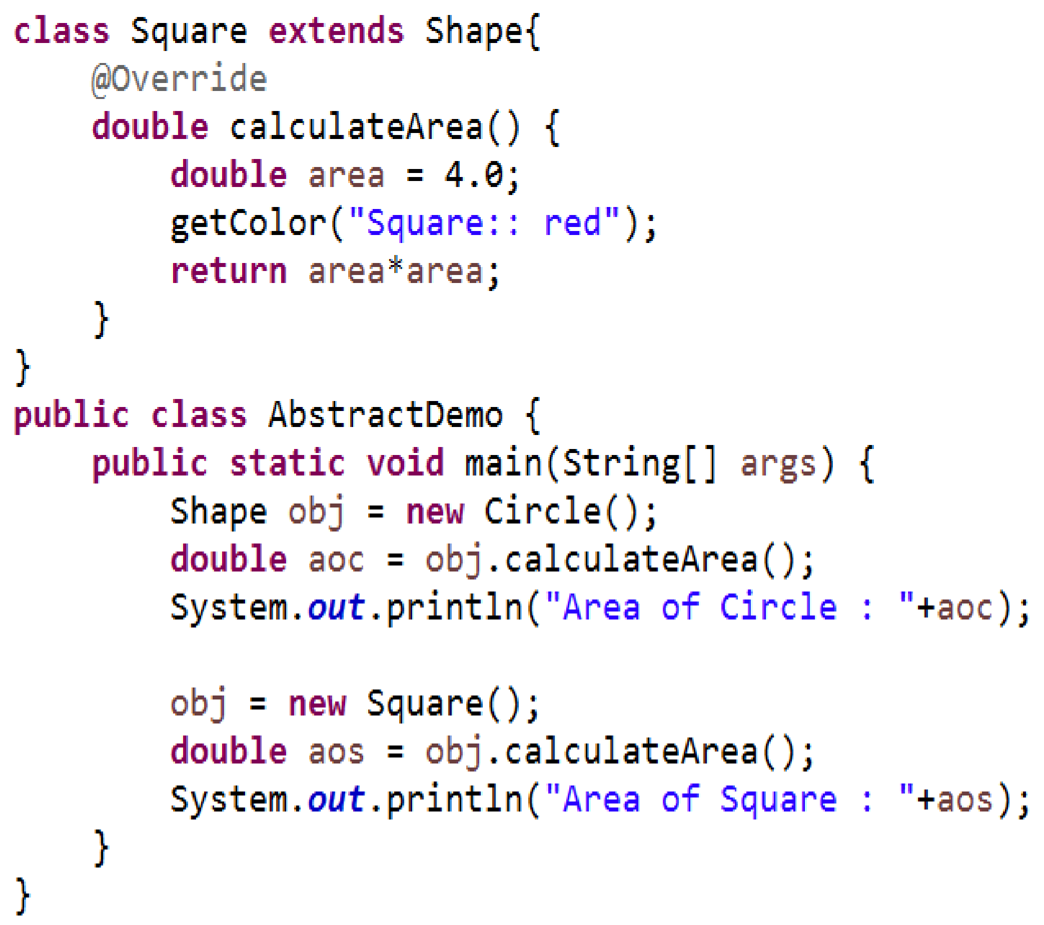
Let us understand more about abstract classes with the below program

1. Create a parent class **Shape.java** with a instance variable color and methods below,
   a. Abstract method **calculateArea**() which returns double value;
   b. Concrete method **setColor** which accepts a String *color* as the parameter and sets the instance variable ‘color’. It should also print the color in the console.
2. Create a sub class **Circle.java** which extends Shape.java and
   a. Implements the ***calculateArea()*** method. It should calculate the area as 3.14*r*r and print the area in the console. Consider radius = 5.0.
3. Create another sub class **Square.java** which extends Shape and
   a. Implements the ***calculateArea()*** method. It should calculate the area as length*breadth and print the area in the console. Consider Length/Breadth = 4.0.
4. Create ***AbstractDemo.java*** class with Main method which invokes the **setColor** method and the **calculateArea**() method on Circle and Square Object

Example - Solution



```java
abstract class Shape{
    String color;
    void getColor(String color) {
        this.color = color;
        System.out.println(color);
    }
    abstract double calculateArea();
}
class Circle extends Shape{
    @Override
    double calculateArea() {
        double radius = 5.0;
        getColor("Circle:: green");
        return 3.14*radius*radius;
    }
}
```



```java
class Square extends Shape{
    @Override
    double calculateArea() {
        double area = 4.0;
        getColor("Square:: red");
        return area*area;
    }
}
public class AbstractDemo {
    public static void main(String[] args) {
        Shape obj = new Circle();
        double aoc = obj.calculateArea();
        System.out.println("Area of Circle : "+aoc);

        obj = new Square();
        double aos = obj.calculateArea();
        System.out.println("Area of Square : "+aos);
    }
}
```

Thank you

You have successfully completed

Abstract Class APIs

JAVA @11

Interfaces

Objective

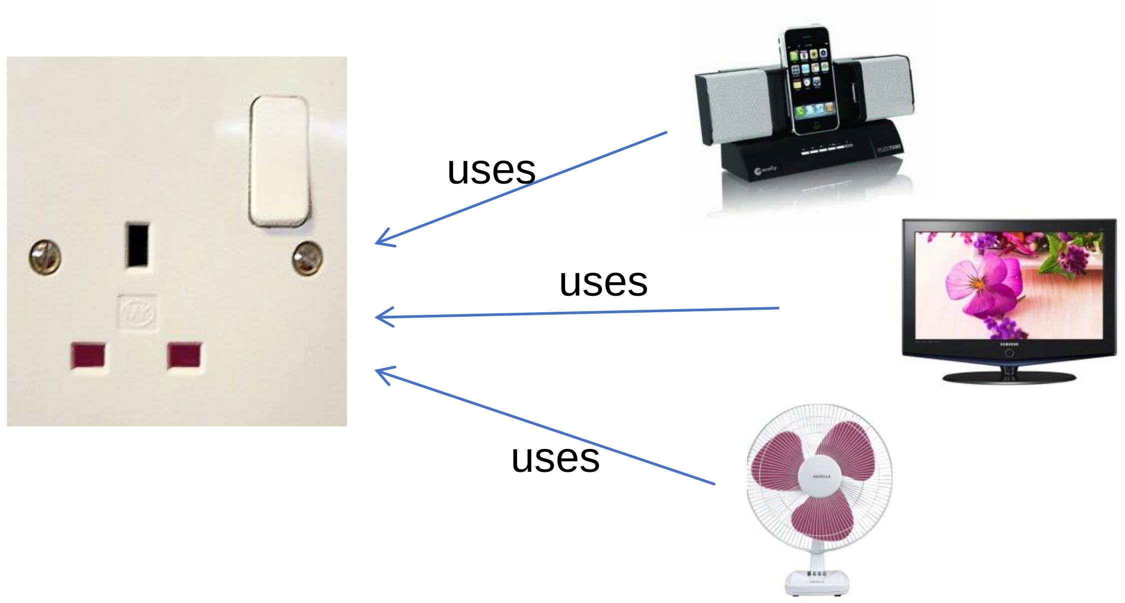
After completing this session you will be able to understand,

- Introduction to Interface
- Implement an Interface
- Why do we use Interfaces
- Interface vs Abstract Classes vs Class
- Interface and Polymorphism
- Java8 Default method in Interface
- Java8 Static method in Interface
- Marker Interface

Interface

What is Interface?

Plug Point is a interface which helps different devices to connect to the electrical line to draw power



So interface is a common protocol that all devices need to adhere for them to operate.

In software terminology the interface is an contract between the class and the outside world, and this contract is enforced at the build time by the compiler

Interface

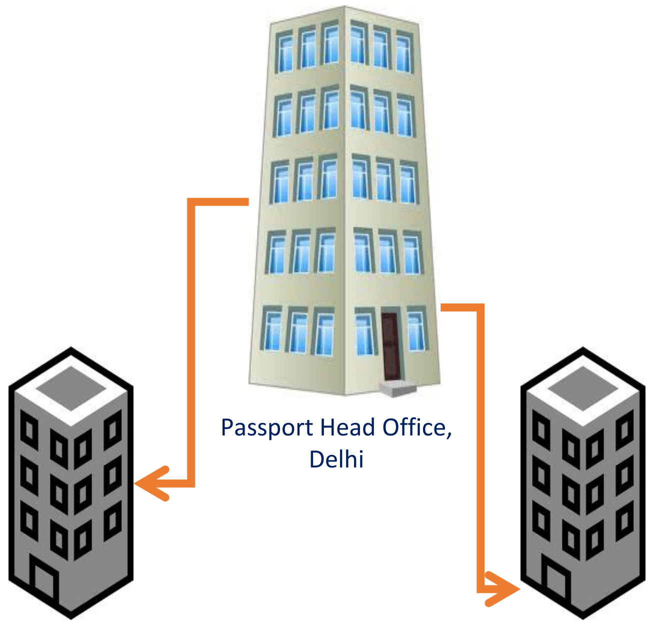
What is Interface?

When **all methods** in a class are ***abstract***, the class can be declared as an Interface.

An Interface defines a contract for a classes to implement the behavior. Similar to the plug point defines the contract for all devices to draw power from electricity lines.

Interface in real world

Scenario: Taking the passport example if in the issuance both the ***verification of photo identity*** and ***final approval*** process needs to be performed by the local offices. The head office at Delhi defines only the protocol/ process for photo identity verification process and approval but the actual process will be performed by the local offices.



Delhi Office,
- Defines a protocol/process that all local passport office should follow for verifying the photo identity & final approval.

Bangaluru,
- Adheres to the protocol and performs photo identification using ration card.
- Approves the document as slated by the HO.

Chennai,
- Adheres to the protocol and performs photo identification using driver license.
- Approves the document as slated by the HO.

Analogy between Interface and Passport office

Passport Head Office	➡	**Interface**
Local passport Offices	➡	**Sub Class**
Photo Identity Verification	➡	**Protocol: interface methods**

NOTE: No people can directly reach Delhi head office for passport application rather they should always go to the local offices in appropriate cities which will follow the process defined by head office to issue passport.

Similarly Interface cannot be instantiated only the classes implementing it can be instantiated and the implemented methods invoked.

Develop an Interface

How to create an Interface?

To create an interface, the 'class' keyword should be replaced with the 'interface' keyword

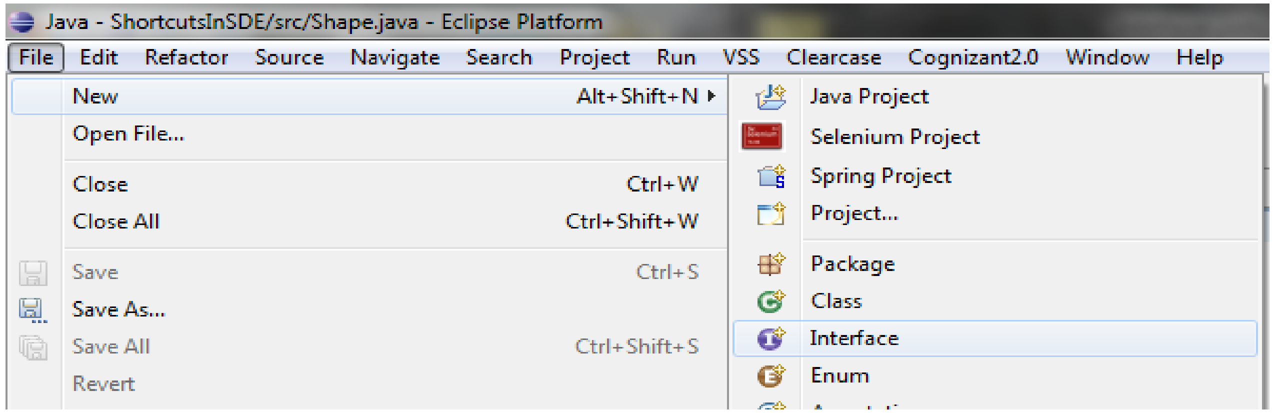
- To create a Shape interface, instead of

  public class Shape

  public interface Shape

- To create an interface using IDE, select

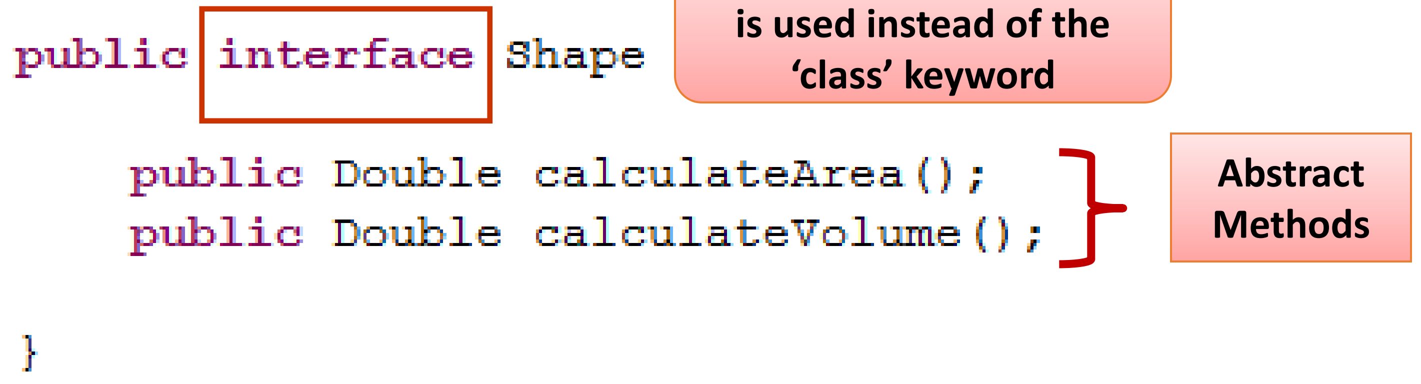
  File → New → Interface



How to implement an Interface?

How to implement an Interface?

Step 1: Create an Interface



```
public interface Shape

    public Double calculateArea();
    public Double calculateVolume();

}
```

The ‘interface’ keyword is used instead of the ‘class’ keyword

Abstract Methods

How to implement an Interface?

How to implement an Interface?

Step 2: Create the implementation class using the '**implements**' keyword

```
public class Circle implements Shape {

    Double radius = 5.0;

    @Override
    public Double calculateArea() {
        Double area = 3.14 * radius * radius;
        return area;
    }

    @Override
    public Double calculateVolume() {
        Double volume = (4/3) * 3.14 * radius * radius * radius;
        return volume;
    }
}
```

'**implements**' keyword is used to indicate that **class Circle** is the **implementation** class for the **interface 'Shape'**

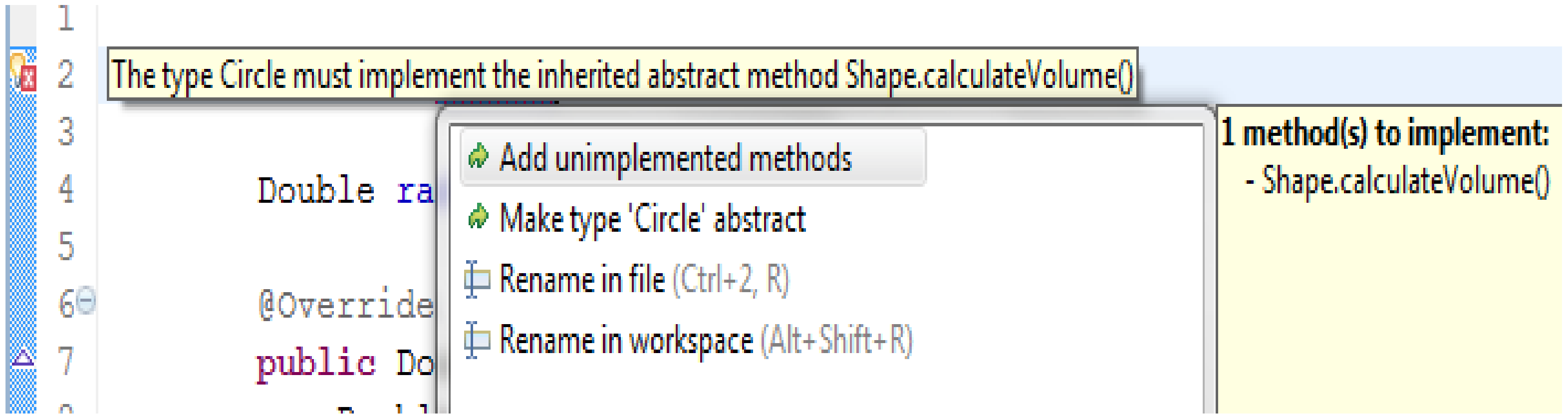
The actual implementation logic for the methods in the interface is provided in this class

Similarly, other shape objects can be created by interface programming by implementing generic **Shape interface.**

Implementing Interface

Some facts about Implementing Interfaces

- A Class can implement any number of interfaces.
- When your class tries to implement an interface, always make sure that you implement all the methods of that interface, or else, you would get the following error at compile time
- Implementing class can have its own methods (which are not present in the Interface class).
- Implementing class can also extend a single super class or abstract class.



```
The type Circle must implement the inherited abstract method Shape.calculateVolume()

        Double ra

        @Override
        public Do
```

Add unimplemented methods
Make type 'Circle' abstract
Rename in file (Ctrl+2, R)
Rename in workspace (Alt+Shift+R)
1 method(s) to implement:
- Shape.calculateVolume()

Why do we use Interface?

Reason #1:

To reveal the programming interface of an object without revealing its implementation:

- This is the concept of encapsulation, hiding implementation from consumers.

Analogy to remote example: Only the plug point is exposed. The method of drawing current using the internal wiring of the electrical appliance is not known to the user

- The implementation can change without affecting the caller of the interface

Analogy to remote example: Any changes required to the wiring in the plug point can be done without any changes in the TV/Fan/Ipod Dock

- The caller does not need the implementation at compile time

Analogy to remote example: The manufacture of the plug point is not related to the manufacture of TV/Fan/Ipod Dock

Why do we use Interface?

Reason #2:

To have unrelated classes implement similar methods (behavior)

The blueprint of the classes will be defined in the interface and all the classes implementing that interface must follow the same blue print (implement the same methods from the interface)

Analogy to the remote example:

All the electrical appliances should adhere to the norms specified by the plug point. Ex. Voltage range from – 220V to 240V

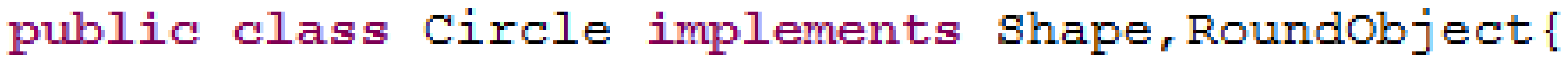
Why do we use Interface?

Reason #3:

To model multiple inheritance:

A class can implement multiple interfaces while it can extend only one class.

Example:



```
public class Circle implements Shape,RoundObject{
```

The concrete class ***Circle*** can implement two(or more) interfaces ***Shape*** and ***RoundObject***.

Here, the class ***Circle*** should implement all the methods declared in both the interfaces

The Java programming language does not support multiple inheritance, but interfaces provide an alternative

Example - Interface

Let us all create interfaces and implementation classes:

1. Create an interface IVehicle with the below methods:

- drive();
- turnLeft();
- brake();

2. Create another interface IPublicTransport with a method

•getNumberOfPeople();

3. Create a class MotorisedVehicle.java with a method checkMotor() which prints the message “The motor of the vehicle is in good condition”

4. Create a class Car.java which extends the MotorisedVehicle class and implements the IVehicle interface. This method should print appropriate messages in the implemented methods. (Ex. “The car is in brake mode” etc)

5. Create a class Train.java which implements both the IVehicle and IPublicTransport interfaces. The implemented methods should print appropriate messages (Ex. “The train is turning left” etc)

Interface vs Abstract classes

Interface	Abstract Classes
All methods of an interface are implicitly abstract, they can only have method declaration.	Abstract classes can have both abstract (method declaration) and concrete methods (methods with implementation).
Can only define constants which are by default *public final static.* Example: int COUNTRY_CODE=1;	Fields with values can be declared. Fields with values declared in abstract classes can be changed in sub class
A class can implement several interfaces.	A class may extend only one abstract class.
Interface definition begins with keyword “interface”, so it is of type interface.	Abstract classes begins with keyword “abstract class”, so it is of type class.
Interface have no implementation and needs to be ‘*implemented*’.	Abstract classes needs to be *extended*.

Interface vs Abstract classes

When to use Interfaces?

Interfaces can be used when common functionalities have to be implemented differently across multiple classes. The user can mandate the use of these common functionalities to all the classes that implement the interface.

When to use abstract classes?

Abstract classes can be used when

- some implemented functionalities are common between classes (this reduces duplicate code)
- some functionalities need to be implemented in sub classes that extend the abstract class

Point to remember: A concrete class can extend only one super class whether that super class is either concrete or abstract class.
But, a concrete class can implement multiple interfaces and at the same time extend one super class

Interface as a Type

How to instantiate an interface?

You cannot create an instance from an Interface.

You can only define an Interface type and assign it to the instance of the concrete class which implemented the interface

Interface as a type

Interface as a Type:

When you define a new Interface, you are defining a new reference type.

If you define a reference variable whose type is an Interface, then any object you assign to it must be an instance of the class that implements the interface

Example:

Person class implements the IPerson interface. You can do:

```
IPerson piVar = new Person();
```

Interface Type

Reference variable of type interface

Instance of class that implements the interface

Example – Declaring Interface reference

Let us now learn to invoke methods of an interface implementation using the same code that we have developed for IVehicle.

Let us learn how to do the below things:

1. Invoke the drive and brake methods of the car object
2. Invoke the checkMotor method on the car object
3. Invoke the drive and brake methods of the train object
4. Invoke the getNumberOfPeople on the train object

Interface – Lend a Hand

To Invoke the brake methods of the car object,

Step 1: Create a InterfaceDemo.java class with a main method to perform all the required actions

Step 2: Create a reference variable of the IVehicle Interface

IVehicle vehicleObj1

Step 3: Assign the reference variable to an instance of the Car Object

IVehicle vehicleObj1 = new Car();

Step 4: Invoke the **drive** and **brake** methods using the interface reference variable

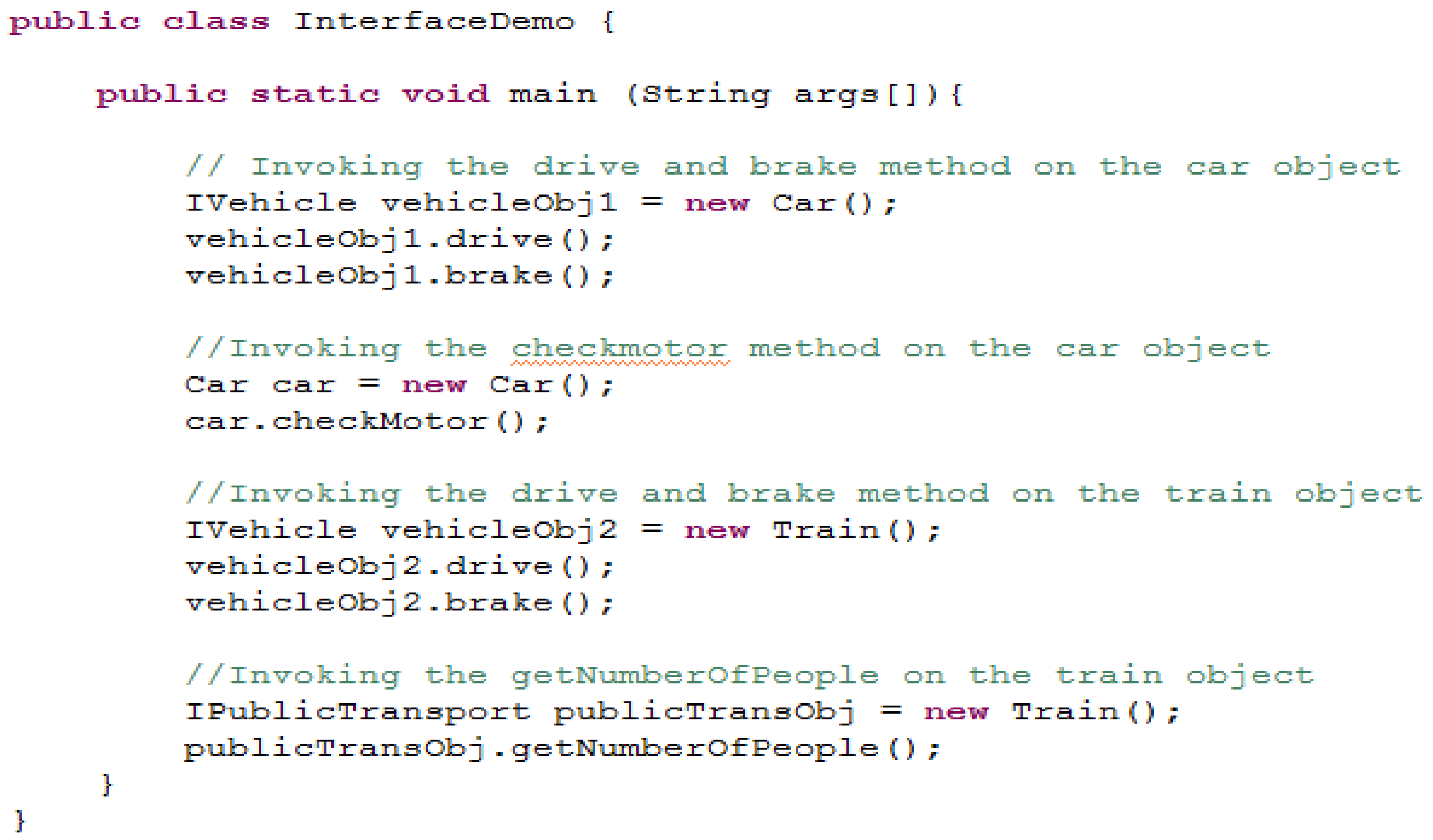
vehicleObj1.drive();

It is your turn to do the rest of the coding !!

Hint: The checkMotor method can be called only on the Car object and
The getNumber of people can be called on the Train object with reference variable pointing to the IPublicTransport interface

Solution - Interface

Solution



```java
public class InterfaceDemo {

    public static void main (String args[]){

        // Invoking the drive and brake method on the car object
        IVehicle vehicleObj1 = new Car();
        vehicleObj1.drive();
        vehicleObj1.brake();

        //Invoking the checkmotor method on the car object
        Car car = new Car();
        car.checkMotor();

        //Invoking the drive and brake method on the train object
        IVehicle vehicleObj2 = new Train();
        vehicleObj2.drive();
        vehicleObj2.brake();

        //Invoking the getNumberOfPeople on the train object
        IPublicTransport publicTransObj = new Train();
        publicTransObj.getNumberOfPeople();
    }
}
```

Interface vs Class

Differences between Interface and a Class:

- The methods of an interface are all abstract methods, they cannot have bodies.
- You cannot create an instance from an interface.

For **Example**:

IPerson piVar = new IPerson(); will throw an ERROR

- An interface can only be implemented by other classes or extended by other interfaces

Developer faces a problem again

Mr. Max, a Java developer from Comcast has a problem !!

He has developed an **interface** called **IVehicle** which can drive and brake.

There are **50 classes** which **implement** this interface.

His client has now asked him to include **a 3rd method** called **accelerate** of his concrete **classes**

If Max makes this change in IVehicle interface, then all 50 classes will break because they do not implement all methods of interface anymore

How can Max enforce this new method in his 25 classes alone without disturbing the other classes?

Max Solution

Use Inheritance of Interfaces

Solution: Create a new Interface **IAdvancedVehicle** that extends the **IVehicle** interface and add the new method in the new interface.

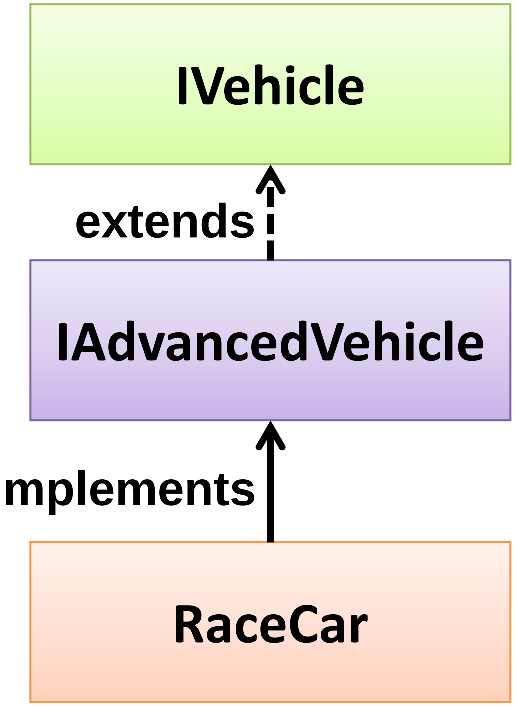
Now, users of your code can choose to continue to apply the old interface or upgrade to the new interface.

People who use the old interface **IVehicle,** need not change their code as the interface is untouched.

Inheritance among Interfaces

Inheritance among Interfaces:

Even though interfaces are not part of the class hierarchy, they can have inheritance relationship among themselves.



```
public interface IVehicle {

    public void drive();
}


public interface IAdvancedVehicle extends IVehicle {

    public void accelerate();

}
public class RaceCar implements IAdvancedVehicle {

    @Override
    public void accelerate() {
        System.out.println("Accelarate");

    }
    @Override
    public void drive() {
        System.out.println("drive");
    }
}
```

Interface and Polymorphism

Interface and Polymorphism:

Interfaces exhibit polymorphism, because interfaces can be used for declaring reference and based on the type of object instance created and injected into the reference the appropriate objects will be triggered.

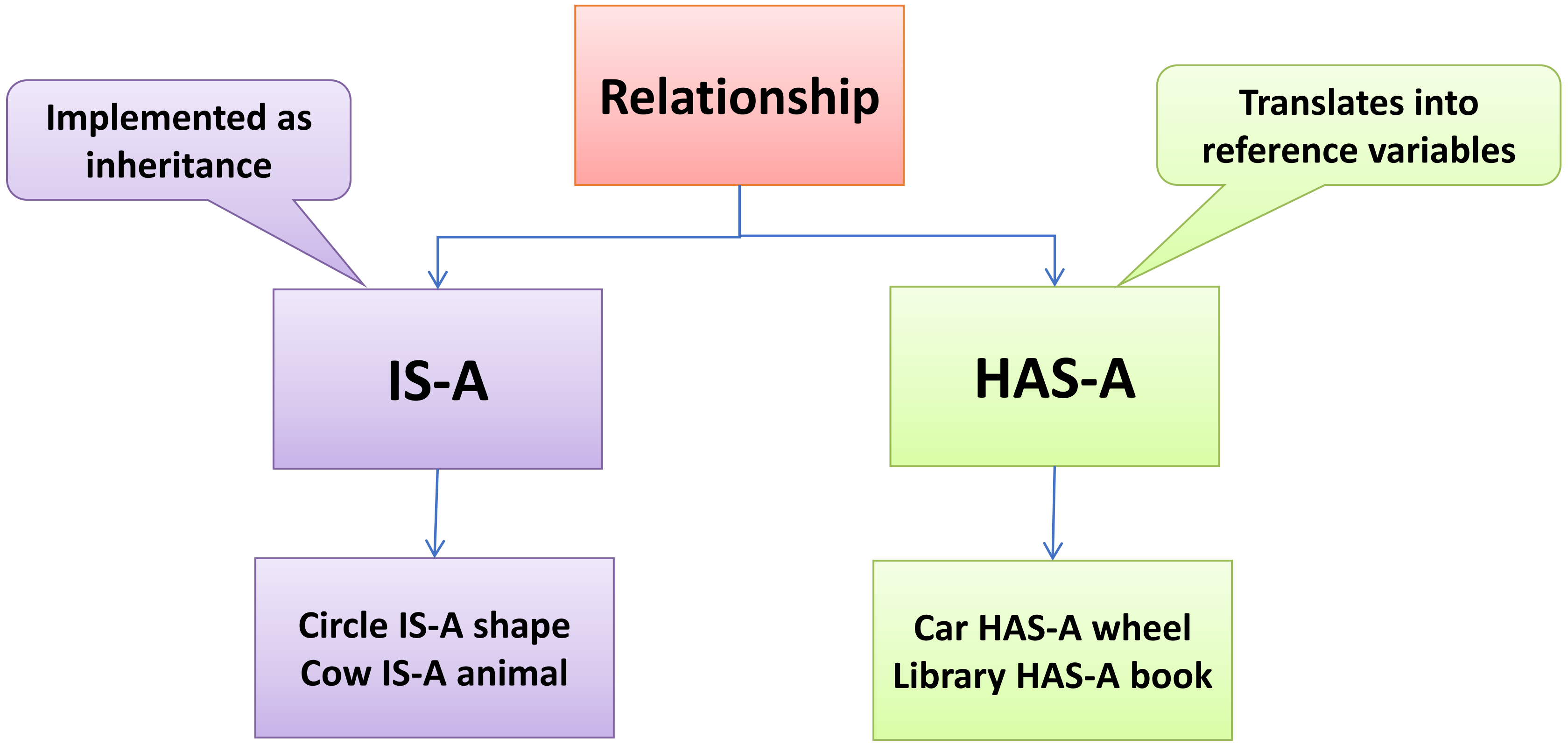
Example:

```
ICar car1= new Honda();
ICar car2= new BMW();
car1.drive();
car2.drive();
```

Based on the objects (**Honda,BMW**) created and injected into the interface reference (**car1, car2**) the appropriate objects method will be invoked.

IS-A and HAS-A Relationship

Java represents two types of relationship



Using IS-A relationship

IS-A relationship:

- When one class **inherits** from another, you say that the subclass **extends** the super class.
- When you want to know if one thing should extend another, use the **IS-A** test.
- **Examples** for IS-A relationship:

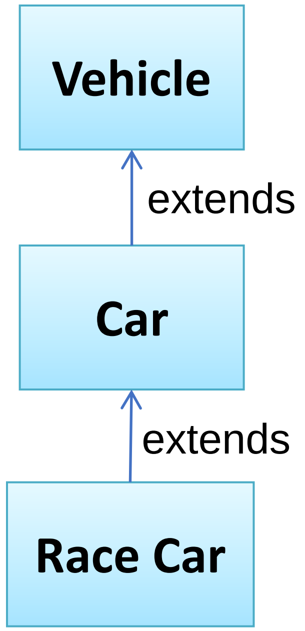
  Triangle IS-A Shape

  Green IS-A Color
- **Do not** apply inheritance if the sub class and super class do not pass the IS-A test

Using IS-A relationship

IS-A relationship:

- The IS-A test works anywhere in the inheritance tree



Car IS-A Vehicle

RaceCar IS-A Car

RaceCar IS-A Vehicle

Note: The IS-A relationship works in only one direction

Example:

Car IS-A vehicle makes sense, but Vehicle IS-A car does not make sense

Using HAS-A relationship

HAS-A relationship:

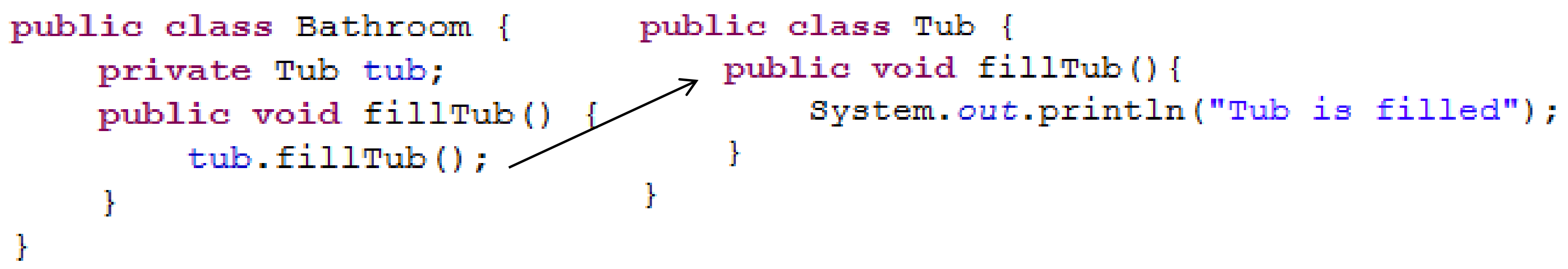
When two classes are related by not through inheritance, then you say that the two classes are joined by HAS-A relationship

Example:

Bathroom HAS-A tub

Tub HAS-A bubble

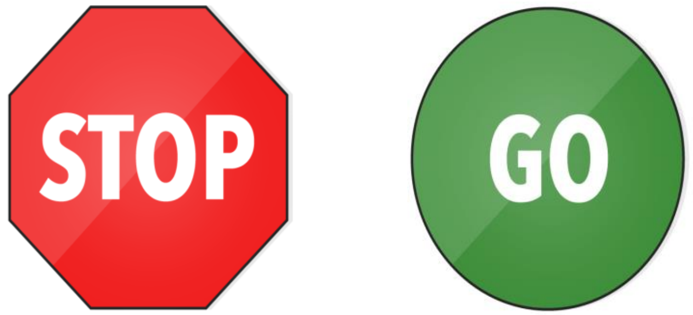
The HAS-A relationship can be translated into reference variables.



```
public class Bathroom {
    private Tub tub;
    public void fillTub() {
        tub.fillTub();
    }
}
```

```
public class Tub {
    public void fillTub(){
        System.out.println("Tub is filled");
    }
}
```

Time To Reflect



Trainees to reflect the following topics before proceeding.

- What is an Interface?
- How to implement the interface?
- Why do we use interfaces?
- Difference between interface and abstract classes?
- How to test if inheritance is required?

Thank you

You have successfully completed

Interface

JAVA @11

Wrapper Classes

Objective

After completing this session you will be able to understand,

- What are Wrapper Classes?
- Need of Wrapper Classes?
- What are the types of Wrapper Classes?
- How to use Wrapper API to manipulate the wrapper classes?

Wrapper Classes

Wrapper Classes are an object representation of primitive data types.

Similarly, the primitive data types (the actual chocolate) are wrapped or converted into objects using wrapper classes

$$\begin{tikzcd}\boxed{\text{Primitive data types}} \arrow[r, "\text{converted to}"] & \boxed{\text{Wrapper Objects}}\end{tikzcd}$$

Why to Wrap primitive Data?

Why do we use wrapper Classes?

- Primitive data type are not objects. Whenever the **data** is **required** as a **object**, wrapper classes can be used to convert the primitive data into an object

  Example: Collection can store only objects so primitives should be converted into Objects using wrapper and stored in collections.

- Wrapper classes provide many **utility** methods for processing the primitive data types.

  Example: Comparing two values, converting number to String etc.

Refer to the Java documentation for the various API's available to process the primitive data using Wrapper classes.
https://docs.oracle.com/javase/8/docs/api/

Primitive To Wrapper Mapping

- Each primitive data type has a corresponding Wrapper class.

- The **name** of the wrapper object is **same** as the primitive data type except that the **first letter** is **capitalized.**

- **Eight** wrapper classes exist in **java.lang** package that represent 8 primitive data types

Example: Wrapper class of

- boolean is **B**oolean

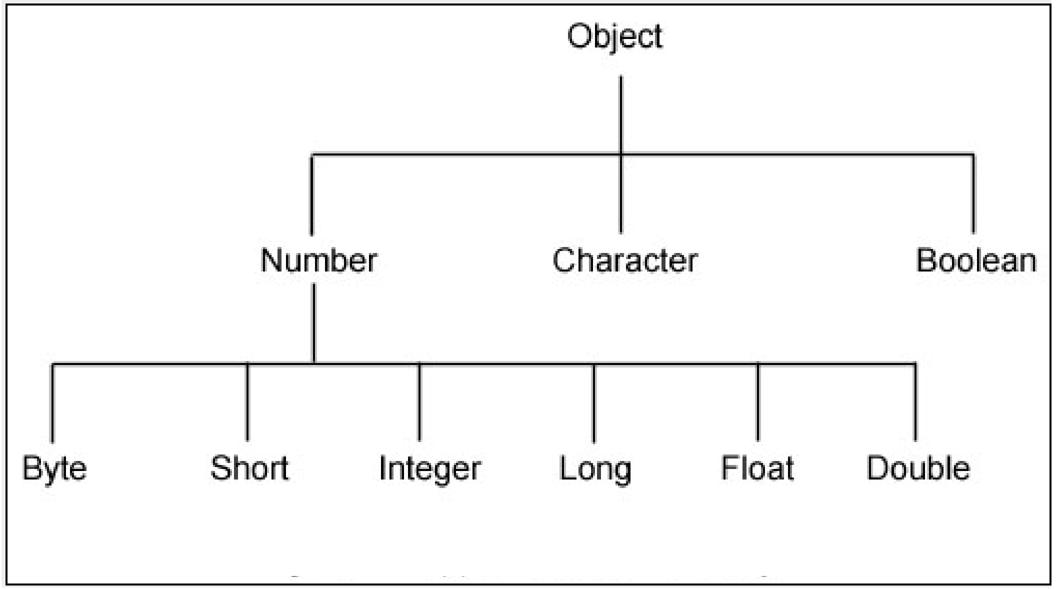
- double is **D**ouble

Primitive To Wrapper Mapping

Primitive Data Type	Wrapper Class
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character

All the wrapper classes comes under the java.lang package.

Wrapper Classes Hierarchy



- Super class of Boolean and Character is Object
- Super class of all Numeric wrapper classes is Number
- Number has six concrete subclasses that hold explicit values of each numeric type Double, Float, Byte, Short, Integer, and Long.

Converting Primitives to Wrappers

How to convert primitive to wrappers?

Wrapper constructors are used to create class objects from primitive types.

Example: **Double** salary = new **Double**(“5.0d”);

Here the double primitive value 5.0 is converted to double wrapper object.

Similarly each primitive can be converted to wrapper object using the respective constructor.

- Boolean boolean1= new Boolean("false");
- Byte byte1= new Byte(˝2˝);
- Short short1= new Short("4");
- Integer int1 = new Integer(˝16˝);
- Long long1 = new Long("123");
- Float float1 = new Float("12.34f");
- Double double1 = new Double("12.56d");
- Character char1 = new Character('c');

Converting Wrappers to Primitives

How to convert wrappers to primitive ?

Each wrapper provides methods to return the primitive value of the Object representation.

Example: double sal = salary.**doubleValue()**;

Here the Salary Double wrapper object is converted to double primitive value.

Wrapper object can be converted to the respective primitive value as follows,

- int i = int1.intValue();
- boolean b = boolean1.booleanValue();
- byte bt = byte1.byteValue();
- short s = short1.shortValue();
- long l = long1.longValue();
- float f = float1.floatValue();
- char c = char1.charValue();

Convert int to String

How to convert int to String & vice versa?

You will often have the need to convert String to int and vice versa. This is done using methods in the Integer class.

Example1:

```
int penn = Integer.parseInt("45000");
```

The above statement converts the String "45000" into an int data type with value as 45000.

Example2:

```
String penn = Integer.toString(100);
```

The above statement converts the int 100 into an String data type with value as 100.

Similarly, String object can be converted to any primitive using the corresponding Wrapper Classes

Wrapper Classes

Let us learn about each of the Wrapper classes in detail

Integer Wrapper Class

The **Integer** class wraps a value of the primitive type **int** into an object.

This class also provides several methods for **converting int** to **String** and vice versa, as well as other useful methods for processing int value.

Let us learn to use the various Integer APIs:

Example 1: To get the primitive int Value of the Integer object

```
Integer intObject = new Integer(5);

int priIntValue = intObject.intValue();
```

The Integer object with value 5 is converted into a primitive int data type.

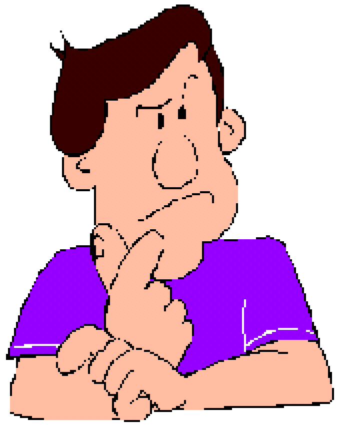
Example 2: To get the value of the Integer as a String.

```
Integer intObject = new Integer(5);

String stringValue = intObject.toString();
```

The Integer object with value 5 is converted into a String Object.

Are You Smart?

The Integer class has lot more APIs.
Are you smart enough to remember all API’s?



You do not have to remember every method in the Integer class. Instead you can refer to the APIs available on the Web.

Navigate to https://docs.oracle.com/javase/8/docs/api/

and go through the various APIs available for the Integer class.

Lend a Hand – Integer **Wrapper**

Refer the Java documentation and choose the right API’s for doing the below exercise.

1. Create a java class “IntegerDemo” inside the package com.wrapper.demos.
2. Create a main method and create two Integer Objects“val1” and “val2” with values 21122 and 43222. Implement the following logic,

   Problem # 1: Convert val1 as int and val2 as long values and print the values as below.

   Expected Output: “The int value of the Integer= ”<int value> & “The long value of the Integer= ”<long value>

   Problem # 2: Compares both the Integer variables val1 and val2 using an API and print the bigger value.

   Expected Output: “ The<value> is bigger than <value>”

   Problem # 3: Retrieves and prints the Maximum and minimum value that an int can have.

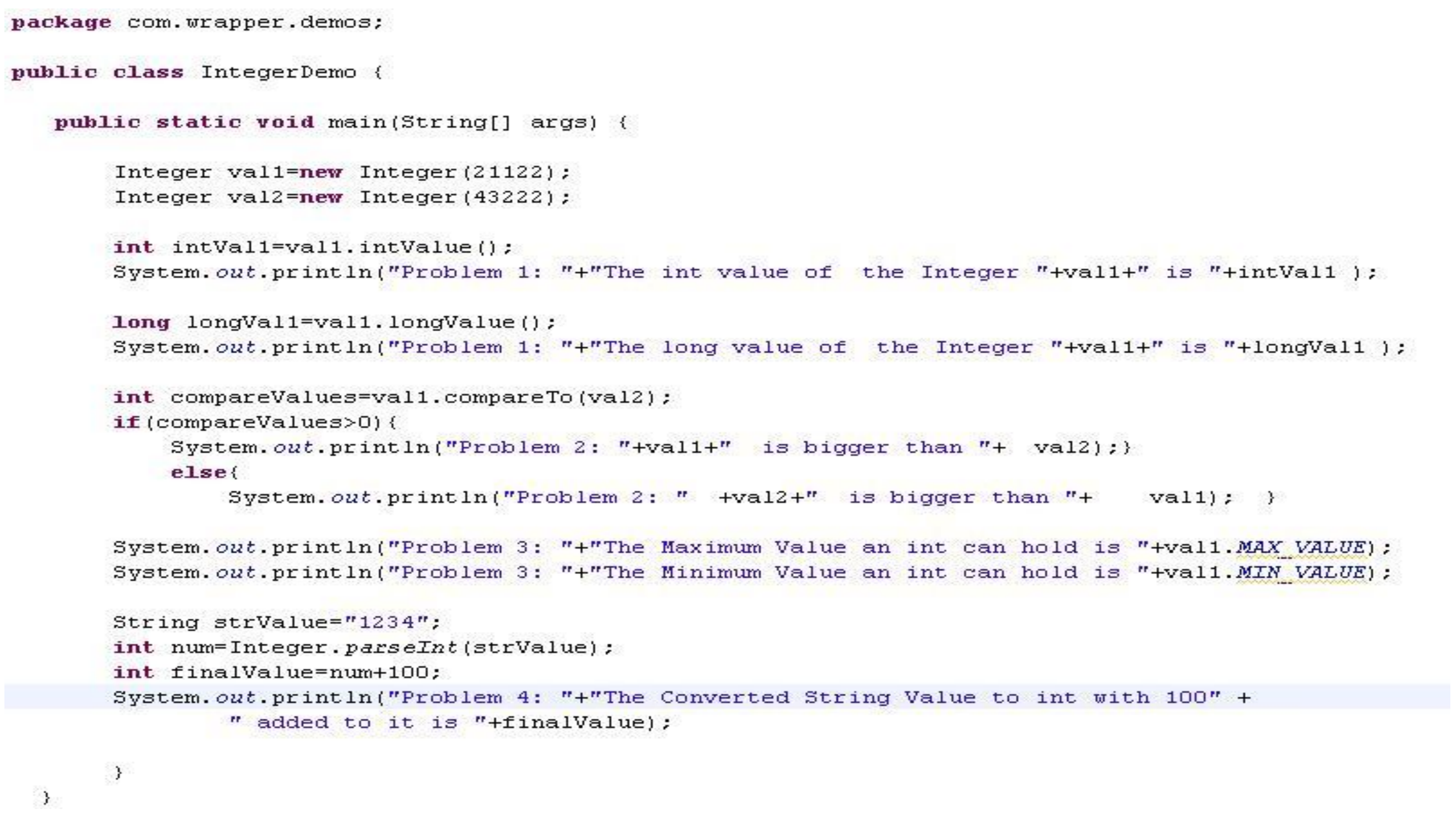
   Expected Output: “Maximum Int value=”<Max Value> “Minimum Int value=”<Min Value>

   Problem # 4: Converts a String value of String s = “1234” to an int value and add 100 to it.

   Expected Output: The string should be converted into a int incremented to 100 and store in a int variable.

Lend a Hand - Solution

Develop the solution as mentioned below and check the output.



```
package com.wrapper.demos;

public class IntegerDemo {

   public static void main(String[] args) {

        Integer val1=new Integer(21122);
        Integer val2=new Integer(43222);

        int intVal1=val1.intValue();
        System.out.println("Problem 1: "+"The int value of  the Integer "+val1+" is "+intVal1 );

        long longVal1=val1.longValue();
        System.out.println("Problem 1: "+"The long value of  the Integer "+val1+" is "+longVal1 );

        int compareValues=val1.compareTo(val2);
        if(compareValues>0){
            System.out.println("Problem 2: "+val1+"  is bigger than "+  val2);}
            else{
                System.out.println("Problem 2: "  +val2+"  is bigger than "+    val1);  }

        System.out.println("Problem 3: "+"The Maximum Value an int can hold is "+val1.MAX_VALUE);
        System.out.println("Problem 3: "+"The Minimum Value an int can hold is "+val1.MIN_VALUE);

        String strValue="1234";
        int num=Integer.parseInt(strValue);
        int finalValue=num+100;
        System.out.println("Problem 4: "+"The Converted String Value to int with 100" +
                " added to it is "+finalValue);

        }
  }
```

Long Wrapper Class

Class Long:

The **Long** class wraps a value of the primitive type **long** in an object.

This class also provides several methods for **converting long** to **String** and vice versa, as well as other **constants** and useful methods when processing a long value.

Long is similar to a Integer, the difference is size of Long is 64 bits rather int is 32 bits.

Syntax:

```
Long salary= new Long("12678") // Converting a String to a Long Wrapper
```

(or)

```
Long salary = new Long(12678); // Converting a long primitive to a Long Wrapper.
```

Float Wrapper Class

Class Float:

The *Float* class wraps a value of the primitive type *float* in an object.

This class also provides several methods for converting *float* to *String* and vice versa, as well as other useful methods when dealing with a float.

Example:

$$\pi = 3.145$$

Lend a Hand – Float Wrapper

Refer the Java documentation and choose the right API's for doing the below exercise.

1. Create a java class "FloatDemo".
2. Create a main method and create two Float objects "val1" and "val2" with values 12.56f and 22.89f. Implement the following logic,

Problem 1: Compares two float variables and displays the maximum value.

Expected Output : " The<value> is bigger than <value>"

Problem 2: Converts a String value of "1234" to a float value and add 10.2f

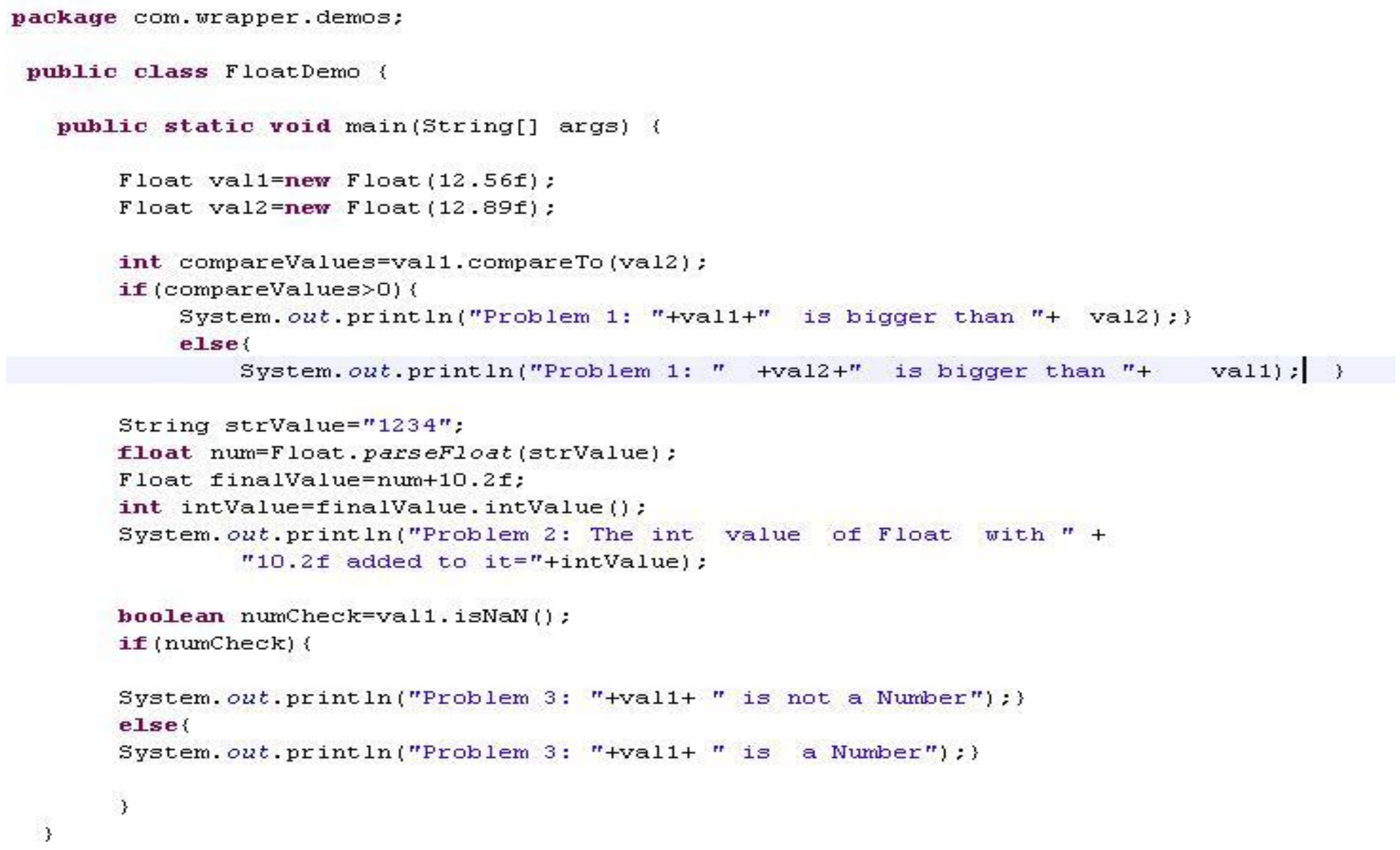
Expected Output: The string should be converted into a float incremented to 100 and store in a int variable.

Problem 3: Checks if the specified float value val1 is a number or not.

Expected Output: If the number is not a number display the message "This is not a Number" else "This is a Number"

Lend a Hand - Solution

Develop the solution as mentioned and check the output.



```
package com.wrapper.demos;

 public class FloatDemo {

   public static void main(String[] args) {

       Float val1=new Float(12.56f);
       Float val2=new Float(12.89f);

       int compareValues=val1.compareTo(val2);
       if(compareValues>0){
           System.out.println("Problem 1: "+val1+"  is bigger than "+  val2);}
           else{
               System.out.println("Problem 1: "  +val2+"  is bigger than "+    val1);  }

       String strValue="1234";
       float num=Float.parseFloat(strValue);
       Float finalValue=num+10.2f;
       int intValue=finalValue.intValue();
       System.out.println("Problem 2: The int  value  of Float  with " +
               "10.2f added to it="+intValue);

       boolean numCheck=val1.isNaN();
       if(numCheck){

       System.out.println("Problem 3: "+val1+ " is not a Number");}
       else{
       System.out.println("Problem 3: "+val1+ " is  a Number");}

       }
 }
```

Double Wrapper class

Class Double:

The *Double* class wraps a value of the primitive type *double* in an object.

This class also provides several methods for converting double to String and vice versa, as well as other useful method to process double value.

Double is similar to a float, the difference is size of double data type is 64 bits rather float is 32 bits.

Where is it used?

This is typically used for storing currency values.

salary = 5000.45

Lend a Hand – Double

Refer the Java documentation and choose the right API's for doing the below exercise.

1. Create a java class "DoubleDemo".
2. Create a main method and create two Double objects "val1" and "val2" with values 87.89 and 212.82.The main method will have to print the following

Problem 1: Converts the val1 to int and val2 to float value.

Expected Output: "The int value of the Double Wrapper = "<double value> & "The float value of the Double Wrapper= "<long value>

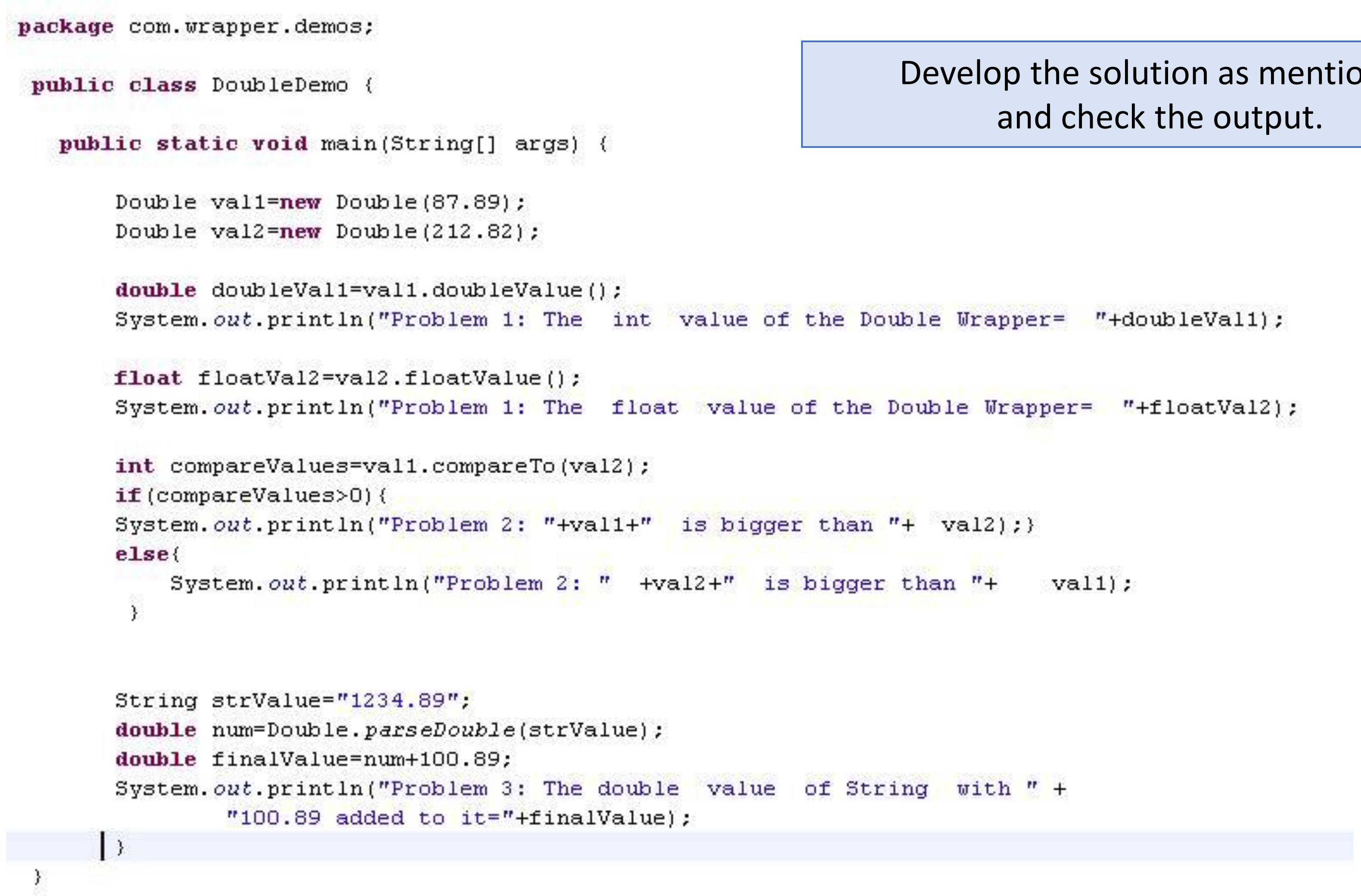
Problem 2: Compares two double variables.

Expected Output: " The<value> is bigger than <value>"

Problem 3: Converts a String value of "1234.89" to an double value and add 100.89 to it.

Expected Output:"The double value of the String with 100.89 added to it ="<double value>".

Lend a Hand - Solution



```
package com.wrapper.demos;

 public class DoubleDemo {

   public static void main(String[] args) {

       Double val1=new Double(87.89);
       Double val2=new Double(212.82);

       double doubleVal1=val1.doubleValue();
       System.out.println("Problem 1: The  int  value of the Double Wrapper=  "+doubleVal1);

       float floatVal2=val2.floatValue();
       System.out.println("Problem 1: The  float  value of the Double Wrapper=  "+floatVal2);

       int compareValues=val1.compareTo(val2);
       if(compareValues>0){
       System.out.println("Problem 2: "+val1+"  is bigger than "+  val2);}
       else{
           System.out.println("Problem 2: "  +val2+"  is bigger than "+    val1);
        }


       String strValue="1234.89";
       double num=Double.parseDouble(strValue);
       double finalValue=num+100.89;
       System.out.println("Problem 3: The double  value  of String  with " +
               "100.89 added to it="+finalValue);
       }
 }
```

Develop the solution as mentioned
and check the output.

Byte Wrapper Class

Class Byte:

The **Byte** class wraps a value of the primitive type **byte** in an object.

This class also provides several methods for **converting byte** to **String** and vice versa, as well as other **constants** and useful methods for processing a byte.

Where is it used?

This is typically used for storing numbers ranging between **-128 and 127**.

Syntax:

```
Byte id = new Byte(“12”) //Converting a String to a Byte Wrapper
```

(or)

```
Byte id = new Byte(12)   //Converting a primitive to a Byte Wrapper
```

Short Wrapper Classes

Class Short:

The **Short** class wraps a value of the primitive type **short** in an object.

This class also provides several methods for **converting short** to **String** and vice versa, as well as other **constants** and useful methods. useful when dealing with a short.

Where is it used?

This is typically used for storing numbers ranging between **-32768 and 32767.**

Syntax:

```
Short id = new Short("1261")      //Converting a String to a Short Wrapper
```

(or)

```
Short salary = new Short (1267);//Converting a short  primitive to a Short Wrapper.
```

Character Wrapper Class

Class Character:

The **Character** class wraps a value of the primitive type **char** in an object.

This class also provides several methods for determining a character’s category(lowercase letter, digit etc.) and for converting characters from uppercase to lowercase and vice versa.

Example: Characters ‘a’, ‘b’, ‘5’, ‘?’, ‘A’ can be represented a Character Object

Syntax:

```
Character flag = new Charachter(‘A’)  //Converting a String to a Boolean Wrapper
```

Boolean Wrapper class

Class Boolean:

The **Boolean** class wraps a value of the primitive type **boolean** in an object.

This class also provides several methods for **converting boolean** to **String** and vice versa, as well as other **constants** and useful methods when processing boolean data.

Syntax:

Boolean flag = new Boolean(“true”) //Converting a String to a Boolean Wrapper

(or)

Boolean flag = new Boolean(true); //Converting a boolean primitive to a Boolean Wrapper.

Lend a Hand – Boolean

Refer the Java documentation and choose the right API's for doing the below exercise.

1. Create a java class "**BooleanDemo**".
2. Create a main method, inside the main method, declare two Boolean wrapper objects, val1 and val2 with values true and false respectively.
3. The main method should implement the following logic,

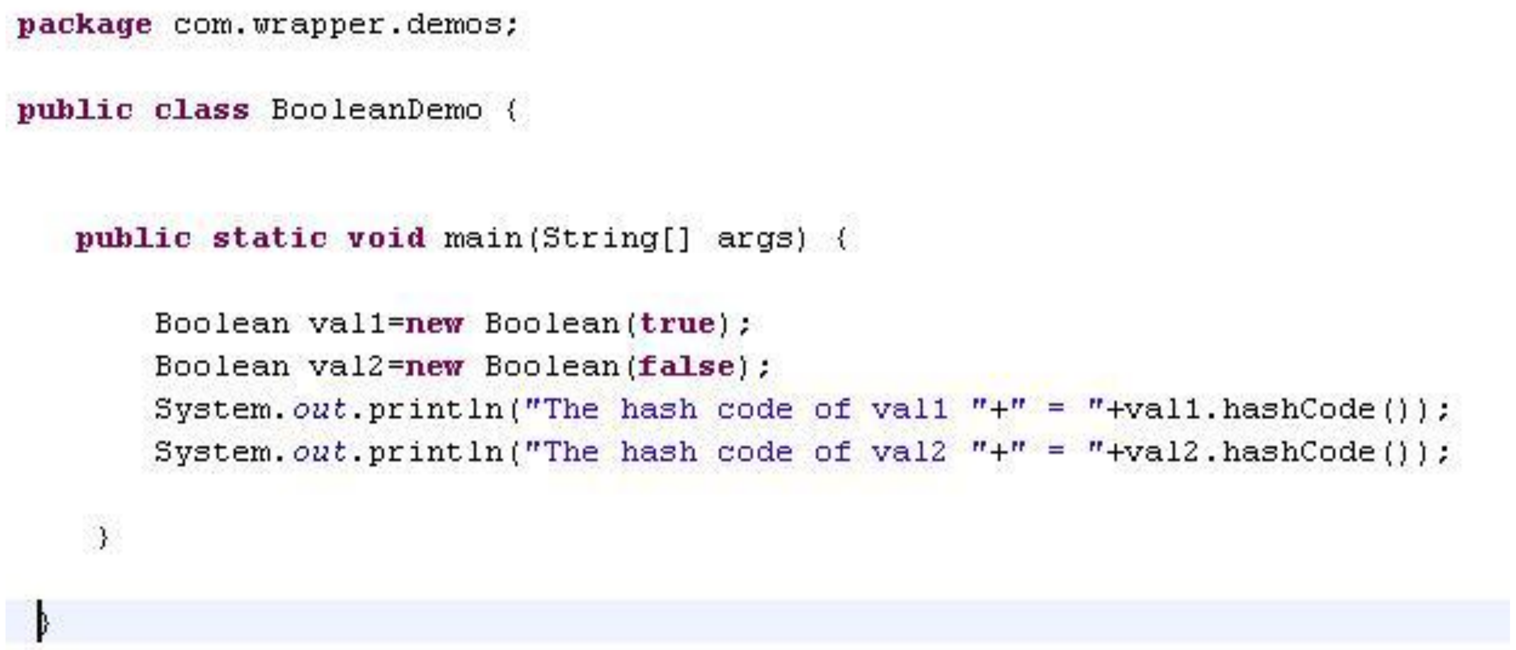
Problem # 1: The hashcode for val1 and val2 needs to be printed.

Expected Output:

"The hash code of val1 = "<value> & "The hash code of val2= "<value>

Lend a Hand - Solution

Develop the solution as mentioned
and check the output.



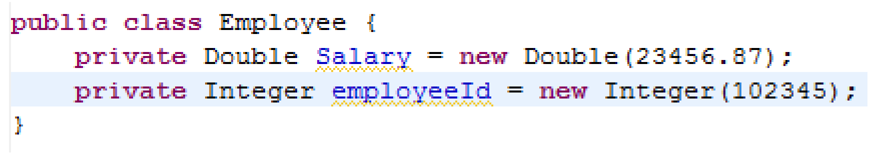
```java
package com.wrapper.demos;

public class BooleanDemo {


    public static void main(String[] args) {

        Boolean val1=new Boolean(true);
        Boolean val2=new Boolean(false);
        System.out.println("The hash code of val1 "+" = "+val1.hashCode());
        System.out.println("The hash code of val2 "+" = "+val2.hashCode());

    }

}
```

Where Wrappers can be used?

Wrappers can be used in the following scenarios,

- As instance variables in class.

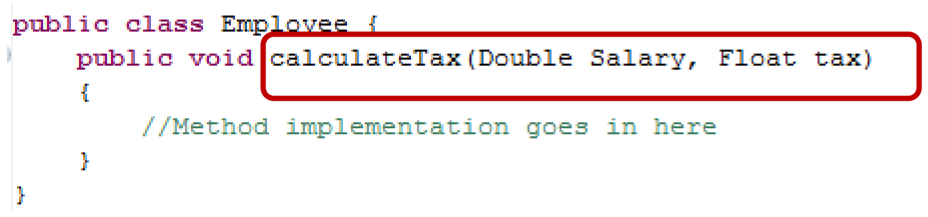
Example:



```
public class Employee {
    private Double Salary = new Double(23456.87);
    private Integer employeeId = new Integer(102345);
}
```

- As method arguments.

Example:

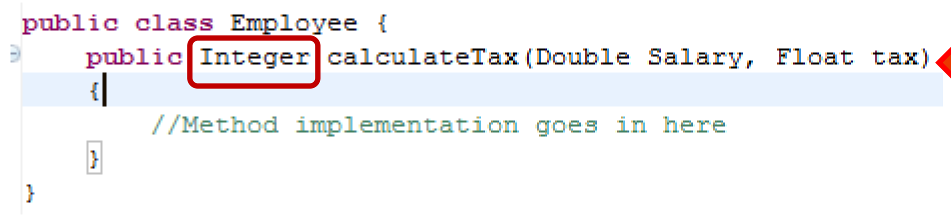


```
public class Employee {
    public void calculateTax(Double Salary, Float tax)
    {
        //Method implementation goes in here
    }
}
```

Method with Double and Float arguments.

- As method return type.

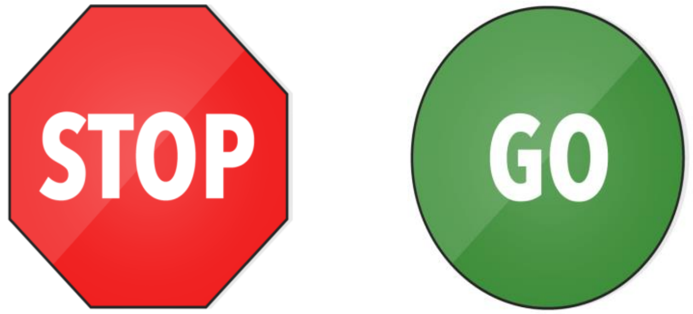
Example:



```
public class Employee {
    public Integer calculateTax(Double Salary, Float tax)
    {
        //Method implementation goes in here
    }
}
```

Method returns Integer type.

Time To Reflect



Trainees to reflect the following topics before proceeding.

- Why do we need Wrapper Objects?
- What Wrapper Object is used to store currency data?
- How many Wrapper Objects are there?
- What is the package under which the Wrapper objects are present?

Thank you

You have successfully completed

Wrapper Class

JAVA @11

Arrays

Objective

After completing this session you will be able to understand,

- what is an Array?
- List out the types of Array.
- Declare and create an array.
- Iterate through an array.

Metaphor for Arrays

A box of Apples. Box is the container for apples.

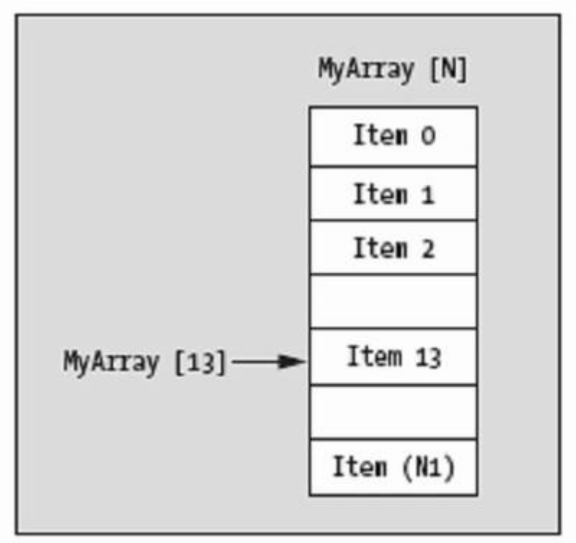
Similar to box Arrays are container for data of similar data type (like the apples).

What is an Array?

What is an Array?

An array is variable that can hold a group of values of *same type* and referred by a common name.

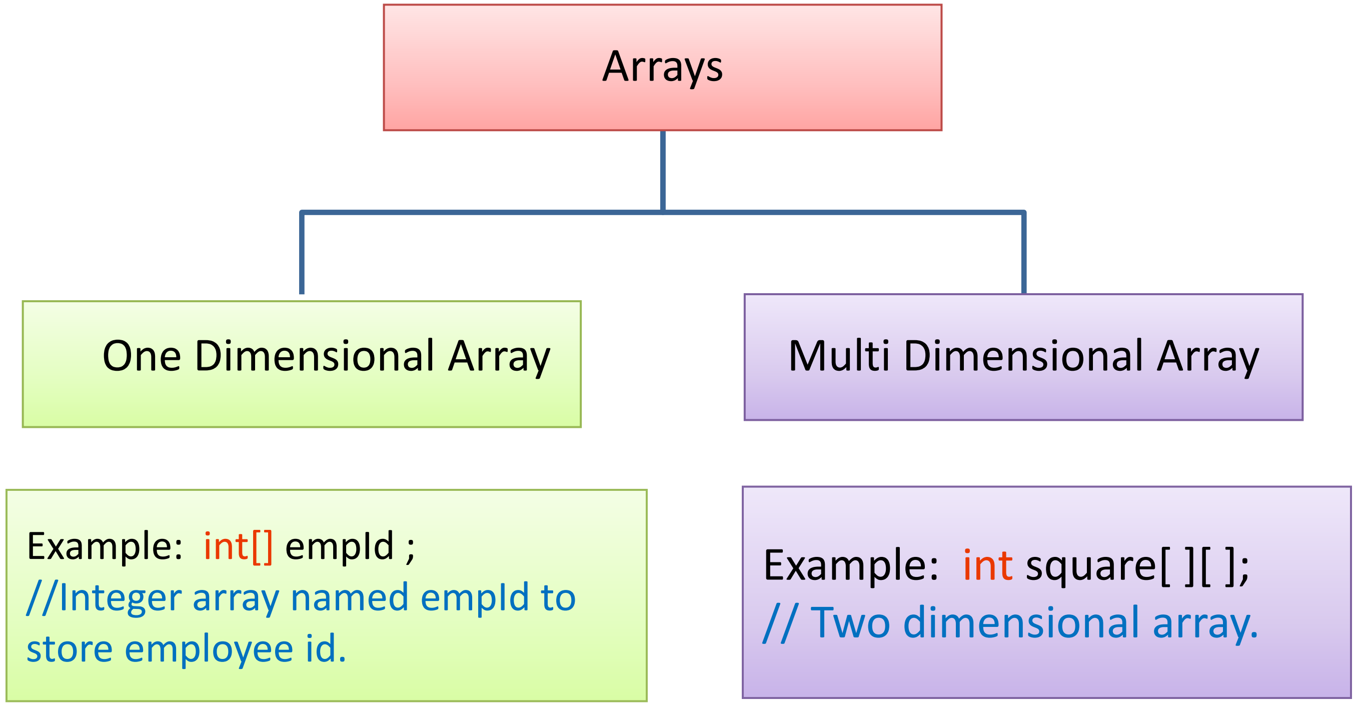
Arrays can hold primitives or objects of same types.



Individual values being stored inside an array named *MyArray*.

Each item can be accessed using the variable name *MyArray*.

Types of Array



Example: `int[] empId ;`
`//Integer array named empId to store employee id.`

Example: `int square[ ][ ];`
`// Two dimensional array.`

Declaring and Creating an One Dimensional Array

1. **Declaring an Array:**

   An array variable declared should has a data type and a valid identifier.

   Syntax : <type> [ ] <array-name>;

   Example :

   int [ ] empId; — Declares an array of type int

   Employee[ ] emp; — Declares an array of Employee Object

2. **Initializing an Array** As arrays are also a type of object, they are created with the new keyword.

   Syntax: <array-name>= new <type>(<size>);

   Example:

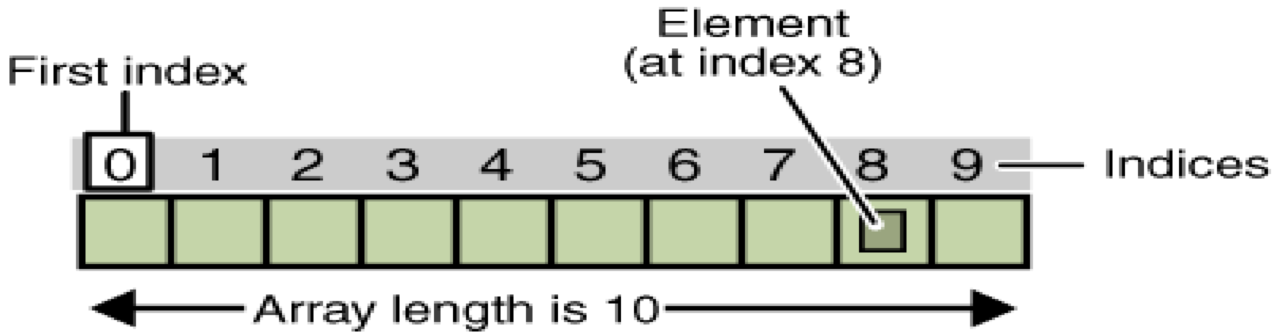
   empId=new int[3]; — Creates an array that can hold 3 int values

   emp=new Employee[3]; — Creates an array that can hold 3 Employee objects.

One Dimensional Arrays

Some Facts about arrays:

- The ***length*** of an array is established when the array is created.
- The ***length*** of an array is ***fixed*** at the time of its creation.
- Each item in an array is called an ***element***, and each element is accessed by its numerical ***index***.



The above diagram depicts an array of ten elements

Adding Elements to an One Dimensional Array

How to add values to a one dimensional array?

Based on the data type of the array the respective values can be stored inside the array.

Syntax:

```
<array-name>[<index-number>] = <value>;
```

Example:

This sets the empName & empId objects in Employee Class

```
empId[0]=76;
empId[1]=13;
empId[2]=56;
```

This adds the employee id's in the 'empId' array`

```
emp[0]=new Employee(“Arun”,12000);
emp[1]=new Employee(“Ram”,11100);
emp[2]=new Employee(“Raj”,8000);
emp[3]=new Employee(“Sam”,10000);
```

This adds employee object

Alternate way of adding elements to an One Dimensional Array

Let us look at an alternate way to create, initialize and assign values in one step,

```
int[] empId = { 76, 13, 56, 87};
```

This declares an array of type int with values 76, 13, 56 and 87

```
Employee[] emp={new Employee(“Ram”,22),new Employee(“Arun”,44)};
```

This declares an Employee object array, stores two employee objects

Here, the length of the array is determined by the number of values stored in the array.

Accessing the Elements of an One Dimensional Array

How to retrieve values from an array?

Array elements are accessed using the element's index.

The index position of the first element in the array is 0, last element is position is array length -1.

Syntax:

```
<type> <array name> = <array-name>[index-number];
```

Example:

```
int id=empId[0];          Employee emp1=emp[3];
```

Retrieves the first item in empId array

Retrieves the fourth item in emp array

How to find the length of an array?
Use empId.length; to get the total number of items in the array.

Lend a Hand–One Dimensional Array

Program to print even numbers between 0 & 100.

1. Create a program “ArrayDemo” add two methods,
    1. storeNumbers – Creates and stores a array with values from 0~100
    2. printEvenNumber – traverse through the array stored and print all the even numbers.

    From the main method invoke both the mentioned above and print the even numbers.

Lend a Hand–Solution

```
public class ArrayDemo {

    int[] numbers;

    public void storeNumbers() {
        numbers = new int[101];
        for (int i = 0; i <= 100; i++) {
            numbers[i] = i;
        }
    }

    public void printEvenNumber() {
        System.out.println("The even numbers between 0 and 100 are ");
        for (int i = 0; i < numbers.length; i++) {
            if ((numbers[i] % 2) == 0) {
                System.out.println(numbers[i]);
            }
        }
    }

    public static void main(String args[]) {
        ArrayDemo arrayDemo = new ArrayDemo();
        arrayDemo.storeNumbers();
        arrayDemo.printEvenNumber();
    }
}
```

This method stores 100 numbers in an array.

This method prints the even numbers between 0 and 100.

Invokes the methods.

Multi Dimensional Arrays

What are multi dimensional arrays?

Multi Dimensional Arrays are *array of arrays*.

The two dimensional array can be termed as a physical table with rows and columns, each row labeled with an index of 0 to its maximum bound.

Syntax:

```
type  array-name = new type[rows][cols];
```

Example:

```
int marks[ ][ ] = new int[2][3 ]; // 3 rows and 4 columns
```

A two dimensional array is similar to a matrix representation as depicted here

	0	1	2	3
0	41	38	28	31
1	32	34	36	43
2	23	31	12	18

Advantages and Disadvantages of Arrays

Advantages of Java Array:

- Arrays can store large number of elements by just specifying the index number and the array name.
- Arrays permit efficient random access
- Iteration in arrays is faster than iterating through its counterparts (such as a linked list of the same size)

Disadvantages of Java Array:

- An array has fixed size.
- An array holds only one type of data.
- Insertion and deletion of elements is not efficient

Thank you

You have successfully completed

Arrays

JAVA @11

Exception Handling

Objective

After completing this session you will be able to understand,

- Introduction to Exception Handling
- Exception Hierarchy
- Try-Catch-Finally
- Multiple catch block
- Nested try block
- Throws keyword
- Throw keyword
- User Defined Exception

Exception

What is an Exception?

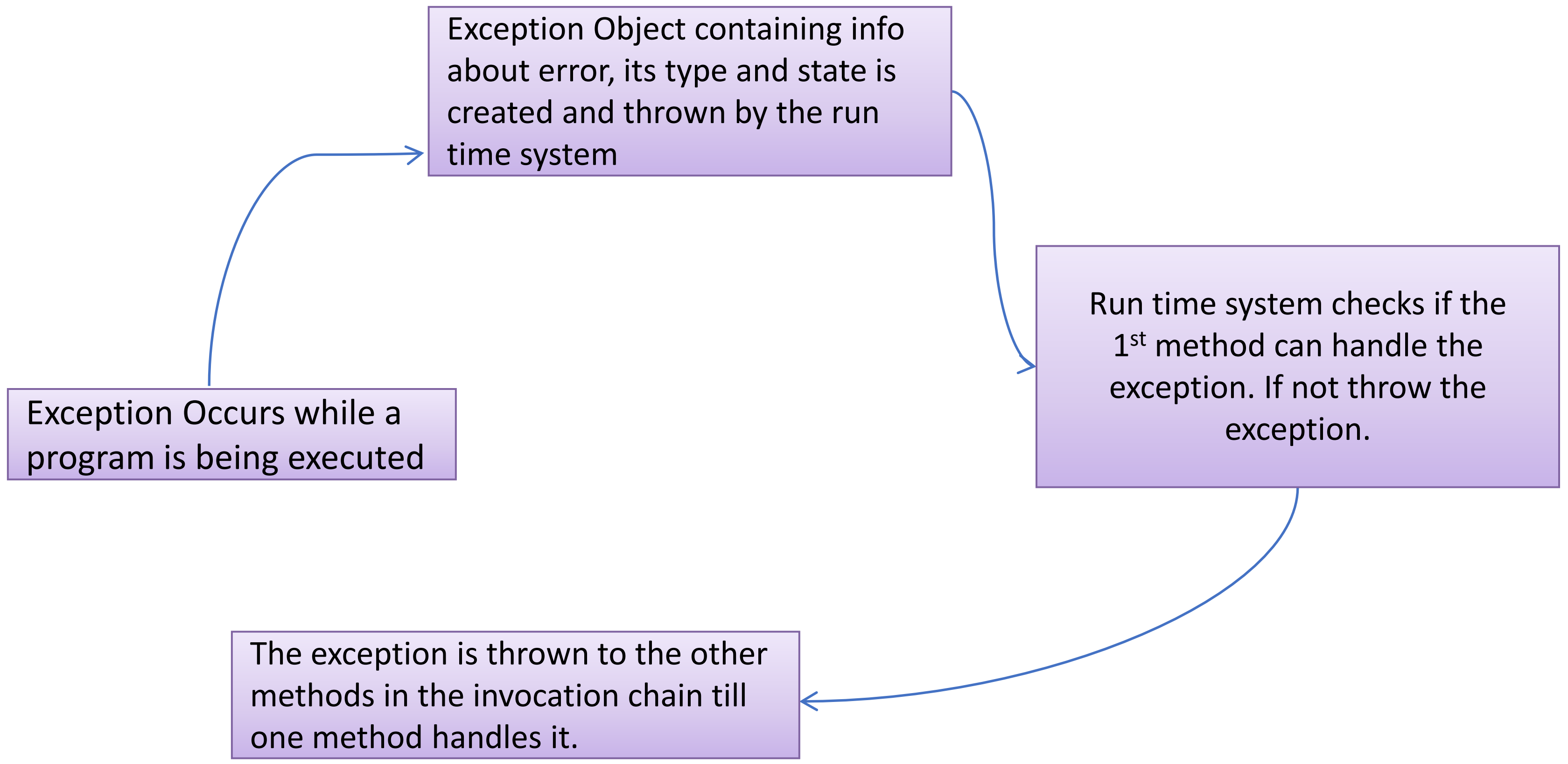
Exception refers to any abnormality or an **error** that occurs during **run time**

- An exception in Java is a signal that indicates the occurrence of some important or unexpected condition during execution of a program at runtime.
- Exception causes normal program flow to be disrupted.

Examples :

- **int num=5/0 – Divide by Zero Error –Arithmetic Exception**
- **Out of Memory Error.**
- **Trying to open a file that has been deleted.**

Exception Handling



Exception Occurs while a program is being executed

Exception Object containing info about error, its type and state is created and thrown by the run time system

Run time system checks if the 1st method can handle the exception. If not throw the exception.

The exception is thrown to the other methods in the invocation chain till one method handles it.

In Java Handlers are used for handling exceptions.

What happen When an Exception occurs in a Program?

What happens when an Exception occurs?

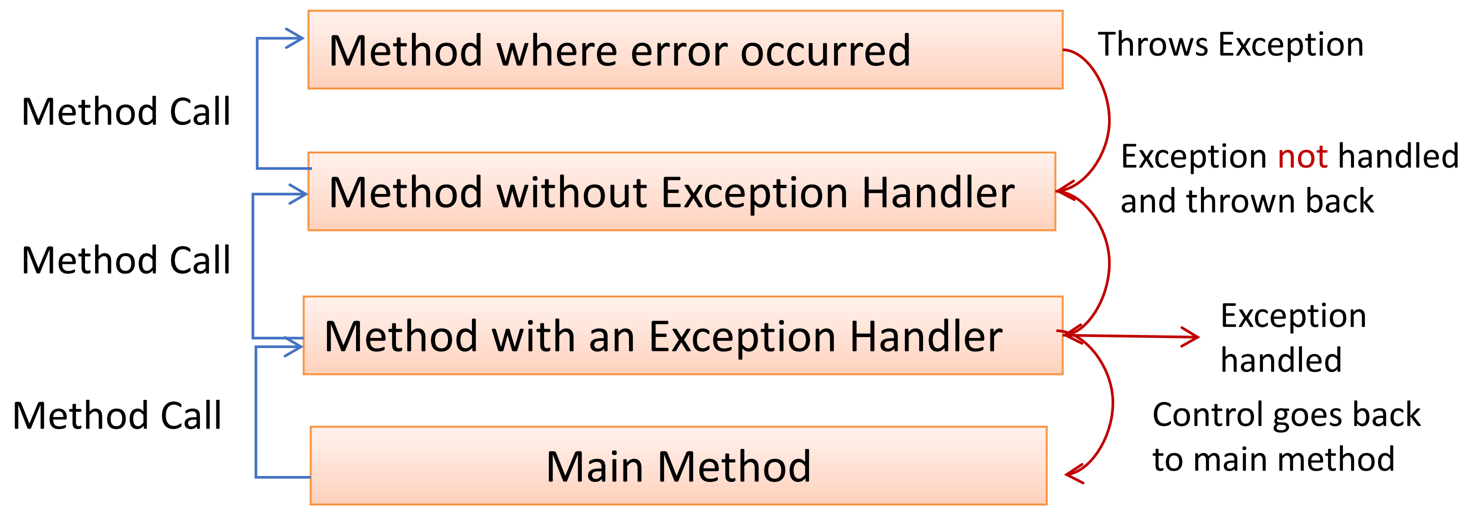
Step 1:

When an exception occurs within a method, the method creates an exception object and hands it off to the run-time system (called throwing an exception)

Exception object contains information about the error, including its type and the state of the program when the error occurred.

Step 2:

The run time system searches the call stack for a method that contains the method handler


Method where error occurred
Method without Exception Handler
Method with an Exception Handler
Main Method
Method Call
Method Call
Method Call
Throws Exception
Exception not handled and thrown back
Exception handled
Control goes back to main method

What happen when an Exception occurs in a Program?

Step 3:

When an appropriate handler is found, the run-time system passes the exception to the handler,

- The exception handler catches the exception and handles the exception.

If the run-time system cannot find an appropriate method to handle the exception, then the run-time system terminates and uses the default exception handler.

We will learn about how the handler handles the exception in the subsequent slides.

Try to run this program in your Eclipse IDE

```
public class DivByZero {
    public static void main(String args[]) {
        System.out.println(3/0);
        System.out.println("Pls. print me");
        }
}
```

You can notice the following exception thrown by the run time system,

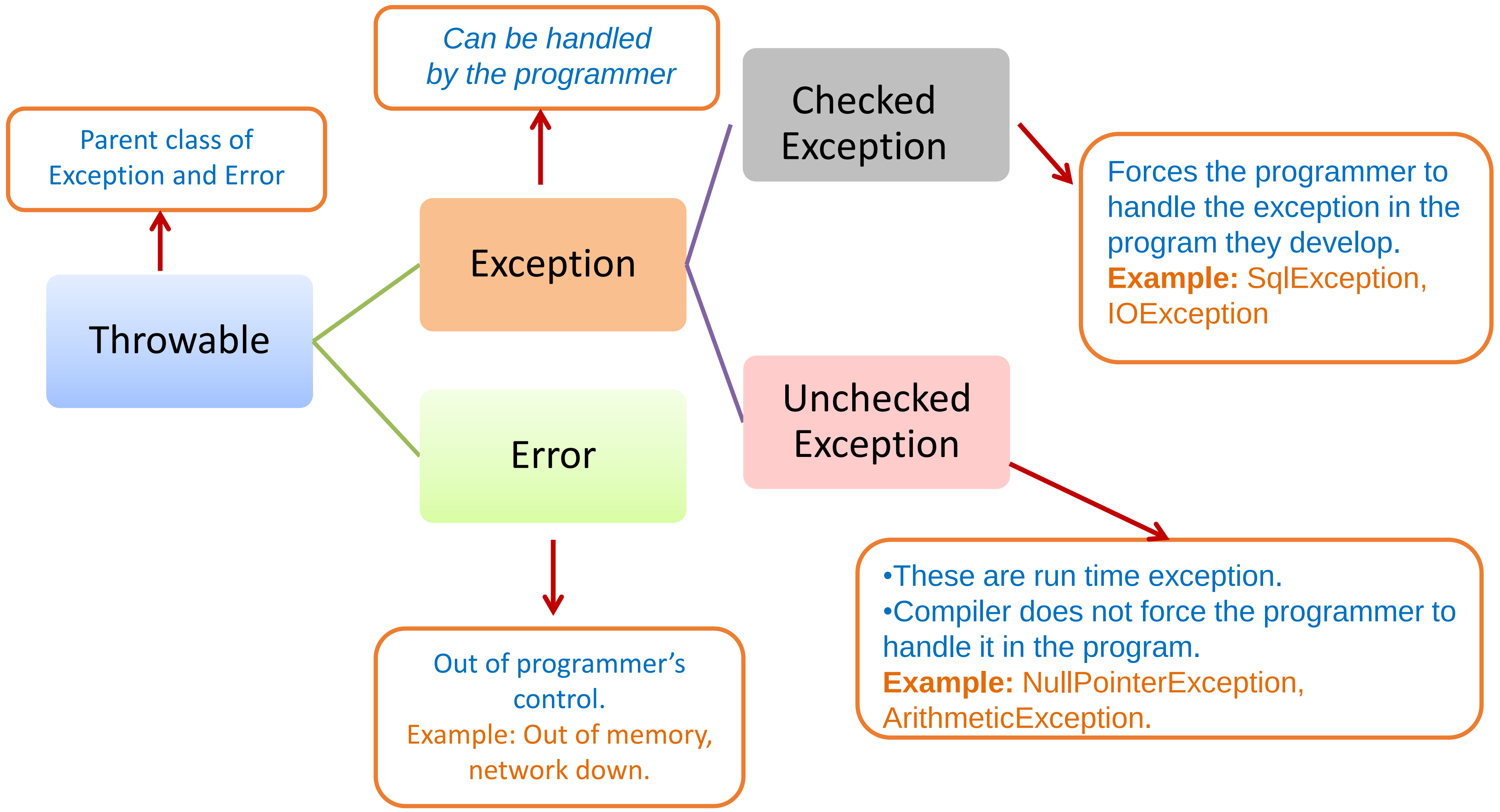
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at com.cognizant.academy.handson.DivByZero.main(DivByZero.java:3)
```

Benefits of Exception

Benefits of Java Exception Handling Framework:

- It separates Error-Handling code from “regular” business logic code.
- It can propagate errors up the call stack till a handler handles the exception.
- It can group and categorize the exception types.

Exception Hierarchy



Parent class of Exception and Error
Throwable
Can be handled by the programmer
Exception
Error
Out of programmer's control.
Example: Out of memory, network down.
Checked Exception
Forces the programmer to handle the exception in the program they develop.
Example: SqlException, IOException
Unchecked Exception
•These are run time exception.
•Compiler does not force the programmer to handle it in the program.
Example: NullPointerException, ArithmeticException.

Checked vs Unchecked

Checked Exception	**Unchecked Exception**
At compile time, the java compiler automatically checks that a program contains handlers for checked exceptions.	The compiler doesn’t force them to be declared in the throws clause.
Checked exceptions must be explicitly caught or propagated using **try-catch-finally** blocks	Unchecked exceptions do not have this requirement. They don't have to be caught or declared thrown.
Checked exceptions in Java extend the *java.lang.**Exception*** class	Unchecked exceptions extend the *java.lang.**RuntimeException***.
Exception handling is mandated by JVM for these exceptions	It is not advisable to catch these exceptions since it might make the code unstable.
Example: **IOException**	Example: **NullPointerException**

Checked Exception

CheckedException	Description
IOException	Signals that an I/O exception of some sort has occurred.
InterruptedException	Thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted, either before or during the activity.
ParseException	Signals that an error has been reached unexpectedly while parsing.
SQLException	An exception that provides information on a database access error or other errors.

Uncheck Exception

UnCheckedException	Description
ArithmeticException	Thrown when an exceptional arithmetic condition has occurred.
ClassCastException	Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance.
NullPointerException	Thrown when an application attempts to use null in a case where an object is required.
ArrayIndexOutOfBoundsException	Thrown to indicate that an array has been accessed with an illegal index.

Exception Handling

How can Exception be handled?

Exception can be handled using a exception handler.

What is a Exception Handler?

A set of code which can handle an error conditions in a program systematically by taking necessary action

Exception Handling Techniques:

- Option I: try-catch-finally
- Option II: throws.

Option I – try, catch & finally

try-catch

When a program performs an operation that causes an exception, an exception will be thrown. Exception can be handled by using the try and catch blocks.

How is it done?

- The suspected code is embraced in the try block, followed by the catch block in which the code to handle the exception is written.
- In the catch block, the programmer can also write the code to recover from the exception and can also print the exception.
- A catch block is executed only if it catches the exception coming from within the corresponding try block.

Option I – try, catch & finally

Syntax:

```
try{

        // The code that is prone to throw exception
}
catch(Exception exception){
        // What to do if this exception is thrown
}
finally{
        //to be done whether exception is thrown or not
        (Will be explained in next slide)
}
```

```
public void divide(int a,int b){
        int quotient=0;
        try{
                quotient=a/b;
        }
        catch(ArithmeticException exceptio
                System.out.println("Exception
        }
        finally{
                System.out.println("The quotie
        }
}
```

Here, there i
an Arithmeti
thrown

finally block

Features of Finally block:

- The finally block will execute whether or not an exception is thrown.
- Finally block can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method.
- The finally block is optional.
- If a finally block is associated with a try, the finally block will be executed upon conclusion of the try
- A try can contain multiple catch block and only one finally block.

Execution flow in try, catch

```
public divide(int dividend, int divisor){
try{
        result = dividend/ divisor;
        // other statements..
}
Catch(ExceptionObject)
{
        // Exception handled
}
        // other statements..
}
```

If exception raised.

Exception Caught

Exception handled

Other statements till the end of the method will be triggered.

Execution flow try, catch & finally

```
public divide(int dividend, int divisor){
  try{
          result = dividend/ divisor;
          //other statements..
  }
  Catch(Exception Object)
  {
          //Exception handled
  }
  finally
  {
          //some logic
  }
          //other statements..
}
```

If exception raised.

Exception Caught

Exception handled

Finally block statements invoked

Other statements till the end of the method will be triggered.

Execution flow when NO exception raised in try, catch & finally block

```
public divide(int dividend, int divisor){
try{
        result = dividend/ divisor;
        //other statements..
}
Catch(ExceptionObject)
{
        //Exception handled
}
finally
{
        //some logic
}
        //other statements..
}
```

If NO exception raised in the method

Finally block statements invoked

Other statements till the end of the method will be triggered.

Multiple catch blocks

Multiple catch blocks:

One block of code can generate different types of exception.

Example:

```
try {
    int den = Integer.parseInt(args[0]);
    System.out.println(3/den);
}
```

This can be h

Example of Multiple Catch blocks:

```
try{
        int den = Integer.parseInt(args[0]);
        System.out.println(3/den);
}catch(ArrayIndexOutOfBoundsException ab){
        // Exception a handled here
}catch(Arithmetic Exception ar){
        // Exception b handled here
}
```

Based on the exception thrown in the try block, the appropriate catch block will be executed

Multiple Exception Instance

The catch clause specifies the types of exceptions that the block can handle, and each exception type is separated with a vertical bar (|)

Example of Multiple exception instance,

```
try{
        int den = Integer.parseInt(args[0]);
        System.out.println(3/den);
}catch(ArrayIndexOutOfBoundsException | Arithmetic Exception e){
        // Exception a handled here
}
```

Nested try blocks

Nested Try Blocks:

Nested try can be used when we want to catch exceptions for specific lines of code

Example of Nested Try block:

```
try{
        int den = Integer.parseInt(args[0]);
        try{
                System.out.println(3/den);
        } catch(ArithmeticException ar){
                // Exception a handled here
        }
}catch(ArrayIndexOutOfBoundsException ab){
        // Exception b handled here
}
```

This is the nested TRY. Arithmetic Exception thrown here will be caught inside this block itself

NOTE: If the inner try does not have a matching catch statement for a particular exception, the control is transferred to the outer try statement's catch handlers where it searches for a matching catch statement

Rules for try, catch & finally

Rules for writing the try-catch-finally:

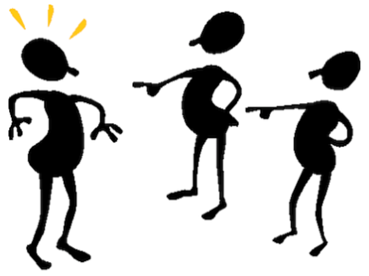
- The try block must be followed by either a catch block or a finally block, or both.
- The try block by itself is not complete.
- Any catch block must immediately follow a try block.
- The finally block must immediately follow the last catch block, or the try block if there is no catch block.

Option II – throws

Alternate way of handling exceptions:

Hope you remember how the guardian delegated

the responsibility back to the parents.



In exception handling, this delegation is done by throwing the exception.

Throws keyword is used to throw exception object from a method to the calling method.

The exception thrown by the method needs to be handled by the calling method.

Syntax - throws

Syntax:

```
<access specifier><return type><method name>() throws Exception-list
{
        //some code here which can throw
        //any type of exception specified in Exception-list
}
```

Exception-list is a comma-separated list of the exceptions that a method can throw.

The method invoking this method should handle the exceptions in the exception list using try catch block.

Example – try/catch, throws

Step 1: Create a class, *Demo* with a method, *division* with two int parameters

- Dividend
- Divisor

This method should divide the dividend by divisor and return the result.

This method should also throw an *ArithmeticException* to the calling method.

Step 2: Create a class, *ThrowsDemo* with a main method

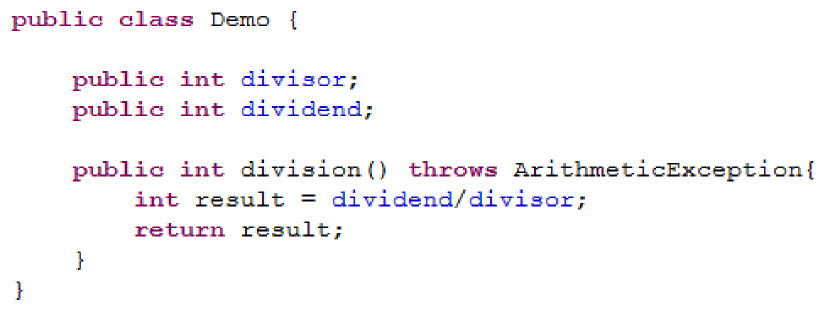
- The main method should invoke the division method in *Demo* class.
- The main method should also *catch* the *ArithmeticException* thrown by the division method and print the Exception "Arithmetic Exception is Thrown"
- The try/catch block should also have a finally block which prints a message "The result is" <Result>

Lets develop the program to demonstrate throws, try/catch.

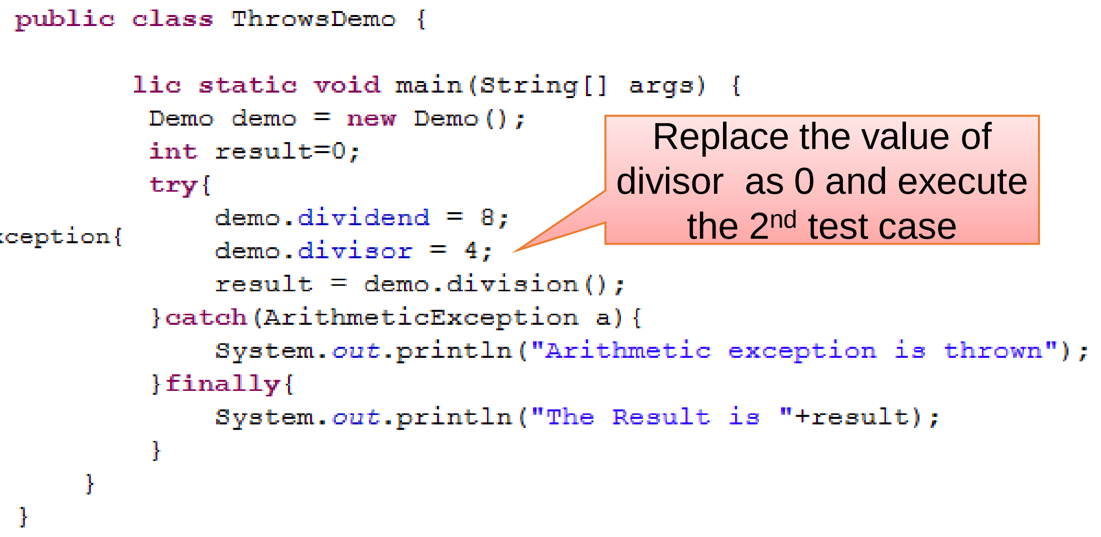
Solution - try/catch, throws

Execute the divide method for two inputs

Dividend	Divisor	What did you notice?
12	3	Finally block executed and result printed as 4.
12	0	Arithmetic exception thrown, exception printed , finally block executed and result printed as ‘0’



```
public class Demo {

    public int divisor;
    public int dividend;

    public int division() throws ArithmeticException{
        int result = dividend/divisor;
        return result;
    }
}
```



```
public class ThrowsDemo {

    lic static void main(String[] args) {
     Demo demo = new Demo();
     int result=0;
     try{
         demo.dividend = 8;
         demo.divisor = 4;
         result = demo.division();
     }catch(ArithmeticException a){
         System.out.println("Arithmetic exception is thrown");
     }finally{
         System.out.println("The Result is "+result);
     }
   }
}
```

Replace the value of divisor as 0 and execute the 2nd test case

throw

Java allows you to **throw Exceptions** and generate Exceptions. An Exception you throw is an **Object**

Syntax:

throw <Exception Object>

Throw should be used in conjunction with ***throws*** clause. So if a method uses throw keyword it should either,

- Surround the ***throw*** statement with ***try/catch*** block and catch the exception thrown **(or)**
- Declare the ***throws*** clause in the methods signature for the exception thrown.

How to create a Exception Object?

```
throw new ArithmeticException("Id not found");
```

When to use throw?

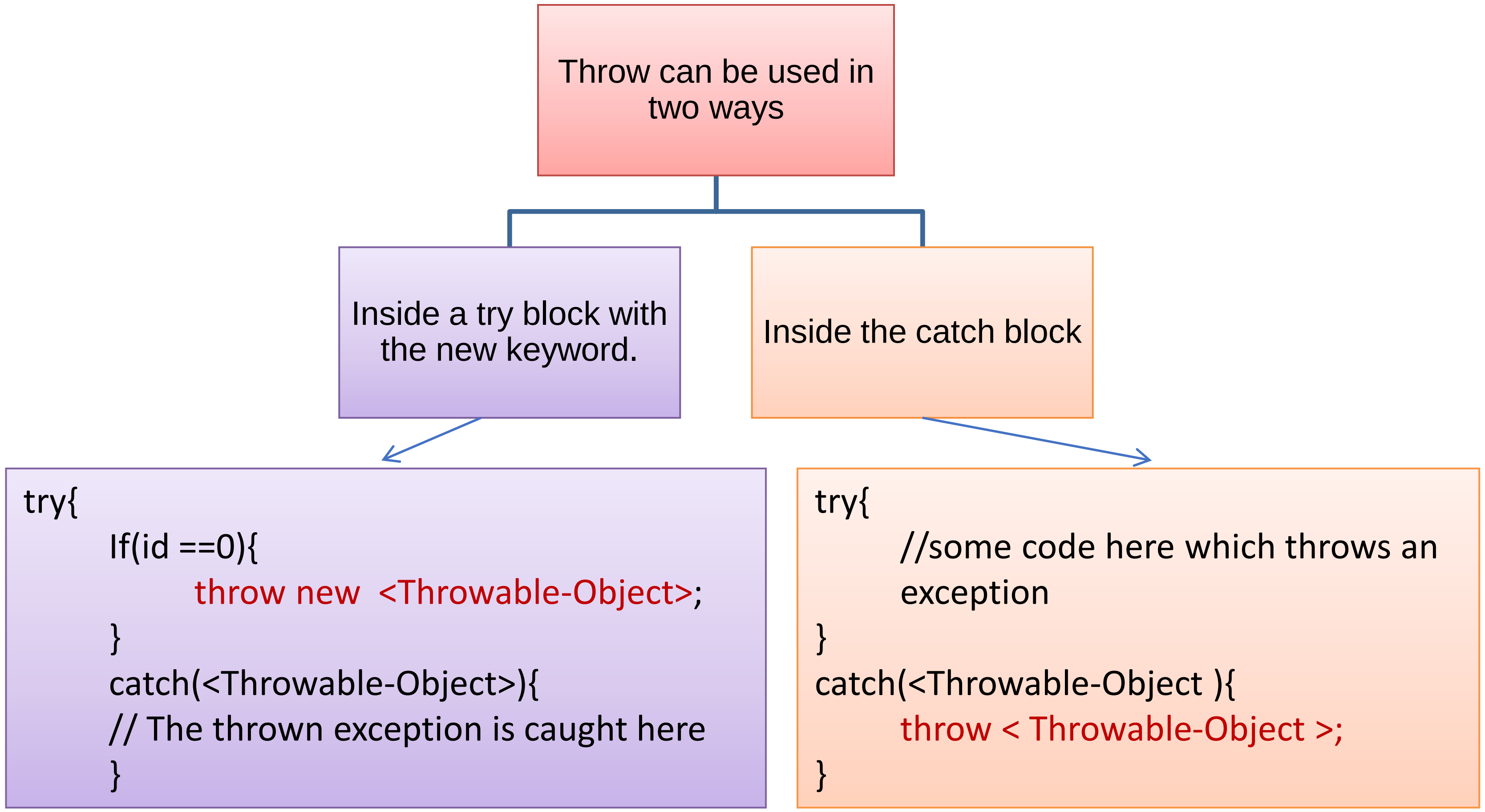
Used for throwing Throwable objects under undesired circumstances in a program and is used for throwing exception explicitly.

- User manually checks if the file is present, if the file is not present, user can throw a FileNotFoundException.
- Used to re-throw the caught exception object in catch block.

Example:

- User catches a FileNotFoundException
- Performs some logic and re-throws it as IOException (or)
- Performs some logic and throw it as FileNotFoundException itself with some additional information.

Usage of throw


Throw can be used in two ways
Inside a try block with the new keyword.
Inside the catch block

```
try{
        If(id ==0){
                throw new  <Throwable-Object>;
        }
        catch(<Throwable-Object>){
        // The thrown exception is caught here
        }
```

```
try{
        //some code here which throws an
        exception
}
catch(<Throwable-Object ){
        throw < Throwable-Object >;
}
```

Example – throw

Step 1: In the Validate class created the validate method that takes integer value as a parameter.

- If the age is less than 18, we are throwing the ArithmeticException with message "Not allowed" otherwise print a message **Welcome to vote**.
- This method should throw this ***ArithmeticException**.*

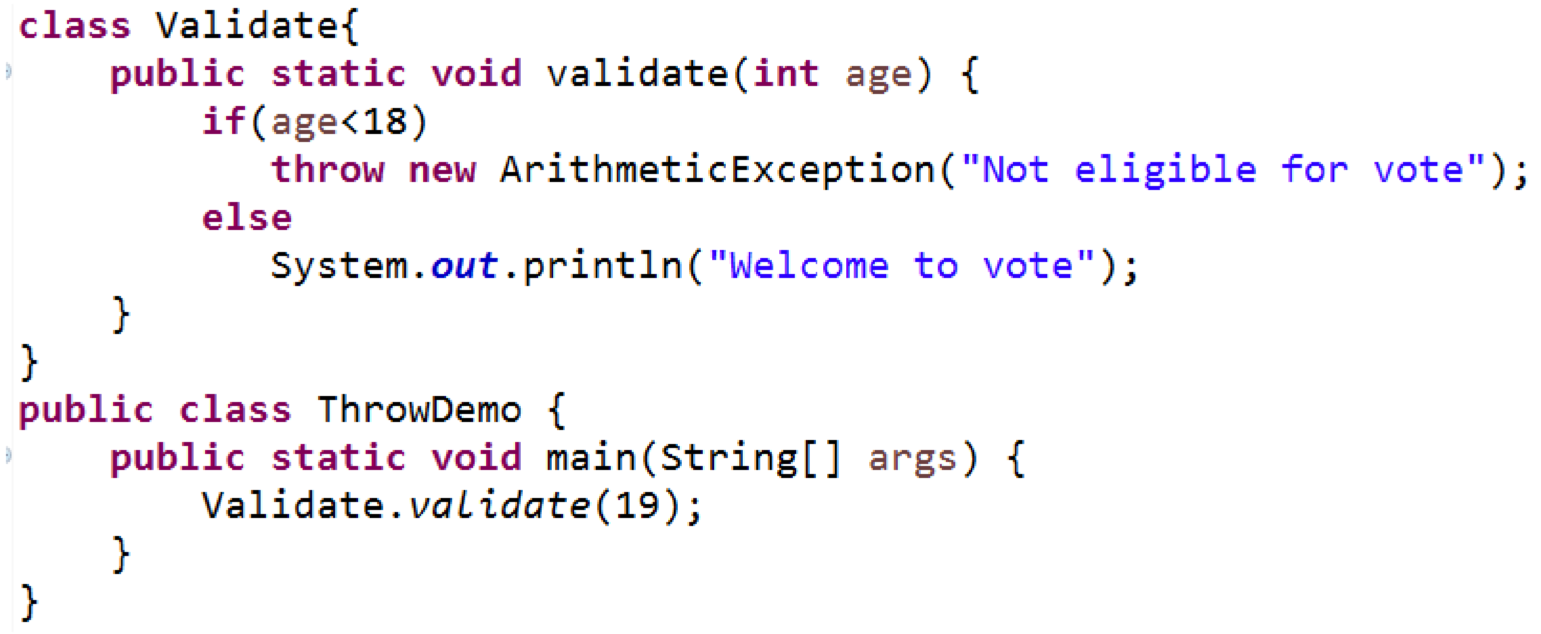
Step 2: The exception thrown needs to be handled in ***ThrowDemo***

- The main method should *catch* the ***ArithmeticException*** thrown by the validate method and print the Exception and print the message in the exception Object.
- The try/catch block should also have a finally block which prints a message "Message from validate"
  <Result>

Solution - throw

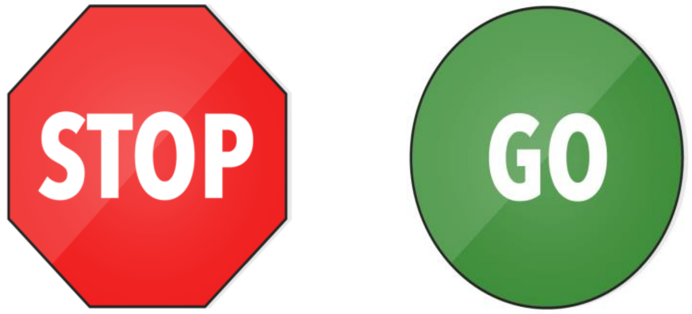
Execute the validate method for age input

age	What did you notice?
19	**Finally block executed and result printed as Welcome to vote**
14	**Arithmetic exception thrown, exception message "Not eligible for vote", finally block executed and result printed.**



```
class Validate{
    public static void validate(int age) {
        if(age<18)
           throw new ArithmeticException("Not eligible for vote");
        else
           System.out.println("Welcome to vote");
    }
}
public class ThrowDemo {
    public static void main(String[] args) {
        Validate.validate(19);
    }
}
```

Time To Reflect



Trainees to reflect the following topics before proceeding.

- What is an Exception?
- What are the types of Exceptions?
- What is an exception handler?
- What are techniques of Handling Exceptions?
- Can I have a try block alone?
- Where will I use finally block?
- What is the keyword used to manually throw the exception?

Exception – Case study

Mr. Hari with problem with development !!!

- Hari has three classes where each class calls the method in the next class.
- The third class needs to check if the employee name and employee designation is empty.
- If the fields are empty then appropriate exception needs to be thrown back to the first class.
- Hari is confused on which Exception object to throw back since there is no exception object that depicts empty values!!!
- How does Hari solve this problem?

Using custom Exceptions

User defined exception

User Defined Exceptions are custom exceptions which are created by programmers to handle the various application specific errors.

Example: In a banking application the developers can create the following exceptions in the specified scenario,

- ***InvalidAccountNumberException*** - Thrown when the account number entered by the user is wrong.
- ***AccountInactiveException*** – Exception thrown when User trying to operate an account which has become inactive.
- ***InsufficientFundException*** - Exception thrown when user trying to transfer amount with insufficient funds.

How to create user defined exception

Step 1: Create a Java class which extends the Exception class.

Step 2: Override the necessary constructors.

Scenario: Assume an application has a business logic of validating the age entered by the user, the programmer may create a user defined exception named "InvalidAgeException" and throw it when the validation fails.

```java
public class AgeValidationException extends Exception {

    public AgeValidationException(final String message) {
        super(message);
    }
    public AgeValidationException(final Throwable exception) {
        super(exception);
    }
    public AgeValidationException(final String message,
            final Throwable exception) {
        super(message, exception);
    }
}
```

Step 1: *Create a java class extending Exception*

Step 2: *Override the necessary constructors.*

How can you throw the exception?

Assume in the application if the user age is < 18 he should not be allowed to register in the site.

```
public void registerProfile() throws AgeValidationException
{
    try{
        if(age< 18){
            throw new AgeValidationException("User Age is not eligible");
        }
    }
}
```

NOTE: The method throwing the exception should also declare the exception in ***throws*** clause.

Example – User Defined Exception

After this demo you will be able to create custom Exceptions and understand how to throw them.

Scenario: A shopping portal provides users to register their profile. During registration the system needs to validate the user age above 18 and should be placed in India. If not the system should throw an appropriate error.

1. Create a user defined exception classes named "***InvalidCountryException***" & "***InvalidAgeException***"
2. Overload the respective constructors.
3. Create a main class "**UserRegistration**" , add the following method,

- ***registerProfile*** - The parameter are String userName , int age, String country. Add the following logic
- *if country is not equal to "India" throw a **invalidCountryException** with error message "User Outside India cannot be registered"*
- *If age < 18 throw a **InvalidAgeException** with error message "User is a Minor"*
- *Invoke the method **registerProfile*** from the main method with the data specified and see how the program behaves,

Solution – User Defined Exception

Create the exception class as mentioned below.

```
public class InvalidCountryException extends Exception{


    public InvalidCountryException(final String message){
        super(message);
    }
    public InvalidCountryException(final Throwable exception)
        super(exception);
    }

    public InvalidCountryException(final String message,
            final Throwable exception){
        super(message,exception);
    }
}
```

```
public class InvalidAgeException extends Exception{

    public InvalidAgeException(final String message){
        super(message);
    }
    public InvalidAgeException(final Throwable exception){
        super(exception);
    }

    public InvalidAgeException(final String message,
            final Throwable exception){
        super(message,exception);
    }

}
```

Execute the program for the following scenarios.

Name	Age	Country	Output
Hari	**7**	**India**	**InvalidAgeException should be thrown.** ***The error message should be "User is a Minor"***
Saro	**23**	**Australia**	***InvalidCountryException should be thrown. The error message should be "User Outside India cannot be registered"***

Solution – User Defined Exception

Create the user registration program as mentioned below.

```
public class UserRegistration {

    public void registerProfile(String userName,int age,String country)
    throws InvalidCountryException,InvalidAgeException{
        if(!"India".equals(country)){
            throw new InvalidCountryException("User Outside India cannot be registered");
        }
        if(age<18){
            throw new InvalidAgeException("User is a Minor");
        }

    }
    public static void main(String[] args) {
        UserRegistration userReg = new UserRegistration();
        try{
            userReg.registerProfile("John", 13, "India");
        }catch(InvalidCountryException ice){
            System.out.println(ice.getMessage());
        }catch(InvalidAgeException iae){
            System.out.println(iae.getMessage());
        }
    }
}
```

Thank you

You have successfully completed

Exception Handling