CSE 220: Object Oriented Sessional - II

Introduction to JAVA

Shahriar Rahman Khan
Lecturer, Dept of CSE
MIST

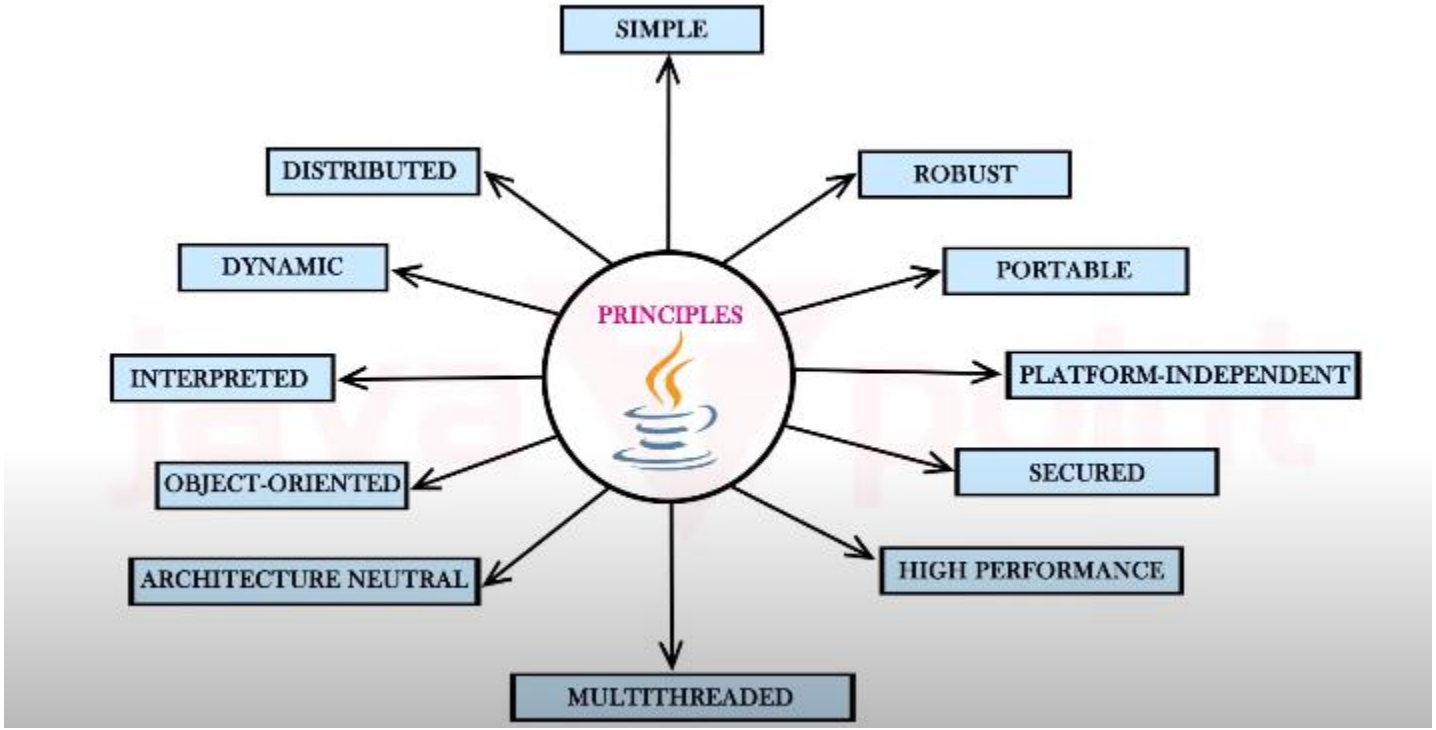
History

- Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time.
- Java was originally developed by Sun Microsystems starting in 1991
  – James Gosling
  – Patrick Naughton

History

- This language was initially called **Oak**
- Renamed **Java** in 1995 due to copyright issue.
- Java is an island in Indonesia where the first coffee was produced (called Java coffee).
- Java name was chosen by James Gosling while having a cup of coffee nearby his office.

What is Java?


SIMPLE
DISTRIBUTED
ROBUST
DYNAMIC
PORTABLE
PRINCIPLES
INTERPRETED
PLATFORM-INDEPENDENT
OBJECT-ORIENTED
SECURED
ARCHITECTURE NEUTRAL
HIGH PERFORMANCE
MULTITHREADED

What is Java?

- A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language

-- Sun

Simple

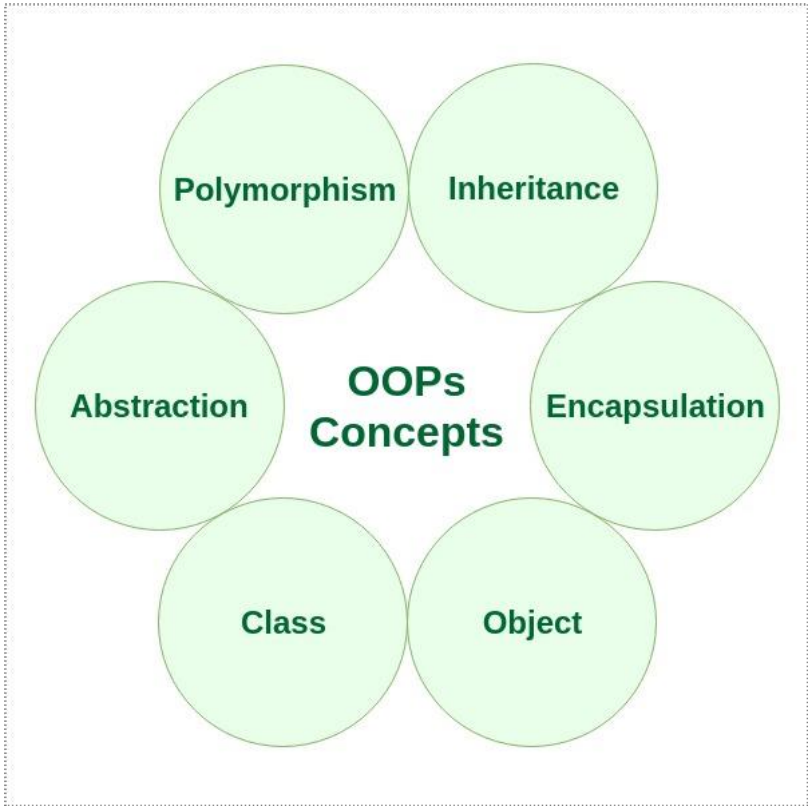
- Easy to learn
- Syntax is Clean, simple and easy to understand
- Java syntax is based on C++ (Easier for programmers who learn it after C++)
- No need to remove unreferenced objects as there's an automatic garbage

What is Java?

Object-Oriented

- Strongly supports the concepts of Object-Oriented Programming, so it is called a pure object-oriented language.
- No free functions
- Supports major Object-Oriented programming features like Encapsulation, Abstraction, and Inheritance.
- For example, we cannot develop an executable program in Java without making use of the class.

What is Java?


Polymorphism
Inheritance
Abstraction
OOPs Concepts
Encapsulation
Class
Object

What is Java?

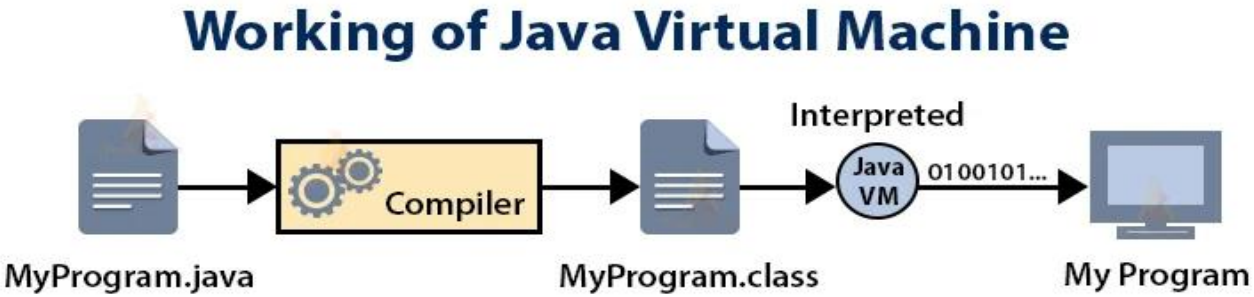
Distributed

- Java encourages users to create distributed applications. In Java, we can split a program into many parts and store these parts on different computers. A Java programmer sitting on a machine can access another program running on the other machine.
- Java comes with an extensive library of classes for interacting, using TCP/IP protocols such as HTTP and FTP, which makes creating network connections much easier than in C/C++.
- It also enables multiple programmers at many locations to work together on a single project.

What is Java?

Interpreted

- Usually, a computer language can be either compiled or interpreted. Java integrates the power of Compiled Languages with the flexibility of Interpreted Languages.
- Java compiler (javac) compiles the java source code into the bytecode.
- Java Virtual Machine (JVM) then executes this bytecode which is executable on many operating systems and is portable.



What is Java?

Robust

– Java is simple

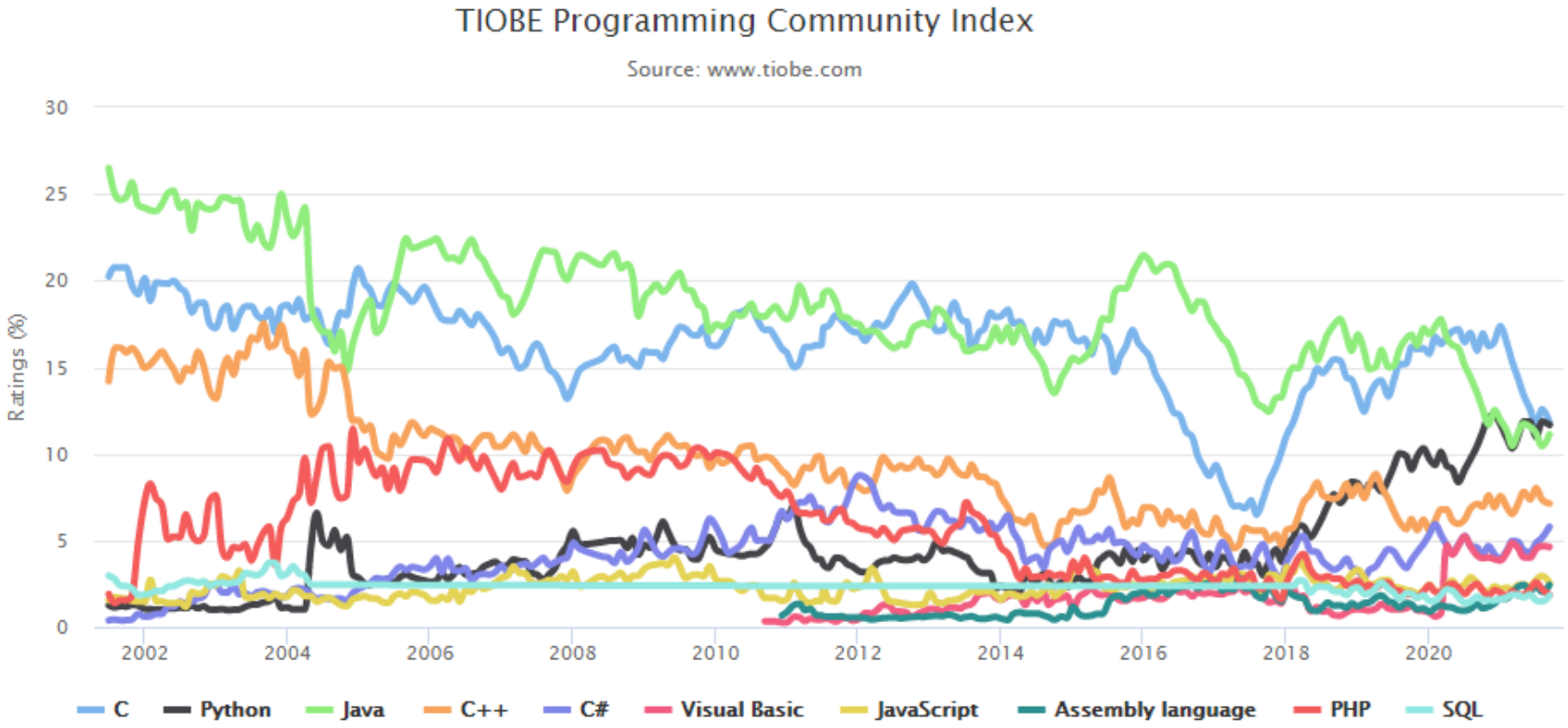
– No pointers/stack concerns

– Exception handling

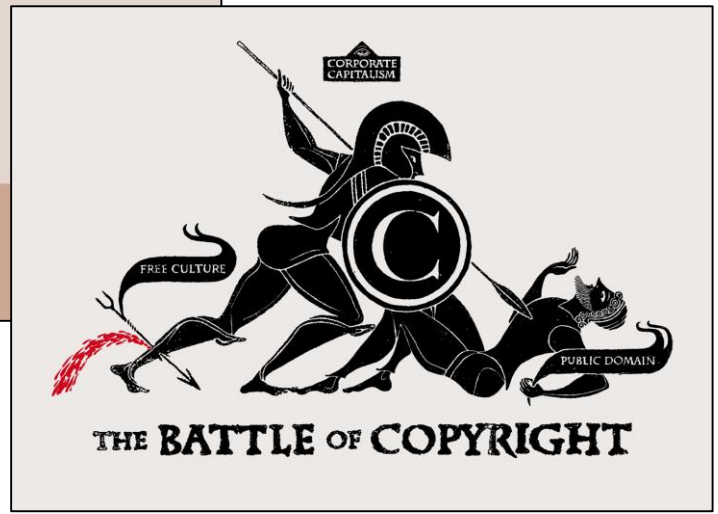
– Try/catch/finally series allows for simplified error recovery

– Many errors caught during compilation. Helps in eliminating errors as it checks the code during both compile and runtime.

Which one is the most popular Programming Language?



Java – The Most Controversial



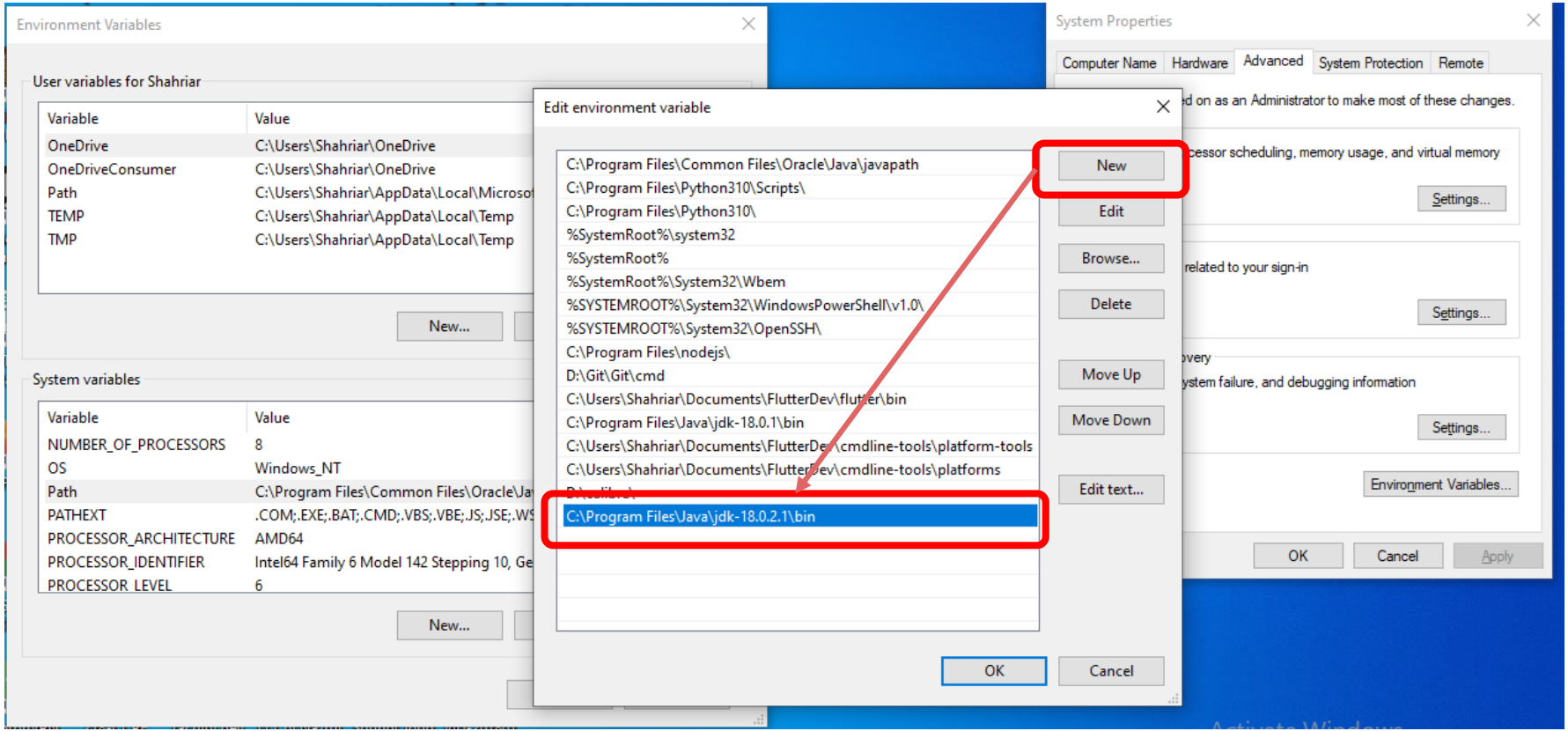
Java Editions

- **Java 2 Platform, Standard Edition (J2SE)**
  – Used for developing Desktop based application and networking applications
- **Java 2 Platform, Enterprise Edition (J2EE)**
  – Used for developing large-scale, distributed networking applications and Web-based applications
- **Java 2 Platform, Micro Edition (J2ME)**
  – Used for developing applications for small memory constrained devices, such as cell phones, pagers and PDAs

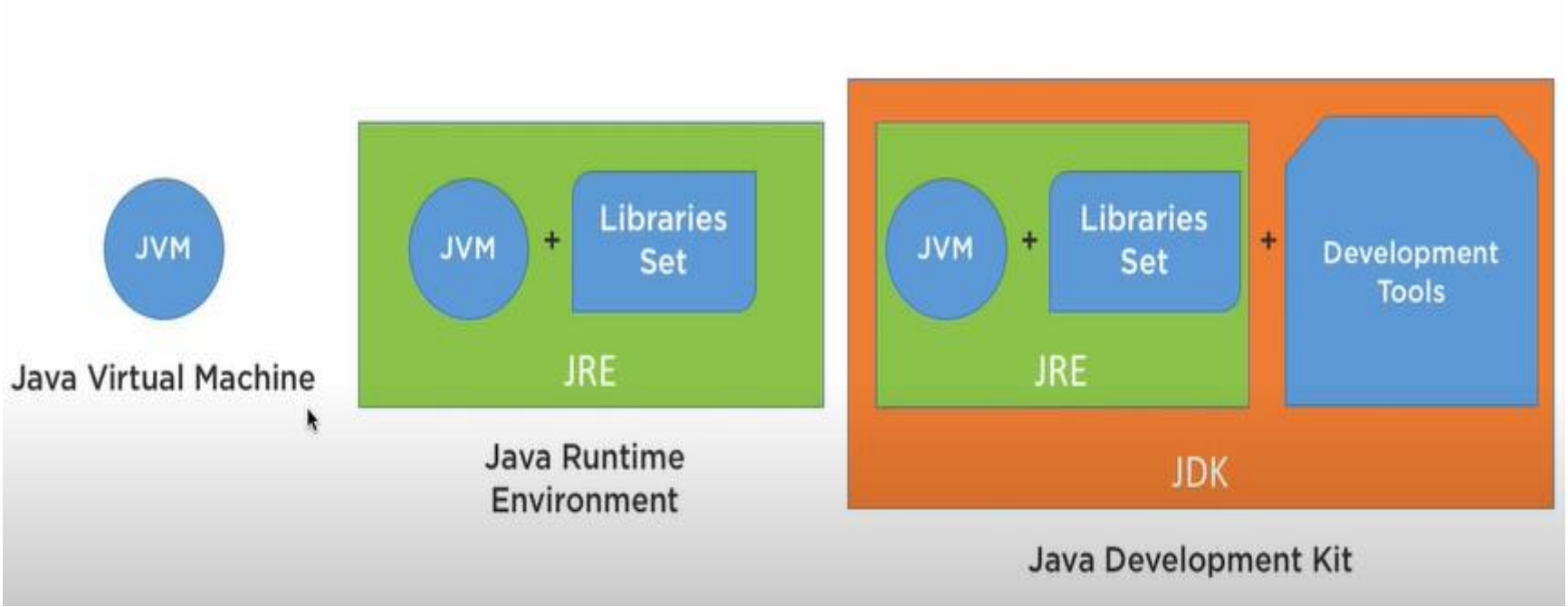
Java Installation

- Go to https://www.oracle.com/java/technologies/downloads/
- Choose Java SE Development Kit 18.0.2.1 (x64 Installer) for windows os
- After downloading, install it in program files.
- Go to C:\Program Files\Java\jdk-xx.x.x.x\bin and copy the path. For example: C:\Program Files\Java\jdk-18.0.2.1\bin
- Now, Go to System Environment Variables and set the path in system variable.

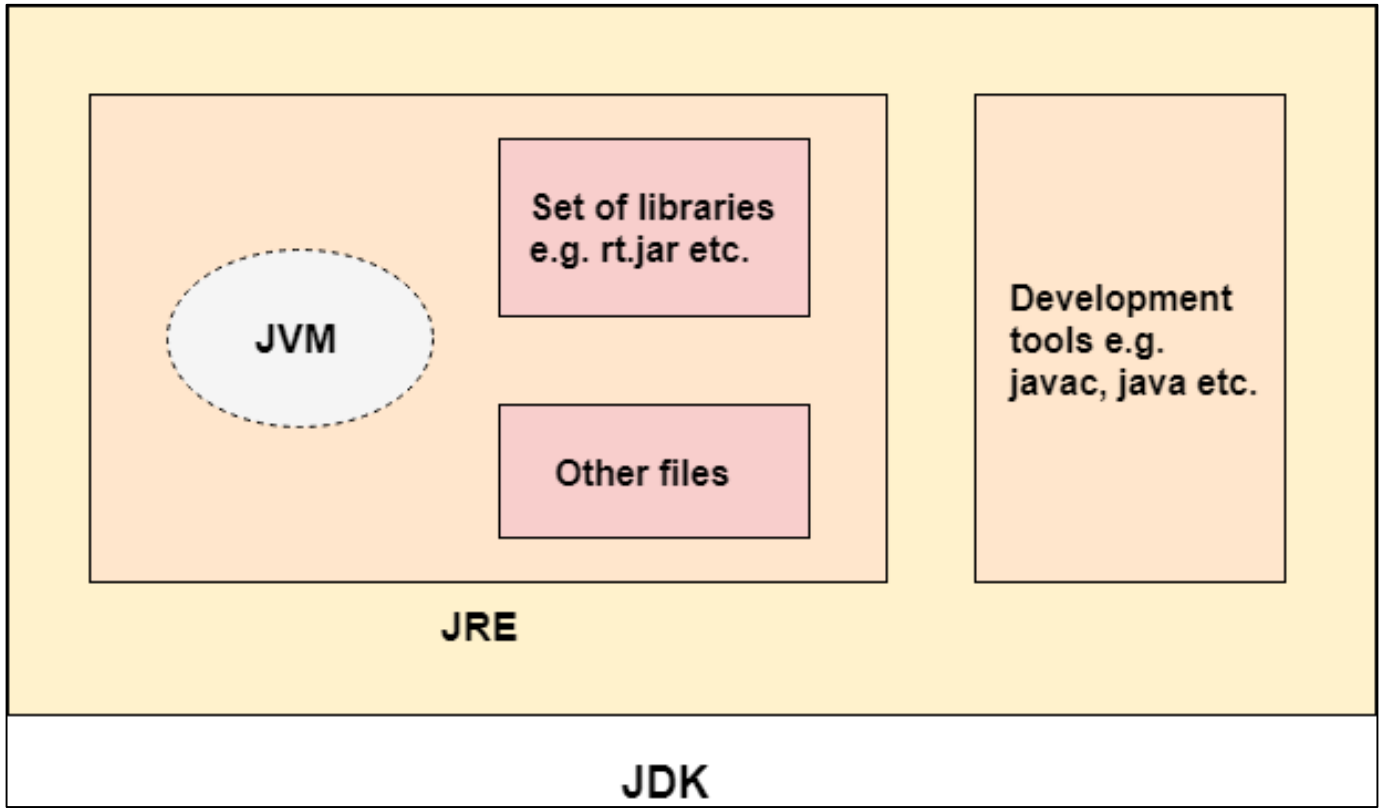
Java Installation



Difference Between JDK JRE JVM



Difference Between JDK JRE JVM



Write Once, Run Anywhere

Java Platform

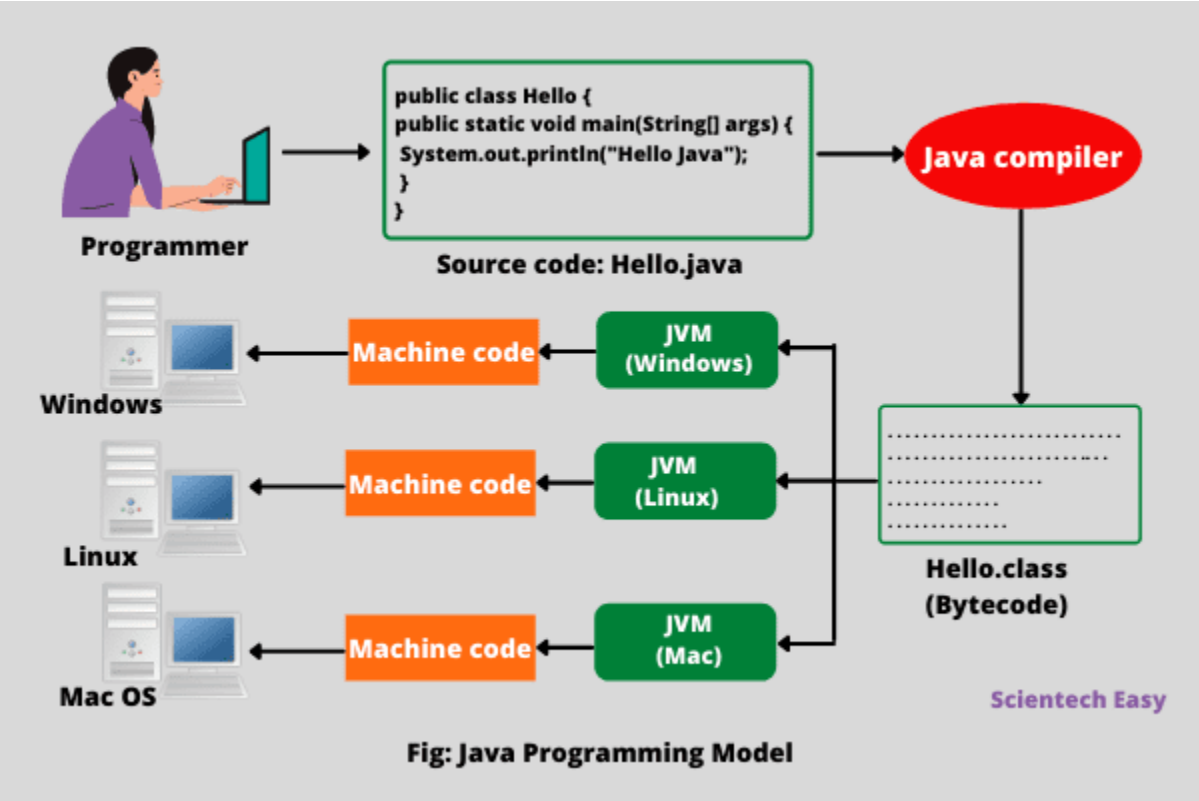


Fig: Java Programming Model

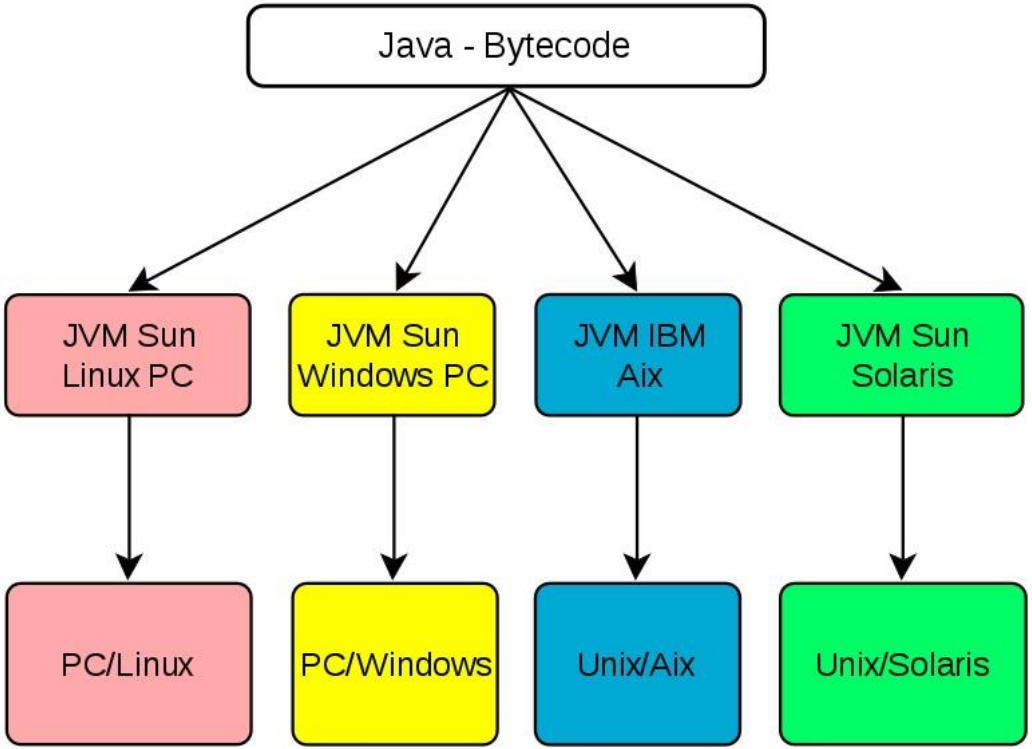
Bytecode

- They are not machine language binary code
- They are independent of any particular microprocessor or hardware platform
- They are platform-independent instructions
- Another entity (interpreter) is required to convert the bytecodes into machine codes that the underlying microprocessor understands
- This is the job of the **JVM** (Java Virtual Machine)

Java Virtual Machine (JVM)

- It provides Runtime Environment in which Java bytecode can be executed.
- Tasks of JVM
  - Loads Code
  - Verifies Code
  - Executes Code
  - Provides Runtime Environment
- JVM is platform dependent. For each Software or Hardware, we have different JVM configuration.

Java Virtual Machine (JVM)



Java Virtual Machine (JVM)

- The same bytecodes can be executed on any platform containing a compatible JVM
- The JVM is invoked by the java command

  – *java* ***Welcome***
- It searches the class Welcome in the current directory and executes the main method of class ***Welcome.***
- It issues an error if it cannot find the class ***Welcome*** or if class Welcome does not contain a method called main with proper signature

Phase 1: Creating a Program

- Any text editor or Java IDE (Integrated Development Environment) can be used to develop Java programs
- Java source-code file names must end with the ***.java*** extension
- Some popular Java IDEs are
  - NetBeans
  - **Eclipse**
  - JCreator
  - IntelliJ

Phase 2: Compiling a Java Program

- ***javac Welcome.java***

  – Searches the file in the current directory

  – Compiles the source file

  – Transforms the Java source code into bytecodes

  – Places the bytecodes in a file named ***Welcome.class***

Phase 3: Loading a Program

- One of the components of the JVM is the class loader
- The class loader takes the *.class* files containing the programs bytecodes and transfers them to RAM
- The class loader also loads any of the *.class* files provided by Java that our program uses

Phase 4: Bytecode Verification

- Another component of the JVM is the bytecode verifier
- Its job is to ensure that bytecodes are valid and do not violate Java's security restrictions
- This feature helps to prevent Java programs arriving over the network from damaging our system

Phase 5: Execution

- Now the actual execution of the program begins
- Bytecodes are converted to machine language suitable for the underlying OS and hardware
- Java programs actually go through two compilation phases
  - Source code -> Bytecodes
  - Bytecodes -> Machine language

All You Need is Java

Fast-paced Switchover from C++

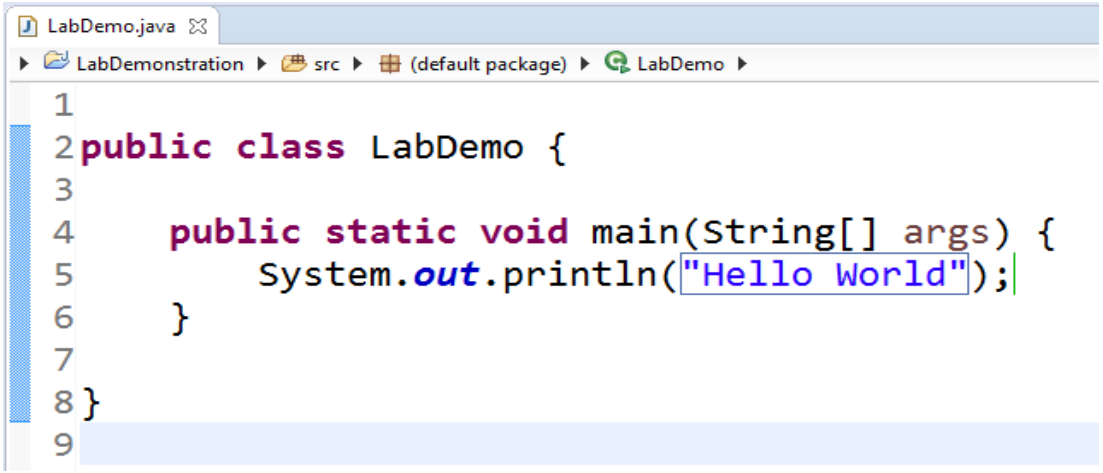
C++ vs Java

Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in Windows-based, web-based, enterprise, and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of the C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed to be easy to use and accessible to a broader audience.
Goto	C++ supports the goto statement.	Java doesn't support the goto statement.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by using interfaces in java.

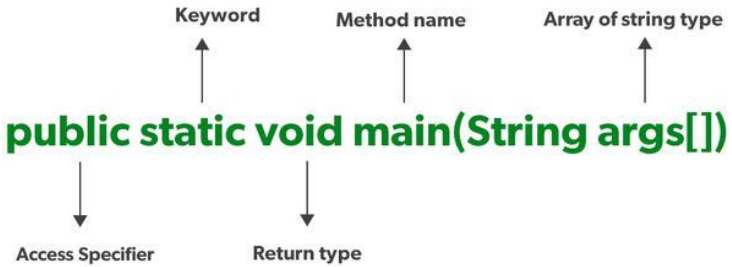
C++ vs Java

Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write a pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.
Compiler and Interpreter	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses both compiler and interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform-independent.
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.

A Hello World Application

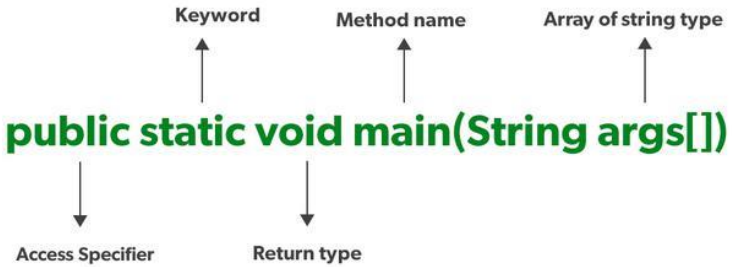


A Hello World Application



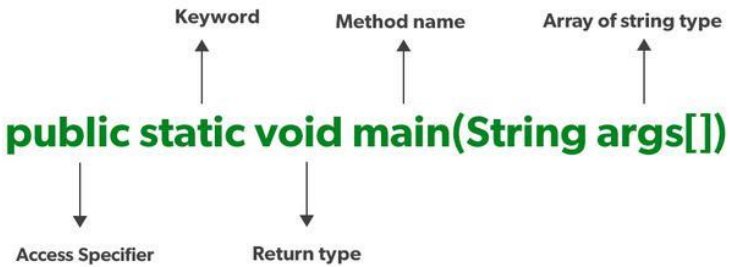
Public: This is the access modifier of the main method. It has to be public so that java runtime can execute this method. Remember that if you make any method non-public then it's not allowed to be executed by any program, there are some access restrictions applied. So it means that the main method has to be public.

A Hello World Application



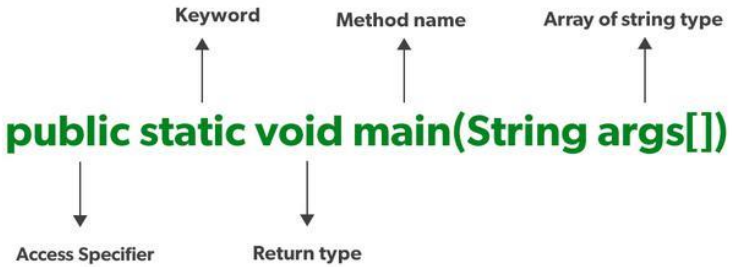
Static: It is a keyword which is when associated with a method, makes it a class related method. The main() method is static so that JVM can invoke it without instantiating the class. This also saves the unnecessary wastage of memory which would have been used by the object declared only for calling the main() method by the JVM. When java runtime starts, there is no object of the class present. That's why the main method has to be static so that JVM can load the class into memory and call the main method.

A Hello World Application



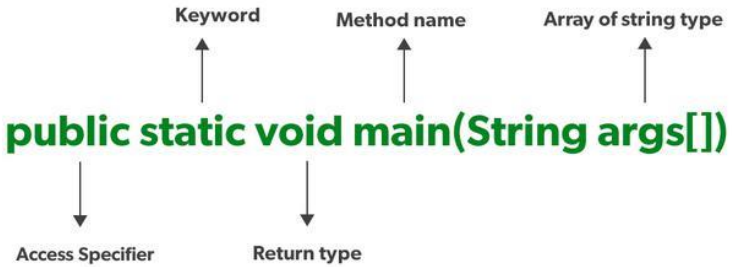
Void: It is a keyword and used to specify that a method doesn't return anything. As main() method doesn't return anything, its return type is void. As soon as the main() method terminates, the java program terminates too. Hence, it doesn't make any sense to return from main() method as JVM can't do anything with the return value of it.

A Hello World Application



main: It is the name of Java main method. It is the identifier that the JVM looks for as the starting point of the java program. It’s not a keyword.

A Hello World Application



String[] args: It stores Java command line arguments and is an array of type java.lang.String class. Here, the name of the String array is args but it is not fixed and user can use any name in place of it. Java main method accepts a single argument of type String array. This is also called as java command line arguments.

Data-types

Integers*: **byte (1 byte), short(2 bytes), int (4 bytes), long(8 bytes)**

Floating-point/ decimal value: **float (4 bytes), double (8 bytes)**

Characters: **char (2 bytes)**

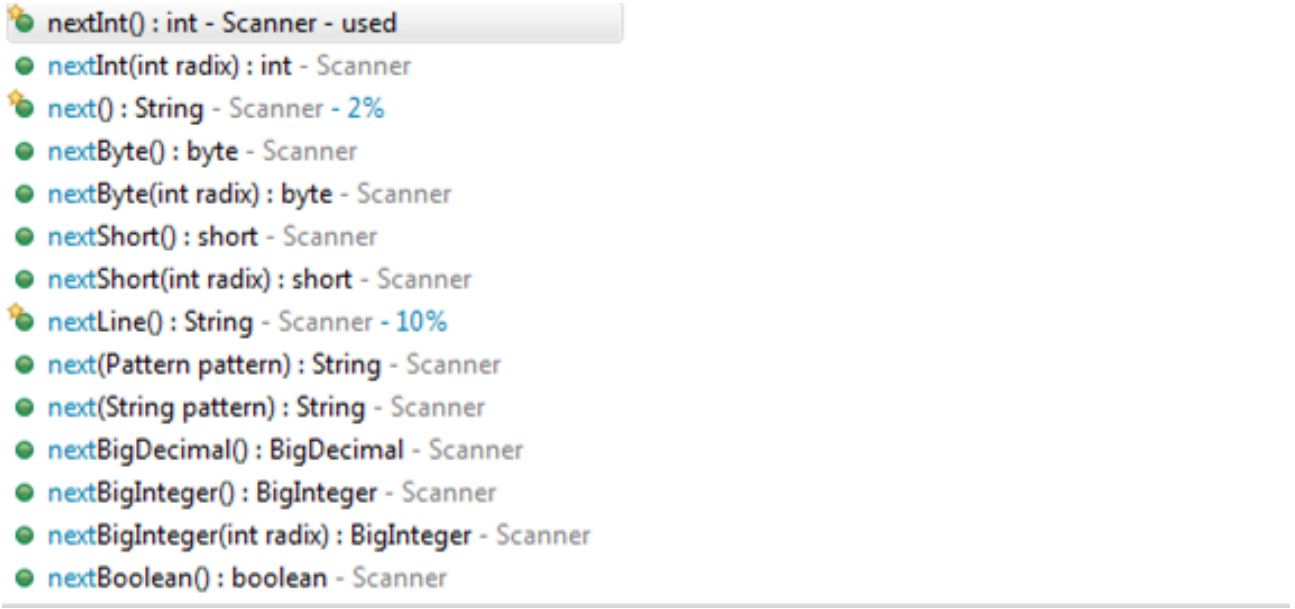
true/ false: **boolean (1 byte)**

*Does not support unsigned (positive only) integers

Taking input from user

```java
import java.util.Scanner;

public class LabDemo {

    public static void main(String[] args) {
        System.out.println("Please enter a number:");


        Scanner ektaScanner = new Scanner(System.in);
        int n = ektaScanner.nextInt();

        System.out.println("Your number was: " + n);
    }

}
```

Taking input from user



Control Statements (1/2)

```java
import java.util.Scanner;

public class LabDemo {

    public static void main(String[] args) {
        System.out.println("Please enter a number:");

        Scanner ektaScanner = new Scanner(System.in);
        int n = ektaScanner.nextInt();

        if (n == 0)
            System.out.println("The number is zero");
        else if (n > 0)
            System.out.println(n + " is positive number");
        else if (n < 0)
            System.out.println(n + " is negative number");
        else
            System.out.println(n + " is not a number");
    }
```

Control Statements (2/2)

```java
import java.util.Scanner;

public class LabDemo {

    public static void main(String[] args) {
        System.out.println("Please enter a number:");

        Scanner ektaScanner = new Scanner(System.in);
        int n = ektaScanner.nextInt();

        for (int i = 0; i<n; i++)
        {
            System.out.println("Testing " + i);
        }
    }
}
```

Arrays

```java
import java.util.Scanner;

public class LabDemo {

    public static void main(String[] args) {
        int marks[];
        marks = new int[30];
        System.out.println(marks.length);   //30
    }
}
```

Arrays

```java
import java.util.Scanner;

public class LabDemo {

    public static void main(String[] args) {
        int marks[] = {1, 2, 3, 4, 5};
        System.out.println(marks.length);   //5
        System.out.println(marks[2]);       //3
    }
}
```

2D Array

```java
import java.util.Scanner;

public class LabDemo {

    public static void main(String[] args) {
        int marks[][] = new int[4][5];
        for (int i = 0; i<4;i++)
            for (int j = 0; j<5; j++)
                marks[i][j] = i*j;

        System.out.println(marks.length);   //4
        System.out.println(marks[0].length);//5
        System.out.println(marks[1][2]);    //2
    }
}
```

String

```
1
2 public class LabDemo {
3
4     public static void main(String[] args) {
5         String s = new String("This is a sample.");
6         System.out.println(s);
7     }
8 }
```

Array of String

```
1
2 public class LabDemo {
3
4     public static void main(String[] args) {
5         String s[] = {"Sunday", "Monday"};
6         System.out.println("MIST is not "
7                 + "closed on " + s[0] + " or " + s[1]);
8
9     }
10 }
```

INHERITANCE

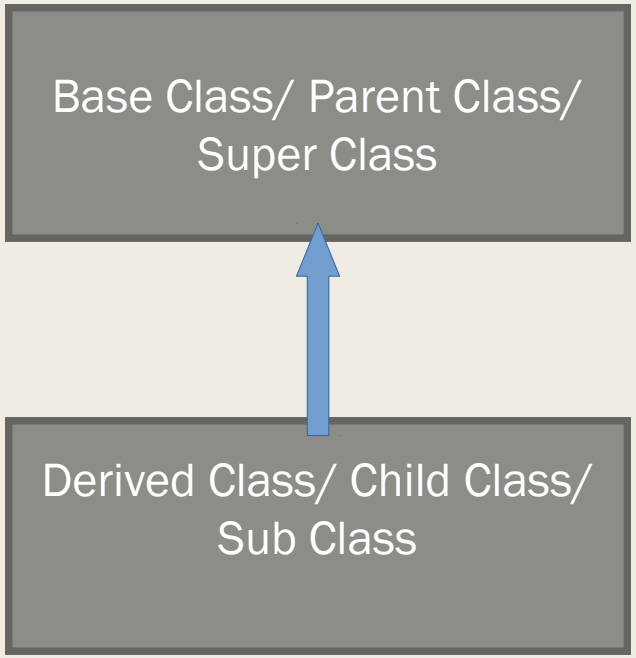
in JAVA

Asst Prof Sharifa Rania Mahmud

Introduction

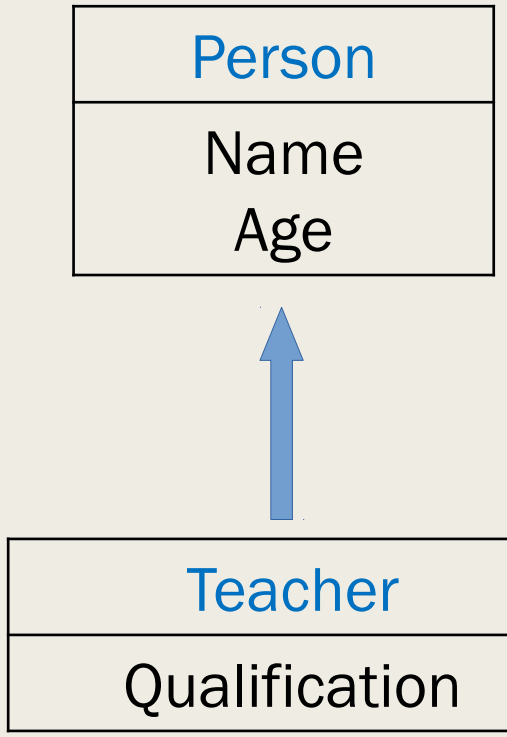
- One of the most important concepts in OOP programming is that of inheritance.
- It is capacity to inherit properties from another class.
- The technique of building new classes from the existing classes is called inheritance.
- Code Reusability is at the heart of inheritance.

Super Class and Sub Class



- Existing class is called the *parent class* or *super class* or *base class.*
- Derived class is called the *child class* or *sub class or derived class*.
- Child class inherits the methods and data defined for the parent class and also can add additional features.

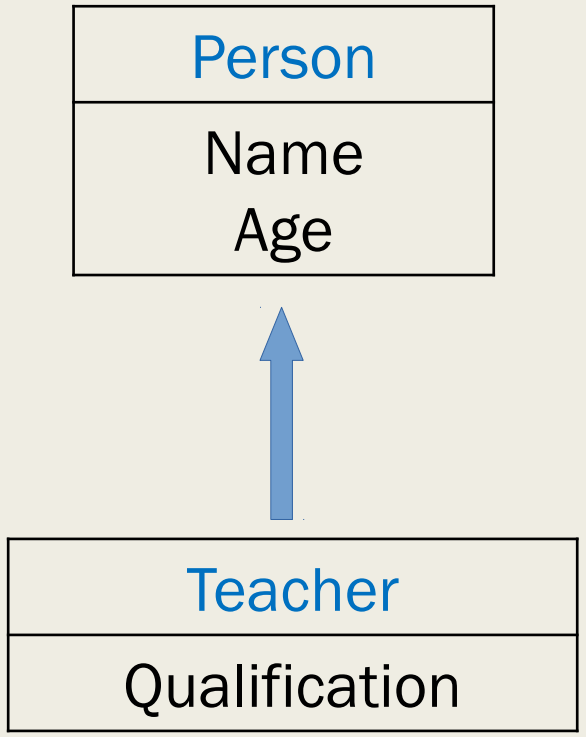
Super Class and Sub Class



parent class / super class / base class

child class / sub class / derived class

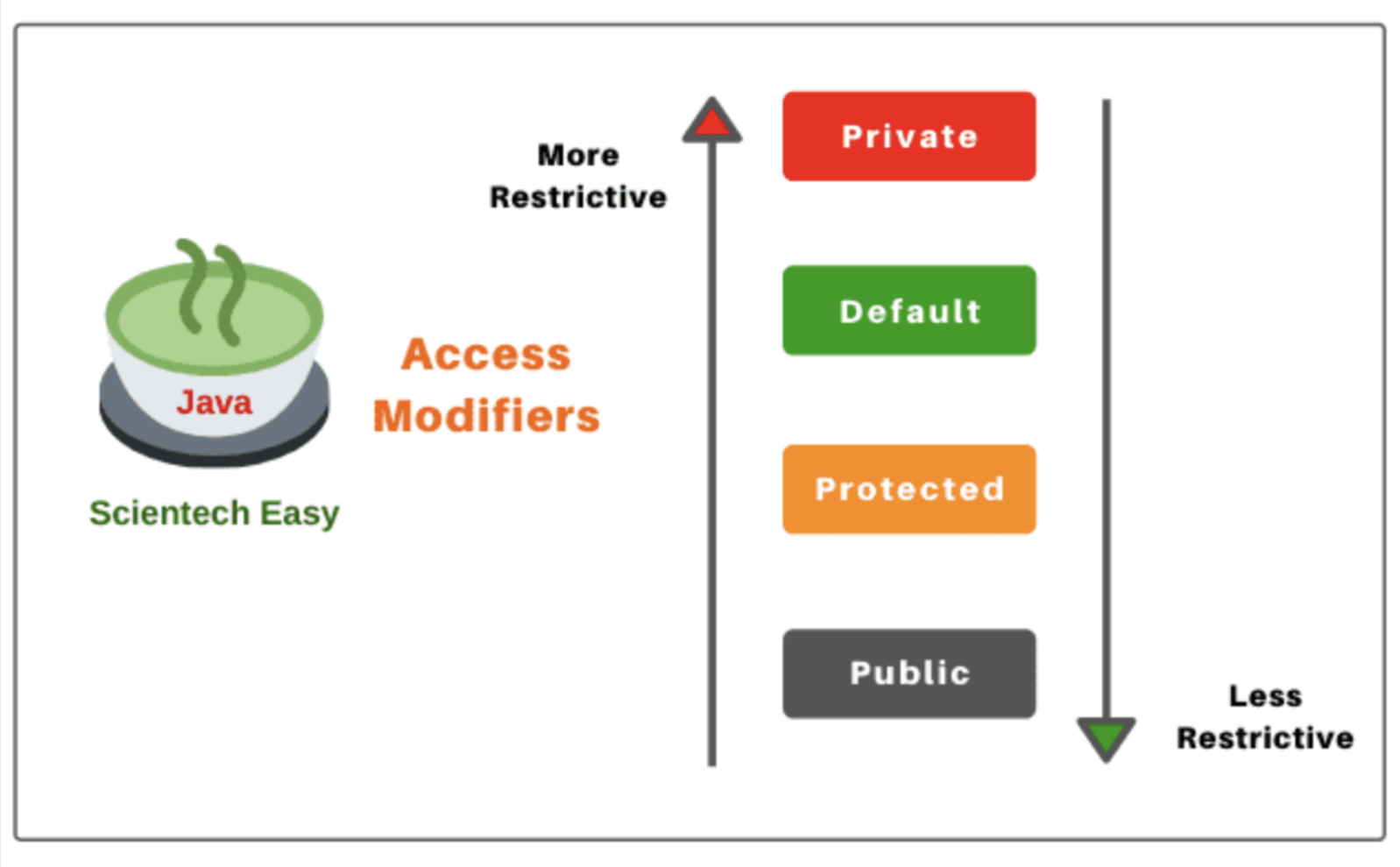
Super Class and Sub Class




```java
class person {

    String name;
    int age;



}
```

```java
class teacher extends person {

    String qualification;

    public static void main(String[] args) {

        teacher t = new teacher();
        t.name = "Mr. A";
        t.age = 40;
        t.qualification = "PhD";

    }

}
```

Access Modifier



Accessibility of Access Modifier

Access Modifier	Accessible by classes in the same package	Accessible by classes in other packages	Accessible by subclasses in the same package	Accessible by subclasses in other packages
Public	Yes	Yes	Yes	Yes
Protected	Yes	No	Yes	Yes
Package (default)	Yes	No	Yes	No
Private	No	No	No	No

- Class can not be declared as “private” and “protected”
- Interface can not be declared as “private”

Access Modifier (subclass in same package)

```java
package inheritance;

class person {

    private String name;
    int age;
    protected String address;
    public String cell;



}
```

```java
package inheritance;

class teacher extends person {

    String qualification;

    public static void main(String[] args) {

        teacher t = new teacher();
        t.name = "Mr. A";       //Error: The field person.name is not visible
        t.age = 40;
        t.address = "Dhaka";
        t.cell = "0987654321";
        t.qualification = "PhD";

    }

}
```

Access Modifier (subclass in diff. package)

```java
package inheritance;

class person {

    private String name;
    int age;
    protected String address;
    public String cell;



}
```

```java
package inheritance1;

import inheritance.person;

public class teacher1 extends person{

        String qualification;

        public static void main(String[] args) {

            teacher1 t = new teacher1();
            t.name = "Mr. A";       //Error: The field person.name is not visible
            t.age = 40;             //Error: The field person.age is not visible
            t.address = "Dhaka";
            t.cell = "0987654321";
            t.qualification = "PhD";

        }
    }
```

“Public” Access Modifier

- Different package
- Non subclass

```java
package inheritance;

public class person {

    public String name;
}
```

```java
package inheritance1;

import inheritance.person;

public class teacher1{      //no sub class

        String qualification;

        public static void main(String[] args) {

            person p = new person();
            p.name = "Mr. A";
            System.out.println("Name: " + p.name);

        }
    }
```

getter and setter method

subclass in same package

```java
package inheritance;

public class person {

    private String name;
    int age;
    protected String address;
    public String cell;

    //getter
    public String getName() {
        return name;
    }

    //setter
    public void setName(String name) {
        this.name = name;
    }
}
```

```java
package inheritance;

class teacher extends person {

    String qualification;

    public static void main(String[] args) {

        teacher t = new teacher();
        t.setName("Mr. A");          //setter
        System.out.println("Name: " + t.getName()); //getter
        t.age = 40;
        t.address = "Dhaka";
        t.cell = "0987654321";
        t.qualification = "PhD";
    }
}
```

getter and setter method (cont..)

```java
package inheritance;

public class person {

    private String name;
    int age;
    protected String address;
    public String cell;

    //getter
    public String getName() {
        return name;
    }

    //setter
    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

}
```

subclass in different package

```java
package inheritance1;

import inheritance.person;

public class teacher1 extends person{

    String qualification;

    public static void main(String[] args) {

        teacher1 t = new teacher1();
        t.setName("Mr. A");                          //setter
        System.out.println("Name: " + t.getName()); //getter
        t.setAge(40);                                //setter
        System.out.println("Age: " + t.getAge());   //getter
        t.address = "Dhaka";
        t.cell = "0987654321";
        t.qualification = "PhD";
    }
}
```

getter and setter method (cont..)

"Public" access modifier (non subclass in different package)

```java
package inheritance;

public class person {

    private String name;

    //getter
    public String getName() {
        return name;
    }

    //setter
    public void setName(String name) {
        this.name = name;
    }
}
```

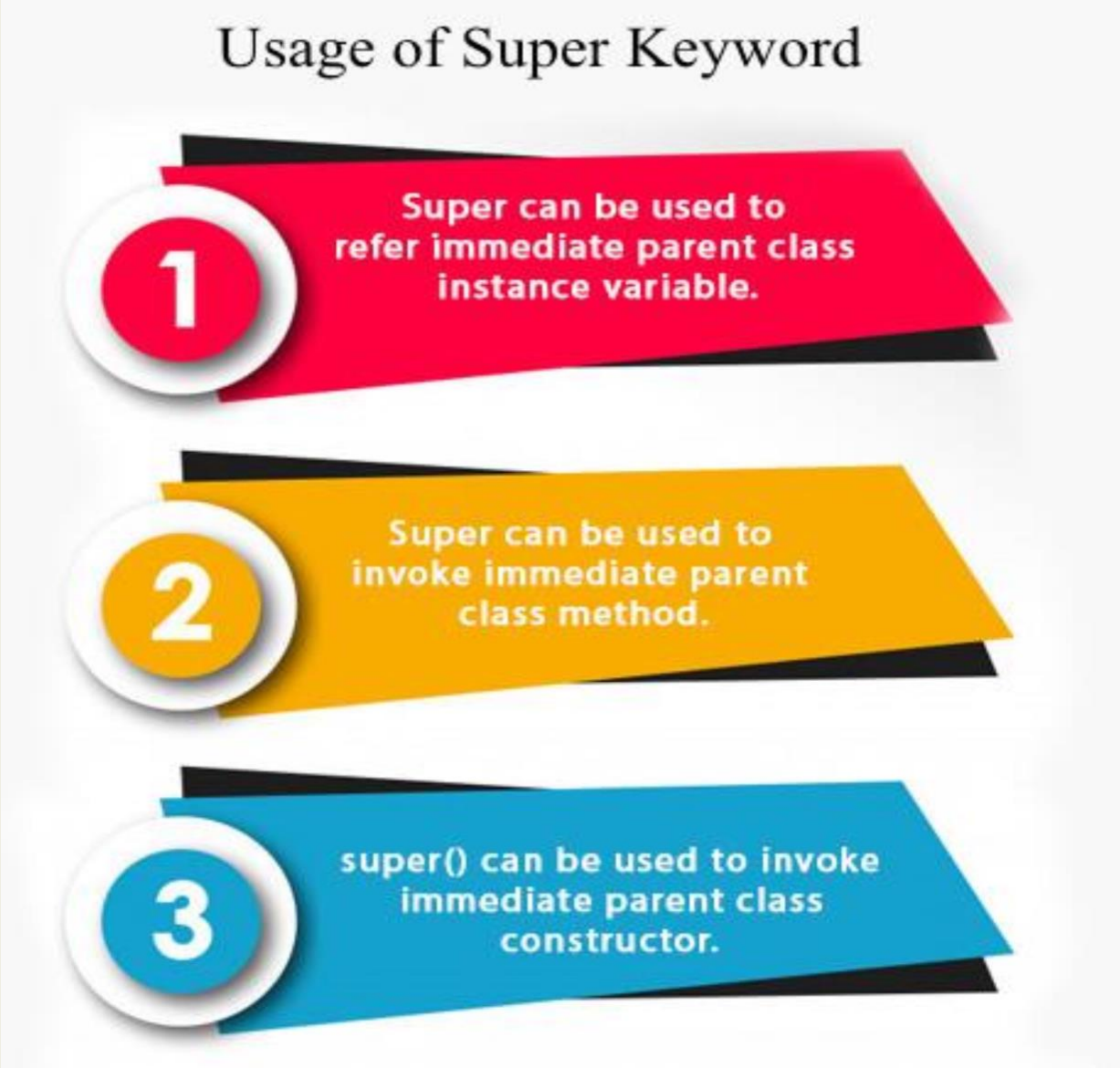
```java
package inheritance1;

import inheritance.person;

public class teacher1{

    String qualification;

    public static void main(String[] args) {

        person p = new person();
        p.setName("Mr. A");                          //setter
        System.out.println("Name: " + p.getName()); //getter

    }
}
```

Constructor

```java
public class person {

    String name;
    int age;


    public person(String name, int age) {
        this.name = name;
        this.age = age;
    }


}
```

```java
public class teacher extends person {


    String qualification;

    public teacher(String name, int age, String qualification) {
        super(name, age);
        this.qualification = qualification;
    }

    public static void main(String [] args) {

        teacher t = new teacher("a", 40, "MSc");
        System.out.println("Name: " + t.name);

    }
}
```

Super Keyword



Super Keyword (cont..)

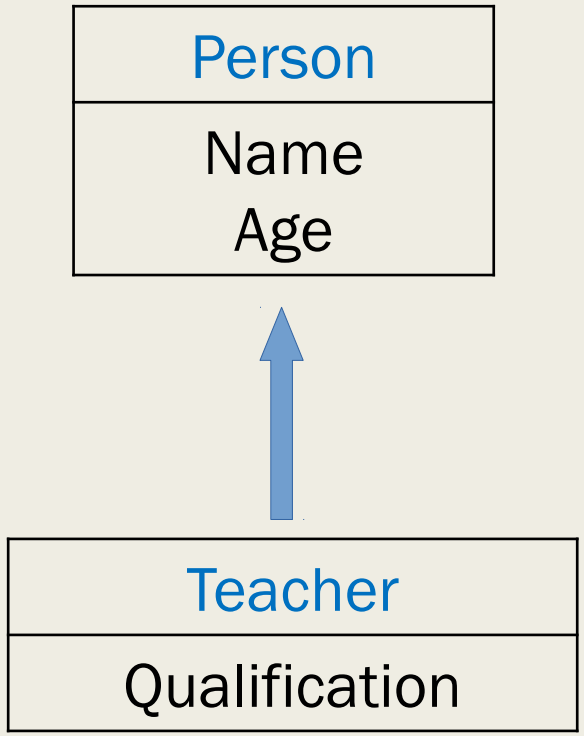
```java
public class person {

    String name;
    int age;


    public person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void display() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

```java
public class teacher extends person {


    String qualification;

    public teacher(String name, int age, String qualification) {
        super(name, age);                              //immediate parent class Constructor
        this.qualification = qualification;
        System.out.println("Name: " + super.name); //immediate parent class instance variable
        super.display();                               //immediate parent class Method
    }

    public static void main(String [] args) {

        teacher t = new teacher("a", 40, "MSc");
        System.out.println("Name: " + t.name + ", Age: " + t.age + ", Qualification: " + t.qualification);
    }
}
```

Types of Inheritance

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance

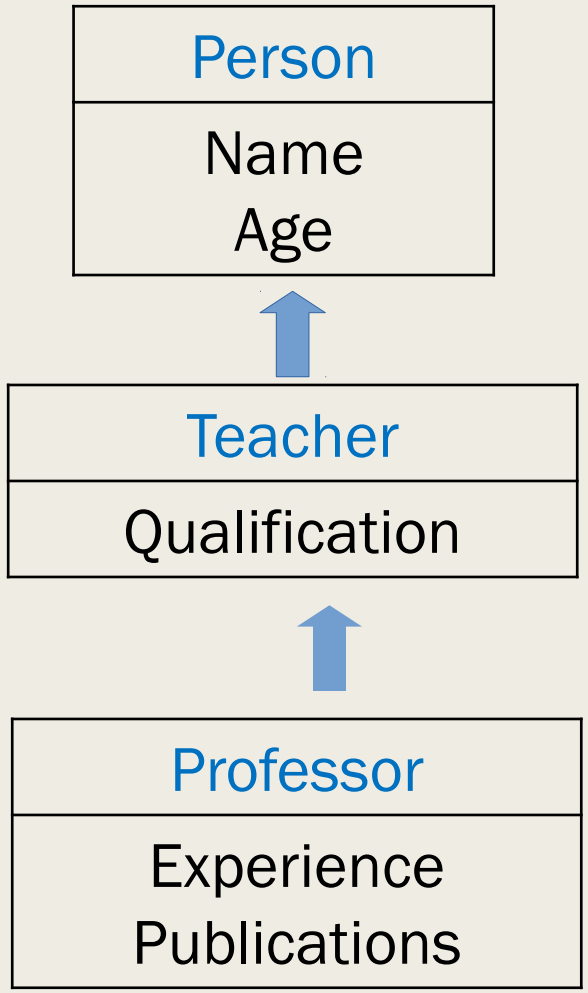
Single Inheritance




```
class person {

    String name;
    int age;



}
```

```
class teacher extends person {

    String qualification;

    public static void main(String[] args) {

        teacher t = new teacher();
        t.name = "Mr. A";
        t.age = 40;
        t.qualification = "PhD";

    }

}
```

Multilevel Inheritance

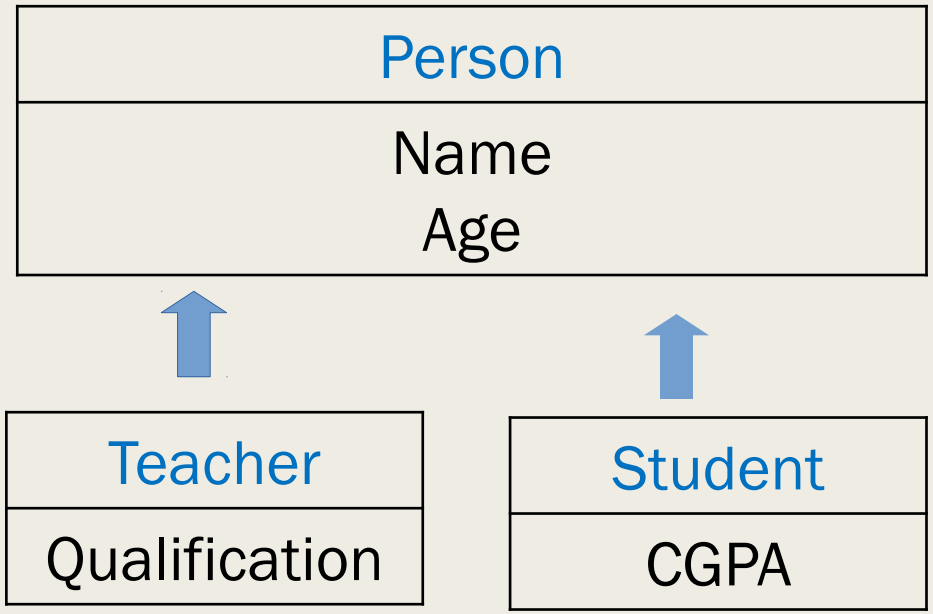



```java
class person {

    String name;
    int age;



}
```

```java
class teacher extends person {

    String qualification;

}
```

```java
class professor extends teacher{
    int experience;
    int publications;


    public  static void main(String [] args) {

        professor pro= new professor();
        pro.name = "a";
        pro.age = 40;
        pro.qualification = "PhD";
        pro.experience = 10;
        pro.publications = 50;

    }



}
```

Hierarchical Inheritance

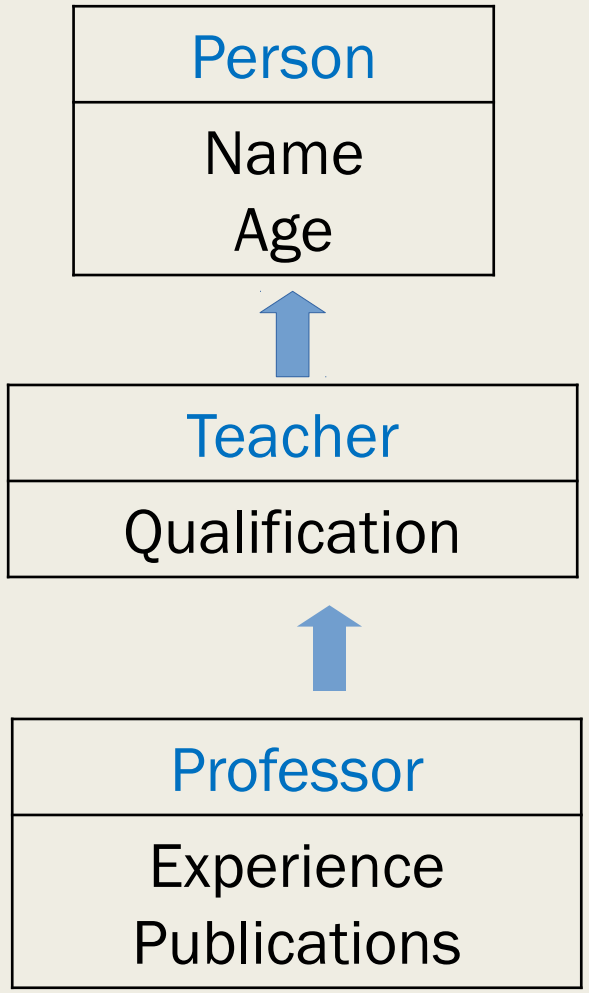



```
class person {

    String name;
    int age;



}
```

```
class teacher extends person {

    String qualification;

}
```

```
class student extends person{

    int cgpa;


}
```

instanceof Operator

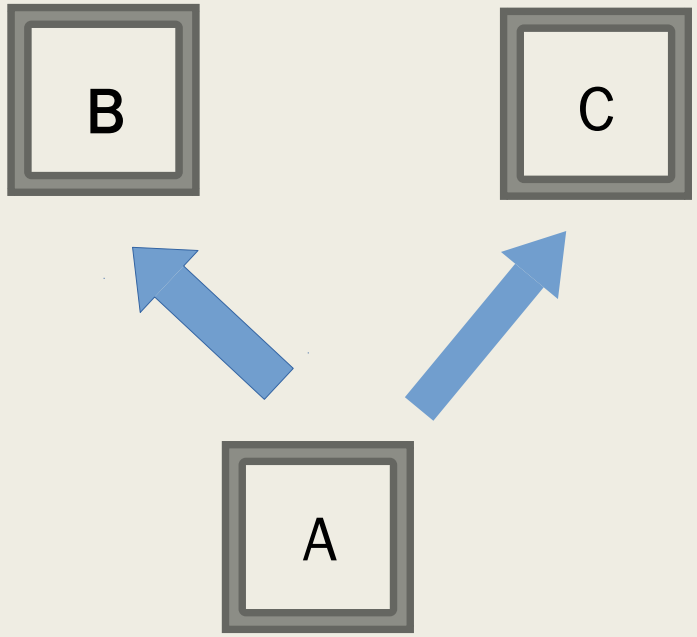


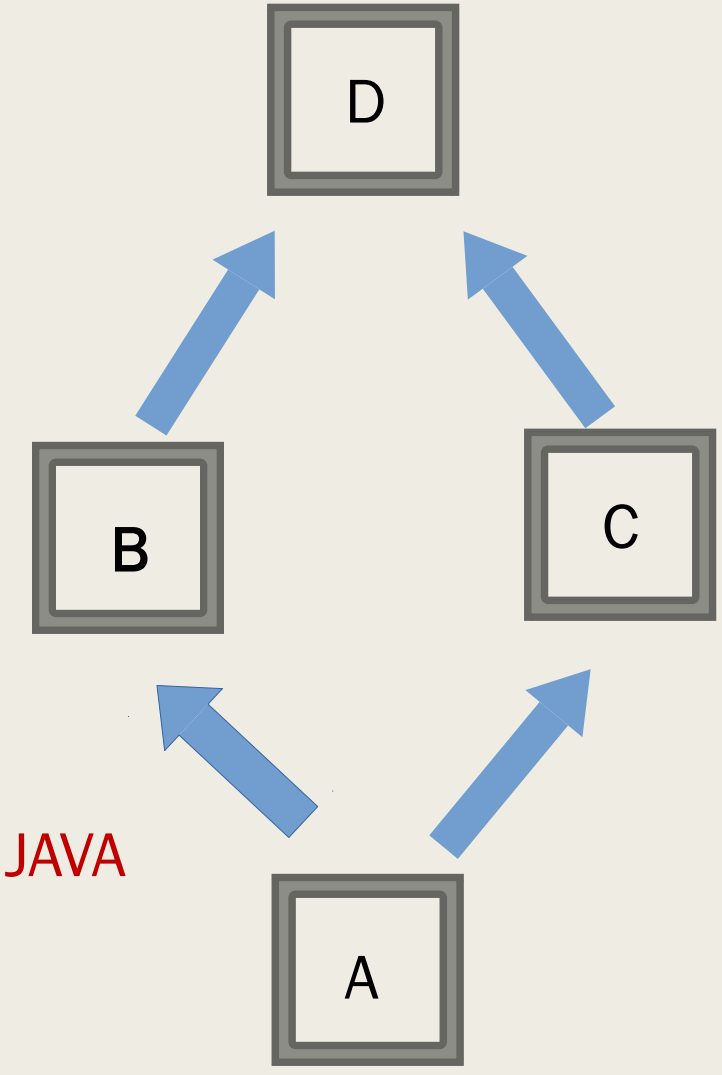

```java
class person {

    String name;
    int age;



}
```

```java
class teacher extends person {

    String qualification;

}
```

```java
class professor extends teacher{
    int experience;
    int publications;


    public  static void main(String [] args) {

        person p = new person();
        teacher t = new teacher();
        professor pro= new professor();

        System.out.println(p instanceof person);    //true
        System.out.println(t instanceof person);    //true
        System.out.println(pro instanceof person);  //true

        System.out.println(p instanceof teacher);   //false
        System.out.println(t instanceof teacher);   //true
        System.out.println(pro instanceof teacher); //true
    }
}
```

Multiple & Hybrid Inheritance





- Multiple Inheritance is not allowed in Classes for JAVA

Abstract Method and Abstract Class

- An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract double getArea();
```

- If a class includes abstract methods, then the class itself **must be** declared abstract, as in:

```
public abstract class Shape{
        abstract double getarea();
}
```

- You cannot create an object of an abstract class.
  - `Shape s1=new Shape();` **//ERROR**

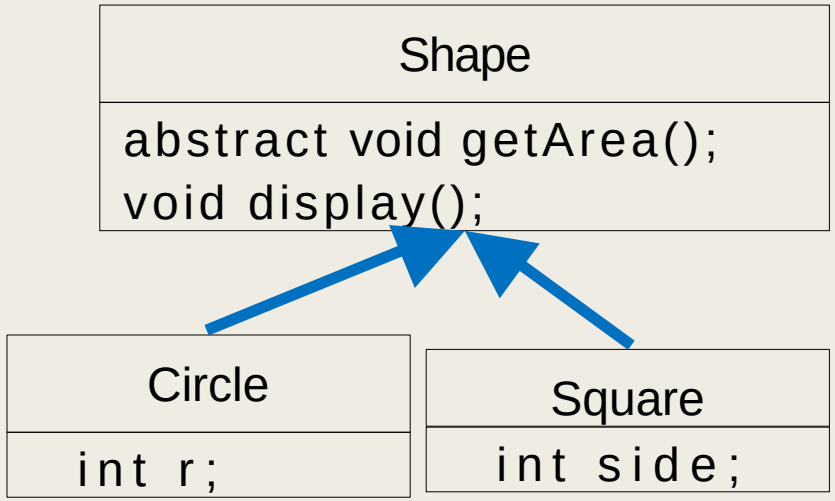
Abstract Class

- A class that cannot be instantiated and contains at least one abstract method
- To access the ***abstract*** class, it must be inherited from another class.

Abstract Class

```java
public class Circle  extends Shape{

    private int r;

    public Circle(int r) {
        this.r=r;
    }

    @Override
    public void getArea() {
        System.out.println("Circle area: " + 3.1416*r*r);

    }
}
```

```java
public abstract class Shape {
    public abstract void getArea();

    void display() {
        System.out.println("non abstract method");
    }
}
```

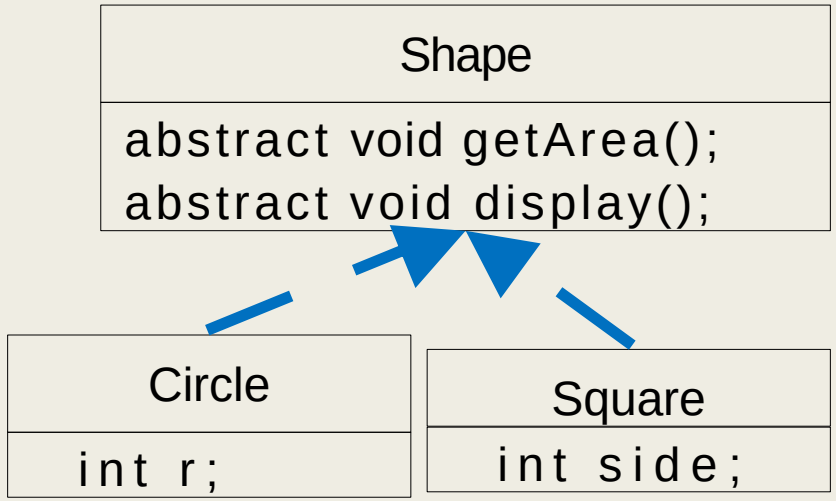



```java
public class Square extends Shape{

    private int side;

    public Square(int side) {
        this.side =side;
    }

    @Override
    public void getArea() {
        System.out.println("Square Area: " + side * side);

    }
    public static void main(String args[]) {
        Shape a = new Square(3);
        a.getArea();

        Shape s = new Circle(3);
        s.getArea();

        s.display();
    }
}
```

Interface

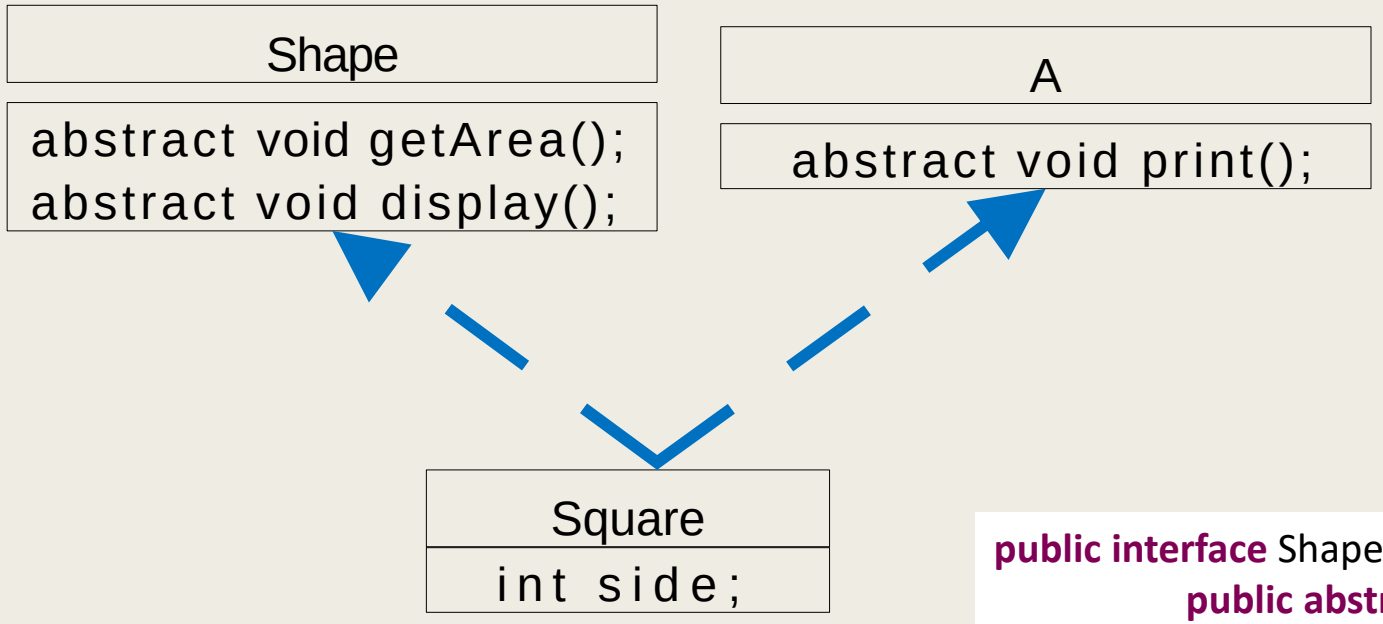
```java
public class Circle  implements Shape{

    private int r;

    public Circle(int r) {
        this.r=r;
    }

    @Override
    public void getArea() {
        System.out.println("Circle area: " + 3.1416*r*r);

    }

    @Override
    public void display() {
        System.out.println("Circle");

    }
}
```

```java
public interface Shape {
        public abstract void getArea();
        public abstract void display();
}
```




```java
public class Square implements Shape{

    private int side;

    public Square(int side) {
        this.side =side;
    }

    @Override
    public void getArea() {
        System.out.println("Square Area: " + side * side);

    }

    @Override
    public void display() {
        System.out.println("Square");

    }

    public static void main(String args[]) {
        Shape a = new Square(3);
        a.getArea();

        Shape s = new Circle(3);
        s.getArea();

        s.display();
    }
}
```

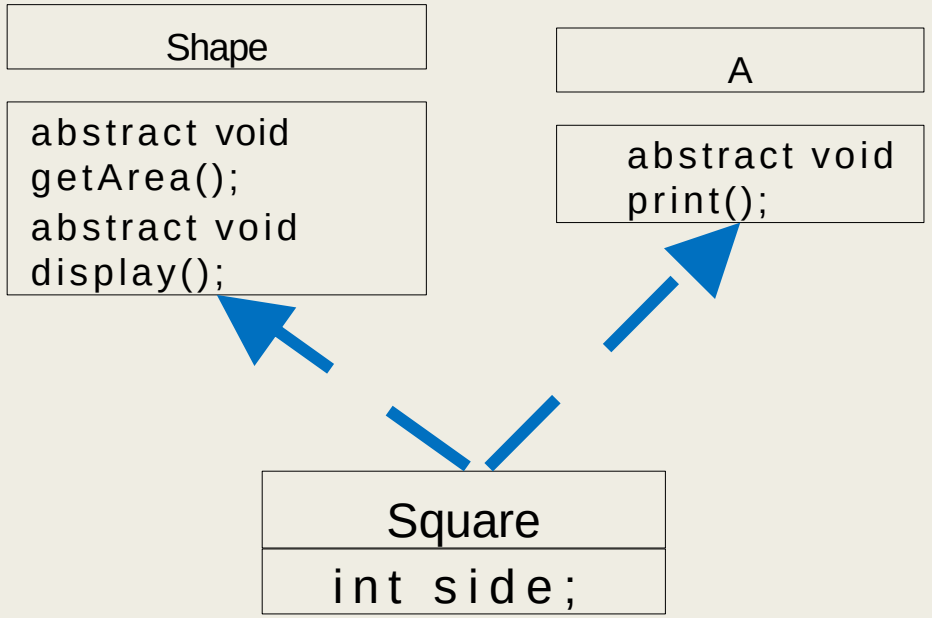
Interface (Multiple)




```
public interface Shape {
        public abstract void getArea();
        public abstract void display();
}
```

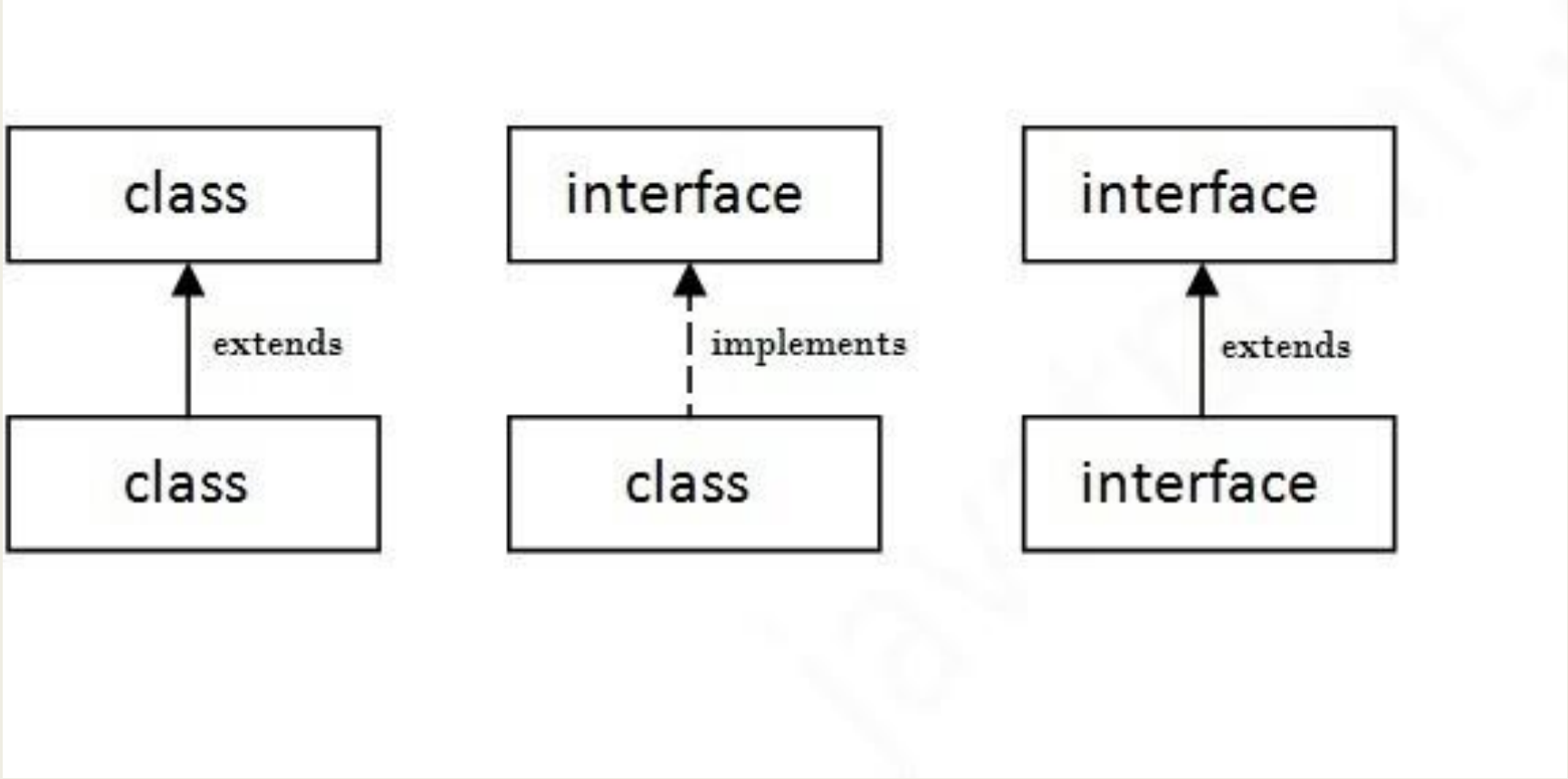
```
public interface A {
    public abstract void print();
}
```

Interface (Multiple)




```java
public class Square implements Shape, A{

    private int side;

    public Square() {}

    public Square(int side) {
        this.side =side;
    }

    @Override
    public void getArea() {
        System.out.println("FROM INTERFACE SHAPE - Square Area: " + side * side);

    }

    @Override
    public void display() {
        System.out.println("Square - FROM INTERFACE SHAPE");

    }

    @Override
    public void print() {
        System.out.println("FROM INTERFACE A");

    }
    public void myprint() {
        System.out.println("METHOD OF SQUARE CLASS");

    }

    public static void main(String args[]) {
        Shape a = new Square(3);
        a.getArea();
        a.display();

        A a2 = new Square();
        a2.print();

        Square a3 = new Square();
        a3.myprint();   // FROM SQUARE CLASS
        a3.print();     // FROM INTERFACE - A
        a3.getArea();   // FROM INTERFACE - SHAPE
    }


}
```

Interface


class
extends
class
interface
implements
class
interface
extends
interface

THANK YOU

CSE 220

Polymorphism

Lec Sumaiya Nuha Mustafina
Dept Of CSE
sumaiyamustafina@cse.mist.ac.bd

Polymorphism

Polymorphism

- Derived from 2 Greek Words
- Poly : Many
- Morph: Forms
- Allows a object/ method to take many forms

Polymorphism

Two types of polymorphism

- Runtime Polymorphism (Dynamic Binding)
- Compile time Polymorphism (Static Binding)

Polymorphism

Two types of polymorphism

- Runtime Polymorphism : Method overriding.
- Compile time Polymorphism : Method overloading.

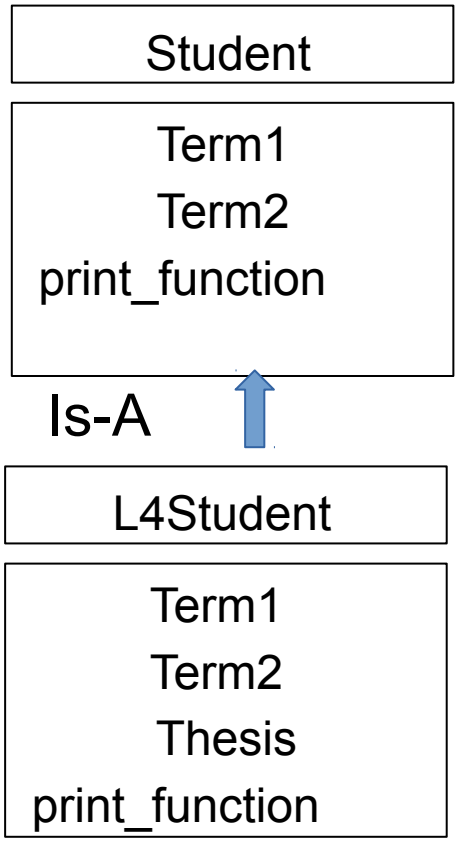
Polymorphism

Two types of polymorphism

- Runtime Polymorphism : Resolved in runtime
- Compile time Polymorphism : Resolved in compile time.

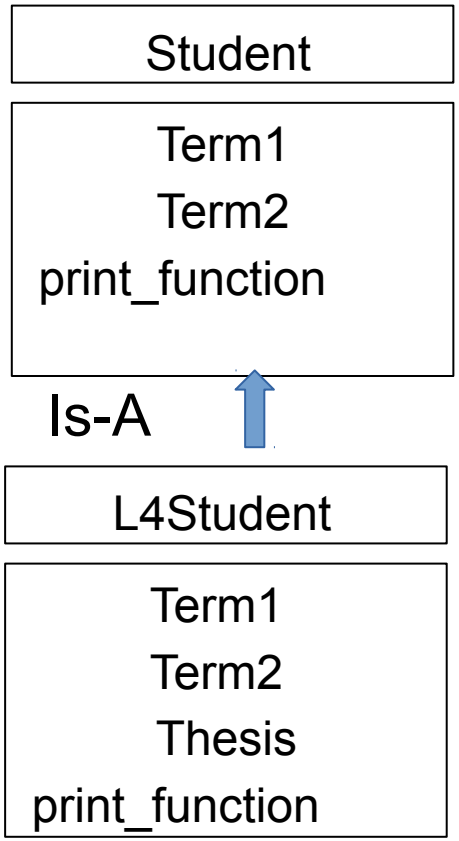
Runtime Polymorphism Or Dynamic Method Dispatch:

```java
public class Student {
    int term1, term2;

    Student(int term1, int term2) {
        this.term1 = term1;
        this.term2 = term2;
    }
    void printvalue()
    {
        System.out.println("Values "+ term1+" "+term2);
    }
}

class L4student extends Student {
    private int thesis;
    L4student(int term1, int term2, int thesis) {
        super(term1, term2);
        this.thesis = thesis;
    }
    @Override
    void printvalue()
    {
        System.out.println("Values "+ term1+" "+term2+ " "+thesis);
    }
    void printclass(L4student s)
    {
        System.out.println("L4Student class");
    }
```



Runtime Polymorphism Or Dynamic Method Dispatch:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Student s=new Student(80,80);
    s.printvalue();
    L4student s1=new L4student(85,80,85);
    s1.printvalue();

    Student ref;
    ref=new L4student(90,90,90); //Upcasting
    ref.printvalue();
}
```



Performed through function overriding and upcasting.

Function Overriding Rules

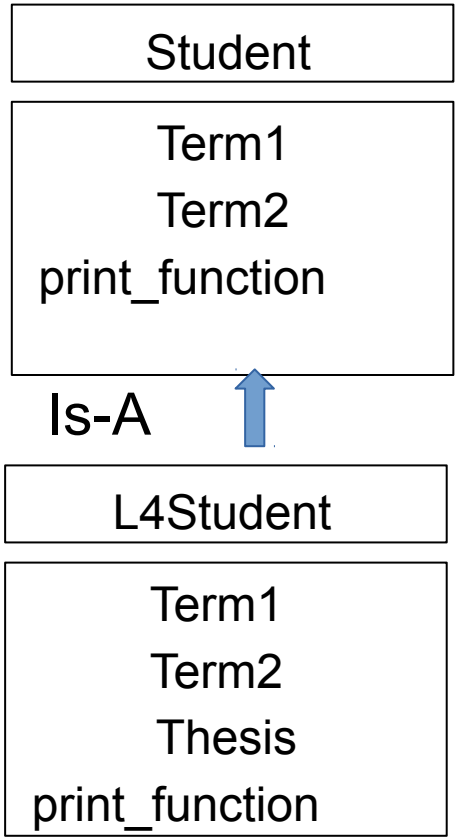
Rules for method overriding:

1. Final methods can not be overridden.
2. Static methods can not be overridden.
3. Private methods can not be overridden.
4. Constructors cannot be overridden.
5. Invoking overridden method from sub-class : We can call parent class method in overriding method using super keyword.
6. You must look for the access modifiers while overriding: For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

Upcasting and Downcasting

Upcasting:

- Upcasting is the typecasting of a child object to a parent object.
- can be done implicitly.
- gives us the flexibility to access the parent class members but it is not possible to access all the child class members using this feature.

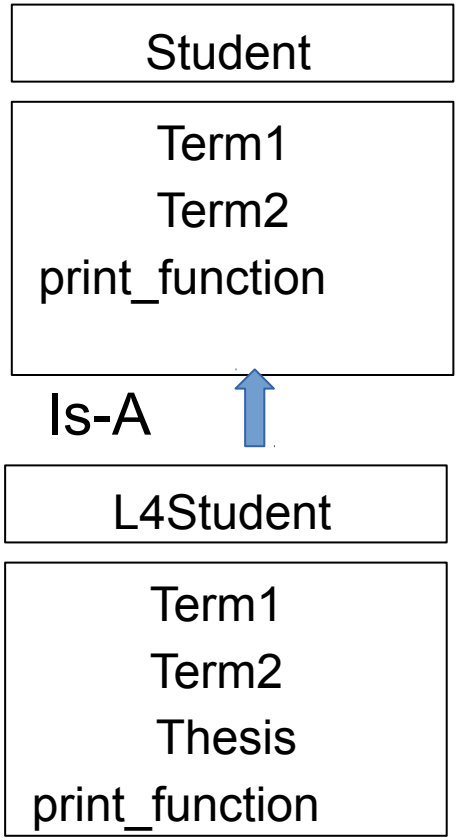



```
Student ref;
ref=new L4student(90,90,90); //Upcasting
```

Upcasting and Downcasting

Downcasting:

- Downcasting means the typecasting of a **parent object to a child object**.
- can not be done implicitly.
- The methods and variables of both super and subclass can be accessed.
- The reference of the parent class object is passed to the child class.
- The object type must be same as the casted type. Check the code snippet shown below. Here the n reference variable is referring to a L4student object and it is being downcasted to L4student.




```
Student n= new L4student(50,50,50);
L4student ref2;
ref2= (L4student) n;  //downcasting
```

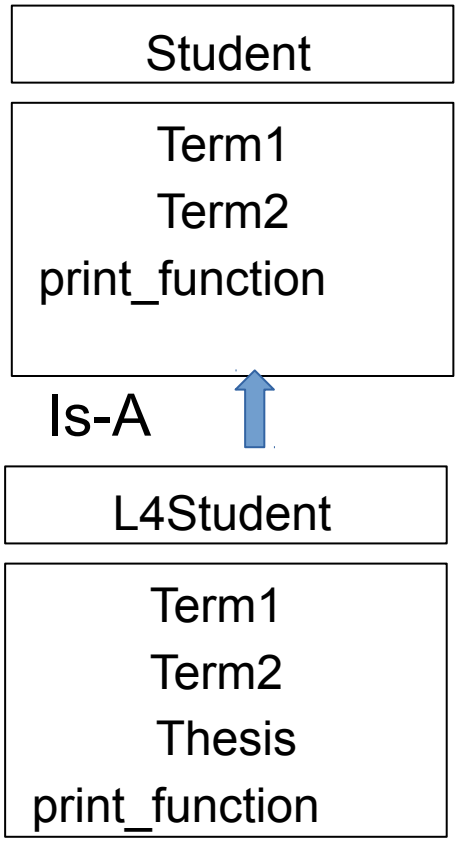
Typecasting

```java
class Main {
  public static void main(String[] args) {
    // create int type variable
    int num = 10;
    System.out.println("The integer value: " + num);

    // convert into double type
    double data = num;
    System.out.println("The double value: " + data);

    String str = "string"+data;
    System.out.println("The string value: " + str);

  }
}
```

```java
class Main {
  public static void main(String[] args) {
    // create string type variable
    String data = "10";
    System.out.println("The string value is: " + data);

    // convert string variable to int
    int num = Integer.parseInt(data);
    System.out.println("The integer value is: " + num);
  }
}
```

Compile Time Polymorphism

```
static void print(Student s)
{
    s.printvalue();
}
static void print(L4student s)
{
    s.printvalue();
}
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Student s=new Student(80,80);
    L4student s1=new L4student(85,80,85);
    print(s);
    print(s1);
}
```



Function overloading -> Static Binding


thank
you

CSE 220

Exception Handling in JAVA

Exception

- An exception is an event that occurs during the execution of a program but disrupts the normal flow of programs' instruction.
- Example:

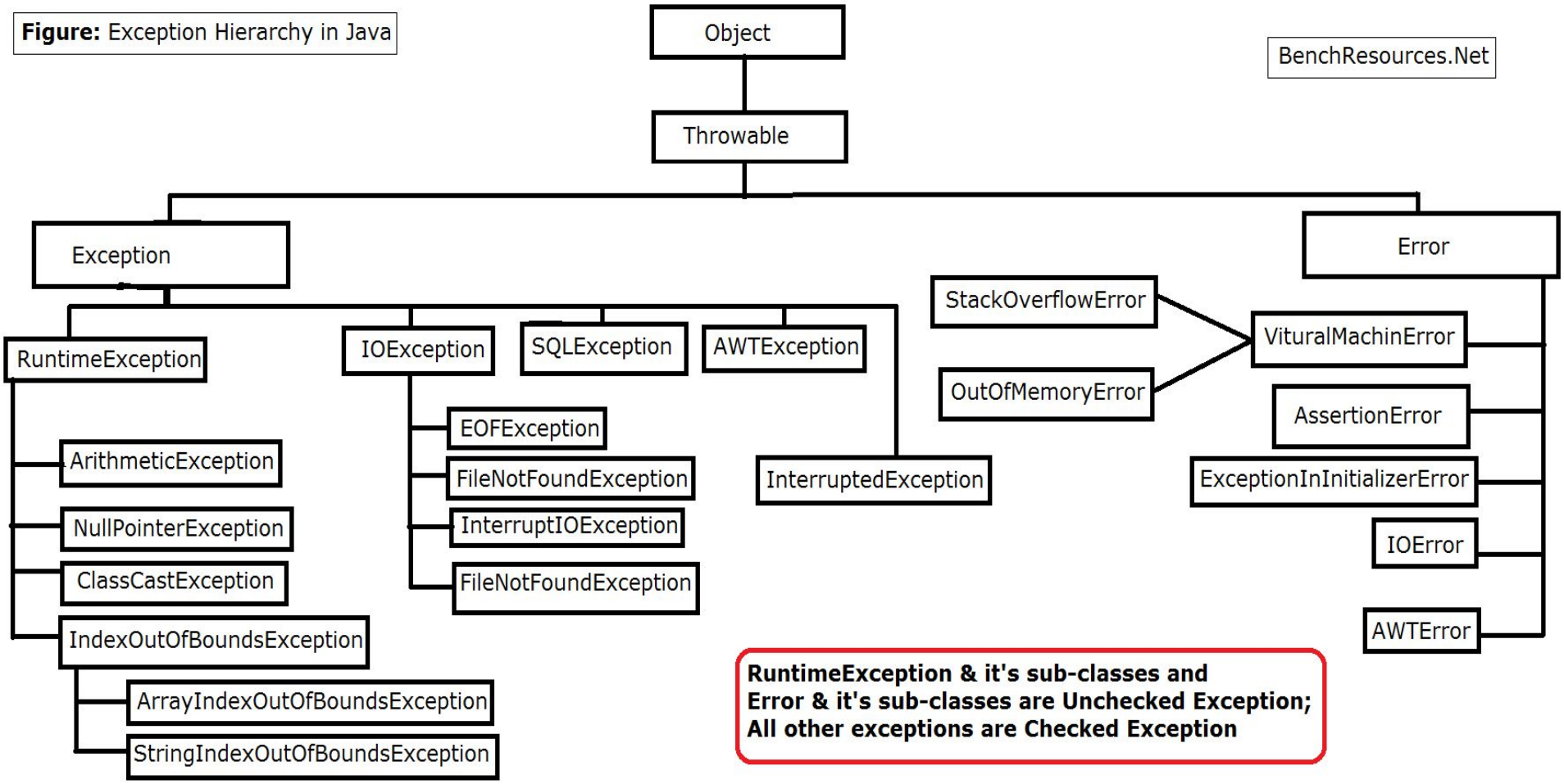
```
1 public class practice1 {
2     public static void main(String[] args) {
3         int a=5,b=0;
4         System.out.println(a/b);
5         System.out.println("Last Line of the code.");
6     }
7 }
```

If we run this code then an exception will occur in line 4. Because we are dividing a number by Zero. As a result line 5 won't be executed disrupting the normal flow of the program. The output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at practice1.main(practice1.java:4)
```

Java Exception Hierarchy

Figure: Exception Hierarchy in Java



Java Exception Hierarchy

Exception is a subclass of Throwable class.

Two types of exception

- **Checked Exception :** Classes that inherit the Throwable class except RuntimeException and Error are known as checked exceptions. Checked exceptions are checked at compile-time.
- **Unchecked Exception :** RuntimeException and its Subclasses , Error and its subclasses are unchecked exception. Unchecked exceptions are checked at runtime.

Java Exception Handling

Now we will learn how to handle the exceptions.

- For example, if we consider the problem of dividing a number by zero (Slide 2), line 5 wasn't being executed due to the exception. So we will handle this exception so that the rest of the code gets executed normally.
- For this we will use **try-catch block.**

try catch block

try block: In this block, we write the statements that may throw an exception.

catch block: In this block, we handle the exception.

- **For one try block, there can be more than one catch blocks**
- **Each catch block must contain a different exception handler.**
- **At a time only one exception occurs and at a time only one catch block is executed.**

Syntax:

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){
}
catch(Exception_class_Name ref){
}
```

Here the **Exception_class_Name** are the ones shown in the **Java Exception Hierarchy. E.g.** for the previous **divide by zero** example, the exception class name will be **ArithmeticException.**

try catch block

So the code:

```
1  public class practice1 {
2      public static void main(String[] args) {
3          int a=5,b=0;
4          try {System.out.println(a/b);}
5          catch(ArithmeticException e) {
6              System.out.println(e); //Prints the exception
7          }
8          System.out.println("Last Line of the code.");
9      }
10 }
```

Here, the statement, a/b is written inside try block. Which causes the execution point to go to the catch block. In catch block, the exception is printed. After that the rest of the program will run normally.

So, the output:

```
java.lang.ArithmeticException: / by zero
Last Line of the code.
```

try- multiple catch blocks

- Let's consider another example. Where we take an integer array, and divide an element of array with another element of the array.
- Like, arr[0]/arr[2]
- Here, two kinds of exception can occur,
  - Array element can be zero, so divide by zero can occur, which will throw ArithmeticException.
  - Array index can be out of bound, which will throw the **ArrayIndexOutOfBoundsException**. (See the hierarchy)
- So here for handling these two exception two catch blocks will be needed.

```
1  public class practice1 {
2      public static void main(String[] args) {
3          int []a= new int[2];
4          try {System.out.println(a[0]/a[2]);}
5          catch(ArithmeticException e) {
6              System.out.println("A.E  "+e); //Prints the exception
7          }
8          catch(ArrayIndexOutOfBoundsException e)
9          {
10             System.out.println("Index out of Bound "+e);
11         }
12         System.out.println("Last Line of the code.");
13     }
14 }
```

try- multiple catch blocks

- In this code, the array had 2 elements and initially they were zero. The indices were 0 and 1.
- In line 4, a[2] was being accessed, which was out of bound and that caused an exception.

```
1  public class practice1 {
2      public static void main(String[] args) {
3          int []a= new int[2];
4          try {System.out.println(a[0]/a[2]);}
5          catch(ArithmeticException e) {
6              System.out.println("A.E  "+e); //Prints the exception
7          }
8          catch(ArrayIndexOutOfBoundsException e)
9          {
10             System.out.println("Index out of Bound "+e);
11         }
12         System.out.println("Last Line of the code.");
13     }
14 }
```

Output:

```
Index out of Bound java.lang.ArrayIndexOutOfBoundsException: Index 2 out of bounds for length 2
Last Line of the code.
```

try- multiple catch blocks

- In this code, the array had 2 elements and initially they were **zero**. The indices were 0 and 1.
- In line 4, a[0] was divided by a[1], which was initialized with zero and that caused an exception (ArithmeticException)

```
1  public class practice1 {
2      public static void main(String[] args) {
3          int []a= new int[2];
4          try {System.out.println(a[0]/a[1]);}
5          catch(ArithmeticException e) {
6              System.out.println("A.E  "+e); //Prints the exception
7          }
8          catch(ArrayIndexOutOfBoundsException e)
9          {
10             System.out.println("Index out of Bound "+e);
11         }
12         System.out.println("Last Line of the code.");
13     }
14 }
```

Output:

```
A.E  java.lang.ArithmeticException: / by zero
Last Line of the code.
```

Try_Catch with General Exception

- If we don't know the name of the exception, then we can **generalize the exception** i.e we can use the **parent class/superclass.**
- e.g. If we consider the previous example, we may consider the array out of index exception as we are working with arrays. But the divide by zero exception may be overlooked by us. In that case we may **generalize** the exception.
- In this code, we have used Exception class which is the parent class/superclass of all exceptions, so it can handle any exception.

Code:

```
1  public class practice1 {
2      public static void main(String[] args) {
3          int []a= new int[2];
4          try {System.out.println(a[0]/a[1]);}
5          catch(ArrayIndexOutOfBoundsException e)
6          {
7              System.out.println("Index out of Bound "+e);
8          }
9          catch(Exception e)
10         {
11             System.out.println("Generalized exception "+e);
12         }
13         System.out.println("Last Line of the code.");
14     }
15 }
```

Output:

```
Generalized exception java.lang.ArithmeticException: / by zero
Last Line of the code.
```

Order of writing multiple catch blocks

- All catch blocks must be ordered from most specific to most general. In other words, write the subclasses before writing the parent class.
- e.g. catch for **ArrayIndexOutOfBoundsException** must come before catch for **Exception** otherwise error will be shown.

Code:

```
1  public class practice1 {
2      public static void main(String[] args) {
3          int []a= new int[2];
4          try {System.out.println(a[0]/a[1]);}
5          catch(Exception e)
6          {
7              System.out.println("Generalized exception "+e);
8          }
9          catch(ArrayIndexOutOfBoundsException e)
10         {
11             System.out.println("Index out of Bound "+e);
12         }
13         System.out.println("Last Line of the code.");
14     }
15 }
```

Output:

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Unreachable catch block for ArrayIndexOutOfBoundsException. It is already handled by the catch block for Exception

at practice1.main(practice1.java:9)

Finally block

- The finally block always executes when try block exists.
- This ensures that the finally block always executes whether exception occurs or not. That doesn't matter.

Code:

```
1  public class practice1 {
2      public static void main(String[] args) {
3          int []a= new int[2];
4          try {System.out.println(a[0]/a[1]);}
5          catch(Exception e)
6          {
7              System.out.println("Generalized exception "+e);
8          }
9          finally {
10             System.out.println("finally block ");
11         }
12         System.out.println("Last Line of the code.");
13     }
14 }
```

Output:

```
Generalized exception java.lang.ArithmeticException: / by zero
finally block
Last Line of the code.
```

For more on finally block: https://www.javatpoint.com/finally-block-in-exception-handling

throw Keyword

- Till now, in all the examples that we had seen, exceptions were thrown implicitly. Like, when a number was divided by zero, then arithmetic exception was invoked implicitly.
- Using throw keyword, we can throw an exception explicitly.
- We can throw either checked or unchecked exceptions in Java by throw keyword.
- Syntax: **throw new** exception_class("error message");
- For example, if we want an exception to be thrown if a number is divided by 1, then

Code:

```
1  public class practice1 {
2      public static void main(String[] args) {
3          int a=5,b=1;
4          if(b==1)
5          {
6              throw new ArithmeticException("Divide by 1");
7          }
8          System.out.println(a/b);
9      }
10 }
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Divide by 1
        at practice1.main(practice1.java:6)
```

For more on throw: https://www.javatpoint.com/throw-keyword

throws Keyword

- Java throws keyword is used in the **method signature** to declare an exception which might be thrown by the function while the execution of the code.
- Syntax:

  return_type method_name() throws exception_class_name{

  //method code

  }

- **We should mainly use it to declare the checked exceptions.**

  Reference：

  https://www.geeksforgeeks.org/throw-throws-java/

  https://www.javatpoint.com/throws-keyword-and-difference-between-throw-and-throws

Exception in Inheritance (RULES)

Rules for exception handling with method overriding in Java

1. If **super class** method has not declared any exception using **throws clause** then subclass overridden method **can't declare any checked exception** but it **can declare unchecked exception** with the throws clause.
2. If **super class** method has declared a checked exception using throws clause then subclass overridden method can do **one of the three things.**
   a. subclass can declare the same exception as declared in the superclass method
   b. Subclass can declare the subtype exception declared in the super class method. But subclass method can not declare any exception that is up in the hierarchy than the exception declared in the super class method.
   c. Subclass method can choose not to declare any exception at all

exception handling with method overriding (RULE 1)

If **super class** method has not declared any exception using **throws clause** then subclass overridden method **can't declare any checked exception** but it **can declare unchecked exception** with the throws clause.

E.g.

```
1  class A{
2      void print()
3      { System.out.println("Class A"); }
4  }
5  class B extends A{
6      String s=null;
7      void print() throws NullPointerException
8      { System.out.println("Class B"+s.length()); }
9  }
10 public class practice1 {
11     public static void main(String[] args) {
12         B ob1=new B();
13         try {
14         ob1.print();
15         }
16         catch(Exception e)
17         {
18             System.out.println("exception handled ");
19         }
20         System.out.println("Last Line of the code.");
21     }
22 }
```

Super class A has No exception
But Sub class B has an unchecked exception

Output:

```
exception handled
Last Line of the code.
```

exception handling with method overriding (RULE 2.a)

If super class method has declared a checked exception using throws clause then subclass can declare the same exception as declared in the superclass method.

E.g.

```
1  import java.io.IOException;
2  class A{
3      void print() throws IOException
4      { System.out.println("Class A"); }
5  }
6  class B extends A{
7      void print() throws IOException
8      { System.out.println("Class B "); }
9  }
10 public class practice1 {
11     public static void main(String[] args) {
12         B ob1=new B();
13         try {
14         ob1.print();
15         }
16         catch(Exception e)
17         {
18             System.out.println("exception handled ");
19         }
20         System.out.println("Last Line of the code.");
21     }
22 }
```

Super class A has a checked exception
and Sub class B has declared the same checked exception

Output:

```
Class B
Last Line of the code.
```

exception handling with method overriding (RULE 2.b)

Subclass can declare the subtype exception declared in the super class method. But subclass method can not declare any exception that is up in the hierarchy than the exception declared in the super class method.

E.g.

```
1  import java.io.IOException;
2  class A{
3      void print() throws IOException
4      { System.out.println("Class A"); }
5  }
6  class B extends A{
7      void print() throws Exception
8      { System.out.println("Class B "); }
9  }
10 public class practice1 {
11     public static void main(String[] args) {
12         B ob1=new B();
13         try {
14         ob1.print();
15         }
16         catch(Exception e)
17         {
18             System.out.println("exception handled ");
19         }
20         System.out.println("Last Line of the code.");
21     }
22 }
```

Super class A has a checked exception named IOException and Sub class B has declared another exception which is the superclass of IOException. So error.

Output: Error!!

exception handling with method overriding (RULE 2.c)

Subclass method can choose not to declare any exception at all

E.g.

```
1  import java.io.IOException;
2  class A{
3      void print() throws IOException
4      { System.out.println("Class A"); }
5  }
6  class B extends A{
7      void print()
8      { System.out.println("Class B "); }
9  }
10 public class practice1 {
11     public static void main(String[] args) {
12         B ob1=new B();
13         try {
14         ob1.print();
15         }
16         catch(Exception e)
17         {
18             System.out.println("exception handled ");
19         }
20         System.out.println("Last Line of the code.");
21     }
22 }
```

Super class A has a checked exception named IOException but subclass B's method has not declared any exception.

Output:

```
Class B
Last Line of the code.
```

References

Basics:

- Exception Handling in Java
- Java Finally block - javatpoint
- Java throw exception - javatpoint
- Java Throws Keyword - javatpoint
- throw and throws in Java - GeeksforGeeks
- Difference between throw and throws in java - javatpoint


thank
you

CSE-220
Object Oriented Programming Language Sessional-II

File I/O in Java

Methods in Java File

CanRead()	Tests whether file is readable or not
CanWrite()	Test whether the file is writable or not
CreateNewFile()	Creates an empty file
Delete()	Deletes a file
Exists()	Tests whether the file exists or not
GetName()	Returns the name of the file
GetAbsolutePath()	Returns the absolute pathname of the file
Length()	Returns the size of the file in bytes
List()	Returns the array of the file in the directory
Mkdir()	Creates a directory

File Operations

Create

Create New File

Read

Read From File

Write

Write Into File

Get

Get File Information

Create a File

package-info.java *CreatFile.java

```java
package FileOperations;

import java.io.File;   //Import the File Class
import java.io.IOException;  //Import the Exception class to handle errors

public class CreatFile {
    public static void main(String [] args) {
        File obj= new File("/Users/rubyeatislam/eclipse-workspace/FileInputOuput/src/File1.txt");
        try {
            if(obj.createNewFile()) {
                System.out.println("File Created "+ obj.getName());
            }
            else {
                System.out.println("File creation failed");
            }
        }catch(IOException e) {
            System.out.println("An error occured");
            e.printStackTrace();
        }


    }

}
```

Problems Javadoc Declaration Console

<terminated> CreatFile [Java Application] /Users/rubyeatislam/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_17.0.8.v20230831-1047/jre/bin/java (Oct 5, 2023, 9:12:31 AM – 9:12:31 AM

```
File CreatedFile2.txt
```

Get File Information

package-info.java | CreatFile.java | *GetFileInfo.java

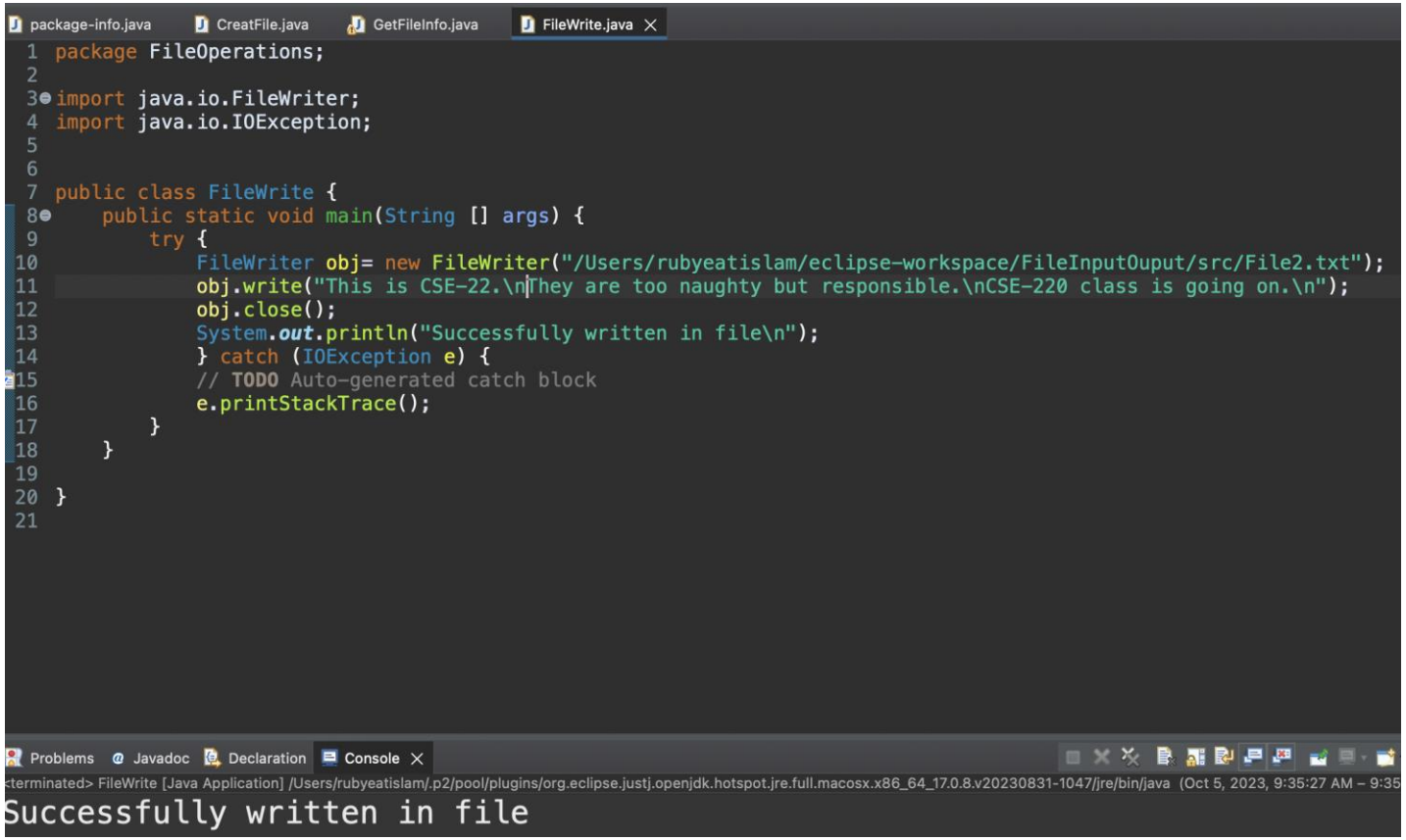
```
1  package FileOperations;
2
3  import java.io.File;
4  import java.io.IOException;
5
6  public class GetFileInfo {
7      public static void main(String [] args) {
8          File obj= new File("/Users/rubyeatislam/eclipse-workspace/FileInputOuput/src/File2.txt");
9              if(obj.exists()) {
10                 System.out.println("File Created "+ obj.getName());
11                 System.out.println("Absolute Path " + obj.getAbsolutePath());
12                 System.out.println("Can Read " + obj.canRead());
13                 System.out.println("Can Write " + obj.canWrite());
14                 System.out.println("File size in lengths " + obj.length());
15             }
16             else {
17                 System.out.println("An error occured");
18             }
19     }
20 }
21
```

Problems | Javadoc | Declaration | Console

<terminated> GetFileInfo [Java Application] /Users/rubyeatislam/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_17.0.8.v20230831-1047/jre/bin/java (Oct 5, 2023, 9:25:22 AM – 9:25:22 AM)

```
File Created File2.txt
Absolute Path /Users/rubyeatislam/eclipse-workspace/FileInputOuput/src/File2.t
Can Read true
Can Write true
File size in lengths 0
```

Write Into File



Read From File

package-info.java | CreatFile.java | GetFileInfo.java | FileWrite.java | FileRead.java

```java
package FileOperations;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileRead {
    public static void main(String [] args) {
        File obj= new File("/Users/rubyeatislam/eclipse-workspace/FileInputOuput/src/File2.txt");
        try {
            Scanner read = new Scanner(obj);
            while(read.hasNextLine()) {
                String data= read.nextLine();
                System.out.println(data);
            }
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
;   }

}
```

Problems | Javadoc | Declaration | Console

<terminated> FileRead [Java Application] /Users/rubyeatislam/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_17.0.8.v20230831-1047/jre/bin/java (Oct 5, 2023, 9:43:56 AM

```
This is CSE-22.
They are too naughty but responsible.
CSE-220 class is going on.
```

Thank You

```
/*
 * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package fileio;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 *
 * @author mmg_lab_67
 */
public class FileIO {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        File obj = new File("E:\\File1.txt");

        try {
            Scanner read= new Scanner(obj);
            while(read.hasNextLine()){
                String Data= read.nextLine();
                System.out.println(Data);
            }


            /*if(obj.exists()){
            System.out.println("Get Name " + obj.getName());
            System.out.println("Get Path " + obj.getAbsolutePath());
            System.out.println("Readable " + obj.canRead());
            System.out.println("Writable " + obj.canWrite());
            System.out.println("File length " + obj.length());
            }
            else{
            System.out.println("Does not exist" );
            }

            try {
            FileWriter obj1= new FileWriter("E:\\File1.txt");
            obj1.write("Good.\n They are responsible.");
            obj1.append("Excellent.\n ");
            obj1.close();
```

```
            } catch (IOException ex) {
            System.out.println("Can't access");
            Logger.getLogger(FileIO.class.getName()).log(Level.SEVERE,
null, ex);
            }*/
            /*try {
            if(obj.createNewFile()){
            System.out.println("File Created "+ obj.getName());
            }
            else{
            System.out.println("Failed");
            }
            } catch (IOException ex) {
            System.out.println("An error occured");
            ex.printStackTrace();
            }*/
        } catch (FileNotFoundException ex) {
            Logger.getLogger(FileIO.class.getName()).log(Level.SEVERE,
null, ex);
        }




    }

}
```

```
package fileio;
```

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;


public class CSVFileOperations {
    public static void main(String [] args){
        String path= "C:/Users/mmg_lab_67/Downloads/ReadFile.csv";

        try {
            BufferedReader br = new BufferedReader(new FileReader(path));

           String line="";
            try {
                while((line=br.readLine())!=null){
                    String [] values= line.split(",");

                    System.out.println(values[0]+"   "+values[1]+"
"+values[2]);
                }

            } catch (IOException ex) {
                ex.printStackTrace();
            }

        } catch (FileNotFoundException ex) {
            ex.printStackTrace();
        }


        String wpath= "C:/Users/mmg_lab_67/Downloads/WriteFile.csv";

        try {
            BufferedWriter bw= new BufferedWriter(new FileWriter(wpath));

            bw.write("Name\n Rubyeat\n Mahin\n");
            bw.close();

            System.out.println("Sucessfully written");
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

}
```

//read from write into CSV

```
package fileio;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class ReadFromWriteIntoCSVFile {
    public static void main(String [] args) throws FileNotFoundException,
IOException{
        String path= "C:/Users/mmg_lab_67/Downloads/ReadFile.csv";

    String wpath= "C:/Users/mmg_lab_67/Downloads/WriteFile.csv";

    BufferedReader br= new BufferedReader(new FileReader(path));

    String line= "";
    BufferedWriter bw= new BufferedWriter(new FileWriter(wpath));
    while((line=br.readLine())!=null){
        String [] values= line.split(",");
        bw.write(values[0]+ "  " + values[1]+ "\n");
        System.out.println("Written");
    }
    bw.close();

    }


}
```

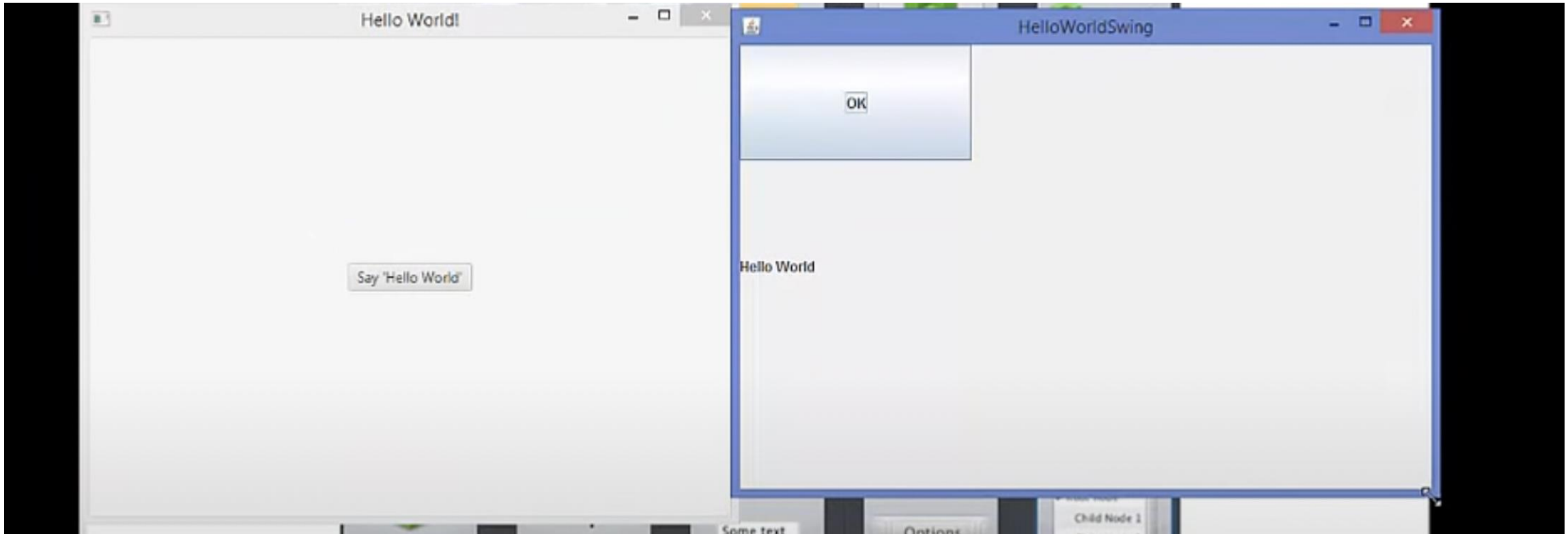
JavaFX

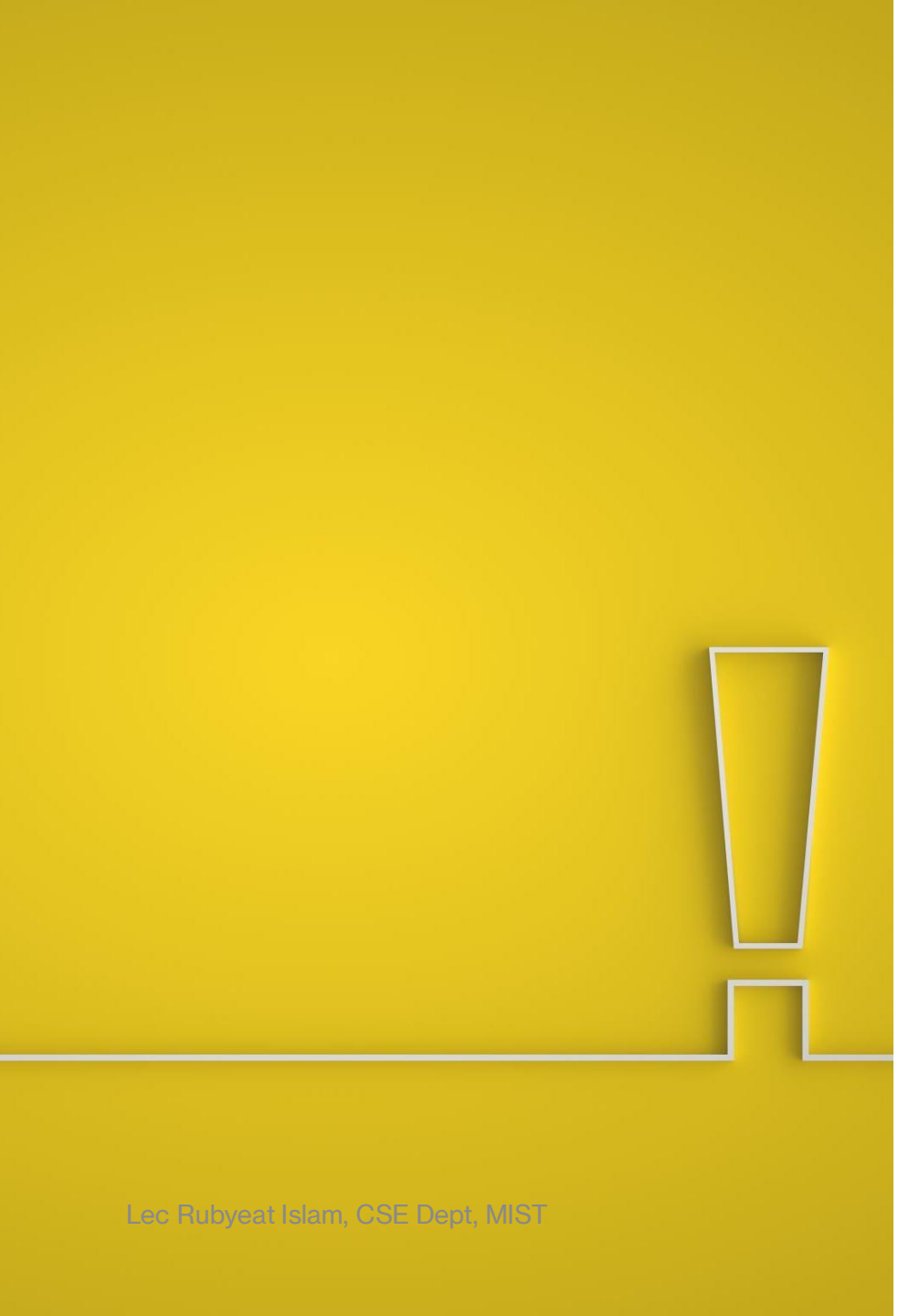
Lec Rubyeat Islam
CSE Dept
MIST

What is JavaFX?

- A media/graphics framework for creating GUIs in Java Applications
- JavaFX is intended to replace Swing as the standard GUI library for Java SE
- More powerful
- Used to create both desktop and web applications
- Like Swing draws its own components, less communication with OS
- Lightweight and hardware-accelerated
- Makes use of FXML- new XML-based mark-up language for defining UIs

Java Swing vs JavaFX



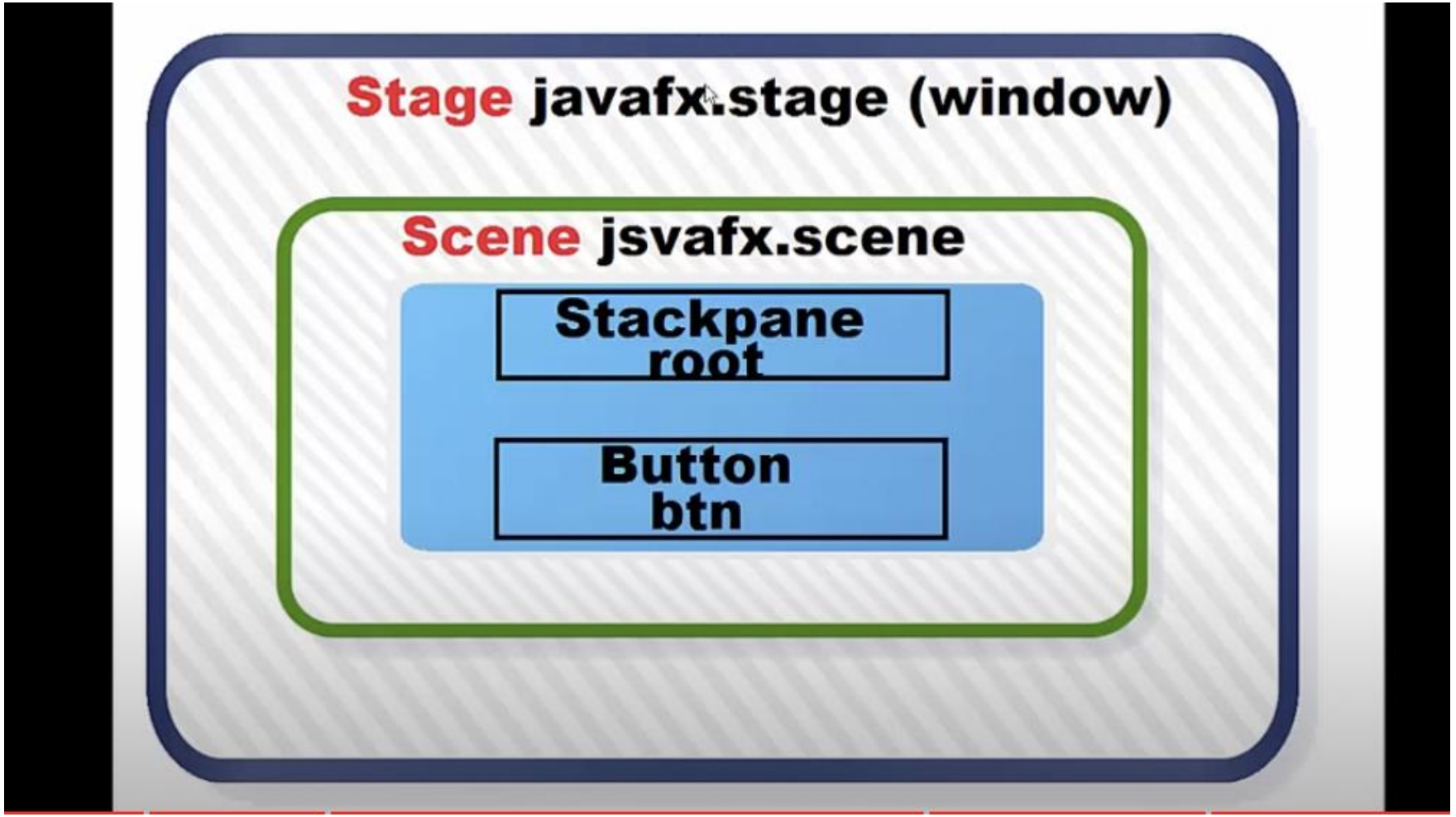


Why Use It

- Oracle provides you with JavaFX Scene Builder
  - Visual UI design tool
  - No need to spend n-hours anymore on trying to move a button a little bit to the left
- Ability to apply CSS on the components of the UI opens a door for better looking applications that are more attractive to the end-user
- 3D support (possibility of games cropping up)

JavaFX or Swing/AWT

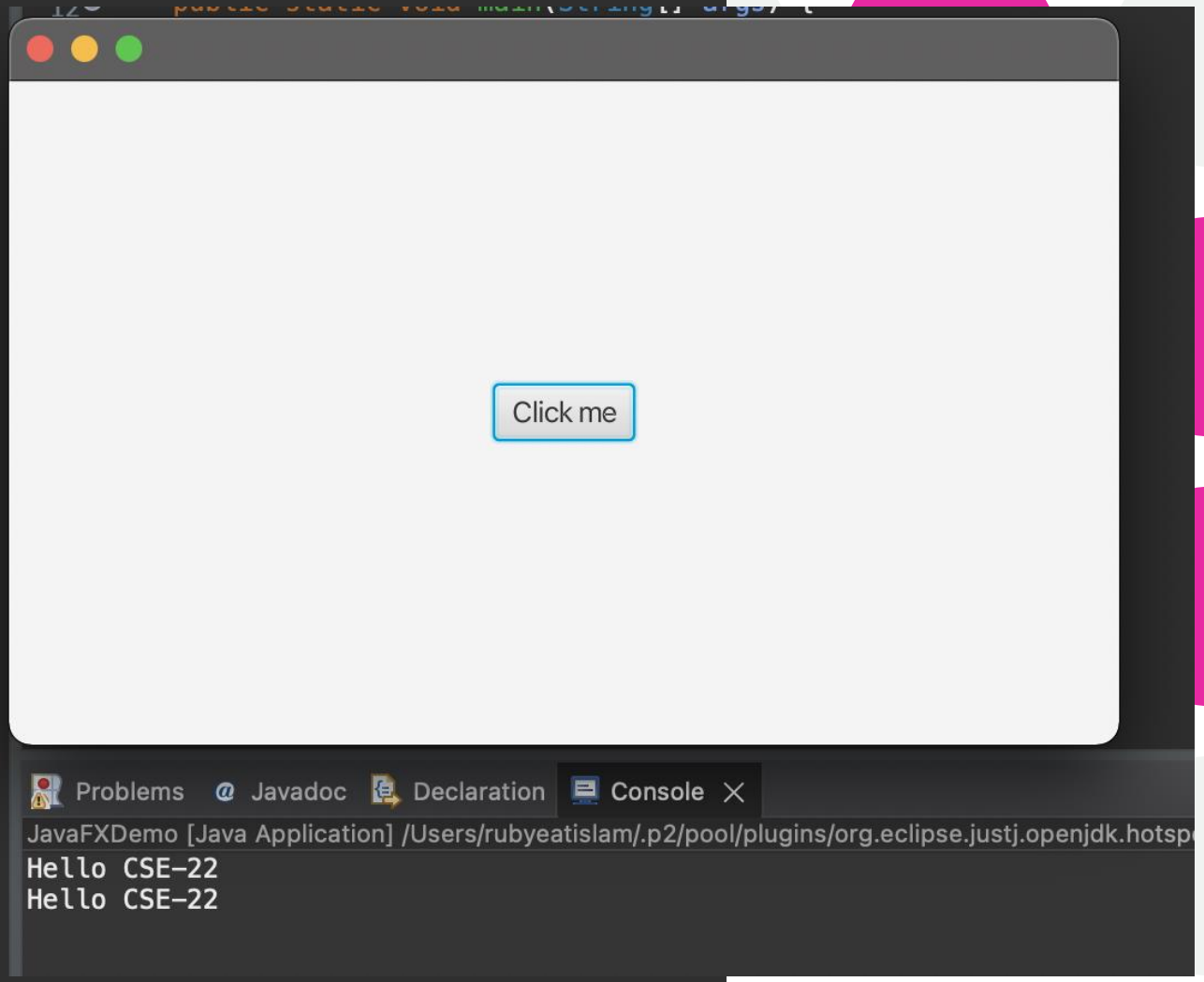
- GUIs are created a lot faster than in Swing and AWT
- More sophisticated and aesthetically pleasing UIs
- Easy integration of sounds, images and videos and of web content
- Code is simplified in JavaFX by separating the UI from the logic of the application
- JavaFX can be integrated in Swing applications, allowing for a smoother transition


Stage javafx.stage (window)
Scene jsvafx.scene
Stackpane
root
Button
btn

First JavaFX Application

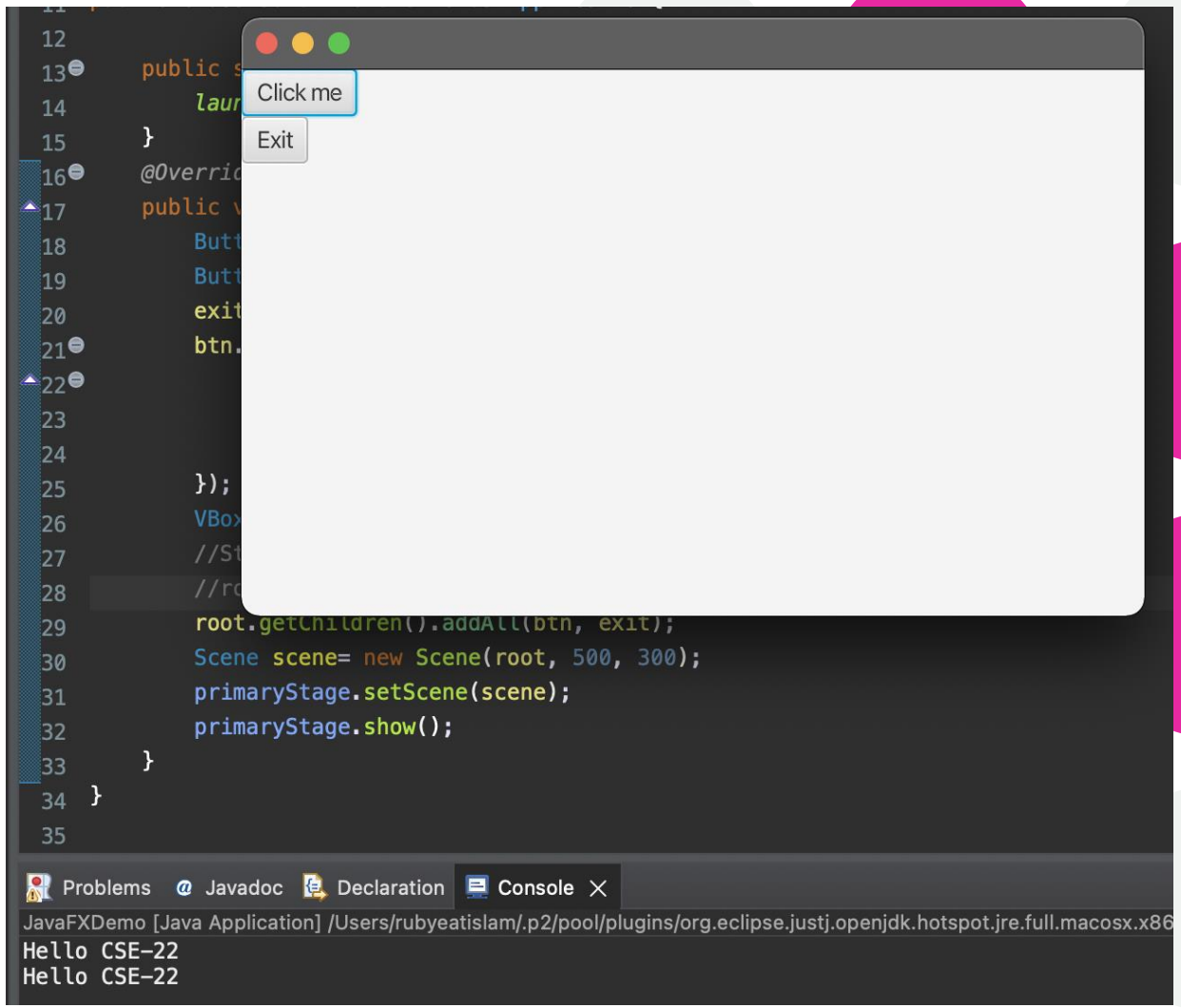
*JavaFXDemo.java

```java
package HelloFX;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class JavaFXDemo extends Application{

    public static void main(String[] args) {
        launch(args);
    }
    @Override
    public void start(Stage primaryStage) throws Exception {
        Button btn = new Button ("Click me");
        btn.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent event) {
                System.out.println("Hello CSE-22");
            }
        });
        StackPane root = new StackPane();
        root.getChildren().add(btn);
        Scene scene= new Scene(root, 500, 300);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```



JavaFX with Lambda Expression

```java
public class JavaFXDemo extends Application{

    public static void main(String[] args) {
        launch(args);
    }
    @Override
    public void start(Stage primaryStage) throws Exception {
        Button btn = new Button ("Click me");
        Button exit= new Button("Exit");
        exit.setOnAction(e -> System.exit(0));
        btn.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent event) {
                System.out.println("Hello CSE-22");
            }
        });
        VBox root = new VBox();
        //StackPane root = new StackPane();
        //root.getChildren().add(btn);
        root.getChildren().addAll(btn, exit);
        Scene scene= new Scene(root, 500, 300);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```



Scene Builder

- Visual Layout tool
- Use a quickly design JavaFX application user interface

Scene Builder with JavaFX

- How we can download and install scene builder
- Integrate with Eclipse IDE
- Oracle doesn't provide the build for Scene Builder
- To do this, lets search on gluonhq.com

Useful Resource Links

- https://download.eclipse.org/efxclipse/updates-released/1.2.0/site/
- https://www.youtube.com/playlist?list=PLS1QulWo1RIaUGP446_pWLgTZPiFizEMq
- https://docs.oracle.com/javase/8/javafx/sample-apps/index.html

Projects

https://drive.google.com/drive/folders/1Ps_MoMS1npoDLt691hZAxliNHTPsOWe?usp=sharing