

Part 2 Java

Lecture 51:

Super Method

- By default, every constructor has super method as its first statement.
- super();
- Means it call the constructor of the super class.

```
class B extends A
{
    public B()
    {
        super();
        System.out.println(x: "in B");
    }
    public B(int n)
    {
        super();
        System.out.println(x: "in B int");
    }
}
```

Note: Every **class** in java extends from **Object** class.

```
class A extends Object
{
    public A()
    {
        super();
        System.out.println(x: "in A");
    }
}
```

This method

- It will execute the **constructor** of **same class**.

```

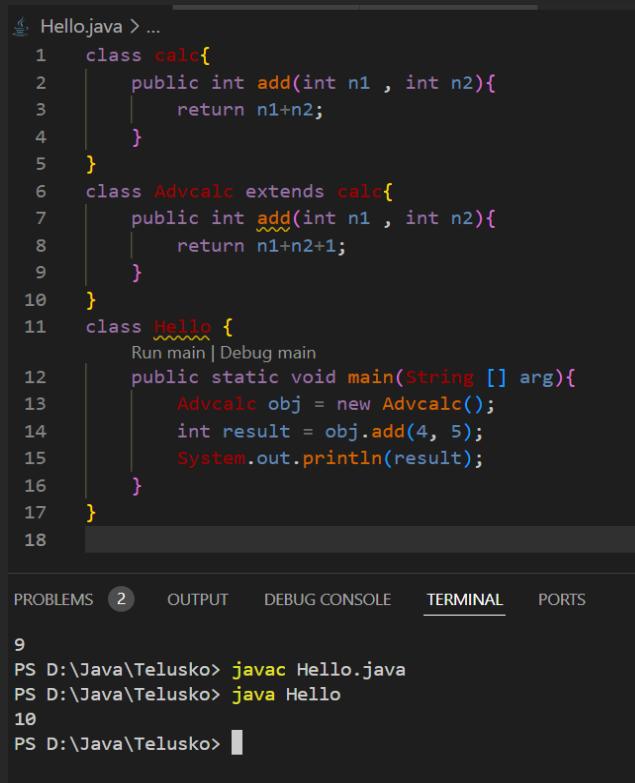
class B extends A
{
    public B()
    {
        super();
        System.out.println(x: "in B");
    }
    public B(int n)
    {
        this();
        System.out.println(x: "in B int");
    }
}

```

Lecture 52:

Method overriding

- Method of child class override the method of parent class with same name.



```

Hello.java > ...
1  class calc{
2      public int add(int n1 , int n2){
3          return n1+n2;
4      }
5  }
6  class Advcalc extends calc{
7      public int add(int n1 , int n2){
8          return n1+n2+1;
9      }
10 }
11 class Hello {
12     Run main | Debug main
13     public static void main(String [] arg){
14         Advcalc obj = new Advcalc();
15         int result = obj.add(4, 5);
16         System.out.println(result);
17     }
18 }

```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

9
PS D:\Java\Telusko> javac Hello.java
PS D:\Java\Telusko> java Hello
10
PS D:\Java\Telusko>

```




	Private	Protected	Public	Default
Same class	Yes	Yes	Yes	Yes
Same package subclass	NO	Yes	Yes	Yes
Same package non-subclass	NO	Yes	Yes	Yes
Different package subclass	NO	Yes	Yes	NO
Different package non-subclass	NO	NO	Yes	NO

Which one to use ?

- Try to make your classes **public**
- Try to keep your instance variable **private**.
- Methods most of the time will be **public** .
- Want to access only in sub class outside the package make it **Protected**
- Avoid **default**.

Lecture 55:

Polymorphism

- Many behaviors
- Polymorphism allows us to perform a single action in different ways.
- Poly – **Many** and Morph – **Forms**

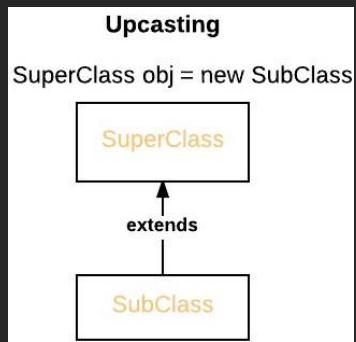
Types of Java Polymorphism

1. **Compile time**: which method will be executed decide at compile time
 - Achieved by **function Overloading**
2. **Run time**: which method will be executed decide at run time
 - Achieved by **function Overriding**.
 - It is also known as **Dynamic Method Dispatch**.

Lecture 56:

Dynamic Method Dispatch

- Implementation of run time polymorphism.
- A superclass reference variable can refer to a subclass object. This is also known as **upcasting**.



- Dynamic method dispatch: It is a process through which a call to overridden method is resolved during runtime rather than compile time.

```

class A{
    public void show(){
        System.out.println("in show A");
    }
}
class B extends A{
    public void show(){
        System.out.println("in show B");
    }
}
class C{
    public void show(){
        System.out.println("in show C");
    }
}
class Hello {
    Run main | Debug main | Run | Debug
    public static void main(String [] arg){
        A obj = new A();
        obj.show();

        obj = new B();
        obj.show();

        // obj = new C(); // error because C class is not
        // child of A. We can assign Object of child class to
        // reference variable of parent class
    }
}

```

Lecture 57:

Final Keyword

- Can be used with : variable, method and class

Final Variable: `final` make variable **constant**.

```

public class Demo
{
    public static void main(String a[])
    {
        final int num = 8;
        num = 9;
        System.out.println(num);
    }
}

```

- Ex:

Final Class:

- Final class can't be inherited.

Final Method:

- Final method can't be Overridden.

```
class A{
    public final void show(){
        System.out.println("in show A");
    }
    public void cal(int a, int b){
        System.out.println(a+b);
    }
}
class B extends A{
    public void show(){
        show() in B cannot override show() in A
        overridden method is final (errors(1): 10:5-10:24)
    }
    Add @Override Annotation (hints(1): 10:17-10:21)
    Cannot override the final method from A Java(67109265)
}
B
public void show()
Go to Super Implementation
void B.show()
View Problem (Alt+F8) Quick Fix... (Ctrl+.)
```

Ex:

Lecture 58:

Object Class

- Every class of java extends the object class.
- Every time you print an object it will call to string method even you don't mention.
System.out.println(obj.toString());
- And he explain how to work with Object class method

```
1  class Laptop{  
2      String model;  
3      int price;  
4      public String toString(){ //Don't re define Object  
5          class methods  
6          return model + " : " + price;  
7      }  
8      public boolean equals(Laptop that){  
9          return this.model == that.model && this.price == that.  
10         price;  
11     }  
12 }  
13 Run main | Debug main  
14 class Hello {  
15     Run | Debug  
16     public static void main(String [] arg){  
17         Laptop obj = new Laptop();  
18         obj.model = "Lenovo";  
19         obj.price = 56444;  
20         System.out.println(obj);  
21  
22         Laptop obj1 = new Laptop();  
23         obj1.model = "Lenovo";  
24         obj1.price = 56444;  
25         System.out.println(obj1);  
26         System.out.println(obj.equals(obj1));  
27     }  
28 }
```

```
PS D:\Java\Telusko> javac Hello.java  
PS D:\Java\Telusko> java Hello  
Lenovo : 56444  
Lenovo : 56444  
true  
PS D:\Java\Telusko> 
```

- Watch that video again if want understand more in 1.5x

Lecture 59:

Upcasting and Downcasting

Typecasting, also known as type conversion, is the process of changing the value of one data type into another.

```

1  class A{
2      public void show(){
3          System.out.println("in show A");
4      }
5  }
6  class B extends A{
7      public void show1(){
8          System.out.println("in show B");
9      }
10 }
11 class Hello {
12     Run main | Debug main | Run | Debug
13     public static void main(String [] arg){
14         //UPCASTING
15         A obj = new B(); // Upcasting :
16         // reference variable is parent type
17         // and object is child type
18         //child object is implicitly
19         //converted to parent type
20         // same like storing int in long
21         obj.show();
22
23         //DOWNCASTING
24         B obj1 = (B)obj;// Downcasting :
25         // reference variable is child type
26         // and object is parent type
27         //parent object have convert child
28         //type explicitly
29         // same like storing long in int
30         obj1.show();
31         obj1.show1();
32     }
33 }

```

PS D:\Java\Telusko> **javac** Hello.java
 PS D:\Java\Telusko> **java** Hello
 in show A
 in show A
 in show B
 PS D:\Java\Telusko>

Lecture 60:

Java is Not Purely Object-Oriented Language

Why -

1. Java Supports **primitive data types** like – byte, short, int, long, float, double, char, and Boolean. Which are not extending from any class. Means are not object.
 - It helps java to improve ProFormance
2. Java supports **static methods and variables**:
 - Static methods and variables can be accessed without creating an object of the class. This violates the principle of encapsulation, which is one of the fundamental principles of object-oriented programming.

Wrapper Classes

- In Java, there are **eight primitive** data types: byte, short, int, long, float, double, char, and boolean.
- **Wrapper classes** provide a way to **treat primitive** data types as **objects**.
- This allows you to use the primitive data types in places where only objects are accepted, such as in collections frameworks.
- wrapper classes are Integer, Double, Boolean, Character, Byte, Short, Long, and Float.

Boxing, unboxing and parseInt();

The screenshot shows a Java code editor and a terminal window. The code editor displays the file `Hello.java` with the following content:

```
1 class Hello {  
    Run main | Debug main | Run | Debug  
2     public static void main(String [] args){  
3         int num = 8;  
4         // boxing  
5         Integer num1 = num; // boxing :  
6         // converting a primitive data type into  
7         // its corresponding wrapper class object.  
8         // this is also called autoboxing  
9         // because it happen automatically in java  
10        System.out.println(num1);  
11  
12        // unboxing  
13        int num2 = num1.intValue(); // unboxing  
14        // : automatic conversion of wrapper class  
15        // objects to their corresponding primitive  
16        // data types.  
17        //int num2 = num1; auto-unboxing  
18        System.out.println(num2);  
19  
20        // parseInt() is a static method of  
21        // Integer wrapper class  
22        // It is used to parse the string  
23        // representation of an integer and convert  
24        // it into its corresponding primitive int  
25        // value  
26        String str = "34";  
27        int value = Integer.parseInt(str);  
28        System.out.println(value*2);  
29    }  
30}
```

The terminal window to the right shows the command `javac Hello.java` followed by `java Hello`. The output is:

```
PS D:\Java\Telusko> javac Hello.java  
PS D:\Java\Telusko> java Hello  
8  
8  
68  
PS D:\Java\Telusko>
```

Lecture 61:

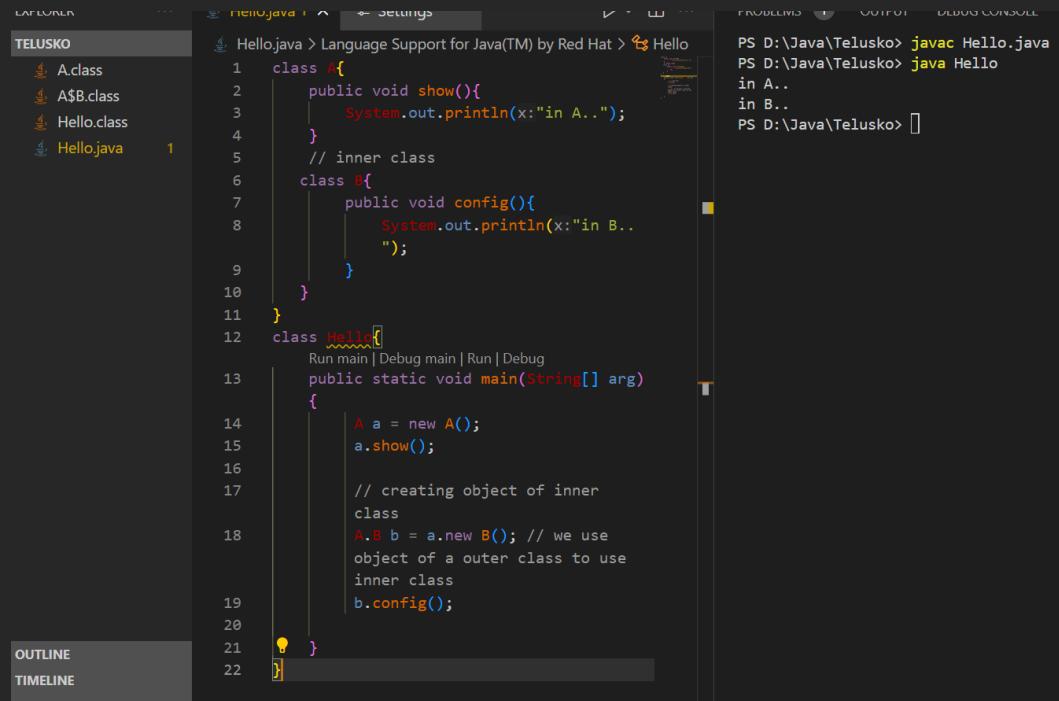
Abstract Keyword

- An **abstract** method belongs to an **abstract** class, and it does not have a body. The body is provided by the subclass.
- Subclass which has the abstract method definition know as- **concrete class**
- We can't create object of abstract class.

The screenshot shows a Java code editor interface with a dark theme. The code editor displays three files: `Hello.java`, `Car.java`, and `WagonR.java`. The `Hello.java` file contains the following code:1 abstract class Car{
2 public abstract void drive(); //Abstract method only
3 declaration, no definition
4 public void playMusic(){
5 System.out.println("playInG mUsic..");
6 }
7 class WagonR extends Car{
8 public void drive(){ // Must have to define all
9 Abstract methods in child class of abstract class
10 System.out.println("DriVing...");
11 }
12 }
13 class Hello{
14 Run main | Debug main | Run | Debug
15 public static void main(String[] args){
16 // Car obj = new Car(); // Car is abstract; cannot be
17 instantiated
18 Car obj = new WagonR(); // Abstract class reference
19 variable can be created
20 obj.drive();
21 obj.playMusic();
22 }
23 }
24}The code editor highlights several parts of the code in different colors. The `drive()` method in the `Car` class is declared as **abstract**. The `WagonR` class extends `Car` and overrides the `drive()` method. The `Hello` class contains a `main` method that creates an instance of `WagonR` and calls its `drive()` and `playMusic()` methods. The `playMusic()` method is defined in the `Car` class. The code editor also shows a terminal window on the right side with the output of the `javac` and `java` commands.

Lecture 61:

Inner Class



The screenshot shows a Java code editor with the file `Hello.java` open. The code defines three classes: `A`, `B`, and `Hello`. The `A` class has a `show` method that prints "in A..". The `B` class has a `config` method that prints "in B..". The `Hello` class contains a main method that creates an `A` object and calls its `show` method. It also creates an `B` object using the `new` operator on the `A` object and calls its `config` method.

```
1  class A{
2      public void show(){
3          System.out.println("in A..");
4      }
5      // inner class
6      class B{
7          public void config(){
8              System.out.println("in B..");
9          }
10     }
11 }
12 class Hello{
13     public static void main(String[] args)
14     {
15         A a = new A();
16         a.show();
17
18         // creating object of inner
19         // class
20         A.B b = a.new B(); // we use
21         // object of a outer class to use
22         // inner class
23         b.config();
24     }
25 }
```

- Use static for inner class, so no need create object of outer class to use inner class.
- Outer class can't be static.

Lecture 62:

Anonymous Class

The screenshot shows a Java code editor interface with the following details:

- EXPLORER** pane: Shows files Aclass, Hello.class, Hello.java (2), and Hello\$1.class.
- EDITOR** pane: Displays the code for `Hello.java`.

```
1  class A{
2      public void show(){
3          System.out.println("in A..");
4      }
5  }
6  class Hello{
7      public static void main(String[] args)
8      {
9          // Anonymous Class
10         A obj = new A()
11         {
12             public void show(){
13                 System.out.println("in
14                 new A..");
15         };
16     }; // this is inner class of
17     // redefine A
18     obj.show();
19 }
```
- PROBLEMS** pane: Shows 2 errors:

```
PS D:\Java\Telusko> javac Hello.java
PS D:\Java\Telusko> java Hello
in new A..
PS D:\Java\Telusko>
```

Lecture 63:

Abstract and Anonymous inner class

The screenshot shows a Java development environment with the following details:

- EXPLORER**: Shows files: A.class, Hello.class, Hello.java (2), and Hello\$1.class.
- EDITOR**: Displays the code for `Hello.java`. The code defines an abstract class `A` with an abstract method `show()`. It then defines an anonymous inner class `Hello` that overrides the `show()` method to print "in new A.". Finally, it creates an object of `A` and calls its `show()` method.
- PROBLEMS**: Shows 2 errors.
- OUTPUT**: Shows the terminal output of running `javac Hello.java` and then `java Hello`, which prints "in new A.".

```
abstract class A{
    //abstract method
    public abstract void show();
}

class Hello{
    Run main | Debug main | Run | Debug
    public static void main(String[] args)
    {
        A obj = new A()
        {
            // abstract method definition
            // inside Anonymous class
            public void show(){
                System.out.println("in
new A.");
            }
        };
        // Here object of abstract class
        // A is not created
        // Object of anonymous class is
        // created
        obj.show();
    }
}
```

Lecture 65:

Need of Interface in Java

The screenshot shows a Java development environment with a code editor and a terminal window.

Code Editor (Hello.java):

```
1 // Developer use computer maybe laptop or desktop to build app
2 interface Computer{
3     void codeRun();
4 }
5 class Laptop implements Computer{
6     public void codeRun(){
7         System.out.println("running code in laptop..");
8     }
9 }
10 class Desktop implements Computer{
11     public void codeRun(){
12         System.out.println("running code in desktop..");
13     }
14 }
15 class Developer{
16     public void devApp(Computer lap){
17         System.out.println("coding..");
18         lap.codeRun();
19     }
20 }
21 class Hello{
22     Run main | Debug main | Run | Debug
23     public static void main(String[] args){
24         Laptop lap = new Laptop();
25         Desktop desk = new Desktop();
26         Developer dev = new Developer();
27         dev.devApp(desk);
28         dev.devApp(lap);
29     }
30 }
```

Terminal:

```
PS D:\Java\Telusko> javac Hello.java
PS D:\Java\Telusko> java Hello
coding..
running code in desktop..
coding..
running code in laptop..
PS D:\Java\Telusko>
```

Lecture 66:

What is Interface?

Interfaces

Another way to achieve abstraction in Java, is with interfaces.

An `interface` is a completely "**abstract class**" that is used to group related methods with empty bodies:

Example

Get you

```
// interface
interface Animal {
    public void animalSound(); // interface method (does not have a body)
    public void run(); // interface method (does not have a body)
}
```

- Interface is not a class
- **Abstraction** in Java is the process in which we only show essential details/functionality to the user. The non-essential implementation details are not displayed to the user.
- To access the interface methods, the interface must be "implemented" (kind of like inherited) by another class with the `implements` keyword (instead of `extends`)

Notes on Interfaces:

- Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the `MyMainClass`)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default `abstract` and `public`
- Interface attributes are by default `public`, `static` and `final`
- An interface cannot contain a constructor (as it cannot be used to create objects)

Why And When To Use Interfaces?

- 1) To achieve security - hide certain details and only show the important details of an object (interface).
- 2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

- **attribute** is a variable that is declared within a class. Also known as instance variable.

Lecture 67:

More on Interface

The screenshot shows a Java code editor with the file `Hello.java` open. The code demonstrates multiple inheritance where the class `Toyota_01` implements both the `Car` and `Toyota` interfaces. The `Car` interface defines a method `showCar()`, and the `Toyota` interface defines a method `showToyota()`. The `Toyota_01` class implements both, providing its own implementations for each. The `Hello` class contains a `main` method that creates an `Toyota_01` object and calls `showCar()`. A terminal window to the right shows the output of the `javac Hello.java` and `java Hello` commands, which prints "function of Car".

```
D:\Java\Telusko>Hello.java • 3 problems in this file Red Hat > Toyota_01 > showCar()
1 // Multiple inheritance
2 // Class Toyota_01 implements interface Car and Toyota
3 interface Car{
4     void showCar();
5 }
6 interface Toyota{
7     void showToyota();
8 }
9 class Toyota_01 implements Car, Toyota{
10    // defining the interface abstract methods
11    public void showCar(){
12        System.out.println(x:"function of Car");
13    }
14    public void showToyota(){
15        System.out.println(x:"function of the
16        Toyota");
17    }
18 class Hello{
19     Run main | Debug main | Run | Debug
20     public static void main(String[] args){
21         Car obj = new Toyota_01();
22
23         obj.showCar();
24         // obj.showToyota(); Error // reference
25         // variable is of type Car so, can't access
26         // methods of Toyota interface
27     }
28 }
```

```
PS D:\Java\Telusko> javac Hello.java
PS D:\Java\Telusko> java Hello
function of Car
PS D:\Java\Telusko>
```

Lecture 68:

Enum (Enumeration)

- Used to create named constants.

Enums

An `enum` is a special "class" that represents a group of **constants** (unchangeable variables, like `final` variables).

To create an `enum`, use the `enum` keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters:

Example

[Get your own Java Server](#)

```
enum Level {  
    LOW,  
    MEDIUM,  
    HIGH  
}
```

You can access `enum` constants with the **dot** syntax:

```
Level myVar = Level.MEDIUM;
```

[Try it Yourself »](#)

Enum is short for "enumerations", which means "specifically listed".

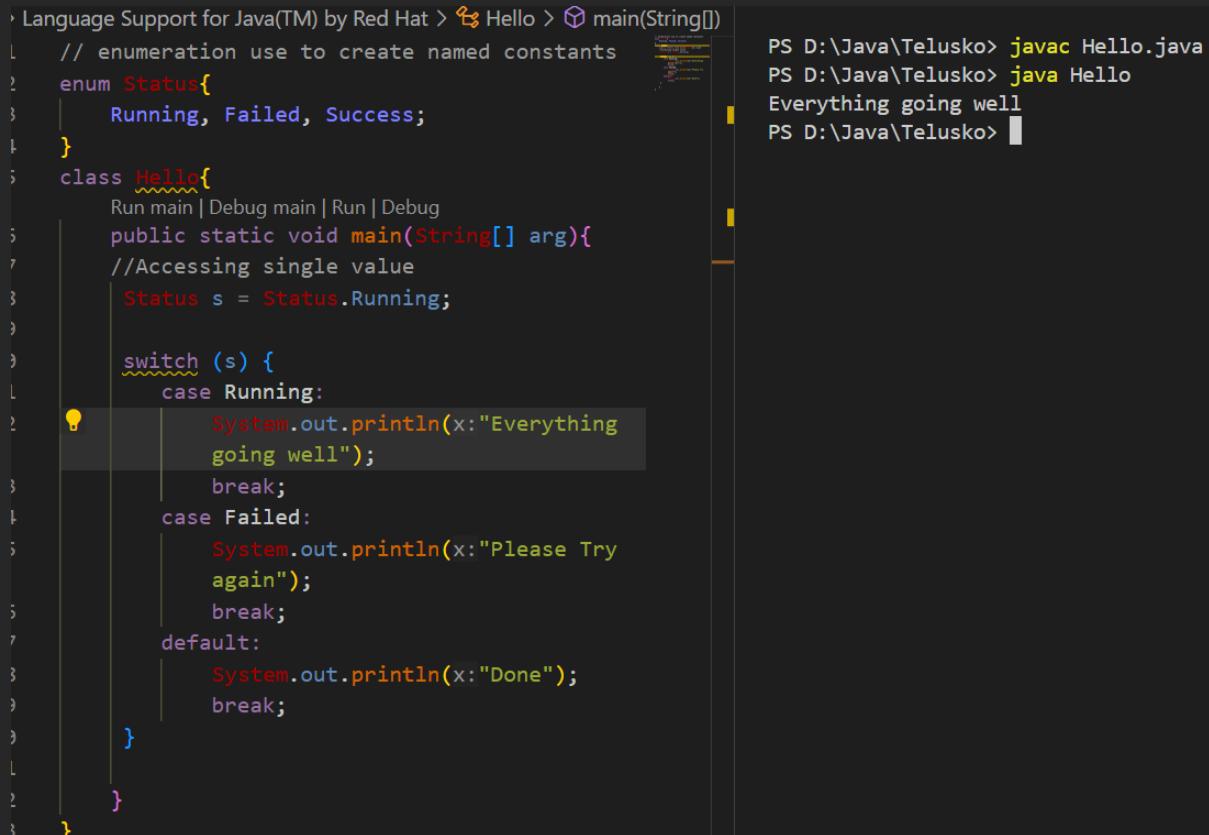
The screenshot shows an IDE interface with two panes. The left pane displays a Java file named `Hello.java`. The code defines an enum `Status` with three values: `Running`, `Failed`, and `Success`. It then defines a class `Hello` with a `main` method. Inside the `main` method, there is code to print the ordinal value of the enum constant `Running`. The right pane shows a terminal window with the command `java Hello` being run, and the output shows the values `Running`, `Failed`, and `Success`.

```
hello.java > Language Support for Java(TM) by Red Hat > ⌂ Hello >  
// enumeration use to create named  
constants  
enum Status{  
    Running, Failed, Success;  
}  
class Hello{  
    Run main | Debug main | Run | Debug  
    public static void main(String[]  
    arg){  
        //Accessing single value  
        Status s = Status.Running;  
        System.out.println(s.ordinal());  
  
        // Accessing all values  
        Status[]ss = Status.values();  
        for(Status ele: ss){  
            System.out.println(ele);  
        }  
    }  
}
```

```
PS D:\Java\Telusko> javac Hello.java  
PS D:\Java\Telusko> java Hello  
0  
Running  
Failed  
Success  
PS D:\Java\Telusko>
```

Lecture 69:

Enum in Switch



The screenshot shows an IDE interface with two panes. The left pane displays a Java file named Hello.java. The code defines an enum Status with three values: Running, Failed, and Success. It then creates a class Hello with a main method. Inside the main method, a switch statement is used to print different messages based on the value of s. The right pane shows the terminal output of running the java compiler (javac) and the resulting application (Hello). The application prints "Everything going well".

```
Language Support for Java(TM) by Red Hat > Hello > main(String[])
1 // enumeration use to create named constants
2 enum Status{
3     Running, Failed, Success;
4 }
5 class Hello{
6     Run main | Debug main | Run | Debug
7     public static void main(String[] args){
8         //Accessing single value
9         Status s = Status.Running;
10
11         switch (s) {
12             case Running:
13                 System.out.println("Everything
14                     going well");
15                 break;
16             case Failed:
17                 System.out.println("Please Try
18                     again");
19                 break;
20             default:
21                 System.out.println("Done");
22                 break;
23         }
24     }
25 }
```

```
PS D:\Java\Telusko> javac Hello.java
PS D:\Java\Telusko> java Hello
Everything going well
PS D:\Java\Telusko>
```

Lecture 70:

Enum Class In java

Difference between Enums and Classes

An `enum` can, just like a `class`, have attributes and methods. The only difference is that enum constants are `public`, `static` and `final` (unchangeable - cannot be overridden).

An `enum` cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).

Why And When To Use Enums?

Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

```

Hello.java > Language Support for Java(TM) by Red Hat > Laptop > Laptop(int)
// enumeration use to create named constants
enum Laptop{
    Macbook(price:89990), AsusTuf, AsusRog(price:99880),
    LenovoIdea(price:67900);

    private int price;

    // Default constructor
    private Laptop() {
        price = 500;
        System.out.println("in default constructor : " +
                           this);
    }

    // Parameterized constructor
    private Laptop(int price) {
        this.price = price;
        System.out.println("in constructor : " + this);
    }

    public int getPrice() {
        return price;
    }
}

class Hello{
    Run main | Debug main | Run | Debug
    public static void main(String[] args){

        for(Laptop lap : Laptop.values()){
            System.out.println(lap + " : " + lap.getPrice());
        }
    }
}

```



```

PS D:\Java\Telusko> javac Hello.java
PS D:\Java\Telusko> java Hello
in constructor : Macbook
in default constructor : AsusTuf
in constructor : AsusRog
in constructor : LenovoIdea
Macbook : 89990
AsusTuf : 500
AsusRog : 99880
LenovoIdea : 67900
PS D:\Java\Telusko>

```