

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 accesses 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**

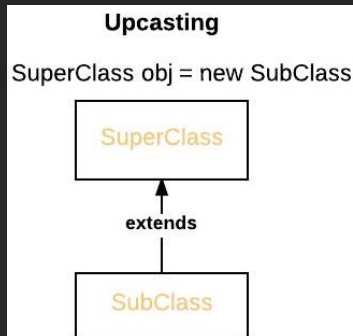
Types of Java Polymorphism

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(x:"in show A");
    }
}
class B extends A{
    public void show(){
        System.out.println(x:"in show B");
    }
}
class C{
    public void show(){
        System.out.println(x:"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(x:"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()

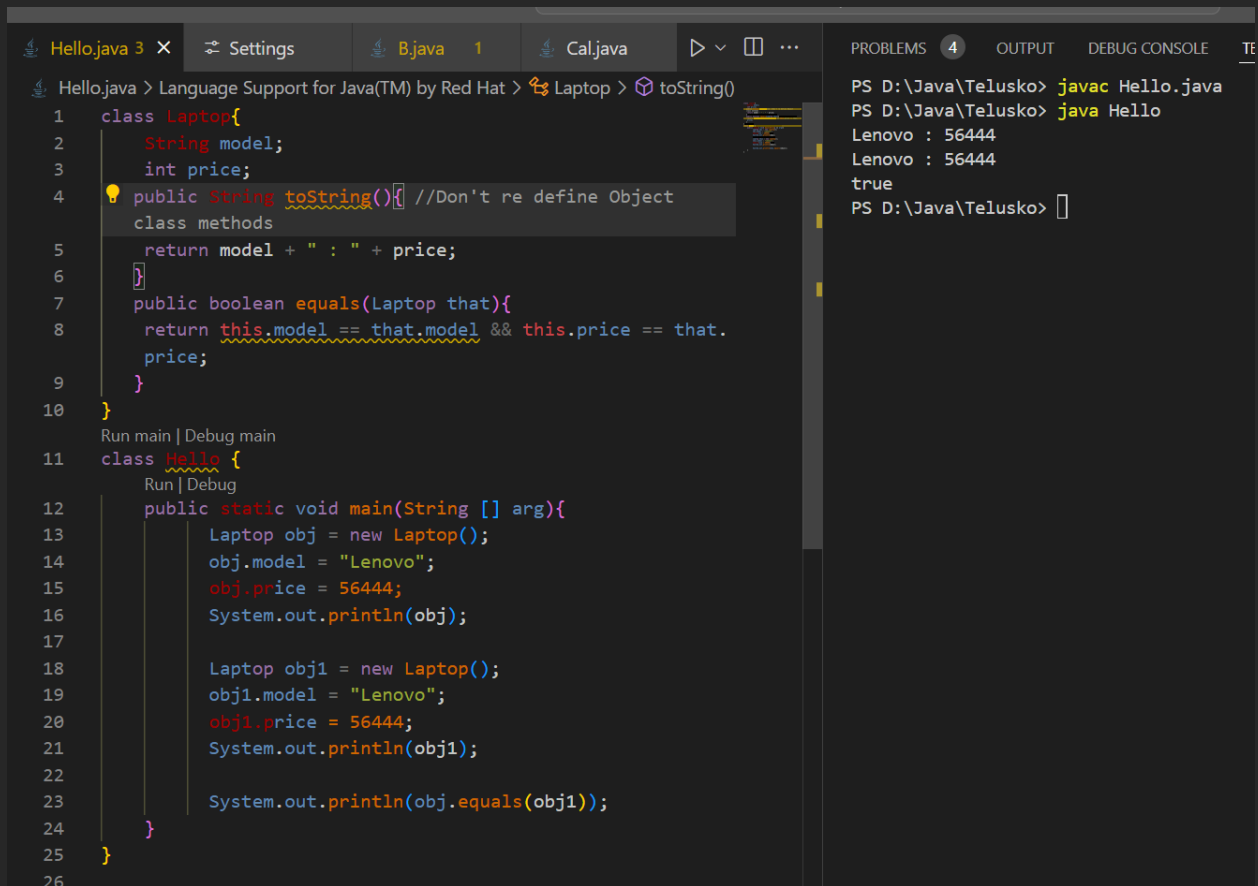
[View Problem \(Alt+F8\)](#) [Quick Fix... \(Ctrl+.\)](#)

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         return model + " : " + price;
6     }
7     public boolean equals(Laptop that){
8         return this.model == that.model && this.price == that.
9             price;
10    }
11 }
12 Run main | Debug main
13 class Hello {
14     Run | Debug
15     public static void main(String [] arg){
16         Laptop obj = new Laptop();
17         obj.model = "Lenovo";
18         obj.price = 56444;
19         System.out.println(obj);
20
21         Laptop obj1 = new Laptop();
22         obj1.model = "Lenovo";
23         obj1.price = 56444;
24         System.out.println(obj1);
25
26         System.out.println(obj.equals(obj1));
27     }
28 }
```

PROBLEMS 4 OUTPUT DEBUG CONSOLE TE

```
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.


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

```
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();

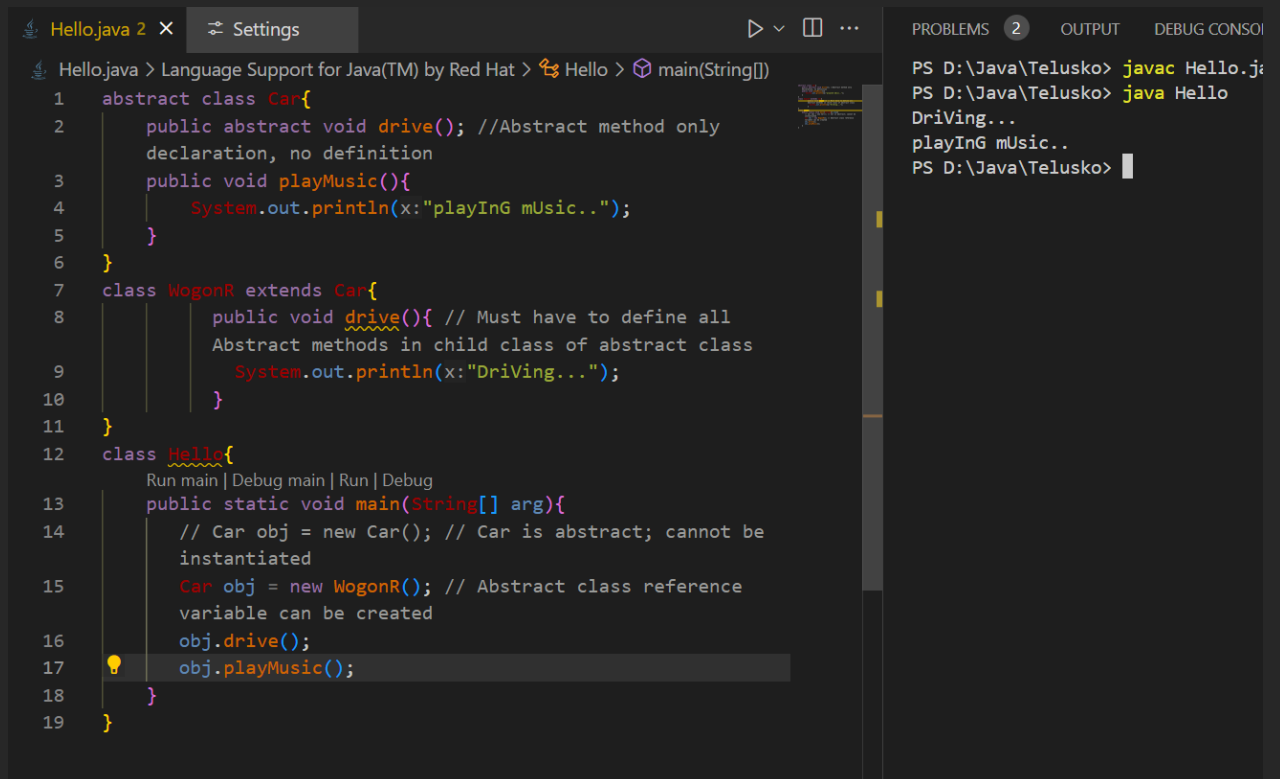
```
Hello.java > Language Support for Java(TM) by Red Hat > Hello > main(St
1  class Hello {
    Run main | Debug main | Run | Debug
2  public static void main(String [] arg){
3      int num = 8;
4      // boxing
5      Integer num1 = num; // boxing :
        converting a primitive data type into
  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

```
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.



```

Hello.java 2 x  Settings
Hello.java > Language Support for Java(TM) by Red Hat > Hello > main(String[])
1  abstract class Car{
2      public abstract void drive(); //Abstract method only
   declaration, no definition
3      public void playMusic(){
4          System.out.println(x:"playInG mUsic..");
5      }
6  }
7  class WogonR extends Car{
8      public void drive(){ // Must have to define all
   Abstract methods in child class of abstract class
9          System.out.println(x:"DrIVING...");
10     }
11 }
12 class Hello{
   Run main | Debug main | Run | Debug
13     public static void main(String[] arg){
14         // Car obj = new Car(); // Car is abstract; cannot be
   instantiated
15         Car obj = new WogonR(); // Abstract class reference
   variable can be created
16         obj.drive();
17         obj.playMusic();
18     }
19 }

```

PROBLEMS 2 OUTPUT DEBUG CONSOLE

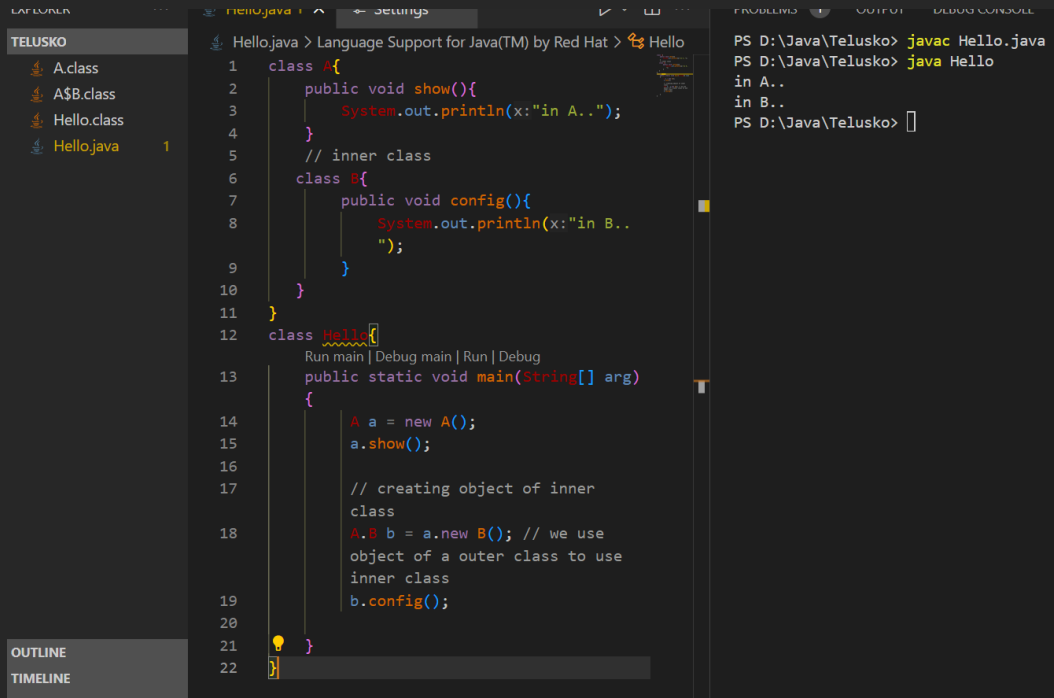
```

PS D:\Java\Telusko> javac Hello.java
PS D:\Java\Telusko> java Hello
Driving...
playInG mUsic..
PS D:\Java\Telusko>

```

Lecture 61:

Inner Class



The screenshot shows an IDE with a project named 'HELUSKO'. The file explorer on the left lists 'A.class', 'A\$B.class', 'Hello.class', and 'Hello.java'. The main editor displays the source code for 'Hello.java', which defines an outer class 'A' with a 'show()' method and an inner class 'B' with a 'config()' method. The 'main' method in 'A' creates an instance of 'A', calls 'show()', and then creates an instance of the inner class 'B' (accessed as 'A.B') and calls 'config()'. The right-hand pane shows the command prompt output, which confirms the execution of 'javac Hello.java' and 'java Hello', resulting in the printed messages 'in A..' and 'in B..'.

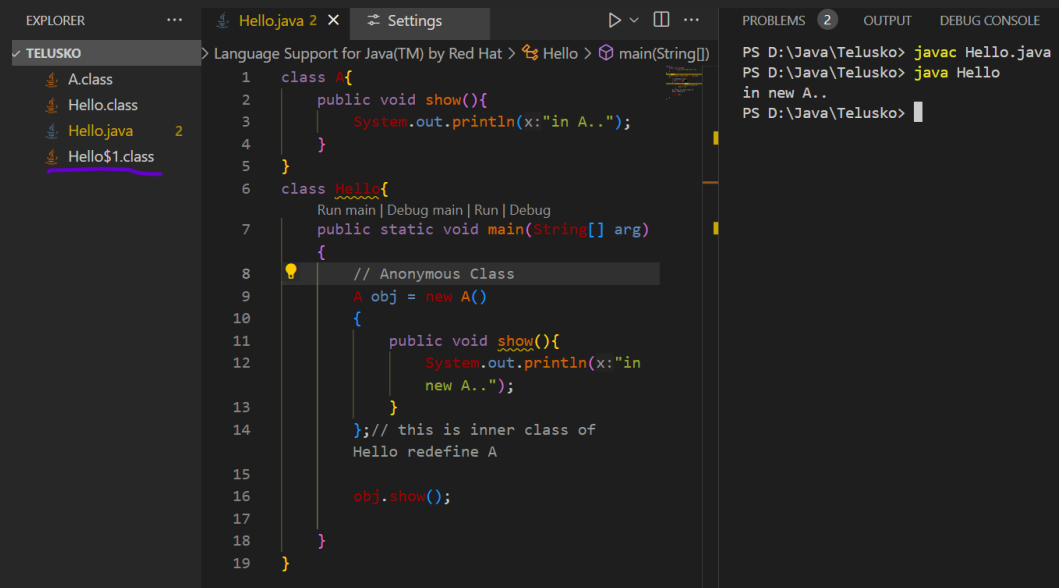
```
1 class A{
2     public void show(){
3         System.out.println(x:"in A..");
4     }
5     // inner class
6     class B{
7         public void config(){
8             System.out.println(x:"in B..");
9         }
10    }
11 }
12 class Hello{
13     public static void main(String[] arg)
14     {
15         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 }
```

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

- 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 an IDE with the following components:

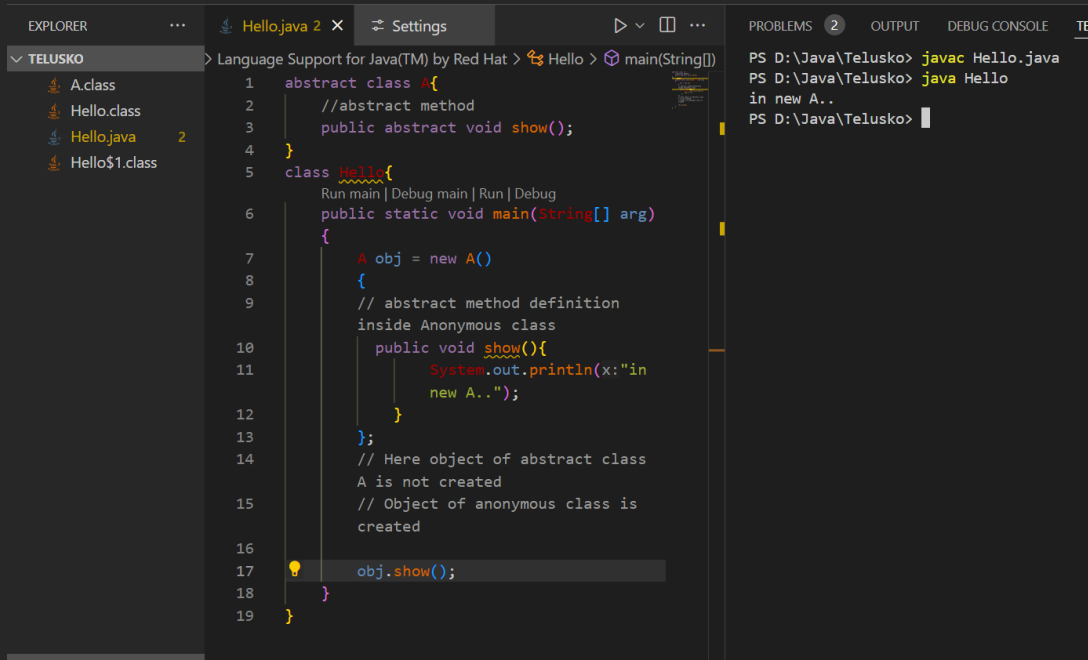
- EXPLORER:** A file tree for 'TELUSKO' containing 'A.class', 'Hello.class', 'Hello.java' (selected), and 'Hello\$1.class'.
- Editor:** Displays the code for 'Hello.java'. It contains two classes: 'A' and 'Hello'. Class 'A' has a 'show()' method that prints 'in A..'. Class 'Hello' has a 'main()' method that creates an instance of 'A' and calls 'show()'. An anonymous class is defined inside 'main()' that inherits from 'A' and overrides 'show()' to print 'in new A..'. Comments indicate this is an inner class of 'Hello' that redefines 'A'.
- PROBLEMS:** Shows 2 errors, but none are visible in the list.
- OUTPUT:** Displays the command prompt output: 'PS D:\Java\Telusko> javac Hello.java', 'PS D:\Java\Telusko> java Hello', and 'in new A..'.

```
1 class A{
2     public void show(){
3         System.out.println(x:"in A..");
4     }
5 }
6 class Hello{
7     public static void main(String[] arg)
8     {
9         // Anonymous Class
10        A obj = new A()
11        {
12            public void show(){
13                System.out.println(x:"in
14                new A..");
15            }
16        };// this is inner class of
17        Hello redefine A
18
19        obj.show();
20    }
21 }
```

```
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



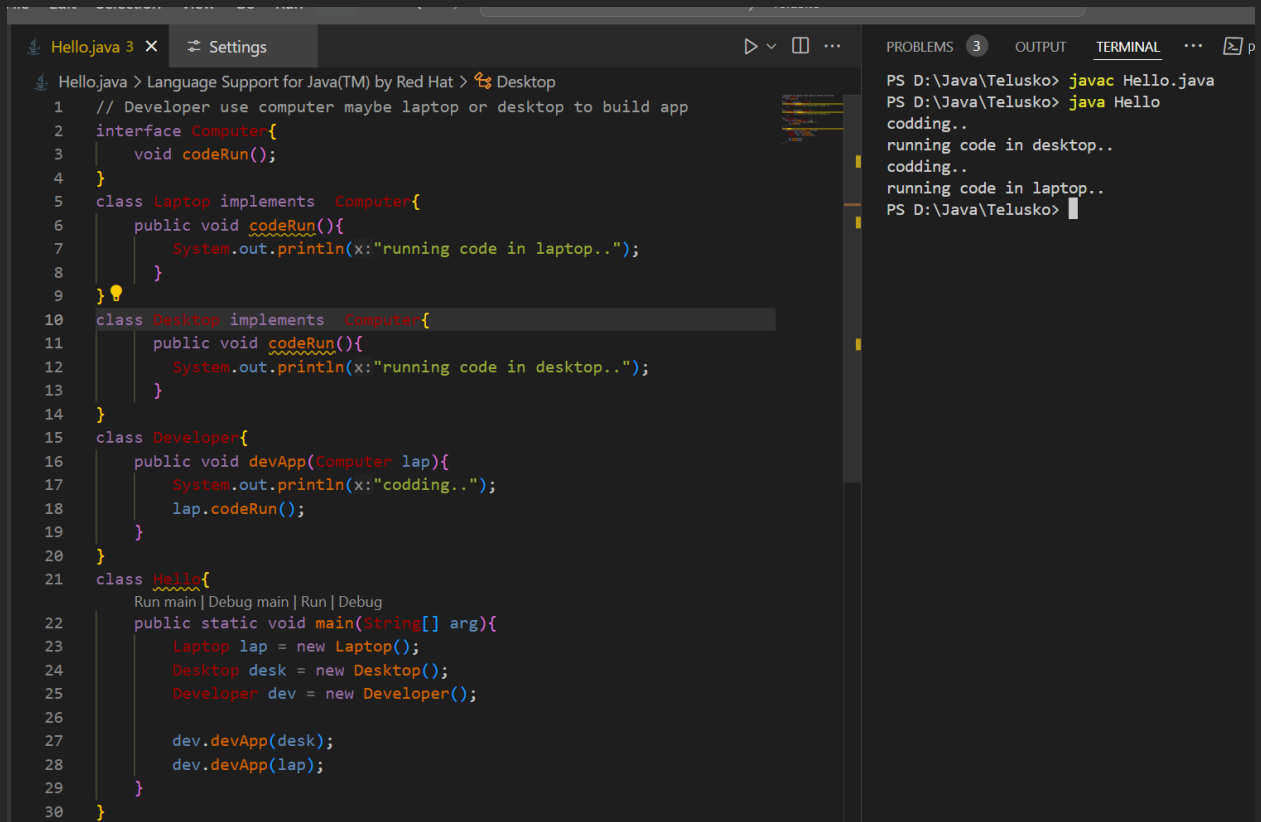
The screenshot shows an IDE with the following components:

- EXPLORER:** A project named 'TELUSKO' containing files 'A.class', 'Hello.class', 'Hello.java' (selected, with a '2' next to it), and 'Hello\$1.class'.
- Editor:** Displays the code for 'Hello.java'. The code defines an abstract class 'A' with an abstract method 'show()'. The 'Hello' class implements 'A' and contains a 'main' method. Inside 'main', an object 'obj' of type 'A' is created using an anonymous inner class that implements the 'show' method by printing 'in new A..'.

```
1  abstract class A{
2      //abstract method
3      public abstract void show();
4  }
5  class Hello{
6      Run main | Debug main | Run | Debug
7      public static void main(String[] arg)
8      {
9          A obj = new A()
10         {
11             // abstract method definition
12             // inside Anonymous class
13             public void show(){
14                 System.out.println(x:"in
15                 new A..");
16             }
17         };
18         // Here object of abstract class
19         // A is not created
20         // Object of anonymous class is
21         // created
22         obj.show();
23     }
24 }
```
- PROBLEMS:** Shows 2 errors. The first error is 'The method show() is undefined for the type A' at line 17, column 17. The second error is 'The method show() is undefined for the type A' at line 17, column 17.
- OUTPUT:** Shows the command 'javac Hello.java' and the output 'PS D:\Java\Telusko> java Hello in new A.. PS D:\Java\Telusko>'.

Lecture 65:

Need of Interface in Java



The screenshot shows an IDE with a Java file named 'Hello.java'. The code defines an interface 'Computer' with a method 'codeRun()'. Two classes, 'Laptop' and 'Desktop', implement this interface. A 'Developer' class uses these objects to demonstrate the interface. The terminal shows the compilation and execution of the program, resulting in the output: 'coddng..' (twice) and 'running code in desktop..' (twice).

```
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(x:"running code in laptop..");
8     }
9 }
10 class Desktop implements Computer{
11     public void codeRun(){
12         System.out.println(x:"running code in desktop..");
13     }
14 }
15 class Developer{
16     public void devApp(Computer lap){
17         System.out.println(x:"coddng..");
18         lap.codeRun();
19     }
20 }
21 class Hello{
22     Run main | Debug main | Run | Debug
23     public static void main(String[] arg){
24         Laptop lap = new Laptop();
25         Desktop desk = new Desktop();
26         Developer dev = new Developer();
27
28         dev.devApp(desk);
29         dev.devApp(lap);
30     }
31 }
```

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

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)

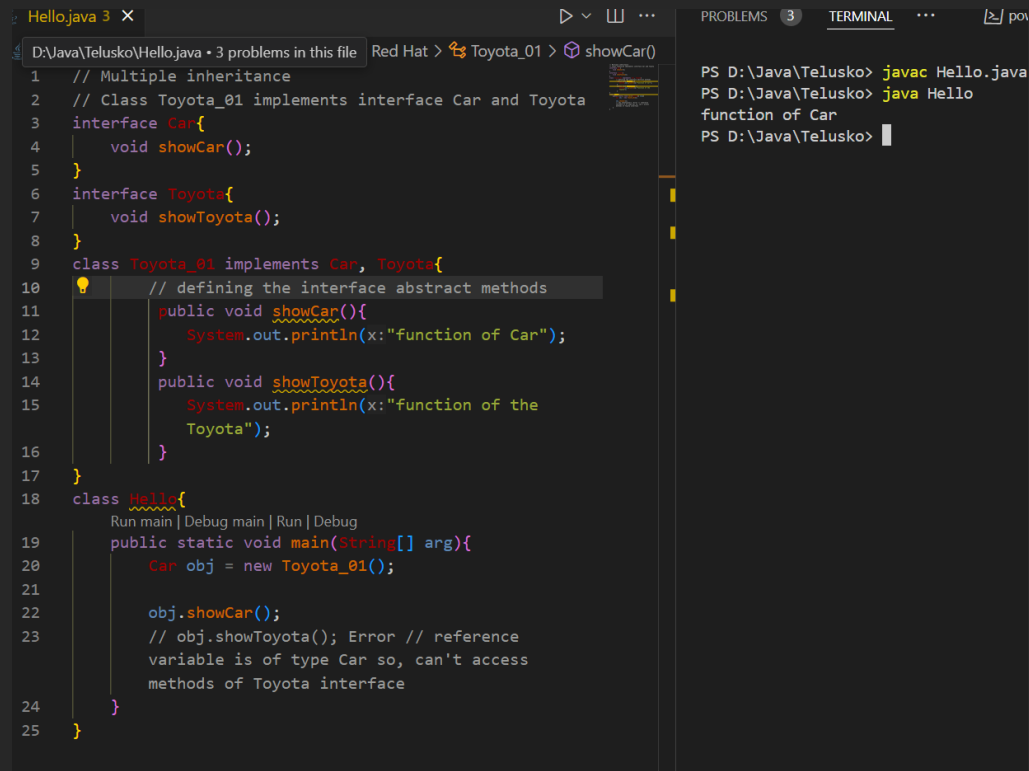
Why And When To Use Interfaces?

- 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 an IDE with a Java file named 'Hello.java'. The code defines two interfaces, 'Car' and 'Toyota', and a class 'Toyota_01' that implements both. The 'Car' interface has a method 'showCar()', and the 'Toyota' interface has a method 'showToyota()'. The 'Toyota_01' class implements both methods. A 'Hello' class contains a 'main' method that creates a 'Toyota_01' object and calls 'showCar()'. A comment indicates that calling 'showToyota()' would result in an error because the reference variable is of type 'Car'.

```
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 }
19 class Hello{
20     Run main | Debug main | Run | Debug
21     public static void main(String[] arg){
22         Car obj = new Toyota_01();
23
24         obj.showCar();
25         // obj.showToyota(); Error // reference
26         variable is of type Car so, can't access
27         methods of Toyota interface
28     }
29 }
```

The terminal window on the right shows the following commands and output:

```
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".

ello.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

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[] arg){
8         //Accessing single value
9         Status s = Status.Running;
10
11         switch (s) {
12             case Running:
13                 System.out.println(x:"Everything
14                     going well");
15                 break;
16             case Failed:
17                 System.out.println(x:"Please Try
18                     again");
19                 break;
20             default:
21                 System.out.println(x:"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.

llo.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[] arg){

        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>