OOP with Java (Revision Purpose)

Created	@November 23, 2023 6:00 PM
Last edited time	@November 25, 2023 9:09 PM
Created by	Ⓑ Borhan
Tags	CSTE Year 2 Term 2

Lecture 7 to 12:

Access Modifier:

- Public
- Private : Can be accessed only by other members of its class
- Protected

There is Default access modifier too,

Boundary Error: **An invalid value entered into an application**. For example, if a number is higher or lower than a range of values or there are too many characters in a text entry, a boundary error occurs.

Java is always Pass by value. A few ways to achieve call-by-reference

- Making a public member in class

```
class myClass{
  public int a;
  public void update(myClass ob){
    ob.a  = 100;
  }
};
class Main{
  public static void main(String []args){
    myClass ob=new myClass();
    ob.a = 5;
    System.out.println(ob.a);
  ob.update(ob);
    System.out.println(ob.a);
  }
}
```

- Return a value from a function and update it

```
class Main{
  public static int update(int x){
      x++;
    return x;
  }
  public static void main(String []args){
    int i = 5;
    System.out.println(i);
    i = update(i);
    System.out.println(i);
  }
}
```

- Creating array/single element array

```
class Main{
  public static void update(int []x){
      x[0]++;
  }
  public static void main(String []args){
    int [] i = {1};
    System.out.println(i[0]);
        update(i);
    System.out.println(i[0]);
  }
}
```

Method Overloading

- Two or more methods withing the same class can share the **same name as long as their parameter** declarations are different

- Polymorphism
- Automatic type conversation also happens while passing the parameters
- The type and/or number of parameters must differ
- It is not sufficient for two methods to differ only in their return types

```
// NOT SUfficient adn will casue an error
class Klass{
  public int func(int a, int b, int c){
    ....
  }
  public double func2(int a, int b, int c){
    .....
  }
};
```

- Constructor can be overloaded too
  - Constructors are overloaded is to allow one object to initialize another

```
class myclass(){
  int x;
  myclass(int y){
    x=y;
  }
  myclass(myclass ob){
    x = ob.x;
  }
}
```

Static Variable

- Why should we use ?
  - To save memory
- Static variable is not related to object, it is related to class
  - when object created no copy of static variable is made

- Works like a Global variable
- We can use static variable, without object

```
class myclass{
  static int t = 1;
};

class Main{
  public static void main(string []args){
    myclass.t = 10;
    ..
  }
};
```

- Why main() is Static ?
    - The main() method is marked static **so that the JVM may call it without having to create an instance of the class that contains the main() function**. Since no class object is existing when the java runtime starts, we must declare the main() function static

Static Method

- Can be called without creating any instance/object of the class
- **Restrictions**
    - They can directly **call only other static methods**
    - They can directly **access only static variables**
    - They **do not have a this reference**

Static Blocks

- When **a class will require some type of initialization before it is ready to create objects**
    - Example : Establish connection to a remote site, initializing static variables
- It is executed when the class is first loaded (even before constructor)

- **Static block is called for once time** (even multiple objects are created)

```
class Test{
  static int x;
    static int y;
    static{
      System.out.println("Static");
    }
    Test(){
      System.out.println("Constructor");
    }
};
public class Main {
  public static void main(String[] args) {
    Test ob = new Test();
    Test ob2 = new Test();
  }
}
/*
Output:
Static
Constructor
Constructor
*/
```

Nested Class

- A class that is declared within another class
- A local class can be nested within a method (function)
- A nested class doesn’t exist independently of its enclosing class
- Type
  - Static
  - Non-Static
- Inner class and outer class
  - Inner class can access all members of outer class

Inheritance

- We can create a general **class that defines traits common to a set of related items**
- A class that is **inherited called superclass**
- The class that **does the inheriting is called a subclass**

```
class shape{
  public int x, y;
};

class circle extends shape{
  ....
}
// shape - superclass
//circle - subclass
```

- Java **doesn’t support the inheritance of multiple superclasses into a single subclass**
- Subclass can’t access private member of superclass directly.
  - To access private members, we need accessor

  ```
  class shape{
    private int x, y;
    int getX(){return x; } // accessor
    int getY() {return y;} //accessor
  };

  class circle extends shape{
    int c_x = getX();
  ...
  }
  ```

Constructor in Inheritance

- Subclass constructor calls automatically (according to parameter or default), if you didn’t mention any superclass constructor default constructor of superclass calls
- To call superclass constructor needs $super(parameter - list)$
- To access something of superclass we can use $super.member$

```
class A {
  int i;
  A(){
    System.out.prinln("A");
  }
  A(int x){
    System.out.prinln("A : X");
  }
}
class B extends A {
  B(){
    super(x);
    System.out.println("B");
    super.i = 10;
  };
};

// Outout
// A
// B
```

- $super$ must be first statement in constructor

  ```
  class A{
    A(int x){
        System.out.println("World");
      }
  }

  class B extends A{
    B(){
        System.out.println("Hello");
          super(5);
      }
  }

  public class Main {
    public static void main(String[] args) {
      B obj = new B();
    }
  }
  // Error : error: call to super must be first statement in constructor super(5);
  ```

- If superclass has only parameterized constructors, then any of them must be called by using $super$

```
class A{
  A(int x){
      System.out.println("World");
    }
  A(int x, int y){
      System.out.println("World");
    }
}

class B extends A{
  B(){
      // super(5,6); // it will call A(int x, int y)
      /*if we don't mention the super(), then it will return an error*/
      System.out.println("Hello");
    }
}

public class Main {
  public static void main(String[] args) {
    B obj = new B();
  }
}
```

Method Overriding

- same return type and signature/parameters
- superclass method (which one is overridden) will be hidden
- To call superclass method (which one is overridden) we can use $super.method()$
- We can override methods only, not the variables

```
class A{
  void hey(){
      System.out.println("From A");
    }
}

class B extends A{
  void hey(){
      System.out.println("From B");
    }
    void callheyb(){
      super.hey();
    }
}

public class Main {
  public static void main(String[] args) {
    B obj = new B();
    obj.hey();
    obj.callheyb();
  }
}
/*
Output:
From A
From B
*/
```

Dynamic Method Dispatch

- an overridden method is resolved at run time rather than compile time
- run-time polymorphism
- How does it work in runtime ?
    - “When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time” (GFG)
- runtime polymorphism cannot be achieved by data members because we can override methods only

```
class A {
  void hello(){
    ...
  }
  void bye(){
    ...
  }
};

class B extends A{
  void howareyou(){
    ....
  }
  void bye(){
    ....
  }
};

public class Main(){
  .....
  A obj = new B(); // Superclass var = new SubClass();
  obj.hello(); //allowed
  obj.bye(); // allower
  obj.howareyou(); // not allowed
  ....
}
```