Chapter 1 –

Java Programming Constructs

- Mrs. Apurwa Barve

Fundamentals of OOP

- **Abstraction** :
  - ✔We focus on the essential qualities of some thing rather than one specific example and we automatically discard what is unimportant or irrelevant.
  - ✔Demonstrated through Data abstraction and Process abstraction
  - ✔Promotes maintability.
  - ✔Reduces complexity
  - ✔Simplifies things at design level
  - ✔Focus is on “WHAT”
  - ✔Java achieves abstraction through abstract class and interface

Activity 1.1

1. Consider a class, Car. Apply abstraction (data and procedural) to finalise the attributes and methods of this class. Discuss within the class how you will finalise the WHAT component of the class Car
2. Consider a class, Bottle. Apply abstraction (data and procedural) to finalise the attributes and methods of this class. Discuss within the class how you will finalise the WHAT component of the class Bottle
3. Consider a class Laptop. Apply abstraction (data and procedural) to finalise the attributes and methods of this class. Discuss within the class how you will finalise the WHAT component of the class Laptop

Fundamentals of OOP

- **Encapsulation** :
  - ✔Binding code and data together in a single unit
  - ✔Wrapping components together to keep them safe from misuse and manipulation
  - ✔A well defined interface controls the access to the code and data.
  - ✔Implemented through class, private access modifier and getter-setter methods

Activity 1.2

1. Consider a class BankAccount. Identify its data members i.e instance variables and methods.
2. Identify if we can implement encapsulation through accessors and mutators.
3. Code the class using the IDE.

Fundamentals of OOP

- **Polymorphism** :

✔One feature delivering different functionality depending on the situation.

✔One interface, multiple methods.

✔It is the job of the compiler to select specific functionality based on the situation.

✔Method overloading is the type of STATIC or COMPILE time polymorphism

✔Method overriding is the type of DYNAMIC or RUN time polymorphism

Activity 1.3.1 – Method overloading

1. Write a class Printer. Add some attributes of the same. In this class, add a method, print(). The class object invokes one version of print() for black and white prints and another for printing coloured prints. Achieve this using method overloading. Add a PolymorphismDemo class containing main(). Create objects of the necessary classes and invoke print() versions. Observe the output messages and understand the flow of the program.

   Refer Printer.java

Fundamentals of OOP

- **Inheritance** :

✔ Allows creation of hierarchies having classes which follow **IS-A** relationship

✔ The class from which you inherit is the SUPERCLASS the class which inherits is the SUBCLASS.

✔ The superclass defines GENERAL attributes and methods. Subclass defines SPECIFIC attributes and methods.

✔ ‘extends’ is the keyword which connects a subclass to the superclass.

syntax :

```
class subclassName extends superclassName
{
        //body of subclass
}
```

Activity 1.4.1

1. Write a class Animal with attributes, age,gender,weight and methods, eat(),sleep(),details().
2. Write a class Mammal with attributes, litterSize, gestationPeriod.
3. Write a class Dog with attributes, tailLength, isTrained and methods getTrained()
4. Connect Mammal class and Dog class such that we form a hierarchy of classes using Inheritance
5. Write a InheritanceDemo.java class containing main(). Instantiate all the classes and call methods using the respective objects. Observe the flow of program execution and output.

Activity 1.4.1

6. Use following combinations of references and objects for method invocation and observe the flow of the program.

a. Superclass reference = superclass object

    reference.superclassMethod();
    reference.subclassMethod()

b. superclass reference = subclass object

    reference.superclassMethod();
    reference.subclassMethod()

c. Subclass reference = subclass object

    reference.superclassMethod();
    reference.subclassMethod()

d. subclass reference = superclass object

    reference.superclassMethod();
    reference.subclassMethod()

Activity 1.4.2 – Method overridding

1. In the previous class, Animal, add a method, sound(). In the Dog class override the sound() to print the message, “Dogs bark”. Add one more class, Lion. Make it the subclass of Animal. In the sound() of Lion class print the message, “Lion roars”. In the PolymorphismDemo class containing main(). Create objects of the necessary classes and invoke sound() on each object. Observe the output messages and understand the flow of the program.

Refer Animal.java, Dog.java, Lion.java and PolymorphismDemo.java

Activity 1.4.3 – Accesing members of the class

1. In Animal class, add an attribute, diet. Make it *private.*
2. In InheritanceDemo.java where we have created the objects for all the classes, add print statements and access the diet attribute using ‘*object.attribute’* syntax. Observe the flow of the program and the output.

It is not possible to access private members of the class from anywhere OUTSIDE that class. Such members are NOT inherited by any subclasses of the said class.

Process of object creation

✔ Consider the code statement given below:

```
objA = new A();
```

The above statement creates a new object, objA of the class A. This is called INSTANTIATING a class. This single statement goes through 3 steps as given below:

a. **Declaration -**

objA – This part of the above statement declares a reference having a certain name. Object names in Java start with a small letter and follow camel casing in case of multiple words.

b. **Initialization -**

new – This is an operator in Java which allocates memory to the object being created.

A() – This calls the constructor on the object who has been just allocated memory. The constructor then initializes the object.

c. **Assignment**

The ‘=‘ sign in the above statement assigns the address/location of the freshly created and initialized object to the reference on its left. Thus after the execution of the above statement, the reference becomes the pointer to the object of the given class.

Constructor and Constructor Chaining

✔Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

✔A constructor initializes an object immediately upon creation. Once defined, the constructor is automatically called when the object is created, before the new operator completes.

✔It has the same name as the class in which it resides and is syntactically similar to a method.

✔Constructors look a little strange because they have no return type, not even void. This is because the implicit return type of a class's constructor is the class type itself.

Activity 1.5

1. Revisit the classes which you have created and add constructor in Animal, Mammal, Dog and Lion classes. Add some print statements in each constructor.
2. Observe the sequence of statements in the output.
3. From what you observe discuss what is **Constructor Chaining?**

Constructor Types and Overloading

- ✔ **Default constructor** - If no explicit constructor is specified, then Java will automatically supply a default constructor
- ✔ **Non parameterized constructor** – A constructor which is explicitly added to the class but takes no arguments.
- ✔ **Parameterized constructor** – A constructor which is explicitly added to the class and has parameters which take arguments at runtime.

Activity 1.5

Add one more constructor to Animal class such that it accepts some parameter. Create objects of Animal class using non parameterised constructor and parameterised constructor. Observe which version of the constructor is called when.

Discuss how does the compiler choose which constructor to call?

Having more that one forms of constructors in the class is called constructor overloading

super keyword

- ✔super used in any form allows the sub class to access its immediate super class member.
- ✔In a constructor super() or super(arg1,ar2,.argn) will invoke either the non-parameterized or parameterized versions of the immediate superclass constructor respectively. If used in a constructor, it has to be the first code statement.
- ✔In a method, super(arg1,arg2,..argn) will invoke the matching overridden version of the superclass method.
- ✔It can be used to access a variable of the superclass.

Activity 1.6

1. To the constructor of Dog class add super() to invoke the parameterised constructor of its immediate superclass.
2. In the sound() in Lion class, add super to invoke the immediate superclass version of sound()
3. Override details() in Mammal class. Add print statements to print the details of the mammal object. Use super with the superclass variables and show how they are accessed from within a subclass.

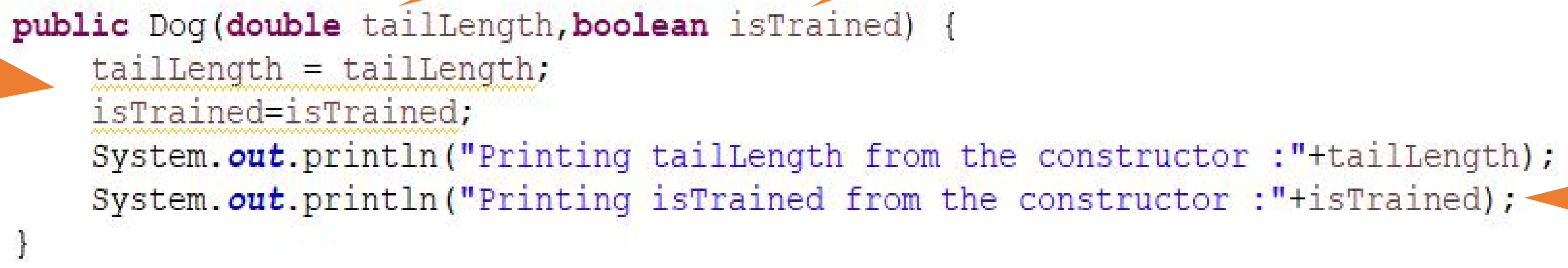
Activity 1.7 - Understanding this keyword wrt variables

1. Add the following parameterised constructor to Dog class.

Local variable

Local variable

Attempt to initialise Instance variables of the same name as local variables

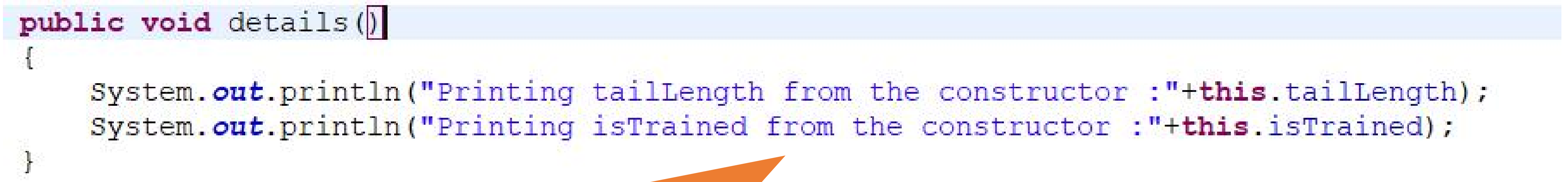


```
public Dog(double tailLength,boolean isTrained) {
    tailLength = tailLength;
    isTrained=isTrained;
    System.out.println("Printing tailLength from the constructor :"+tailLength);
    System.out.println("Printing isTrained from the constructor :"+isTrained);
}
```

Observe the output on the console

Activity 1.7 - Understanding this keyword wrt variables

2. Add the following statements to details() within Dog class



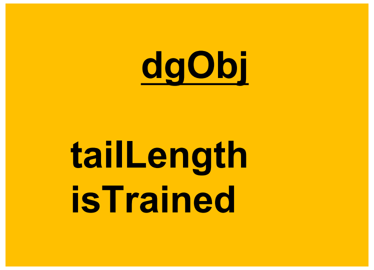
```
public void details()
{
    System.out.println("Printing tailLength from the constructor :"+this.tailLength);
    System.out.println("Printing isTrained from the constructor :"+this.isTrained);
}
```

Printing values of instance variables which should have been initialised in the constructor after changes done in step 1.
Run the code and observe the output on the console

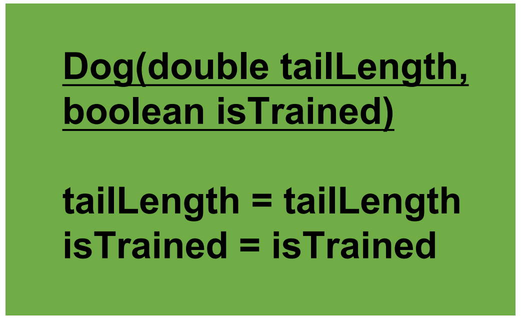
Discuss why the instance variables do NOT have the same values as the local variables despite the initialization in the constructor

this keyword - avoiding instance variable hiding

✔ In the previous example, the instance variables of Dog class are:

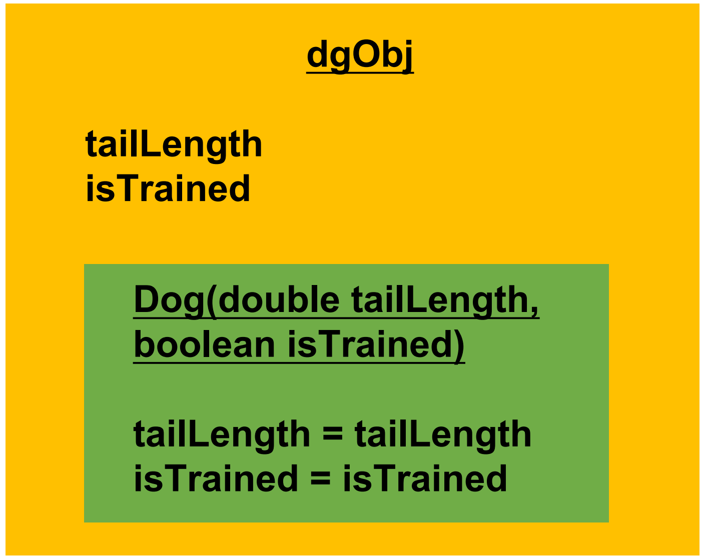


✔ The local variables in the parameterised constructor are:


```
Dog(double tailLength,
boolean isTrained)

tailLength = tailLength
isTrained = isTrained
```

✔The object, dgObj, looks like :



```
Dog(double tailLength,
boolean isTrained)

tailLength = tailLength
isTrained = isTrained
```

✔The names of the instance variables and local variables are SAME. This casues naming conflict.

✔ The local variable **HIDES** the instance variable hence values from the local variables from within the constructor DO NOT get assigned to the instance variables.

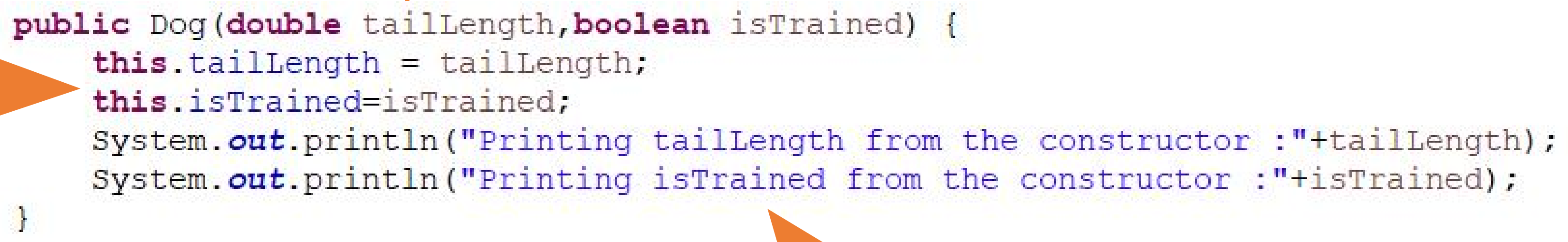
Activity 1.8 - Understanding this keyword wrt variables

Modify the parameterised constructor to Dog class using **this** keyword.

Local variable

Local variable

Attempt to initialise Instance variables of the same name as local variables using this keyword



```
public Dog(double tailLength,boolean isTrained) {
    this.tailLength = tailLength;
    this.isTrained=isTrained;
    System.out.println("Printing tailLength from the constructor :"+tailLength);
    System.out.println("Printing isTrained from the constructor :"+isTrained);
}
```

Observe the output on the console

this keyword - avoiding instance variable hiding

- ✔ *this* can be used inside any method to refer to the current object.
- ✔ In case of naming conflict between local variables and instance variables, *this* keyword is used to refer to the instance variables as they belong to the object.
- ✔ Similarly, *this()* is used to invoke a specific copy of the overloaded method from within another overloaded method. (Refer Dog.java)
- ✔ *this()* when used within an overloaded constructor will help in calling another version of that constructor (Refer ThisExample.java)

Parameter passing

- ✔The way arguments are passed to the method at runtime determines parameter passing in that language.
- ✔Other languages allow two ways in which parameters are passed as arguments to the called method. These two ways are, **call by value and call by reference**.
- ✔Java permits **ONLY** call by value.
- ✔It handles **primitive parameters and object(s)** as parameter(s) using **call by value** approach but they are handled differently.

Activity 1.9.1 Parameter passing primitives

1. Add a class Student having the attributes, rollNbr, name, course, semester, course1, course2, course3, course4, course5, percentage, grade.
2. Using a parameterized constructor, assign the values of rollNbr, name, course, semester, course1, course2, course3, course4, course5 to the newly created object.
3. Write a ArgumentPassingDemo class containing main(). Create object of Student class and initialise the object with the necessary values.

Activity 1.9.2 Parameter passing primitives

1. Create a class Marksheet having attributes totalMarks, percentage, grade. This class should have business methods such as, calculateGrade() which takes the marks in each course as parameters and calculates the total and percentage. If the percentage is less than 50% then the grade is F (Fail), if the percentage is between 51 and 59 then the grade is P (pass) , if the percentage is between 60 and 70 then the grade is G (Good) , if the percentage is between 71 and 75 then the percentage is E (Excellent) and if the percentage is above 75 then the grade is O (Outstanding).
2. Add another method, printMarksheet() which prints the recently calculated percentage and grade of the student.
3. Create object of Marksheet class in the Demo class and invoke the necessary methods from there.
4. Check the status of percentage and grade attributes of Student class object whose marks were used to calculate the total and percentage in points 1 and 2.

Understand how primitives are passed in this example.

Activity 1.9.3 Parameter passing - passing Object

1. In the Marksheet class written before, overload the method, calculateGrade(), to accept Student object. From within this overloaded method, use the Student class object course attributes to access individual course marks. Calculate the total, percentage and grade. Now assign the values of percentage and grade to the Student object's instance variables, percentage and grade respectively.
2. Now overload printMarksheet() to take Student class object as an argument. Access the necessary attributes of this object and print those in the marksheet.

Understand how Objects are passed in this example.

Discuss which approach should be used when – pass primitives v/s pass object as argument?

final keyword

- ✓ final is one of the non-access keywords given by Java.
- ✓ It can be used with variables, methods and class
- ✓ final with variable :
  - It prevents the value of that variable from getting changed. Thus the variable acts like a CONSTANT.
  - Variable declared as final should either be initialized at the time of declaration or within the constructor.
  - It is common coding convention to name the final variable in ALL CAPS.
  - Instance variables, local variables from within the method and local variables passed as arguments all can be declared as FINAL.

Activity 1.10 final variable

Refer **FinalVariableExample.java –** Here we are trying to provide ticket booking system using our class. We take the day of the weak from the user. For each day there is an upper limit fixed for available tickets. Only those many tickets can be sold. We also have a fixed value for TOTAL TICKETS.

The user can book as many tickets as the user wants by invoking the relevant booking method. If the tickets are available, we calculate the booking amount and ask the user to pay that much. Else we display a message that the booking could not be completed.

final keyword

- ✓ final with method :
  - It prevents the method from getting overridden.
  - This prevents the subclass from making any changes to the original, superclass implementation of the method.

Activity 1.10.1 final method

1. Write a class, CharacterUtility.java. This class should have the following methods, countVowels(String str) and countOccurance(char ch,String str).
2. Make both the methods, final.
3. Write another subclass, MyCharacterUtility.java extending CharacterUtility.java. Override countVowels(String str). Observe what happens.

final keyword

- ✓ final with class :
  - It prevents the class from getting inherited.
  - This prevents any other class from becoming a subclass of the original class.
  - Which classes in Java are final?

Activity 1.10.2 final class

1. Write a class, CharacterUtility.java. This class should have the following methods, countVowels(String str) and countOccurance(char ch,String str).
2. Make the class, final.
3. Write another subclass, MyCharacterUtility.java extending CharacterUtility.java. Observe what happens.

final keyword

- ✓ We can conclude that use of final keyword

  with a variable – prevents changes to the variable

  with a method – prevents method overriding

  with a class – prevents class inheriting

static keyword

- ✓ When we use the non-access keyword static for a variable or method, we allow it to be used independently.
- ✓ static keyword can be used for variables and methods.
- ✓ Such data element can be used without creating a class object. So, it does not need the following syntax,

  object.dataMember

- ✓ public static void main(String arg[]) is a popular static method in Java. Understand its significance and syntax.
- ✓ From Java 8, static methods are allowed in the interface too.

Activity 1.11 static variable

Refer **`StaticExample`.java –** Here we are trying to provide ticket booking system using our class. We take the day of the weak from the user. For each day there is an upper limit fixed for available tickets. Only those many tickets can be sold. We also have a fixed value for TOTAL TICKETS.

There are multiple users booking the tickets from different ticket windows.

Each user can book as many tickets as the user wants by invoking the relevant booking method. If the tickets are available, we calculate the booking amount and ask the user to pay that much. Else we display a message that the booking could not be completed.

Thank You