
WEBONISE LAB

Fundamentals of JAVA

What is Java?

- Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE])

- Java's DNA lies in the C++ programming language and traditionally follows the Object Oriented Programming paradigm

- Every line of code is encapsulated into classes.

- Java was developed under the philosophy of Write Once, Run Anywhere.

Features

- Platform Independant (Write once Run Anywhere)
- Supports Multithreading
- Object Oriented.
- Highly Secure
- Combination of Compiler and Interpreter used to excute the programs
- Dynamic Binding
- In built networking

Object and Classes

- Class - Template/blueprint that describes state/behaviour of the object
- Object - Real entities/instance of class.
- Provide a shell/boundary for your code
  eg: barackObama is an object of the class POTUS.
- Objects are accessed only by reference.

  eg:

  barackObama= new POTUS();

  barackObama.getAddress()
  ->White House,Washington D.C

Object and Classes

- Local Variables - defined inside a method, constructor or block.
- Instance Variable - Inside class but outside methods.
- Class Variable - static variables.
- Constructors:-
  - Used to create an object of Class.
  - Name is same as the class name.
  - Doesn’t have any return type.
  - If you don’t create a constructor, Java compiler creates a default for you.
  - You can have more than one constructor.

JAVA primitives

int 4 bytes

short 2 bytes

long 8 bytes

byte 1 byte

float 4 bytes

double 8 bytes

char Unicode encoding (2 bytes)

boolean {true,false}

Access specifiers

- **public member (function/data)**
  Can be called/modified from outside.

- **protected**
  Can be called/modified from derived classes

- **private**
  Can be called/modified only from the current class

- **default (if no access modifier stated)**
  Usually referred to as “Friendly”.
  Can be called/modified/instantiated from the same package.

Clean up of objects in java

- Called garbage collection
  - Memory Management technique.
  - Process of freeing objects.
  - No longer referenced by the program.

- Purpose
  - Find and delete unreachable objects.
  - Free space as much as possible.

- When??
  - GC is under control of JVM.
  - One can request but no guarantees.

- How??
  - Discovery of unreachable objects.
  - With the help of Algorithms.

Garbage Collection in Java

- Ways for making objects eligible for collection
  - Nulling a reference
  - Reassigning a reference variable
  - Isolating a reference

- Forcing garbage collection
  - Methods available to perform GC
  - Only requests and no demands
  - Using Runtime class

  Using static methods like System.gc()

Static keyword

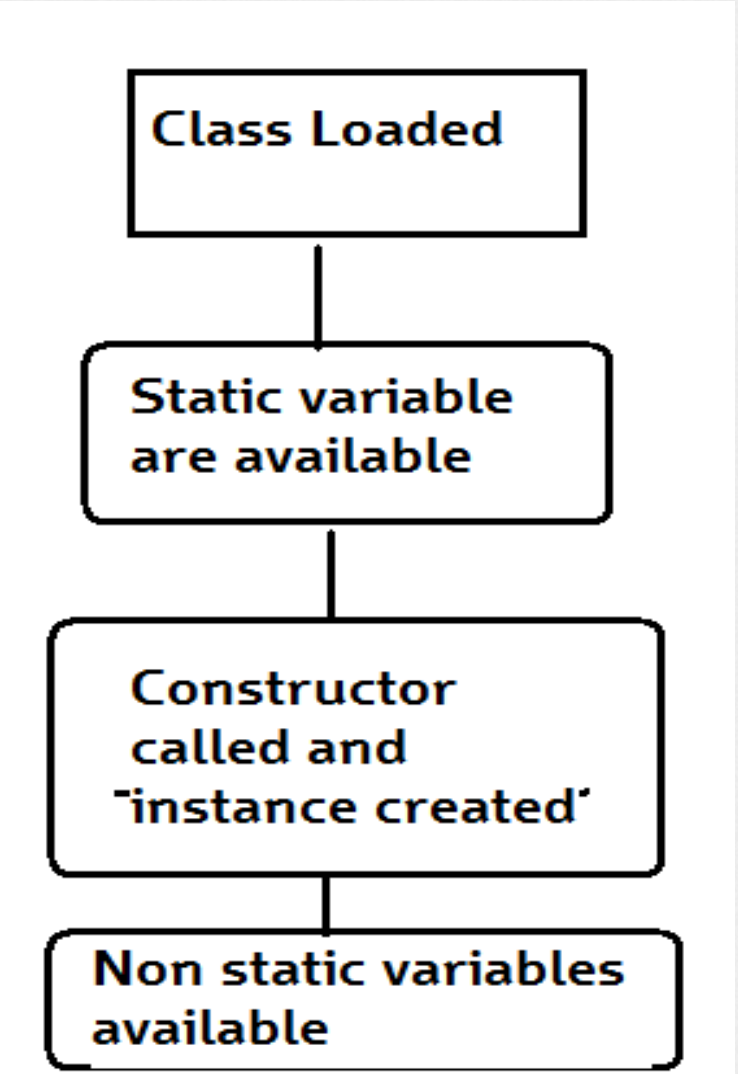
- The static keyword is used when a member variable of a class has to be shared between all the instances of the class.

- All static variables and methods belong to the class and not to any instance of the class

When can we access static variable

- When a class is loaded by the virtual machine all the static variables and methods are available for use.
- Hence we don’t need to create any instance of the class for using the static variables or methods.
- Variables which don’t have static keyword in the definition are implicitly non static.

How it works


Class Loaded
Static variable are available
Constructor called and "instance created"
Non static variables available

Why do we need this

- Static methods are identified to be mostly used when we are writing any utility methods.
- We can also use static variables when sharing data.
- When sharing data do keep in mind about multithreading can cause inconsistency in the value. (synchronize the variable)

Can class be a static?

YES.

- Java allows us to define a class within another class. Such a class is called a nested class.

- The class which enclosed nested class is known as Outer class. In java, we can’t make Top level class static.

“ Only nested classes can be static.(inner classes) “

Interfaces

Interfaces are useful for the following:

- Capturing similarities among unrelated classes without artificially forcing a class relationship.

- Declaring methods that one or more classes are expected to implement.

- Revealing an object's programming interface without revealing its class.

When to use an interface?

Java's compensation for removing the multiple inheritance. You can "inherit" as many interfaces as you want.

Perfect tool for encapsulating the classes inner structure. Only the interface will be exposed

Final keyword

- If a class declared with the final modifier, then it cannot be extended or subclassed

- If a field is declared with final, then the value of it cannot be changed.

- If a method is declared with final, then it cannot be overridden in subclass

Anonymous classes

- If an object is to be created, but there is no need to name the object's class, then an anonymous class definition can be used The class definition is embedded inside the expression with the new operator
- An anonymous class is an abbreviated notation for creating a simple local object "in-line" within any expression, simply by wrapping the desired code in a "new" expression.
- Anonymous classes are sometimes used when they are to be assigned to a variable of another type

Anonymous classes

- If an object is to be created, but there is no need to name the object's class, then an anonymous class definition can be used
  - The class definition is embedded inside the expression with the new operator
  - An anonymous class is an abbreviated notation for creating a simple local object "in-line" within any expression, simply by wrapping the desired code in a "new" expression.

Anonymous classes

- Anonymous classes are sometimes used when they are to be assigned to a variable of another type
  - The other type must be such that an object of the anonymous class is also an object of the other type
  - The other type is usually a Java interface
  - Not every inner class should be anonymous, but very simple "one-shot" local objects are such a common case that they merit some syntactic sugar.

Anonymous classes

Display 13.11 Anonymous Classes *(Part 1 of 2)*

This is just a toy example to demonstrate the Java syntax for anonymous classes.

```
1  public class AnonymousClassDemo
2  {
3      public static void main(String[] args)
4      {
5          NumberCarrier anObject =
6                  new NumberCarrier()
7                  {
8                      private int number;
9                      public void setNumber(int value)
10                     {
11                         number = value;
12                     }
13                     public int getNumber()
14                     {
15                        return number;
16                     }
17                 };
```

Anonymous classes

Display 13.11 Anonymous Classes *(Part 1 of 2)*

```
18              NumberCarrier anotherObject =
19                      new NumberCarrier()
20                      {
21                          private int number;
22                          public void setNumber(int value)
23                          {
24                              number = 2*value;
25                          }
26                          public int getNumber()
27                          {
28                              return number;
29                          }
30                      };

31              anObject.setNumber(42);
32              anotherObject.setNumber(42);
33              showNumber(anObject);
34              showNumber(anotherObject);
35              System.out.println("End of program.");
36          }

37          public static void showNumber(NumberCarrier o)
38          {
39              System.out.println(o.getNumber());
40          }

41      }
```

This is still the file AnonymousClassDemo.java.

Anonymous classes

Display 13.11 Anonymous Classes *(Part 2 of 2)*

SAMPLE DIALOGUE

```
42
84
End of program.
```

This is the file NumberCarrier.java.

```
1  public interface NumberCarrier
2  {
3      public void setNumber(int value);
4      public int getNumber();
5  }
```

OOPs Concepts

- **Inheritance**
- **Polymorphism**
- **Encapsulation**
- **Abstraction**

Inheritance

- Inheritance is a code re-use solution.
- Inheritance supports polymorphism at the language level
- In Java we can inherit from only 1 class.
- Inheriting from multiple classes is not possible due to diamond ambiguity problem.
- Few pointers while considering inheritance
  - Cant override final instances
  - For an abstract inheritance ,you must supply an implementation in the sub class
  - Final classes cant be overriden
  - Interfaces can have multiple inheritances ,by which you can achieve multiple inheritance in java

Inheritance

- Rules of Inheritance:-
  - All fields and methods of the superclass are inherited by the subclass.
  - Inheritance creates an **is-a** relationship - an instance of the subclass is also considered an instance of the superclass.
  - A subclass is typically a more specialized version of its superclass.
  - A subclass can NOT refer to private methods and fields of its superclass.
  - Superclasses are ordinary Java classes - superclasses can have instances and superclass methods have bodies.

Inheritance

- Inheritance and overriding:-
  - An overriding method in the subclass must have exactly the same signature (name, number and order of parameters, return type) as the superclass method it is overriding.
  - The overriding method definition will be used for subclass objects.
  - The keyword super can be used to call overridden superclass methods.

Inheritance

```
class Base {
    Base(){}
    Base(int i) {}
    protected void foo() {…}
}
class Derived extends Base {
    Derived() {}
    protected void foo() {…}
    Derived(int i) {
       super(i);
    …
    super.foo();
    }
}
```

Composition

- Composition
  - A class can have references to objects of other classes as members
  - Sometimes referred to as a has-a relationship
- One form of software reuse is composition, in which a class has as members references to objects of other classes.

Polymorphism

- Polymorphism allows you define one interface and have multiple implementations
- Polymorphism is essentially considered into two versions.

  - **Compile time polymorphism (static binding or method overloading)**

  In this an object can have two or more methods with same name, BUT, with their method parameters different. These parameters may be different on two bases:

  Parameter type: Type of method parameters can be different

  Parameter count: Functions accepting different number of parameters

  - **Runtime polymorphism (dynamic binding or method overriding)**

  Method overriding is a feature which you get when you implement inheritance in your program.

Encapsulation

- Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods.

- If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class.

- Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class.

Abstraction

- A class must be declared ***abstract*** when it has one or more abstract *methods*. A method is declared abstract when it has a method heading, but no body

- An abstract class is one that cannot be instantiated.

- All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner.

- You just cannot create an instance of the abstract class.

Interfaces vs abstract class

Feature	Interface	Abstract class
Multiple inheritance	A class may inherit several interfaces.	A class may inherit only one abstract class.
Default implementation	An interface cannot provide any code, just the signature.	An abstract class can provide complete, default code and/or just the details that have to be overridden.
Access Modfiers	An interface cannot have access modifiers for the subs, functions, properties etc everything is assumed as public	An abstract class can contain access modifiers for the subs, functions, properties
Core VS Peripheral	Interfaces are used to define the peripheral abilities of a class. In other words both Human and Vehicle can inherit from a IMovable interface.	An abstract class defines the core identity of a class and there it is used for objects of the same type.

Interfaces vs abstract class

Homogeneity	If various implementations only share method signatures then it is better to use Interfaces.	If various implementations are of the same kind and use common behaviour or status then abstract class is better to use.
Speed	Requires more time to find the actual method in the corresponding classes.	Fast
Adding functionality (Versioning)	If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method.	If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly.
Fields and Constants	No fields can be defined in interfaces	An abstract class can have fields and constrants defined

Upcasting and DownCasting

Upcasting:

- An object of a sub class can be referred by its super class automatically.
- You can cast an object implicitly to a super class type is Called UpCasting.

Downcasting

- Explicit casting is to be done by the super class.
- Which is performed manually by the developers.

Upcasting and DownCasting

```
class Animal {
int health = 100;
}

class Mammal extends Animal { }

class Cat extends Mammal { }

class Dog extends Mammal { }

public class Test {
public static void main(String[] args) {
Cat c = new Cat();
System.out.println(c.health);
Dog d = new Dog();
System.out.println(d.health);
}
}
```

```
Cat c = new Cat();
System.out.println(c);
Mammal m = c; // upcasting
System.out.println(m);
```

```
Cat c1 = new Cat();
Animal a = c1; //automatic
upcasting to Animal
Cat c2 = (Cat) a; //manual
downcasting back to a Cat
```

Summary

- Java is a language that follows OOPS to the fullest.
- Every implementation and every well written piece of code will have the hallmarks like inheritance,polymorphism,encapsulation and data hiding
- This was just an introduction to the language which you will discover as you go on and hopefully fall in love with.

Assignment

Write a program to calculate the sum of interior angles for a polygon with side from 3 to 10.Also calculate the perimeter of each.Use concepts of inheritance and polymorphism

Thank You