Bootcamp Training

Java Spring AWS Frontend Training

SummitWorks™
GLOBAL SOLUTION ARCHITECTS

Authorized & published by Summitworks Technologies Inc

Agenda: Day -2: Object Oriented Programming: Object, Class, Abstraction, Encapsulation, Inheritance & Polymorphism

- Introduction to Object Oriented Programming
- Objects and Classes in Java
- Scope, Instance and Static variables and methods in Java
  - Local Variables
  - Instance and static Variables
  - Instance methods and Class/static methods
- Create and import Java Packages
- Object class Methods
- Constructors
  - Default and Parameterized constructor
  - The this Reference
- Inheritance
  - extends Keyword
  - The super and this keyword
  - IS-A and HAS-A Relationship
  - The instance of Keyword
  - Types of Inheritance and Multiple Inheritance
- Interfaces
  - Declaring, Extending and Implementing
  - Extending Multiple Interfaces
  - default and static methods in Interfaces
  - Functional Interfaces and Lambda Expressions
- Abstraction
  - Abstract Class and Abstract Methods
  - Inheriting the Abstract Class
- Modifiers in Java
  - Access Modifiers
    - Default and Public
    - Private and protected
  - Non-Access Modifiers
    - The static , final and abstract Modifiers
    - synchronized, transient and volatile Modifiers
- Encapsulation
  - Benefits of Encapsulation
- Polymorphism
  - Static and Dynamic Polymorphism
- Rules for Method Overriding

OOPs (Object Oriented Programming system)

Object means a real word entity such as pen, chair, table etc. Object**-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts.

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

- Abstraction is hiding the functionality details.
- Abstraction is achieved by creating either Abstract Classes or Interfaces on top of your class.
- hide the unnecessary things from user so providing easiness.
- Hiding the internal implementation of software so providing security.

Encapsulation

Encapsulation is hiding information.

- A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.
- Encapsulation is performed by constructing the class.
- Encapsulation links the data with the code that manipulates it.
  Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods.

Difference between Abstraction and Encapsulation

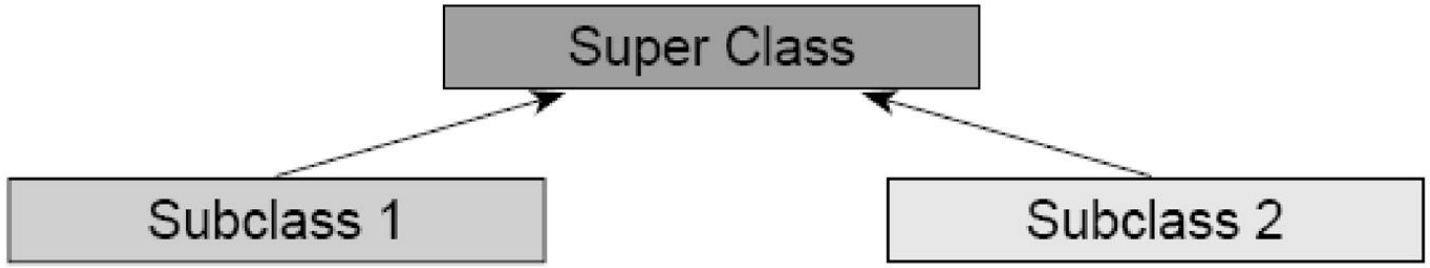
Encapsulation is hiding WHAT THE PHONE USES to achieve whatever it does;
Abstraction is hiding HOW IT DOES it.

Encapsulation = Data Hiding + Abstraction.

Inheritance[IS-A]

- Is the ability to derive something specific from something generic.
- *aids in the reuse of code.*
- A class can inherit the features of another class and add its own modification.
- The parent class is the *super class* and the child class is known as the *subclass*.
- A subclass inherits all the properties and methods of the super class.

Inheritance

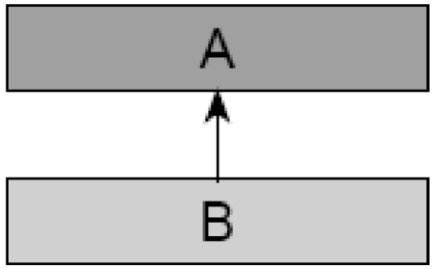


Types of Inheritance

- Single
- Multiple
- Multilevel
- Hierarchical
- Hybrid

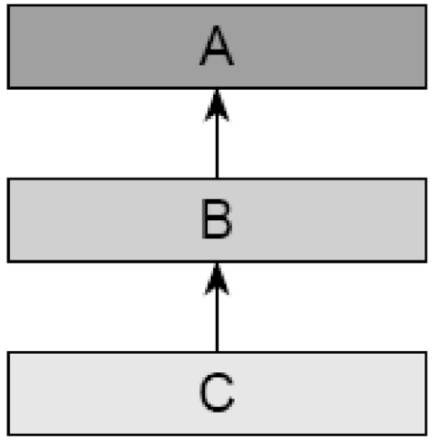
Single level inheritance

Classes have only base class



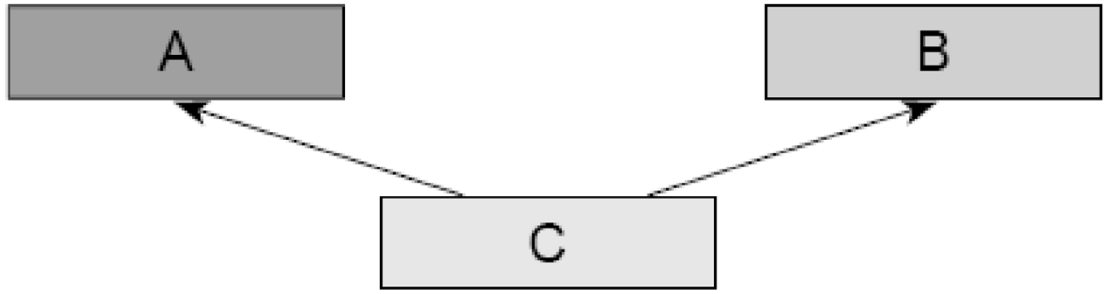
Multi level

There is no limit to this chain of inheritance (as shown below) but getting down deeper to four or five levels makes code excessively complex.



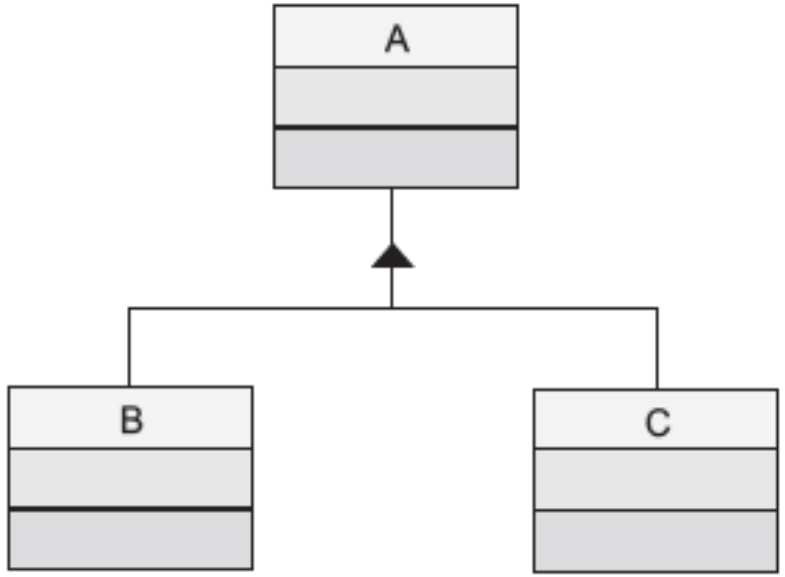
Multiple Inheritance

A class can inherit from more than one unrelated class



Hierarchical Inheritance

- In hierarchical inheritance, more than one class can inherit from a single class. Class C inherits from both A and B.



Object class Methods

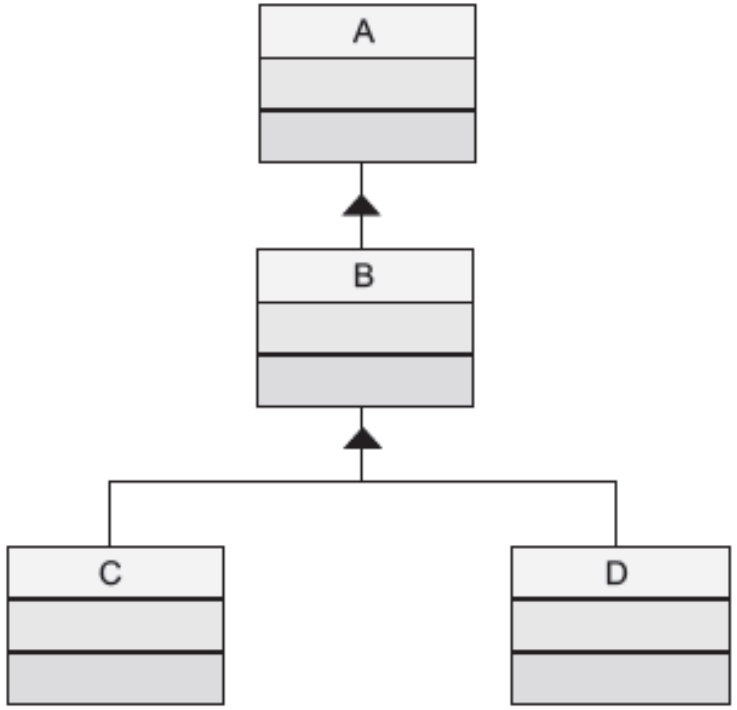
The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

- The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.
- Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

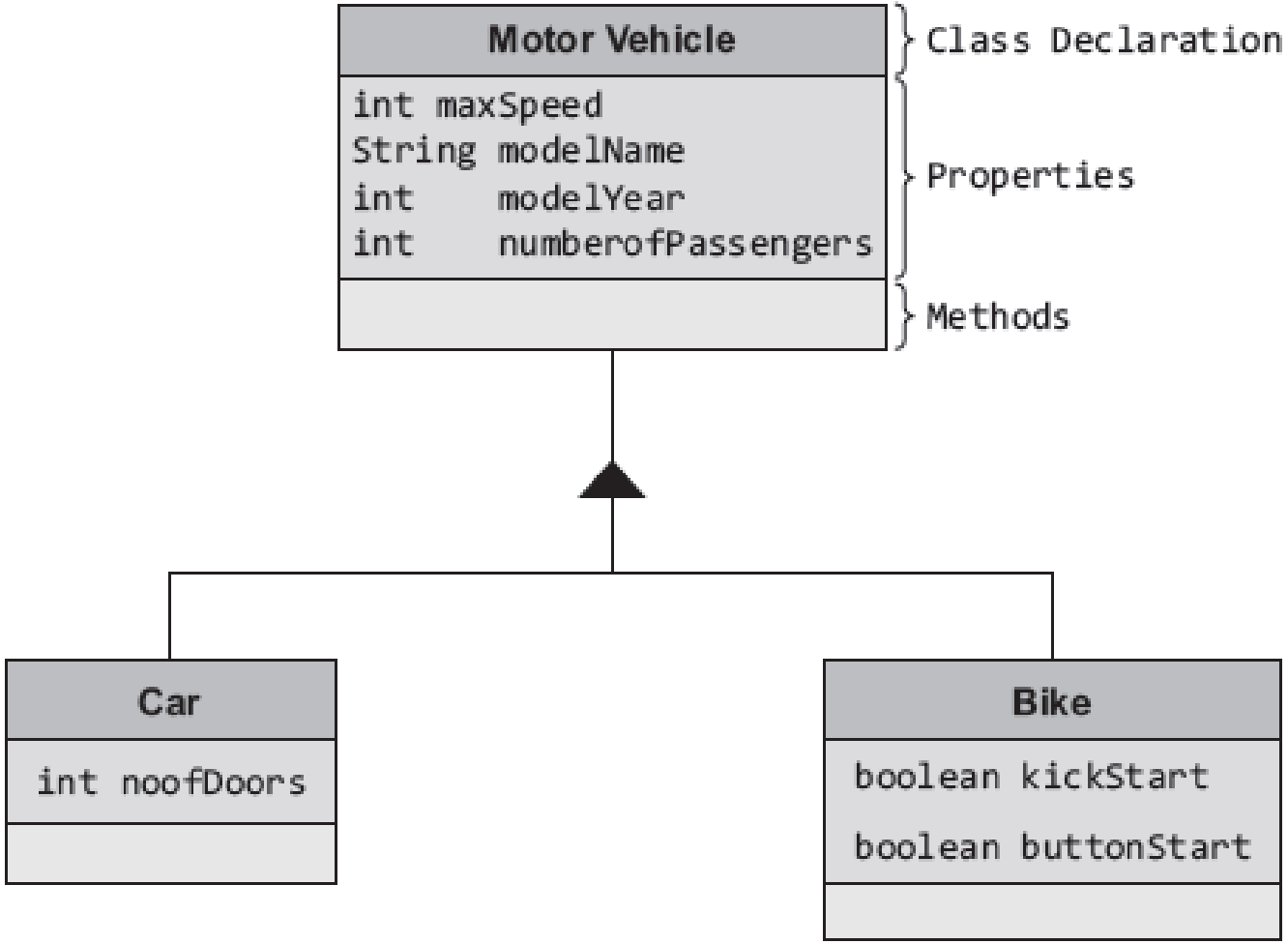
Object obj=getObject();//we don't know what object will be returned fro m this method

Hybrid Inheritance

- is any combination of the above defined inheritances



Inheritance



Aggregation in Java [HAS-A]

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

- Consider a situation, Employee object contains many information's such as id, name, email ID etc. It contains one more object named address, which contains its own information's such as city, state, country, zip code etc. as given below.

```
class Employee{
int id;
String name;
Address address;//Address is a class
...
}
```

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

When use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Super and this keywords

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object. Here is given the 6 usage of java this keyword.

- this can be used to refer current class instance variable.
- this can be used to invoke current class method (implicitly)
- this() can be used to invoke current class constructor.
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this can be used to return the current class instance from the method.

super keyword in java

The **super** keyword in java is a reference variable which is used to refer immediate parent class object.

- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of java super Keyword

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

Polymorphism

- **Polymorphism in java** is a concept by which we can perform a *single action by different ways*. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

Method Overloading in Java

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.

Advantage of method overloading

- Method overloading *increases the readability of the program*.

Different ways to overload the method

- By changing number of arguments
- By changing the data type

Method Overriding

- a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.
- it is a feature that supports polymorphism.
- When an overridden method is called through the subclass object, it will always refer to the version of the method defined by the subclass.
- The superclass version of the method is hidden.

Usage of Java Method Overriding

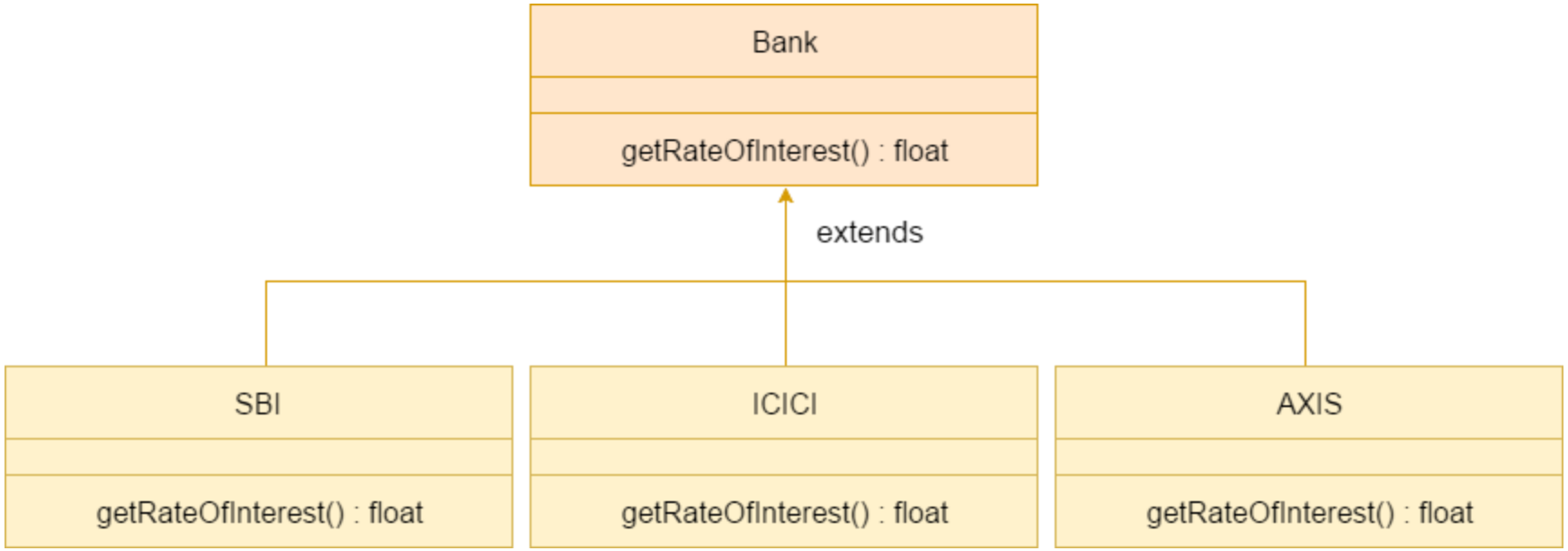
- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- method must have same name as in the parent class
- method must have same parameter as in the parent class.
- must be IS-A relationship (inheritance).

Real example of Java Method Overriding:

Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.



Difference between method overloading and method overriding in java

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

final keyword

- Declaring constants (used with variable and argument declaration)
  - final int MAX=100;
- Disallowing method overriding in chaild class (used with method declaration)
  - final void show (final int x)
- Disallowing inheritance (used with class declaration).
  - final class Demo {}

Optional Modifiers Used with Methods

Optional Modifiers	
Modifier	**Description**
`public, protected, default or private`	Can be one of these values. Defines the scope—what class can invoke which method. (see Chapter 6, Section 6.2.4)
`static`	The method can be invoked on the class without creating an instance of the class (see Chapter 4, section 4.7)
`abstract`	The class must be extended and abstract method must be overridden in the subclass (see Chapter 5, Section 5.5)
`final`	The method cannot be overridden in a subclass (see Chapter 5, Section 5.4)
`native`	The method is implemented in another language (out of scope of this book)
`synchronized`	The method requires that a monitor (lock) be obtained by calling code before the method is executed (see Chapter 8, Section 8.8)
`throws`	A list of exceptions thrown from this method (see Chapter 7, Section 7.2.3)

Access Modifiers in java

- There are two types of modifiers in java: **access modifiers** and **non-access modifiers**.
- The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.
- There are 4 types of java access modifiers:

➢ private
➢ default
➢ protected
➢ public

There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will learn access modifiers.

Access Modifiers in java

Visibility	Public	Protected	*Default*	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, *through inheritance*	No	No
From any non-subclass class outside the package	Yes	No	No	No

Command Line Arguments

- Suppose, you have a Java application, called sort, that sorts the lines in a file named Sort.text. You would invoke the Sort application as,
  - java Sort Example.txt
- When the application is invoked, the runtime system passes the command line arguments to the applications main( ) method via an array of string.
- In the statement above, command line arguments (Example.txt) is passed to the Sort application an array of String that contains a single string, i.e. Example.txt.
- This String array is passed to the main method and it is copied in *args*.

```
public static void main (String args[]) {
. . . . . .
} // end of the main( ) method
```

Wrapper classes

- Java is an object-oriented language and can view everything as an object. A simple file can be treated as an object (with java.io.File), an address of a system can be seen as an object (with java.util.URL), an image can be treated as an object (with java.awt.Image) and a simple data type can be converted into an object (with wrapper classes)
- The primitive data types are not objects; they do not belong to any class; they are defined in the language itself. Sometimes, it is required to convert data types into objects in Java language.

Wrapper classes

- Primitive Type Wrapper class
- boolean Boolean
- charCharacter
- byte Byte
- short Short
- int Integer
- longLong
- float Float
- double Double

Importance of Wrapper classes

- To convert simple data types into objects, that is, to give object form to a data type; here constructors are used.
- To convert strings into data types (known as parsing operations), here methods of type parseXXX() are used.

Class Modifiers

A class declaration may include *class modifiers*.

- public
- protected
- private
- abstract
- static
- final

Field Modifiers

Field Modifiers:

- public
- protected
- private
- static
- final
- transient
- volatile

Public

- public means that **any class can access the data/methods**
- private means that **only the class can access the data/methods**
- protected means that only **the class and its subclasses can** access the data/methods

Access Modifiers

Access Modifier	Class or member can be referenced by…
public	methods of the same class, and methods of other classes
private	methods of the same class only
protected	methods of the same class, methods of subclasses, and methods of classes in the same package
No access modifier (package access)	methods in the same package only

public vs. *private*

- Classes are usually declared to be *public*
- Instance variables are usually declared to be *private*
- Methods that will be called by the client of the class are usually declared to be *public*
- Methods that will be called only by other methods of the class are usually declared to be *private*

Public

- Public access
- Most liberal kind of access
- Class or field is accessible everywhere
- When a method or variable is labelled with the keyword public it means that **any other class or object can use that *public method or variable***

  When a class is labelled with the keyword public, it means the class can be used by **any other class**
- The keyword private is used to restrict access and prevent inheritance!

Private

- Private if its only visible from inside the class definition
- This is compromised somewhat by public access methods

Rules for *static* and Non-*static* Methods

	static Method	Non-*static* Method
Access instance variables?	no	yes
Access *static* class variables?	yes	yes
Call *static* class methods?	yes	yes
Call non-*static* instance methods?	no	yes
Use the object reference *this*?	no	yes

Abstract class

- Abstract classes are classes with a generic concept, not related to a specific class.
- Abstract classes define partial behaviour and leave the rest for the subclasses to provide.
- contain one or more abstract methods.
- abstract method contains no implementation, i.e. no body.
- Abstract classes cannot be instantiated, but they can have reference variable.
- If the subclasses does not override the abstract methods of the abstract class, then it is mandatory for the subclasses to tag itself as *abstract.*

Why create abstract methods?

- to force *same name and signature pattern* in all the subclasses
- subclasses should not use their own naming patterns
- They should have the flexibility to code these methods with their own specific requirements.

Interfaces

- Interfaces is a collection of methods which are public and abstract by default.
- The implementing objects have to inherit the interface and provide implementation for all these methods.
- *multiple inheritance* in Java is allowed through interfaces
- Interfaces are declared with help of a keyword *interface*

Syntax for Creating Interface

```
interface interfacename
{
returntype methodname(argumentlist);
…
}
```

The class can inherit interfaces using implements keyword

- class classname implements interfacename{}

Variables in Interface

- They are implicitly *public*, *final*, and *static*
- As they are *final*, they need to be assigned a value compulsorily.
- Being *static*, they can be accessed directly with the help of an interface name
- as they are *public* we can access them from anywhere.

Extending Interfaces

- One interface can inherit another interface using the *extends* keyword and not the *implements* keyword.
- For example,

  interface A extends B { }

Interface Vs Abstract class

Interface	Abstract Class
Multiple inheritance possible; a class can inherit any number of interfaces.	Multiple inheritance not possible; a class can inherit only one class.
implements keyword is used to inherit an interface.	**extends** keyword is used to inherit a class.
By default, all methods in an interface are **public** and **abstract**; no need to tag it as **public** and **abstract**.	Methods have to be tagged as **public** or **abstract** or both, if required.
Interfaces have no implementation at all.	Abstract classes can have partial implementation.
All methods of an interface need to be overridden.	Only abstract methods need to be overridden.
All variables declared in an interface are by default **public**, **static**, and **final**.	Variables, if required, have to be declared as **public**, **static**, and **final**.
Interfaces do not have any constructors.	Abstract classes can have constructors
Methods in an interface cannot be static.	Non-abstract methods can be static.

Similarities Between Interface and abstract class

- Both cannot be instantiated, i.e. objects cannot be created for both of them.
- Both can have reference variables referring to their implementing classes objects.
- Interfaces can be extended, i.e. one interface can inherit another interface, similar to that of abstract classes (using extends keyword).
- **static** / **final** methods can neither be created in an interface nor can they be used with abstract methods.

When we use interfaces

1. Recommended to start making the application with interfaces.[to get some idea about methods to implement in implemented class]
2. If the properties of the object is different for different object
3. Can be interfaces between the services.

Methods of Object class

public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout) throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos) throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait() throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize() throws Throwable	is invoked by the garbage collector before object is being garbage collected.

The *toString* Method

- Returns a *String* representing the data of an object
- Client can call *toString* explicitly by coding the method call.

The *equals* Method

- Determines if the data in another object is equal to the data in this object

Return value`	Method name and argument list
`boolean`	`equals( Object obj )` returns *true* if the data in the *Object obj* is the same as in this object; *false* otherwise.

Any queries