# So why is JavaScript so popular?

**Scaleable, Asynchronous**

JavaScript is a client-side as well as server side scripting language that can be inserted into HTML pages and is understood by web browsers. JavaScript is also an Object based Programming language

Abstract class cannot be made final.

# [Is it possible to have an abstract method in a final class?](https://javarevisited.blogspot.com/2017/07/is-it-possible-to-have-abstract-method-in-final-class-java.html)

as soon as you declare an abstract method in a Java class, the class automatically becomes an abstract class and you [cannot make an abstract class final](http://www.java67.com/2017/07/can-you-make-abstract-class-method-final-in-java.html) in Java as discussed before, hence it's not possible to have an abstract method in a final class in Java.

**can an abstract class have static methods in Java?** The answer is yes, there is no problem with declaring a [static method](http://javarevisited.blogspot.sg/2013/11/difference-between-static-vs-non-static-method-java.html) inside an abstract class in Java because you don't need to instantiate a class to use the static method, you can just call them by using the class name.

List<String> list = Arrays.asList("One", "Two", "Three");

String result = String.join(", ", list);

System.out.println(result);

Iterator it = map.entrySet().iterator();

while (it.hasNext())

{

Entry item = it.next();

map.remove(item.getKey());

}

This will throw a ConcurrentModificationException when the it.hasNext() is called the second time.  This will occur if the underlying collection that is being iterated over is modified by anything other than the Iterator itself.

The correct approach would be

Iterator it = map.entrySet().iterator();

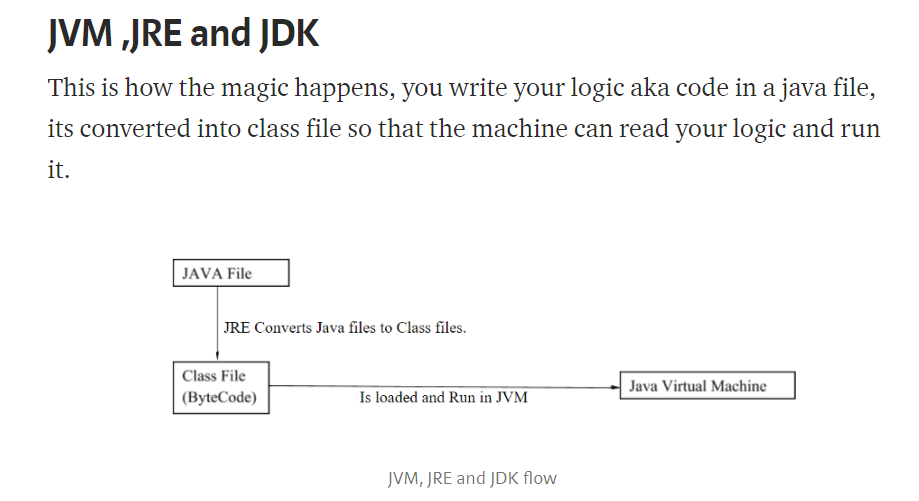
while (it.hasNext())

{

Entry item = it.next();

it.remove();

}



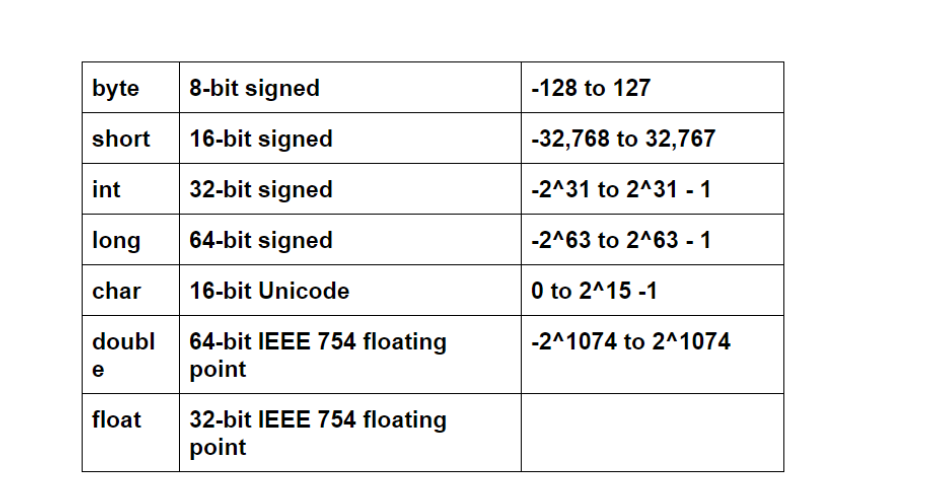
Briefly these points covers it all:

* JVM is the java virtual machine that runs the java byte code.
* JVM can be loaded on various hardware platforms, byte codes are the machine language of JVM. So Java is a better portable language. JVM is the entity that makes Java portable; there are different implementations of JVM for different OS (mac, windows, linux) etc.
* JRE is java runtime environment that is sufficient to run the program.
* JRE = JVM + library files/java package classes (Util, Lang, Math etc).
* JDK is java development kit, required to write, compile and run a program.
* JDK = JRE + Tools needed to develop java program.

# Memory Allocation

* Each time object is created in Java it is stored in heap memory.
* Primitive variables and local are stored in stack, member variables in heap.
* In multithreading each thread will have its own stack but will share same heap. We will discuss multithreading later in part 2.
* Methods and variables are pushed to the stack when a method is invoked and stack pointer is decremented when call is completed.

**Datatype:**



**import** java.util.Scanner;

**public class** sumOfTwoNumbers {  
 **public static void** main(String[] args){

)

Scanner scanner = **new** Scanner(System.***in***); *// Setting to receive user input*

System.***out***.println(**"Enter first integer"**);  
**int** firstNum = **scanner.nextInt**(); // gets user input as int

Scanner scanner = **new** Scanner(System.***in***); *// Setting to receive user input*String input = **scanner.nextLine()**; *// getting user input as string*

**int** normalNum = 2;   
 **double** weirdNum = 3.2;  
  
 System.***out***.println(**"The normal number is "** + normalNum); *//what will print at println #1?* System.***out***.println(**"The weird number is "** + weirdNum); *//what will print at println #2?* **double** sum = normalNum + weirdNum;  
 System.***out***.println(**"The sum is "** + sum); *//what will print at println #3?* **double** product = normalNum \* normalNum;  
 System.***out***.println(**"The product is "** + product); *//what will print at println #4?* sum = 5;  
 System.***out***.println(**"The sum is "** + sum); *//what will print at println #5?* **int** remainder = 5 % normalNum;  
 System.***out***.println(**"The remainder is "** + remainder); *//what will print at println #6?*}

**int**[] temp = **new int**[50];  
**int** i;  
**for**(i=0; i<temp.**length**; i++){  
 Random random = **new** Random();  
 temp[i] = random.nextInt(100);  
 System.***out***.println(**"temp["** + i + **"]: "** + temp[i]);

Enhanced for loop:

1. char[] vowels = {'a', 'e', 'i', 'o', 'u'};
2. // foreach loop
3. for (char item: vowels) {
4. System.out.println(item);
5. for(data\_type item : collection) {
6. ...
7. }

In the above syntax,

* collection is a collection or array variable which you have to loop through.
* item is a single item from the collection.

**Important points regarding arrays**

* 1. You can use any loop e.g. for, while, and do-while or enhanced for loop to loop over an array.  
       
     2) If you need a counter to hold the current index of the array to implement your algorithm than use the for loop.  
       
     3) If your looping depends upon current element then use while and do-while loop.  
       
     4) If you just want to iterate or traverse over all elements of an array then use enhanced for loop. It doesn't need any counter and much more readable than traditional loops.  
       
     5) Use traditional for loop if you want full control of looping over an array. It allows both forward and backward traversal and jumping over elements by incrementing the counter.  
       
     6) You need to use the [nested loops](http://javarevisited.blogspot.com/2012/05/break-continue-and-lablel-in-loop-java.html) i.e.  a loop inside another loop to iterate over a multi-dimensional array in Java.  
       
     Read more: <https://javarevisited.blogspot.com/2016/02/how-to-loop-through-array-in-java-with.html#ixzz5rpFOKrJE>

**Enhanced for loop:**

On the other hand enhanced for loop provides the simplest way to loop through an array. It was originally intended to [loop over collection](http://javarevisited.blogspot.com/2016/01/3-ways-to-loop-over-set-or-hashset-in-java.html) e.g. List, Set or Queue, but you can also use an array with enhanced for loop. It doesn't require the programmer to maintain the counter, instead, it automatically moves from one element to another.  
  
It's best suited for iteration where you want to access each element of array one by one, but it's not as powerful as looping over traditional for loop because you can not jump elements, you cannot traverse backward etc.  
  
  
Read more: <https://javarevisited.blogspot.com/2016/02/how-to-loop-through-array-in-java-with.html#ixzz5rpFb37Wc>

System.***out***.println(**"#1 is "** + (4 > 5 || isACodingWhiz)); *//What will print at println #1?*System.***out***.println(**"#1 is "** + (4 < 5 || isACodingWhiz)); *//What will print at println #2?*System.***out***.println(**"#1 is "** + (4 > 5 && isACodingWhiz)); *//What will print at println #3?*System.***out***.println(**"#1 is "** + (4 < 5 && isACodingWhiz)); *//What will print at println #4?*System.***out***.println(**"#4 is "** + (!isACodingWhiz)); *//What will print at println #5?*

The **primitive** data types include byte, int, long, short, float, double, and char. They are part of the core of Java and you don't need anything special to use them.

**Non-primitive**, or reference data types, are the more sophisticated members of the data type family. They don't store the value, but store a reference to that value. Instead of partNumber 4030023, Java keeps the reference, also called address, to that value, not the value itself.

Reference types can be a class, interface, or array variable. Remember that a **class** is a set of plans for a given object. There are thousands of tree objects, but the parent set of plans would belong in the tree class. Variables can exist inside the tree class, such as height or tree type. These are reference variables.

Arrays:

**String[] classmatesName = {"Amber","Pam", "Ava","Shelly", "Nancy", "Adelita", "Sabrina"};**

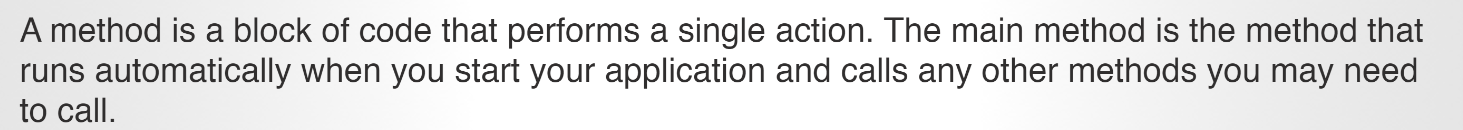
**String[] names = new String[7];**names[0] = **"Amber"**;  
names[1] = **"Pam"**;  
names[2] = **"Ava"**;  
names[3] = **"Shelly"**;  
names[4] = **"Nancy"**;  
names[5] = **"Adelita"**;  
names[6] = **"Sabrina"**;  
  
System.***out***.println(**"Second method: Second name in the array is: "**+ classmatesName[1]);

**for**(**int** i=0; i<3; i++)  
 System.***out***.println(**"For loop: "**+ i);  
*///////////////////////////////////***int** j=0;  
  
**while**(j<1){  
 System.***out***.println(**"While loop: "**+j);  
 j++;  
}  
  
**int** k = 0;  
  
**do** {  
  
 System.***out***.println(**"do-while loop: "**+ k);  
 k++;  
  
}**while** (k<2);

**do** {  
  
 System.***out***.println(**"Enter a name"**);  
 name = scanner.nextLine();  
  
}**while** (!name.equals(**"Louise"**));

Class is a container of variables and methods.

**Method:**



**public class** printNamesMethod {  
 **public static void** main(String[] args){  
  
*printNames*();  
*printNames*();  
  
 }  
  
 *// Defining a method* **public static void** printNames(){  
  
 System.***out***.println(**"My name is Pam"**);  
 System.***out***.println((**"My name is Oam"**));  
 System.***out***.println((**"My name is Cam"**));  
 }  
}

# Overloading in Java

Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters or both. Overloading is related to compile time (or static) polymorphism.

It varies with the number of parameters / data type for the same method name.

Return type is not considered here;

**Datatype:**

String – alphanumeric in double quotes

Int – stores whole numbers

Boolean – true / false

Double – 64 bit - decimal numbers – 5.0 5.5 5.6

Float – 32 bit – decimal

Long – longer number – integers – big ints

Char- single character – ‘a’ – single quotes

Byte –

Variable declaration & assignment

Variable used to store data

Datatype <variable name>= assignment ;

String name1 = “M”;

**Arrays**:

Collection of different values – all of same data types

String[] arr ={“A”,”B”, “C”} - > use this when you know all the values

String[] array = new string[10]; -> use this when you do not know the values

arr[0] – used to access elements

arr[0] = “Y”

**Loops** – Running same set of code multiple times

While(condition) {

} // as long the condition is true, the loop continues say a<5 when a=1

Do{

}while(condition); // loop executes atleast once

For (initialization; condition; increment) {

}

For(int I = 0; i< 2; i++){

System.out.println(“Hi”);

}

Scanner scanner = new Scanner(System.in);

String name;

do {

System.out.println("What is your name?");

name = scanner.nextLine();

} while (!name.equals("Louise"));

Operators:

Arithmetic / comparison:

Int x = 4;

X = x+1;

X+= 1

X++;

\*\*

X = 4, int a;

A=X++ 🡪 a=4; x=5

X = 4; int a;

A = ++x; 🡪 a=5, x=5

Comparison operator:

== != < <= > >= !

&& || =🡺 and or operators

System.out.println(5==5);

System.***out***.println(**"A"**.equals(**"A"**)==**true**);

**If statement**

If (4>2){

Print 4>2

}

Else {

Print 4<2

}

Nested if’s

**Switch – case**

```switch(x) {

case 2:

System.out.println("X is 2");

break;

case 4:

System.out.println("X is 4");

break;

case 6:

System.out.println("X is 6");

break;

default:

System.out.println("X is none");

}

}```

```String name = "SMU";

switch(name) {

case "SMU":

System.out.println("X is 2");

break;

case "UTD":

System.out.println("X is 4");

break;

case "BLAH":

System.out.println("X is 6");

break;

default:

System.out.println("X is none");

}```

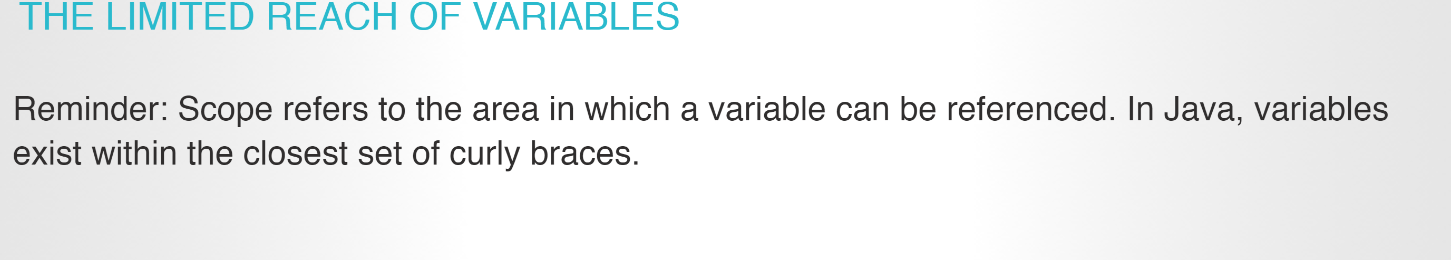
Method:

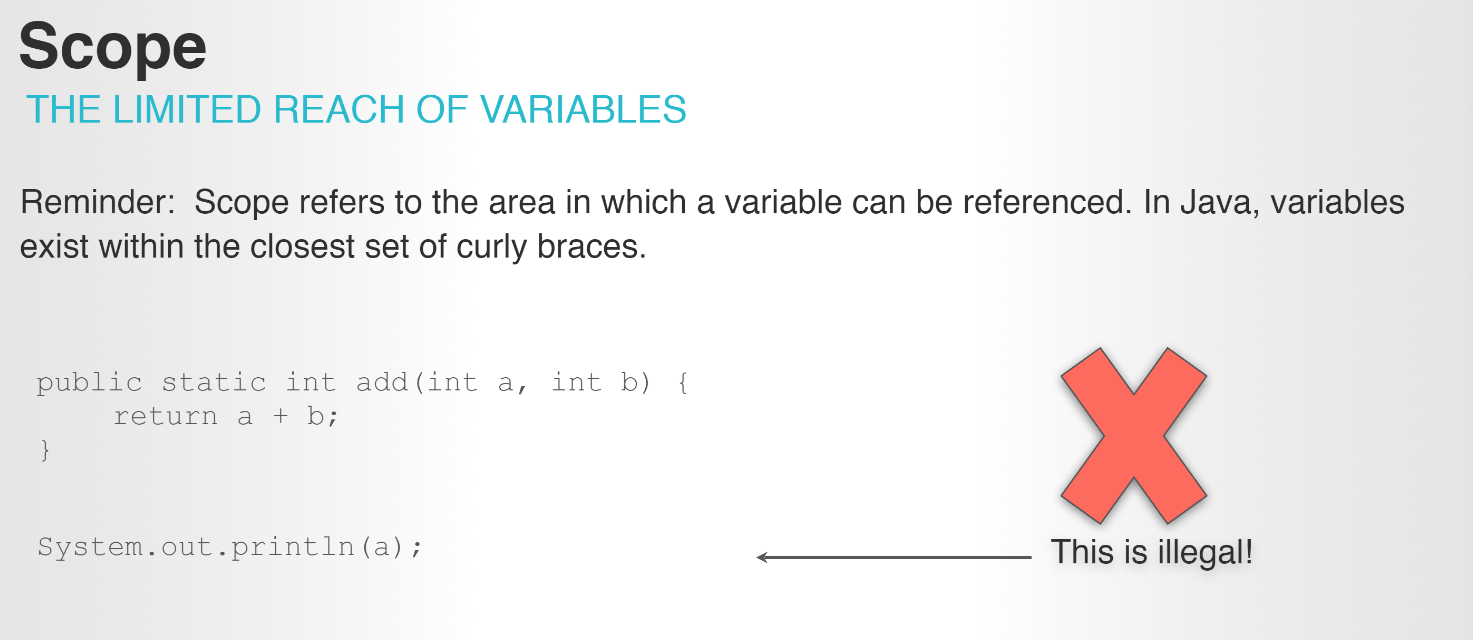
Dead code: Throws error since that part of code is not unreachable

Primitive is pass by value

And arrays passed are passed by reference

**Scope:**

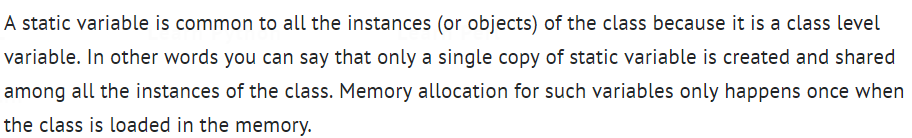




**Static**

[**https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html**](https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html)

If a variable is needed to be accessed in all the methods in a class, make it a class variable and suffix it with public static



As I mentioned above that the static variables are shared among all the instances of the class, they are useful when we need to do memory management. In some cases we want to have a common value for all the instances like global variable then it is much better to declare them static as this can save memory (because only single copy is created for static variables).

Class{

Public static int I =10;

Method 1(){

Print I // no error

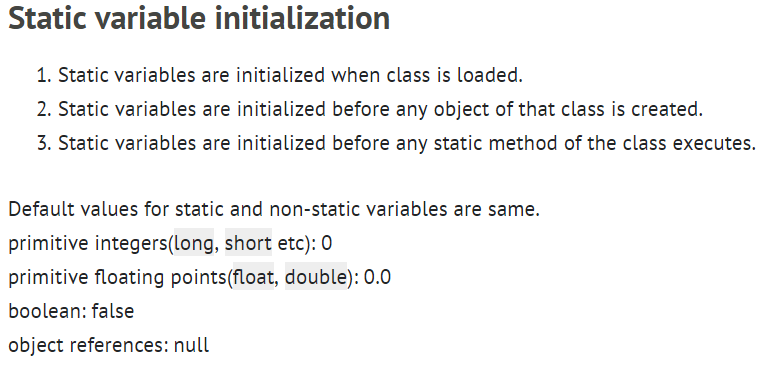
}

Method2{

Print I // no error

}

}



Example:

When a number of objects are created from the same class blueprint, they each have their own distinct copies of *instance variables*. In the case of the Bicycle class, the instance variables are cadence, gear, and speed. Each Bicycle object has its own values for these variables, stored in different memory locations.

Sometimes, you want to have variables that are common to all objects. This is accomplished with the static modifier. Fields that have the static modifier in their declaration are called *static fields* or *class variables*. They are associated with the class, rather than with any object. Every instance of the class shares a class variable, which is in one fixed location in memory. Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.

For example, suppose you want to create a number of Bicycle objects and assign each a serial number, beginning with 1 for the first object. This ID number is unique to each object and is therefore an instance variable. At the same time, you need a field to keep track of how many Bicycle objects have been created so that you know what ID to assign to the next one. Such a field is not related to any individual object, but to the class as a whole. For this you need a class variable, numberOfBicycles, as follows:

public class Bicycle {

private int cadence;

private int gear;

private int speed;

// **add an instance variable for the object ID**

private int id;

// **add a class variable for the**

// **number of Bicycle objects instantiated**

private **static** int numberOfBicycles = 0;

...

}

Class variables are referenced by the class name itself, as in

Bicycle.numberOfBicycles

This makes it clear that they are class variables.

You can use the Bicycle constructor to set the id instance variable and increment the numberOfBicycles class variable:

public Bicycle(int startCadence, int startSpeed, int startGear){

gear = startGear;

cadence = startCadence;

speed = startSpeed;

// **increment number of Bicycles**

// **and assign ID number**

**id = ++numberOfBicycles;**

}

Methods

Method overloading

**Class – Objects:**

Object oriented programming

Class is a blueprint of an object

AN object is an instance of a class

Class – blueprint of a building, recipe

Object – House, cake

Each object has its own properties

Student S1 = new Student();

Student S2 = new Student();

S1.name=”a”;

S2.name = “B”;

Each object has its own properties(values)

Private members in class – values can be only changed inside class and not objects

Public members – can be accessed outside class using objects

This – specific properties of that object

Getter, setter function

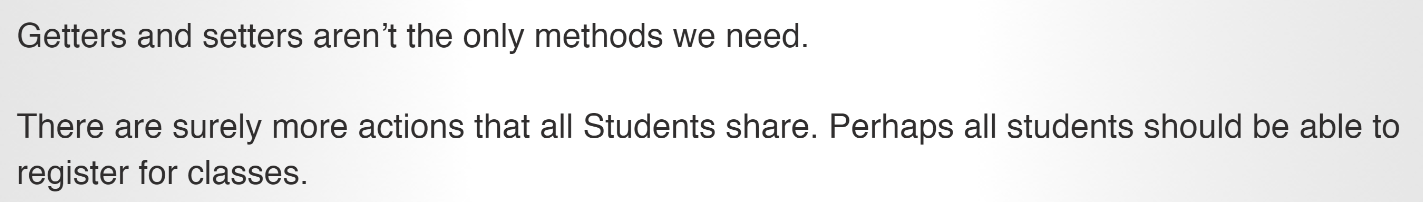
This.name = “Hello”

Every class variable can be set using the **setter** method. In the below example, id is a class variable. Its value is set using *setId* 🡺 set<Class variable>. The value can be accessed from the object using the **getter** method. In this case it is the *getId* 🡪 get<Class variable>

**public class** student\_id\_gpa {  
  
 **private int id**;  
 **private double gpa**;  
  
  
 **public int** getId() {  
 **return this**.**id**;  
  
 }  
  
 **public void** setId(**int** id){  
 **this**.**id** = id;  
 }  
  
 **public double** getGpa() {  
 **return this**.**gpa**;  
 }  
  
 **public void** setGpa( **double** gpa){  
 **if**(gpa >= 4.0)  
 {  
 **this**.**gpa** = 4.0;  
 }  
 **else  
 this**.**gpa** = gpa;  
 }  
}

----

**public class** id\_gpa {  
  
 **public static void** main(String[] args){  
  
 *// creating 2 objects* student\_id\_gpa s1 = **new** student\_id\_gpa();  
 student\_id\_gpa s2 = **new** student\_id\_gpa();  
  
 Scanner scanner = **new** Scanner(System.***in***);  
 **int** id;  
 **double** gpa;  
  
 System.***out***.println(**"Enter id for first student"**);  
 id = scanner.nextInt();  
 scanner.nextLine();  
  
 System.***out***.println((**"Enter gpa for first Student"**));;  
 gpa = scanner.nextDouble();  
  
  
 *// setting the values for the first student* s1.setGpa(gpa);  
 s1.setId(id);  
  
 *//Getting the values for the second student* System.***out***.println(**"Enter id for second student"**);  
 id = scanner.nextInt();  
 scanner.nextLine();  
  
 System.***out***.println((**"Enter gpa for second Student"**));;  
 gpa = scanner.nextDouble();  
  
 s2.setGpa(gpa);  
 s2.setId(id);  
  
 **for** (**int** i = 0; i<=1; i++)  
 {  
 System.***out***.println(**"For "**+ i+1 + **" student"**);  
 System.***out***.println(**"-----------------------"**);  
  
 **if** (i==0)  
 {  
 System.***out***.println(**"Id: "**+ s1.getId());  
 System.***out***.println(**"gpa: "**+ s1.getGpa());  
  
 }  
  
 **if** (i==1)  
 {  
 System.***out***.println(**"Id: "**+ s2.getId());  
 System.***out***.println(**"gpa: "**+ s2.getGpa());  
  
 }  
 }



Constructors

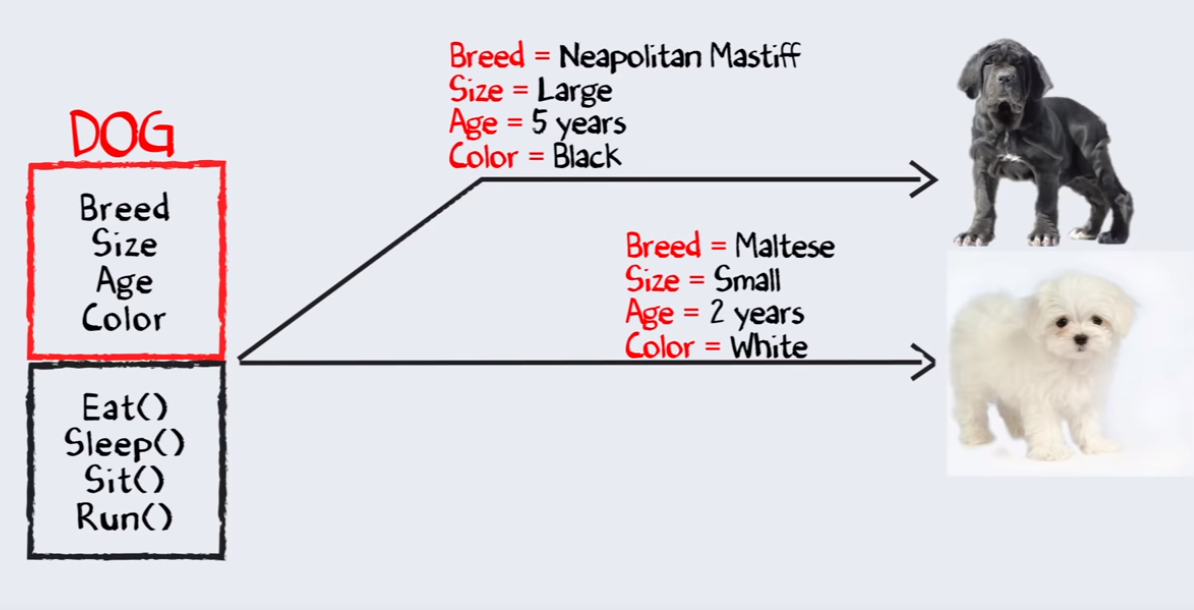
Overloading constructors

Class variables: The common things among various objects

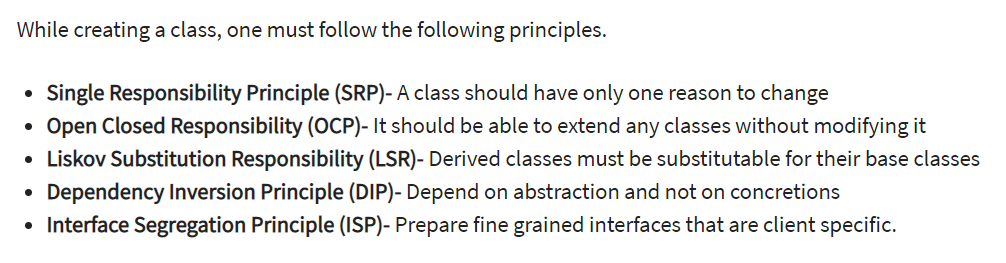
Class methods – Common actions among many objects

Example – Among Dogs like Chihuaha, lab, Hutch dog – the common thing / characteristics that all this dogs have are the breed, size, age, color – data members

Common actions – eat, sleep, sit, run – methods



<https://www.youtube.com/watch?v=MeP1CztNMdo>



A class is an entity that determines how an object will behave and what the object will contain. In other words, it is a blueprint or a set of instruction to build a specific type of object.

An object is nothing but a self-contained component which consists of methods and properties to make a particular type of data useful. Object determines the behavior of the class. When you send a message to an object, you are asking the object to invoke or execute one of its methods.

From a programming point of view, an object can be a data structure, a variable or a function. It has a memory location allocated. The object is designed as class hierarchies.

A **class**is a **blueprint or prototype** that defines the variables and the methods (functions) common to all objects of a certain kind.

An **object**is a specimen of a class. Software objects are often used to model real-world objects you find in everyday life.

**Inheritance**: Once class extending to another

<https://www.tutorialspoint.com/java/java_inheritance.htm>

## Subclass Constructors

The following example illustrates how to use the super keyword to invoke a superclass's constructor. Recall from the [Bicycle](https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html)example that MountainBike is a subclass of Bicycle. Here is the MountainBike (subclass) constructor that calls the superclass constructor and then adds initialization code of its own:

public MountainBike(int startHeight,

int startCadence,

int startSpeed,

int startGear) {

super(startCadence, startSpeed, startGear);

seatHeight = startHeight;

}

Invocation of a superclass constructor must be the first line in the subclass constructor.

The syntax for calling a superclass constructor is

super();

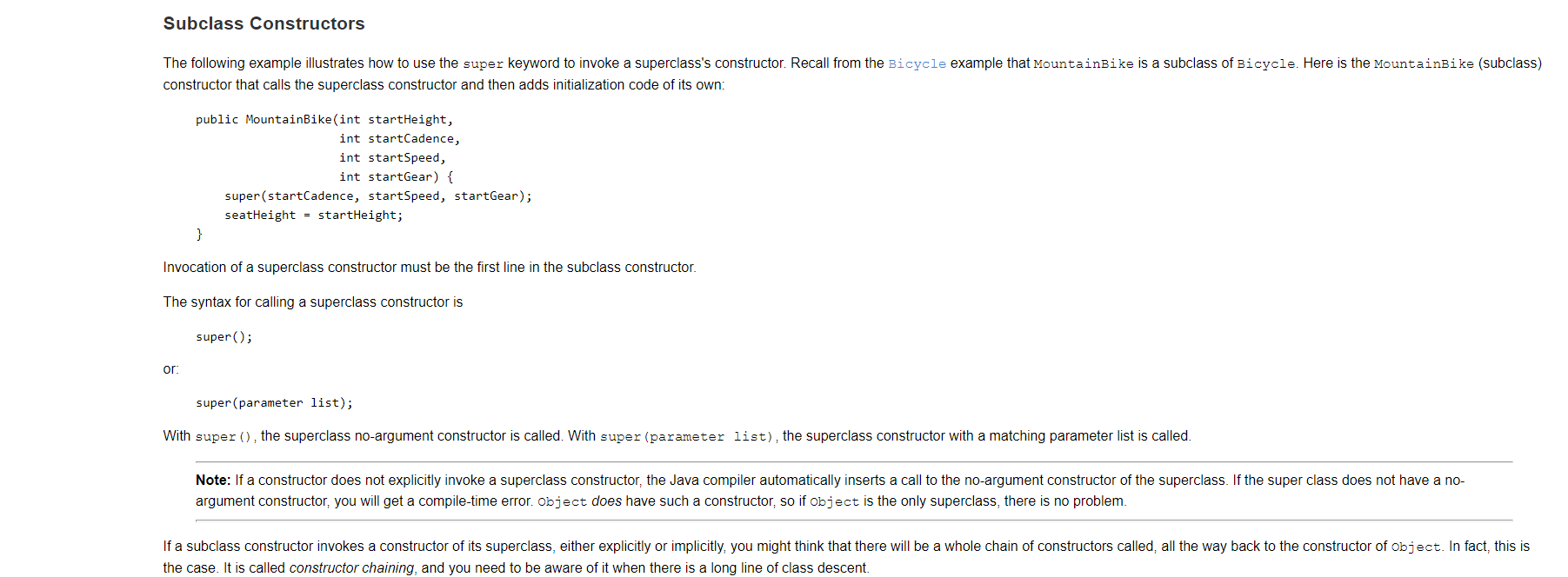
or:

super(parameter list);

With super(), the superclass no-argument constructor is called. With super(parameter list), the superclass constructor with a matching parameter list is called.

**Note:** If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts **a call** to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.

If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that there will be a whole chain of constructors called, all the way back to the constructor of Object. In fact, this is the case. It is called *constructor chaining*, and you need to be aware of it when there is a long line of class descent.



* + - Base class – no constructor ; Sub class – no constructor – no error
  + Base class – no Constructor; sub class – no arg constructor – no error since sub class no arg constructor will call default base class no arg constructor
  + Base class – no Constructor; sub class – no arg constructor, one arg constructor (created object with one arg) – no error since sub class one arg constructor will call default base class no arg constructor
  + Base class- one arg constructor; sub class - no arg constructor, one arg constructor (created object with one arg); In sub class constructor – there is no calling base class constructor – it throws error saying in sub class that there is **no default constructor in Base class**

No error until any base class constructor (with args) is defined. The moment when a base class constructor (with args) is defined, sub class should have an explicit call to the base class constructor.

Base class has no arg constructor; Sub class has all possibilities of constructor ; there is no error when sub class is not explicitly invoking base class constructor since Compiler automatically invokes base class – no arg constructor

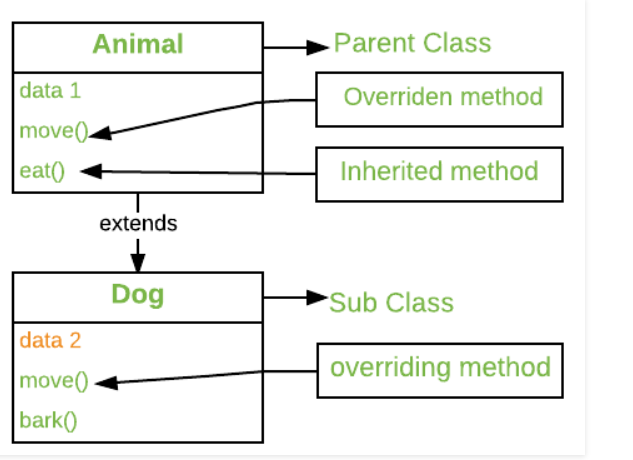
1. **--> Call to super() must be first statement in Derived(Student) Class constructor.**

Project: **learnInheritanceHome (Animal, Dog, AnimalDogMain – java files)**

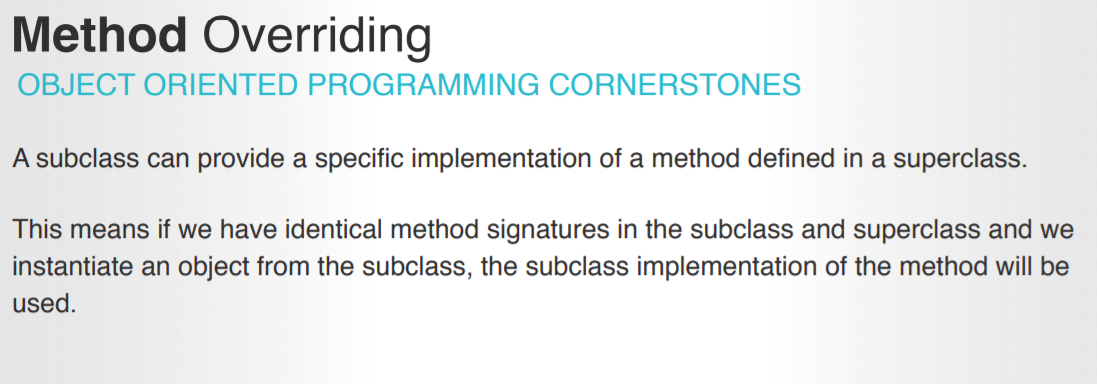
* + Method overriding:

# Overriding in Java

In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.



Method overriding is one of the way by which java achieve [Run Time Polymorphism](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/).The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.



Class a – paint()

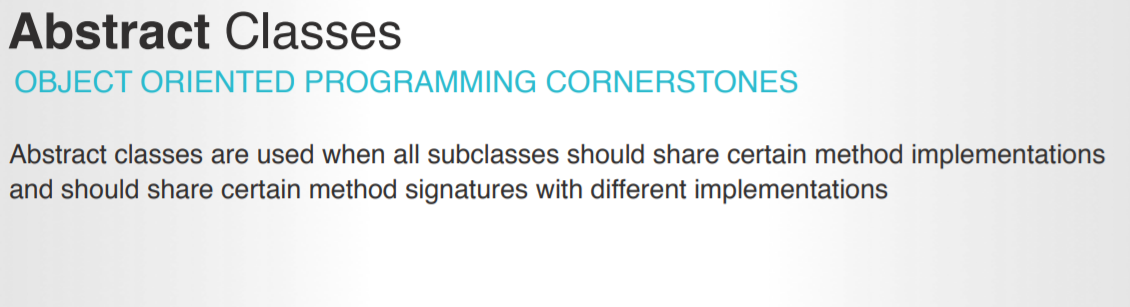
Class b extends a – paint()

B b = new B()

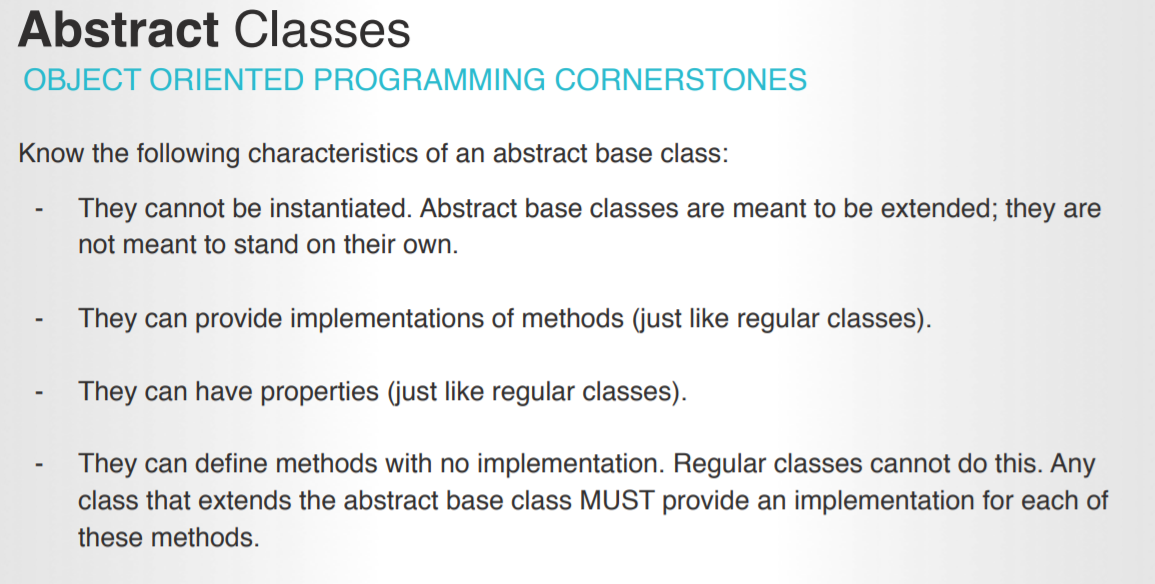
b.paint() -> invokes B’s paint() . This is called method overriding

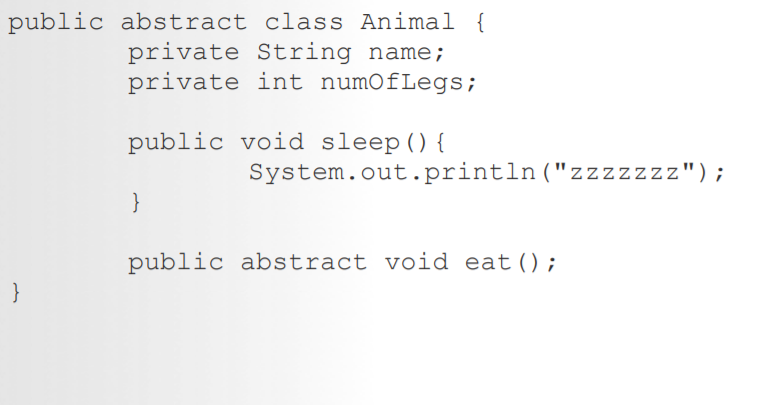
### Rules for method overriding:

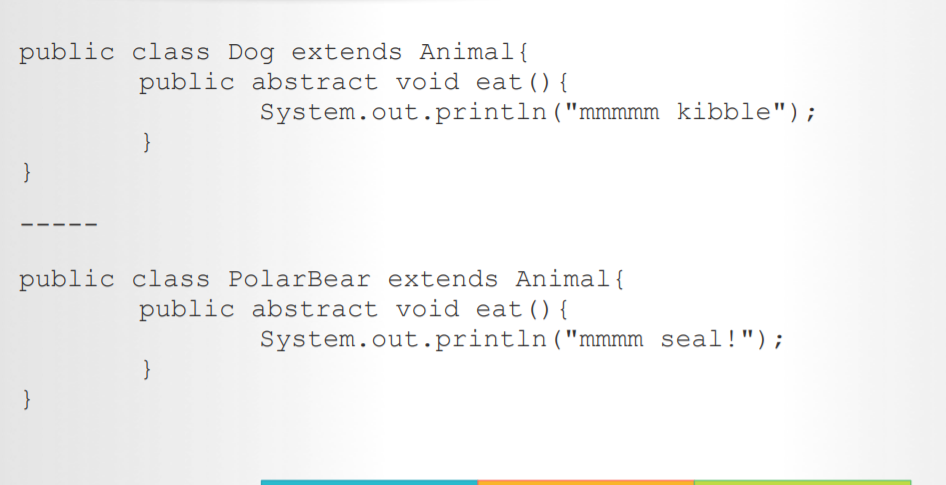
* In java, a method can only be written in [Subclass](https://crunchify.com/top-10-java-interview-questions-answers-must-read-before-appearing-for-any-java-interview/), not in same class.
* The argument list should be exactly the same as that of the overridden method.
* The return type should be the same or a subtype of the return type declared in the original overridden method in the [super class](https://crunchify.com/create-simple-pojo-and-multiple-java-reflection-examples/).
* The access level cannot be more restrictive than the overridden method’s access level.
  + For example: if the super class method is declared public then the over-ridding method in the sub class cannot be either private or [protected](https://crunchify.com/java-how-to-create-your-own-logging-level-in-log4j-configuring-log4j/).
* Instance methods can be overridden only if they are inherited by the subclass.
* A method declared final cannot be overridden.
* A [method declared static](https://crunchify.com/fundamentals-of-java-static-methods-and-variables/) cannot be overridden but can be re-declared.
* If a method cannot be inherited then it cannot be overridden.
* A subclass within the same package as the instance’s superclass can override any superclass method that is not declared private or final.
* A subclass in a different package can only override the non-final methods declared public or protected.
* An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws [exceptions](https://crunchify.com/how-to-fix-exception-in-thread-main-java-lang-illegalmonitorstateexception-error-on-thread-wait/) or not.
  + However the overriding method should not throw [checked exceptions](https://crunchify.com/better-understanding-on-checked-vs-unchecked-exceptions-how-to-handle-exception-better-way-in-java/) that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
* [Constructors](https://crunchify.com/how-to-implement-simple-circulararraylist-in-java/) cannot be overridden.



* + - It has abstract method – method without implementation







Abstract method can have a return type and can take parameter. The signature must match in the sub class implementation of the abstract methods.

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

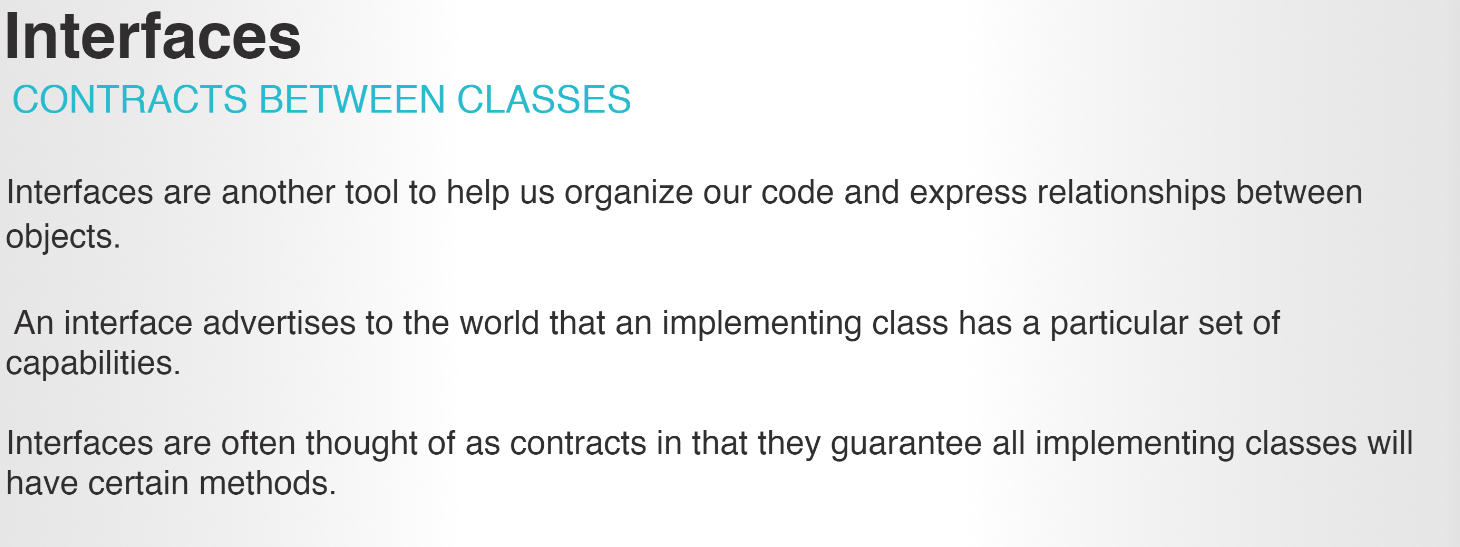
* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have constructors and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.

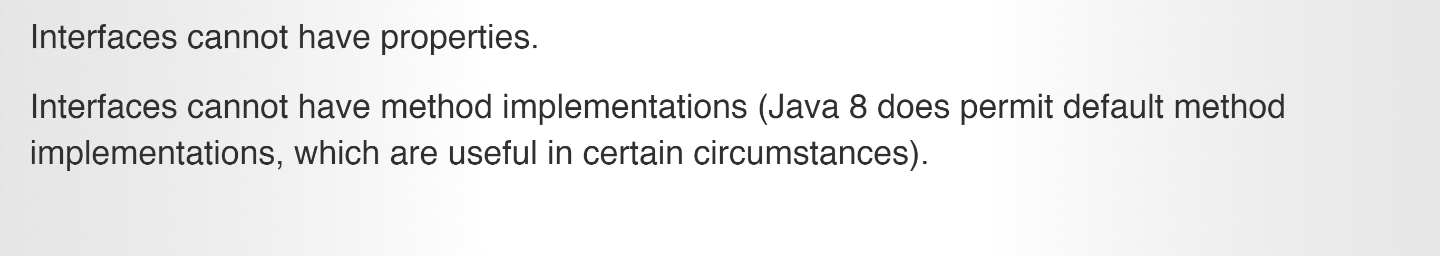
<https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html>

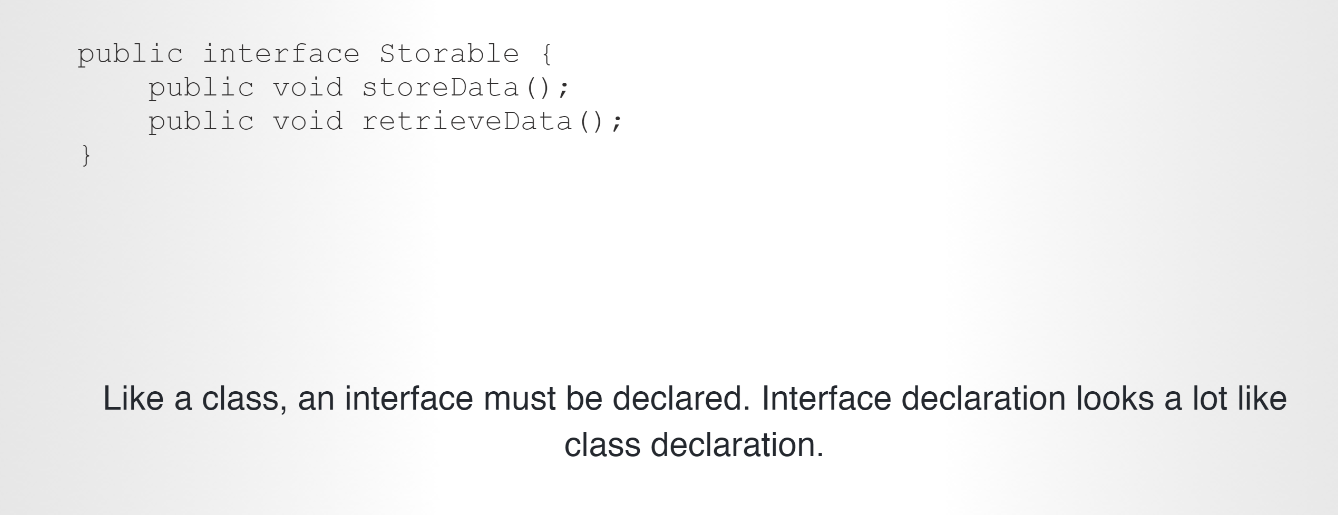
**Protected** Access Modifier - **Protected**

Variables, methods, and constructors, which are declared **protected** in a superclass can be accessed only by the subclasses in other package or any class within the package of the **protected** members' class. The **protected** access modifier cannot be applied to class and interfaces.

**Interface:**

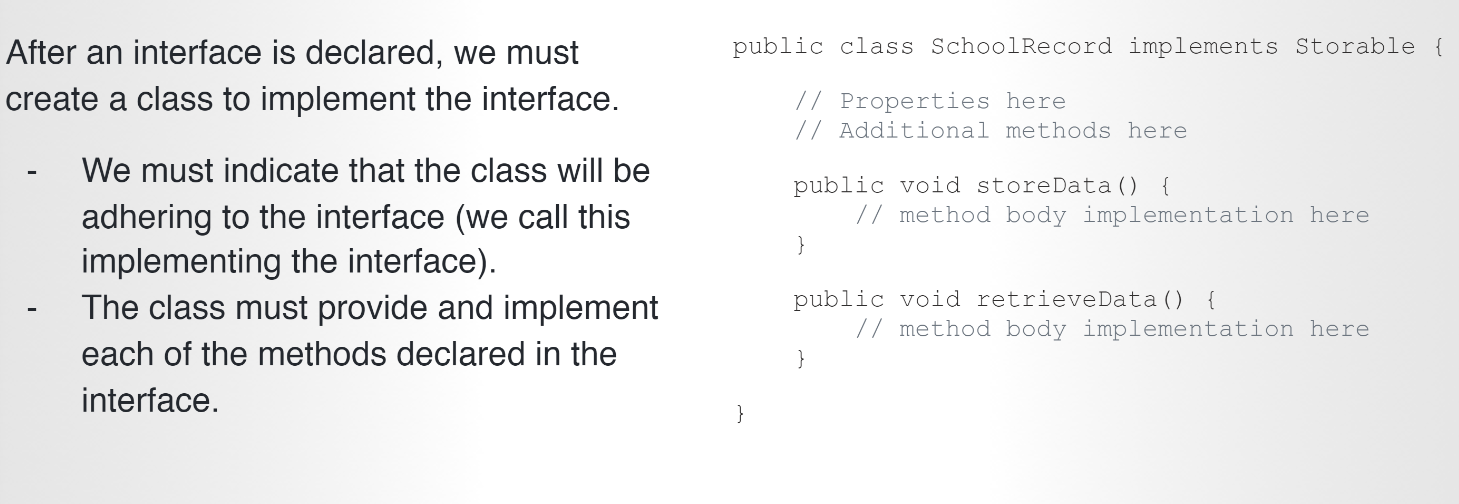






**\_> No variables**

* + **Only method declaration**



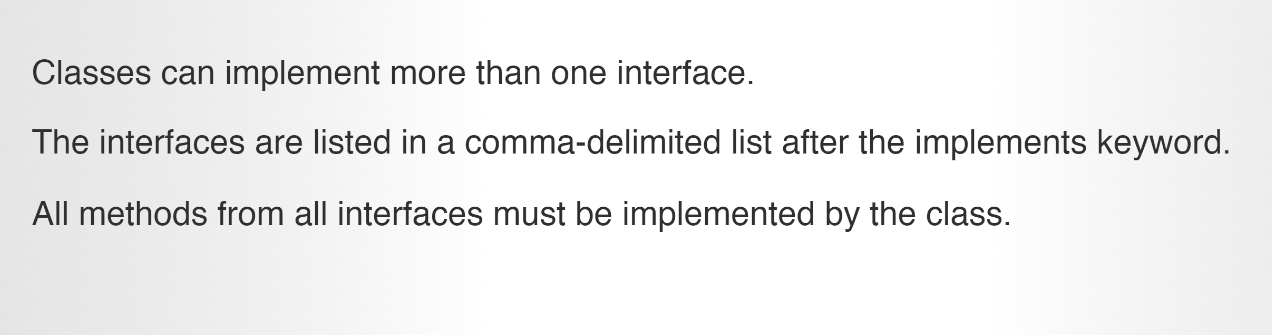
**Difference between abstract class and Interface**

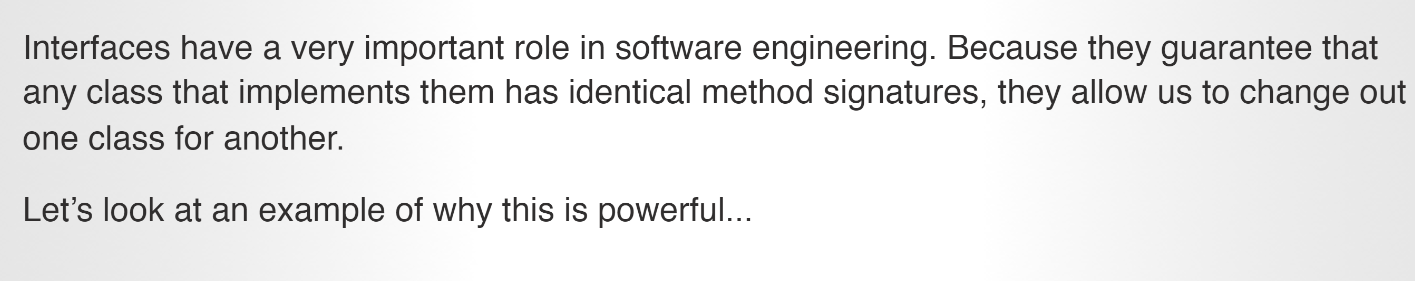
Abstract:

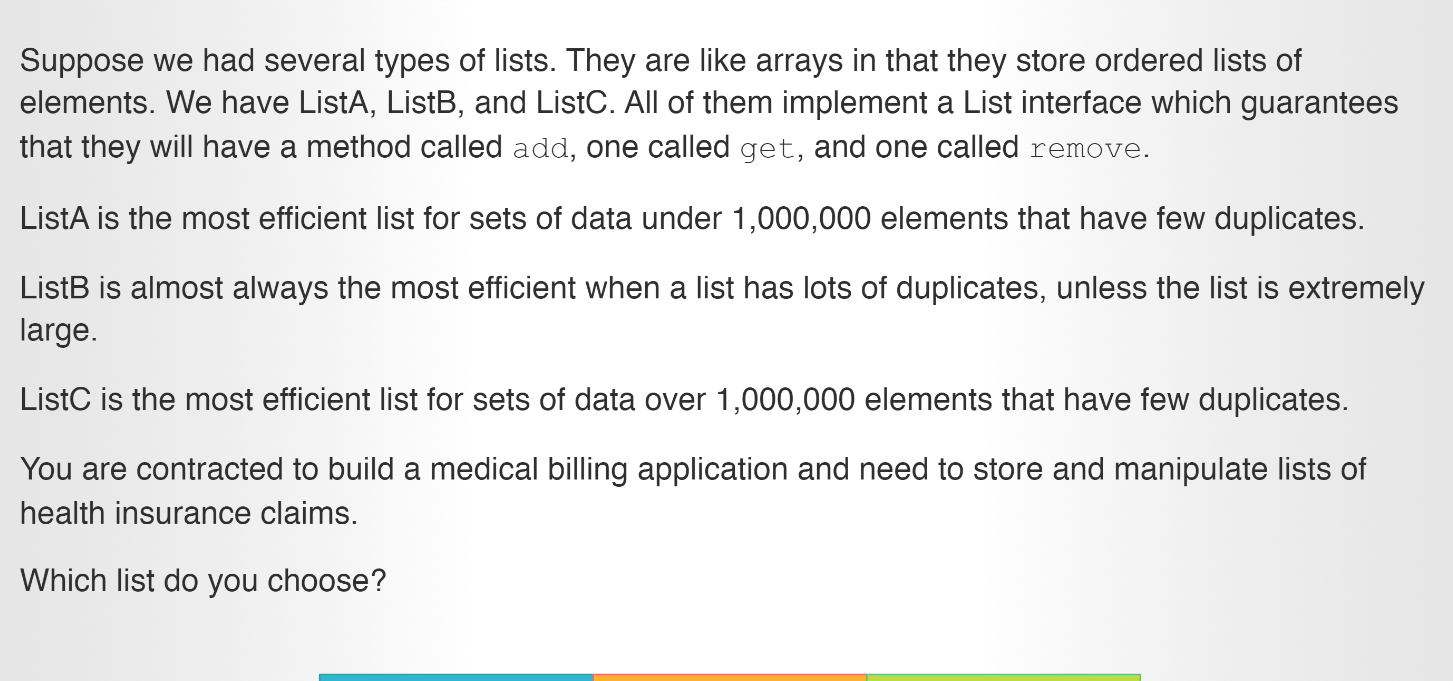
1. Can contain Properties (class level variable)
2. Can contain abstract methods – method signatures
3. Can contain regular method implementations – real methods with code

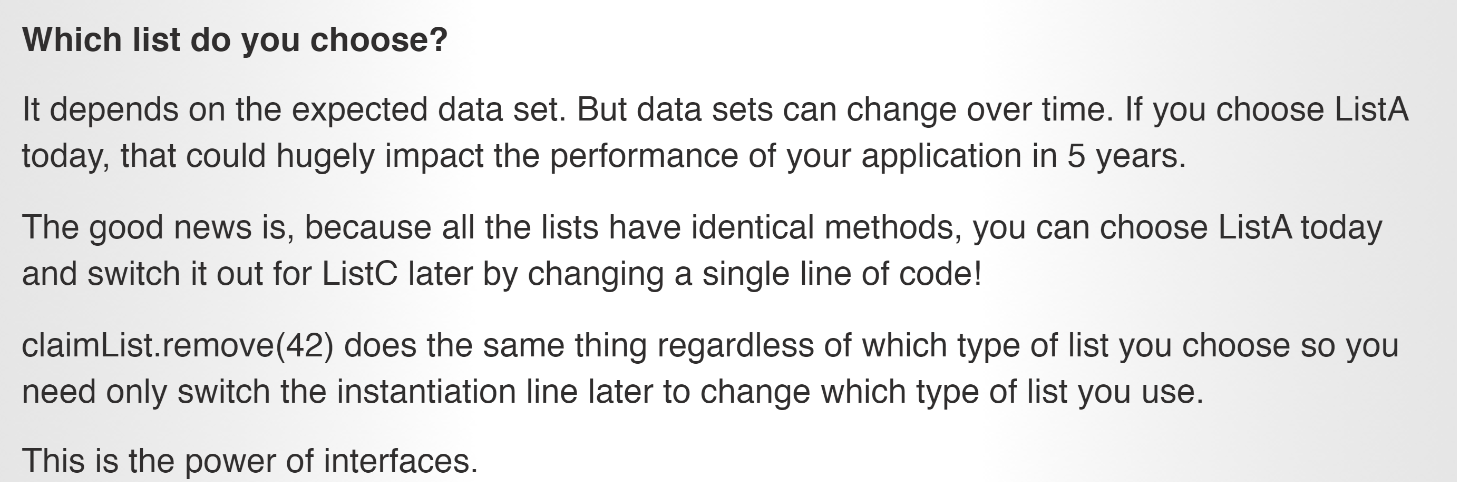
Interface:

1. Only contain method signatures
2. No variables









**public interface** Storable {  
  
 **public void** storeData();  
 **public void** retrieveData();  
}

**public interface** Auditable {  
  
 **public void** runAudit();  
 **public void** sendAuditToState();  
}

**public class** SchoolRecord **implements** Auditable, Storable{  
  
 @Override  
 **public void** runAudit() {  
 System.***out***.println(**"Auditing"**);  
 }  
  
 @Override  
 **public void** sendAuditToState() {  
 System.***out***.println(**"Confirmed"**);  
 }  
  
 @Override  
 **public void** retrieveData() {  
 System.***out***.println(**"Retrieving data"**);  
 }  
  
 @Override  
 **public void** storeData() {  
 System.***out***.println(**"Here, I am storing data"**);  
  
 }  
}

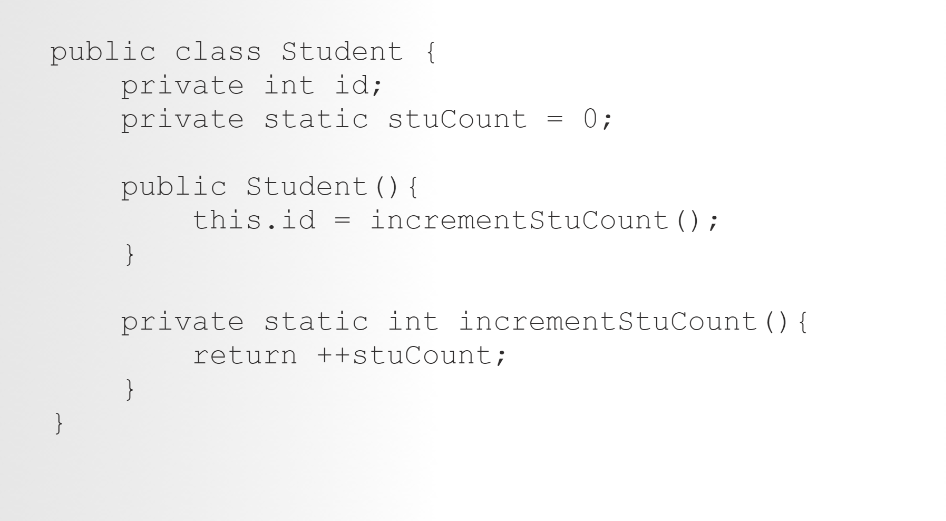
**When to use abstract vs Interface:** <https://dzone.com/articles/when-to-use-abstract-class-and-intreface>

**Static:**



Static method can only access static variables

Regular methods can also access static variables



No need to use of **this** to access static variables inside methods

Static is not associated with Objects. It is tied to a class.

**Why is main method static?**

This is neccesary because **main**() is called by the JVM before any objects are made. Since it is **static** it can be directly invoked via the class. Similarly, we use **static**sometime for user defined methods so that we need not to make objects. void indicates that the **main**() method being declared does not return a value.

Static: <https://www.baeldung.com/java-static>

**Project:** dailyAssessment\_07\_03

From the memory perspective, **static variables go in a particular pool in JVM memory called Metaspace**(before Java 8, this pool was called Permanent Generation or PermGen, which was completely removed and replaced with Metaspace).

* When the value of variable is independent of objects
* When the value is supposed to be shared across all objects
* Since*static*variables belong to a class, they can be accessed directly using class name and don’t need any object reference
* *static* variables can only be declared at the class level
* *static* fields can be accessed without object initialization
* Although we can access *static* fields using an object reference (like *ford.numberOfCars++*) , we should refrain from using it as in this case it becomes difficult to figure whether it’s an instance variable or a class variable; instead, we should always refer to *static* variables using class name (for example, in this case, *Car.numberOfCars++*)

**Static methods:**

* + *static* methods are also widely used to create utility or helper classes so that they can be obtained without creating a new object of these classes.

1. To access/manipulate static variables and other static methods that don’t depend upon objects
2. *static*methods are widely used in utility and helper classes

* *static* methods in Java are resolved at compile time. Since method overriding is part of Runtime Polymorphism, **so static methods can’t be overridden**
* abstract methods can’t be static
* *static* methods cannot use *this* or *super* keywords
* The following combinations of the instance, class methods and variables are valid:
  1. Instance methods can directly access both instance methods and instance variables
  2. Instance methods can also access *static* variables and *static* methods directly
  3. *static* methods can access all *static* variables and other *static* methods
  4. ***static* methods cannot access instance variables and instance methods directly**; they need some object reference to do so

**Static block:**

A *static* block is used for initializing *static* variables. Although *static* variables can be initialized directly during declaration, there are situations when we’re required to do the multiline processing.

In such cases, *static* blocks come in handy.

**If *static* variables require additional, multi-statement logic while initialization, then a *static* block can be used.**

**5.1. The *static* Block Example**

Suppose we want to initialize a list object with some pre-defined values.

This becomes easy with *static*blocks:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class StaticBlockDemo {      public static List<String> ranks = new LinkedList<>();        static {          ranks.add("Lieutenant");          ranks.add("Captain");          ranks.add("Major");      }        static {          ranks.add("Colonel");          ranks.add("General");      }  } |

In this example, it wouldn’t be possible to initialize *List* object with all the initial values along with declaration; and that’s why we’ve utilized the *static* block here.

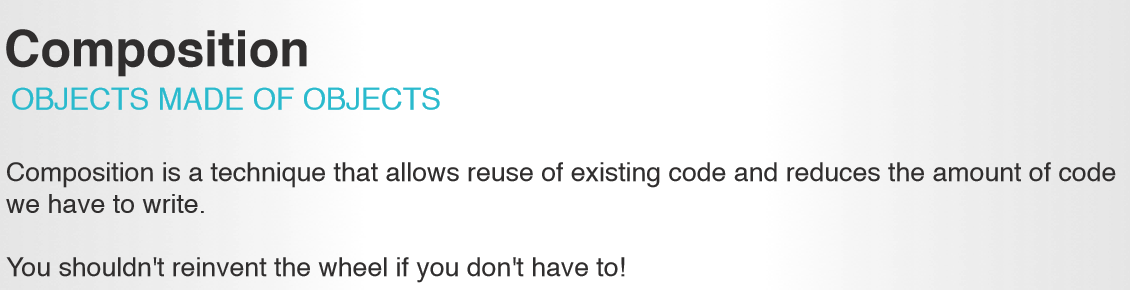
**5.2. Compelling Reasons to Use *static* Blocks**

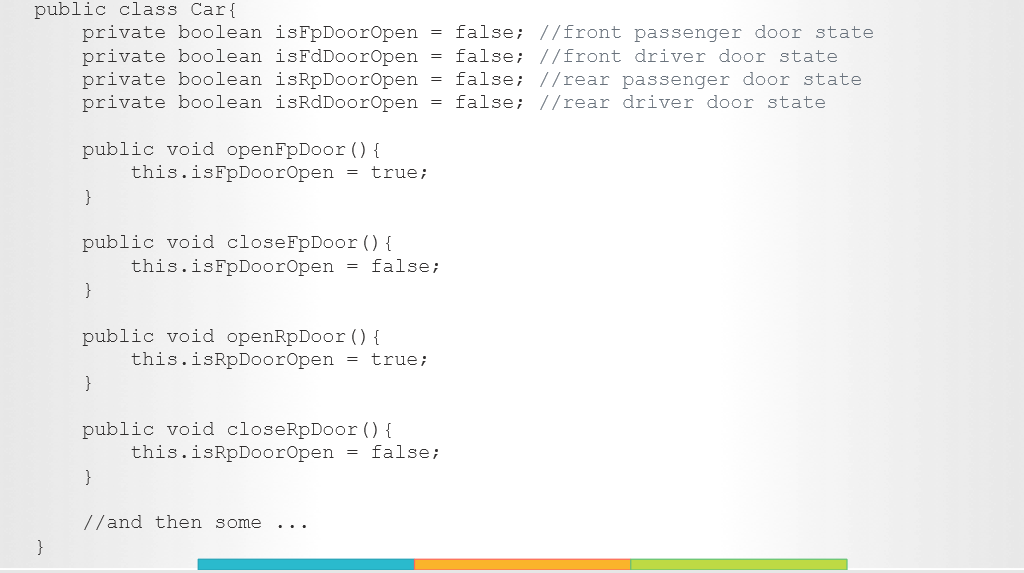
* If initialization of *static*variables requires some additional logic except the assignment
* If the initialization of static variables is error-prone and requires exception handling

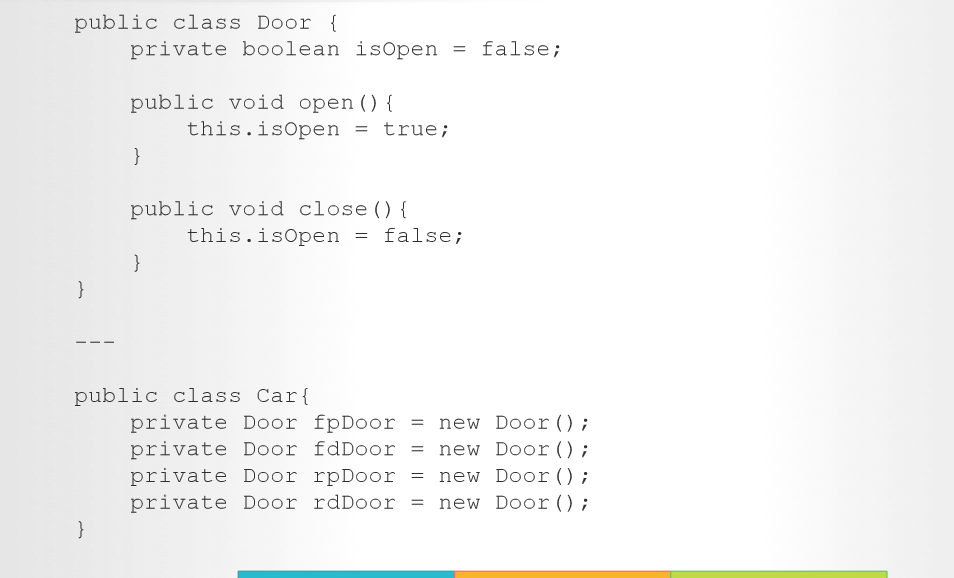
**5.3. Key Points to Remember**

* A class can have multiple *static* blocks
* *static* fields and *static* blocks are resolved and executed in the same order as they are present in the class

**Composition:**







**Composition is made up of other classes**

* + Smaller part of other classes
  + Class person{

Class head;

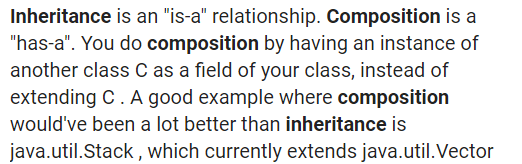
Arm leftArm = new Arm()

Arm rightArm; = new Arm()

Class leg

}

It reduces the amount of code



**Dog is a mammal - so inheritance involved**

**Whereas Car has doors – here composition is involved**

**Multiple inheritance is not supported in Java .**

Class a has show()

Class b has show()

Class C extends a&b. When creating object for C

C cObj = new C()🡪 When calling C.show(), compiler doesn’t know which show() to invoke whethere a->show() or b->show(). To avoid this error, java didn’t implement multiple inheritance

The same can be done through interfaces

Interface a ->show()

Interface b->show()

Class c implements interface a and b

Class c{

Show(){

Interfacea.super.show()

Interfaceb.super.show()

}}

Using the above, we can tell the compiler which one to invoke explicitly

[**https://www.geeksforgeeks.org/java-and-multiple-inheritance/**](https://www.geeksforgeeks.org/java-and-multiple-inheritance/)

**07/08/2019**

**Lists**

Array list:

## Java ArrayList

The ArrayList class is a resizable [array](https://www.w3schools.com/java/java_arrays.asp), which can be found in the java.util package.

The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an ArrayListwhenever you want. The syntax is also slightly different:

Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

public class MyClass {   
  public static void main(String[] args) {   
    ArrayList<String> cars = new ArrayList<String>();  
    cars.add("Volvo");  
    cars.add("BMW");  
    cars.add("Ford");-🡪 adds an item  
    cars.add("Mazda");  
    System.out.println(cars);

cars.get(0); 🡪 access an item

cars.set(0, "Opel"); 🡪 updates an item

cars.remove(0); 🡪 removes an item

cars.clear();🡪 removes all items

cars.size(); 🡪 To find out how many elements an ArrayList have, use the size method:

loops through the list:

for (int i = 0; i < cars.size(); i++) {  
      System.out.println(cars.get(i));

or

for each

 for (String i : cars) {  
      System.out.println(i);  
    }

## Sort an ArrayList

ArrayList<String> cars = new ArrayList<String>();  
    cars.add("Volvo");  
    cars.add("BMW");  
    cars.add("Ford");  
    cars.add("Mazda");  
  
    Collections.sort(cars);  // Sort cars

**Maps: Key value pair**

Arrays store items as an ordered collection, and you have to access them with an index number (int type). A HashMap however, store items in "**key**/**value**" pairs, and you can access them by an index of another type (e.g. a String).

HashMap<String, String> capitalCities = new HashMap<String, String>();

ublic class MyClass {  
  public static void main(String[] args) {  
  
    // Create a HashMap object called capitalCities  
    HashMap<String, String> capitalCities = new HashMap<String, String>();  
  
    // Add keys and values (Country, City)  
    capitalCities.put("England", "London");  
    capitalCities.put("Germany", "Berlin");  
    capitalCities.put("Norway", "Oslo");  
    capitalCities.put("USA", "Washington DC");  
    System.out.println(capitalCities);   
  }  
}

capitalCities.get("England"); -🡪 access an item

capitalCities.remove("England"); 🡪 removes an item

capitalCities.clear(); => removes all items

capitalCities.size();

**Note:** Use the keySet() method if you only want the keys, and use the values() method if you only want the values:

// Print keys  
for (String i : capitalCities.keySet()) {  
  System.out.println(i);  
}

// Print values  
for (String i : capitalCities.values()) {  
  System.out.println(i);  
}

for (String i : capitalCities.keySet()) {  
  System.out.println("key: " + i + " value: " + capitalCities.get(i));  
}

**public class** Student {  
  
 **private static int** *statNum*;  
 **private int id**;  
 **private** String **name**;  
  
 **public** Student(String name) {  
 **this**.**id** = ++*statNum*;  
 **this**.**name** = name;  
 }

Map<Integer,Student> map = **new** HashMap<>();  
 **int** numStat = 1000;  
  
  
map.put(++numStat, **new** Student(**"a"**));  
map.put(++numStat, **new** Student(**"b"**));  
map.put(++numStat, **new** Student(**"c"**));  
map.put(++numStat, **new** Student(**"d"**));  
map.put(++numStat, **new** Student(**"e"**));  
  
Iterator<Map.Entry<Integer,Student>> entryIterator = map.entrySet().iterator();  
  
**while** (entryIterator.hasNext()){  
 Map.Entry<Integer,Student> pair = entryIterator.next();  
 Student stu = pair.getValue();  
 System.***out***.println(**"Student info: "**+ stu.getName()+**"\t"**+ stu.getId() );  
}

**https://docs.oracle.com/javase/8/docs/api/java/util/Map.html**

Movie:

Movie name: <name>

Year: 2000

Genre: thriller

No order for maps

Map<String, Integer> heightsMap = **new** HashMap<>();

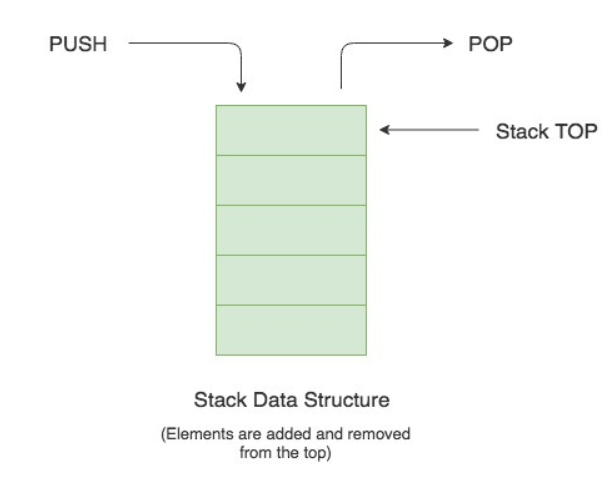
Queue- FIFO

Stack – LIFO

Linked list

**Stack:**

Java Collection framework provides a Stack class which models and implements Stack data structure. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek.



Stack<String> stackOfCards = new Stack<>();

// Pushing new items to the Stack

stackOfCards.push("Jack");

stackOfCards.push("Queen");

stackOfCards.push("King");

stackOfCards.push("Ace");

String cardAtTop = stackOfCards.pop(); // Throws EmptyStackException if the stack is empty

System.out.println("Stack.pop() => " + cardAtTop);

System.out.println("Current Stack => " + stackOfCards);

System.out.println();

// Get the item at the top of the stack without removing it

cardAtTop = stackOfCards.peek();

System.out.println("Stack.peek() => " + cardAtTop);

System.out.println("Current Stack => " + stackOfCards);

Push/pop/peek/empty/search

Search returns the index number of the searched object from the top

integerStack.search(10)

integerstack has inserted 10,20,30,40.

Search for 40 will return 1

30->2

20-> 3

40->4

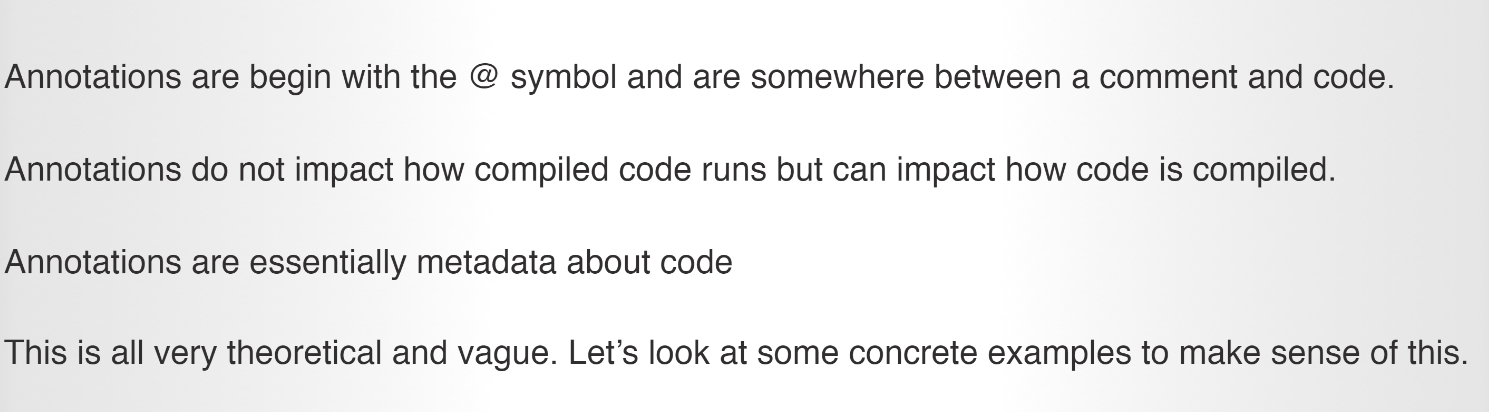
**Queue:** it follows the FIFO or the First-In-First-Out principle.

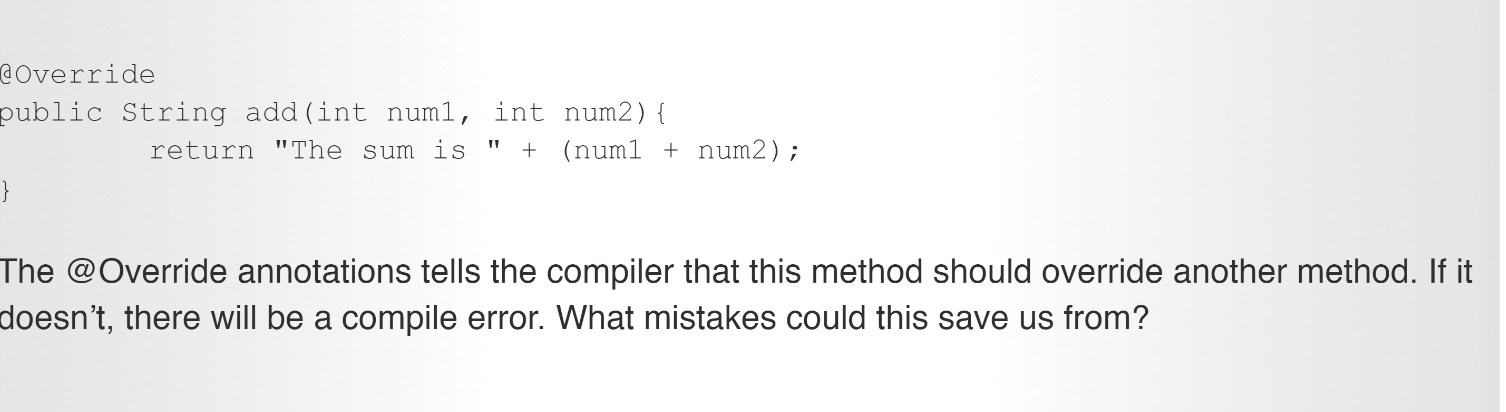
* The Queue is used to insert elements at the end of the queue and removes from the beginning of the queue. It follows FIFO concept.

**Linked list**: List<Integer> integerList = **new** LinkedList<>();  
  
integerList.add(10);  
integerList.add(20);  
integerList.add(30);  
integerList.add(40);  
  
integerList.get(1);  
  
System.***out***.println(**"Second element in the linked list: "**+ integerList.get(1));  
  
**for** (**int** i : integerList  
 ) {  
 System.***out***.println(i);  
}  
  
integerList.remove(2);  
  
  
**for** (**int** i : integerList  
) {  
 System.***out***.println(i);  
}

@ - not comment but telling compiler something

**@Annotations: @Deprecated, @Suppressedwarnings**





## @SuppressWarnings****Annotation****

Compiler warning messages are usually helpful. Sometimes warnings can get noisy, though.

. If we don’t want to fix the warning, then **we can suppress it with the @SuppressWarnings annotation**.

## What is REST?

REST stands for [Representational state transfer](http://en.wikipedia.org/wiki/Representational_state_transfer) which essentially refers to a style of web architecture that has many underlying characteristics and governs the behavior of clients and servers.

-> Restricting access

-> Request & response

-> Request: Get/balance - URI

-> respose will be in format of json

Get

Post

Put

Delete

Status: 2 – success

Spring

<https://start.spring.io/>

Using spring, api are written

@RestController  
**public class** FirstLastName {  
  
 @RequestMapping (value=**"/firstLast/{first}/{last}"**, method= RequestMethod.***GET***)  
 **public** String getFirstLast(@PathVariable String first, @PathVariable String last) {  
 **return** first+ **" "**+ last;  
 }  
  
}

Link: [http://localhost:8080/**firstLast/**kk/tp](http://localhost:8080/firstLast/kk/tp)

Here firstLast in link and value in @ RequestMapping should match. This tells the server should invoke this value. When that is invoked, the corresponding function gets invoked which is the getFirstLast is executed –

Later the query parameters that is in the link after

Integer.parseInt – converts string to int

String function – Str.endWith(“a”)

Exception: - unchecked & checked

Unchecked: program will compile , you don’t need to handle it in try catch but the pgm may crash

Checked: Program will not compile until the exceptions are handled

Handling errors. It is there not to make the program crash

RuntimeException – UnChecked

Exception – Checked

Exception names: IllegalArgumentException

IllegalArgumentException

IllegalArgumentException

Checked and unchecked exceptions

**Reversing a list:**

List<Product> expectedListAsc = Arrays.*asList*(**product3**, **product4**, **product2**, **product1**);  
List<Product> expectedListDesc = **new** ArrayList<>(expectedListAsc);  
Collections.*reverse*(expectedListDesc);

**Sorting using streams**

**if** (sort.toLowerCase().equals(**"asc"**))  
 sortedList = productList.stream()  
 .sorted(Comparator.*comparingDouble*(Product::getProductPrice))  
 .collect(Collectors.*toList*());  
**else** sortedList = productList.stream()  
 .sorted(Comparator.*comparingDouble*(Product::getProductPrice ).reversed())  
 .collect(Collectors.*toList*());

**filtering for rating> 4:**

**productRepo**.findAll().stream()  
 .filter(product -> product.getProductRating() >= 4 &&  
 product.getProductCategory().getProductCategoryName().toLowerCase()  
 .equals(productCategoryName.toLowerCase())  
 ).collect(Collectors.*toList*());

**Copy of a list – clone:**

ArrayList cloned = new ArrayList(collection c);

@OneToMAny

@JSonIgnore

This resolves recursion

Math –min function

Math – max function

function differenceInAges(ages){

minAge = Math.min.apply(Math, ages);

maxAge = Math.max.apply(Math, ages);

diffAge = maxAge - minAge;

return [minAge,maxAge,diffAge];

}

const nums = [1, 2, 3]  
Math.min(nums) // NaN  
Math.max(nums) // Nan

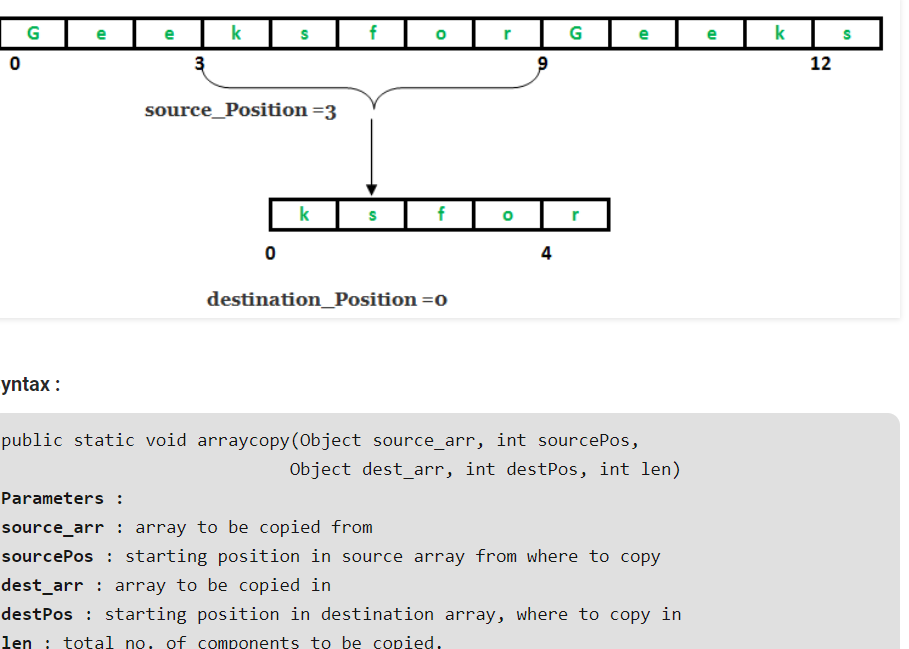
That is because Math.min or Math.max functions expect distinct variables and not an array.

var nums = [1, 2, 3]  
Math.min.apply(Math, nums) // 1  
Math.max.apply(Math, nums) // 3  
Math.min.apply(null, nums) // 1  
Math.max.apply(null, nums) // 3

assignment it becomes easier

*The****destructuring assignment****syntax is a JavaScript expression that makes it possible to extract data from arrays or objects into distinct variables.*

const nums = [1, 2, 3]  
Math.min(...nums) // 1  
Math.max(...nums) // 3



**Array reduce:**

The **reduce()** method executes a **reducer** function (that you provide) on each element of the array, resulting in a single output value.

*arr.reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])*

**callback**

A function to execute on each element in the array (except for the first, if no initialValueis supplied), taking four arguments:

**accumulator**

The accumulator accumulates the callback's return values. It is the accumulated value previously returned in the last invocation of the callback, or initialValue, if supplied (see below).

**currentValue**

The current element being processed in the array.

**index**Optional

The index of the current element being processed in the array. Starts from index 0 if an initialValue is provided. Otherwise, starts from index 1.

**array**Optional

The array reduce() was called upon.

**initialValue**Optional

A value to use as the first argument to the first call of the callback. If no initialValue is supplied, the first element in the array will be used and skipped. Calling reduce() on an empty array without an initialValue will throw a TypeError.

numArr = [2,4,6,8,10];

function addArray(total, num){

return total + num;

}

const i = numArr.reduce(addArray);

console.log("Value of i: "+ i);

/\* \*/

objArr = [

{

price: 2.0,

num: 10

},

{

price: 3.0,

num: 20

}

]

const totalObjArr = objArr.reduce(calcTotal, 0);

function calcTotal(total, arrMember){

return total + (arrMember.price\* arrMember.num);

}

console.log("Total after reduce: "+ totalObjArr);

calculateTotal() {

this.total = this.itemsInCart.reduce((total, currVal) => total + (currVal.price \* currVal.quantity),0)

}

**String to other data types:**

