**Modern Java: The Big Picture**

**Java Platform:**

* Programming language
* Runtime Environment (JRE): used to execute the java code. End users normally require only this.
* Standard Library: contains many commonly used functionalities so that we don’t have to write a lot of components from scratch. (no re-inventing the wheel)
* These three parts of Java are bundled together in a Java Development Kit (JDK)
* Any time a new release comes out, any part of the above can be improved
* To run the java application, we need a Java Virtual Machine (JVM) which establish JRE
* JVM basically knows how to execute the java byte code on the actual machine

When we compile Java code using ‘javac’ command it creates .class file which is basically byte code representation of java file.

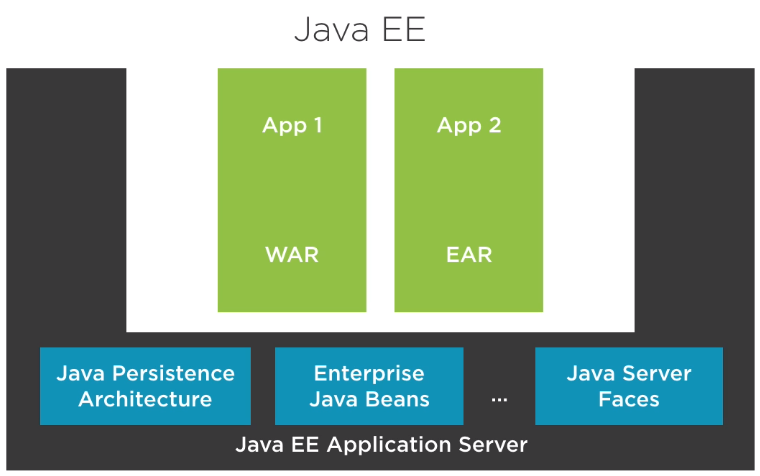
**Java EE:**

Java EE provides us with a lot of APIs to perform several complex tasks for enterprise application. For example:

* Java persistence Architecture (JPA): used for storing and retrieving data from relational databases.
* Enterprise Java Beans: used to create transactional business logic
* Java Server Faces: used to create web front ends

Normally Java applications run on top of the JVM. But in case of Java EE the Java EE Application Server is the application that runs on top of JVM.

The Java applications that you develop are deployed to the Java EE application server packaged as a WAR (web archive) or EAR (Enterprise Archive (slightly complex functionalities)) file. We can deploy multiple applications on this server.



Some other application servers for Java EE:

* WildFly (RedHat)
* WebSphere (IBM)
* WebLogic (Oracle)
* Tomcat (Apache)

Microframeworks in Java:

* Spring boot:

Built on top of spring framework. Most well-knowns libraries that spring uses is Netflix open source tech libraries.

* MicroProfile
* Vert.x
* Play Framework

Alternatives JVM Languages:

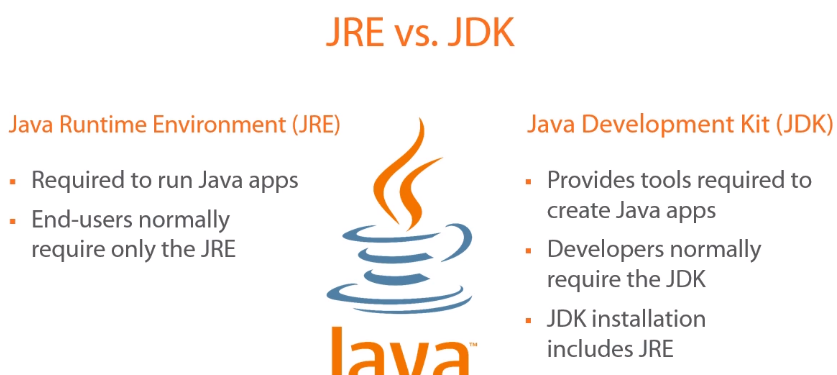
The following languages compiles to the java byte code therefore they can easily be run on JVM.

* Scala
* Kotlin

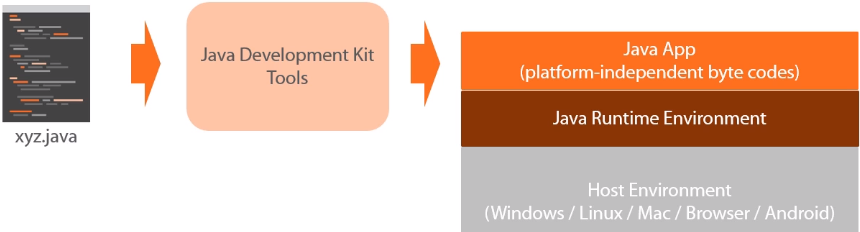
**Java Fundamentals: The Java Language**

**JRE vs JDK:**

Quick comparison:



You write the Java code and feeds it to the JDK and it produces a Java app which is basically platform-independent byte code. This code can be run on windows, linux, mac, android. It just requires the JRE.



**Primitive data types:**

* Integer (Byte, Short, Int, Long)
* Floating Point (Float, Double): allows to store values containing fractional portion
* Character: stores a single Unicode character
* Boolean

**Concept of Encapsulation:**

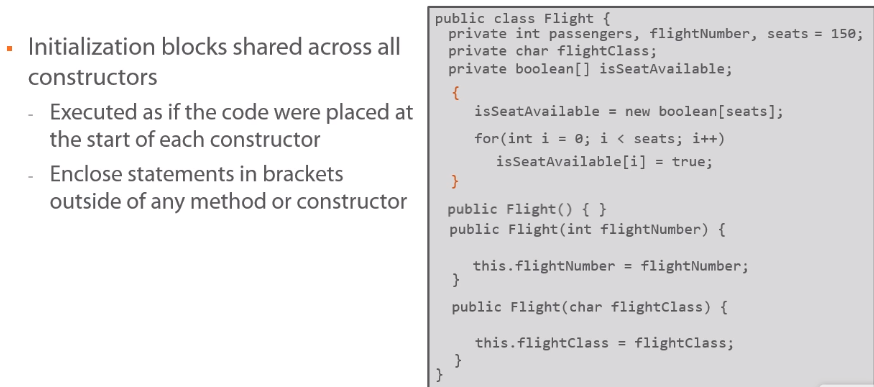
* Idea of hiding the internal representation of a class is known as encapsulation
* The way we achieve it is by using access modifiers

**Accessors/Mutators Pattern:**

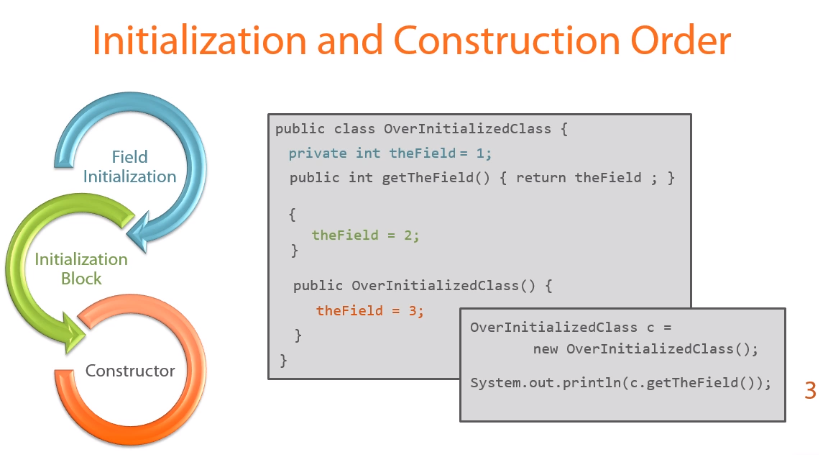
* They are used to control access to the fields (private fields) inside the class.
* Accessors also known as getters. Mutators as setters.

**Setting initial state:**

* Can be done in three ways:
* Field initializer: just initialize a field with in bare class.
* Constructor: you know it
* Initialization blocks: if you want a piece of code to run whenever any constructor from ur class is used, you can use initialization block. See example below:

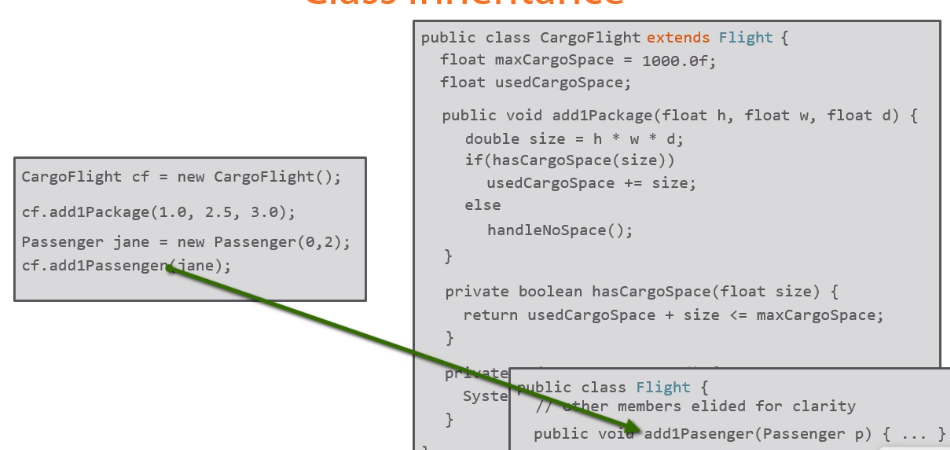


**Initialization and Construction order:**

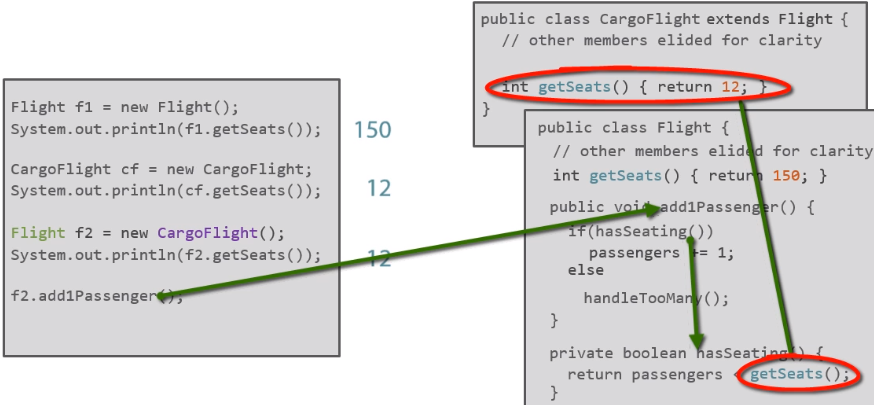


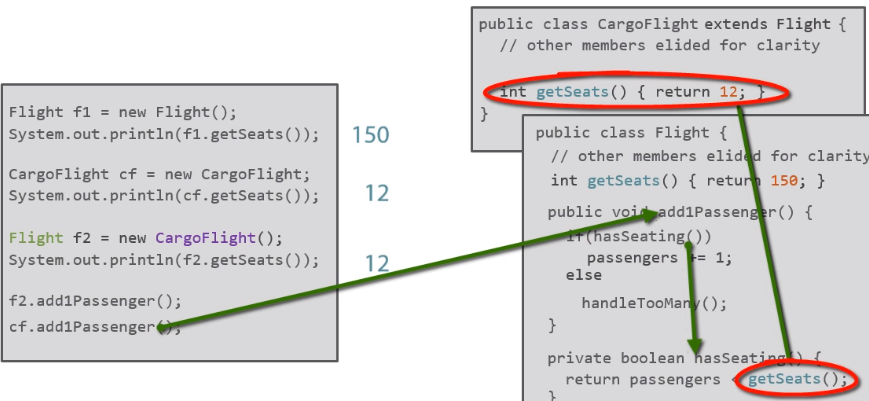
**Class Inheritance:**

* Extending a class from a base class.
* Inheritance helps us to use existing characteristics of a class while building new characteristic of a new class.
* By default all classes can be extended and derived classes have the option to use inherited methods or override them



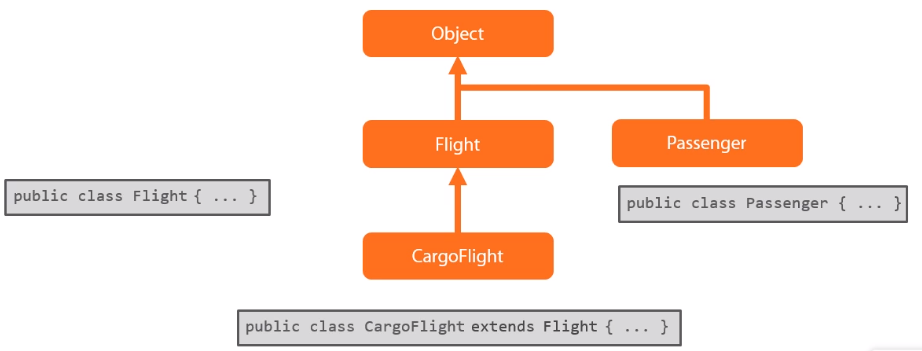
* In case of creating an object of derived class using reference of base class (ex: Flight f2 = new CargoFlight()) the derived class methods will override the base class methods.
* @Override annotation in derived class before a method in derived class will make sure 1) that the base class includes a method with the same signature 2) base class method is overridden by this method
* In below example we can see that java calls the method based on the type of object created instead of the reference used.



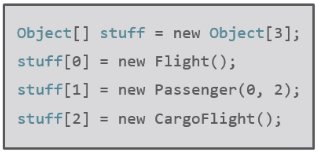


**Object Class:**

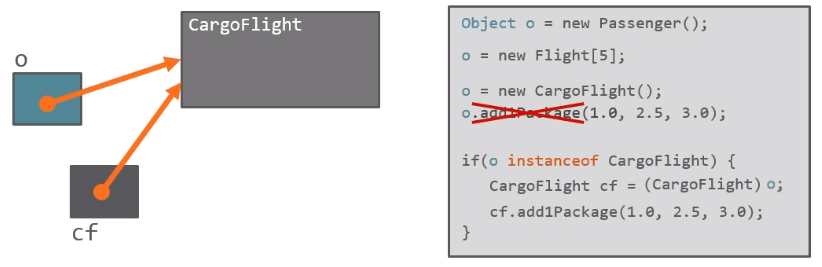
* Every class in Java extends from Object class directly or indirectly
* There is a hidden ‘extends Object’ keyword after name of the class by default if a class does not extends from anything which means it extends from Object class



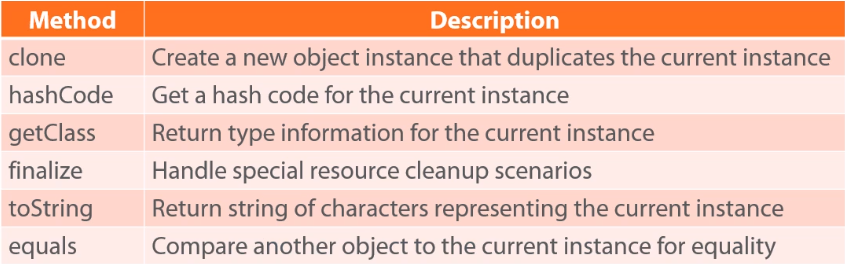
* That means we can create an object of any class using Object class reference



To access methods of this class we would have to create a reference of specific class as follows

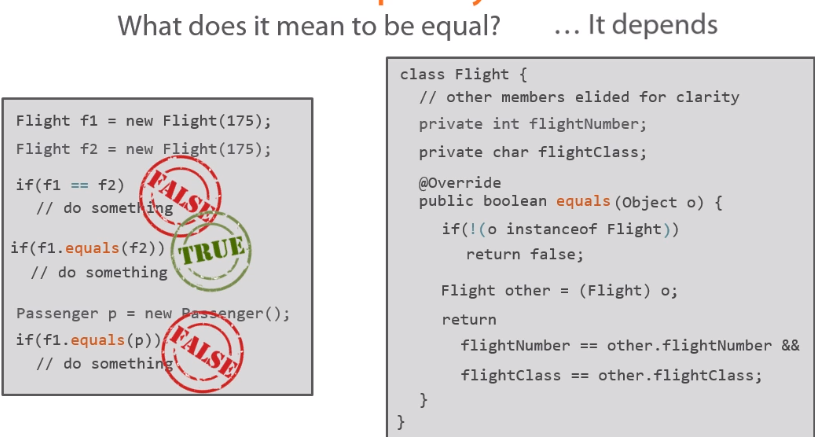


* Methods of the object class:



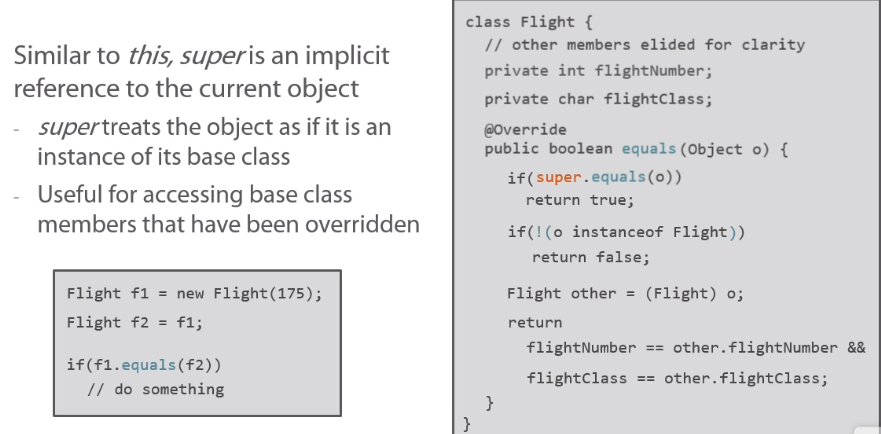
**Equality of Object References:**

* By default, object references will be equal only if they point to the same object
* Otherwise we have to define what equals mean for them by overriding the equals() method of object class as follows:



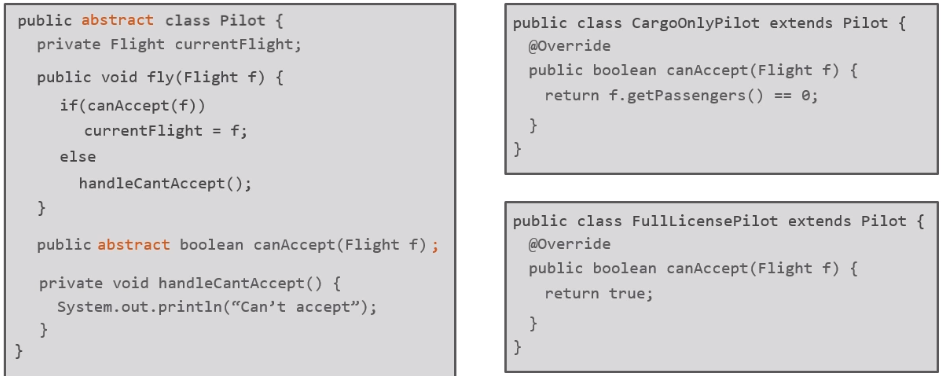
**Special Reference: Super:**

* If we implement @Override annotation to override a method of the base class but for some reason we need to specifically call the base class method despite of overriding that method we use *super* reference.
* With *super* we have the ability to access the base class behavior when we want to while still having our own behavior
* Example: we check if the references are pointing to the same objects then no need to check further equality, just return True



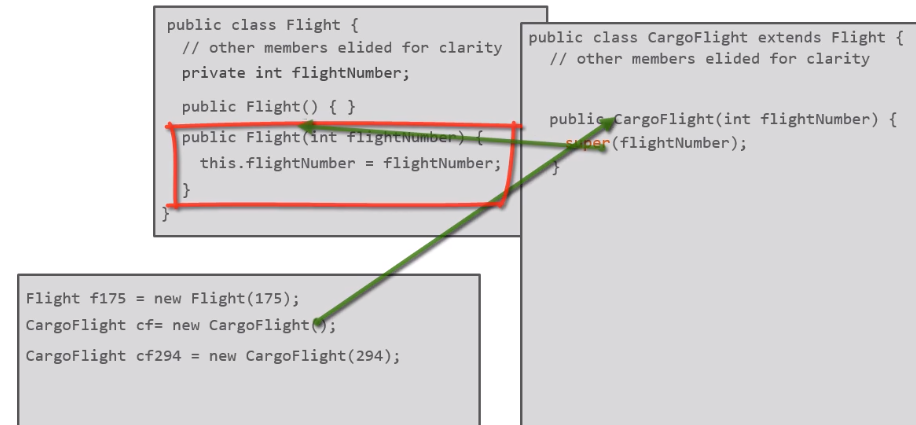
**Using Final and Abstract (class and method level):**

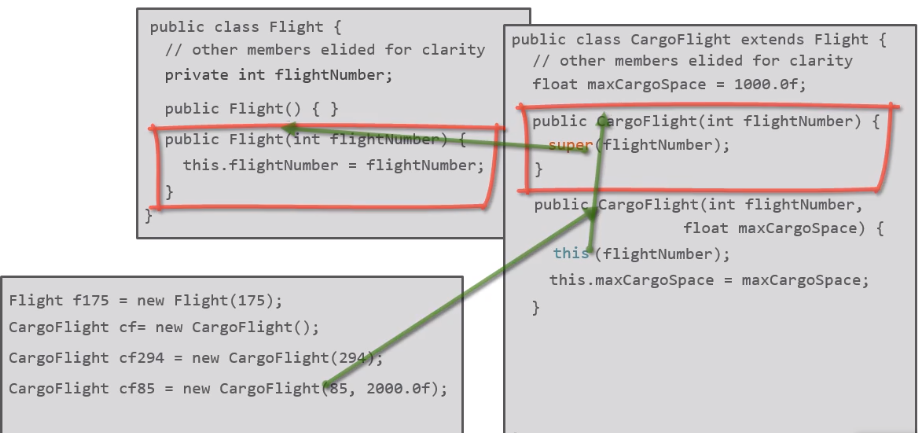
* If we don’t want any class to be extended, we can use final keyword in front of class name.
* If we want to allow to inherit from class but don’t want specific methods to be changed (block overriding), we can just put final keyword in front of method names.
* As we saw final blocks inheritance, on the other hand **abstract** requires inheritance
* For details of abstract go to Udemy java notes. Here is an example:



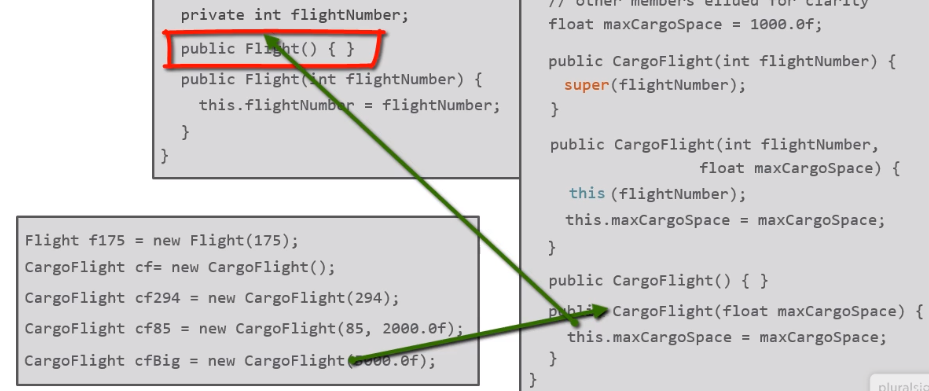
**Constructors calling patterns in Inheritance:**

* Examples below:





* Java will always call our base class constructor even if we don’t specify one and then run our constructor:



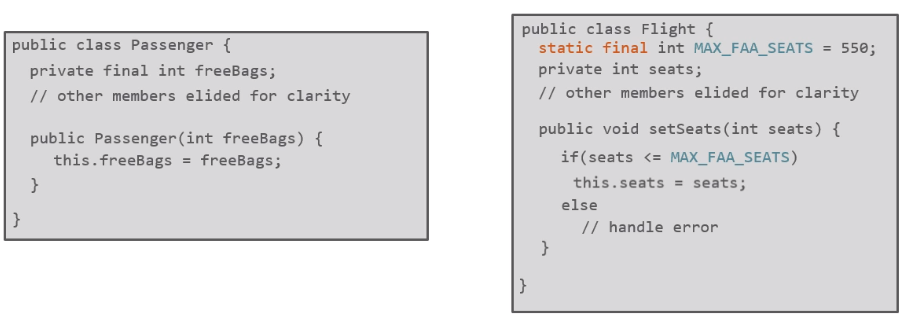
**String Builder:**

* String builder is an efficient way to build and manipulate strings
* Since strings are immutable (means when a string object is created it can’t be changed, if we change it, it will create a new string object and point to that) its better to use string builder

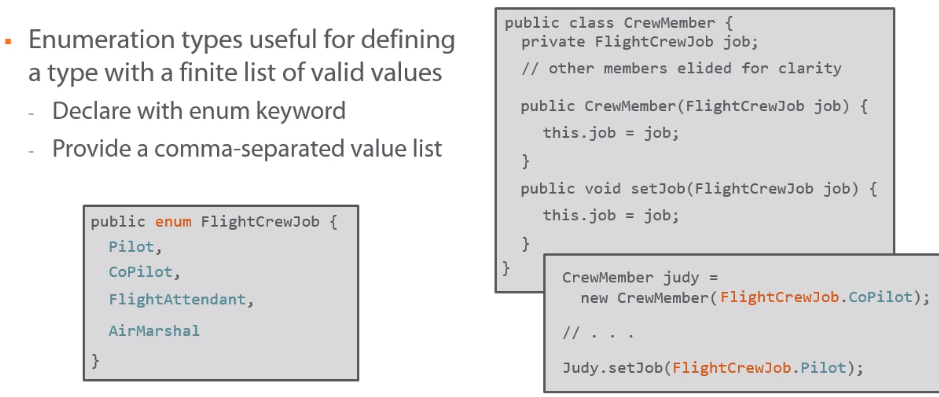
**Final variables:**

* A variable can be set as final if we don’t want it to be changed once it initialized. For example it can be initialized with the constructor and can’t be changed going forward.
* A variable can be set as *static final* if we want to assign a specific hardcoded value and don’t want anyone to change to change it. For example, FAA has a rule that number of maximum seats in a flight can be 550, we can make it a *static final* and no one can change it while it can be used to check number of seats in a flight
* Convention: When we use *static final,* we use ALL\_CAPS.





**Enumeration types:**



**Exception and methods**

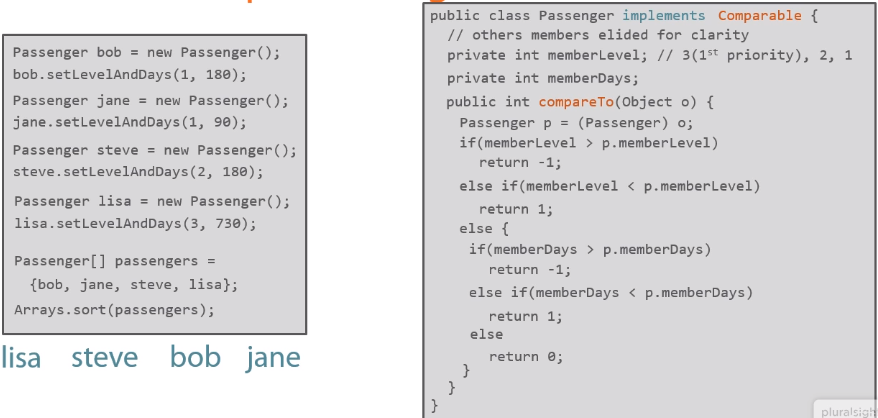
* Methods are responsible for any checked exceptions that may occur in them
* Either they have to catch that exception or they have to document that exception using *throws* clause

**Interface:**

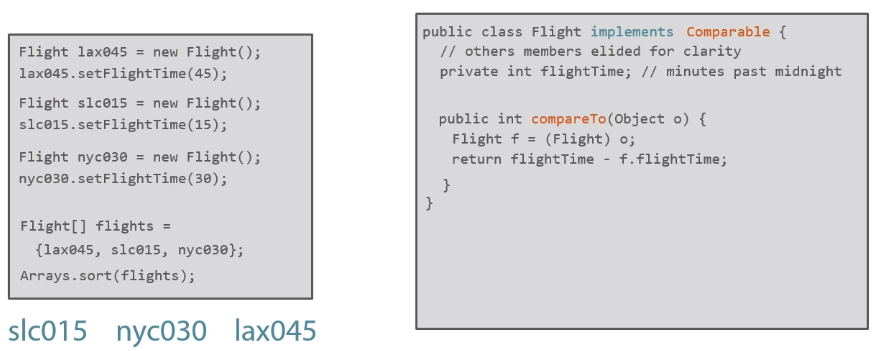
* A type that defines the contract. They don’t provide any implementation but just the contract
* Classes can implement an interface which indicates that class conforms to the contract
* A class can implement as many interfaces as it needs. (in contrast: it can only extend only one class)
* Example of an interface: java.lang.Comparable (used for determining relative order)
* Contract: It has a method called compareTo. Returns one of three values
  + negative value: should come before
  + positive value: should come after
  + zero value: equal

Returned value from this method indicates current instance relative sequence.

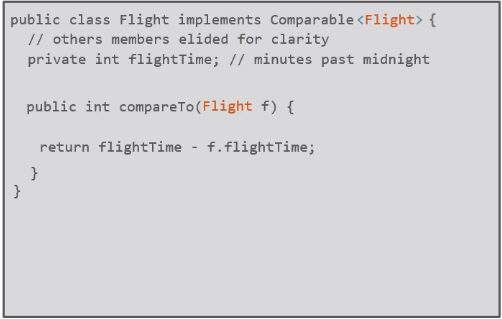
* So, if we provide implementation to this method in our class, we can make use of it efficiently where ever we need a comparison in objects of our class.
* In the following example we used it to sort our objects on basis of a comparison criteria:



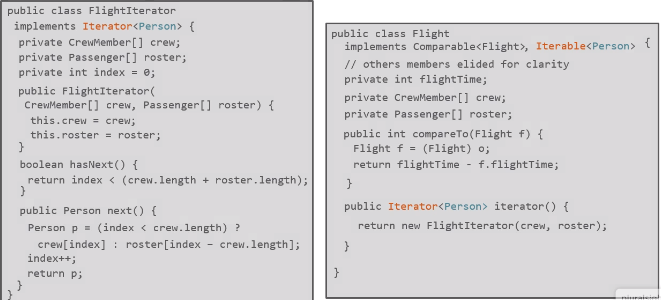
* Another example:



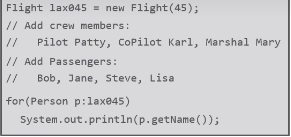
* We can get rid of casting the object as Flight or Passenger type by using typed interface:



* Implementing multiple interfaces in the following example (Comparable and Iterable):



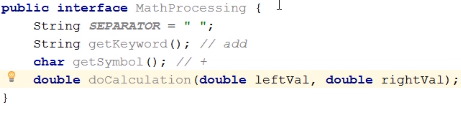
* Usage of above example: (foreach statement under the hood interacts with the interface we wrote above and prints out all of the crew members and passenger names)



**Use of Interface:** ability to take a problem, abstractily model it to its individual pieces, model the contract and fulfill those pieces as an interface and then you can very easily build the implementations of this interface. With very little incremental work each time we can leverage the power of interfaces to create new implementations focused on very specific set of work

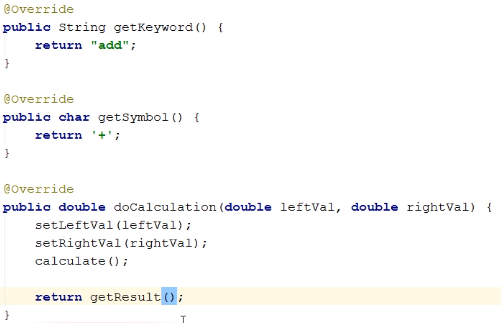
**Declaring an interface:**

* Declaration of interface is similar to class.
* We provide all the constant and method names but don’t provide any implementations
* All the constants are considered as public, private and static
* All methods treated as public

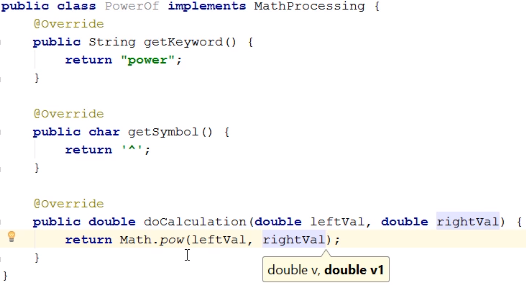


* Implementation in the class:



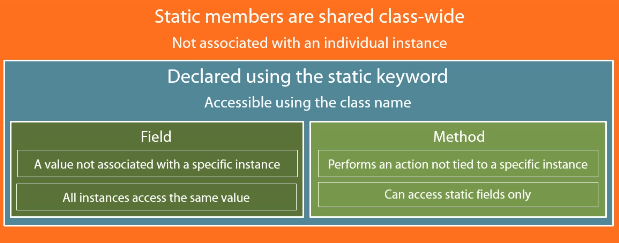


* Another implementation of same interface:



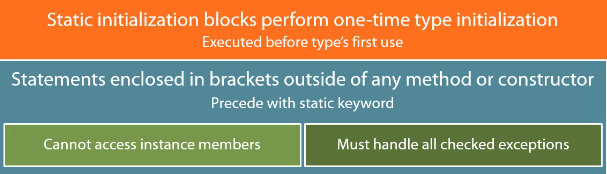
**Static Members:**

* Summary:

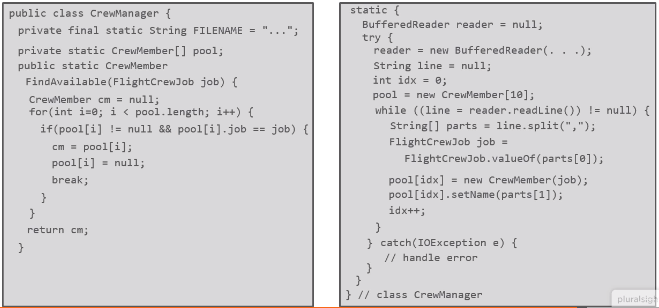


* Static initialization blocks: They get executed automatically before the class’s first use. *They are used when we need to perform some type of work to prepare a class before we use it.*

**Remember:** initialization blocks automatically executed before a constructor while static initialization blocks before you even start using the type



* Example of static initialization block: lets suppose your class need some info that is sitting in a text file to perform its important functionalities. It would be very good if we load the file in a static initialization block so that it gets loaded and ready to serve data prior to our first use. User doesn’t have to make any special call to initialize it.



**Streams:**

* Ordered sequence of data
* Streams are unidirectional. Either you can read from it or write to it
* Two categories of streams: byte streams and text streams