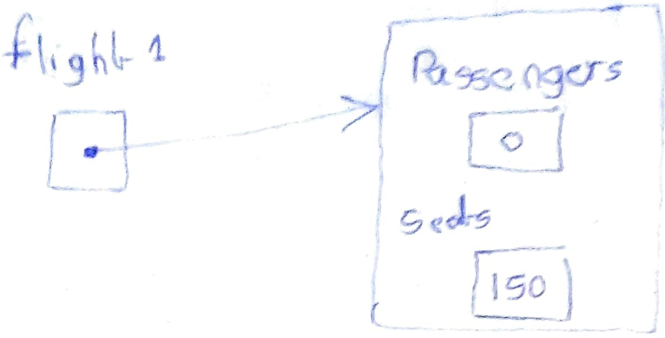
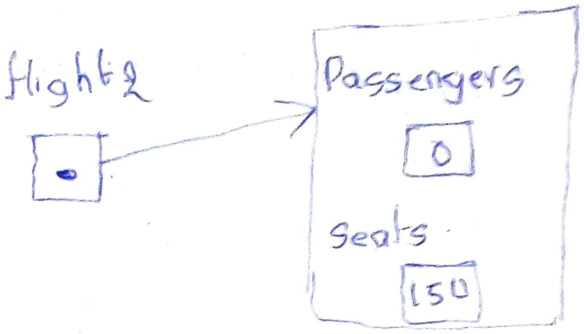
Classes are Reference Types :-

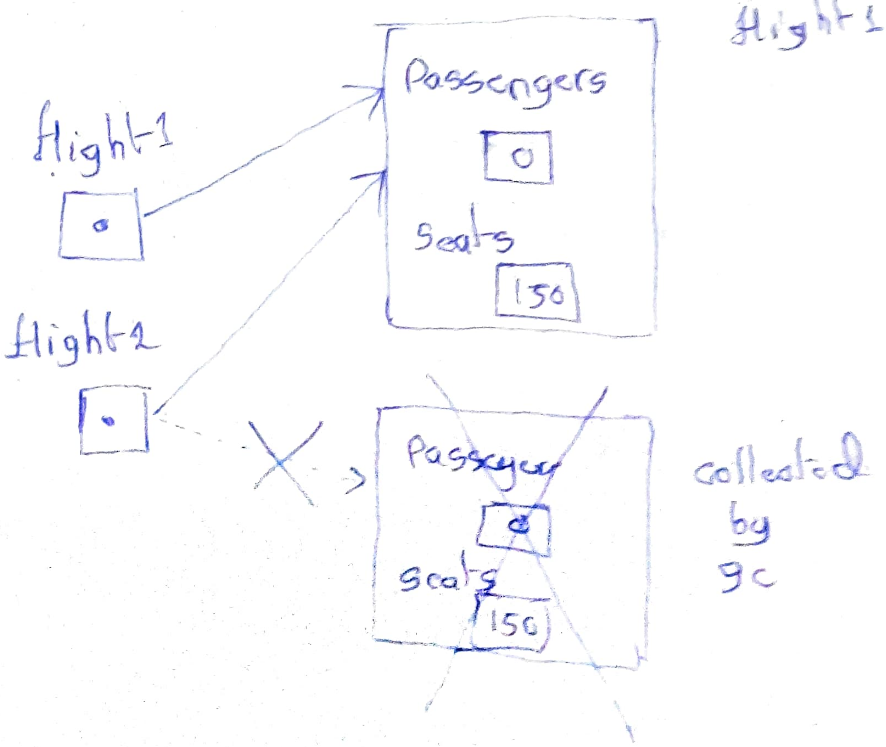
Flight flight1 = new Flight();



Fligh flight2 = new Flight();



flight2 = flight1; // here we are changing reference of flight2 to flight1



Encapsulation and Access modifiers

→ Java uses access modifiers to achieve encapsulation

Naming classes :- "Pascal case"

Accessors and Mutators

→ Use the A/M ~~[illegible]~~ patten to control field access

→ Accessor retrives field value
Also called getter (get method)

→ Mutator modifies field value
Also called setter (set method)

Class Initializers and constructors :-

chaining constructors ::

One constructor can call another constructor

→ Use this keyword followed by parameter list

→ this (parameter list) must be first line

We can use access modifiers to control
constructors visibility

→ Limits what code can perform specific
creations.

Encapsulation and Access modifiers

→ Java uses access modifiers to achieve encapsulation

Naming classes: - "Pascal Case"

Accessors and Mutators

→ Use the A/M ~~[illegible]~~ pattern to control field access

→ Accessor retrives field value
Also calld getter (get method)

→ Mutator modifies field value
Also called setter (set method)

Class Initializers and constructors :-

chaining constructors :

One constructor can call another constructor

→ Use this keyword followed by parameter list

→ this (parameter list) must be first line

We can use access modifiers to control
constructors visibility

→ Limits what code can perform specific
creations.

Initialization Blocks :-

```
public class Flight {
    private int passengers, flightNumber, seats = 150;
    private char flightClass ;
    private boolean[] isSeatAvailable;

    {
        isSeatAvailable = new boolean [seats];
            for(int i = 0; i < seats ; i++)
                isSeatAvailable [i] = true;
    }

    public Flight () { }
                      → The above block code will be executed automatically

    public Flight (int flightNumber) {
→  Here the block code will be executed automatically
        this.flightNumber = flightNumber ;

    public Flight (char flight-class)
        {
    →   Here also the block code will be executed
        }
```

boolean default value is "false"

→ Initialization block shared and executed as if the code were placed at the start of each constructor parameters immutability :- Changes made to passed values are not visible outside of method.

overloading :

Each constructor and method must have unique Method signature

$\boxed{\text{Name}}$ $\boxed{\text{Number of parameters}}$ $\boxed{\text{Type of each parameter}}$

Variable number of parameters :-

A method can be declared to accept a varying numbers of parameter values

```
public void addPassenger(Passenger... list)
{
  ==
}
```

it takes any number of arguments from 0 to ....

→ place an ellipse (...) after parameter type.

Object Class : builtin class

The object class is the root of the java class hierarchy

→ Every class has the characteristics of of object class.

Every class inherits Object class either directly or indirectly.

→ Object type reference can hold any hold any type of object.

Equality :-

equals() method in object class works as normal "==" which checks ~~ref~~ whether both the references referencing same object or not.

→ NOTE: it is important to override equals() method in the class with which you are comparing

f1.equals(f2);

↙

f1 object class should override equals method and write code to compare the object data in it.

Special Reference: super

→ similar to this, super is an implicit reference to the current object

→ super treats the object as if it is an instance of its base class

→ Useful for accessing base class members that have been overriden

→ constructors are not inherited

A base class constructor must always be called.

By default, base class' no-argument constructor is called.

can explicitly call a base class constructor using super followed by parameter list.

→ must be first line of constructor.

@override → optional annotation used for compiler will check ~~overiding~~ overriding method has correct syntax or not

String class :-

Strings are immutable → any changes made in string creats new string object

The string class stores a sequence of Unicode characters.

stored using UTF-16 encoding

String Equality:-

Converting non-string types to strings :-

```
int ival = 100;
string sval = string.valueOf(iVal);
```

String Builder

→ StringBuilder provides mutable string buffer

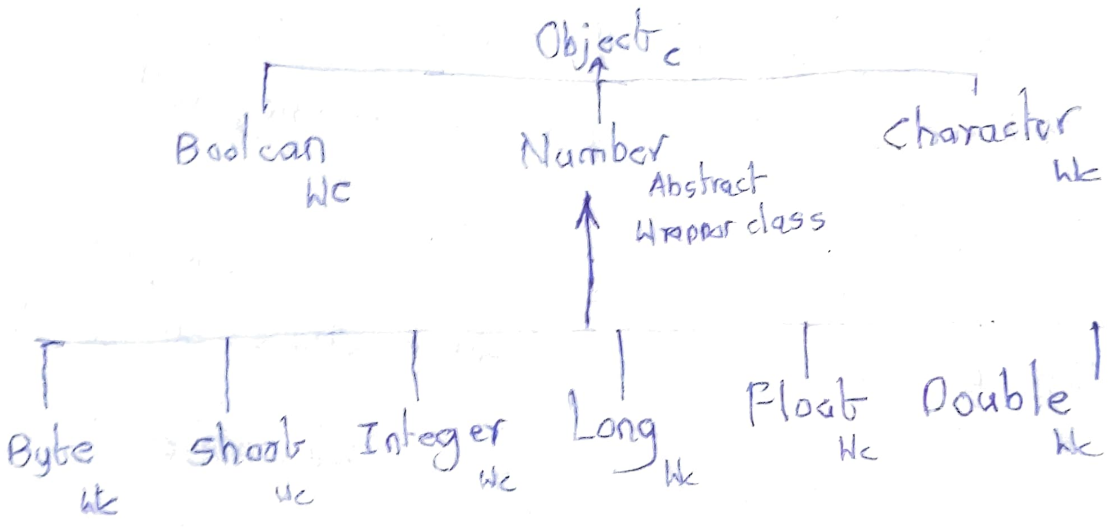
→ for best performance pre-size buffer

→ will grow automatically if needed

→ Most common method : append , insert

→ Use "toString" to extract result string .

primitive Wrapper classes :-



→ All wrapper class instances are immutable

```
Integer a = 100;   Auto-boxing → compiler will do implicitly
  int b = a;       Auto-Unboxing
Integer c = b;
Integer d = Integer.valueOf(100);  Boxing ← programmer do Explicitly
int e = d.intValue();  UnBoxing
Integer f = Integer.valueOf(e);
```

```
string s = "57.44";
double s1 = Double.parseDouble(s);
Double s2 = Double.valueOf(s);
```

Wrapper Class Equality :-

```
Integer i1000A = 10 * 10 * 10;

Integer i1000B = 100 * 10;

if( i1000A == i1000B) false
if ( i1000A.equals(i1000B)) True

Integer i8A = 4 * 2;
Integer i8B = 2 * 2 * 2;

if( i8A == i8B) true
```

Here Boxing conversions that always return the "same wrapper class instance" for range given mentioned

primitive Type	values
int	-128 to 127
short	-128 to 127
byte	-128 to 127
chare	'\u0000' to '\u00ff'
boolean	true, false

Static Initialization Blocks:-

static initialization blocks perform one-time type initialization.

→ Executed before type's first use

```
static
{
  ==
}
```

} → Outside of any method or constructor

→ can't access instance members
must handle all checked exceptions

Nested Types:

A nested type is a type declared within another type.

1) classes can be declared within classes and interfaces

2) Interfaces can be declared within cla and intr

Nested types are members of the enclosing type
→ Private members of the enclosing type are visible to the nested type

Nested types serve differing purposes
structure & scoping

No relationship b/w instances of nested and enclosing type.

→ static classes nested within interfaces
→ All classes nested within interfaces
→ All nested interfaces

Inner classes :-

Each instance of nested class is associated with in an instance of enclosing class

Non-static nested within class.

Anonymus Classes :-

These are inner classes

Anonymus instances is associated with the containing class instance.

→ Create as if you are constructing an instance of the interface or base class