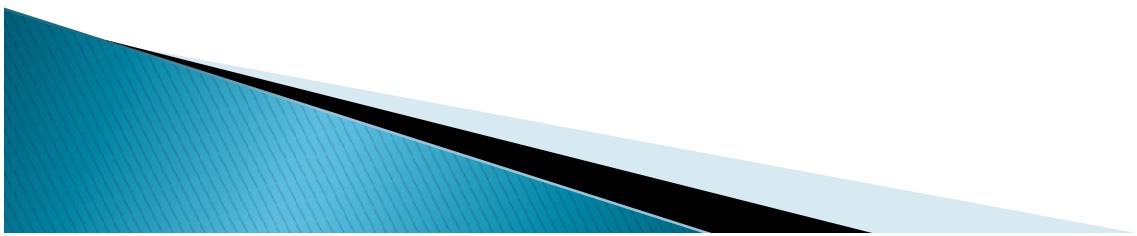
Java Constructor and Class Initializers

By
Sunil Kumar(Master of Sc.)
Bangalore, India

1. Agenda

- Establishing Initial State
- Field Initializers
- Constructors
- Constructor Chaining and visibility
- Initialization blocks
- Initialization and Construction Order



Establishing Initial State

- In last slide we have studied that class is made up of STATE and EXECUTABLE code. When an object is created, it is expected to be in a useful state. Java provide default state which is useful. But often the default state established by Java is not enough. The object may need to set values or execute code.

Mechanism for establishing Initial State

- Java provide 3 mechanism for establishing initial state.
  - Field Initializers
  - Constructors
  - Initialization Blocks

Field Initial State

- In case of variable we explicitly assigned value. Initializing variable and initializing field is 2 different thing. A field's initial state is established as part of object construction. Fields receive a zero value by default. Different data type fields will have different values.

Byte **Short** **Int** **Long**	**Float** **Double**	**Char**	**Boolean**	**Reference Type**
0	0.0	'\u0000'	False	Null

Field Initial State Contd..

- Sometimes this default initial state is not acceptable as part of requirement.
- Java provide field Initializers which allows programmer to specify a field's initial value as part of declaration.
  - Can be a simple assignment
  - Can be an equation
  - Can reference other fields
  - Can be a method call

Field Initial State Contd..

- We can see through an example.

```
public class Earth{
 long circumferenceInMiles = 24901;
 //long circumferenceInKilometers = (long)24901 * 1.6d;
 long circumferenceInKilometers = (long)(circumferenceInMiles * 1.6d);
 long circumferenceInKilometers =Math.round(circumferenceInMiles* 1.6d);
}
```

- So we have seen that these filed Initializers are very powerful where we can assign using simple assignment, a method call or a field reference.

Constructor

- A constructor in Java is a block of code similar to a method that's called when an instance of an object is created. A constructor doesn't have a return type. The name of the constructor must be the same as the name of the class. We can say, it is a special type of method that is used to initialize the object. Remember that a constructor is not a method but a executable code used during object creation to set initial state. Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Constructor Contd..

- Thus a constructor is a executable code where we must have to name it same as CLASS.
  - Have no return type
  - Every class has at least one constructor
- From our last example flight class

```
public class Flight{
private int passenger;
private int seats;
    public Flight(){
            seats = 150;
            passenger = 0;
    }
}
```

Constructor Contd..

- In the last example, declaration of passenger =0 is not required as by default field value is initialized to 0. Also seats can be declared in the above during declaration. So we don’t really require a constructor here.
- What happened when we don’t really need an explicit constructor.
- We will try to understand this using an example.

Constructor Contd..

```
public class Passenger{
      private int checkedBags;
      private int freeBags;
   //accessors and mutators elided for clarity
      private double perBagFee;
      public Passenger(){
      }
      public Passenger (int freeBags){
        this.freeBags = freeBags;
      }
}
```

```
Passenger bob = new Passenger();
bob.setCheckedBags(3);

Passenger jane = new Passenger(2);
jane.setCheckedBags(3);
```

Constructor Contd..

- In case of requirement like this, where we don't require an explicit constructor
  - Java provides a default constructor with no argument
- A class can have a multiple constructors
  - Each with a different parameter list
- Once explicit constructor is declared, we have to explicitly declared the definition of default constructor as well, else program will have an error.

Chaining Constructor

- One Constructor can call another
  - Use the this keyword followed by parameter list
- Constructor Visibility
- Use access modifier to control constructor visibility
  - Limit what code can perform specific creations

Constructor Visibility

- Chaining of Constructor and Constructor Visibility Example

```
Public class Passenger
        //fields and methods elided for clarity
        Public Passenger(){
}
Public Passenger(int freeBags){
//this(freeBags > 1 ? 25.0d : 35.0d);
this.freeBags = freeBags;
}

public Passenger(int freeBags, int checkedBags){
this(freeBags);
this.checkedBags = checkedBags;
}

public Passenger(double perBagFee){
this.perBagFee = perBagFee;
}
}
```

Chaining Constructor Contd..

- Executing Program outside this above class

```
Passenger nyasha = new Passenger(2);
Passenger sunil = new Passenger(2,3);
~~Passenger James = new Passenger(2.0d);~~
```

Initialization Block

- Constructor is not the only way to run code as part of setting the initial state of the class object. We can do it by initialization block as well.
- Initialization blocks shared across all constructors
  - Executed as if the code were placed at the start of each constructor
  - Enclose statements in brackets outside of any method or constructor.
- Let’s take an example

Initialization Block Contd..

```
public class Flight{
private int passengers, flightNumber, seats = 100;
private char flightClass;
private boolean[] isSeatAvailable;

public Flight(){
isSeatAvailable = new boolean[seats];
for(int i = 0; i< seats; i++)
        isSeatAvailable[i] = true;
}
public Flight(int flightNumber){
        this();
        this.flightNumber = flightNumber;
}
public Flight(char flightClass){
        this();
        this.flightClass = flightClass;
}
}
```

Initialization Block Contd..

```
public class Flight{
private int passengers, flightNumber, seats = 100;
private char flightClass;
private boolean[] isSeatAvailable;

{
isSeatAvailable = new boolean[seats];
for(int i = 0; i< seats; i++)
            isSeatAvailable[i] = true;
}
public Flight(){}
public Flight(int flightNumber){
//              this();
            this.flightNumber = flightNumber;
}
public Flight(char flightClass){
  //            this();
            this.flightClass = flightClass;
}
}
```

Initialization Constructor Order

- The order of preference is as follows (from lower to higher)
  - Field Initialization
  - Initialization Block
  - Constructor
- Lets take an example

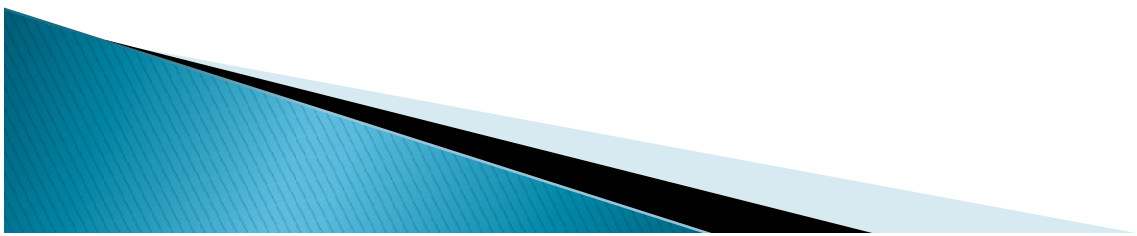
```
public class OverInitializeClass{
private int theVal =1;
public int getTheVal(){
        return theVal;
    }
{
    theVal = 2;
}
pulic OverInitializeClass(){
    theVal = 3;
    }
}
```

Summary

- Object should be created in some useful state.
- Field Initializers provide an initial value as part of the declaration.
- Every class has at least one constructor
  - If no explicit constructor, java provides one with no argument.
  - We can provide multiple constructors with different argument lists.
- One constructor can call another
  - Call must be first line
- Initialization blocks share code across constructors.
- Keep the initialization and constructor order in mind.

2. Agenda

- Parameter Immutability
- Constructor & Method overloading
- Variable number and parameters
- Inheritance Basics
- Member hiding and overriding
- The Object Class
- Objet equality
- The Super keyword
- Final and abstract
- Inheritance and constructors



Parameter Immutability

- Parameters are passed by making a copy of the value.
  - This is known as passing by-value.
  - Changes made to passed value are not visible outside of method.
  - Changes made to members of passed class instances are visible outside of method
- We will take an example to understand the above concept.

Parameter Immutability

- **Primitive Type**

```
int val1 = 10;
int val2 = 20;
//print val1 & val2          --val1 → 10 val2→20
swap(val1, val2);
//print val1 & val2          --val1 → 10 val2→20

void swap(int i, int j){
Int k = i;
i = j;
j= k;
//print i & j  ----vali → 20 valj→10
}
```

Parameter Immutability

- Class

```
public class Flight{
        int flightNumber;
         //access and mutator elided for clarity
         public Flight(int flightNumber){
         this.flightNumber = flightNumber;
        }
}

Flight val1 = new Flight(10);
Flight val2 = new Flight(20);
//print val1 & val2  flight#    --val1 → 10 val2→20
swap(val1, val2);
//print val1 & val2  flight#    --val1 → 10 val2→20


void swap(Flight i, Flight j){
Flight k = i;
i =j;
J = k;
//print i & j  flight#              -- i→ 20  j→10
}
```

Parameter Immutability

```
public class Flight{
        int flightNumber;
        //access and mutator elided for clarity
        public Flight(int flightNumber){
       this.flightNumber = flightNumber;
       }
}

Flight val1 = new Flight(10);
Flight val2 = new Flight(20);
//print val1 & val2  flight#                --val1 → 10 val2→20
swap(val1, val2);
//print val1 & val2  flight#                --val1 → 20 val2→10


void swapNumber(Flight i, Flight j){
int k = i.getFlightNumber();
i.setFlightNumber(j.getFlightNumber());
j.setFlightNumber(k);
//print i & j  flight#          -- i→ 20  j→10
}
```

Overloading

- A class may have multiple versions of its constructor or methods called as overloading.
- Each constructor and method must have a unique signature.
  - Signature is made up of 3 parts.
    - Number of Parameters
    - Type of each Parameters
    - Name (In case of constructor, name should always match with Class so not much significant here. But in case of method it can be anything).
- We will see some of the example for the above concept.

Overloading

```
public class Flight{
int seats = 150, passengers;
int totalCheckedBags;
int maxCarryOns = seats * 2, totalCarryOns;
public void add1Passenger(){
        if(hasSeating()) //if(passengers < seats)
        passengers += 1;
else
        handleTooMany();
}
private boolean hasSeating(){
        return passengers < seats;
        }
private boolean hasCarryOnSpace(int carryOns){
        return totalCarryOns + carryOns <= maxCarryOns;
}
}
```

Overloading

```
public class Flight{
//other members elided for clarity
public void add1Passenger(){
If(hasSeating())
passenger += 1;
else
handleTooMany();
}
public void add1Passenger(int bags){
If(hasSeating()){
add1Passeneger();
totalCheckedBags += bags;
}}
public void add1Passenger(Passenger p){
add1Passenger(p.getCheckedBags());
}
public void add1Passenger(int bags, int carryOns){
if(hasSeating() && hasCarryOnSpace(carryOns)){
add1Passenger(bags());
totalCarryOns += carryOns;
}}
public void add1Passenger(Passenger p, int carryOns){
add1Passenger(p.getCheckedBags(), carryOns);
}}
```

Overloading- A look

- Now just concentrating on the signature of the method we have below.

```
public class Flight{
public void add1Passenger(int bags){…}

public void add1Passenger(Passenger P){…}

public void add1Passenger(int bags, int carryOns){..}

public void add1Passenger(Passenger P, int carryOns){..}

public void add1Passenger(){…}

}
```

```
Flight f = new Flight();
passenger p1 = new Passenger(0,1);
passenger p2 = new Passenger(0,2);

f.add1Passenger();
f.add1Passenger(2);
f.add1Passenger(p1);

short threeBags = 3;
f.add1Passenger(threeBags, 2);
f.add1Passenger(p2, 1);
```

Overloading

- Some points to be noted during method overloading.
  - Unlike constructor overloading, we have to use special keyword but there is no such rules with method overloading. We can simply call the method name.
  - While chaining of constructor, it should be the 1st line of the constructor definition. In case of method overloading, it can be written anywhere.

Variable No. of Parameters

```
public class Flight{
//other members elided for clarity
public void addPassenger(Passenger[] list){
If(hasSeating(list.length)){
passengers += list.length;
for(Passenger passenger : list)
totalCheckedBags += passenger.getCheckedBags();
}
else handleTooMany();
}
private boolean hasSeating(int count){
return passengers + count <= seats;
}
}
```

Variable No. of Parameters

```
Flight f = new Flight();
Passenger nyasha = new Passenger(0,1);
Passenger sunil = new Passenger(0,2);
f.addPassengers(new Passenger[] {nyasha, sunil});

/*If we want to add a family of 3, here we need to declare and then
write for each passenger. It would be nice if could have simply list each
of the passenger we want to add. We can do this by declaring list
which we will see in next example*/
Passenger john = new Passenger(0,2);
Passenger jack = new Passenger(0,2);
Passenger brad = new Passenger(0,2);
f.addPassengers(new Passenger[] {john, jack, brad});
```

Variable No. of Parameters

```
public class Flight{
//other members elided for clarity
public void addPassenger(Passenger… list){
If(hasSeating(list.length)){
passengers += list.length;
for(Passenger passenger : list)
totalCheckedBags += passenger.getCheckedBags();
}
else handleTooMany();
}
private boolean hasSeating(int count){
return passengers + count <= seats;
}}
```

```
Flight f = new Flight();
Passenger nyasha = new Passenger(0,1);
Passenger sunil = new Passenger(0,2);
/*So here we can simply list the passenger
to add instead of declaring an array variable
1st and then listing it as we did in the
previous example.*/
f.addPassengers(nyasha, sunil});

Passenger john = new Passenger(0,2);
Passenger jack = new Passenger(0,2);
Passenger brad = new Passenger(0,2);
f.addPassengers({john, jack, brad});
```

Variable No. of Parameters

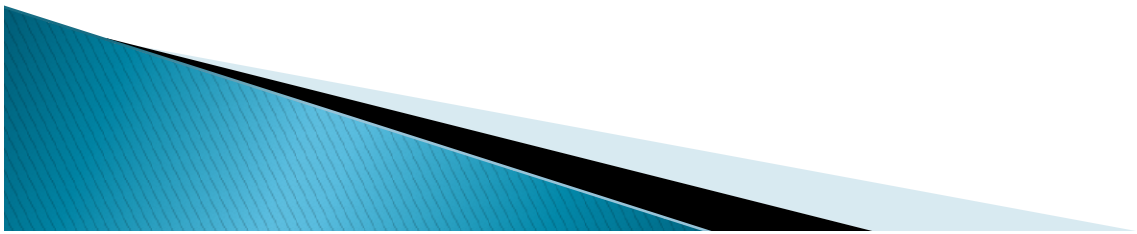
- A method can be declared to accept a varying number of parameter values
  - Place an ellipse after parameter type.
  - Can only be the last parameter.
  - Method receives values as an array.
- Point to be noted here.
  - The old definition of code will still work as ellipse notation is meant for receiving an array.
  - Ellipse notation will just simplify the calling of method where no need to declare an array and we can simply list the value. In this case, Java will by default create an array for us.

Summary

- Parameters are immutable.
  - Changes made to passed values are not visible outside of method.
  - Changes made to members of passed class instances are visible outside of method.
- A class may have multiple versions of its constructor or methods.
  - Each must have a unique signature
  - Signature includes name, parameters, type of each parameter.
- A method ca be declared to accept varying number of parameter values
  - Values received as an array
  - Only restriction is that must be last parameter received by that method or constructor.

3. Agenda

- Inheritance Basics
- Member hiding and overriding
- The Object Class
- Objet equality
- The Super keyword
- Final and abstract
- Inheritance and constructors



Class Inheritance

- A class can be declared to inherit from another class.
  - Use the extends keyword
- Derived class has characteristics of base class.
  - Can add specialization
  - Can be assigned to base class type references
  - If a derived class adds a field that has the same name as the fields in the base class, it will actually hide the base class fields.
  - Methods override base class methods with same signature

Class Inheritance

```
public class CargoFlight extends Flight{
float maxCargoSpace = 1000.0f;
float usedCargoSpace;
public void add1Package(float h, float w, float d){
double size = h*w*d;
If(hasCargoSpace(size))
      usedCargoSpace += size;
else
      handleNoSpace();
}
private boolean hasCargoSpace(float size){
return usedCargoSpace + size <= maxCargoSpace;
}
private void handleNoSpace(){
System.out.println("Not enough space");
}
}
```

```
CargoFlight cf = new CargoFlight();
cf.add1Package(1.0, 2.5, 3.0);
Passenger nyasha = new Passenger(0, 2);
Passenger sunil = new Passenger(1);
cf.add1Passenger(nyasha);
cf.add1Passenger(sunil);
```

Thus we can say, derived class has characteristics of base class.

Class Inheritance

```
public class CargoFlight extends Flight{
float maxCargoSpace = 1000.0f;
float usedCargoSpace;
public void add1Package(float h, float w, float d){
double size = h*w*d;
If(hasCargoSpace(size))
      usedCargoSpace += size;
else
      handleNoSpace();
}
private boolean hasCargoSpace(float size){
return usedCargoSpace + size <= maxCargoSpace;
}
private void handleNoSpace(){
System.out.println("Not enough space");
}
}
```

```
Flight f = new CargoFlight();
Passenger nyasha = new Passenger(0,2);
Passeneger sunil = new Passenger(1);
f.add1Passenger(nyasha);
f.add1Passenger(sunil);
~~f.add1Package(1.0, 2.5, 3.0);~~
```

```
Flight [] squadron = new Flight[5];
squadron[0] = new Flight();
squadron[1] = new CargoFlight();
squadron[2] = new CargoFlight();
squadron[3] = new Flight();
squadron[4] = new CargoFlight();
```

Thus we can say, Can be assigned to base class type references

Member Hiding and Overriding

```
public class Flight{
//other members elided for clarity
int seats = 150;
public void add1Passenger(){
If(hasSeating())
                passengers += 1;
else
                handleTooMany();
}
private boolean hasSeating(){
                return passengers < seats;
}
}
```

```
public class CargoFlight extends Flight{
    //other members elided for clarity
    int seats = 12;
}
```

```
Flight f1 = new Flight();
System.out.println(f1.seats);          /It will print 150

CargoFlight cf = new CargoFlight;
System.out.println(cf.seats);          //It will print 12

Flight f2 = new CargoFlight();
System.out.println(f2.seats);          //It will print 150

f2.add1Passenger();
/*This method will check the availability based on
Seats = 150  This is because when a method is
called it basically checks for within the method
definition what is the number of seats defined*/

Cf.add1Passenger();
/*This method will check the availability based on
Seats = 150 */
```

Member Hiding and Overriding

- Thus through the example we have seen that idea of hiding fields is very dangerous as we have to take care of the method definition where exactly it is defined.
- Method overriding is little different compare to field overriding.
  - Method can override base class methods if it has exactly the same signature as of base class.
  - In Java all methods are automatically overridden unless we prevent it by using special keyword.
  - What if the signature become different by mistake. In that case we mention @override before the method definition in derived class. It will not have any impact on runtime but will have impact only during compile time.

Member Hiding and Overriding

```
public class Flight{
//other members elided for clarity
int getSeats() {return 150} ;
public void add1Passenger(){
If(hasSeating())
                passengers += 1;
else
                handleTooMany();
}
private boolean hasSeating(){
                return passengers < getSeats();
}
}
```

```
public class CargoFlight extends Flight{
    //other members elided for clarity
    @override
    int getSeats() {return 12};
}
```

```
Flight f1 = new Flight();
System.out.println(f1.getSeats());   /It will print 150

CargoFlight cf = new CargoFlight;
System.out.println(cf.getSeats());    //It will print 12

Flight f2 = new CargoFlight();
System.out.println(f2.getSeats);   //It will print 12

f2.add1Passenger();
/*This method will check the availability based on
Seats = 12  This is because method of base class is
overridden by derived class  assuring that the
appropriate method implementation gets called
based on the type of object created, not the type
of reference used.*/

Cf.add1Passenger();
/*This method will check the availability based on
Seats = 12 */
```

Object Class

- The object class is the root of the Java hierarchy
  - Every class has the characteristics of the object class.
  - Useful for declaring variables, fields and parameters that can reference any class or array instance.
  - Define a number of methods that are inherited by all objects.

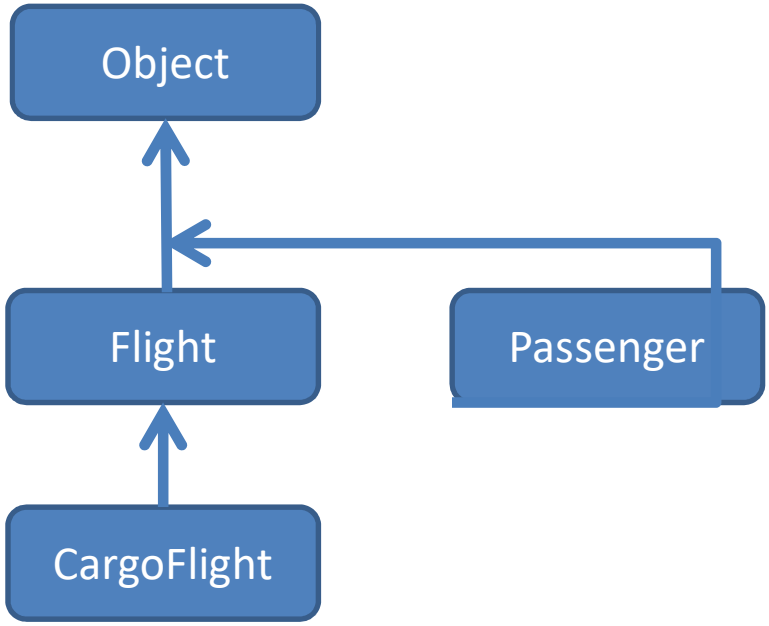
Inheriting from Object

- Every class inherits directly or indirectly from object class.

public class Flight *extends Object*{….}

public class CargoFlight extends Flight{…}

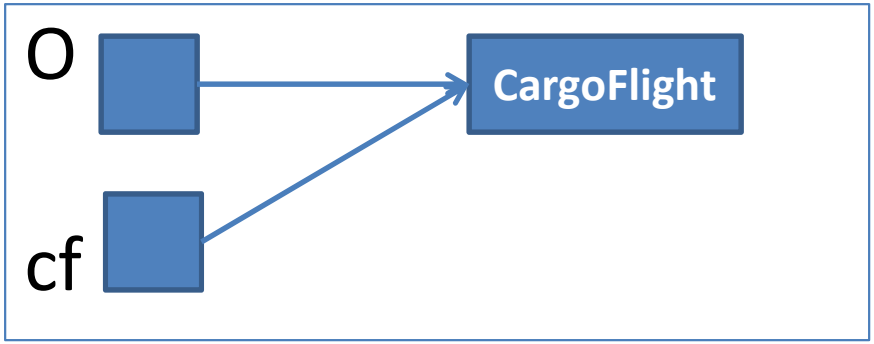
public class Passenger {…}



Inheriting from Object

- Thus we can say that a class is related to other class or not, it always extends Object class.

```
Object[] stuff = new Object[3];
stuff[0] = new Flight();
stuff[1] = new Passenger(0, 2);
stuff[2] = new CargoFlight();
```




```
Object o = new Passenger();
o = new Flight[5];
o = new CargoFlight();

o.add1Package(1.0, 2.5, 3.0);

If(o instanceof CargoFlight){
        CargoFlight cf = (CargoFlight) o;
        cf.add1Package(1.0, 2.5, 3.0);
}
```

Inheriting from Object

- Some points to be noted during inheriting from object class.
  - We can only access the capabilities that are visible to that reference.
  - So in this example, if we try to take o and call add1Package, a method that is specific to CargoFlight, that is not going to work, because the reference o doesn't know anything about this method add1Package.
  - When we directly assign this o to CargoFlight, it will give a compiler error as o can point to different many type. To make the compiler understand, we have to explicitly tell the compiler by adding type casting.

Object Class Methods

Method	Description
clone	Create a new object instance that duplicates the current instance
hashCode	Get a hash code of current instance
getClass	Return type information for the current instance
finalize	Handle special resource cleanup scenarios
toString	Return string of characters representing the current instance
equals	Compare another object to the current instance for equality

Equality

- What does it mean to be equal ? …It depends
- By default definition, it means that if both references point to the exact same object instance then only it is equal

```
Flight f1 = new Flight(175);
Flight f2 = new Flight(175);
if(f1 == f2)                    //always false
// some stmnt
if(f1.equals(f2))
//always false as per default definition of object class
Passenger p = new Passenger();
If(f1.equals(p))
//some stmnt
```

But we can override the equal method provided by object class as per our requirement. Here flight properties are defined by flight number and flight class. If both are equal then we can say it is equal.

```
Class Flight{
private int  flightNumber;
private int  flightNumber;
Private char flightClass;
@override
public boolean equals(Object  o){
if(!(o instanceof Flight))
   return false;
   Flight other = (Flight) o;

flightNumber == other.flightNumber && flightClass
== other.flightClass;
}
}
```

Special Reference: Super

- Similar to this and null references, 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 overridden.

Special Reference: super

```
Flight f1 = new Flight(175);
Flight f2 = f1
Flight f2 = new Flight(175);
if(f1.equals(f2))
//some stmnt
```

```
class Flight{
private int  flightNumber;
private int  flightNumber;
Private char flightClass;
@override
public boolean equals(Object  o){
 if(super.equals(o))
    return true;
if(!(o instanceof Flight))
    return false;
    Flight other = (Flight) o;

flightNumber == other.flightNumber
&& flightClass == other.flightClass;
}
}
```

Controlling Inheritance & Overriding

- By default all classes can be extended and derived classes have the option to use or override inherited methods.
  - A class can change these defaults
  - Use final to prevent inheriting and/or overriding
  - We can use final keyword for a class which means that whole class and its method can’t be overridden.
  - We can use final keyword for a method only, which means that particular method can be overridden else can be.

Using Final

```
public final class Passenger{
  //….
}
```

Or

```
final public class Passenger{
  //….
}
```

```
public class CargoFlight extends Flight{
float maxCargoSpace = 1000.0f;
float usedCargoSpace;
public final void add1Package(float h, float w, float d){
double size = h*w*d;
If(hasCargoSpace(size))
      usedCargoSpace += size;
else
      handleNoSpace();
}
private boolean hasCargoSpace(float size){
return usedCargoSpace + size <= maxCargoSpace;
}
private void handleNoSpace(){
System.out.println(“Not enough space”);
}
}
```

Controlling Inheritance & Overriding

- There may be some situation where instead of preventing for inheritance, we may require inheritance. That's where use of abstract comes in.
- Use abstract to require inheriting and/or overriding.
- Abstract allows us to require that a class be inherited, and require that a particular method be overridden.

Using Abstract

```
public class Pilot{
    private Flight currentFlight;
    public void fly(Flight f){
        if(canAccept(f))
            currentFlight = f;
        else
            handleCantAccept();
}
    private void handleCantAccpet(){
        System.out.println("Can't accpet");
    public abstract boolean canAccept(Flight f);
}
}
```

```
public class CargoOnlyPilot extends Pilot{
    @override
    public boolean canAccept(Flight f){
        return f.getPassenger() == 0;
    }
}
```

```
public class anyFlightPilot extends Pilot{
    @override
    public boolean canAccept(Flight f){
        return true;
    }
}
```

Inheritance and Constructors

- 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 1st line of constructor.

Inheritance and Constructors

```
public class Flight{
//other members elided for clarity
 private int flightNumber;
public Flight() {   }
public Flight(int flightNumber){
       this.flightNumber = flightNumber;
  }
}
```

```
Flight f175 = new Flight(175);
CargoFlight cf= new CargoFlight();
CargoFlight cf294 = new   CargoFlight(294);
CargoFlight cf85 = new CargoFlight(85, 2000.0f);
CargoFlight cfBig = new CargoFlight(5000.0f);
```

```
public class CargoFlight extends Flight{
//other members elided for clarity
 public CargoFlight(int flightNumber){
       super(flightNumber);
 }
 public CargoFligt(int flightNumber, float
       maxCargoSpace){
       this(flightNumber);
       this.maxCargoSpace = maxCargoSpace;
 }
  public CargoFlight() {  }
  public CargoFlight(float maxCargoSpace){
       this.maxCargoSpace = maxCargoSpace;
  }
}
```

Summary

- Inheritance allow a new class to be defined with the characteristics of another
  - Use the extend class
- Derived class can override base class methods.
  - Optionally use @override annotation
- All classes derive from Object class either directly or indirectly.
- By default, object references are only equal when referencing the same instance.
  - Can override Object.equal to provide new behavior.
- Super accesses current object as if instance of base class.
- final and abstract provide control over class inehritance and method overriding.
- Constructors are not inherited