

# Java Generics.

Generics allows us to take classes , interfaces, methods that takes types as parameters.

```
public class Main {  
    public static void main(String[] args) {  
        ArrayList items=new ArrayList();  
        items.add(1);  
        items.add(2);  
        items.add(3);  
        items.add(4);  
        items.add(5);  
  
        printDoubled(items);  
    }  
    public static void printDoubled(ArrayList n){  
        for(Object i :n){  
            System.out.println((Integer) i*2);  
        }  
    }  
}
```

The problem above there is no type checking, which

can be inconvenient.

When we actually provide type parameter to a generic type, it is called as parameterized type.

```
ArrayList<Integer> items=new ArrayList<Integer>();  
items.add(1);  
items.add(2);  
items.add(3);
```

```
public class Team {  
    private String name;  
    int played=0;  
    int won=0;  
    int lost=0;  
    int tied=0;  
    private ArrayList<Player> members=new ArrayList<>();  
    public Team(String name){  
        this.name=name;  
    }  
    public String getName(){  
        return this.name;  
    }  
    public boolean addPlayer(Player player){
```

```

public static void main(String[] args) {
    FootballPlayer joe= new FootballPlayer( name: "JOE");
    BaseballPlayer pat= new BaseballPlayer( name: "Pat");
    SoccerPlayer beckham=new SoccerPlayer( name: "Bechkam");

    Team adelaideCrows = new Team( name: "Adelaide Crows");
    adelaideCrows.addPlayer(joe);
    adelaideCrows.addPlayer(pat);
    adelaideCrows.addPlayer(beckham);
    System.out.print(adelaideCrows.numPlayers());
}

```

Above the problem we want only to add football players to a football team.

So we would create 3 classes, Football team, Baseball team, Soccer team

But in that case the code will be largely identical.

We can obviously extend Team class, but in that we need to have functionality unique to each team type in the code as well.

So common method playGame, recordScore etc would be in the base team class.

Fortunately java got generics and it enables us to specify a type when we are creating our class.

```

public class Team<T> {
    private String name;
    int played=0;
    int won=0;
    int lost=0;
    int tied=0;
    private ArrayList<T> members=new ArrayList<>();
    public Team(String name){
        this.name=name;
    }
    public String getName(){
        return this.name;
    }
    public boolean addPlayer(T player){
        if(members.contains(player)){
            System.out.print("\n"+player.getName()+" is
            return false;
        }
    }
}

```

We are changing the type, so its going to work for any type of player. So when we instantiating a class, the **T** will be replaced with the class we passed.

Two ways to solve issues:

```
public boolean addPlayer(T player){
    if(members.contains(player)){
        System.out.print("\n"+((Player) player).getName()+"
        return false;
    }
    else{
        members.add(player);
        Player x= (Player) player;
        String name=x.getName();
        System.out.print("\n"+name+" picked for team "+this
        return true;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        FootballPlayer joe= new FootballPlayer( name: "JOE");
        BaseballPlayer pat= new BaseballPlayer( name: "Pat");
        SoccerPlayer beckham=new SoccerPlayer( name: "Bechkam");

        Team<FootballPlayer> adelaideCrows = new Team<FootballPlayer>( name: "Adelaide Crows");
        adelaideCrows.addPlayer(joe);
        adelaideCrows.addPlayer(pat);
        adelaideCrows.addPlayer(beckham);
        System.out.print(adelaideCrows.numPlayers());
    }
}
```

```
11      adelaideCrows.addPlayer(pat);
```

|   |                |
|---|----------------|
| Required type:  | FootballPlayer |
| Provided:   | BaseballPlayer |
| Change 1st parameter of method 'addPlayer' from 'T' to 'BaseballPlayer' Alt+Shift+E |                |

We can also restrict more: this is called as bounded type parameters

```
public class Team<T extends Player> {  
    private String name;  
    int played=0;  
    int won=0;  
    int lost=0;  
    int tied=0;  
    private ArrayList<T> members=new ArrayList<>();  
    public Team(String name){  
        this.name=name;  
    }  
}
```

An argument pass as type parameter can either be a class or interface.

Interface can themselves specify type parameters.

We use a Single bound when we specify Team type parameter, restricting teams to be created for objects that'll inherit from player class or subclass of player.

Java does allow multiple bounds, so u can have class T extends Player and coach and manager.

For e.g to allow multiple instances of player,coach or manager to be added to the team.

But with that in mind, the normal inheritance implements rules still apply.

But u can extends from only single class and multiple instances.

If u specify multiple bounds then the class must be listed first otherwise Java's gonna raise an error when you try to actually compile it.

IOW class first and then interfaces.

```
public class Team<T extends Player & Coach & Manager> {
```

2.

```
public class Team<T extends Player> implements Comparable<Team<T>>
```

```
    @Override  
    public int compareTo(Team<T> o) {  
        return 0;  
    }  
}
```

If u got a list of objects that implements comparable such as ArrayList of Team what you can do is you can quickly sort the list using the static sort method of the collection class.

```
ArrayList<Team> teams;  
Collections.sort(teams);
```

The whole thing is gonna be sorted using the compareTo() func that we wrote for the Team class.

E.g

```
public void showLeagueTable(){  
    Collections.sort(league);  
    for(T t:league){  
        System.out.println(t.getName()+" : "+t.ranking());  
    }  
}
```

### Generic Challenge.

```
public static void main(String[] args) {  
    League<Team<FootballPlayer>> footballLeague=new League<>( name: "FIFA");  
    Team<FootballPlayer> adelaideCrows = new Team<>( name: "Adelaide Crows");  
    Team<FootballPlayer> melbourne= new Team<>( name: "Melbourne");  
  
    footballLeague.add(adelaideCrows);  
    footballLeague.add(melbourne);  
}
```

You can also

```
public class League <U extends Player,T extends Team> {
```

```
    League< FootballPlayer,Team<FootballPlayer>> footballLeague=new League<>( name: "FIFA");
```

# Naming conventions, packages, scope, static and final keywords.

## Packages

The things you will name in Java are:

- Packages
- Classes
- Interfaces
- Methods
- Constants
- Variables
- Type Parameters

## Packages

- Always lower case.
- Package names should be unique
- Use your internet domain name, reversed, as a prefix for the package name.
- Oracle specify a convention for package names at <http://docs.oracle.com/javase/specs/jls/se6/html/packages.html#7>.

## Invalid domain name components

- Replace invalid characters (i.e. -) in domain name with an underscore
- Domain name components starting with a number should instead start with an underscore\_
- Domain name components that are Java keywords should have that component start with an underscore.

## Invalid domain name components

Examples with replacements

- Switch.supplier.com → com.supplier.\_switch
- 1world.com → com.\_1world
- Experts-exchange.com → com.experts\_exchange

## Example package names

---

- `java.lang`
- `java.io`
- `org.xml.sax.helpers`
- `com.timbuchalka.autoboxing`

## Class names

---

- CamelCase
- Class names should be nouns (they represent things).
- Should start with a capital letter
- Each word in the name should also start with a capital (e.g. `LinkedList`)

## Interface names

---

- Capitalized like class names (CamelCase).
- Consider what objects implementing the interface will become of what they will be able to do

Examples:

- `List`
- `Comparable`
- `Serializeable`

## Method names

---

- mixedCase.
- Often verbs.
- Reflect the function performed or the result returned.

Examples:

- `size()`
- `getName()`
- `addPlayer()`

## Constats

---

- ALL UPPER\_CASE.
- Separate words with underscore \_.
- Declared using the final keyword

Examples:

- Static final int MAX\_INT
- Static final short SEVERITY\_ERROR
- Static final double P1 = 3.141592653

## Variable names

---

- mixedCase
- Meaningful and indicative.
- Start with lower case letter.
- Do not use underscores \_.

Examples:

- i
- league
- SydneySwans
- BoxLength

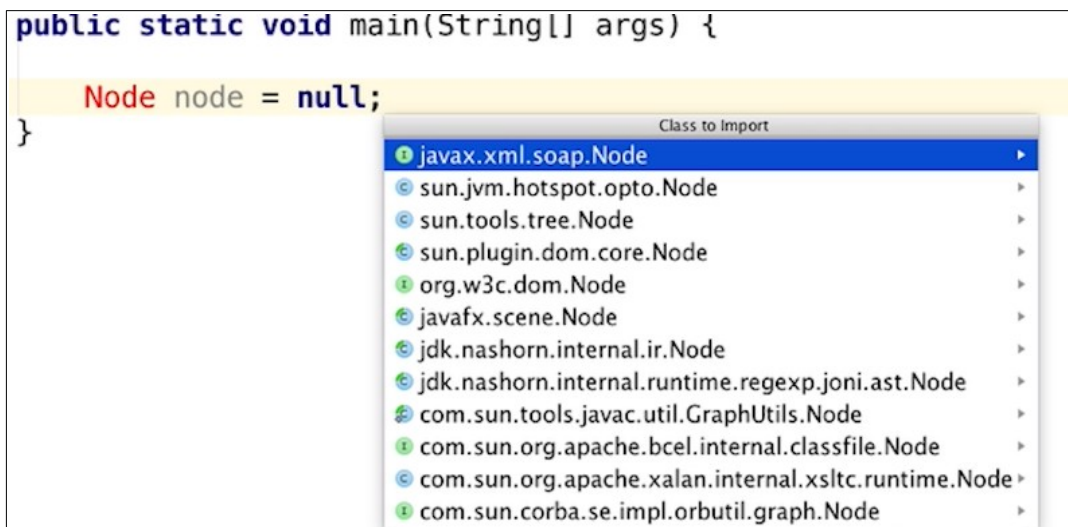
## Type Parameters Examples

---

- E – Element (used extensively by the Java Collections Framework)
- K – Key
- T – Type
- V – Value
- S, U, V, etc. – 2<sup>nd</sup>, 3<sup>rd</sup>, 4<sup>th</sup> types



# Packages



Package is a way of grouping related classes and interfaces together, so the actual mechanism provides a way to manage the name space of object types and it also extends access protection beyond the traditional access, modifier you see before public, protected, private, etc.

Another way of doing

```
public static void main(String[] args) {  
    javafx.scene.Node node = null;  
}
```

If you using 2 different Node class, then one of them has to be use like Above.

```
import javafx.scene.Node;  
  
public class Main {  
    public static void main(String[] args) {  
        Node node = null;  
        org.w3c.dom.Node anotherNode = null;  
    }  
}
```

If u use full Name: it would be clear which one you meant.

Why you wanna use Packages:

- 1) To easily determine Classes are related to one another.
- 2) Easy to find classes and interfaces that can provide the functions provided by the package.

- 3) Because the package creates a new namespace, class and interface name conflicts are avoided.
- 4) Classes within the package can have unrestricted access to one another while still restricting access for classes outside the package.
- 5) Major doubt solved

```
import java.awt.*;  
import java.awt.event.WindowAdapter;  
import java.awt.event.WindowEvent;
```

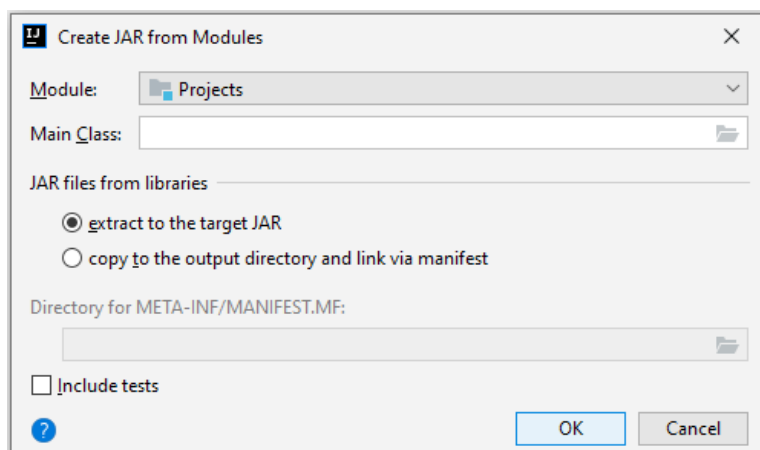
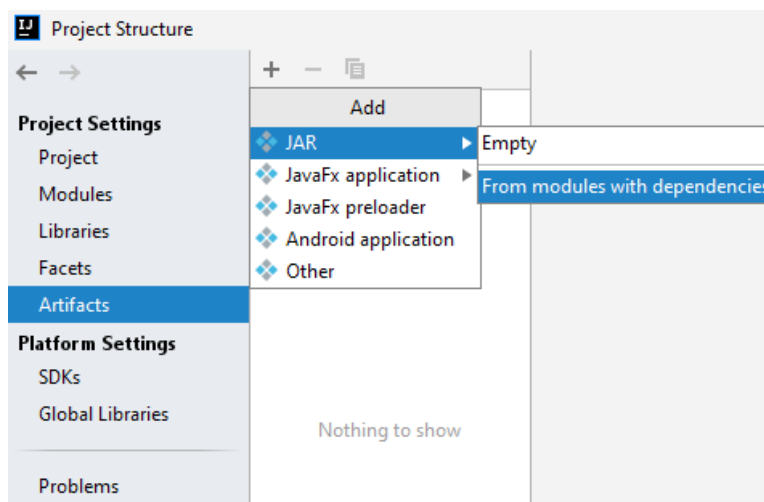
java.awt.\* doesnt cover java.awt.event.WindowAdapter.

These are seperate packages.

Even though its nested.

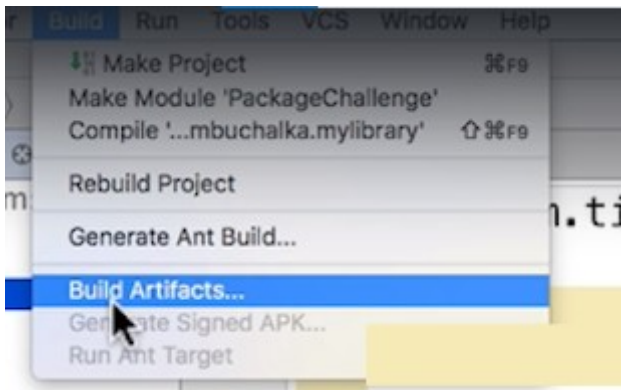
### Packages part 3:

Creating Java Archive (like a zip file)

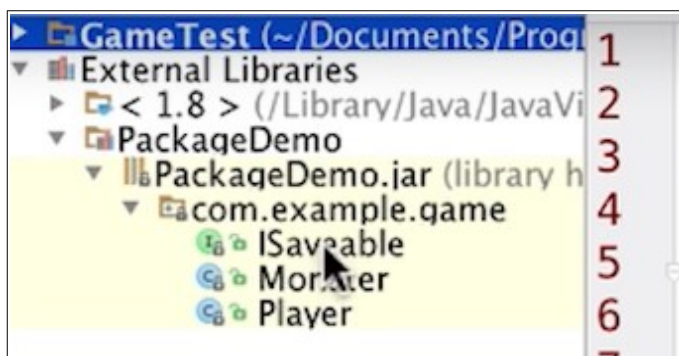
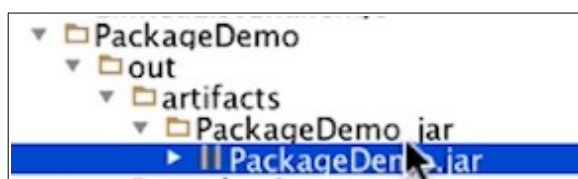
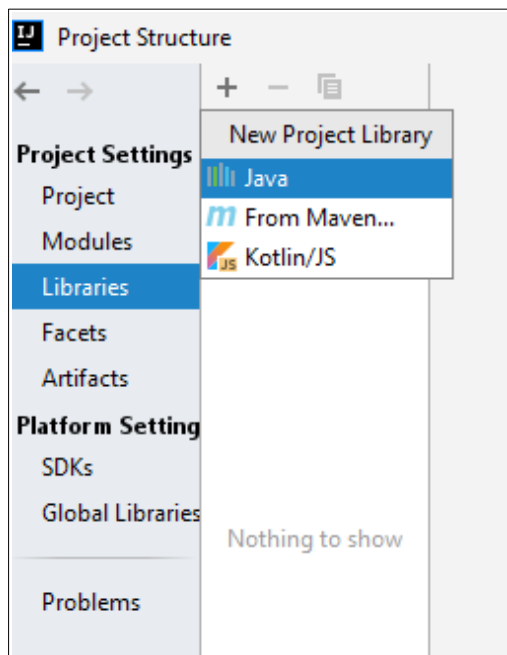


If u want the jar file to be executable specify the Main class.

Click on ok and see the output directory. Thats it.



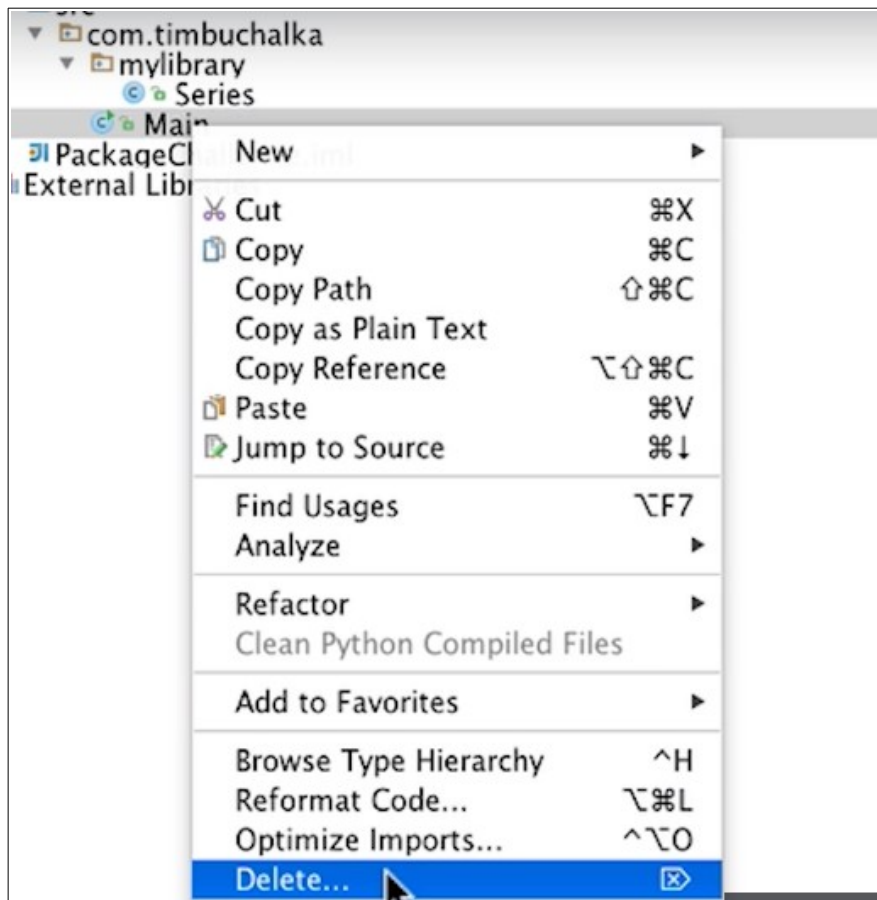
Step 2: Add jar files to projects.



```
Main.java x
1 package com.timbuchalka;
2
3 import com.example.game.ISaveable;
4 import com.example.game.Monster;
5 import com.example.game.Player;
6
7 import java.util.ArrayList;
8 import java.util.Scanner;
9
10 public class Main {
11
```

Here , we are able to execute the code.

## Package Challenge.



Delete the Main file after use and create ur jar file.

Scope: You already know

Access Modifiers:

```
public void withdraw(int amount) {
    int withdrawal = -amount;
    if (withdrawal < 0) {
        this.transactions.add(withdrawal);
        this.balance += withdrawal;
        System.out.println(amount + " withdrawn. Balance is now "
            + this.balance);
    } else {
        System.out.println("You dont have that much money");
    }
}

public void calculateBalance() {
    this.balance = 0;
    System.out.print(this.transactions.toString());
    for (int i : this.transactions) {
        this.balance += i;
    }
}
```

Main x

```
Balance is now 1000
500 withdrawn. Balance is now 500
You dont have that much money
Deposit failed Attempt!
[1000, -500]Calculated balance is 500

Process finished with exit code 0
```

**THIS IS Pretty smart.**

So access Control is granted by the top level or at the member level. So at the top level you can make ur classes or interfaces public or package private. You cant define a private class at the top level.

## Top Level

---

Only classes, interfaces and enums can exist at the top level, everything else must be included within one of these.

**public:** the object is visible to all classes everywhere, whether they are in the same package or have imported the package containing the public class.

**Package-private:** the object is only available within its own package (and is visible to every class within the same package). Package-private is specified by not specifying, i.e it is the default if you do not specify public. There is not a "package-private" keyword.

## Member level

---

**public:** at the member level, public has the same meaning as at top level. A public class member (or field) and public method can be accessed from any other class anywhere, even in a different package.

**Package-private:** this also has the same meaning as it does at the top level. An object with no access modifier is visible to every class within the same package (but not to classes in external packages).

**private:** the object is only visible within the class it is declared. It is not visible anywhere else (including in subclasses of its class).

**Protected:** the object is visible anywhere in its own package (like package-private) but also in subclasses even if they are in another package.

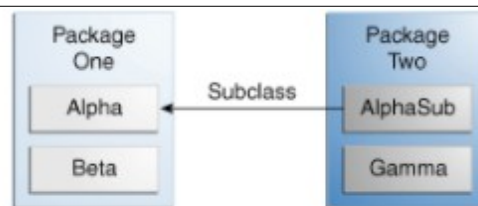


Link to understand: [Access Modifiers](#)

**Access Levels**

| Modifier    | Class | Package | Subclass | World |
|-------------|-------|---------|----------|-------|
| public      | Y     | Y       | Y        | Y     |
| protected   | Y     | Y       | Y        | N     |
| no modifier | Y     | Y       | N        | N     |
| private     | Y     | N       | N        | N     |

The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class declared outside this package have access to the member. The fourth column indicates whether all classes have access to the member.



**Classes and Packages of the Example Used to Illustrate Access Levels**

The following table shows where the members of the Alpha class are visible for each of the access modifiers that can be applied to them.

**Visibility**

| Modifier    | Alpha | Beta | Alphasub | Gamma |
|-------------|-------|------|----------|-------|
| public      | Y     | Y    | Y        | Y     |
| protected   | Y     | Y    | Y        | N     |
| no modifier | Y     | Y    | N        | N     |
| private     | Y     | N    | N        | N     |

Above Alpha is a subclass of AlphaSub.

Notes:

In protected you can extend, if u extending then u obviously have access to member variables.



```
// Challenge:
// In the following interface declaration, what is the visibility of:
//
// 1. the Accessible interface?
// 2. the int variable SOME_CONSTANT?
// 3. methodA?
// 4. methodB and methodC?
//
// Hint: think back to the lecture on interfaces before answering.

interface Accessible {
    int SOME_CONSTANT = 100;
    public void methodA();
    void methodB();
    boolean methodC();
}
```

1.package private, 2.public static final, 3.public ,4. public, 5. public

Note: You can make the methods package-private by making the interface here itself package private (duh!)

Static statement : [static statement](#)

```
public class StaticTest {  
    private static int numInstances = 0;  
    private String name;  
  
    public StaticTest(String name) {  
        this.name = name;  
        numInstances++;  
    }  
  
    public static int getNumInstances() {  
        return numInstances;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

```
5 Main {  
    static void main(String[] args) {  
        StaticTest firstInstance = new StaticTest("1st Instance");  
        System.out.println(firstInstance.getName() + " is instance number " + StaticTest.getNumInstances());  
  
        StaticTest secondInstance = new StaticTest("2nd instance");  
        System.out.println(secondInstance.getName() + " is instance number " + StaticTest.getNumInstances());  
  
        StaticTest thirdInstance = new StaticTest("3rd instance");  
        System.out.println(thirdInstance.getName() + " is instance number " + thirdInstance.getNumInstances());  
    }  
}
```

Static member/fields belongs to the class and not the instance of the class.

At time 12 bookmark, likewise psvm, a static method can access nonstatic fields and methods in another class because it creates an instance of a class in order to do so.

Final statement: [final statement](#)

-Generally use to define constant values, so strictly speaking final fields are not actually constants because they can be modified but only once, and any modification must be performed before the class constructor finishes.

- That means we can assign final field value when we first declare it or in the constructor.

```
public class SomeClass {
    public final int some_no;

    public SomeClass(){
        some_no=3;
    }

    public int getSome_no() {
        return some_no;
    }
}
```

```
public class SomeClass {
    public static int some_no;

    public SomeClass(){
        some_no=3;
    }

    public int getSome_no() {
        return some_no;
    }
}
```

```
public class SomeClass {
    public static final int some_no;

    public SomeClass(){
        some_no=3;
    }

    public int getSome_no() {
        return some_no;
    }
}
```

Variable 'some\_no' might not have been initialized  
Initialize variable 'some\_no' Alt+Shift+Enter

```
public class SomeClass {
    public final int some_no;

    public SomeClass(){
        some_no=3;
    }

    public int getSome_no() {
        return some_no;
    }
}
```

```
public class SomeClass {
    public static int some_no;

    public SomeClass(){
        some_no=3;
    }

    public int getSome_no() {
        return some_no;
    }
}
```

```
System.out.print(SomeClass.some_no);|
}
```

Main x

"C:\Program Files\Amazon Corretto\jdk15.0.2\_7\bin\

0

Static → shared  
final → modified only once  
before constructor  
finishes-

static final

You don't need a instance.

It's shared

Bad idea to put into a  
constructor cuz it's final

---

That's why It's necessary to  
initialize when declared.

```

public class SomeClass {

    private static int classCounter = 0;
    public final int instanceNumber;
    private final String name;

    public SomeClass(String name) {
        this.name = name;
        classCounter++;
        instanceNumber = classCounter;
        System.out.println(name + " created, instance is " + instanceNumber);
    }

    public int getInstanceNumber() {
        return instanceNumber;
    }
}

```

Here instanceNumber is consider constant after we assign its value.

Thats the reason why its not uppercase.

PSF variable is a value which is not gonna change and it doesnt make sense to store a copy of that for every class instance.

```

public final class Math {

    | Don't let anyone instantiate this class.

    private Math() {}
}

```

The constructor is marked private to prevent instance to be created.

The reason that is done in the Math class cuz all math class methods are static.

To enforce this the creator make the constructor private.

You'll think I can go around that by creating a subclass which you instantiate, and the creator have thoughted that pretty well by declaring the Math class **final**.

Which prevents the class to have any subclass.

Same for method declared as **final**. You dont want method over-riding.

Video : [Final part 2 and static initializer](#)

Opposite to our normal instance constructor we have static initialization block.

In this that block is executed once the class is first loaded into the project.

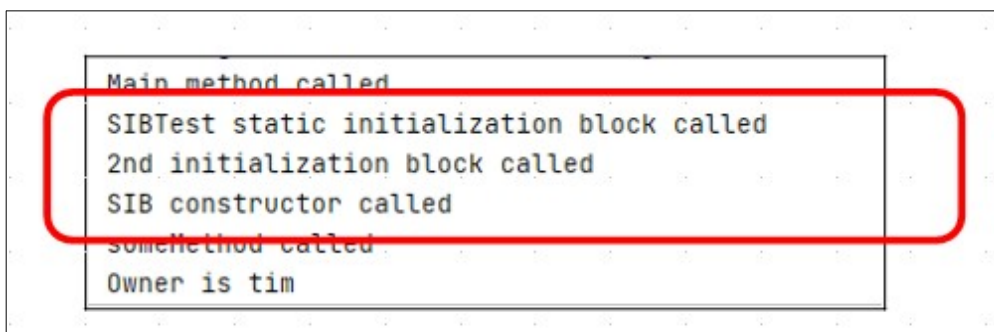
Static final variable must be initialized by the time or static initialization blocks terminate. So in the same way that we can set the value of final field in the constructor. We can also assign the value of a static final variable in the static initialization block.

Static initialization block is an Advanced feature and theres rarely a case you would use them.

```
public class SIBTest {  
    public static final String owner;  
    static{  
        owner="tim";  
        System.out.println("SIBTest static initialization block calle  
    }  
    public SIBTest(){  
        System.out.println("SIB constructor called");  
    }  
    static {  
        System.out.println("2nd initialization block called");  
    }  
    public void someMethod(){  
        System.out.println("someMethod called");  
    }  
}
```

You can have multiple static initialization block and they are called in the order they are declared in the class.

There is a **Warning**, the second initialization block should be after first initialization block to avoid confusion. It doesnt matter but still.



```
Main method called  
SIBTest static initialization block called  
2nd initialization block called  
SIB constructor called  
someMethod called  
Owner is tim
```

