# JAVA OOP

Purpose or OOP languages is to implement real word entities

OOP language over procedural

* It’s easy to develop and support applications over procedural language
* OOP provides data hiding. Whereas in procedural language global data can be accessed anywhere
* Make code reuale

**CLASS** is blue print from which you can create [objects]

Class is logical entity

**OBJECT** is runtime entity that has state and behavior. Object is created from the class or instance of class. Everything that surrounds us is ab object (pen, table, chair, car)

For example, dog is an object.

* It has state - color, name, whiskers
* It has behavior – barking, eating, wagging tail

Object has address to the memory and therefore it takes up some space

Object is physical entity

Object has:

* **state** - represent data (value) of an object
* **behavior** – represents functionality (deposit, withdraw)
* **identity** – is ID. It is not visible to external users. It’s used internally by JVM to identify each object uniquely

[**new**] keyword is used to allocate memory at runtime. All objects get memory in [Heap memory] are

**Declaration -> Instantiation-> Initialization**

**1.Declaration** assign type to the variable

Person p;

**2.Instantiation** means to creating new object. It means allocation memory at runtime in [Heap] memory ] area

Person p = new Person();

Person p = new Person(“Alex”);

**3.Initialization** – passing parameters to constructor to define (=*storing data in Object*)

public class Person

{

private String name;

private String color;

public Person(String name)

{…}

public Person(String name, String color)

{…}

}

# INITIALIZE OBJECT

There are 3 ways to initialize object:

1. **by reference variable**



1. **by method**



1. **by constructor**



# INSTANTIATE OBJECT

It means allocation memory at runtime in [Heap] memory

There are 5 ways to do it:

* By [new ] keyword
* By [newInstance()] method
* By [clone] method
* By deserialization
* By factory method

# CONSTRUCTOR

Constructor can not be:

* status
* final
* synchronized

Constructor *overloading* is allowed in Java. Overloading is just have constructor with different parameters

|  |  |
| --- | --- |
| CONSTRUCTOR | MEHTOD |
| *Constructor is used to initialize the state of Object* | Method is to make some behavior |
| It does not have return type | It must have return type |
| Constructor is invoked implicitly | method is invoked explicitly |
| Java provides default constructor | Method must be developed |
| Constructor name must match with class name | Any name |

public class Student

{

public Student()

{…}

public Student(String name)

{…}

public Student(int id, String name)

{…}

}

# STATIC

Static relates to class than instance.

Static can be

* **variable** - it is used for cases when variables is used across all object. For example, SK, company name, university name. Static variable **gets memory only once**.
* **method** -
* static block – this block executed when class is loaded in the memory. It’s used to initialize static data
* nested class -

note: is it possible to execute java program without main method? It was possible in static bloch till JGK 1.6. After it’s not possible

# INHERITANCE

**INHERITANCE** – is OOP concept. When one object acquires all the properties and behaviors of parent class it’s called inheritance.

It’s used to achieve [runtime polymorphism]

# POLYMORFISM

**POLYMORFISM** is when task can be performed in different ways

**POLYMORFISM** is the ability of object takes different forms

Polymorphism:

* static binding
* dynamic binding

# ADBSTRACTION

**ABSTRACTION** is hiding details and showing only functionality

# COUPLING

**COUPLING** refers to dependency to another class. You can use interface to achieve weakly coupling, because it does not have concrete implementation.

# COHESION

Weakly cohesive method will split the task into separate parts

# ASSOCIATION

**ASSOCIATION** represents relationship between objects:

* 1:1
* 1:n
* n:1
* n:n

# AGGREGATION

**AGGREGATION** (**HAS-A**) is a way to achieve association.

**AGGREGATION** (**HAS-A**) represents the relationship where one object contains others as a part of its own state. It represents a **weak relationship** between objects

represents re

# COMPOSITION

**COMPOSITION**(**IS-A**) is also way to represents the relationship where one object contains others as a part of its own state. It represents a **strong relationship** between objects

# [this] keyword

[this] can be used to refer to instance variable of current class

[this] can be used to invoke current class method

[this] can be used to invoke current constructor method

[this] can be used to pass argument to the method

1.[this] can be used to refer to instance variable of current class



2.[this] can be used to invoke current class method



3.[this] can be used to invoke current constructor method



4.[this] can be used to pass argument to the method



|  |  |
| --- | --- |
| **HOW ACHIEVE CONCEPT** | **DESC** |
| abstraction | Using abstract class and inheritance |
| runtime polymorphism | Inheritance (reference variables) |
| STRING |  |
|  |  |

# STRING

**String** is object that represents sequence of char values.

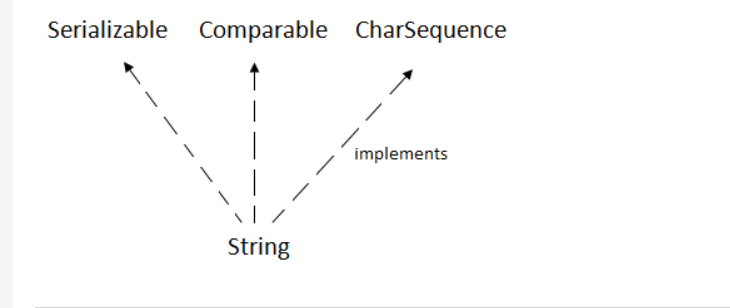
**String is immutable. It means it cannot be changed. It means if we change new instance in created**

For mutable use:

StringBuffer and StringBuilder

String implements interfaces:

* Serializable
* Comparable
* CharSequence



There are 2 ways to create String object

* **By string literals**. Use double quotes

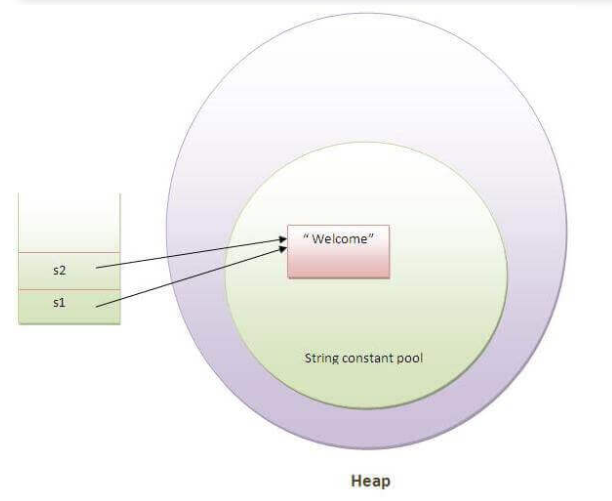
**When you create in this way JVM checks “string constant pool” first**.

1. If string already exists it returns reference to the pooled instance.
2. If string does not exists in the pool. It creates new string instance and placed in the pool

For, example.

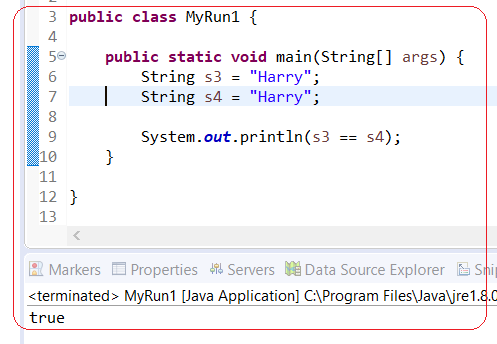
String s1=”abc”;

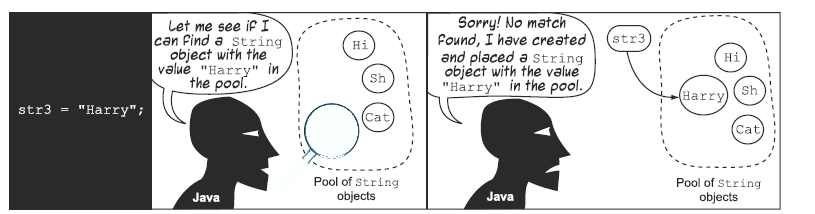
String s2=”abc”; -- it does not create the second time

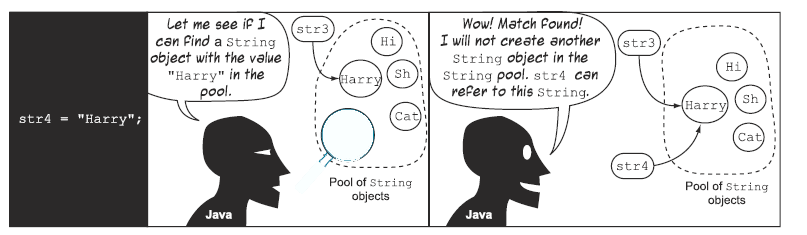


- 

Example,







* **By [new] keyword** –

**Example,**

* **In this case JVM will create a new String object in normal (non-pool) heap memory**
* **Literal “abc” will be placed in the string constant pool**
* **Variable [s] will refer to the object in heap memory(non-pool)**

 **it creates 2 objects and 1 reference variable**

Note: String created by [new] operator always refer to Separate objects

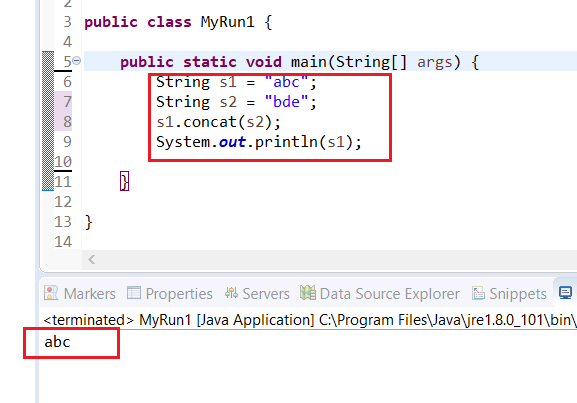
# IMMUTABLE

**String is Immutable**. Immutable means – not changeable. It helps JVM to reuse objects

**If String variable modified it means it will return new String object**.

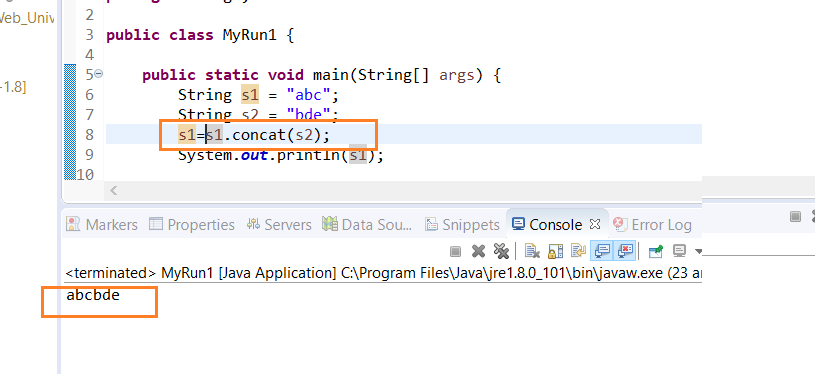
**CASE1**

In this example, [concat] adds variable to the end. However, because [s1] is not changeable s1 will be not



**CASE2**

But if we explicitly add value we will get desirable result



# COMPARE STRING

There are 3 ways to compare strings

1. By equals method – it compares original content of the string. It provides 2 methods

[equals] – it compares this string to the specified object

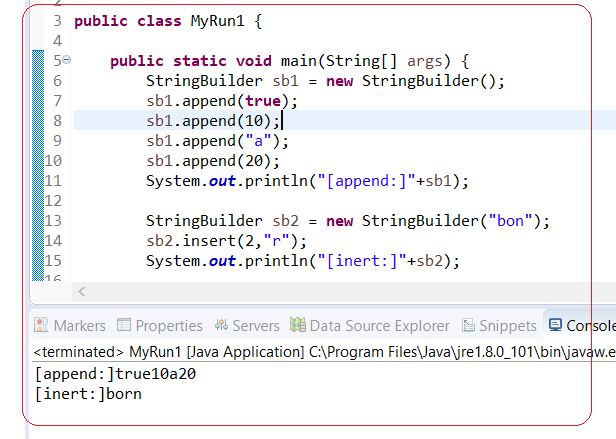
1. By == operator
2. By CompareTo method

# STRINGBUILDER

StringBuilder is mutable class. You can use it when String is *modified quite often*

There are 3 methods that are missed in String class

* **append** -
* **insert** –a difference between [insert] and [append] is that [insert] can input in the middle
* **delete** – can also delete in the middle



# STRINGBUFFER

**StringBuffer** – is thread safe (synchronized) class. It means only one thread out of multiple threads can execute your method. But it affects your performance

# EQUALS

**Identity “==” (Reference equality)** – is defined by reference it hold. If two variables hold the same reference they are identical

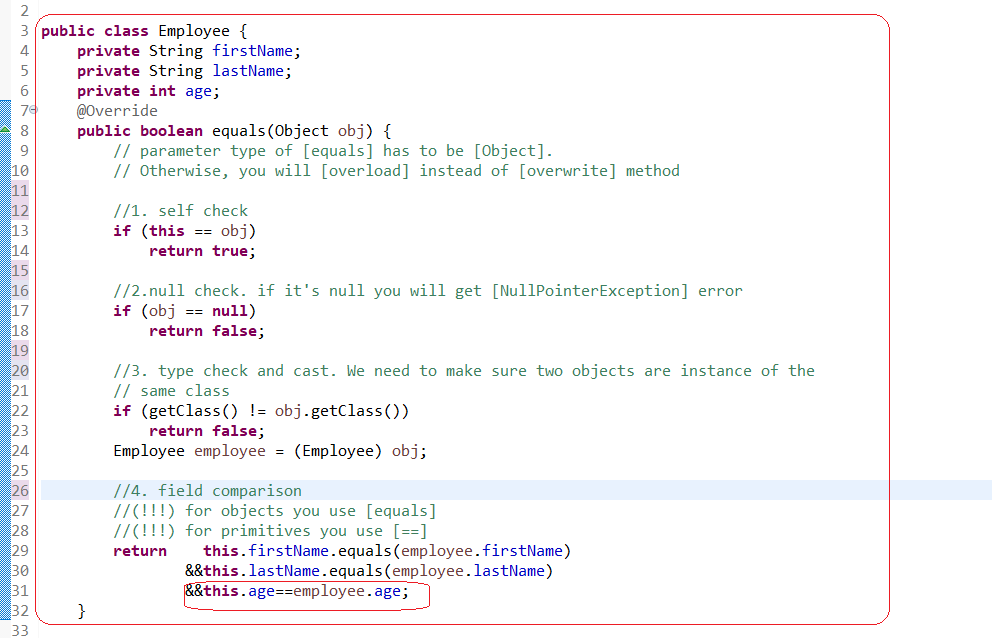
**Equality** – is defined by the value

**(!) By default [equals] check identity (only references**).

**[equals] has to satisfy:**

* It is ***reflexive***: for any non-null reference value x, x.equals(x) should return true.
* It is ***symmetric***: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
* It is ***transitive***: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
* It is ***consistent***: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

# EQUALS IMPLEMENTATION



# HASHCODE

Hashing is fundamental concept of computer science. It stands behind HashMap, HashSet, HashTable. These collections HashMap, HashSet, HashTable claclucalate hash value for a given key using HashCode() value. And use this value internally to store data.

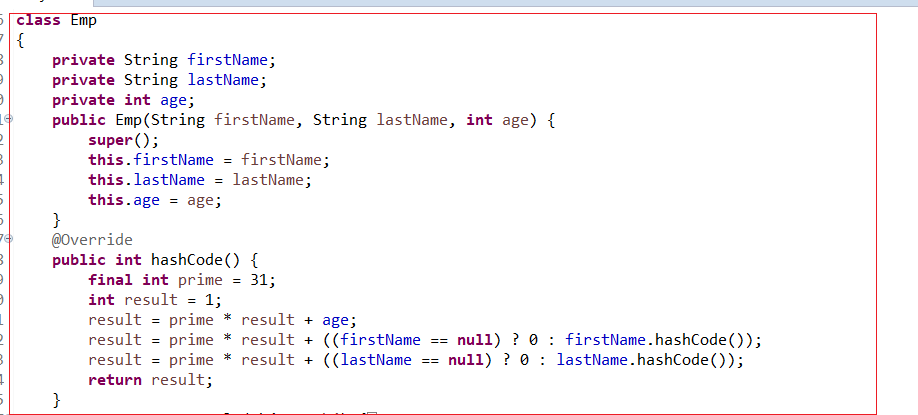
**HashCode** –is integer value generated by hashing algorithm

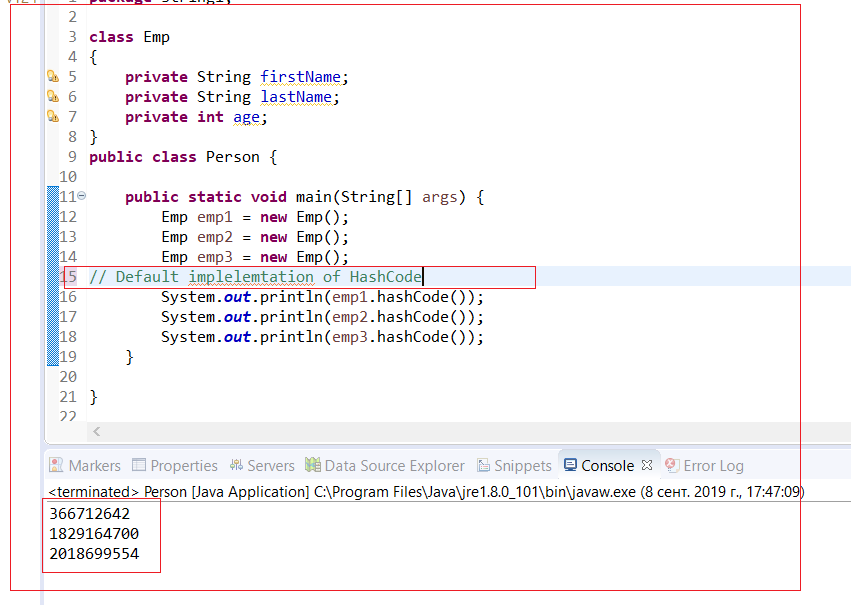
***If [hashcode] comparison returns false [equals] returns false as well***

***By default, [hashcode] method returns a random integer that is unique for each instance. This integer might change between several executions of the application and won't stay the same .So, if you don’t override [hashcode] ot will violate principle of [Equal Consistency]: “*equal objects should return the same hashCode**.***”***

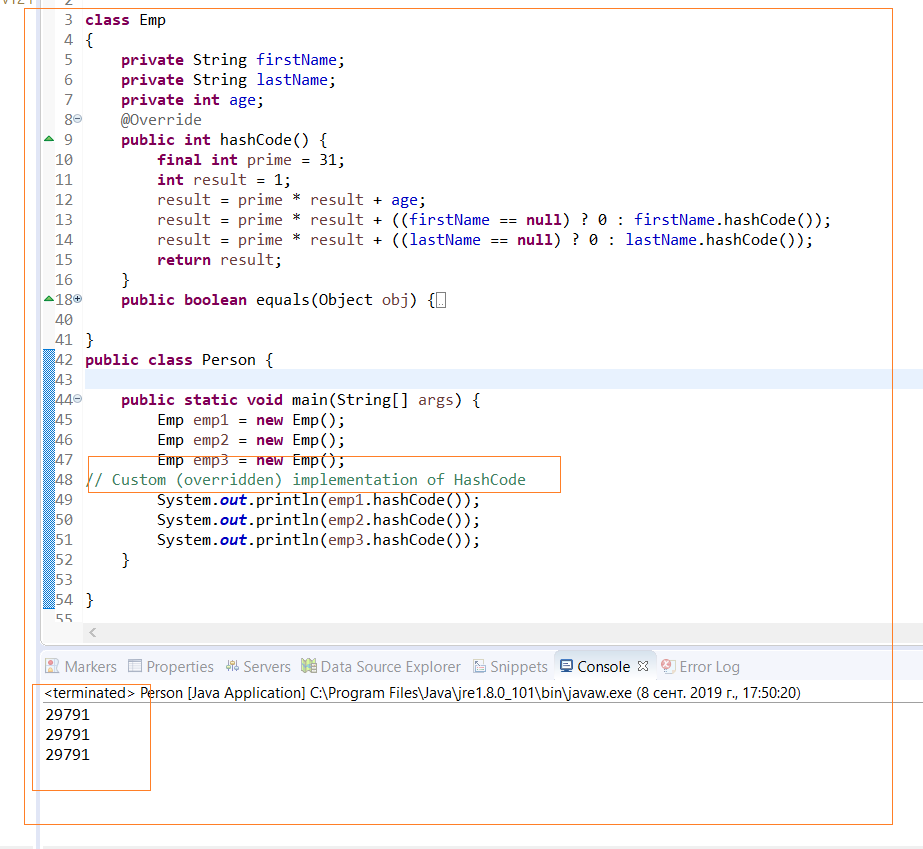
Note: **Prime Number** = in Russian as “Простое число”

[Prime number] is used in hashing algorithm just for good number distribution (to *avoid collision* )

******

**1.Default implelemtation of HashCode****

**2.Customs implementation of HashCode.**

******

**Hashcode has to satisfy 3 criteria**:

* **Internal consistency** – the value of of [hashcode()] may only change if a property of [equals()] is change
* **Equals Consistency** – equal objects should return the same hashcode. It means if we override [equals() we also should override [hashcode()]
* **Collisions** - unequal objects may have the same hashcode

# EXAMPLE WHY YOU NEED TO OVERRIDE HASHCODE FOR HASHSET

Example with hashset, why you need to override equals and hashcode

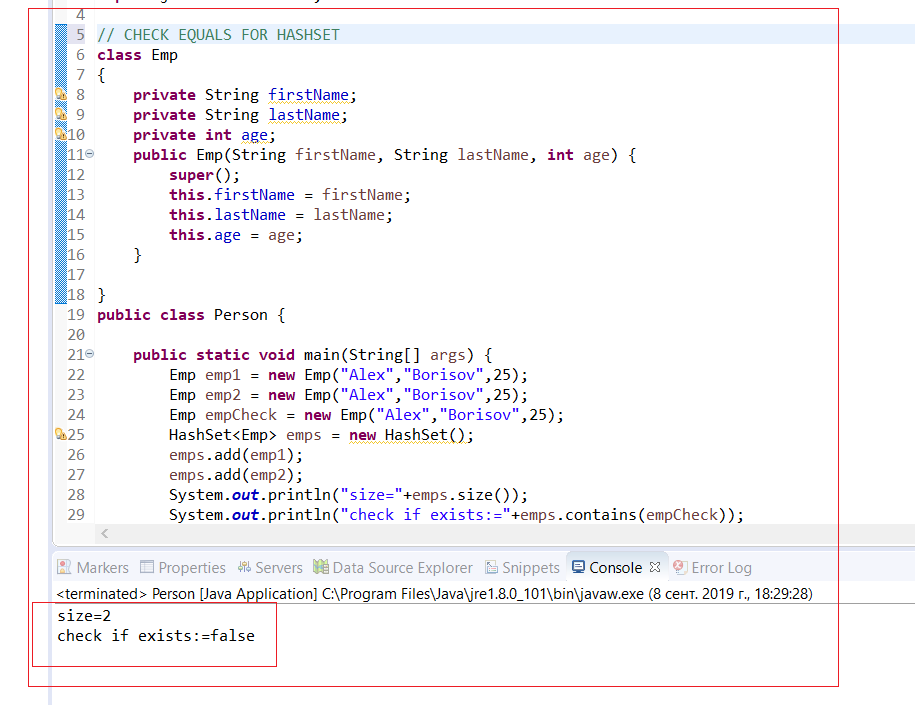
**Hashset does not allow duplicates. However, if you only override equals and not hashcode you will get not correct result with duplicates**

**HashSet stores element in memory buckets. Each bucket is linked to particular hashcode.**

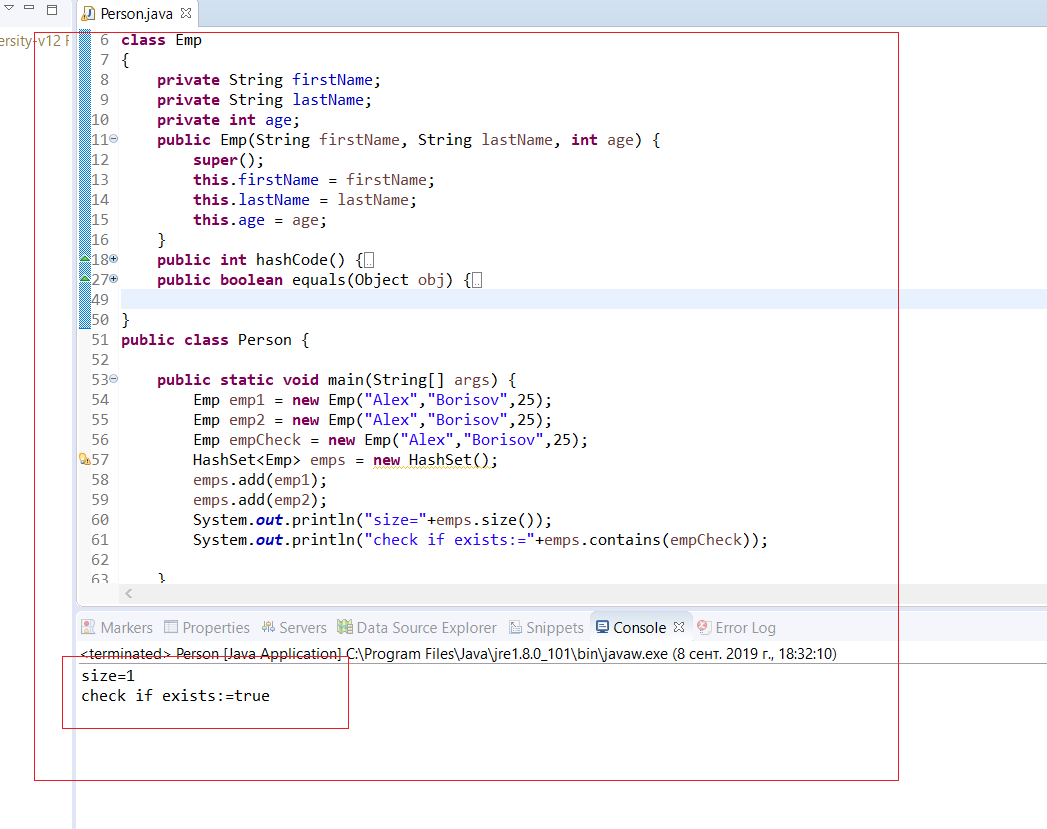
**However, because emp1 and emp2 has fifferent hashcode that’s why HashSet keeps them in different buckets and thus allow duplicates.**

**The same it applies to HashMap and HashTable that uses hashing mechanism for storing elements**

**1.Default implementation of HashCode and equals. Resuls is incorrect**



**2.Custom implementation of HashCode and equals. Resuls is correct**



# OBJECT

Object – is element of package [java.lang]. Every class is [directly] or [indirectly] of class [Objects]. If class does not [extend] any classes it means it is a child of [Object]

# METHODS OF OBJECT

|  |  |  |
| --- | --- | --- |
| ID | METHOD |  |
| 1 | **toString()** |  |
| 2 | **hashCode()** |  |
| 3 | **equals()** |  |
| 4 | **getClass()** |  |
| 5 | **finalize()** |  |
| 6 | **clone()** |  |
| 7 | **wait()** |  |
| 8 | **notify()** |  |
| 9 | **notifyAll()** |  |

**[toString()]**

The default implementation is not in readable format

It returns:

* Name of class + @ + hexadecimal representation of hashcode object

Note: it’s a common practice don’t use it in production/enterprise projects. It’s more for developing, internal logging

**[getClass()]**

Returns runtime class of your [Object]

=it returns Object of type of your Object

Note: it often is used in [instanceOF]

**[finalize()]**

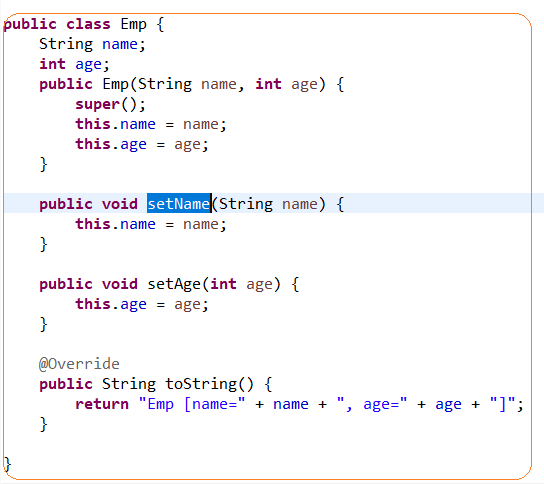
This method is called by Garbage Collector

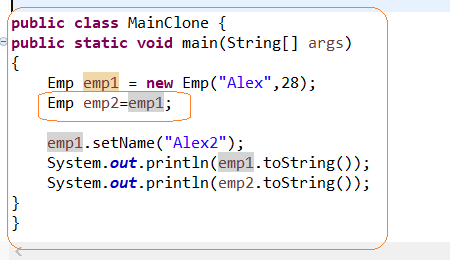
**[clone()]**

It creates exact the same copy of object and initializes all its fields. *To clone object you need to implement [Cloneable] interface*

OPERATOR “=”

If use “=” to make a copy of object then it will be not real copy. New object will have reference on the same address location. Due to this any changes in reference object will be in original object

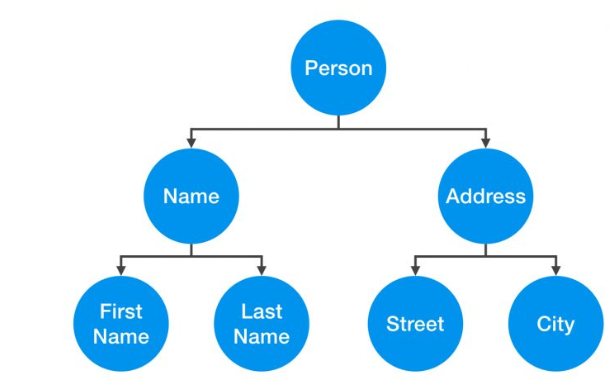


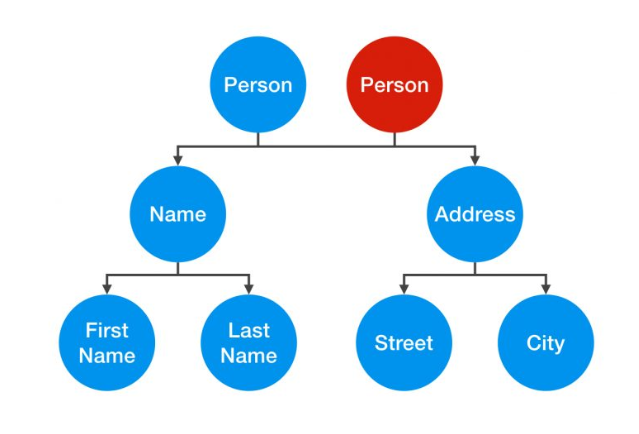


There are 2 copies of clone

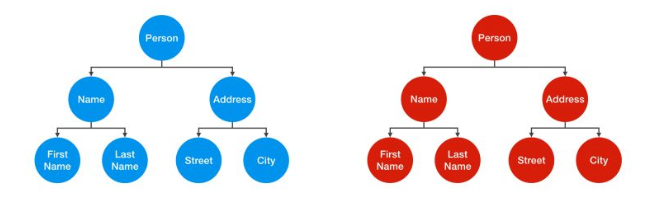
**Shallow copy** (default implementation of clone method) - it copies “main” object, but does not copy “inner” objects. If you modify inner object in reference object then changes will be reflected in first object. The problem of [shallow copy] is that two objects are not completely independent

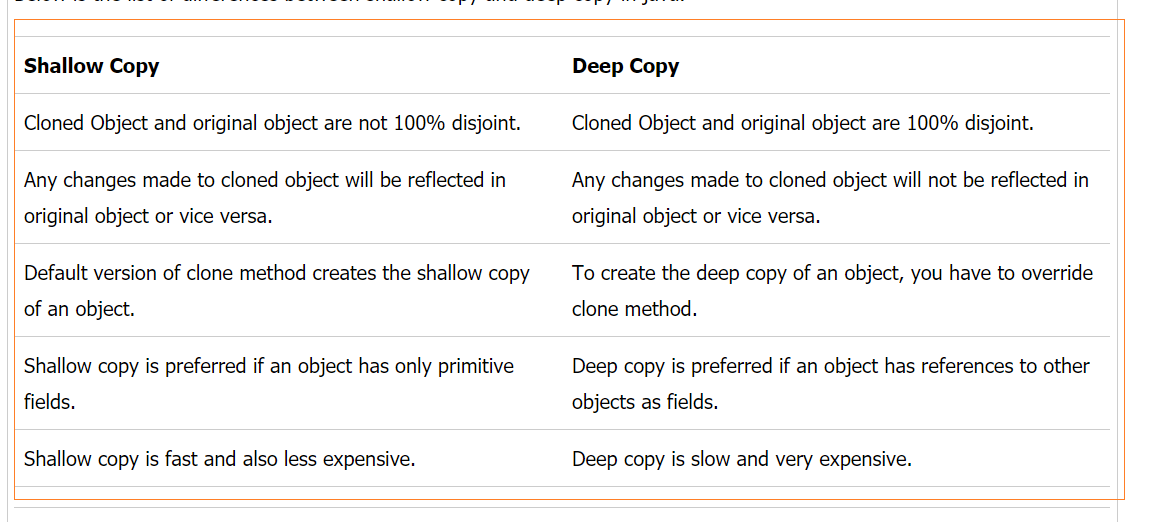
For example, Class [Person] consists of 2 classes [Address] and [Name]. when we clone “inner” classes will be reference to one Object in memory





**Deep Copy** – is fully independent copy. A change in “inner classes” would not be reflected in other object. If you want to have [deep copy] you will need to override [clone method]





# LINKS

Links:

<https://www.javatpoint.com/this-keyword>

* equals:

<http://users.csc.calpoly.edu/~gfisher/classes/102/info/howToOverrideEquals.html>

* loose coupling

<https://www.interviewsansar.com/2018/03/24/loose-coupling-and-tight-coupling-in-java/>

* generics

<https://www.baeldung.com/java-generics-interview-questions>

<https://www.journaldev.com/1663/java-generics-example-method-class-interface>

<https://howtodoinjava.com/java/generics/complete-java-generics-tutorial/amp/>

* web

<https://www.javatpoint.com/get-vs-post>

(!!!!)

<https://www.restapitutorial.com/lessons/httpmethods.html>

<https://www.journaldev.com/1854/java-web-application-tutorial-for-beginners>

<https://www.journaldev.com/2513/tomcat-datasource-jndi-example-java>

<https://www.journaldev.com/1854/java-web-application-tutorial-for-beginners>

* jsp

<https://www.tutorialspoint.com/jsp/>

* Rest-api

<https://zapier.com/learn/apis/>