METH

# STRING

**String** is class of [java.lang] package that represents sequence of char values

String class is immutable in java. **Immutable** means once the object is created, its value can not be changed.

* String is final class, because String is final it can be safely shared between multiple threads without any extra synchronization.

*The string is final by design in Java, some of the points which make sense why String is final is Security, optimization and to maintain a pool of String in Java*

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

**String**

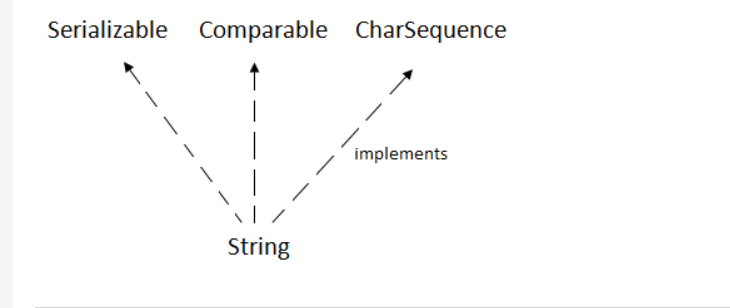
* **Final – it means nobody can extend from this class**
* **Immutable – once created the value can not be changed If yes then the other object is created**

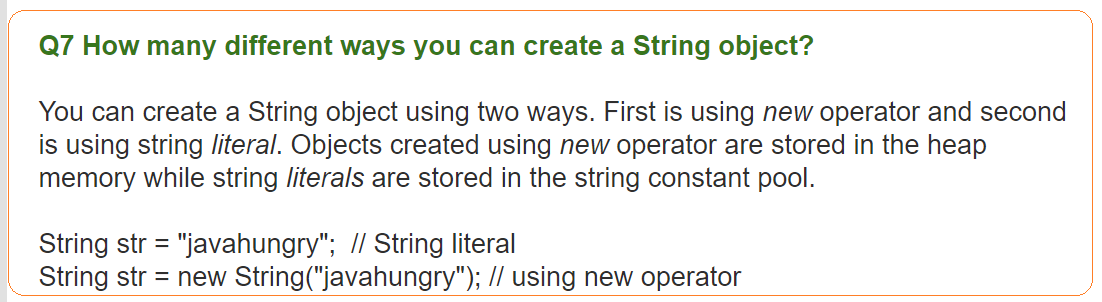
For mutable use:

StringBuffer and StringBuilder

String implements interfaces:

* Serializable
* Comparable
* CharSequence





There are 2 ways to create String object

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

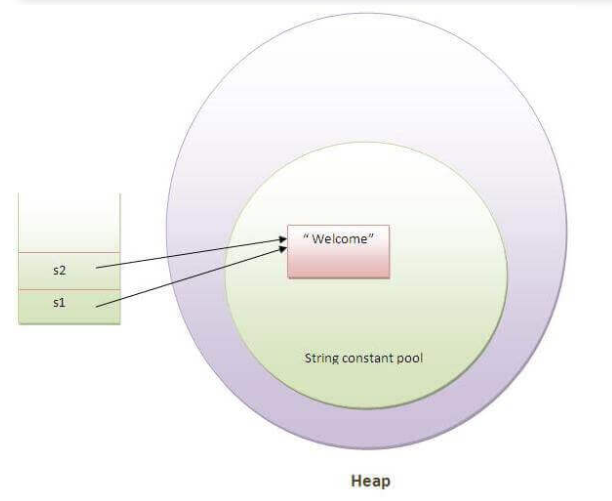
**When you create in this way JVM checks “string constant pool” first**.

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.

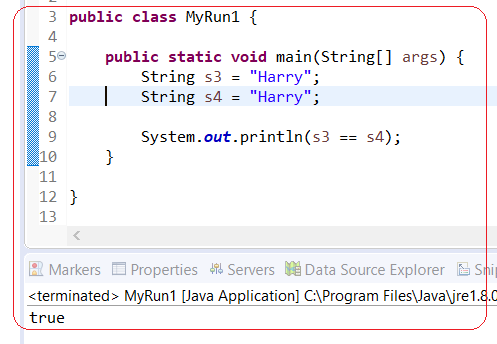
String s1=”abc”;

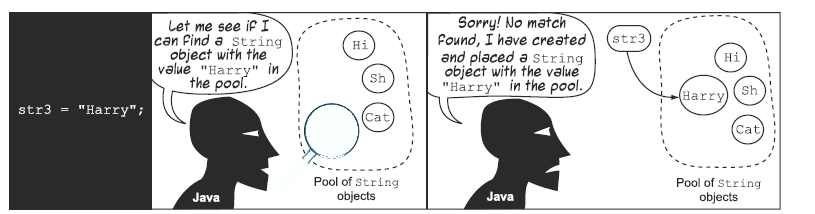
String s2=”abc”; -- it does not create the second time

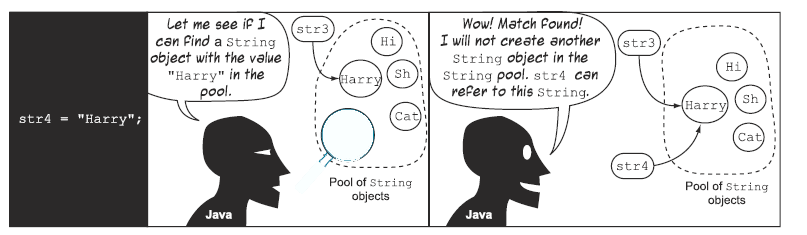


- 

Example,







* **By [new] keyword** –

**Example,**

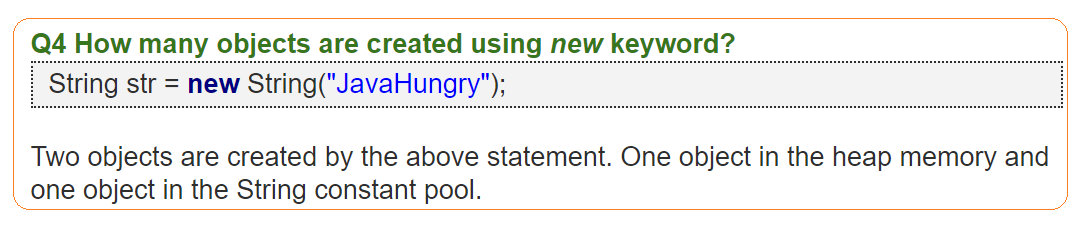
* **In this case JVM will create a new String object in normal (non-pool) heap memory**
* **Literal “abc” will be placed in the string constant pool**
* **Variable [s] will refer to the object in heap memory(non-pool)**

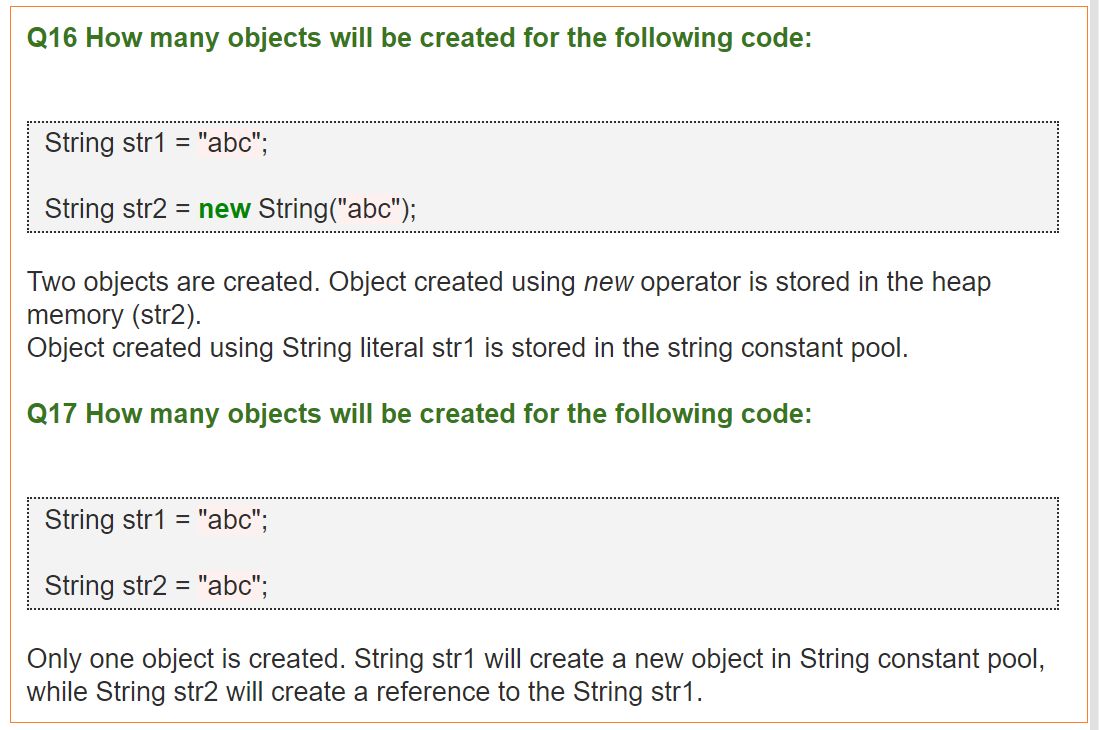


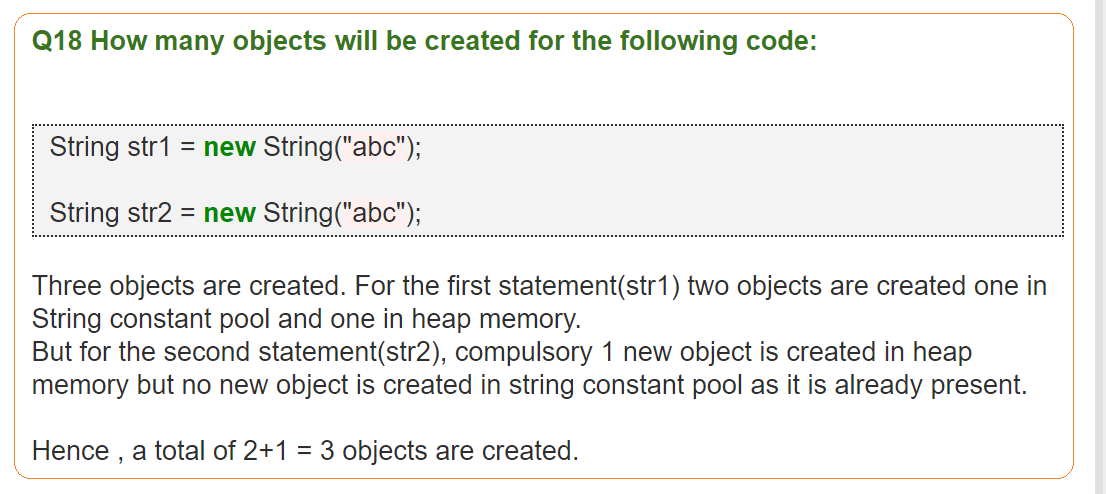
**it creates 2 objects and 1 reference variable:**

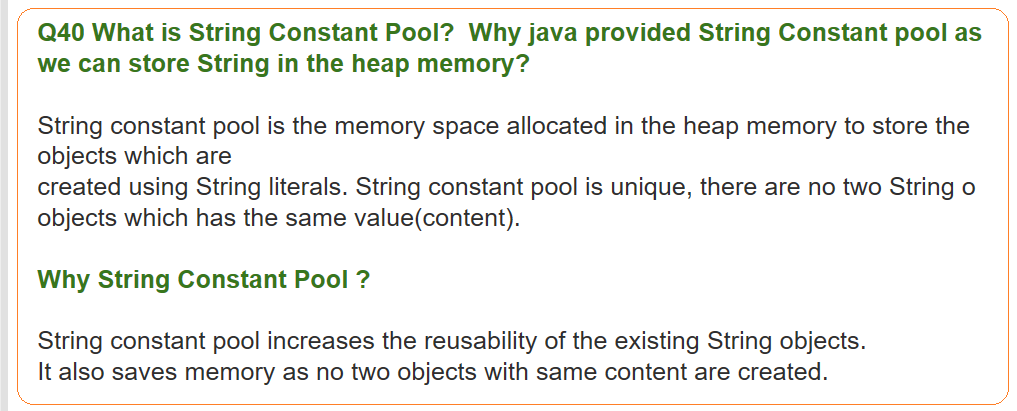
* **heap memory**
* **one object in the String constant pool.**

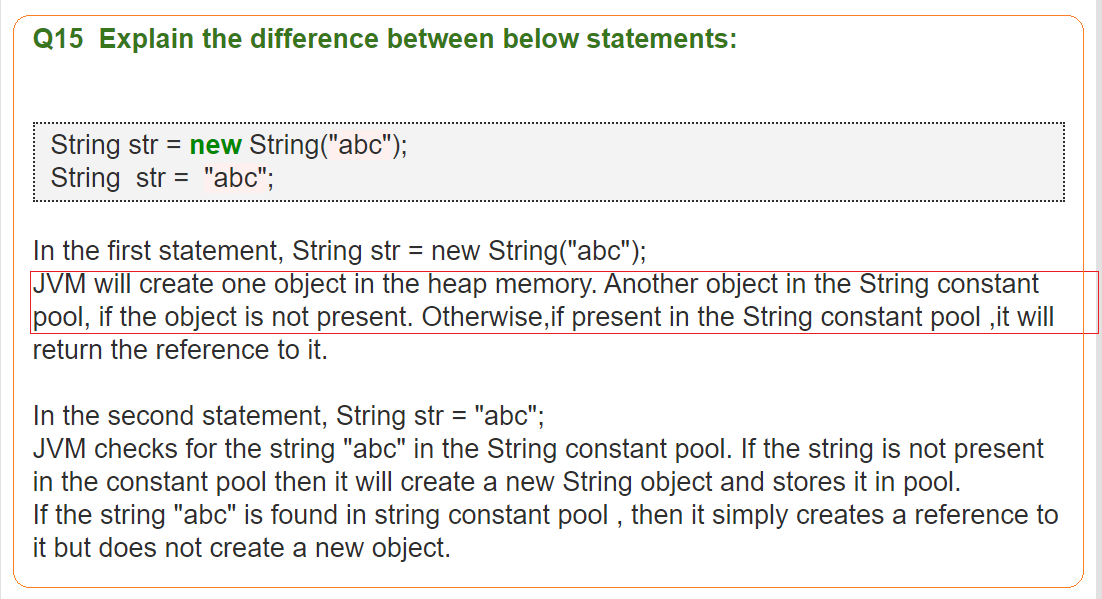
Note: String created by [new] operator always refer to Separate objects











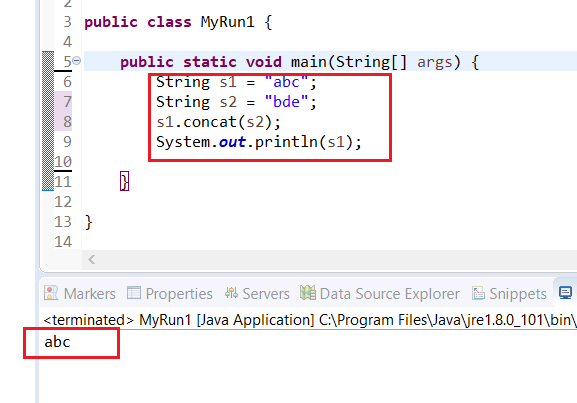
# IMMUTABLE

**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**.

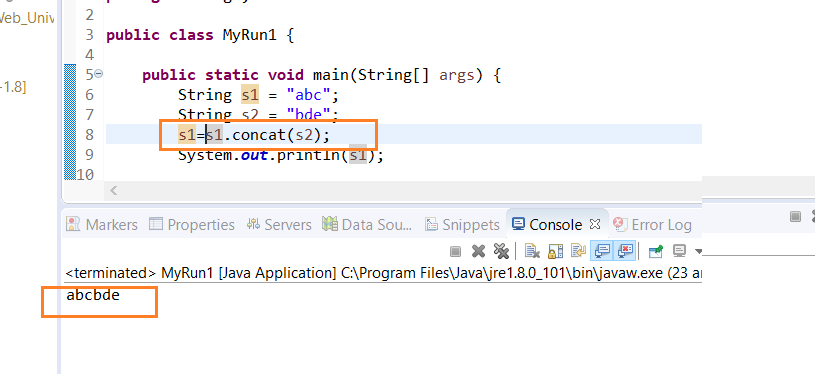
**CASE1**

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

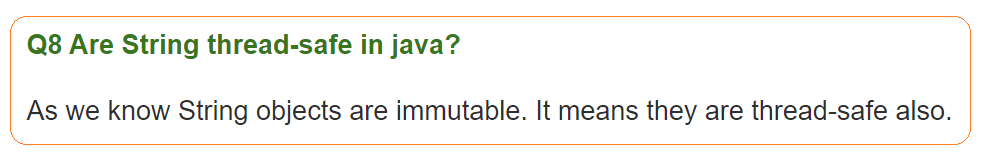


**CASE2**

But if we explicitly add value we will get desirable result



**CASE3**



# 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 - The compareTo() method compares two strings lexicographically and returns integer value. The comparison is based on the Unicode value of each character in the strings.

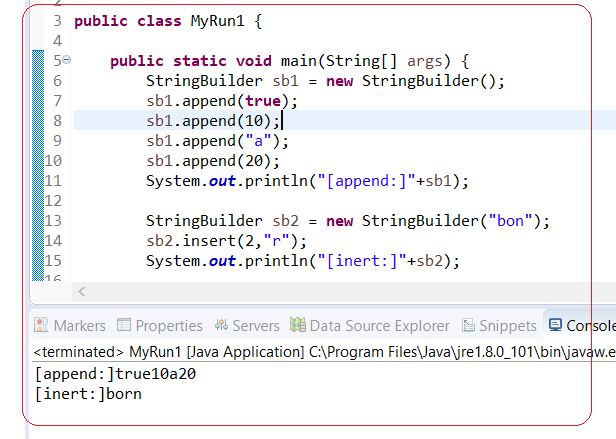
**note: Use the equals() method to compare two strings without consideration of Unicode values.**

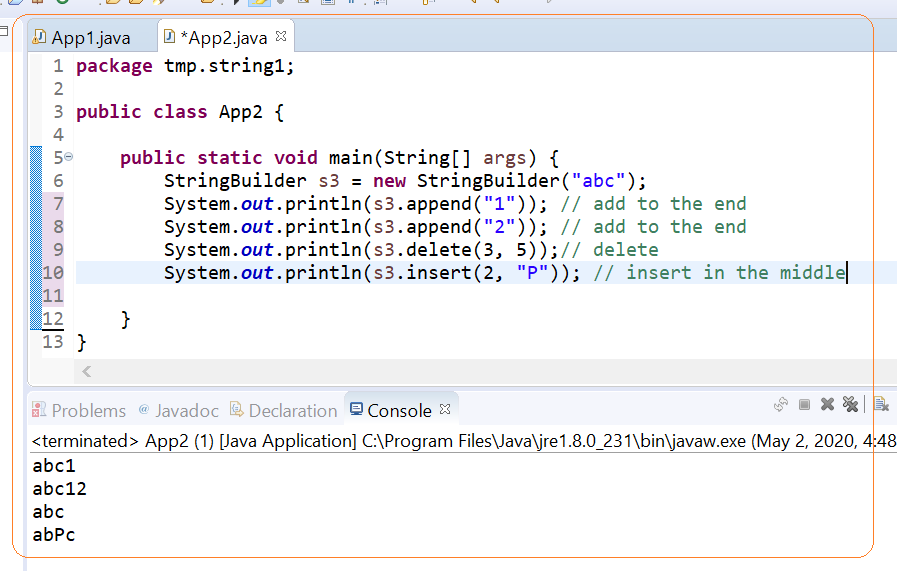
# STRINGBUILDER

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

There are 3 methods that are missed in String class

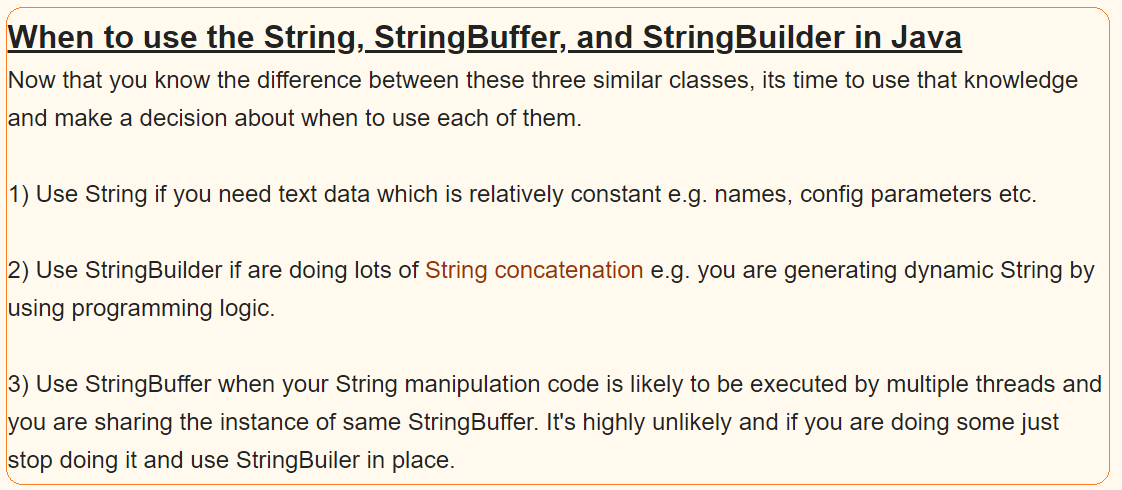
* **append** -
* **insert** –a difference between [insert] and [append] is that [insert] can input in the middle
* **delete** – can also delete in the middle





# 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

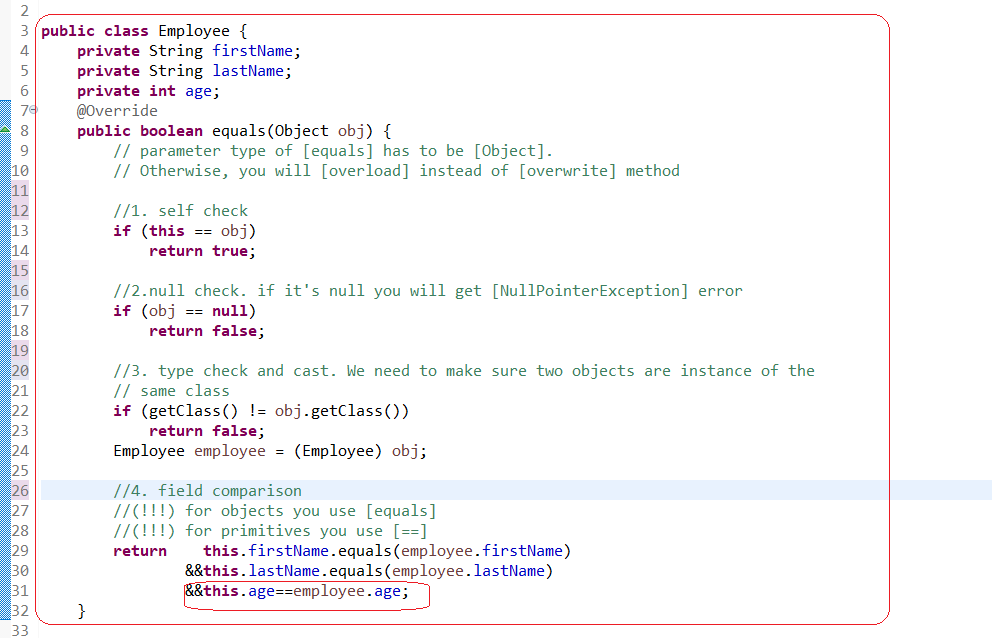
**Equality** – is defined by the value

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

**[equals] has to satisfy:**

* It is ***reflexive***: for any non-null reference value x, the formula [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.equv4als(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

**Hash Function** is a way to create a compact representation of an arbitrarily large amount of data.

***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**.***”***

*What would happen if the two objects are equal but return different hashCodes? Your code would run perfectly fine. You will never come in trouble unless and until you have not stored your object in a collection like HashSet or HashMap. But when you do that, you might get strange problems at runtime.*

*To understand this better, you have to first understand how collection classes such as HashMap and HashSet work. These collections classes depend on the fact that the objects that you put as a key in them must obey the above contract. You will get strange and unpredictable results at runtime if you do not obey the contract and try to store them in a collection.*

*Consider an example of HashMap. When you store the values in HashMap, the values are actually stored in a set of buckets. Each of those buckets has been assigned a number which is used to identify it. When you put a value in the HashMap, it stores the data in one of those buckets. Which bucket is used depends on the hashCode that will return by your object. Let’s say, if hashCode() method return 49 for an object, then it gets stored in the bucket 49 in the HashMap.*

*Later when you try to check whether that collection contains an element or not by invoking Contains(element) method, the HashMap first gets the hashCode of that “element." Afterwards, it will look into the bucket that corresponds with the hashCode. If the bucket is empty, then it means we are done and it's return false which means the HashMap does not contain the element.*

*If there are one or more objects in the bucket, then it will compare “element” with all other elements in that bucket using your defined equal() function.*

***Why Buckets***

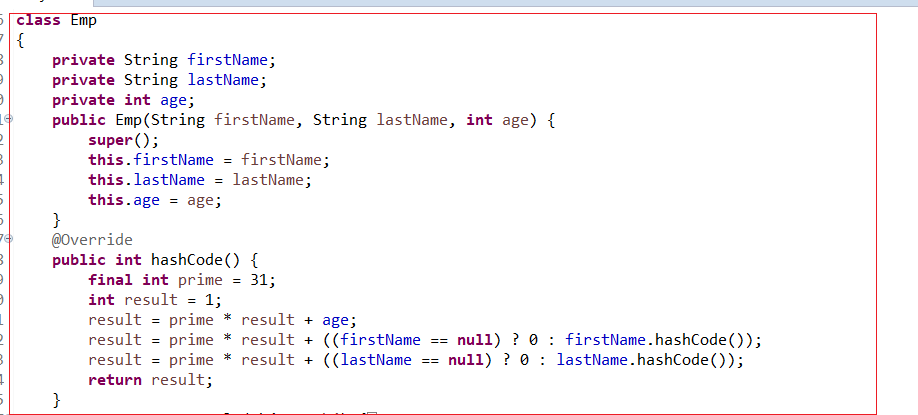
*The reason why bucket mechanism is used is its efficiency. You can imagine that if all the objects you put in the HashMap would be stored in one big list, then you have to compare your input with all the objects in the list when you want to check if a particular element is in the Map. With the use of buckets, you will now compare only the elements of the specific bucket and any bucket usually holds only a small portion of all the elements in the HashMap.*

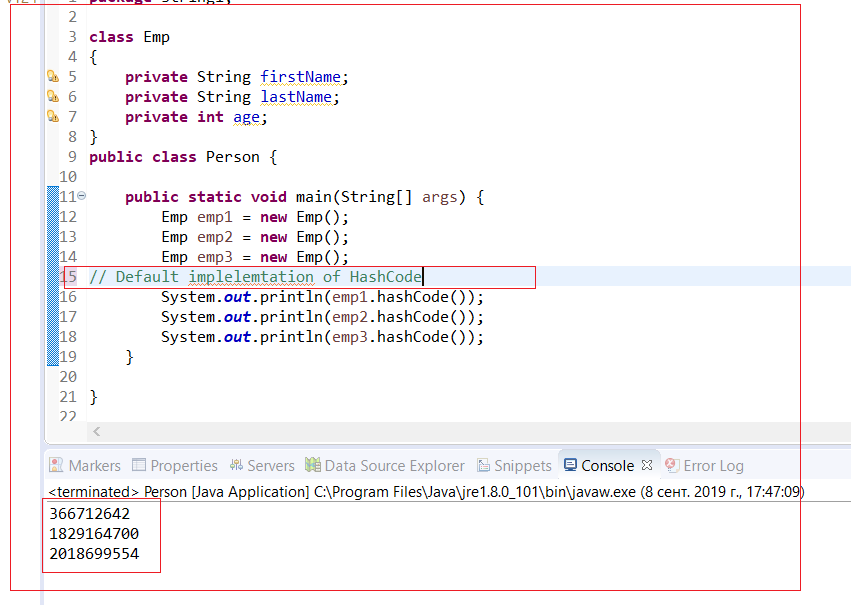
If 2 objects have the same hashcode they are *potentially* the same.

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

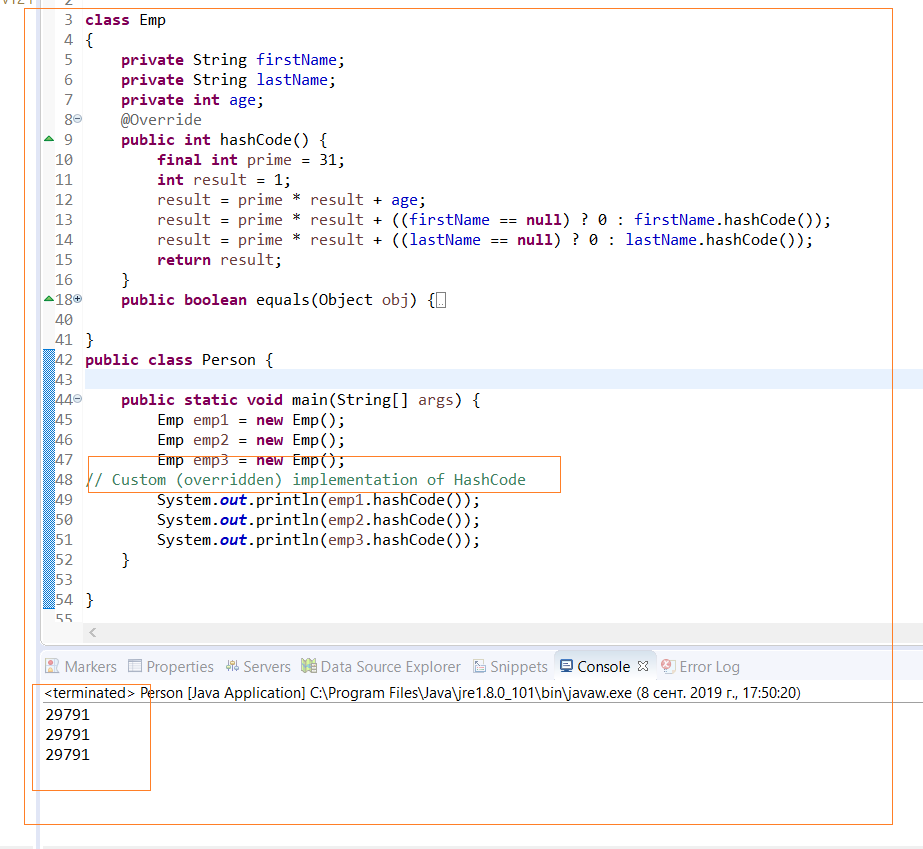
[Prime number] is used in hashing algorithm just for good number distribution (to *avoid collision* )

**We can choose any other prime number**. For example, 17 or 31 or so on. Using prime number the objects will be distributed better over the buckets We can choose any other prime number. Using prime number the objects will be distributed better over the buckets

******

**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

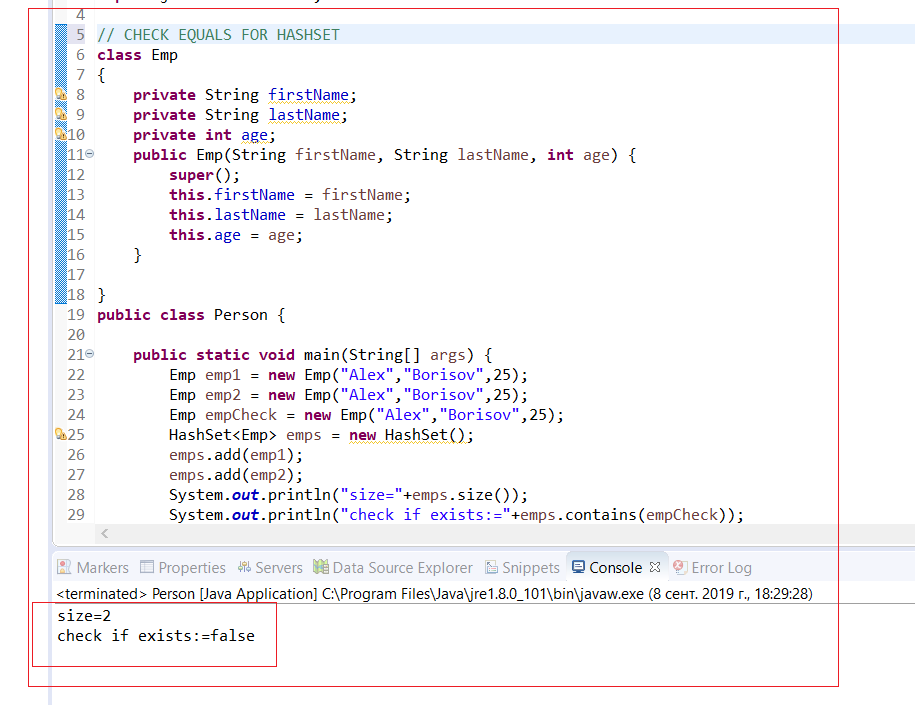
**Hashset does not allow duplicates. However, if you only override equals and not hashcode you will get not correct result with duplicates**

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

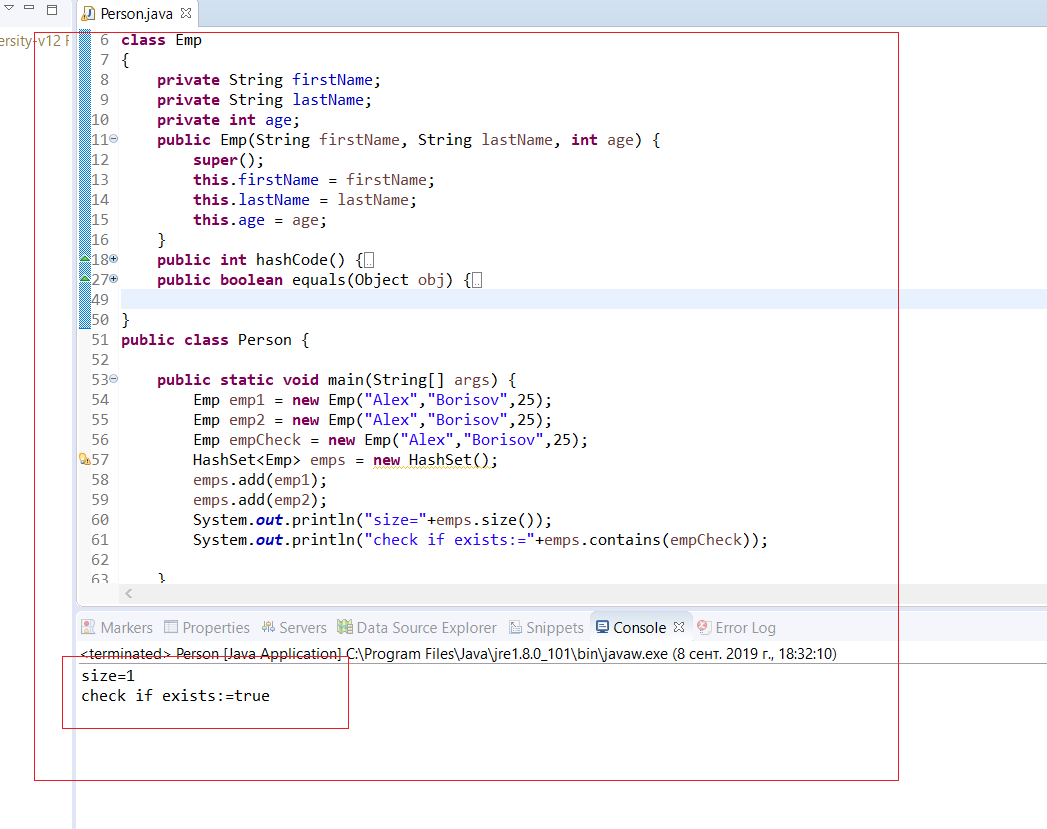
**However, because emp1 and emp2 has different hashcode that’s why HashSet keeps them in different buckets and thus allow duplicates.**

**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**



## **CompareTo method**

The **compareTo()** method compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The method returns 0 if the string is equal to the other string. A value less than 0 is returned if the string is less than the other string (less characters) and a value greater than 0 if the string is greater than the other string (more characters).

**note: Use the equals() method to compare two strings without consideration of Unicode values.**

# 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()** | Destructor. This method is called before the object will be called destroyed by Garbage Collector. It all happens under the hood |
| 6 | **clone()** |  |
| 7 | **wait()** |  |
| 8 | **notify()** |  |
| 9 | **notifyAll()** |  |

**[toString()]**

The default implementation is not in readable format (hashcode – checksum)

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

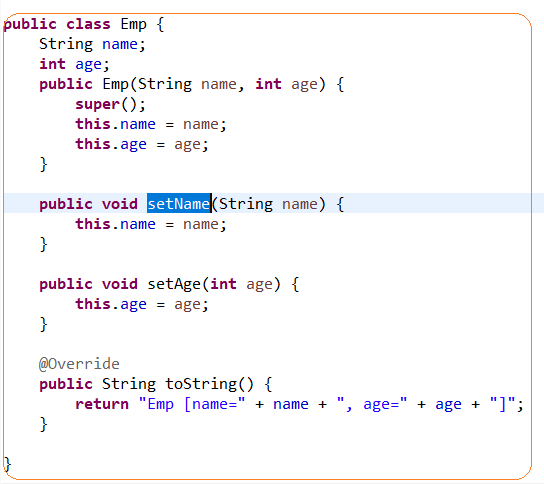
**[clone()]**

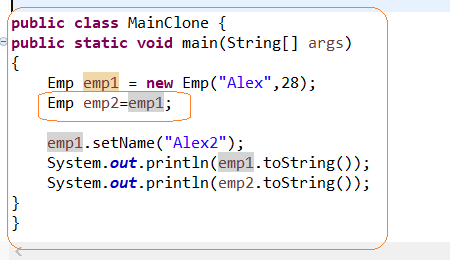
It creates exact the same copy of object and initializes all its fields. *To clone object you need to implement [Cloneable] interface*

[Cloneable] interface does not have any methods. [Cloneable] interface is already implemented into [Object] class. However, you can not directly call method [clone] on your class. You need to implement interface [Cloneable] in order to be able to call method [clone]

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

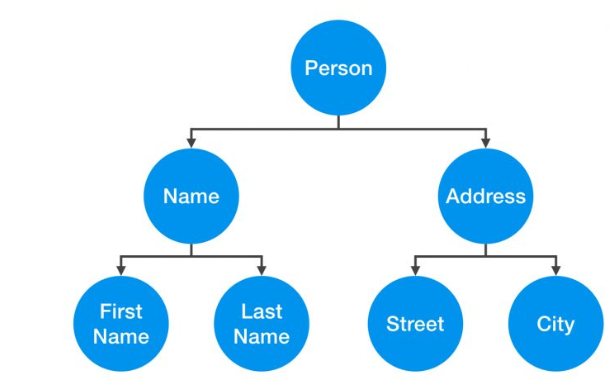


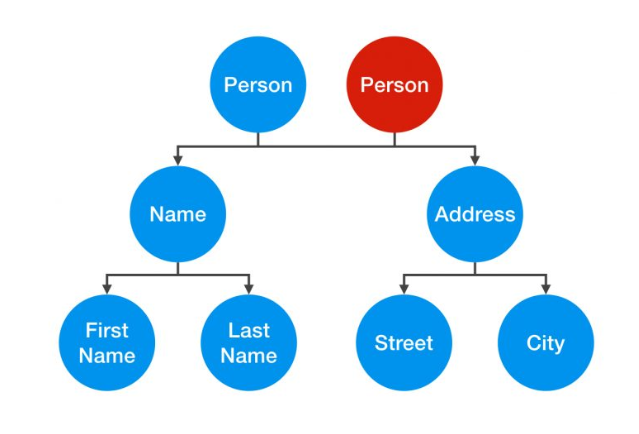


There are 2 copies of clone

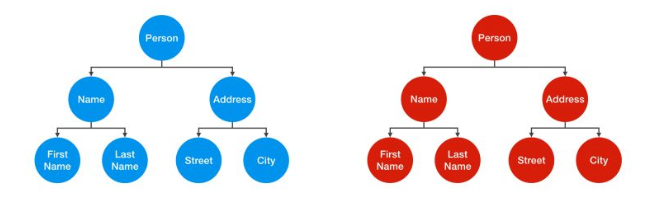
**Shallow copy** (default implementation of clone method) - it copies “main” object, but does not copy “inner” objects. If you modify inner object in reference object then changes will be reflected in first object. The problem of [shallow copy] is that two objects are not completely independent

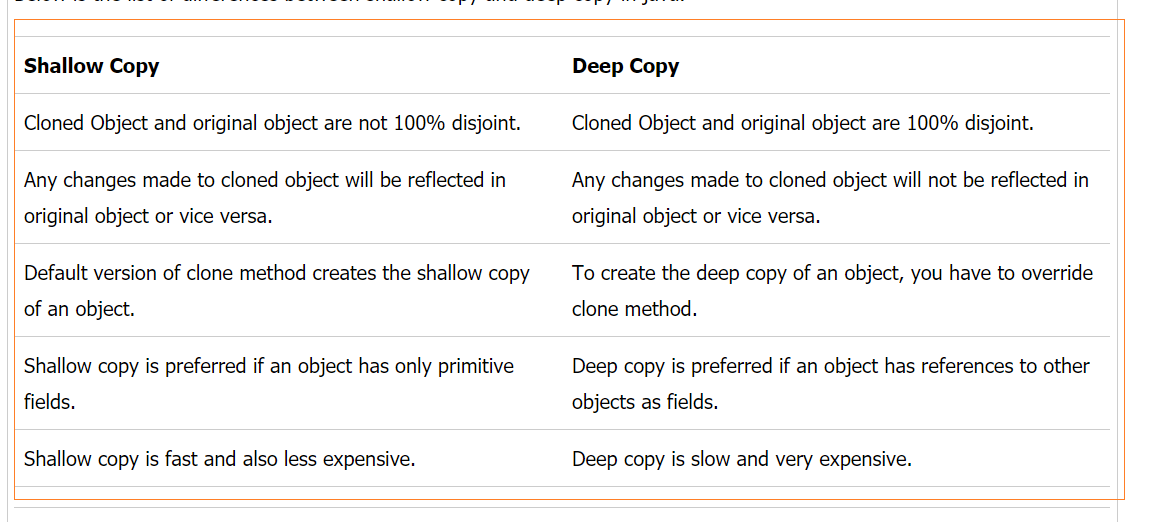
For example, Class [Person] consists of 2 classes [Address] and [Name]. when we clone “inner” classes will be reference to one Object in memory





**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]





# **WAIT(), NOTIFY(), NOTIFYALL()**

WAIT(), NOTIFY(), NOTIFYALL() are used for multi threading

[synchronized] key word. It means if method marked as synchronized it can receive only one thread

<https://javahungry.blogspot.com/2018/09/top-50-java-string-interview-questions-and-answers.html>