# Generics and Collections in Java programming language

**What are generics?**

According to Oracle java document website:

*In a nutshell, generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar formal parameters used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types*

Generics were added in Java version 5.0. This is was implemented with an intention to allow a type (like Strings, Integers, custom objects etc.) or method to operate on different types. Generics also offers compile time safety. It catches invalid or mismatched types at the compilation. For example let us take a look at the code below:



Figure1: Casting required as no generics is used



Figure 2: No casting required as generics is used

Generics enables programmers to enable write code to implement generic algorithms to work on different collections or data structures. Generics are not only safe they are also easy to interpret and read the code. The generic method declaration have a place for type parameter sections are written within angular brackets. (< / > ). Generics can be written by comma separated.

*Before going more into generics and collections and their usages we need to have some fair idea on some the common methods requires. These are described below.*

**The toString() method:**

This method belongs to the Object class i.e. the top level class and so this method is available to all the classes in the Java. The implementation of the *toString()* method in implemented as below:



toString() method in Java Object class

As we can see if we use the toString() method of the object class we will get the class name with some alphanumeric hexadecimal expression of the object’s hashcode. We will get some information but in reality gives very little human readable information. To get some information regarding the object the toString() methods needs to be overridden in every class or at least in the classes whose information might be required in future. Let us see the following examples below to see the differences between overridden toString method and normal Object class toString method.



toString() method not overridden

The output of the above code will be something like this: *PlayerClass@48987* (The number may vary).

But if we override the toString() method we will modify the code like below:



toString() method overridden by the class/object

The output of the above code will be:

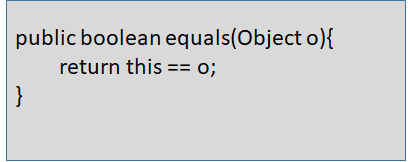
*The class name is PlayerClass and the player’s name is Lionel Messi. (It will also work similarly for C Ronaldo).*

Now we have got some information about the class which are useful to a human person. We can override the toString() method and can as many information as we need to pass.

Thus it is important and a good coding style to override the toString() method in a class so that if required some proper, human readable information can be get from the class.

**The equals(Object o) method:**

This method belongs to the object class and is accessible by any class. The implementation to the method in the Object class is shown in below:



We can see here basically, the method is only using the operator “==” check and returns the value. If the reference variable is same or refers to same object it returns true else false. It checks the bits in the variable and returns the Boolean value accordingly i.e. if they are identical true is returned or else it returns false.

“==” are used if we want to know whether two reference variables are equal or not.

equal() method is used on objects to check whether the contents of the two objects are same or not.

So if we use the Object#equals(Object o) method to check two objects it basically performs the action of “==” operator. The whole objective of the equals(Object o) method gets invalid. So we need to override the method in each class to get the proper result. Let’s see the following codes how with and without overriding the method can change the output.



The main class



The sample class without overriding the equals(Object o)

The output will be as followed:

*sample1 & sample2 equals(): false*

*sample1 & sample2 == operator false*

Now, let us override the toString method in the SampleClass and run the program again. The code will become as shown below:



toString() method overridden

The output will be as followed:

sample1 & sample2 equals(): true

sample1 & sample2 == operator false

In the first case as we have not overridden the method equals(Object o) the equals value implemented the Object#toString() method and basically performed ‘==’ operation. Both the reference variable sample1 and sample2 creates similar objects but are different reference variable. So both the results came as false.

In the second case we have overridden the equals(Object 0) method which checks the contents of the objects now, instead of the reference variable. So we see the equal(Object o) checks the contents of both the objects and when it was found same it returns *true.*  But still the ‘==’ operator returns *false* as expected,since both are different reference variable and are not the same.

But, as we have seen in String class without even overriding the euals(Object o) method we can use them and also get the proper responses. That is because in String class, Java has already overridden method. This is the equals(Object o) method of string class.



euals(Object o) method in String class

Many such classes have internal implementation, but not all. So we need to be careful while using .equals(Object o) method on objects to get the required proper value.

*It gives us one important fact: if ‘==’ returns true .equals(Object o) will always be true, but the opposite doesn’t work that way.*

**The contract of equals(Object o) method**

* Reflexive: a.equals(a) will always returns true
* Symmetric: if a.equals(b) is true, b.equal(a) will always be true
* Transitive: if a.equals(b) is true and b.equals(c) is true then a.equals(c) will always be true

Consistent:If not information provided to the classes is changed it will always returns the same, no matter how many time it is called.

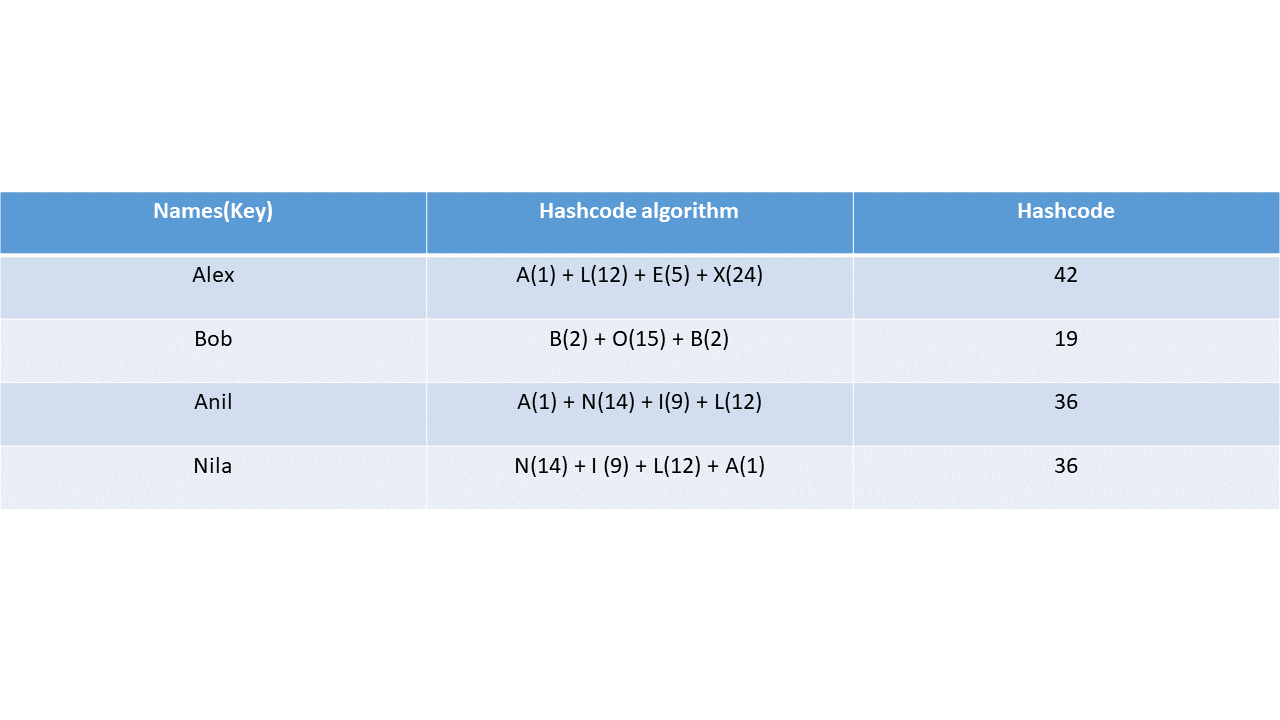
* a.equals(null) will always returns to null.

**The hashCode() method:**

This method belongs in the Object class so any class so any class can use and override it. The hashCode() method is used to increase the performance of searching in large quantities of data. Hashcodes are generated using a mathematical or hashcode algorithm and generating a output.

Hashcode may not are mostly unique for each object but they may not be unique in certain corner cases. Let’s have a look into it.

Consider we have few names like Alex, Bob, Anil and Nila. We want to implement hashcode algorithm to them. Our algorithm is to assign each letters their respective position number i.e. A=1, B=2, C=3 and so on. Then add those numbers and generate the hashcode.



In this table, we can see, even though we have 4 different hascodes. But two of them though are different in names have same hashcodes. In real scenarios hashing generally have more than one entry. One i.e. hashcode used to find first, then to use equals(Object o) to find the exact object. *But if two objects are equal, their hascodes must always be equal, the opposite is not true.*

Hashcodes can return same value for all objects or some objects depending upon its implementation. If a hascode function is written that it will return a constant value all the time, every objects will have same hashcode even though they are not the same. This makes hashcode not so useful and the equals(Object o) method finally will find out the exact object. This is an inefficient coding. *The goal is that two equal objects should have equal hashcodes.*

So to make an efficient program both the methods equals(Object o) and hashCode() should be properly overridden to fetch the exact object and also to make the fetching faster. For example if a hashcode is implemented improperly along with the equal(Object o) method (shown below), it will still be legal but extremely inefficient and improper.



Improper use of hashCode() and equals(Object o) methods

**The contract of hashCode() method:**

* In single execution the hascode must be same for same object. But it may vary in another execution of the process.
* If two objects are equals, i.e. equals(Object o) returns *true*, then their hashcode will definitely be *equal*.
* If two objects are *not* equals, i.e. equals(Object o) returns I, still their hashcode can be same.

## Collections in Java:

Collections, Collection and collection, these three words have three completely different interpretation in Java. So let us clear it out, before we dig into them.

1. collection (lower case c): Represents any data structures in which objects are stored, retrieved and iterated over.
2. Collection (Upper case c): It is an interface of java.lang.Collection. Set, Queue and List are extended from it. There is no direct implementation of Collection.
3. Collections (Upper case C and ends with s): It represents a class off java.lang.Collections. It contains a set of utility methods for the use of data structures or collections.

**Basic functions of Collection:**

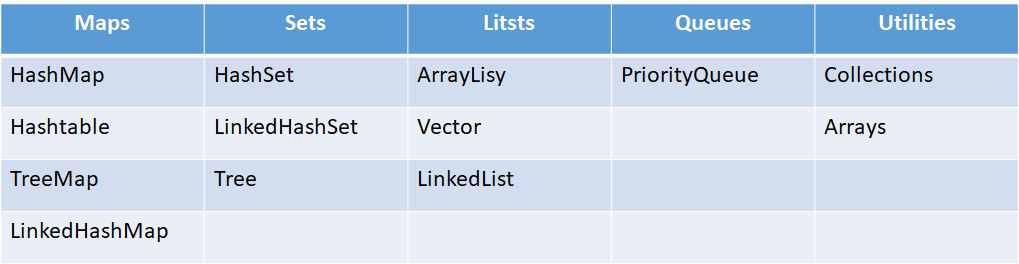
* Add an object to the collection
* Remove an object to the collection
* Find out, if an object is present in the collection
* Retrieve an object from the collection without deleting it.
* Iterating through the collection on each objects

**Important interfaces of the Collections framework and their concrete classes:**

1. **Interfaces:**

* Collection
* List
* Queue
* Set
* Map
* Navigable Set
* Navigable Map
* Sorted Set
* Sorted Map

1. **Concrete implementation of classes**

****

Each and every collection doesn’t implement the Collection interface. Utilities and maps have separate super classes. The following image will display the hierarchy structure of the Collections framework.

//TODO: Insert hierarchy image (if required)

Collections can be Lists, Queues, Maps or sets. These data structures can be sorted/unsorted or ordered/unordered.

Any above mentioned collections can be of the following three states:

1. Unsorted and unordered
2. Unsorted and ordered
3. Sorted and ordered

But it can never be sorted and unordered. Because sorted is a specific type of ordering. When an object is sorted by some parameter it is known as sorted and so automatically it gets ordered in a particular fashion.

**Ordered:**

* An ordered collections can be iterated in a non-random specific order. Generally by the index
* Hastable though internally maintains order is not an ordered collection as multiple calls of iteration can results in different outputs. Though in LinkedHashSet maintains order by insertion. The last element inserted is the last one.
* ArraryList is strictly ordered by the index of the object inserted.
* Some data structures uses natural ordering (like alphabetically or numerically) and they are also sorted.

**Sorted:**

* A sorted collection is a collection which maintains a order of the objects by some rule or rules known as sorted-rules.
* A sorted collection has nothing to do with the indexes, or the time of insertion, deletion or the position where it is inserted.
* A sorted collection depends on some parameter(s) of the object, to arrange them in a system i.e. make an order of them. It is done on the properties of the object itself.
* Sorting can be done using natural order mostly, but they can also be customized depending upon the requirements.