Краткое содержание: записывая пару ключ-значение (запись) в HashMap, мы помещаем ее в одну из ячеек (бакет) массива. То, в какую именно ячейку будет помещена запись (индекс ячейки в массиве) определяется по значению hashcode() ключа. Поскольку в один бакет может быть помещено несколько записей, то в бакет записываются не сами записи, а содержащие их связные списки.

В HashMap могут быть помещены записи только с уникальными ключами. Уникальность определяется методом equals(). То есть если

new Integer(1).equals(new Integer(1)); - вернул true - объекты равны.

Чтобы вся эта система корректно работала, необходимо соблюдать 2 правила (и пожалуйста, дальше - читать обязательно, потому что именно этот вопрос любят задавать на собеседованиях):

1. равные объекты должны иметь один и тот же hashcode

то есть, если equals - true, hashcode() возвращает одинаковое число

2. объекты, имеющие разный hashcode - неравны

то есть если hashcode() возвращает разные числа, equals обязано вернуть false

Почему соблюдать эти правила необходимо?

Легче всего это аргументировать, представив, что будет если эти правила будут нарушены.

Представим что у нас объект А и объект В - равны по equals, но возвращают разные hashcode. Мы добавляем запись с ключом-объектом А в HashMap. Далее мы пытаемся туда же добавить запись с ключом-объектом В. Что должно произойти? Поскольку А и В равны, вторая запись не должна добавиться. А что происходит на самом деле?

При добавлении второй записи высчитывается hashcode() для В, и по нему находится бакет, в котором предположительно будет находиться новая запись, если ключ В будет уникален. Для проверки уникальности вызывается метод equals() ключей записей, которые находятся в этом бакете. По счастливой случайности в этом бакете может находиться запись с ключом А, тогда все отработает корректно и вторая запись не добавится. Но вероятнее всего запись А будет в другом бакете, поэтому проверки покажут уникальность В. В результате мы будем иметь в HashMap с равными ключами А и В, что нарушает контракт HashMap.

Проблемы возникают и при поиске объектов. Ну а представить что в этом случае происходит - это уже на самостоятельную работу

Примечание: неравные объекты могут иметь одинаковый hashcode. В этом случае они хоть и попадают в один бакет, но все равно отсекаются при проверке объектов на equals().

Примечание 2: соблюдать правила взаимосвязь equals()-hashcode() необходимо, если вы переопределяете эти функции. В реализации по умолчанию (эти функции уже определены в базовом объекте - Object) эти правила уже соблюдены.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html>

**Override equals and hashCode in Java**

Equals and hashCode in Java are two fundamental method which is declared in Object class and part or core Java library. equals() method is used to [compare Objects](http://javarevisited.blogspot.sg/2011/06/comparator-and-comparable-in-java.html) for equality while hashCode is used to generate an integer code corresponding to that object. equals and hashCode has used extensively in Java core library like they are used while inserting and retrieving Object in HashMap, see [how get method of HashMap works in Java](http://java67.blogspot.com/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html) for full story, equals method is also used to avoid duplicates on HashSet and other Set implementation and every other place where you need to compare Objects. Default implementation of equals() class provided by java.lang.Object compares memory location and only return true if two reference variable are pointing to same memory location i.e. essentially they are same object. Java recommends to override equals and hashCode method if equality is going to be define by logical way or via some business logic and many classes in Java standard library does override it e.g. Stringoverrides equals,  whose implementation of equals() method return true if content of two String objects are exactly same. Integer wrapper class overrides equals to perform numerical comparison etc.

Since [HashMap and Hashtable](http://javarevisited.blogspot.sg/2010/10/difference-between-hashmap-and.html) in Java relies on equals() and hashCode() method for comparing keys and values, Java provides following rules to override equals method Java. As per following rule equals method in Java should be:

1) **Reflexive :** Object must be equal to itself.

2) **Symmetric :** if a.equals(b) is true then b.equals(a) must be true.

3) **Transitive :** if a.equals(b) is true and b.equals(c) is true then c.equals(a) must be true.

4) **Consistent :** multiple invocation of equals() method must result same value until any of properties are modified. So if two objects are equals in Java they will remain equals until any of there property is modified.

5) **Null comparison :** comparing any object to null must be false and should not result in NullPointerException. For example a.equals(null) must be false, passing unknown object, which could be null,  to equals in Java is is actually a Java coding [best practice to avoid NullPointerException in Java](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html).

## Equals and hashCode contract in Java

And equals method in Java must follow its contract with hashcode method in Java as stated below.

1) If two objects are equal by equals() method then there hashcode must be same.

2) If two objects are not equal by equals() method then there hashcode could be same or different.

So this was the basic theory about equals method in Java now we are going to discuss the approach on *how to override equals() method*, yes I know you all know this stuff :) but I have seen some of equals() code which can be improved by following correct approach. For illustration purpose we will see an example of Person class and discuss *How to write equals() method in Java* for that [class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html).

## Steps to Override equals method in Java

Here is my approach for overriding equals method in Java. This is based on standard approach most of Java programmer follows while writing equals method in Java.

1) Do [this](http://javarevisited.blogspot.sg/2012/01/this-keyword-java-example-tutorial.html) check -- if yes then return true.

2) Do null check -- if yes then return false.

3) Do the instanceof check,  if instanceof return false than return false from equals in Java , after some research I found that instead of instanceof we can use getClass() method for type identification because instanceof check returns true for subclass also, so its not strictly equals comparison until required by business logic. But instanceof check is fine if your class is [immutable](http://avdheshsemwal.blogspot.sg/2012/02/why-string-is-immutable-or-final-in.html) and no one is going to sub class it. For example we can replace instanceof check by below code

**if**((obj == **null**) || (obj.getClass() != **this**.getClass())) {

**return** **false**;

}

4) Type cast the object; note the sequence instanceof check must be prior to casting object.

5) Compare individual attribute starting with numeric attribute because comparing numeric attribute is fast and use short circuit operator for combining checks.  If first field does not match, don't try to match rest of attribute and return false. It’s also worth to remember doing null check on individual attribute before calling equals() method on them recursively to avoid NullPointerExceptionduring equals check in Java.

## Code Example of overriding equals method in Java

[equals and hashCode method in Java and Hibernate example tutorial](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html)Let’s see a code example based on my approach of overriding equals method in Java as discussed in above paragraph and hashCode() method is generated by Eclipse IDE, see my post  [5 tips to override hashcode in Java](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html) for detailed example and explanation of the right way to implement hashcode method.

/\*\*

\* Person class with equals and hashcode implementation in Java

\* @author Javin Paul

\*/

**public** **class** **Person** {

**private** **int** id;

**private** String firstName;

**private** String lastName;

**public** **int** **getId**() { **return** id; }

**public** **void** **setId**(**int** id) { **this**.id = id;}

**public** String **getFirstName**() { **return** firstName; }

**public** **void** **setFirstName**(String firstName) { **this**.firstName = firstName; }

**public** String **getLastName**() { **return** lastName; }

**public** **void** **setLastName**(String lastName) { **this**.lastName = lastName; }

**@Override**

**public** **boolean** **equals**(Object obj) {

**if** (obj == **this**) {

**return** **true**;

}

**if** (obj == **null** || obj.getClass() != **this**.getClass()) {

**return** **false**;

}

Person guest = (Person) obj;

**return** id == guest.id

&& (firstName == guest.firstName

|| (firstName != **null** && firstName.equals(guest.getFirstName())))

&& (lastName == guest.lastName

|| (lastName != **null** && lastName .equals(guest.getLastName())));

}

**@Override**

**public** **int** **hashCode**() {

**final** **int** prime = **31**;

**int** result = **1**;

result = prime \* result

+ ((firstName == **null**) ? **0** : firstName.hashCode());

result = prime \* result + id;

result = prime \* result

+ ((lastName == **null**) ? **0** : lastName.hashCode());

**return** result;

}

}

If you look above method we are first checking for "this" check which is fastest available check for equals method then we are verifying whether object is null or not and object is of same type or not. only after verifying type of object we are casting it into desired object to avoid any ClassCastException in Java. Also while comparing individual attribute we are comparing numeric attribute first using short circuit operator to avoid further calculation if its already unequal and doing null check on member attribute to avoid NullPointerException.

### Common Errors while overriding equals in Java

Though equals() and hashcode() method are defined in Object class along with [wait, notify and notifyAll](http://javarevisited.blogspot.sg/2012/02/why-wait-notify-and-notifyall-is.html),  and one of fundamental part of Java programming I have seen many programmers making mistake while writing equals() method in Java. I recommend all Java programmer who has just started programming to write couple of equals and hashcode method for there domain or value object to get feel of it. Here I am listing some of common mistakes I have observed on various equals method in Java, if you like to learn more about common mistakes in Java programming then see my post [Don’t use float and double for monetary calculation](http://javarevisited.blogspot.sg/2012/02/java-mistake-1-using-float-and-double.html) and [Mixing static and non static synchronized method](http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html). Now let’s see common mistakes by Java programmers while overriding equals in Java :

1) Instead of overriding equals() method programmer overloaded it.

This is the most common error I have seen while overriding equals method in Java. Syntax of equals method defined in Object class is public boolean equals(Object obj) but many people unintentionally overloads equals method in Java by writing public boolean equals(Person obj), instead of using Object as argument they use there class name. This error is very hard to detect because of [static binding](http://javarevisited.blogspot.sg/2012/03/what-is-static-and-dynamic-binding-in.html). So if you call this method in your class object it will not only compile but also execute correctly but if you try to put your object in collection e.g. ArrayList and call contains() method which is based on equals() method in Java it will not able to detect your object. So beware of it. This question is also a frequently asked question in Java interviews as part of Overloading vs Overriding in Java as how do you prevent this from happening ? Thankfully along-with Generics, Enum, autoboxing and varargs Java 5 also introduces @Override annotation which can be used to tell compiler that you are overriding a method and than compiler will be able to detect this error during compile time. Consistently using @Override annotation is also a best practice in Java.

2) Second mistake I have seen while overriding equals() method is not doing null check for member variables which ultimately results in [NullPointerExceptionin Java](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html)during equals() invocation. For example in above code correct way of calling equals() method of member variable is after doing null check as shown below:

firstname **==** guest**.**firstname **||** **(**firstname **!=** **null** **&**amp**;&**amp**;** firstname**.**equals**(**guest**.**firstname**)));**

3) Third common mistake is not overriding hashCode method in Java and only overriding equals() method**.** You must have to override both equals() and hashCode() method in Java , otherwise your value object will not be able to use as key object in HashMap because working of HashMap is based on equals() and hashCode to read more see , How HashMap works in Java.

4) Last common mistake programmer make while overriding equals() in Java is not keeping equals() and compareTo() method consistent which is a non formal requirement in order to obey contract of Set to avoid duplicates. SortedSet implementation like TreeSet uses compareTo to compare two objects like String and if compareTo() and equals() will not be consistent than TreeSet will allow duplicates which will break Set contract of not having duplicates. To learn more about this issue see my post [Things to remember while overriding compareTo in Java](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html)

### Writing JUnit tests for equals method in Java

Its good [coding practice](http://javarevisited.blogspot.sg/2011/09/code-review-checklist-best-practice.html) to write JUnit test cases to test your equals and hashCode method. Here is my approach for writing JUnit test case for equals method in Java. I will write test cases to check equals behaviour, contract of equals and hasCode method and properties of equals method in Java on different circumstances. You can also [JUnit4 annotation](http://javarevisited.blogspot.gr/2012/06/junit4-annotations-test-examples-and.html) to write JUnit test cases, than you don’t need to use test prefix on test method, just use @Test annotations.

testReflexive() this method will test reflexive nature of equals() method in Java.

testSymmeteric() this method will verify symmetric nature of equals() in Java.

testNull() this method will verify null comparison and will pass if equals method returns false.

testConsistent() should verify consistent nature of equals method in Java.

testNotEquals() should verify if two object which are not supposed to equals is actually not equal, having negative test cases in test suite is mandatory.

testHashCode() will verify that if two objects are equal by equals() method in Java then there hashcode must be same. This is an important test if you are thinking to use this object as [key in HashMap or Hashtable](http://javarevisited.blogspot.sg/2010/10/difference-between-hashmap-and.html)

## 5 Tips on writing equals method in Java

Here are some tips to implement equals and hashCode method in Java, this will help you to do it correctly and with ease:

1) Most of the IDE like NetBeans, Eclipse and IntelliJ IDEA provides support to generate equals() and hashcode() method. In Eclipse do the right click-> source -> generate hashCode() and equals().

2) If your domain class has any unique business key then just comparing that field in equals method would be enough instead of comparing all the fields e.g. in case of our example if "id" is unique for every Person and by just comparing id we can identify whether two Person are equal or not.

3) While [overriding hashCode in Java](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html) makes sure you use all fields which have been used in equals method in Java.

4) String and Wrapper classes like Integer, Float and Double override equals method but StringBuffer doesn’t override it.

5) Whenever possible try to make your fields immutable by using [final variables in Java](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html), equals method based on immutable fields are much secure than on mutable fields.

6) Don't use instanceof check in equals method, as it could break contract of equals() method in sub-class, results in non-symmetric equals, because instanceof return true for child class as well. For example, if you compare two objects Parent and Child from same type-hierarchy; Parent.equals(child) will return true, because child instanceof Parent is true, but Child.equals(parent) will return false, because Parent instanceof Child is false. This means equals() is not following symmetric contract, which states if a.equals(b) == true than b.equals(a) == true, as shown below :

**public** **class** Parent **{**

**}**

**public** **class** Child **extends** Parent **{**

**}**

**public** **class** InstanceOfCheck **{**

**public** **static** **void** main**(**String args**[])** **{**

Parent p **=** **new** Parent**();**

Child c **=** **new** Child**();**

System**.**out**.**println**(**"child instanceof Parent " **+** **(**c **instanceof** Parent**));**

System**.**out**.**println**(**"parent instanceof Child " **+** **(**p **instanceof** Child**));**

**}**

**}**

Output **:**

child **instanceof** Parent **true**

parent **instanceof** Child **false**

Alternatively, you can mark equals as final method to prevent it from being overridden.

7) While comparing String object in Java, [prefer equals() than == operator](http://javarevisited.blogspot.sg/2012/12/difference-between-equals-method-and-equality-operator-java.html).

8) Use IDE or Apache commons EqualsBuilder and HashCodeBuilder utility classes to automatically implement equals() and hashcode() method in Java.

9) Two object which is logically equal but loaded from different ClassLoader cannot be equals. Remember that getClass() check, it will return false if class loader is different.

10) Use [@Override annotation](http://javarevisited.blogspot.sg/2012/11/why-use-override-annotation-in-java.html) on hashcode() as well, as it also prevents subtle mistakes over return type. e.g. return type of hashcode() method is int, but many times programmers mistakenly put long.

11) One  example, where equals method is not consistent with compareTo is java.math.BigDecimal class. If you compare two BigDecimal object e.g. 120.00 and 120.000, equals method will return false, while compareTo will return zero. Both are inconsistent, because equals take both scale and value in consideration, while compareTo method only consider values.

12) From Java 7 you can also use a new utility class called java.util.Objects for null safe equality check and calculating hash code. You can replace our null-safe code for check equality :

**(**name **==** guest**.**name **||** **(**name **!=** **null** **&**amp**;&**amp**;** name**.**equals**(**guest**.**getName**())))**

to much concise

Objects**.**equals**(**name**,** guest**.**getName**());**

### Use of Equals and Hashcode Method in Hibernate

Hibernate is a popular, open source Java persistent framework, which provides Object Relational Mapping, also known as ORM framework. It uses equals and hashcode method to provide object's equality in Java side.  You should **override equals() and hashcode()** if :

1) You are storing instance of persistent class in a Set for representing many-valued associations.

2) You are using reattachment of detached persistent instances.

Another worth noting point is that Hibernate only guarantees equivalence of database row (persistent identity) and Java object inside a particular Session. Which means if you store instances retrieved in different Sessions in a Set, you will be having duplicates. Now the most important aspect of overriding equals and hashcode() for hibernate entity classes, you should never decide equality just based upon identifier. Though it’s convenient to compare identifier to see if the belong to same database row, Unfortunately, we can't use this approach with generated identifiers. Since Hibernate only assign identifier values to the object that are persistent, a newly created instance will not have any identifier value. Similarly, if an instance is not persisted, and currently in a Set, saving it to database will assigned an identifier value, which will further change the value of hashcode() method, finally results in breaking the contract of the Set. That's why it's best to implement equals and hashcode in Hibernate using business key equality e.g. an Employee is same if it's name, surname, father's name, department, date of birth is same. Properties which are not prone to change e.g. date of birth are better candidate of business equality than those which is easier to change e.g. address and contact number.

In short, remember these best practices while overriding equals() and hashcode() for Hibernate entity class :

1) Don't let your equals() method only uses identifier values for equivalence check.

2) Implement equals() and hashCode() using real word key that would identify instance in real world.

3) Use Immutable and unique properties of objects for equality.  
  
  
4) Don't use [getClass()](http://javarevisited.blogspot.sg/2012/09/how-to-determine-type-of-object-runtime-identification.html) to compare object equality because Hibernate uses proxy and this check will always fail. Instead use instanceof operator, it respect proxy because they have IS-A relationship with actual object.  
  
  
5) Use getter and setter methods to access properties instead of directly accessing the, because hibernate lazily initialize object, when there getProperty() method is called. Using name may return null but getName() may return value from database.

That’s all about **overriding equals() and hashcode() methods in Java**, I am reiterating this but its imperative for a Java programmer to be able to write equals , hashcode(), compareTo() method by hand. It is not just useful for learning purpose but to clear any coding exercise during Java interviews. Writing code for equals and hashcode is very popular programming interview questions now days. For Hibernate persistent class its rather tricky to override equals() and hashCode() because otherwise bad practices turns into best practices because of extensive of proxy. You should not use Eclipse IDE code generator for equals() and hashCode() for hibernate entity class, as they use getClass() to check type equality.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.com/2011/11/how-to-override-compareto-method-in.html>

How to override compareTo method in Java - Example Tutorial

**compareTo in Java** is in the same league of [equals()](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html) and [hashcode()](http://javarevisited.blogspot.com/2011/10/override-hashcode-in-java-example.html) and used to implement **natural order of object**, compareTo is slightly different to compare() method of Comparator interface which is used to implement custom sorting order. I have seen during [java interviews](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html) that many Java programmers not able to correctly write or implement equals(), hashCode() and compareTo() method for common business objects like Order or Employee. Simple reason behind this is that they either not understand the concept well enough or doesn't write this stuff at all. I will try to fill that gap in this Java tutorial and will see What is compareTo() method in java, **how to write compareTo in Java** and things to remember while implementing compareTo in Java.

## What is compareTo() method in Java

compareTo() method is defined in interface **java.lang.Comparable** and it is used to implement natural sorting on [java classes](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html). natural sorting means the the sort order which naturally applies on object e.g. lexical order for String, numeric order for Integer or Sorting employee by there ID etc. most of the java core classes including **String and Integer implements CompareTo()** method and provide natural sorting.

### Why do you need CompareTo()

[How to write compareTo method in Object - Java Example](http://3.bp.blogspot.com/-K6q0DQ1v-tw/TWu8owBtc2I/AAAAAAAAADA/oBoHDBiJ8ag/s1600/17.jpg)Sorting is an essential part of application development, which you often required to implement in your system. in Java sorting is implemented using [Comparator and Comparable in Java](http://javarevisited.blogspot.com/2011/06/comparator-and-comparable-in-java.html). Since we store java objects in Collection there are also certain Set and Map which provides automating sorting when you insert element on that e.g. TreeSet and [TreeMap](http://javarevisited.blogspot.com/2011/12/treemap-java-tutorial-example-program.html). to implement sorting you need to override either compareTo(Object o) method or Comparable class or compare(Object o1, Object o2) method of Comparator class. Most of the classes implement Comparable to implement natural order. for example if you are writing Employee object you probably want to implement Comparable interface and override compareTo() method to compare current employee with other employee based on ID. So essentially you need to override compareTo() because you need to [sort elements in ArrayList](http://javarevisited.blogspot.com/2012/01/how-to-sort-arraylist-in-java-example.html) or any other Collection.

## How to implement compareTo in Java

There are certain rules and important points to remember while overriding compareTo method:

1) CompareTo method must return negative number if current object is less than other object, positive number if current object is greater than other object and zero if both objects are equal to each other.

2) CompareTo must be in consistent with [equals method](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html) e.g. if two objects are equal via equals() , there compareTo() must return zero otherwise if those objects are stored in SortedSet or SortedMap they will not behave properly. Since SortedSet or SortedMap use compareTo() to check the object if two unequal object are returned equal by compareTo those will not be added into Set or Map if they are not using external Comparator.  One example where compareTo is not consistent with equals in JDK is BigDecimal class. two BigDecimal number for which compareTo returns zero, equals returns false as clear from following BigDecimal comparison example:

**BigDecimal** bd1 = **new** **BigDecimal**("2.0");  
**BigDecimal** bd2 = **new** **BigDecimal**("2.00");  
        
**System**.out.println("comparing BigDecimal using equals: " + bd1.equals(bd2));  
**System**.out.println("comparing BigDecimal using compareTo: " + bd1.compareTo(bd2));  
  
Output:  
comparing **BigDecimal** using equals: **false**  
comparing **BigDecimal** using compareTo: 0

How does it affect BigDecimal ? well if you store these two BigDecimal in HashSet you will end up with duplicates (violation of Set Contract) i.e. two elements while if you store them in TreeSet you will end up with just 1 element because HashSet uses equals to check duplicates while TreeSet uses compareTo to check duplicates. That's why its suggested to keep **compareTo consistent with equals method in java**.

3) CompareTo() must throw NullPointerException if current object get compared to null object as opposed to equals() which return false on such scenario.

4) Another important point to note is **don't use subtraction for comparing integral values** because result of subtraction can overflow as every int operation in Java is modulo 2^32. use either Integer.compareTo()  or logical operators for comparison. There is one scenario where you can use subtraction to reduce clutter and improve performance. As we know compareTo doesn't care magnitude, it just care whether result is positive or negative. While comparing two integral fields you can use subtraction if you are absolutely sure that both operands are positive integer or more precisely there different must be less than Integer.MAX\_VALUE. In this case there will be no overflow and your compareTo will be concise and faster.

5. Use relational operator to compare integral numeric value i.e. < or > but use Float.compareTo() or Double.compareTo() to compare [floating point number](http://javarevisited.blogspot.com/2011/10/convert-double-to-string-example.html) as relational operator doesn't obey contract of compareTo for floating point numbers.

6. CompareTo() method is for comparison so **order in which you compare two object matters**. If you have more than one significant field to compare than always *start comparing from most significant field* to least significant field. here **compareTo is different with equals** because in case of equality check order doesn't matter. like in above *example of compareTo* if we don't consider Id and compare two student by its name and age than name should be first compare and than age, so if two student have same name one that has higher age should result in greater.

Student john12 = **new** Student(1001, "John", 12);  
Student john13 = **new** Student(1002, "John", 13);  
        
*//compareTo will return -1 as age of john12 is less than john13*  
**System**.out.println("comparing John, 12 and John, 13 with compareTo :" + john12.compareTo(john13));  
  
Output:  
comparing John, 12 and John, 13 with compareTo :-1

7. Another important point while comparing String using compareTo is to consider case. just like equals() doesn't consider case, compareTo also do not consider case, if you want to compare regardless of case than use String.compareToIgnoreCase() as we have used in above example.

**Where compareTo() method used in Java**

---------------------------------------------------

In Java API compareTo() method is used in SortedSet e.g. TreeSet and SortedMap e.g. TreeMap for sorting elements on natural order if no explicit Comparator is passed to Collections.sort() method e.g.

[**List**](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List) stocks = getListOfStocks();   
[**Collections**](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collections.html).sort(stocks);

as mentioned earlier if compareTo is not consistent with equals then it could produce strange result. let took another example you put Stock A and Stock B on StockSet which is a TreeSet. Both Stock A and Stock B are equal by equals() method but compareTo return non zero values for it which makes that StockB will also be landed into TreeSet which was voilation of Set itself because it is not supposed to allow duplicates.

### Example of compareTo() in Java --------------------------------------

Let’s see an example of **how to override compareTo method in Java**. This method is very similar to equals and hashcode, key thing is compareTo should provide **natural ordering** e.g. in this example order of object based on Student ID.

**public** **class** Student **implements** [**Comparable**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html) {   
    **private** **int** id;  
    **private** **String** name;  
    **private** **int** age;  
    
    */\*  
     \*Compare a given Student with current(this) object.  
     \*If current Student id is greater than the received object,  
     \*then current object is greater than the other.  
     \*/*     
    **public** **int** compareTo(Student otherStudent) {  
       *// return this.id - otherStudent.id ; //result of this operation can overflow*  
       **return** (**this**.id &lt; otherStudent.id ) ? -1: (**this**.id &gt; otherStudent.id) ? 1:0 ;  
  
    }   
}

here is another example of compareTo method in Java on which compareTo uses two significant field to compare objects:

**public** **class** Student **implements** [**Comparable**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html)<Student> {  
   .....      
    ***/\*\*  
     \* Compare a given Student with current(this) object.  
     \* first compare name and than age  
     \*/***  
    @**Override**  
    **public** **int** compareTo(Student otherStudent) {         
        *//compare name*  
        **int** nameDiff = name.compareToIgnoreCase(otherStudent.name);  
        **if**(nameDiff != 0){  
            **return** nameDiff;  
        }  
        *//names are equals compare age*  
        **return** age - otherStudent.age;  
    }  
    
}

That’s all on implementing compareTo method in Java. Please add any other fact which you think important to note while overriding compareTo. In summary compareTo should provide natural ordering and compareTo must be consistent with equals() method in Java.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html>

How to override compareTo method in Java - Example Tutorial

**compareTo in Java** is in the same league of [equals()](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html) and [hashcode()](http://javarevisited.blogspot.com/2011/10/override-hashcode-in-java-example.html) and used to implement **natural order of object**, compareTo is slightly different to compare() method of Comparator interface which is used to implement custom sorting order. I have seen during [java interviews](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html) that many Java programmers not able to correctly write or implement equals(), hashCode() and compareTo() method for common business objects like Order or Employee. Simple reason behind this is that they either not understand the concept well enough or doesn't write this stuff at all. I will try to fill that gap in this Java tutorial and will see What is compareTo() method in java, **how to write compareTo in Java** and things to remember while implementing compareTo in Java.

## What is compareTo() method in Java

compareTo() method is defined in interface **java.lang.Comparable** and it is used to implement natural sorting on [java classes](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html). natural sorting means the the sort order which naturally applies on object e.g. lexical order for String, numeric order for Integer or Sorting employee by there ID etc. most of the java core classes including **String and Integer implements CompareTo()** method and provide natural sorting.

### Why do you need CompareTo()

[How to write compareTo method in Object - Java Example](http://3.bp.blogspot.com/-K6q0DQ1v-tw/TWu8owBtc2I/AAAAAAAAADA/oBoHDBiJ8ag/s1600/17.jpg)Sorting is an essential part of application development, which you often required to implement in your system. in Java sorting is implemented using [Comparator and Comparable in Java](http://javarevisited.blogspot.com/2011/06/comparator-and-comparable-in-java.html). Since we store java objects in Collection there are also certain Set and Map which provides automating sorting when you insert element on that e.g. TreeSet and [TreeMap](http://javarevisited.blogspot.com/2011/12/treemap-java-tutorial-example-program.html). to implement sorting you need to override either compareTo(Object o) method or Comparable class or compare(Object o1, Object o2) method of Comparator class. Most of the classes implement Comparable to implement natural order. for example if you are writing Employee object you probably want to implement Comparable interface and override compareTo() method to compare current employee with other employee based on ID. So essentially you need to override compareTo() because you need to [sort elements in ArrayList](http://javarevisited.blogspot.com/2012/01/how-to-sort-arraylist-in-java-example.html) or any other Collection.

## How to implement compareTo in Java

There are certain rules and important points to remember while overriding compareTo method:

1) CompareTo method must return negative number if current object is less than other object, positive number if current object is greater than other object and zero if both objects are equal to each other.

2) CompareTo must be in consistent with [equals method](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html) e.g. if two objects are equal via equals() , there compareTo() must return zero otherwise if those objects are stored in SortedSet or SortedMap they will not behave properly. Since SortedSet or SortedMap use compareTo() to check the object if two unequal object are returned equal by compareTo those will not be added into Set or Map if they are not using external Comparator.  One example where compareTo is not consistent with equals in JDK is BigDecimal class. two BigDecimal number for which compareTo returns zero, equals returns false as clear from following BigDecimal comparison example:

**BigDecimal** bd1 = **new** **BigDecimal**("2.0");  
**BigDecimal** bd2 = **new** **BigDecimal**("2.00");  
        
**System**.out.println("comparing BigDecimal using equals: " + bd1.equals(bd2));  
**System**.out.println("comparing BigDecimal using compareTo: " + bd1.compareTo(bd2));  
  
Output:  
comparing **BigDecimal** using equals: **false**  
comparing **BigDecimal** using compareTo: 0

How does it affect BigDecimal ? well if you store these two BigDecimal in HashSet you will end up with duplicates (violation of Set Contract) i.e. two elements while if you store them in TreeSet you will end up with just 1 element because HashSet uses equals to check duplicates while TreeSet uses compareTo to check duplicates. That's why its suggested to keep **compareTo consistent with equals method in java**.

3) CompareTo() must throw NullPointerException if current object get compared to null object as opposed to equals() which return false on such scenario.

4) Another important point to note is **don't use subtraction for comparing integral values** because result of subtraction can overflow as every int operation in Java is modulo 2^32. use either Integer.compareTo()  or logical operators for comparison. There is one scenario where you can use subtraction to reduce clutter and improve performance. As we know compareTo doesn't care magnitude, it just care whether result is positive or negative. While comparing two integral fields you can use subtraction if you are absolutely sure that both operands are positive integer or more precisely there different must be less than Integer.MAX\_VALUE. In this case there will be no overflow and your compareTo will be concise and faster.

5. Use relational operator to compare integral numeric value i.e. < or > but use Float.compareTo() or Double.compareTo() to compare [floating point number](http://javarevisited.blogspot.com/2011/10/convert-double-to-string-example.html) as relational operator doesn't obey contract of compareTo for floating point numbers.

6. CompareTo() method is for comparison so **order in which you compare two object matters**. If you have more than one significant field to compare than always *start comparing from most significant field* to least significant field. here **compareTo is different with equals** because in case of equality check order doesn't matter. like in above *example of compareTo* if we don't consider Id and compare two student by its name and age than name should be first compare and than age, so if two student have same name one that has higher age should result in greater.

Student john12 = **new** Student(1001, "John", 12);  
Student john13 = **new** Student(1002, "John", 13);  
        
*//compareTo will return -1 as age of john12 is less than john13*  
**System**.out.println("comparing John, 12 and John, 13 with compareTo :" + john12.compareTo(john13));  
  
Output:  
comparing John, 12 and John, 13 with compareTo :-1

7. Another important point while comparing String using compareTo is to consider case. just like equals() doesn't consider case, compareTo also do not consider case, if you want to compare regardless of case than use String.compareToIgnoreCase() as we have used in above example.

**Where compareTo() method used in Java**

---------------------------------------------------

In Java API compareTo() method is used in SortedSet e.g. TreeSet and SortedMap e.g. TreeMap for sorting elements on natural order if no explicit Comparator is passed to Collections.sort() method e.g.

[**List**](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List) stocks = getListOfStocks();   
[**Collections**](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collections.html).sort(stocks);

as mentioned earlier if compareTo is not consistent with equals then it could produce strange result. let took another example you put Stock A and Stock B on StockSet which is a TreeSet. Both Stock A and Stock B are equal by equals() method but compareTo return non zero values for it which makes that StockB will also be landed into TreeSet which was voilation of Set itself because it is not supposed to allow duplicates.

### Example of compareTo() in Java --------------------------------------

Let’s see an example of **how to override compareTo method in Java**. This method is very similar to equals and hashcode, key thing is compareTo should provide **natural ordering** e.g. in this example order of object based on Student ID.

**public** **class** Student **implements** [**Comparable**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html) {   
    **private** **int** id;  
    **private** **String** name;  
    **private** **int** age;  
    
    */\*  
     \*Compare a given Student with current(this) object.  
     \*If current Student id is greater than the received object,  
     \*then current object is greater than the other.  
     \*/*     
    **public** **int** compareTo(Student otherStudent) {  
       *// return this.id - otherStudent.id ; //result of this operation can overflow*  
       **return** (**this**.id &lt; otherStudent.id ) ? -1: (**this**.id &gt; otherStudent.id) ? 1:0 ;  
  
    }   
}

here is another example of compareTo method in Java on which compareTo uses two significant field to compare objects:

**public** **class** Student **implements** [**Comparable**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html)<Student> {  
   .....      
    ***/\*\*  
     \* Compare a given Student with current(this) object.  
     \* first compare name and than age  
     \*/***  
    @**Override**  
    **public** **int** compareTo(Student otherStudent) {         
        *//compare name*  
        **int** nameDiff = name.compareToIgnoreCase(otherStudent.name);  
        **if**(nameDiff != 0){  
            **return** nameDiff;  
        }  
        *//names are equals compare age*  
        **return** age - otherStudent.age;  
    }  
    
}

That’s all on implementing compareTo method in Java. Please add any other fact which you think important to note while overriding compareTo. In summary compareTo should provide natural ordering and compareTo must be consistent with equals() method in Java.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html>

Overriding equals() and hashCode() method in Java and Hibernate

**Override equals and hashCode in Java**

Equals and hashCode in Java are two fundamental method which is declared in Object class and part or core Java library. equals() method is used to [compare Objects](http://javarevisited.blogspot.sg/2011/06/comparator-and-comparable-in-java.html) for equality while hashCode is used to generate an integer code corresponding to that object. equals and hashCode has used extensively in Java core library like they are used while inserting and retrieving Object in HashMap, see [how get method of HashMap works in Java](http://java67.blogspot.com/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html) for full story, equals method is also used to avoid duplicates on HashSet and other Set implementation and every other place where you need to compare Objects. Default implementation of equals() class provided by java.lang.Object compares memory location and only return true if two reference variable are pointing to same memory location i.e. essentially they are same object. Java recommends to override equals and hashCode method if equality is going to be define by logical way or via some business logic and many classes in Java standard library does override it e.g. Stringoverrides equals,  whose implementation of equals() method return true if content of two String objects are exactly same. Integer wrapper class overrides equals to perform numerical comparison etc.

Since [HashMap and Hashtable](http://javarevisited.blogspot.sg/2010/10/difference-between-hashmap-and.html) in Java relies on equals() and hashCode() method for comparing keys and values, Java provides following rules to override equals method Java. As per following rule equals method in Java should be:

1) **Reflexive :** Object must be equal to itself.

2) **Symmetric :** if a.equals(b) is true then b.equals(a) must be true.

3) **Transitive :** if a.equals(b) is true and b.equals(c) is true then c.equals(a) must be true.

4) **Consistent :** multiple invocation of equals() method must result same value until any of properties are modified. So if two objects are equals in Java they will remain equals until any of there property is modified.

5) **Null comparison :** comparing any object to null must be false and should not result in NullPointerException. For example a.equals(null) must be false, passing unknown object, which could be null,  to equals in Java is is actually a Java coding [best practice to avoid NullPointerException in Java](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html).

## Equals and hashCode contract in Java

And equals method in Java must follow its contract with hashcode method in Java as stated below.

1) If two objects are equal by equals() method then there hashcode must be same.

2) If two objects are not equal by equals() method then there hashcode could be same or different.

So this was the basic theory about equals method in Java now we are going to discuss the approach on *how to override equals() method*, yes I know you all know this stuff :) but I have seen some of equals() code which can be improved by following correct approach. For illustration purpose we will see an example of Person class and discuss *How to write equals() method in Java* for that [class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html).

## Steps to Override equals method in Java

Here is my approach for overriding equals method in Java. This is based on standard approach most of Java programmer follows while writing equals method in Java.

1) Do [this](http://javarevisited.blogspot.sg/2012/01/this-keyword-java-example-tutorial.html) check -- if yes then return true.

2) Do null check -- if yes then return false.

3) Do the instanceof check,  if instanceof return false than return false from equals in Java , after some research I found that instead of instanceof we can use getClass() method for type identification because instanceof check returns true for subclass also, so its not strictly equals comparison until required by business logic. But instanceof check is fine if your class is [immutable](http://avdheshsemwal.blogspot.sg/2012/02/why-string-is-immutable-or-final-in.html) and no one is going to sub class it. For example we can replace instanceof check by below code

**if**((obj == **null**) || (obj.getClass() != **this**.getClass())) {

**return** **false**;

}

4) Type cast the object; note the sequence instanceof check must be prior to casting object.

5) Compare individual attribute starting with numeric attribute because comparing numeric attribute is fast and use short circuit operator for combining checks.  If first field does not match, don't try to match rest of attribute and return false. It’s also worth to remember doing null check on individual attribute before calling equals() method on them recursively to avoid NullPointerExceptionduring equals check in Java.

## Code Example of overriding equals method in Java

Let’s see a code example based on my approach of overriding equals method in Java as discussed in above paragraph and hashCode() method is generated by Eclipse IDE, see my post  [5 tips to override hashcode in Java](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html) for detailed example and explanation of the right way to implement hashcode method.

/\*\*

\* Person class with equals and hashcode implementation in Java

\* @author Javin Paul

\*/

**public** **class** **Person** {

**private** **int** id;

**private** String firstName;

**private** String lastName;

**public** **int** **getId**() { **return** id; }

**public** **void** **setId**(**int** id) { **this**.id = id;}

**public** String **getFirstName**() { **return** firstName; }

**public** **void** **setFirstName**(String firstName) { **this**.firstName = firstName; }

**public** String **getLastName**() { **return** lastName; }

**public** **void** **setLastName**(String lastName) { **this**.lastName = lastName; }

**@Override**

**public** **boolean** **equals**(Object obj) {

**if** (obj == **this**) {

**return** **true**;

}

**if** (obj == **null** || obj.getClass() != **this**.getClass()) {

**return** **false**;

}

Person guest = (Person) obj;

**return** id == guest.id

&& (firstName == guest.firstName

|| (firstName != **null** && firstName.equals(guest.getFirstName())))

&& (lastName == guest.lastName

|| (lastName != **null** && lastName .equals(guest.getLastName())));

}

**@Override**

**public** **int** **hashCode**() {

**final** **int** prime = **31**;

**int** result = **1**;

result = prime \* result

+ ((firstName == **null**) ? **0** : firstName.hashCode());

result = prime \* result + id;

result = prime \* result

+ ((lastName == **null**) ? **0** : lastName.hashCode());

**return** result;

}

}

If you look above method we are first checking for "this" check which is fastest available check for equals method then we are verifying whether object is null or not and object is of same type or not. only after verifying type of object we are casting it into desired object to avoid any ClassCastException in Java. Also while comparing individual attribute we are comparing numeric attribute first using short circuit operator to avoid further calculation if its already unequal and doing null check on member attribute to avoid NullPointerException.

### Common Errors while overriding equals in Java

Though equals() and hashcode() method are defined in Object class along with [wait, notify and notifyAll](http://javarevisited.blogspot.sg/2012/02/why-wait-notify-and-notifyall-is.html),  and one of fundamental part of Java programming I have seen many programmers making mistake while writing equals() method in Java. I recommend all Java programmer who has just started programming to write couple of equals and hashcode method for there domain or value object to get feel of it. Here I am listing some of common mistakes I have observed on various equals method in Java, if you like to learn more about common mistakes in Java programming then see my post [Don’t use float and double for monetary calculation](http://javarevisited.blogspot.sg/2012/02/java-mistake-1-using-float-and-double.html) and [Mixing static and non static synchronized method](http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html). Now let’s see common mistakes by Java programmers while overriding equals in Java :

1) Instead of overriding equals() method programmer overloaded it.

This is the most common error I have seen while overriding equals method in Java. Syntax of equals method defined in Object class is public boolean equals(Object obj) but many people unintentionally overloads equals method in Java by writing public boolean equals(Person obj), instead of using Object as argument they use there class name. This error is very hard to detect because of [static binding](http://javarevisited.blogspot.sg/2012/03/what-is-static-and-dynamic-binding-in.html). So if you call this method in your class object it will not only compile but also execute correctly but if you try to put your object in collection e.g. ArrayList and call contains() method which is based on equals() method in Java it will not able to detect your object. So beware of it. This question is also a frequently asked question in Java interviews as part of Overloading vs Overriding in Java as how do you prevent this from happening ? Thankfully along-with Generics, Enum, autoboxing and varargs Java 5 also introduces @Override annotation which can be used to tell compiler that you are overriding a method and than compiler will be able to detect this error during compile time. Consistently using @Override annotation is also a best practice in Java.

2) Second mistake I have seen while overriding equals() method is not doing null check for member variables which ultimately results in [NullPointerExceptionin Java](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html)during equals() invocation. For example in above code correct way of calling equals() method of member variable is after doing null check as shown below:

firstname **==** guest**.**firstname **||** **(**firstname **!=** **null** **&**amp**;&**amp**;** firstname**.**equals**(**guest**.**firstname**)));**

3) Third common mistake is not overriding hashCode method in Java and only overriding equals() method**.** You must have to override both equals() and hashCode() method in Java , otherwise your value object will not be able to use as key object in HashMap because working of HashMap is based on equals() and hashCode to read more see , How HashMap works in Java.

4) Last common mistake programmer make while overriding equals() in Java is not keeping equals() and compareTo() method consistent which is a non formal requirement in order to obey contract of Set to avoid duplicates. SortedSet implementation like TreeSet uses compareTo to compare two objects like String and if compareTo() and equals() will not be consistent than TreeSet will allow duplicates which will break Set contract of not having duplicates. To learn more about this issue see my post [Things to remember while overriding compareTo in Java](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html)

### Writing JUnit tests for equals method in Java

Its good [coding practice](http://javarevisited.blogspot.sg/2011/09/code-review-checklist-best-practice.html) to write JUnit test cases to test your equals and hashCode method. Here is my approach for writing JUnit test case for equals method in Java. I will write test cases to check equals behaviour, contract of equals and hasCode method and properties of equals method in Java on different circumstances. You can also [JUnit4 annotation](http://javarevisited.blogspot.gr/2012/06/junit4-annotations-test-examples-and.html) to write JUnit test cases, than you don’t need to use test prefix on test method, just use @Test annotations.

testReflexive() this method will test reflexive nature of equals() method in Java.

testSymmeteric() this method will verify symmetric nature of equals() in Java.

testNull() this method will verify null comparison and will pass if equals method returns false.

testConsistent() should verify consistent nature of equals method in Java.

testNotEquals() should verify if two object which are not supposed to equals is actually not equal, having negative test cases in test suite is mandatory.

testHashCode() will verify that if two objects are equal by equals() method in Java then there hashcode must be same. This is an important test if you are thinking to use this object as [key in HashMap or Hashtable](http://javarevisited.blogspot.sg/2010/10/difference-between-hashmap-and.html)

## 5 Tips on writing equals method in Java

Here are some tips to implement equals and hashCode method in Java, this will help you to do it correctly and with ease:

1) Most of the IDE like NetBeans, Eclipse and IntelliJ IDEA provides support to generate equals() and hashcode() method. In Eclipse do the right click-> source -> generate hashCode() and equals().

2) If your domain class has any unique business key then just comparing that field in equals method would be enough instead of comparing all the fields e.g. in case of our example if "id" is unique for every Person and by just comparing id we can identify whether two Person are equal or not.

3) While [overriding hashCode in Java](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html) makes sure you use all fields which have been used in equals method in Java.

4) String and Wrapper classes like Integer, Float and Double override equals method but StringBuffer doesn’t override it.

5) Whenever possible try to make your fields immutable by using [final variables in Java](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html), equals method based on immutable fields are much secure than on mutable fields.

6) Don't use instanceof check in equals method, as it could break contract of equals() method in sub-class, results in non-symmetric equals, because instanceof return true for child class as well. For example, if you compare two objects Parent and Child from same type-hierarchy; Parent.equals(child) will return true, because child instanceof Parent is true, but Child.equals(parent) will return false, because Parent instanceof Child is false. This means equals() is not following symmetric contract, which states if a.equals(b) == true than b.equals(a) == true, as shown below :

**public** **class** Parent **{**

**}**

**public** **class** Child **extends** Parent **{**

**}**

**public** **class** InstanceOfCheck **{**

**public** **static** **void** main**(**String args**[])** **{**

Parent p **=** **new** Parent**();**

Child c **=** **new** Child**();**

System**.**out**.**println**(**"child instanceof Parent " **+** **(**c **instanceof** Parent**));**

System**.**out**.**println**(**"parent instanceof Child " **+** **(**p **instanceof** Child**));**

**}**

**}**

Output **:**

child **instanceof** Parent **true**

parent **instanceof** Child **false**

Alternatively, you can mark equals as final method to prevent it from being overridden.

7) While comparing String object in Java, [prefer equals() than == operator](http://javarevisited.blogspot.sg/2012/12/difference-between-equals-method-and-equality-operator-java.html).

8) Use IDE or Apache commons EqualsBuilder and HashCodeBuilder utility classes to automatically implement equals() and hashcode() method in Java.

9) Two object which is logically equal but loaded from different ClassLoader cannot be equals. Remember that getClass() check, it will return false if class loader is different.

10) Use [@Override annotation](http://javarevisited.blogspot.sg/2012/11/why-use-override-annotation-in-java.html) on hashcode() as well, as it also prevents subtle mistakes over return type. e.g. return type of hashcode() method is int, but many times programmers mistakenly put long.

11) One  example, where equals method is not consistent with compareTo is java.math.BigDecimal class. If you compare two BigDecimal object e.g. 120.00 and 120.000, equals method will return false, while compareTo will return zero. Both are inconsistent, because equals take both scale and value in consideration, while compareTo method only consider values.

12) From Java 7 you can also use a new utility class called java.util.Objects for null safe equality check and calculating hash code. You can replace our null-safe code for check equality :

**(**name **==** guest**.**name **||** **(**name **!=** **null** **&**amp**;&**amp**;** name**.**equals**(**guest**.**getName**())))**

to much concise

Objects**.**equals**(**name**,** guest**.**getName**());**

### Use of Equals and Hashcode Method in Hibernate

Hibernate is a popular, open source Java persistent framework, which provides Object Relational Mapping, also known as ORM framework. It uses equals and hashcode method to provide object's equality in Java side.  You should **override equals() and hashcode()** if :

1) You are storing instance of persistent class in a Set for representing many-valued associations.

2) You are using reattachment of detached persistent instances.

Another worth noting point is that Hibernate only guarantees equivalence of database row (persistent identity) and Java object inside a particular Session. Which means if you store instances retrieved in different Sessions in a Set, you will be having duplicates. Now the most important aspect of overriding equals and hashcode() for hibernate entity classes, you should never decide equality just based upon identifier. Though it’s convenient to compare identifier to see if the belong to same database row, Unfortunately, we can't use this approach with generated identifiers. Since Hibernate only assign identifier values to the object that are persistent, a newly created instance will not have any identifier value. Similarly, if an instance is not persisted, and currently in a Set, saving it to database will assigned an identifier value, which will further change the value of hashcode() method, finally results in breaking the contract of the Set. That's why it's best to implement equals and hashcode in Hibernate using business key equality e.g. an Employee is same if it's name, surname, father's name, department, date of birth is same. Properties which are not prone to change e.g. date of birth are better candidate of business equality than those which is easier to change e.g. address and contact number.

In short, remember these best practices while overriding equals() and hashcode() for Hibernate entity class :

1) Don't let your equals() method only uses identifier values for equivalence check.

2) Implement equals() and hashCode() using real word key that would identify instance in real world.

3) Use Immutable and unique properties of objects for equality.  
  
  
4) Don't use [getClass()](http://javarevisited.blogspot.sg/2012/09/how-to-determine-type-of-object-runtime-identification.html) to compare object equality because Hibernate uses proxy and this check will always fail. Instead use instanceof operator, it respect proxy because they have IS-A relationship with actual object.  
  
  
5) Use getter and setter methods to access properties instead of directly accessing the, because hibernate lazily initialize object, when there getProperty() method is called. Using name may return null but getName() may return value from database.

That’s all about **overriding equals() and hashcode() methods in Java**, I am reiterating this but its imperative for a Java programmer to be able to write equals , hashcode(), compareTo() method by hand. It is not just useful for learning purpose but to clear any coding exercise during Java interviews. Writing code for equals and hashcode is very popular programming interview questions now days. For Hibernate persistent class its rather tricky to override equals() and hashCode() because otherwise bad practices turns into best practices because of extensive of proxy. You should not use Eclipse IDE code generator for equals() and hashCode() for hibernate entity class, as they use getClass() to check type equality.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.com/2011/10/override-hashcode-in-java-example.html>

How to override hashcode in Java example – Tutorial

Equals and hashcode methods are two primary but yet one of most important methods for java developers to be aware of. Java intends to provide equals and hashcode for every class to test the equality and to provide a hash or digest based on content of class. Importance of hashcode increases when we use the object in different collection classes which works on hashing principle e.g. [hashtable and hashmap](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html). A well written hashcode method can improve performance drastically by distributing objects uniformly and avoiding collision. In this article we will see **how to correctly override hashcode() method in java with a simple example**. We will also examine important aspect of hashcode contracts in java. This is in continuation of my earlier post on [overriding equals method in Java](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html), if you haven’t read already I would suggest go through it.

### General Contracts for hashCode() in Java

1) If two objects are **equal** by equals() method then there **hashcode** returned by hashCode() method must be same.

2) Whenever *hashCode() mehtod* is invoked on the same object more than once within single execution of application, hashCode() must return same integer provided no information or fields used in equals and hashcode is modified. This integer is not required to be same during multiple execution of application though.

3) If two objects are not equals by equals() method it is not require that there hashcode must be different. Though it’s always good practice to return different hashCode for unequal object. Different hashCode for distinct object can improve performance of hashmap or hashtable by reducing collision.

To better understand concept of equals and hashcode and what happens if you don’t override them properly I would recommend understanding of [How HashMap works in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html)

### Overriding hashCode method in Java

We will follow step by step approach for *overriding hashCode method*. This will enable us to understand the concept and process better.

1) Take a prime hash e.g. 5, 7, 17 or 31 (prime number as hash, results in distinct hashcode for distinct object)

2) Take another prime as multiplier different than hash is good.

3) Compute hashcode for each member and add them into final hash. Repeat this for all members which participated in equals.

4) Return hash

  Here is an example of hashCode() method

   @Override

**public int hashCode()** {

        int hash = 5;

        hash = 89  hash + (this.name != null ? this.name.hashCode() : 0);

        hash = 89  hash + (int) (this.id ^ (this.id >>> 32));

        hash = 89  hash + this.age;

        return hash;

    }

It’s always good to *check null before calling hashCode()* method on members or fields to avoid NullPointerException, if member is null than return zero. Different data types has different way to compute hashCode.Integer members are simplest we just add there value into hash, for other numeric data-type are converted into int and then added into hash. Joshua bloach has full tables on this. I mostly relied on IDE for this.

### Better way to override equals and hashCode

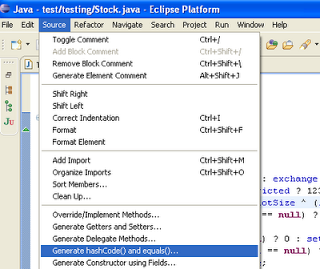
In my opinion better way to **override both equals and hashcode method** should be left to IDE. I have seen Netbeans and Eclipse and found that both has excellent support of generating *code for equals and hashcode* and there implementations seems to follow all best practice and requirement e.g. null check , instanceof check etc and also frees you to remember how to compute hashcode for different data-types.

Let’s see how we can override hashcode method in Netbeans and Eclipse.

**In Netbeans**

1) Write your Class.

2) Right click + insert code + Generate equals() and hashCode().

[](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html)

**In Eclipse**

1) Write your Class.

2) Go to Source Menu + Generate hashCode() and equals()

### Things to remember while overriding hashcode in Java

1. Whenever you override equals method, hashcode should be overridden to be in compliant of equals hashcode contract.

2. hashCode() is declared in Object class and **return type of hashcode method is int** and not long.

3. For immutable object you can cache the hashcode once generated for improved performance.

4. Test your hashcode method for equals hashcode compliance.

5. If you **don't** **override hashCode() method** properly your Object may not function correctly on hash based collection e.g. HashMap, Hashtable or HashSet.

### Complete example of equals and hashCode

**public** **class** Stock {

**private** String symbol;

**private** String exchange;

**private** **long** lotSize;

**private** **int** tickSize;

**private** **boolean** isRestricted;

**private** Date settlementDate;

**private** BigDecimal price;

       @Override

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

              result = prime \* result

                           + ((exchange == **null**) ? 0 : exchange.hashCode());

              result = prime \* result + (isRestricted ? 1231 : 1237);

              result = prime \* result + (**int**) (lotSize ^ (lotSize >>> 32));

              result = prime \* result + ((price == **null**) ? 0 : price.hashCode());

              result = prime \* result

                           + ((settlementDate == **null**) ? 0 : settlementDate.hashCode());

              result = prime \* result + ((symbol == **null**) ? 0 : symbol.hashCode());

              result = prime \* result + tickSize;

**return** result;

       }

       @Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj) **return** **true**;

**if** (obj == **null** || **this**.getClass() != obj.getClass()){

**return** **false**;

              }

              Stock other = (Stock) obj;

**return**

**this**.tickSize == other.tickSize && **this**.lotSize == other.lotSize &&

**this**.isRestricted == other.isRestricted &&

(**this**.symbol == other.symbol|| (**this**.symbol != **null** && **this**.symbol.equals(other.symbol)))&&

(**this**.exchange == other.exchange|| (**this**.exchange != **null** && **this**.exchange.equals(other.exchange))) &&

(**this**.settlementDate == other.settlementDate|| (**this**.settlementDate != **null** && **this**.settlementDate.equals(other.settlementDate))) &&

(**this**.price == other.price|| (**this**.price != **null** && **this**.price.equals(other.price)));

 }

}

### Writing equals and hashcode using Apache Commons EqualsBuilder and HashCodeBuilder

EqualsBuilder and HashCodeBuilder from Apache commons are much better way to override equals and hashcode method, at least much better than ugly equals, hashcode generated by Eclipse. I have written same example by using HashCodebuilder and EqualsBuilder and now you can see how clear and concise they are.

    @Override

    public boolean **equals**(Object obj){

        if (obj instanceof Stock) {

            Stock other = (Stock) obj;

            EqualsBuilder builder = new **EqualsBuilder**();

            builder.append(this.symbol, other.symbol);

            builder.append(this.exchange, other.exchange);

            builder.append(this.lotSize, other.lotSize);

            builder.append(this.tickSize, other.tickSize);

            builder.append(this.isRestricted, other.isRestricted);

            builder.append(this.settlementDate, other.settlementDate);

            builder.append(this.price, other.price);

            return builder.isEquals();

        }

        return false;

    }

    @Override

    public int **hashCode**(){

        HashCodeBuilder builder = new **HashCodeBuilder**();

        builder.append(symbol);

        builder.append(exchange);

        builder.append(lotSize);

        builder.append(tickSize);

        builder.append(isRestricted);

        builder.append(settlementDate);

        builder.append(price);

        return builder.toHashCode();

    }

    public static void main(String args[]){

        Stock sony = new Stock("6758.T", "Tkyo Stock Exchange", 1000, 10, false, new Date(), BigDecimal.valueOf(2200));

        Stock sony2 = new Stock("6758.T", "Tokyo Stock Exchange", 1000, 10, false, new Date(), BigDecimal.valueOf(2200));

        System.out.println("Equals result: " + sony.**equals**(sony2));

        System.out.println("HashCode result: " + (sony.**hashCode**()== sony2.**hashCode**()));

    }

Onlything to concern is that it adds dependency on apache commons jar, most people use it but if you are not using than you need to include it for writing equals and hashcode method.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html>

How to override hashcode in Java example – Tutorial

Equals and hashcode methods are two primary but yet one of most important methods for java developers to be aware of. Java intends to provide equals and hashcode for every class to test the equality and to provide a hash or digest based on content of class. Importance of hashcode increases when we use the object in different collection classes which works on hashing principle e.g. [hashtable and hashmap](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html). A well written hashcode method can improve performance drastically by distributing objects uniformly and avoiding collision. In this article we will see **how to correctly override hashcode() method in java with a simple example**. We will also examine important aspect of hashcode contracts in java. This is in continuation of my earlier post on [overriding equals method in Java](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html), if you haven’t read already I would suggest go through it.

### General Contracts for hashCode() in Java

1) If two objects are **equal** by equals() method then there **hashcode** returned by hashCode() method must be same.

2) Whenever *hashCode() mehtod* is invoked on the same object more than once within single execution of application, hashCode() must return same integer provided no information or fields used in equals and hashcode is modified. This integer is not required to be same during multiple execution of application though.

3) If two objects are not equals by equals() method it is not require that there hashcode must be different. Though it’s always good practice to return different hashCode for unequal object. Different hashCode for distinct object can improve performance of hashmap or hashtable by reducing collision.

To better understand concept of equals and hashcode and what happens if you don’t override them properly I would recommend understanding of [How HashMap works in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html)

### Overriding hashCode method in Java

[Override java hashcode example](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html)We will follow step by step approach for *overriding hashCode method*. This will enable us to understand the concept and process better.

1) Take a prime hash e.g. 5, 7, 17 or 31 (prime number as hash, results in distinct hashcode for distinct object)

2) Take another prime as multiplier different than hash is good.

3) Compute hashcode for each member and add them into final hash. Repeat this for all members which participated in equals.

4) Return hash

  Here is an example of hashCode() method

   @Override

**public int hashCode()** {

        int hash = 5;

        hash = 89  hash + (this.name != null ? this.name.hashCode() : 0);

        hash = 89  hash + (int) (this.id ^ (this.id >>> 32));

        hash = 89  hash + this.age;

        return hash;

    }

It’s always good to *check null before calling hashCode()* method on members or fields to avoid NullPointerException, if member is null than return zero. Different data types has different way to compute hashCode.Integer members are simplest we just add there value into hash, for other numeric data-type are converted into int and then added into hash. Joshua bloach has full tables on this. I mostly relied on IDE for this.

### Better way to override equals and hashCode

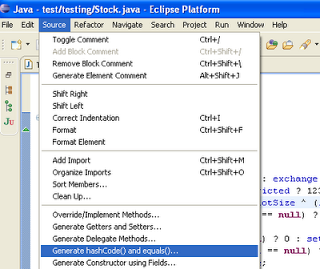
In my opinion better way to **override both equals and hashcode method** should be left to IDE. I have seen Netbeans and Eclipse and found that both has excellent support of generating *code for equals and hashcode* and there implementations seems to follow all best practice and requirement e.g. null check , instanceof check etc and also frees you to remember how to compute hashcode for different data-types.

Let’s see how we can override hashcode method in Netbeans and Eclipse.

**In Netbeans**

1) Write your Class.

2) Right click + insert code + Generate equals() and hashCode().

[](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html)

**In Eclipse**

1) Write your Class.

2) Go to Source Menu + Generate hashCode() and equals()

### Things to remember while overriding hashcode in Java

1. Whenever you override equals method, hashcode should be overridden to be in compliant of equals hashcode contract.

2. hashCode() is declared in Object class and **return type of hashcode method is int** and not long.

3. For immutable object you can cache the hashcode once generated for improved performance.

4. Test your hashcode method for equals hashcode compliance.

5. If you **don't** **override hashCode() method** properly your Object may not function correctly on hash based collection e.g. HashMap, Hashtable or HashSet.

### Complete example of equals and hashCode

**public** **class** Stock {

**private** String symbol;

**private** String exchange;

**private** **long** lotSize;

**private** **int** tickSize;

**private** **boolean** isRestricted;

**private** Date settlementDate;

**private** BigDecimal price;

       @Override

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

              result = prime \* result

                           + ((exchange == **null**) ? 0 : exchange.hashCode());

              result = prime \* result + (isRestricted ? 1231 : 1237);

              result = prime \* result + (**int**) (lotSize ^ (lotSize >>> 32));

              result = prime \* result + ((price == **null**) ? 0 : price.hashCode());

              result = prime \* result

                           + ((settlementDate == **null**) ? 0 : settlementDate.hashCode());

              result = prime \* result + ((symbol == **null**) ? 0 : symbol.hashCode());

              result = prime \* result + tickSize;

**return** result;

       }

       @Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj) **return** **true**;

**if** (obj == **null** || **this**.getClass() != obj.getClass()){

**return** **false**;

              }

              Stock other = (Stock) obj;

**return**

**this**.tickSize == other.tickSize && **this**.lotSize == other.lotSize &&

**this**.isRestricted == other.isRestricted &&

(**this**.symbol == other.symbol|| (**this**.symbol != **null** && **this**.symbol.equals(other.symbol)))&&

(**this**.exchange == other.exchange|| (**this**.exchange != **null** && **this**.exchange.equals(other.exchange))) &&

(**this**.settlementDate == other.settlementDate|| (**this**.settlementDate != **null** && **this**.settlementDate.equals(other.settlementDate))) &&

(**this**.price == other.price|| (**this**.price != **null** && **this**.price.equals(other.price)));

 }

}

### Writing equals and hashcode using Apache Commons EqualsBuilder and HashCodeBuilder

EqualsBuilder and HashCodeBuilder from Apache commons are much better way to override equals and hashcode method, at least much better than ugly equals, hashcode generated by Eclipse. I have written same example by using HashCodebuilder and EqualsBuilder and now you can see how clear and concise they are.

    @Override

    public boolean **equals**(Object obj){

        if (obj instanceof Stock) {

            Stock other = (Stock) obj;

            EqualsBuilder builder = new **EqualsBuilder**();

            builder.append(this.symbol, other.symbol);

            builder.append(this.exchange, other.exchange);

            builder.append(this.lotSize, other.lotSize);

            builder.append(this.tickSize, other.tickSize);

            builder.append(this.isRestricted, other.isRestricted);

            builder.append(this.settlementDate, other.settlementDate);

            builder.append(this.price, other.price);

            return builder.isEquals();

        }

        return false;

    }

    @Override

    public int **hashCode**(){

        HashCodeBuilder builder = new **HashCodeBuilder**();

        builder.append(symbol);

        builder.append(exchange);

        builder.append(lotSize);

        builder.append(tickSize);

        builder.append(isRestricted);

        builder.append(settlementDate);

        builder.append(price);

        return builder.toHashCode();

    }

    public static void main(String args[]){

        Stock sony = new Stock("6758.T", "Tkyo Stock Exchange", 1000, 10, false, new Date(), BigDecimal.valueOf(2200));

        Stock sony2 = new Stock("6758.T", "Tokyo Stock Exchange", 1000, 10, false, new Date(), BigDecimal.valueOf(2200));

        System.out.println("Equals result: " + sony.**equals**(sony2));

        System.out.println("HashCode result: " + (sony.**hashCode**()== sony2.**hashCode**()));

    }

Onlything to concern is that it adds dependency on apache commons jar, most people use it but if you are not using than you need to include it for writing equals and hashcode method.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.com.by/2011/06/comparator-and-comparable-in-java.html>

How to use Comparator and Comparable in Java? With example

**Comparator and Comparable in Java Examples**

Difference between Comparator and Comparable in Java is very [popular Java interview question](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) mostly asked in telephonic round and writing code to sort object using Comparable or Comparator is popular on  written test round of interview.The question was this “How you will sort Employee object based on his EmployeeID and his name” and this involves the use of both Comparable as well as Comparator interface in Java. This post is my revision on Java fundamentals similar to I did about [equals method in Java](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) and  some tips to [override hashCode in Java](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html). All of these methods are fundamentals in Java programming language and correct understanding is must for any Java developer. **Comparators and comparable** in Java are two interfaces which is used to implement sorting in Java. It’s often required to sort objects stored in any collection classes like ArrayList, HashSet or in Array and that time we need to use either  compare() or  compareTo() method defined in java.util.Comparator and java.lang.Comparable. In this Java tutorial we will see example of  Comparator and Comparable to sort object in Java and discuss some best practices around when to use Comparator interface etc. Any way before moving ahead Let’s see some important differences between Comparable and Comparator in Java.

## Comparator vs Comparable in Java

[Difference between Comparator vs Comparable in Java](http://2.bp.blogspot.com/-wrzDeQGAe1I/TWu8pLuLr4I/AAAAAAAAADE/V017G-6Q61w/s1600/java_logo_50_50.jpg)Here are some of the common differences, which is worth remembering to answer this question if asked during a telephonic or face to face interview:

1) Comparator in Java is defined in java.util package while Comparable interface in Java is defined in java.lang package, which very much says that Comparator should be used as an utility to sort objects which Comparable should be provided by default.

2) Comparator interface in Java has method public int compare (Object o1, Object o2) which returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second. While Comparable interface has method public int compareTo(Object o) which returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

3) If you see then logical difference between these two is *Comparator in Java* compare two objects provided to him, while Comparable interface compares "this" reference with the object specified. I have shared lot of tips on [how to override compareTo() method](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html) and avoid some common mistakes programmer makes while implementing Comparable interface.

4) Comparable in Java is used to implement **natural ordering of object**. In Java API String, Date and wrapper classes implements Comparable interface.Its always good practice to override compareTo() for value objects.

5) If any class implement Comparable interface in Java then collection of that object either [List](http://javarevisited.blogspot.sg/2012/04/difference-between-list-and-set-in-java.html) or Array can be sorted automatically by using  Collections.sort() or Arrays.sort() method and object will be sorted based on there natural order defined by CompareTo method.

6)Objects which implement *Comparable in Java*  can be used as keys in a SortedMap like [TreeMap](http://javarevisited.blogspot.sg/2011/12/treemap-java-tutorial-example-program.html) or elements in a SortedSet  for example TreeSet, without specifying any Comparator.

These were combination of some theoretical and practical differences between Comparator and Comparator interface in Java. It does help you to decide when to use Comparator vs Comparable but things will be more clear when we some best practices around using both of these interfaces. Now let’s see an example of Comparator in Java:

### Example of using Comparator and Comparable in Java

So in Summary if you want to **sort objects based on natural order** then use Comparable in Java and if you want to sort on some other attribute of object then use Comparator in Java. Now to understand these concepts lets see an example or real life coding:

1) There is class called Person, sort the Person based on person\_id, which is primary key in database

2) Sort the Person based on there name.

For a Person class, sorting based on person\_id can be treated as *natural order sorting* and sorting based on name field can be implemented using Comparator interface. To sort based on person\_id we need to implement compareTo() method.

**public** **class** Person **implements** **Comparable** {  
    **private** **int** person\_id;  
    **private** **String** name;  
      
    /\*\*  
     \* Compare current person with specified person  
     \* return zero if person\_id for both person is same   
     \* return negative if current person\_id is less than specified one  
     \* return positive if specified person\_id is greater than specified one  
     \*/  
  @Override   
    **public** **int** compareTo(Object o) {  
        Person p = (Person) o;   
        **return** **this**.person\_id - o.person\_id ;  
    }  
    ….  
}

Generally you should not use difference of integers to decide output of compareTo method as result of **integer subtraction can overflow** but if you are sure that both operands are positive then its one of the quickest way to compare two objects. See my post [things to remember while overriding compareTo in Java](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html) for more tips on compareTo.

And for sorting based on person name we can implement compare(Object o1, Object o2) method of Java Comparator class.

/\*\*  
 \* Comparator implementation which sorts Person objects on person\_id field  
 \*/  
**public** **class** SortByPerson\_ID **implements** **Comparator**{  
  
    **public** **int** compare(Object o1, Object o2) {  
        Person p1 = (Person) o;  
        Person p2 = (Person) o;   
        **return** p1.getPersonId() - p2.getPersonId();  
    }  
}

Similar guidelines applies while implementing compare() method as well and instead of using subtraction operator, its better to use logical operator to compare whether two integers are equal to, less than or greater than. You can write several types of Java Comparator based upon your need for example  reverseComparator , ANDComparator , ORComparator etc which will return negative or positive number based upon logical results. [String in Java](http://javarevisited.blogspot.sg/2011/07/string-vs-stringbuffer-vs-stringbuilder.html) even provides an special comparator called CASE\_INSENSITIVE\_ORDER, to perform case insensitive comparison of String objects.

**How to Compare String in Java**

[String is immutable in Java](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html) and one of the most used value class. For comparing String in Java we should not be worrying because String implements Comparable interface and provides a lexicographic implementation for CompareTo method which compare two strings based on contents of characters or you can say in lexical order. You just need to call String.compareTo(AnotherString) and Java will determine whether specified String is greater than , equal to or less than current object. See my post [4 example to compare String in Java](http://javarevisited.blogspot.sg/2012/03/how-to-compare-two-string-in-java.html) for alternatives ways of comparing String.

**How to Compare Dates in Java**

Dates are represented by java.util.Date class in Java and like String,  Date also implements Comparable in Java so they will be automatically sorted based on there natural ordering if they got stored in any sorted collection like TreeSet or TreeMap. If you explicitly wants to compare two dates in Java you can call Date.compareTo(AnotherDate) method in Java and it will tell whether specified date is greater than , equal to or less than current String. See my post [3 ways to compare Dates in Java](http://javarevisited.blogspot.sg/2012/02/3-example-to-compare-two-dates-in-java.html) for more alternatives of comparing two dates.

**When to use Comparator and Comparable in Java**

At last let’s see some best practices and recommendation on when to use Comparator or Comparable in Java:

1) If there is a natural or default way of sorting Object already exist during development of Class than use Comparable. This is intuitive and you given the class name people should be able to guess it correctly like Strings are sorted chronically, Employee can be sorted by there Id etc. On the other hand if an Object can be sorted on multiple ways and client is specifying on which parameter sorting should take place than use Comparator interface. for example Employee can again be sorted on name, salary or department and clients needs an API to do that. Comparator implementation can sort out this problem.

2) Some time you write code to sort object of a class for which you are not the original author, or you don't have access to code. In these cases you can not implement Comparable and Comparator is only way to sort those objects.

3) Beware with the fact that How those object will behave if stored in SorteSet or SortedMap like TreeSet and [TreeMap](http://javarevisited.blogspot.sg/2011/12/treemap-java-tutorial-example-program.html). If an object doesn't implement Comparable than while putting them into SortedMap, always provided corresponding Comparator which can provide sorting logic.

4) Order of comparison is very important while implementing Comparable or Comparator interface. for example if you are sorting object based upon name than you can compare first name or last name on any order, so decide it judiciously. I have shared more detailed tips on compareTo on my post how to implement CompareTo in Java.

5) Comparator has a distinct advantage of being self descriptive  for example if you are writing Comparator to compare two Employees based upon there salary than name that comparator as SalaryComparator, on the other hand compareTo()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On consistency of CompareTo() and equals() you probably would like to add when you compare any instance with null ,compareTo() method of comparable interface in java throws NullPointerException while in case of equals(null)  
it should return false and if your compareTo() is not consistent with equals then those object will behave strangely in case if they will be used as key on SortedMap or value on SortedSet effectively they violates general contact  
for set and map interface in java which is defined in terms of equals method. though this problem will not arise if you provide explicit comparator to sorted collection. anyway good effort and your example of comparator and comparable is decent.

------------------------

In any case, your Person should implement Comparable<Person>, not the raw type Comparable.  
  
Furthermore, using subtraction: a-b &lt 0, == 0, or > 0 to determine ordering is generally unreliable. It is possible for a < b but a - b > 0 because integer arithmetic in Java is modulo 2^32. Integer.MIN\_VALUE < 1. However, Integer.MIN\_VALUE - 1 == Integer\_MAX\_VALUE, as the subtraction overflows. Always use code like:  
  
return a < b ? -1 : (a > b ? 1 : 0);  
  
Even if it doesn't seem necessary, you will do the right thing when it is.

-----------------------------------------------------

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*