This might help:

1) "Build" menu -> "Rebuild Project". Sometimes Intellij doesn't rewrite the classes because they already exist, this way you ask Intellij to rewrite everything.

2) "Run" menu -> "Edit configuration" -> delete the profile -> add back the profile ("Application" if it's a Java application), choose your main class from the "Main Class" dropdown menu.

3)"Build" menu -> "Rebuild Project".

Core Java

1. Difference between java6/java7/java8?

Java 5.0 not 1.5 to The number was changed to "better reflect the level of maturity, stability, scalability and security of the J2SE.

* Generic : provide compile time (static ) type safty for collection and eliminate the need for most type cast (type conversion)
* Metadata: (Annotation) allow language construct like classes and method to be taged with some additional data which can then be processed by meta-data aware utilities.
* Autoboxing /unboxing : automatic conversion between primitive types and primitive Wrapper classes.
* Enumaration : the enum keyword creates typesafe, ordered list of values(Day.MONDAY,DAY.TUESDAY etc.), previously achived by non type safe constanat and manualy constructed classes.
* Varargs: last parameter of method can be declared as type name followed by three dots(eg. Int add(int … numbers) )
* Enhanced for-each loop : extended for each loop syntax for any Iterable.
* Improved symantics of execustion of multi-threded java program. New java memory model.
* Static import.
* Atomatic stub generation for RMI objects
* Swing : new skinnable look and feel callied synth.
* The concurancy utilities in package java.util.concurent
* Scanner class for parsing data from various input stream and buffers.
* Improved startup time and memory footprint. Sharing of read-only data between multiple running JVMs. Remote monitoring and management. A new JVM profiling API. Programmatic generation of stack traces. Support for XML 1.1 with Namespaces, XML Schema, SAX 2.0.2, DOM Level 3, and XSLT with a fast XSLTC compiler. Unicode 4.0 support.

Java 6

* Scripting language support: generic api for tight integration with scripting language and script in Mozilla javascript rohino integration .
* Performance improvement for core platform.
* Improved web service support thrue JAX-WS
* Jdbc 4.0 support
* Java compiler api : select and invocke java compiler programmatically.
* Upgrade jaxb-to 2.0 including integration of stAX parser
* SUPPort for plugbale annotation.
* Pre-installed relational data base java derby.
* Many gui improvement.
* Jvm improvement: synchronization and compiler performance optimization. New algorithem and upgrades to garbage collection algorthem.and application start up performance.

Java 7:

* Jvm suppor t for dyanamic language
* String in switch
* Automatic resource management in try statement.
* Improved type reference for generic instance creation aka the diamond operator.
* Simplified varg method declaration .
* Allowing underscore in numeric litrals.
* Catching multiple exception types and rethrowing exception with improved type checking.
* Concurancey utilities.
* New file I/o libraray adding support for multiple file system, file metadata and symbolic link new package java.nio.file, java.nio.file.attribute and java.nio.file.spi

Java 8

* Language level support for lymbda expression and default method.
* Project Nashron , a javascript runtime which allow developer to embed javascript codewithin application .
* Annotation on java types.
* Unsigned integer arthmatic
* Repeating annotaition
* Date and Time api
* Statically linked jni librararies.
* Remove permanent generation

Java 9

* Java platform module system.
* The java shell
* Ahed of time compilation
* Xml catalogs
* More concurancy update , reactive stream, Flow class.
* Variable handeles.
* jlink

**Abstraction with real scenario?**

In java abstraction concept is base of all other oops concept like Plolymorphism, Inheritance, Encapsulation.

Abstraction means hiding complexity with respective to context.

Also Abstraction is modeling real life entity in programing word,

We often reuse abstractions when attempting to model a new concept.

Producing an abstraction of a system to be built, known as a model, is in some senses second nature to us, and yet paradoxically is one of the hardest things that software developers have to do in the life cycle of an information systems project. It’s also one of the most important.

Abstraction is the process of generalization: taking a concrete implementation and making it applicable to different, albeit somewhat related, types of data. "Abstract" is an antonym of "concrete". With abstractions you represent notions and ideas, rather than the concrete way these ideas are implemented. This fits into your understanding of abstraction - you are hiding the details and you only show the interface.

But this also fits with abstract classes—they are not concrete (they can't be instantiated, for one), and they don't specify implementations. They specify abstract ideas that subclasses have to take care of.

Abstract classes, unlike interfaces, are classes. They are more expensive to use, because there is a look-up to do when you inherit from them.

Abstract classes look a lot like interfaces, but they have something more—you can define a behavior for them.

**Interfaces have following properties:**

Define well known public contract, abilities of the type

Applicable to show horizontal inheritance, i.e. branching on the first level of inheritance (e.g. ILog to define logging facilities to database, text file, XML, SOAP etc.)

All members are public

No implementation allowed

Inheritance child can have many interfaces to implement

Useful for third party integration

Naming usually starts with I

**With abstract classes, properties differ as follows:**

Define structure, identity and some default supported behavior Applicable to show vertical inheritance, i.e. deep branching on the several levels Members can have different visibility (from public to private) .You can implement some members (e.g. \*Reader classes) Inheritance child can have only one base abstract class

**When To Use Interfaces**

An interface allows somebody to start from scratch to implement your interface or implement your interface in some other code whose original or primary purpose was quite different from your interface. To them, your interface is only incidental, something that you have to add on to the their code to be able to use your package. The disadvantage is every method in the interface must be public. You might not want to expose everything.

**When To Use Abstract classes**

An abstract class, in contrast, provides more structure. It usually defines some default implementations and provides some tools useful for a full implementation. The catch is, code that uses it must use your class as the base. That may be highly inconvenient if the other programmers wanting to use your package have already developed their own class hierarchy independently. In Java, a class can inherit from only one base class.

**When to Use Both**

You can offer the best of both worlds, an interface and an abstract class. Implementers can ignore your abstract class if they choose. The only drawback of doing this is that calling methods via their interface name is slightly slower than calling them via their abstract class name.

Encapsulation is the packing of data and functions operating on that data into a single component and restricting the access to some of the object's components.

Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition.

Abstraction is a mechanism which represent the essential features without including implementation details.

Encapsulation:-- Information hiding.

Abstraction:-- Implementation hiding.

Abstraction : Abstraction means to show What part of functionality.

Encapsulation : Encapsulation means to hide the How part of the functionality.

Abstraction--- Hiding Implementation--at Design---Using Interface/Abstract calsses

Encapsulation--Hiding Data --At Development---Using access modifiers(public/private)

ABSTRACTION

"A view of a problem that extracts the essential information

relevant to a particular purpose and ignores the remainder of

the information."

-- [IEEE, 1983]

"The essence of abstraction is to extract essential properties

while omitting inessential details."

-- [Ross et al, 1975]

"Abstraction is a process whereby we identify the important

aspects of a phenomenon and ignore its details."

-- [Ghezzi et al, 1991]

"Abstraction is generally defined as 'the process of

formulating generalised concepts by extracting common qualities

from specific examples.'"

-- [Blair et al, 1991]

"Abstraction is the selective examination of certain aspects of

a problem. The goal of abstraction is to isolate those aspects

that are important for some purpose and suppress those aspects

that are unimportant."

-- [Rumbaugh et al, 1991]

"The meaning [of abstraction] given by the Oxford English

Dictionary (OED) closest to the meaning intended here is 'The

act of separating in thought'. A better definition might be

'Representing the essential features of something without

including background or inessential detail.'"

-- [Graham, 1991]

"[A] simplified description, or specification, of a system that

emphasizes some of the system's details or properties while

suppressing others. A good abstraction is one that emphasizes

details that are significant to the reader or user and suppress

details that are, at least for the moment, immaterial or

diversionary."

-- [Shaw, 1984]

"An abstraction denotes the essential characteristics of an

object that distinguish it from all other kinds of object and thus

provide crisply defined conceptual boundaries, relative to the

perspective of the viewer."

-- [Booch, 1991]

One point of confusion regarding abstraction is its use as both a

process and an entity. Abstraction, as a process, denotes the

extracting of the essential details about an item, or a group of items,

while ignoring the inessential details. Abstraction, as an entity,

denotes a model, a view, or some other focused representation for an

actual item. Abstraction is most often used as a complexity mastering

technique. For example, we often hear people say such things as: "just

give me the highlights" or "just the facts, please." What these people

are asking for are abstractions.

INFORMATION HIDING

"The second decomposition was made using 'information hiding'

... as a criterion. The modules no longer correspond to steps in

the processing. ... Every module in the second decomposition is

characterized by its knowledge of a design decision which it hides

from all others. Its interface or definition was chosen to reveal as

little as possible about its inner workings."

-- [Parnas, 1972b]

"... the purpose of hiding is to make inaccessible certain

details that should not affect other parts of a system."

-- [Ross et al, 1975]

"... [I]nformation hiding: a module is characterized by the

information it hides from other modules, which are called its

clients. The hidden information remains a secret to the client

modules."

-- [Ghezzi et al, 1991]

"[Information hiding is] the principle that users of a software

component (such as a class) need to know only the essential

details of how to initialize and access the component, and do not

need to know the details of the implementation."

-- [Budd, 1991]

"The technique of encapsulating software design decisions in

modules in such a way that the module's interfaces reveal little

as possible about the module's inner workings; thus each module is

a 'black box' to the other modules in the system."

-- [IEEE, 1983]

"The process of hiding all the details of an object that do not

contribute to its essential characteristics; typically, the

structure of an object is hidden, as well as the implementation

of its methods. The terms information hiding and encapsulation

are usually interchangeable."

-- [Booch, 1991]

"The principle of information hiding is central. It says that

modules are used via their specifications, not their

implementations. All information about a module, whether

concerning data or function, is encapsulated with it and,

unless specifically declared public, hidden from other modules."

-- [Graham, 1991]

In his classic 1972 article ([Parnas, 1972b]), D.L. Parnas describes two

different implementation scenarios for a simple key word in context

(KWIC) application. One is decomposed and modularized based on the

steps one might take in accomplishing the purpose of the application.

(Parnas speculates that this approach would be taken by someone who is

basing their design on a flowchart.)

ENCAPSULATION

"1. to enclose in or as if in a capsule"

-- [Mish, 1988]

"The concept of encapsulation as used in an object-oriented

context is not essentially different from its dictionary

definition. It still refers to building a capsule, in the case a

conceptual barrier, around some collection of things."

-- [Wirfs-Brock et al, 1990]

"It is a simple, yet reasonable effective, system-building

tool. It allows suppliers to present cleanly specified

interfaces around the services they provide. A consumer has full

visibility to the procedures offered by an object, and no visibility

to its data. From a consumer's point of view, and object is a

seamless capsule that offers a number of services, with no

visibility as to how these services are implemented ... The

technical term for this is encapsulation."

-- [Cox, 1986]

"Encapsulation or equivalently information hiding refers to the

practice of including within an object everything it needs, and

furthermore doing this in such a way that no other object need ever

be aware of this internal structure."

-- [Graham, 1991]

"We say that the changeable, hidden information becomes the

secret of the module; also, according to a widely used jargon, we

say that such information is encapsulated within the implementation."

-- [Ghezzi et al, 1991]

"Data hiding is sometimes called encapsulation because the data

and its code are put together in a package or 'capsule.'"

-- [Smith, 1991]

"Encapsulation is used as a generic term for techniques which

realize data abstraction. Encapsulation therefore implies the

provision of mechanisms to support both modularity and information

hiding. There is therefore a one to one correspondence in this

case between the technique of encapsulation and the principle of

data abstraction."

-- [Blair et al, 1991]

"Encapsulation (also information hiding) consists of separating

the external aspects of an object which are accessible to other

objects, from the internal implementation details of the object,

which are hidden from other objects."

-- [Rumbaugh et al, 1991]

"[E]ncapsulation -- also known as information hiding --

prevents clients from seeing its inside view, were the behavior

of the abstraction is implemented."

-- [Booch, 1991]

Like abstraction, the word "encapsulation" can be used to describe

either a process or an entity. As a process, encapsulation means the

act of enclosing one or more items within a (physical or logical)

container. Encapsulation, as an entity, refers to a package or an

enclosure that holds (contains, encloses) one or more items. It is

extremely important to note that nothing is said about "the walls of

the enclosure." Specifically, they may be "transparent," "translucent,"

or even "opaque."

Programming languages have long supported encapsulation. For example,

subprograms (e.g., procedures, functions, and subroutines), arrays, and

record structures are common examples of encapsulation mechanisms

supported by most programming languages. Newer programming languages

support larger encapsulation mechanisms, e.g., "classes" in Simula

([Birtwistle et al. 1973]), Smalltalk ([Goldberg and Robson, 1983]),

and C++, "modules" in Modula ([Wirth, 1983]), and "packages" in Ada.

Abstraction, information hiding, and encapsulation are very different,

but highly-related, concepts. One could argue that abstraction is a

technique that helps us identify which specific information should be

visible, and which information should be hidden. Encapsulation is then

the technique for packaging the information in such a way as to hide

what should be hidden, and make visible what is intended to be visible.

It is not hard to see how abstraction, information hiding, and

encapsulation became confused with one another. Further, one could

argue that, regardless of their "dictionary definitions," these terms

have evolved new meanings in the context of software engineering, e.g.,

in much the same way as "paradigm" has. (See, e.g., [Kuhn, 1962].)

However, a stronger argument can be made for keeping the concepts, and

thus the terms, distinct.

**ABSTRACT everything you need and ENCAPSULATE everything you don't need ;)**

Technical difference :

Abstract class :

* Have instance method that implement default behavior.
* Have constructor.
* More structured and can have state.
* May contain non final variable/data
* Class member can be public, protected, default. Etc.
* Class can be extended using extends keyword(extends one class only implement more than one interfaces)
* Can not be instantiated. Can be invocked if main() exist.
* o implement the same or different behaviour among multiple related objects

Interface

* We can create default as well as static methods in the interfaces and provide implementation for them.
* Variable declared in interface by default final.
* Members of interface are public by default.
* Interface can extend another java interface only
* Interface is absulitly abstract and can not be instantiated.
* To implement a contract by multiple unrelated objects

*Consider using abstract classes* if :

1. You want to share code among several closely related classes.
2. You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
3. You want to declare non-static or non-final fields.

*Consider using interfaces* if :

1. You expect that unrelated classes would implement your interface. For example,many unrelated objects can implement Serializable interface.
2. You want to specify the behaviour of a particular data type, but not concerned about who implements its behaviour.
3. You want to take advantage of multiple inheritance of type.

*abstract class establishes "is a" relation with concrete classes. interface provides "has a" capability for classes.*

When you want to provide polymorphic behaviour in an inheritance hierarchy, use abstract classes.

When you want polymorphic behaviour for classes which are completely unrelated, use an interface.

Abstract classes are useful for modeling a class hierarchy. At first glance of any requirement, we are partially clear on what **exactly** is to be built, but we know **what to build.** And so your abstract classes are your base classes.

Interfaces are useful for letting other hierarchy or classes to know that what I am capable of doing. And when you say I am capable of something, you must have that capacity. Interfaces will mark it as compulsory for a class to implement the same functionalities.

I am constructing a building of 300 floors

The building's blueprint **interface**

* For example, Servlet(I)

Building constructed up to 200 floors - partially completed---**abstract**

* Partial implementation, for example, generic and HTTP servlet

Building construction completed-**concrete**

* Full implementation, for example, own servlet

Interface

* We don't know anything about implementation, just requirements. We can go for an interface.
* Every method is public and abstract by default
* It is a 100% pure abstract class
* If we declare public we cannot declare private and protected
* If we declare abstract we cannot declare final, static, synchronized, strictfp and native
* Every interface has public, static and final
* Serialization and transient is not applicable, because we can't create an instance for in interface
* Non-volatile because it is final
* Every variable is static
* When we declare a variable inside an interface we need to initialize variables while declaring
* Instance and static block not allowed

Abstract

* Partial implementation
* It has an abstract method. An addition, it uses concrete
* No restriction for abstract class method modifiers
* No restriction for abstract class variable modifiers
* We cannot declare other modifiers except abstract
* No restriction to initialize variables

Java 8 has reduced the gap between interface and abstract classes to some extent by providing a default method feature. *An interface does not have an implementation for a method* is no longer valid now.

**Java 8 interface**

Static method : we can define staitic and default method in interface

Default method can be overridden in implementing class , static can not be.

Static method belogs to only interface class and can be accessed by interface name. but not on class implementing interface.

Both class and interface can have static method with same name and nither override other.

If class is implementing more than one interface and these interface have same default method then implementing class should have to provide implementation

Users who have classes that implement interfaces enhanced with new default or static methods do not have to modify or recompile them to accommodate the additional methods.

Interface **default** methods:

* It helps in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.
* It helps in extending interfaces without having the fear of breaking implementation classes.

Interface **static** methods:

* They are part of interface, we can’t use it for implementation class objects.
* It helps in providing security by not allowing implementation classes to override them.

**Important points about java interface static method:**

1. Java interface static method is part of interface, we can’t use it for implementation class objects.
2. Java interface static methods are good for providing utility methods, for example null check, collection sorting etc.
3. Java interface static method helps us in providing security by not allowing implementation classes to override them.
4. We can’t define interface static method for Object class methods, we will get compiler error as “This static method cannot hide the instance method from Object”. This is because it’s not allowed in java, since Object is the base class for all the classes and we can’t have one class level static method and another instance method with same signature.
5. We can use java interface static methods to remove utility classes such as Collections and move all of it’s static methods to the corresponding interface, that would be easy to find and use.

**Important points about java interface default methods:**

1. Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes.
2. Java interface default methods has bridge down the differences between interfaces and abstract classes.
3. Java 8 interface default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.
4. Java interface default methods will help us in removing base implementation classes, we can provide default implementation and the implementation classes can chose which one to override.
5. One of the major reason for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.
6. If any class in the hierarchy has a method with same signature, then default methods become irrelevant. A default method cannot override a method from java.lang.Object. The reasoning is very simple, it’s because Object is the base class for all the java classes. So even if we have Object class methods defined as default methods in interfaces, it will be useless because Object class method will always be used. That’s why to avoid confusion, we can’t have default methods that are overriding Object class methods.
7. Java interface default methods are also referred to as Defender Methods or Virtual extension methods.

### Java Functional Interfaces

Before I conclude the post, I would like to provide a brief introduction to Functional interfaces. An interface with exactly one abstract method is known as Functional Interface.

A new annotation @FunctionalInterface has been introduced to mark an interface as Functional Interface. @FunctionalInterface annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces. It’s optional but good practice to use it.

Functional interfaces are long awaited and much sought out feature of Java 8 because it enables us to use **lambda expressions** to instantiate them. A new package java.util.function with bunch of functional interfaces are added to provide target types for lambda expressions and method references.

**Important Points/Observations:**

1. A functional interface has only one abstract method but it can have multiple default methods.
2. @FunctionalInterface annotation is used to ensure an interface can’t have more than one abstract method. The use of this annotation is optional.
3. The java.util.function package contains many builtin functional interfaces in Java 8.

Functional Interfaces In Java

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. Runnable, ActionListener, Comparable are some of the examples of functional interfaces.

Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

// Java program to demonstrate functional interface

class Test

{

public static void main(String args[])

{

// create anonymous inner class object

new Thread(new Runnable()

{

@Override

public void run()

{

System.out.println("New thread created");

}

}).start();

}

}

Java 8 onwards, we can assign [lambda expression](https://www.geeksforgeeks.org/lambda-expressions-java-8/) to its functional interface object like this:

|  |
| --- |
| // Java program to demonstrate Implementation of  // functional interface using lambda expressions    class Test  {    public static void main(String args[])    {        // lambda expression to create the object      new Thread(()->         {System.out.println("New thread created");}).start();    }  } |

The interface can also declare the abstract methods from the java.lang.Object class, but still the interface can be called as a Functional Interface:

**What is java.util.function.Function**  
**Function<T, R>** is an in-built functional interface introduced in Java 8 in the **java.util.function** package. The primary purpose for which **Function<T, R>** has been created is for mapping scenarios i.e when an object of a type is taken as input and it is converted(or mapped) to another type. Common usage of Function is in streams where-in the map function of a stream accepts an instance of Function to convert the stream of one type to a stream of another type.  
Since **Function<T, R>** is a [functional interface](https://www.javabrahman.com/java-8/functional-interfaces-java-8/), hence it can be used as the assignment target for a [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) or a [method reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/).

**Function Descriptor of Function<T, R>**  
**Function<T, R>**’s Function Descriptor is **T -> R**. This means an object of type T is input to the lambda and an object of type R is obtained as return value. To understand Function Descriptors in details you can refer the [function descriptor tutorial](https://www.javabrahman.com/java-8/function-descriptors-java-8-explained/).

**Advantage of predefined java.util.function.Function**: In all scenarios where an object of a particular type is the input, an operation is performed on it and and object of another type is returned as output, the in-built functional interface **Function<T, R>** can be used without the need to define a new functional interface every time.

To be continued with other java 8 feture

**JAVA Collections:**

**JAVA Collections:**

**Interfaces**: Interface in Java refers to the abstract data types. They allow Java collections to be manipulated independently from the details of their representation. Also, they form a hierarchy in object-oriented programming languages.

**Classes:**Classes in Java are the implementation of the collection interface. It basically refers to the data structures that are used again and again.

**Algorithm:** Algorithm refers to the methods which are used to perform operations such as searching and sorting, on objects that implement collection interfaces. Algorithms are polymorphic in nature as the same method can be used to take many forms or you can say perform different implementations of the Java collection interface.

**Interfaces:** [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html), [Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html), [Iterable](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html)<E>, [Collection](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)<E>, [List](https://docs.oracle.com/javase/8/docs/api/java/util/List.html)<E>, [RandomAccess](https://docs.oracle.com/javase/8/docs/api/java/util/RandomAccess.html).

**Class :** java.lang.Object , java.util.AbstractCollection<E>, java.util.AbstractList<E>, **java.util.ArrayList<E>.**

**Algorithm: Groable Array.**

**this class's iterator and listIterator methods are fail-fast.**

**Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.**

**An application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation.**

this implementation is not synchronized

If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.)

This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the Collections.synchronizedList method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

List list = Collections.synchronizedList(new ArrayList(...));

The iterators returned by this class's iterator and listIterator methods are fail-fast:

if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

*int newCapacity = (oldCapacity \* 3)/2 + 1;*

Methods inherited from class java.util.AbstractList

equals, hashCode

Methods inherited from class java.util.AbstractCollection

containsAll, toString

Methods inherited from class java.lang.Object

finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.List

containsAll, equals, hashCode

Methods inherited from interface java.util.Collection

parallelStream, stream

Java ArrayList class

Java ArrayList class uses a dynamic array for storing the elements.It extends AbstractList class and implements List interface.

Java ArrayList class can contain duplicate elements.

Java ArrayList class maintains insertion order.

Java ArrayList class is non synchronized.

Java ArrayList allows random access because array works at the index basis.

In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

## Map:

A Map stores data in key and value association. Both key and values are objects. The key must be unique but the values can be duplicate. Although Maps are a part of Collection Framework, they can not actually be called as collections because of some properties that they posses. However we can obtain a **collection-view** of maps.

Map (I) : Maps unique key to value.

Map.Entry (I) : describe an element in key and value pair. Here Entry is sub-interface of Map.

NevigableMap (i): Extend sortedMap to handle the retrieval of entries based on closest match searches.

SortedMap (I) : Extend Map so that key are maintained in ascending order

* The Map Interface
  1. Sorted Map Interface

Extends map interface

Ensure that entries are in ascending order based on key.

* 1. Nevigable Map Interface

Extends SortedMap interface

Declares behavior that support retrival of entries based on closest match.

**Map interface methods:**

* boolean **containsKey**(Object *k*): returns true if map contain *k* as key. Otherwise false.
* Object **get**(Object *k*) : returns values associated with the key *k*.
* Object **put**(Object *k*, Object *v*) : stores an entry in map.
* Object **putAll**(Map *m*) : put all entries from *m* in this map.
* Set **keySet**() : returns **Set** that contains the key in a map.
* Set **entrySet**() : returns **Set** that contains the entries in a map

Has.

Map implementation classes:

* java.util.HashMap\*
* java.util.Hashtable
* java.util.EnumMap
* java.util.IdentityHashMap
* java.util.LinkedHashMap
* java.util.Properties
* java.util.TreeMap\*
* java.util.WeakHashMap

**HashMap**

HashMap class extends AbastratMap class and implements Map interface.

It uses hashtable to store the map. This allow execution time to get() and put() same.

it is unsynchronized and permits nulls(permits null values and the null key).

This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

**this implementation is not synchronized**

If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the [Collections.synchronizedMap](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html" \l "synchronizedMap-java.util.Map-) method. This is best done at creation time, to prevent accidental unsynchronized access to the map

Map m = Collections.synchronizedMap(new HashMap(...));

The iterators returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a [ConcurrentModificationException](https://docs.oracle.com/javase/8/docs/api/java/util/ConcurrentModificationException.html" \o "class in java.util). Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Constructor :

[**HashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#HashMap--)()

Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).

[**HashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#HashMap-int-)(int initialCapacity)

Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).

|  |
| --- |
| [**HashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#HashMap-int-float-)(int initialCapacity, float loadFactor)  Constructs an empty HashMap with the specified initial capacity and load factor. |
| [**HashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#HashMap-java.util.Map-)([**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<? extends [**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> m)  Constructs a new HashMap with the same mappings as the specified Map. |

Basic syntax:

HashMap<String, Object> hashMap = new HashMap<>();

hashMap.put(“key”, “value");

Philosophy behind hashMap:

If you are looking for an element in a "list", the order of lookup would be proportional to the length of the list,

however, if you split this list into multiple mini-lists AND IF you can quickly tell in which mini-list this element may reside, then the time-complexity for look-up can be reduced greatly. That is what hashMap does, it builds an array of these mini-lists.

So with basics out of the way, here are 10 things every java programmer should know about HashMap.

**(1) Internals of lookup process:**

Lookup process is at the heart of HashMap and almost all the complexity of hashMap lies here. The lookup process consists of 2 steps:

Step# 1: Quickly determine the bucket number in which this element may reside (using key.hashCode()).

Step# 2: Go over the mini-list and return the element that matches the key (using key.equals()).

Note the the value object is not used in any of the calculations within hashMap.

**(2) Immutability of keys:**

A class that we plan to use as a “key” in hashMap needs to follow certain restrictions.

If the object which is used as key in hashMap is modified, then we may have problem retrieving values from hashMap.

Let's say I use an object as key, and put this key and associated value into hashMap. Later I modify one of the properties of this key. If hashCode() and equals() method make use of this property, then we may not find this key in hashMap. If the property being modified is not being used by hashCode() and equals(), then we will still be able to find the key in hashMap.

To do away with this issue altogether, recommendation is to use immutable classes as keys.

Note, that even if only one of the methods hashCode() or equals() return different results(when a property is modified), the key becomes useless from the lookup perspective.

**(3) Load factor and resize:**

When a hashMap resizes, it will double in size and create a new instance and populate it.

When new hashHap is being populated, the linkedList associated with each bucket of source hashMap is iterated and nodes are copied to the destination bucket. However, note that these new nodes are prepended to the head of the destination linkedList. So resizing has an side effect of reversing the order of the items in the list. Default load factor for hashMap is 0.75.

**(4) Worst-case performance:**

In the worst case, a hashMap reduces to a linkedList.

However with Java 8, there is a change,

Java 8 intelligently determines if we are running in the worst-case performance scenario and converts the list into a binary search tree.

**(5) Collisions:**

Collisions happen when 2 distinct keys generate the same hashCode() value. Multiple collisions are the result of bad hashCode() algorithm.

The more the collisions the worse the performance of the hashMap.

There are many collision-resolution strategies - chaining, double-hashing, clustering.

However, java has chosen chaining strategy for hashMap, so in case of collisions, items are chained together just like in a linkedList.

**(6) Adding duplicate entries into hashMap:**

Attempt to put the same key with a different value, will overwrite the old value.

1: hashHap.put(myKey, oldValue);

2: hashHap.put(myKey, newValue);

3: hashMap.get(myKey);// This line will return newValue

Line# 2, above will essentially over-write oldValue with newValue. so hashMap.put() is actually "add or overwrite".

**(7) Concurrency:**

Do not use HashMap in multi-threaded environment.

What if 2 threads starts resizing the hashMap at the same time?

**(8) Map of maps:**

HashMap of hashMaps are very popular.

Pseudo code below to give an idea:

Map<String, Map<String, Object>> multiHashMap = new HashMap<>();

Map<String, Object> myHashMap1;

Map<String, Object> myHashMap2;

multiHashMap.put(“one”, myHashMap1);

multiHashMap.put(“two”, myHashMap2);

Also, "multimap" is a common dataStructure, where value is a collection.

You use the key to retrieve the collection and the manipulate the collection.

Map<Double,List<Object>> multiMap = new TreeMap<Double,List<Object>>();

**(9) Some specialized hashMaps for specific purposes:**

- ConcurrentHashMap: HashMap to be used in multithreaded applications.

- EnumMap: HashMap with Enum values as keys.

- LinkedHashMap: HashMap with predictable iteration order (great for FIFO/LIFO caches)

**(10) Code samples:**

It is really insightful to jump into the source, here is how hashCode() and equals() are implemented in jdk.

(a) Here is how **[String.hashCode()](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/String.java" \l "String.hashCode%28%29" \t "_blank) and [String.equals()](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/String.java" \l "String.equals%28java.lang.Object%29" \t "_blank)** are implemented.

(b) Here is how **[Object.hashCode()](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/Object.java" \l "Object.hashCode%28%29" \t "_blank) and [Object.equals()](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/Object.java" \l "Object.equals%28java.lang.Object%29" \t "_blank)**are implemented.

ConcurrentHashMap Interview Questions In Java.

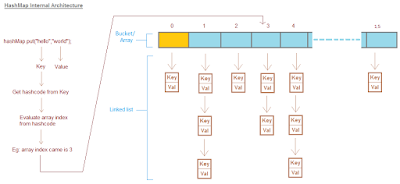
### Question 1.  What is the need of ConcurrentHashMap when there is HashMap and Hashtable already present?

**Performance and Thread safety are 2 parameter on which ConcurrentHashMap is focused.**

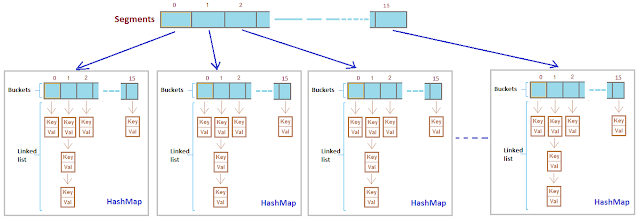
**Imagine a scenario where we have frequent reads(get) and less writes(put) and need thread safety,**  
  
**Can we use Hashtable in this scenario?**  
No. Hashtableis thread safe but give poor performance in case of multiple thread reading from hashtable because all methods of Hashtable including get() method is synchronized and due to which invokation to any method has to wait until any other thread working on hashtable complete its operation(get, put etc).  
  
**Can we use HashMap in this scenario?**  
No. Hashmap will solve performance issue by giving parallel access to multiple threads reading hashmap simultaneously.  
But Hashmap is not thread safe, so what will happen if one thread tries to put data and requires Rehashing and at same time other thread tries to read data from hashmap, It will go in infinite loop.  
**Infinite loop problem discussed in detail:**[**Infinite loop in HashMap**](http://javabypatel.blogspot.in/2016/01/infinite-loop-in-hashmap.html)  
  
  
**ConcurrentHashMap combines good features of hashmap and hashtable and solves performance and thread safety problem nicely.**

### Question 2.  HashMap and Hashtable uses Array and Linkedlist as datastructure to store data, How is it different in ConcurrentHashMap?

If you are not familiar with HashMap and Hashtable, Please go through it first:  
[How HashMap works](http://javabypatel.blogspot.in/2015/09/hashmap-data-structure-and-hashcode.html)   
  
Below diagram shows how hashtable/hashmap look like,

[](https://2.bp.blogspot.com/-9JEpgm-_nVw/V-pS4HnlajI/AAAAAAAABPE/p3t6GfbM4hsYrJvRhbs0u2Y8yrCu8-4tACLcB/s1600/hashmap-internal-structure.png)

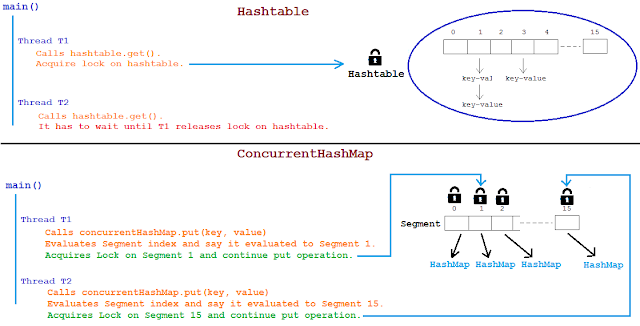
ConcurrentHashMap added one Array on top of it and each index of this additional array represents complete HashMap. Additional array is called Segment in ConcurrentHashMap.  
  
Architecture of ConcurrentHashMap looks like below,

[](https://4.bp.blogspot.com/-1W2vuBYf740/V-pnAFenccI/AAAAAAAABPU/lKSaVIfAzpMmrpRanTydHXSAA5uhC_p7wCLcB/s1600/consurrenthashmap-internal-structure.png)

**Putting key-value pair:**   
  
1. Putting key-value pair in ConcurrentHashMap requires first identifying exact index in   
    Segment array.   
    (Once Segment array index is identified, Now flow will be exactly same as putting the data in   
    hashmap/hashtable.)   
2. After identifying index in Segment array, next task is to identify index of internal bucket/array   
    present in internal hashmap as shown in figure above.   
 3. After identying bucket(internal array index), iterate key-value pairs and check each key with key   
    to store, wherever match is found replace stored value with value to store.  
    If there is no match, store key-value pair at the last of list.  
  
  
**Getting key-value pair:**  
  
1. Getting key-value pair in ConcurrentHashMap requires first identifying exact index in   
    Segment array.   
    (Once Segment array index is identified, Now flow will be exactly same as getting the data from   
    hashmap/hashtable.)   
2. After identifying index in Segment array, next task is to identify index of internal bucket/array   
    present in internal hashmap as shown in figure above.   
3. After identying bucket(internal array index), iterate key-value pairs and match each key with   
    given key, wherever match is found return value stored against key.  
    If there is no match, return null.

### Question 3.  How ConcurrentHashMap is efficient in terms of Performance and Thread safety?

ConcurrentHashMap provides better Performance by replacing the Hashtable's map wide lock to Segment level lock.  
  
Hashtable is not efficient beacause it uses map wide lock, it means lock is applied on map object itself,   
  
**So if 2 threads tries to call hashtable.get(key),**  
Thread T1 calls to get() method will acquire a lock on hashtable object and then execute get() method. (Lock is on complete 'hashtable object')  
  
Now if Thread T2 calls hashtable.get(key) method, then it will also try to acquire lock on hashtable object, but T2 will not able to acquire lock as lock on 'hashtable' is currently held by T1,   
  
So T2 waits until T1 finishes get() operation and release lock on hashtable object.

[](https://3.bp.blogspot.com/-aD_Z4JJtrhc/V-rLvoLHVaI/AAAAAAAABPk/M1VpuzyvZcY56vjBlnBGaczd-HifRuwvwCEw/s1600/hashtable-vs-concurrenthashmap-lock.png)

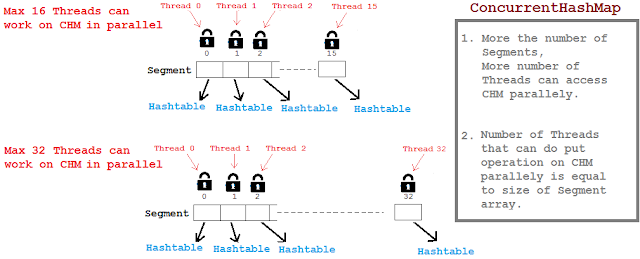
ConcurrentHashMap works bit different here and instead of locking complete map object it Locks per Segment.   
It means instead of single map wide lock, it has multiple Segment level lock.  
  
So 2 Threads can execute put operation simultaneously by acquiring lock on different Segments.  
  
Thread T1 calls concurrentHashMap.put(key, value), It acquires lock on say Segment 1 and invokes put method.  
Thread T2 calls concurrentHashMap.put(key, value), It acquires lock on say Segment 4 and invokes put method.  
  
Both threads doesn't interfere with each other and both can proceed simultaneously as they are working on separate Segment locks.  
  
**This is how ConcurrentHashMap improves Performance and provide Thread safety as well.**

### Question 4.  Can multiple threads read and write from same or different Segments of ConcurrentHashMap simultaneously?

**Read Operation: get(key)**  
**Same Segment/Different Segment : Yes.**   
Two threads T1 and T2 both can simultaneously read data from same Segment or different Segment of CHM simultaneously without blocking each other.  
  
  
**Write Operation: put(key, value)**  
**Different Segment :Yes**  
Multiple threads can write data to different Segment of CHM simultaneously without blocking each other.  
**Same Segment : No**    
Multiple threads CANNOT write data to same Segment of CHM simultaneously and need to wait for one thread to come write operation and then only other write operation can be proceed.   
     
  
**Read-Write Operation: get and put**  
**Say T1 is writing data in Segment 1 and T2 is reading data from same Segment 1, can read be allowed while writing operation is going on?**  
**YES.**  
Both operation that is T1 writing and T2 reading can be done parallely**.**  
  
**What data will T2 read if T1 is updating same data?**Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove).   
Latest updated value present will be returned by get operation that is value updated by most recently completed update operationswill be returned.  
  
**Note: Get operations are lock free and can be performed simulateneously irrespective of other thread writing data on same or different Segment of CHM.**

### Question 5.  What is the default size of Segment array? how it is tuned? What is ConcurrenyLevel in case of CHM?

**Default size of Segment array is 16.**  
   
ConcurrentHashMap differes from Hashtable in terms of Performance by introducing Segment array.  
Each index of Segment array is guarded by a lock for put operation.

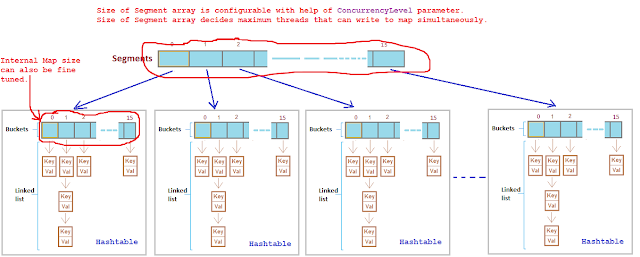
[](https://4.bp.blogspot.com/-icREX-PLyG0/V_YsC15kxrI/AAAAAAAABRc/CDLvHf-xyRkeu8U_UaZbfos55t0E5NvAgCLcB/s1600/concurrency-level-concurrenthashmap.png)

Threads working on separate Segments index doesn't affect each other.   
By default Segments array size is 16, So maximum 16 threads can simultaneously put data in map considering each thread is working on separate Segment array index.

### How Segment array size is tuned?

Segment size decides the number of Threads that can paralley write to a map.  
Segment array size is configured using ConcurrencyLevel parameter as shown below,

ConcurrentHashMap m = new ConcurrentHashMap(initialCapacity, loadFactor, concurrencyLevel)

[](https://4.bp.blogspot.com/-0HRMmJEuDys/V_YtFEKQ8BI/AAAAAAAABRk/Z195nyfoOwgwBc53mHn9pkVsFAm4ORYMQCLcB/s1600/consurrenthashmap-concurrency-level.png)

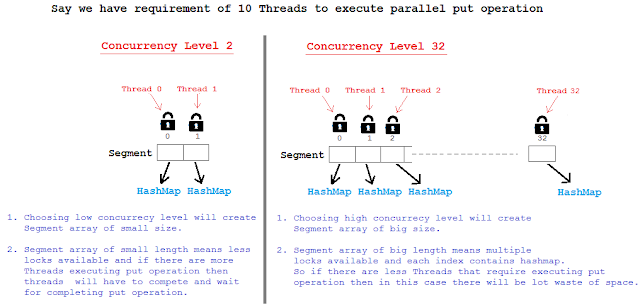
**It takes 3 parameters,**  
  
**Example:**

ConcurrentHashMap m = new ConcurrentHashMap(200 , 0.75f, 10);

**Initial capacity** is 200, it means CHM make sure it has space for adding 200 key-value pairs after creation.  
  
**Load factor**is 0.75, it means when average number of elements per map exceeds 150 (intital capacity \* load factor = 200 \* 0.75 = 150) at that time map size will be increased and existing items in map are rehashed to put in new larger size map.  
For more details on Load Factor: [**Load factor in Map**](http://javabypatel.blogspot.in/2015/10/what-is-load-factor-and-rehashing-in-hashmap.html)  
  
**Concurrency level**is 10, it means at any given point of time Segment array size will be 10 or greater than 10, so that 10 threads can able to parallely write to a map.

### Question 6.  What will happen if the size of Segment array is too small or too large?

Choosing correct **ConcurrencyLevel** is very important because ConcurrencyLevel decides what will be the size of Segment array.  
  
Segment array size will decide how many parallel Threads will be able to execute put operation on map parallely.

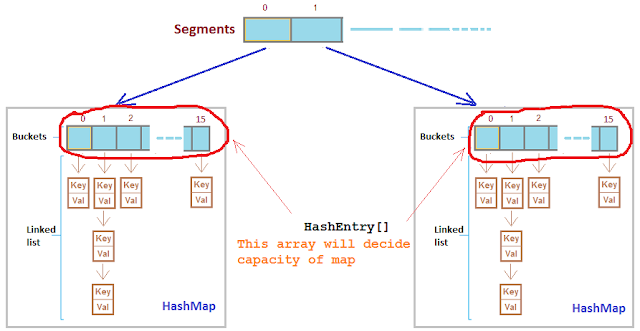
[](https://2.bp.blogspot.com/-Dpl-QWxn8jM/V_CbnodjnOI/AAAAAAAABQc/TZwMkByf6RcEK6iC6BTcklkTnNIeiirnACLcB/s1600/concurrency-level-too-small-large.png)

So Segment array size should not be too big or should not be too small because,   
Using a significantly higher value than we will waste space and time, and a significantly lower value can lead to thread competition. 

### Question 6.  If we choose ConcurrenyLevel as 10 then what will be size of Segment array? Is Segment array size exactly same as concurrenyLevel? If No, then how is the Segment array size calculated?

Segment array size is calculated based on concurrenyLevel specified but it doesn't mean it will be exactly same as concurrenyLevel.  
  
**If concurrenyLevel is 10 then Segment array size will be 16.**  
  
Segment array size = 2 to the power x, where result should be >= concurrenyLevel(in our case it is 10)  
Segment array size = 2 to the power x >= 10  
  
Segment array size = 2 ^ 1 = 2   >= 10 (False)  
Segment array size = 2 ^ 2 = 4   >= 10 (False)  
Segment array size = 2 ^ 3 = 8   >= 10 (False)  
Segment array size = 2 ^ 4 = 16 >= 10 **(True)**  
 **Segment array size is 16.**  
  
**Example: 2**  
concurrenyLevel = 8 then Segment array size = ?  
Find 2 ^ x >= 8   
  
2 ^ 1 >= 2   
2 ^ 2 >= 4   
2 ^ 3 >= 8   
Segment array size will be 8.

### Question 7.  What is HashEntry array and how is the size of HashEntry decided?

Default initial capacity of CHM is 16. It means CHM make sure there is sufficient space to accomodate 16 key-value pairs after CHM is created.  
  
**What is HashEntry[]?**  
  
We saw that each index of Segment array itself i**s**complete HashMap,   
[](https://2.bp.blogspot.com/-5XRckpIGUFU/V_Hk_Hvj9AI/AAAAAAAABQ8/qhwhnlGcrrscqhIcBrBJRXcQu34eJKL-wCLcB/s1600/consurrenthashmap-capacity.png)    
  
**In HashMap the bucket/array is of class Entry[] and in CHM the array is of class HashEntry[].**

static final class Segment<K,V> extends ReentrantLock implements Serializable {

transient volatile HashEntry<K,V>[] table;

}

static final class HashEntry<K,V> {

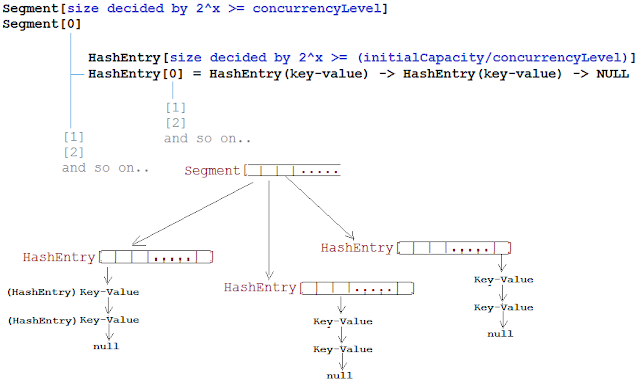
final int hash;

final K key;

volatile V value;

volatile HashEntry<K,V> next;

}

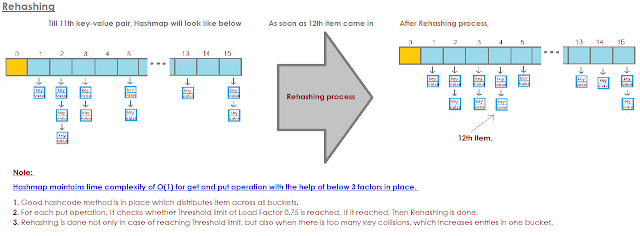
**Lets see how internal architecture of CHM looks like,**  
  
[](https://2.bp.blogspot.com/-SHDKO9Lb3ok/V_HuycFrD0I/AAAAAAAABRM/N7qw1kN2WcEou4zE99hjycjs2AdNAZ8UwCLcB/s1600/segment-hashentry-array.png)  
  
**How HashEntry[] array size will is calculated?**

ConcurrentHashMap m = new ConcurrentHashMap(initialCapacity, loadFactor, concurrencyLevel)

HashEntry[] array size  =   2 ^ x   >=  (initialCapacity / concurrenyLevel)  
  
**Eg: ConcurrentHashMap(32,   0.75f,   4);**  
HashEntry[] array size  =  2 ^ 1 = 2   >=  8(32/4) (False)  
HashEntry[] array size  =  2 ^ 2 = 4   >=  8 (False)  
HashEntry[] array size  =  2 ^ 3 = 8   >=  8 **(True)**  
HashEntry[] array size =8**.**  
**It means there will always be capacity of 8 key-value pairs that can be put in CHM after its creation.**

### Question 8.  How does ConcurrentHashMap handle rehashing while another thread is still writing on another segment/partition?

For understanding rehashing, please understand Load Factor first.  
**Load Factor**is a measure, which decides when exactly to increase the HashMap/CHM capacity(buckets) to maintain get and put operation complexity of O(1).  
  
Default load factor of Hashmap/CHM is 0.75f (i.e 75% of current map size).  
  
**For more details on Load factor, Please refer:**[**Load Factor in HashMap**](http://javabypatel.blogspot.in/2015/10/what-is-load-factor-and-rehashing-in-hashmap.html)

[](https://3.bp.blogspot.com/-eHGD8zneRbI/VhNYVf6_AaI/AAAAAAAAAd8/BaP2VrEfD7QEnLICoHoOl27LZoUNzE7ywCPcB/s1600/hashmapRehashing.png)

**In CHM, Every segment is separately rehashed so there is no collision between Thread 1 writing to Segment index 2 and Thread 2 writing to Segment index 5.**  
  
**Example:** If say Thread 1 which is putting data in Segment[] array index 2 finds that HashEntry[] array needs to be rehashed due to exceed Load factor capacity then it will rehash HashEntry[] array present at Segment[] array index 2 only.  
HashEntry[] array at other Segment indexes will still be intact, unaffected and continue to serve put and get request parallely.

<http://javabypatel.blogspot.in/2016/09/concurrenthashmap-interview-questions.html>

## Top 10 Mistakes Java Developers Make

1. **Convert Array to ArrayList**

To convert an array to an ArrayList, developers often do this:

List<String> list = Arrays.asList(arr);  
Arrays.asList() will return an ArrayList which is a private static class inside Arrays, it is not the java.util.ArrayList class. The java.util.Arrays.ArrayList class has set(), get(), contains() methods, but does not have any methods for adding elements, so its size is fixed. To create a real ArrayList, you should do:

ArrayList<String> arrayList = new ArrayList<String>(Arrays.asList(arr));  
The constructor of ArrayList can accept a Collection type, which is also a super type for java.util.Arrays.ArrayList.

**2. Check If an Array Contains a Value**

Developers often do:

Set<String> set = new HashSet<String>(Arrays.asList(arr));  
return set.contains(targetValue);  
The code works, but there is no need to convert a list to set first. Converting a list to a set requires extra time. It can as simple as:

Arrays.asList(arr).contains(targetValue);  
or

for(String s: arr){  
if(s.equals(targetValue))  
return true;  
}  
return false;  
The first one is more readable than the second one.

**3. Remove an Element from a List Inside a Loop**

Consider the following code which removes elements during iteration:

ArrayList<String> list = new ArrayList<String>(Arrays.asList(“a”, “b”, “c”, “d”));  
for (int i = 0; i < list.size(); i++) {  
list.remove(i);  
}  
System.out.println(list);  
The output is:

[b, d]  
There is a serious problem in this method. When an element is removed, the size of the list shrinks and the index changes. So if you want to delete multiple elements inside a loop by using the index, that will not work properly.

You may know that using iterator is the right way to delete elements inside loops, and you know foreach loop in Java works like an iterator, but actually it is not. Consider the following code:

ArrayList<String> list = new ArrayList<String>(Arrays.asList(“a”, “b”, “c”, “d”));

for (String s : list) {  
if (s.equals(“a”))  
list.remove(s);  
}  
It will throw out ConcurrentModificationException.

Instead the following is OK:

ArrayList<String> list = new ArrayList<String>(Arrays.asList(“a”, “b”, “c”, “d”));  
Iterator<String> iter = list.iterator();  
while (iter.hasNext()) {  
String s = iter.next();

if (s.equals(“a”)) {  
iter.remove();  
}  
}  
.next() must be called before .remove(). In the foreach loop, compiler will make the .next() called after the operation of removing element, which caused the ConcurrentModificationException. You may want to take a look at the source code of ArrayList.iterator().

**4. Hashtable vs HashMap**

By conventions in algorithm, Hashtable is the name of the data structure. But in Java, the data structure’s name is HashMap. One of the key differences between Hashtable and HashMap is that Hashtable is synchronized. So very often you don’t need Hashtable, instead HashMap should be used.

HashMap vs. TreeMap vs. Hashtable vs. LinkedHashMap  
Top 10 questions about Map

**5. Use Raw Type of Collection**

In Java, raw type and unbounded wildcard type are easy to mixed together. Take Set for example, Set is raw type, while Set<?> is unbounded wildcard type.

Consider the following code which uses a raw type List as a parameter:

public static void add(List list, Object o){  
list.add(o);  
}  
public static void main(String[] args){  
List<String> list = new ArrayList<String>();  
add(list, 10);  
String s = list.get(0);  
}  
This code will throw an exception:

Exception in thread “main” java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String  
at …  
Using raw type collection is dangerous as the raw type collections skip the generic type checking and not safe. There are huge differences between Set, Set<?>, and Set<Object>. Check out  
Raw type vs. Unbounded wildcard and Type Erasure.

**6. Access Level**

Very often developers use public for class field. It is easy to get the field value by directly referencing, but this is a very bad design. The rule of thumb is giving access level for members as low as possible.

public, default, protected, and private

**7. ArrayList vs. LinkedList**

When developers do not know the difference between ArrayList and LinkedList, they often use ArrayList, because it looks familiar. However, there is a huge performance difference between them. In brief, LinkedList should be preferred if there are a large number of add/remove operations and there are not a lot of random access operations. Check out ArrayList vs. LinkedList to get more information about their performance if this is new to you.

**8. Mutable vs. Immutable**

Immutable objects have many advantages such simplicity, safety, etc. But it requires a separate object for each distinct value, and too many objects might cause high cost of garbage collection. There should be a balance when choosing between mutable and immutable.

In general, mutable objects are used to avoid producing too many intermediate objects. One classic example is concatenating a large number of strings. If you use an immutable string, you would produce a lot of objects that are eligible for garbage collection immediately. This wastes time and energy on the CPU, using a mutable object the right solution (e.g. StringBuilder).

String result=””;  
for(String s: arr){  
result = result + s;  
}  
There are other situations when mutable objects are desirable. For example passing mutable objects into methods lets you collect multiple results without jumping through too many syntactic hoops. Another example is sorting and filtering: of course, you could make a method that takes the original collection, and returns a sorted one, but that would become extremely wasteful for larger collections. (From dasblinkenlight’s answer on Stack Overflow)

Why String is Immutable?

**9. Constructor of Super and Sub**

This compilation error occurs because the default super constructor is undefined. In Java, if a class does not define a constructor, compiler will insert a default no-argument constructor for the class by default. If a constructor is defined in Super class, in this case Super(String s), compiler will not insert the default no-argument constructor. This is the situation for the Super class above.

The constructors of the Sub class, either with-argument or no-argument, will call the no-argument Super constructor. Since compiler tries to insert super() to the 2 constructors in the Sub class, but the Super’s default constructor is not defined, compiler reports the error message.

To fix this problem, simply 1) add a Super() constructor to the Super class like

public Super(){  
System.out.println(“Super”);  
}  
, or 2) remove the self-defined Super constructor, or 3) add super(value) to sub constructors.

Constructor of Super and Sub

**10. “” or Constructor?**

String can be created by two ways:

//1. use double quotes  
String x = “abc”;  
//2. use constructor  
String y = new String(“abc”);  
What is the difference?

The following examples can provide a quick answer:

String a = “abcd”;  
String b = “abcd”;  
System.out.println(a == b); // True  
System.out.println(a.equals(b)); // True

String c = new String(“abcd”);  
String d = new String(“abcd”);  
System.out.println(c == d); // False  
System.out.println(c.equals(d)); // True

Hashtable :

Key value pair, No ordering, can access randomly, No duplicate key, No Nulls, is Thread Safe- lock whole Hashtable object to make It synchronoised.

Similar to HashMap, do not allows null values or keys, entire map is locked for thread saftyl.

TreeMap:

Key value pair, Ordering, can access randomly, No duplicate allowed, No Null,

No thread safe, No blocking operation, No upper bonds.

Very nice alternative for HashMap if sorted key are important.

1. The goal of the TreeMap is to have a tree of keys where keys that are lower than the parent's key are to the left and keys higher than the parent's key are to the right. So, if you add C, then E, you will have this tree:
2. C
3. \

E

If you then add D, initially you will have:

C

\

E

/

D

But this tree is unbalanced and therefore searches would be slower. So, the tree is rebalanced. After balancing, the tree now becomes much more efficient:

C C

\ rotate \ rotate D

E --- right ---> D --- left ---> / \

/ around \ around C E

D E E D

1. Rebalancing takes place inside the fixAfterInsertion() method, which checks whether the *red-black properties* of the tree are still maintained after insertion. And, if it doesn't, then it rebalances the tree performing either rotateLeft() or rotateRight() on the offending branch to restore the balance. Then it moves up the tree and checks the balance and so on until it reaches the root node.