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. |