|  |  |  |
| --- | --- | --- |
| Default method Implementation | It can have default method implementation | Interfaces are pure abstraction.It can not have implementation at all but in java 8, you can have default methods in interface. |
| Implementation | Subclasses use **extends**keyword to extend an abstract class and they need to provide implementation of all the declared methods in the abstract class unless the subclass is also an abstract class | subclasses use **implements**keyword to implement interfaces and should provide implementation for all the methods declared in the interface |
| Constructor | Abstract class can have constructor | Interface  can not have constructor |
| Different from normal java class | Abstract classes are almost same as java classes except you can not instantiate it. | Interfaces are altogether different type |
| Access Modifier | Abstract class methods can have public ,protected,private and default modifier | Interface methods are by default public. you can not use any other access modifier with it |
| Main() method | Abstract classes can have main method so we can run it | Interface do not have main method so we can not run it. |
| Multiple inheritance | Abstract class can extends one other class and can implement one or more interface. | Interface can extends to one or more interfaces only |
| Speed | It is faster than interface | Interface is somewhat slower as it takes some time to find implemented method in class |
| Adding new method | If you add new method to abstract class, you can provide default implementation of it. So you don’t need to change your current code | If you add new method to interface, you have to change the classes which are implementing that interface |

Why interface variable is final?

(*This is not a philosophical answer but more of a practical one*). The requirement for staticmodifier is obvious which has been answered by others. Basically, since the interfaces cannot be instantiated, the only way to access its fields are to make them a class field -- static.

The reason behind the interface fields automatically becoming final (constant) is to prevent different implementations accidentally changing the value of interface variable which can inadvertently affect the behaviour of the other implementations. Imagine the scenario below where an interface property did not explicitly become final by Java:

public interface Actionable {

public static boolean isActionable = false;

public void performAction();

}

public NuclearAction implements Actionable {

public void performAction() {

// Code that depends on isActionable variable

if (isActionable) {

// Launch nuclear weapon!!!

}

}

}

Now, just think what would happen if another class that implements Actionable alters the state of the interface variable:

public CleanAction implements Actionable {

public void performAction() {

// Code that can alter isActionable state since it is not constant

isActionable = true;

}

}

If these classes are loaded within a single JVM by a classloader, then the behavior of NuclearAction can be affected by another class, CleanAction, when its performAction() is invoke after CleanAction's is executed (in the same thread or otherwise), which in this case can be disastrous (semantically that is).

Since we do not know how each implementation of an interface is going to use these variables, they must implicitly be final.

When to use abstract method and interface

Main Link

<https://stackoverflow.com/questions/18777989/how-should-i-have-explained-the-difference-between-an-interface-and-an-abstract/34978606#34978606>

*All your statements are valid except your first statement (after the Java 8 release):*

Methods of a Java interface are implicitly abstract and cannot have implementations

From the documentation [page](https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html):

What is an interface ?

An interface is a reference type, similar to a class, that can contain only *constants, method signatures, default methods, static methods,and nested types*

Method bodies exist only for default methods and static methods.

*Default methods:*

An interface can have [default methods](https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html), but are different than abstract methods in abstract classes.

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

When you extend an interface that contains a default method, you can do the following:

1. Not mention the default method at all, which lets your extended interface inherit the default method.
2. Redeclare the default method, which makes it abstract.
3. Redefine the default method, which overrides it.

*Static Methods:*

In addition to default methods, you can define static methods in interfaces. (A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.)

This makes it easier for you to organize helper methods in your libraries;

Example code from documentation page about interface having static and default methods.

import java.time.\*;

public interface TimeClient {

void setTime(int hour, int minute, int second);

void setDate(int day, int month, int year);

void setDateAndTime(int day, int month, int year,

int hour, int minute, int second);

LocalDateTime getLocalDateTime();

static ZoneId getZoneId (String zoneString) {

try {

return ZoneId.of(zoneString);

} catch (DateTimeException e) {

System.err.println("Invalid time zone: " + zoneString +

"; using default time zone instead.");

return ZoneId.systemDefault();

}

}

default ZonedDateTime getZonedDateTime(String zoneString) {

return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));

}

}

Use the below guidelines to chose whether to use an interface or abstract class.

***Interface:***

1. To define a ***contract*** ( preferably stateless - I mean no variables )
2. To link unrelated classes with ***has a*** capabilities.
3. To declare public constant variables (***immutable state***)

***Abstract class:***

1. Share code among several closely related classes. It establishes ***is a*** relation.
2. Share common state among ***related classes*** ( state can be modified in concrete classes)

Related posts:

[Interface vs Abstract Class (general OO)](https://stackoverflow.com/questions/761194/interface-vs-abstract-class-general-oo/33963650#33963650)

[Implements vs extends: When to use? What's the difference?](https://stackoverflow.com/questions/10839131/implements-vs-extends-when-to-use-whats-the-difference/34977257#34977257)

By going through these examples, you can understand that

***Unrelated classes can have capabilities through interface but related classes change the behaviour through extension of base classes.***

If you are looking at java as OOP language,

"***interface does not provide method implementation***" is no longer valid with Java 8 launch. Now java provides implementation in interface for default methods.

In simple terms, I would like to use

***interface:*** To implement a contract by multiple unrelated objects. It provides "**HAS A**" capability.

***abstract class:*** To implement the same or different behaviour among multiple related objects. It establishes "**IS A**" relation.

Oracle [website](https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html) provides key differences between interface and abstract class.

*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.

Example:

Abstract class ( **IS A** relation)

[Reader](https://docs.oracle.com/javase/8/docs/api/java/io/Reader.html) is an abstract class.

[BufferedReader](https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html) is a Reader

[FileReader](https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html) is a Reader

FileReader and BufferedReader are used for common purpose : Reading data, and they are related through Reader class.

Interface ( **HAS A** capability )

[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html) is an interface.

Assume that you have two classes in your application, which are implementing Serializableinterface

Employee implements Serializable

Game implements Serializable

Here you can't establish any relation through Serializable interface between Employee and Game, which are meant for different purpose. Both are capable of Serializing the state and the comparasion ends there.

Abstract classes can be Derived whereas Interfaces can be Implemented. There is some difference between the two. When you derive an Abstract class, the relationship between the derived class and the base class is 'is a' relationship. e.g., a Dog is an Animal, a Sheep is an Animal which means that a Derived class is inheriting some properties from the base class.

Whereas for implementation of interfaces, the relationship is "can be". e.g., a Dog can be a spy dog. A dog can be a circus dog. A dog can be a race dog. Which means that you implement certain methods to acquire something.

Qualifiier

**Spring** @**Qualifier** Annotation. There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property. In such cases, you can use the @**Qualifier** annotation along with @Autowired to remove the confusion by specifying which exact bean will be wired.

**declaration:** A statement that establishes an identifier and associates attributes with it, without necessarily reserving its storage (for data) or providing the implementation (for methods).

**definition:** A declaration that reserves storage (for data) or provides implementation (for methods)

Why abstract class have constructors and interface don`t have constructor

Interfaces in Java don’t have constructor because all data members in interfaces are public static final by default, they are constants(assign values at the time of declaration) .There are no data members in interfaces to initialize them through constructor.

But in abstract class you require constructor to initialize the data members and instance variables.

Why abstract class cannot be initiated

The abstract keyword ensures that no one would accidentally initiate this incomplete class.

Think of repairing a car. Someone has removed the brake pads and is going to replace them in the next day. Now, to prevent someone accidentally driving this car(which has no brakes installed), the mechanic installs a lock on the steering wheel. It's a fail-safe measure.

What is a concrete class

A **concrete class** is a **class** that has an implementation for all of its methods that were inherited from abstract or implemented via interfaces. It also does not define any abstract methods of its own. ... Therefore it can be inferred that any **class** that is not an abstract **class** or interface is a **concrete class**

When does JVM allocates memory ?

Compile time no memory allocation happens. Only at load and runtime.

Compile time generates .class files that's it.

Remember you need to have a main class to run the program. When you run your program using Java with classpath to .class file, there will be steps like loading & linking etc.,

Classloaders loads the files to permgen.

When main method invoked, there will be stack created and local variables will be placed there

When runtime encounters **new** it creates object on heap and allocates required memory there like memory required for Test.

Reference types

when you create an object from a class, Java allocates the amount of memory the object requires to store the object. Then, if you assign the object to a variable, the variable is actually assigned a *reference*to the object, not the object itself. This reference is the address of the memory location where the object is stored.

For example, the following statement defines a variable that can reference objects created from a class named Ball:

Ball b;

To create a new instance of an object from a class, you use the new keyword along with the class name:

Ball b = new Ball();

When to Use List and Map ?

Lists and Maps are different data structures. Maps are used for when you want to associate a key with a value and Lists are an ordered collection.

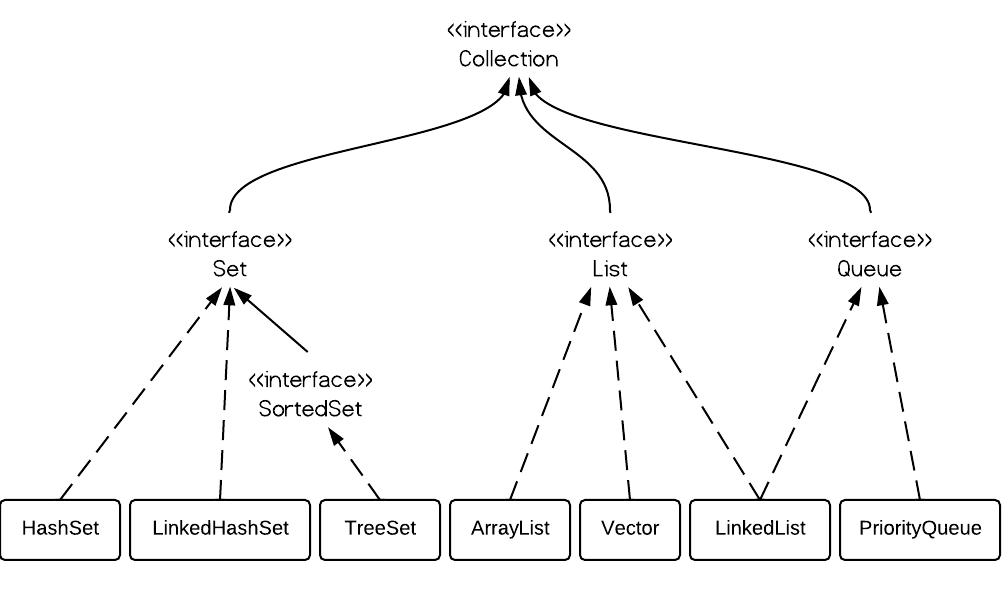
Map is an interface in the Java Collection Framework and a HashMap is one implementation of the Map interface. HashMap are efficient for locating a value based on a key and inserting and deleting values based on a key. The entries of a HashMap are not ordered.

ArrayList and LinkedList are an implementation of the List interface. LinkedList provides sequential access and is generally more efficient at inserting and deleting elements in the list, however, it is it less efficient at accessing elements in a list. ArrayList provides random access and is more efficient at accessing elements but is generally slower at inserting and deleting elements.

# ArrayList vs. LinkedList vs. Vector

https://dzone.com/articles/arraylist-vs-linkedlist-vs

**1. List Overview**

List, as its name indicates, is an ordered sequence of elements. When we talk about List, it is a good idea to compare it with Set which is a set of elements which is unordered and every element is unique.  The following is the class hierarchy diagram of Collection.  

**2. ArrayList vs. LinkedList vs. Vector**

From the hierarchy diagram, they all implement List interface. They are very similar to use. Their main difference is their implementation which causes different performance for different operations.  ArrayList is implemented as a resizable array. As more elements are added to ArrayList, its size is increased dynamically. It's elements can be accessed directly by using the get and set methods, since ArrayList is essentially an array. LinkedList is implemented as a double linked list. Its performance on add and remove is better than Arraylist, but worse on get and set methods. Vector is similar with ArrayList, but it is synchronized. ArrayList is a better choice if your program is thread-safe. Vector and ArrayList require space as more elements are added. Vector each time doubles its array size, while ArrayList grow 50% of its size each time. LinkedList, however, also implements Queue interface which adds more methods than ArrayList and Vector, such as offer(), peek(), poll(), etc.    Note: The default initial capacity of an ArrayList is pretty small. It is a good habit to construct the ArrayList with a higher initial capacity. This can avoid the resizing cost.

**3. ArrayList example**

ArrayList al = new ArrayList();

al.add(3);

al.add(2);

al.add(1);

al.add(4);

al.add(5);

al.add(6);

al.add(6);

Iterator iter1 = al.iterator();

while(iter1.hasNext()){

System.out.println(iter1.next());

}

**4. LinkedList example**

LinkedList ll = new LinkedList();

ll.add(3);

ll.add(2);

ll.add(1);

ll.add(4);

ll.add(5);

ll.add(6);

ll.add(6);

Iterator iter2 = al.iterator();

while(iter2.hasNext()){

System.out.println(iter2.next());

}

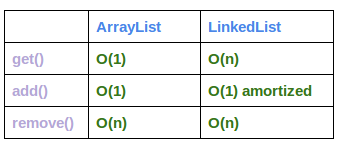
As shown in the examples above, they are similar to use. The real difference is their underlying implementation and their operation complexity.

**5. Vector**

Vector is almost identical to ArrayList, and the difference is that Vector is synchronized. Because of this, it has an overhead than ArrayList. Normally, most Java programmers use ArrayList instead of Vector because they can synchronize explicitly by themselves.

**6. Performance of ArrayList vs. LinkedList**

The time complexity comparison is as follows:

[](http://www.programcreek.com/wp-content/uploads/2013/03/arraylist-vs-linkedlist-complexity.png)

I use the following code to test their performance:

ArrayList arrayList = new ArrayList();

LinkedList linkedList = new LinkedList();

// ArrayList add

long startTime = System.nanoTime();

for (int i = 0; i < 100000; i++) {

arrayList.add(i);

}

long endTime = System.nanoTime();

long duration = endTime - startTime;

System.out.println("ArrayList add: " + duration);

// LinkedList add

startTime = System.nanoTime();

for (int i = 0; i < 100000; i++) {

linkedList.add(i);

}

endTime = System.nanoTime();

duration = endTime - startTime;

System.out.println("LinkedList add: " + duration);

// ArrayList get

startTime = System.nanoTime();

for (int i = 0; i < 10000; i++) {

arrayList.get(i);

}

endTime = System.nanoTime();

duration = endTime - startTime;

System.out.println("ArrayList get: " + duration);

// LinkedList get

startTime = System.nanoTime();

for (int i = 0; i < 10000; i++) {

linkedList.get(i);

}

endTime = System.nanoTime();

duration = endTime - startTime;

System.out.println("LinkedList get: " + duration);

// ArrayList remove

startTime = System.nanoTime();

for (int i = 9999; i >=0; i--) {

arrayList.remove(i);

}

endTime = System.nanoTime();

duration = endTime - startTime;

System.out.println("ArrayList remove: " + duration);

// LinkedList remove

startTime = System.nanoTime();

for (int i = 9999; i >=0; i--) {

linkedList.remove(i);

}

endTime = System.nanoTime();

duration = endTime - startTime;

System.out.println("LinkedList remove: " + duration);

And the output is:

ArrayList add: 13265642

LinkedList add: 9550057

ArrayList get: 1543352

LinkedList get: 85085551

ArrayList remove: 199961301

LinkedList remove: 85768810

The difference of their performance is obvious. LinkedList is faster in add and remove, but slower in get. Based on the complexity table and testing results, we can figure out when to use ArrayList or LinkedList.

In brief, LinkedList should be preferred if:

* there are no large number of random access of element
* there are a large number of add/remove operations

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **No.** | **Method Overloading** | **Method Overriding** | | 1) | Method overloading is used to increase the readability of the program. | Method overriding is used to provide the specific implementation of the method that is already provided by its super class. | | 2) | Method overloading is performed within class. | Method overriding occurs in two classes that have IS-A (inheritance) relationship. | | 3) | In case of method overloading, parameter must be different. | In case of method overriding, parameter must be same. | | 4) | Method overloading is the example of compile time polymorphism. | Method overriding is the example of run time polymorphism. | | 5) | In java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading. But you must have to change the parameter. | Return type must be same or covariant in method overriding. |  Java Method Overloading example  1. **class** OverloadingExample{ 2. **static** **int** add(**int** a,**int** b){**return** a+b;} 3. **static** **int** add(**int** a,**int** b,**int** c){**return** a+b+c;} 4. }  Java Method Overriding example  1. **class** Animal{ 2. **void** eat(){System.out.println("eating...");} 3. } 4. **class** Dog **extends** Animal{ 5. **void** eat(){System.out.println("eating bread...");} 6. } |

**Method Overloading:** Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, drawString, drawInteger, drawFloat, and so on. In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.

public class DataArtist {

...

public void draw(String s) {

...

}

public void draw(int i) {

...

}

public void draw(double f) {

...

}

public void draw(int i, double f) {

...

}

}

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

**Method overriding**, in object oriented programming, is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes. The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a method that has same name, same parameters or signature, and same return type as the method in the parent class. The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.