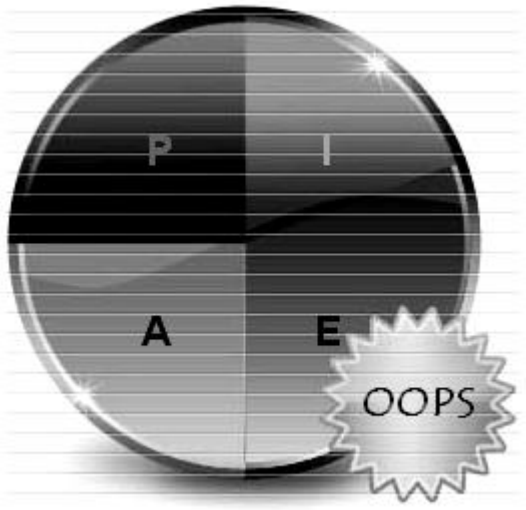
Chapter 1 - OOPS

Object Oriented Programming Systems (OOPS)

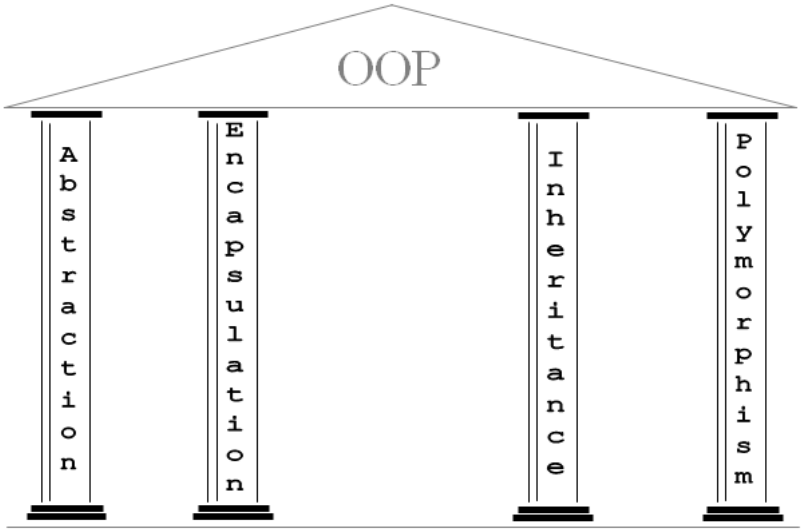
A PIE – **A**bstraction, **P**olymorphism, **I**nheritance and **E**ncapsulation.



Object Orientation involving encapsulation, inheritance, polymorphism, and abstraction, is an important approach in programming and program design. It is widely accepted and used in industry and is growing in popularity.

So, the four principle concepts upon which object oriented design and programming rest are:

- Abstraction
- Polymorphism
- Inheritance
- Encapsulation



Abstraction

Abstraction refers to the act of representing essential features without including the background details or explanations.

Encapsulation

Encapsulation is a technique used for hiding the properties and behaviors of an object and allowing outside access only as appropriate. It prevents other objects from directly altering or accessing the properties or methods of the encapsulated object.

Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class.

- A class that is inherited is called a superclass.
- The class that does the inheriting is called a subclass.
- Inheritance is done by using the keyword extends.
- The two most common reasons to use inheritance are:

    1. To promote code reuse
    2. To use polymorphism

Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass. Shared characteristics can be attributes, associations, or methods.

Polymorphism

Polymorphism is briefly described as "one interface, many implementations." Polymorphism is a characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form.

Different forms of Polymorphism

There are two types of polymorphism one is Compile time polymorphism and the other is run time polymorphism. Compile time polymorphism is method overloading. Runtime time polymorphism is method overriding done using inheritance and interface.

Note: From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in Java:

- Method overloading
- Method overriding through inheritance
- Method overriding through the Java interface

Inheritance, Overloading and Overriding are used to achieve Polymorphism in java. Polymorphism manifests itself in Java in the form of multiple methods having the same name.
In some cases, multiple methods have the same name, but different formal argument lists (overloaded methods). In other cases, multiple methods have the same name, same return type, and same formal argument list (overridden methods).

Runtime polymorphism or Dynamic method dispatch

In Java, runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance.

Method Overloading

Method Overloading means to have two or more methods with same name in the same class with different arguments. The benefit of method overloading is that it allows you to implement methods that support the same semantic operation but differ by argument number or type.

- Overloaded methods MUST change the argument list
- Overloaded methods CAN change the return type
- Overloaded methods CAN change the access modifier
- Overloaded methods CAN declare new or broader checked exceptions
- A method can be overloaded in the same class or in a subclass

Method Overriding

Method overriding occurs when sub class declares a method that has the same type arguments as a method declared by one of its superclass. The key benefit of overriding is the ability to define behavior that's specific to a particular subclass type.

- The overriding method cannot have a more restrictive access modifier than the method being overridden (Ex: You can't override a method marked public and make it protected).
- You cannot override a method marked final
- You cannot override a method marked static

	Overloaded Method	Overridden Method
Arguments	Must change	Must not change
Return type	Can change	Can't change except for covariant returns
Exceptions	Can change	Can reduce or eliminate. Must not throw new or broader checked exceptions
Access	Can change	Must not make more restrictive (can be less restrictive)
Invocation	Reference type determines which overloaded version is selected. Happens at compile time.	Object type determines which method is selected. Happens at runtime.

Difference between Abstraction and Encapsulation

Abstraction focuses on the outside view of an object (i.e. the interface) Encapsulation (information hiding) prevents clients from seeing it's inside view, where the behavior of the abstraction is implemented.

Abstraction solves the problem in the design side while Encapsulation is the Implementation.
Encapsulation is the deliverables of Abstraction. Encapsulation barely talks about grouping up your abstraction to suit the developer needs.

Interface versus Abstract Class

1. Abstract class is a class which contains one or more abstract methods, which has to be implemented by sub classes. An abstract class can contain no abstract methods also i.e. abstract class may contain concrete methods. A Java Interface can contain only method declarations and public static final constants and doesn't contain their implementation. The classes which implement the Interface must provide the method definition for all the methods present.

2. Abstract class definition begins with the keyword "abstract" keyword followed by Class definition. An Interface definition begins with the keyword "interface".

3. Abstract classes are useful in a situation when some general methods should be implemented and specialization behavior should be implemented by subclasses. Interfaces are useful in a situation when all its properties need to be implemented by subclasses

4. All variables in an Interface are by default - public static final while an abstract class can have instance variables.

5. An interface is also used in situations when a class needs to extend another class apart from the abstract class. In such situations it's not possible to have multiple inheritances of classes. An interface on the other hand can be used when it is required to implement one or more interfaces. Abstract class does not support Multiple Inheritance whereas an Interface supports multiple Inheritances.

6. An Interface can only have public members whereas an abstract class can contain private as well as protected members.

7. A class implementing an interface must implement all of the methods defined in the interface, while a class extending an abstract class need not implement any of the methods defined in the abstract class.

8. The problem with an interface is, if you want to add a new feature (method) in its contract, then you MUST implement those methods in all of the classes which implement that interface. However, in the case of an abstract class, the method can be simply implemented in the abstract class and the same can be called by its subclass

9. Interfaces are slow as it requires extra indirection to to find corresponding method in in the actual class. Abstract classes are fast

10. Interfaces are often used to describe the peripheral abilities of a class, and not its central identity, E.g. an Automobile class might implement the Recyclable interface, which could apply to many otherwise totally unrelated objects.

Chapter 2 - Core Java

JVM - Write Once, Run Everywhere

JVM, or the Java Virtual Machine, is an interpreter which accepts 'Byte code' and executes it.

Java has been termed as a 'Platform Independent Language' as it primarily works on the notion of 'compile once, run everywhere'. Here's a sequential step establishing the Platform independence feature in Java:

- The Java Compiler outputs Non-Executable Codes called 'Byte code'.
- Byte code is a highly optimized set of computer instruction which could be executed by the Java Virtual Machine (JVM).
- The translation into Byte code makes a program easier to be executed across a wide range of platforms, since all we need is a JVM designed for that particular platform.
- JVMs for various platforms might vary in configuration, those they would all understand the same set of Byte code, thereby making the Java Program 'Platform Independent'.

JDK versus JRE

The "JDK" is the Java Development Kit. i.e., the JDK is bundle of software that you can use to develop Java based software. Typically, each JDK contains one (or more) JRE's along with the various development tools like the Java source compilers, bundling and deployment tools, debuggers, development libraries, etc.

The "JRE" is the Java Runtime Environment. I.e., the JRE is an implementation of the Java Virtual Machine which actually executes Java programs.

Java History

Java SE 8 - Jigsaw 2014

Java SE 7 - Dolphin 2011

Java SE 6 - Mustang 2006

- JDBC 4.0 support
- Support for pluggable annotations

J2SE 5.0 (1.5) - Tiger 2004

1. **Scanner** - Used to convert text into primitives or Strings

2. **Printf** - The format() and printf() methods were added to java.io.PrintStream

3. **Auto-boxing** - Eliminates the manual conversion between primitive types(int)and wrapper types(Integer)

4. **Type safe Enums** - Type Code "enum" (new keyword) with methods and fields

5. **For each loop (Enhanced For Loop)** - Eliminates the error-proneness of iterators

6. **Generics** - Adds compile-time type safety to Collections Framework and eliminates the need for casting

7. **Static Imports** - Import static fields and methods

8. **VarArgs** - Eliminates the need for manual boxing up argument list into an array

9. **Metadata (Annotations)** - This leads to a "declarative" programming style where the programmer says what should be done and tools emit the code to do it.

10. **Reflection** - Added support for generics, annotations and enums

11. **Collections Framework** - Three new interfaces (Queue -2 - Concurrent, Blocking Queue - 5 - Concurrent, ConcurrentMap - 1 - ConcurrentHashMap) Copy-on-write List and Set implementations - CopyOnWriteArrayList

12. **Concurrent Utilities** – It provides a powerful, extensible framework of high performance, scalable and thread-safe concurrent apps (java.util.concurrent.*).

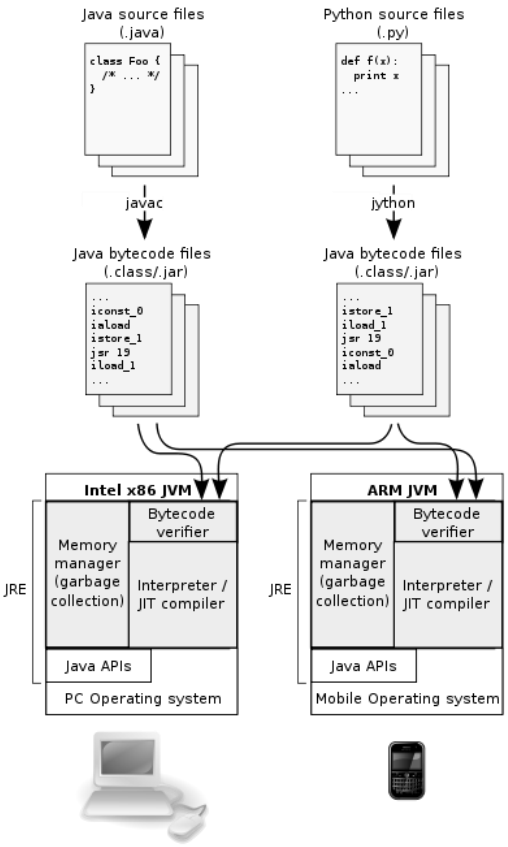
    Mutex - a non-entrant mutual exclusion lock which is another term for a lock. This utility class is used to control locking mechanism.

J2SE 1.4 - Merlin 2002
J2SE 1.3 - Kestrel 2000
J2SE 1.2 - Playground 1998
JDK 1.1 - 1997
JDK 1.0 - Code Name: Oak 1996 Initial Release – Founder: James Gosling

Chapter 3 – JVM and Class Loader

Java Virtual Machine (JVM)

A Java virtual machine is software that is implemented on non-virtual hardware and on standard operating systems. A JVM provides an environment in which Java byte code can be executed, enabling such features as automated exception handling, which provides "root-cause" debugging information for every software error (exception), independent of the source code. A JVM is distributed along with a set of standard class libraries that implement the Java application programming interface (API). Appropriate APIs bundled together with JVM form the Java Runtime Environment (JRE).



Java Class Loader

Each Java class must be loaded by a class loader. Furthermore, Java programs may make use of external libraries (that is, libraries written and provided by someone other than the author of the program) or they may be composed, at least in part, of a number of libraries.

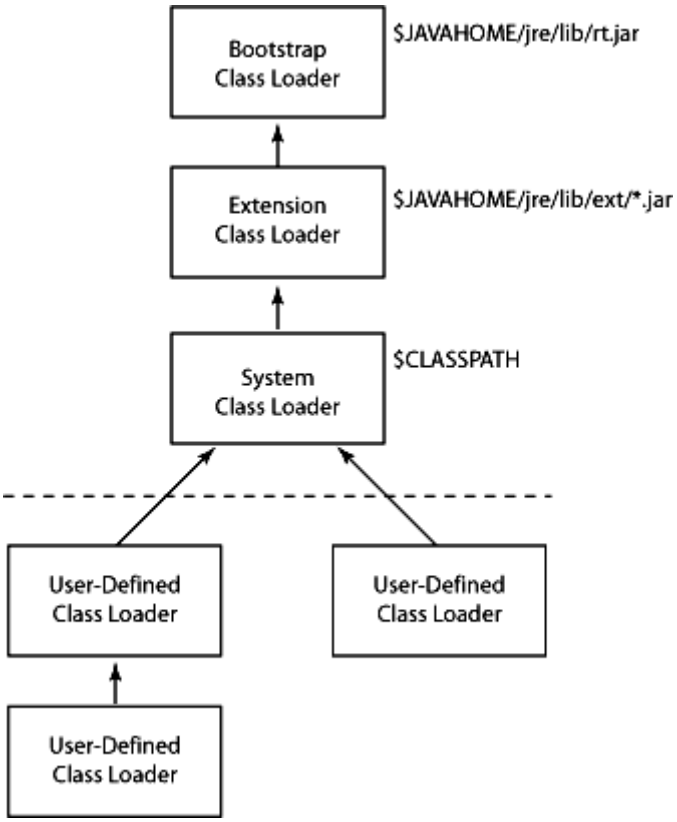
When the JVM is started, three class loaders are used:

1. Bootstrap class loader
2. Extensions class loader
3. System class loader

The bootstrap class loader loads the core Java libraries (<JAVA_HOME>/lib directory). This class loader, which is part of the core JVM, is written in native code.

The extensions class loader loads the code in the extensions directories (<JAVA_HOME>/lib/ext or any other directory specified by the java.ext.dirs system property). It is implemented by the sun.misc.Launcher$ExtClassLoader class.

The system class loader loads code found on java.class.path, which maps to the system CLASSPATH variable. This is implemented by the sun.misc.Launcher$AppClassLoader class.



Class Loader Delegation

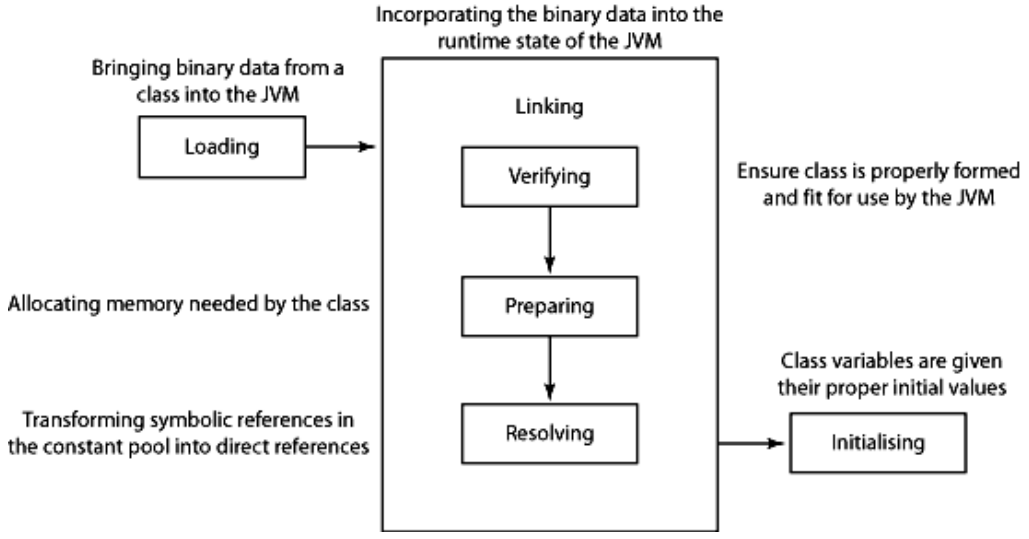
The loading of Java classes is performed by class loaders (CL), they are responsible for loading classes into the JVM. Simple applications can use the Java platform's built-in class loading facility to load their classes, more complex applications tend to define their own custom class loaders.

The class loaders in Java are organized in a tree. By request a class loader determines if the class has already been loaded in the past, looking up in its own cache. If the class is present in the cache the CL returns the class, if not, it delegates the request to the parent. If the parent is not set (is Null) or cannot load the class and throws a ClassNotFoundException the classloader tries to load the class itself and searches its own path for the class file. If the class can be loaded it is returned, otherwise a ClassNotFoundException is thrown. The cache lookup goes on recursively from child to parent, until the tree root is reached or a class is found in cache. If the root is reached the class loaders try to load the class and unfold the recursion from parent to child. Summarizing that we have following order:

- Cache
- Parent
- Self

This mechanism ensures that classes tending to be loaded by class loaders nearest to the root. Remember, that parent class loader is always has the opportunity to load a class first. It is important to ensure that core Java classes are loaded by the bootstrap loader, which guarantees that the correct versions of classes such as java.lang.Object are loaded. Furthermore it ensures that one class loader sees only classes loaded by itself or its parent (or further ancestors) and it cannot see classes loaded by its children or siblings.

Phases of class loading

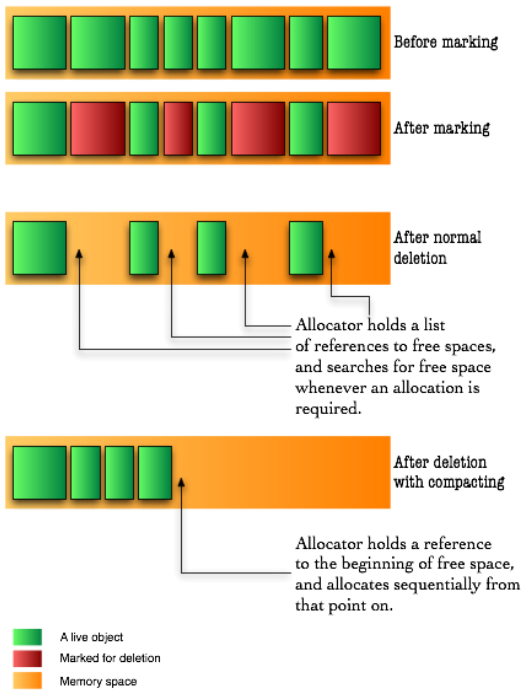


Chapter 4 – Garbage Collector

The basis of garbage collection

The garbage collector first performs a task called marking. The garbage collector traverses the application graph, starting with the root objects; those are objects that are represented by all active stack frames and all the static variables loaded into the system. Each object the garbage collector meets is marked as being used, and will not be deleted in the sweeping stage.

The sweeping stage is where the deletion of objects takes place. There are many ways to delete an object: The traditional C way was to mark the space as free, and let the allocator methods use complex data structures to search the memory for the required free space. This was later improved by providing a defragmenting system which compacted memory by moving objects closer to each other, removing any fragments of free space and therefore allowing allocation to be much faster:



For the last trick to be possible a new idea was introduced in garbage collected languages: even though objects are represented by references, much like in C, they don't really reference their real memory

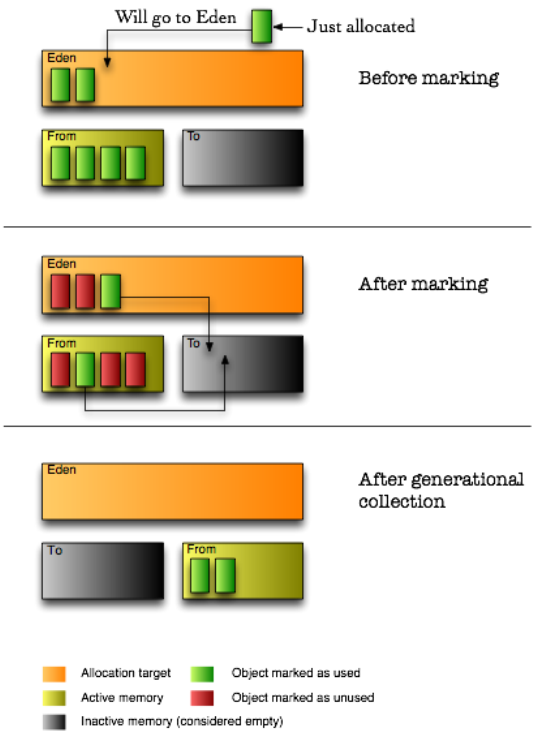
location. Instead, they refer to a location in a dictionary which keeps track of where the object is at any moment.

Fortunately for us – but unfortunately for these garbage collection algorithms – our servers and personal computers got faster (and multiple) processors and bigger memory capacities. Compacting memory areas this large often was very taxing on the application, especially considering that when doing that, the whole application had to freeze due to the changes in the virtual memory map. Fortunately for us though, some smart people improved those algorithms in three ways: concurrency, parallelization and generational collection.

Generational Garbage Collection

In any application, objects could be categorized according to their life-line. Some objects are short-lived, such as most local variables, and some are long-lived such as the backbone of the application. The thought about generational garbage collection was made possible with the understanding that in an application's lifetime, most instantiated objects are short-lived, and that there are few connections between long-lived objects to short-lived objects.

In order to take advantage of this information, the memory space is divided to two sections: young generation and old generation. In Java, the long-lived objects are further divided again to permanent objects and old generation objects. Permanent objects are usually objects the Java VM itself created for caching like code, reflection information etc. Old generation objects are objects that survived a few collections in the young generation area.



Since we know that objects in the young generation memory space become garbage early, we collect that area frequently while leaving the old generation's memory space to be collected in larger intervals. The young generation memory space is much smaller, thus having shorter collection times.

An additional advantage to the knowledge that objects die quickly in this area, we can also skip the compacting step and do something else called copying. This means that instead of seeking free areas (by seeking the areas marked as unused after the marking step), we copy the live objects from one young generation area to another young generation area. The originating area is called the 'From' area, and the target area is called the 'To' area, and after the copying is completed the roles switch: the 'From' becomes the 'To', and the 'To' becomes the 'From'.

In addition, the Java VM splits the young generation to three areas, by adding an area called Eden which is where all objects are allocated into. To my understanding this is done to make allocation faster by always having the allocator reference to the beginning of Eden after a collection.

By using the copying method, garbage collection achieves defragmentation without seeking for dead memory blocks. However, this method proves itself to be more efficient in areas where most objects are garbage, so it is not a good approach to take on the old generation memory area. Indeed, that area is still collected using the compacting algorithm – but now, thanks to the separation of young and old generations, it is done in much larger intervals.

Chapter 5 – Java Collection Framework

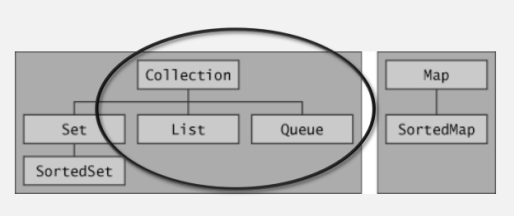
Java Collection versus Java Collections

Java Collection

It's a root interface in the collections hirerachy.

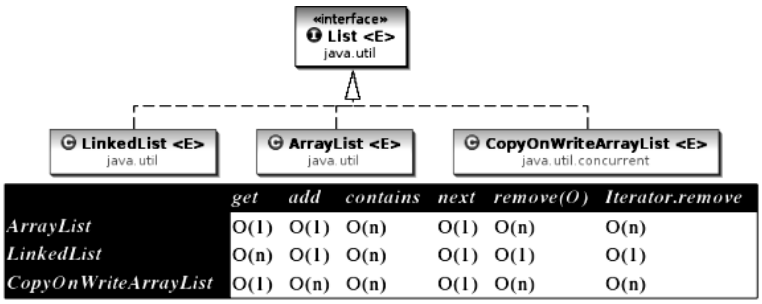
Java Collections

It implements the polymorphic algorithms that operate on the collections. It means the same method can be used on many different implementations of the appropriate Collection interface. (Collections.synchronizeMap(collection), Collections.sort(collection), Collections.unmodifiableList(collection), Collections.unmodifiableMap(collection), etc.)



List Interface

- The List interface provides support for **ordered collections** of objects.
- Lists may contain **duplicate** elements.



	get	*add*	*contains*	*next*	*remove(O)*	*Iterator.remove*
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)
CopyOnWriteArrayList	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

Main Implementations of List Interface

1. **ArrayList:** Resizable-array implementation of the List interface. It's the best all-around implementation of the List interface.

2. **Vector:** Synchronized resizable-array implementation of the List interface with additional "legacy methods."

3. **LinkedList:** Doubly-linked list implementation of the List interface. It may provide better performance than the ArrayList implementation if elements are frequently inserted or deleted within the list. It's useful for queues and double-ended queues (deques).

Advantages of ArrayList over Array

Some of the advantages ArrayList has over Array are:

1. It can grow dynamically
2. It provides more powerful insertion and search mechanisms than arrays

Differences between ArrayList and Array

ArrayList	Array
It stores only objects	It can store primitive values and objects
It can grow dynamically and re-sizable	It has fixed size
It can be only single dimensional	It can have multi dimensional
It resides in the Collection framework (java.util)	It resides in Java core package (java.lang)

Differences between ArrayList and Vector

ArrayList	Vector
ArrayList is NOT synchronized by default	Vector List is synchronized by default
ArrayList can use only Iterator to access the elements	Vector list can use Iterator and Enumeration Interface to access the elements
The ArrayList increases its array size by 50 percent if it runs out of room	A Vector defaults to doubling the size of its array if it runs out of room
ArrayList has no default size	While vector has a default size of 10

Conversion from ArrayList to Array and Vice-versa

ArrayList to Array	Array to ArrayList
List<Element> arrayList = new ArrayList<Element>(); Object[] array = arrayList.toArray(new Object[arrayList.size]);	List<Element> arrayList = new ArrayList<Element>(Arrays.asList(array))

ArrayList versus LinkedList

1. ArrayList internally uses and array to store the elements, when that array gets filled by inserting elements a new array of roughly 1.5 times the size of the original array is created and all the data of old array is copied to new array. During deletion, all elements present in the array after the deleted elements have to be moved one step back to fill the space created by deletion.

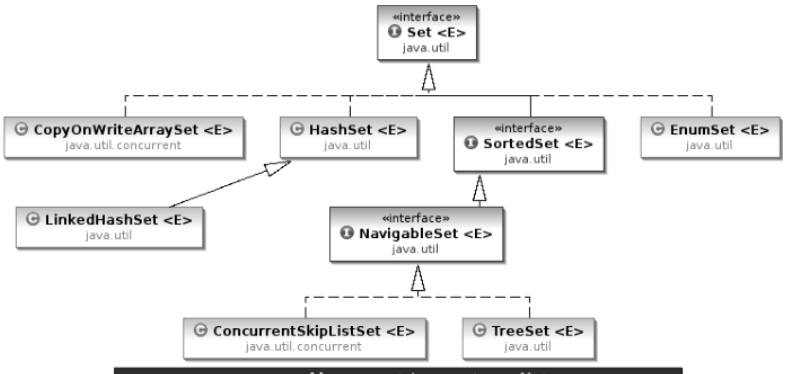
2. In linked list data is stored in nodes that have reference to the previous node and the next node so adding element is simple as creating the node an updating the next pointer on the last node and the previous pointer on the new node. Deletion in linked list is fast because it involves only updating the next pointer in the node before the deleted node and updating the previous pointer in the node after the deleted node.

3. If you need to support **random access**, without inserting or removing elements from any place other than the end, then ArrayList offers the optimal collection.

4. If, however, you need to **frequently add and remove elements** from the middle of the list and only access the list elements **sequentially**, then LinkedList offers the better implementation.

Set Interface

- The Set interface provides methods for accessing the elements of a finite mathematical set. It provides an **unordered collection** of objects.
- Sets do not **allow duplicate elements**
- Contains no methods other than those inherited from Collection
- It adds the restriction that duplicate elements are prohibited
- Two Set objects are equal if they contain the same elements



	add	*contains*	*next*	*Note*
HashSet	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashSet	O(1)	O(1)	O(1)	
CopyOnWriteArraySet	O(n)	O(n)	O(1)	
EnumSet	O(1)	O(1)	O(1)	
TreeSet	O(log n)	O(log n)	O(log n)	
ConcurrentSkipListSet	O(log n)	O(log n)	O(1)	

Main Implementations of Set Interface

1. HashSet
2. TreeSet
3. LinkedHashSet
4. EnumSet

HashSet

- A HashSet is an unsorted, unordered Set.
- It uses the hashcode of the object being inserted (so the more efficient your hashcode() implementation the better access performance you'll get).
- Use this class when you want a collection with no duplicates and you don't care about order when you iterate through it.

TreeSet

TreeSet is a SortedSet implementation that keeps the elements in sorted order. The elements are sorted according to the natural order of elements or by the comparator provided at creation time.

EnumSet

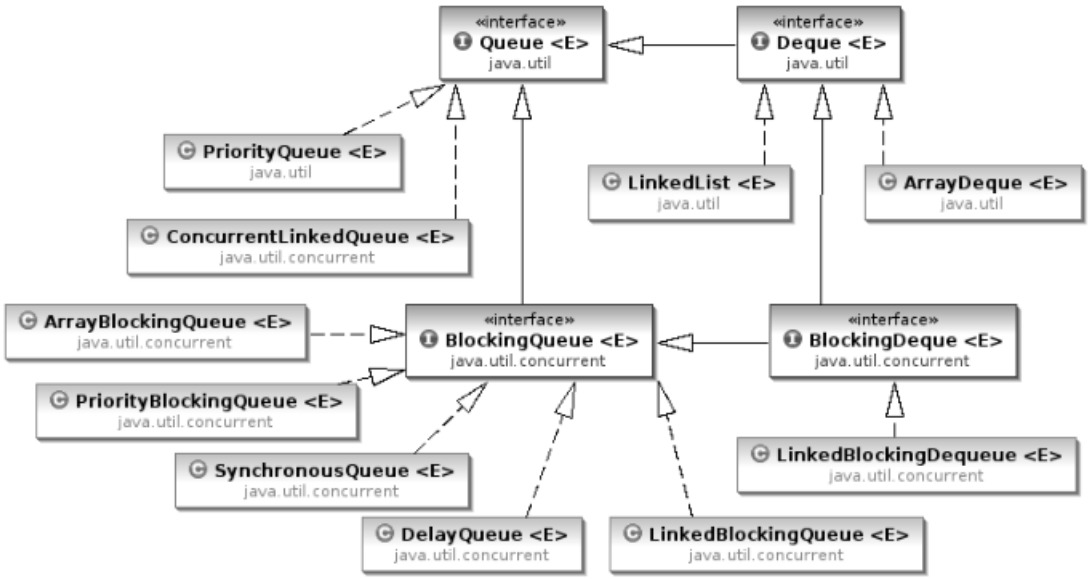
An EnumSet is a specialized set for use with enum types, all of the elements in the EnumSet type that is specified, explicitly or implicitly, when the set is created.

Differences between HashSet and TreeSet

HashSet	TreeSet
HashSet is under Set interface. So, it does not guarantee for either sorted order or sequence order.	TreeSet is under SortedSet interface. So, it provides elements in a sorted order (natural - asceding order).
We can add any type of elements to HashSet.	We can add only similar types of elements to TreeSet.

Queue Interface

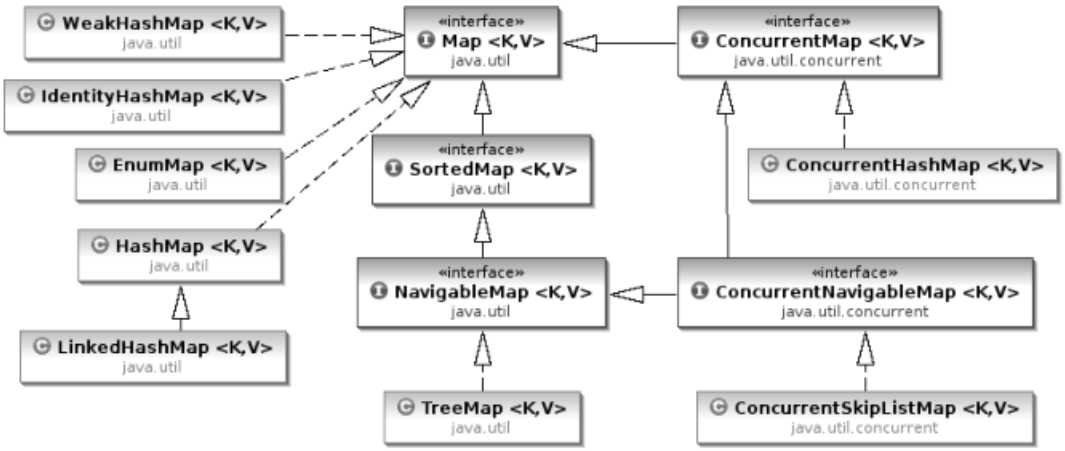
Stack is a data structure that is based on Last in First out (**LIFO**) rule, while Queues are based on First in First out (**FIFO**) rule.



	offer	*peek*	*poll*	*size*
PriorityQueue	O(log n)	O(1)	O(log n)	O(1)
ConcurrentLinkedQueue	O(1)	O(1)	O(1)	O(n)
ArrayBlockingQueue	O(1)	O(1)	O(1)	O(1)
LinkedBlockingQueue	O(1)	O(1)	O(1)	O(1)
PriorityBlockingQueue	O(log n)	O(1)	O(log n)	O(1)
DelayQueue	O(log n)	O(1)	O(log n)	O(1)
LinkedList	O(1)	O(1)	O(1)	O(1)
ArrayDeque	O(1)	O(1)	O(1)	O(1)
LinkedBlockingDequeue	O(1)	O(1)	O(1)	O(1)

Map Interface

- A map is an object that stores associations between **keys and values** (key/value pairs).
- Given a key, you can find its value. Both keys and values are objects.
- The keys must be unique, but the values may be duplicated.
- Some maps can accept a null key and null values, others cannot.
- The key class of Map should override **equals** and **hashcode** methods of Object superclass.



	get	*containsKey*	*next*	*Note*
HashMap	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashMap	O(1)	O(1)	O(1)	
IdentityHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
EnumMap	O(1)	O(1)	O(1)	
TreeMap	O(log n)	O(log n)	O(log n)	
ConcurrentHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)	

Main Implementations of Map

1. HashMap
2. HashTable
3. TreeMap
4. EnumMap

TreeMap

TreeMap actually implements the SortedMap interface which extends the Map interface. In a TreeMap the data will be sorted in ascending order of keys according to the natural order for the key's class, or by the comparator provided at creation time. TreeMap is based on the Red-Black tree data structure.

HashMap versus TreeMap

- For inserting, deleting, and locating elements in a Map, the HashMap offers the best alternative.
- If, however, you need to traverse the keys in a sorted order, then TreeMap is your better alternative.
- Depending upon the size of your collection, it may be faster to add elements to a HashMap, and then convert the map to a TreeMap for sorted key traversal.

Differences between HashMap and HashTable

HashMap	HashTable
HashMap lets you have null values as well as one null key. Only one NULL is allowed as a key in HashMap. HashMap does not allow multiple keys to be NULL. Nevertheless, it can have multiple NULL values.	HashTable does not allow null values as key and value. The **properties** class is a subclass of HashTable that can be read from or written to a stream.
The iterator in the HashMap is fail-safe.	The enumerator for the Hashtable is not fail-safe.
HashMap is NOT synchronized.	Hashtable is synchronized.

Collection Views provided by Map

1. **Key Set -** allow a map's contents to be viewed as a set of keys. KeySet is a set returned by the **keySet ()** method of the Map interface.
2. **Values Collection -** allow a map's contents to be viewed as a set of values. Values Collection View is a collection returned by the **values ()** method of the Map interface.
3. **Entry Set -** allow a map's contents to be viewed as a set of key-value mappings. Entry Set view is a set that is returned by the **entrySet ()** method in the Map interface.

Traverse Collection

There are two ways to traverse collections:

1. Using Iterator or Enumeration
2. Using Enhanced For Loop

Iterator interface provides functions for:

Creating the data structure

- add(e)
- addAll(c)

Querying the data structure

- size()
- isEmpty()
- contains(e)
- containsAll(c)
- toArray()
- equals(e)

Modifying the data structure

- remove(e)
- removeAll(c)
- retainAll(c)
- clear()

Iterator

- The Iterator interface is used to step through the elements of a Collection.
- Iterators let you process each element of a Collection.
- Iterators are a generic way to go through all the elements of a Collection no matter how it is organized.
- Iterator is an Interface implemented a different way for every Collection.
- To use an iterator to traverse through the contents of a collection, follow these steps:
- Obtain an iterator to the start of the collection by calling the collection's **iterator**() method.
- Set up a loop that makes a call to **hasNext**(). Have the loop iterate as long as hasNext() returns true.
- Within the loop, obtain each element by calling **next**().
- Iterator also has a method **remove**() when remove is called, the current element in the iteration is deleted.

Fail Fast/Safe Iterator

Because, if 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.

Differences between Enumeration and Iterator

Enumeration	Iterator
Enumeration doesn't have a remove() method	Iterator has a remove() method
Enumeration acts as Read-only interface, because it has the methods only to traverse and fetch the objects. So, Enumeration is used whenever we want to make Collection objects as Read-Only.	Iterator can be used to edit the collection by adding or removing elements. It can be abstract, final, native, static, or synchronized.

ListIterator

ListIterator is just like Iterator, except it allows us to access the collection in either the **forward or backward direction** and lets us modify an element.

Sorting Mechanisms

Collections can use either Comparable or Comparator interfaces to define the sorting order.

Comparable Interface

The Comparable interface is used to sort collections and arrays of objects using the **Collections.sort()** and **java.utils.Arrays.sort()** methods respectively. The objects of the class implementing the Comparable interface can be ordered.

The Comparable interface in the generic form is written as follows:

```
interface Comparable<T>
```

where T is the name of the type parameter.

All classes implementing the Comparable interface must implement the compareTo() method that has the return type as an integer. The signature of the compareTo() method is as follows:

```
int i = object1.compareTo(object2)
```

- If object1 < object2: The value of i returned will be negative.
- If object1 > object2: The value of i returned will be positive.
- If object1 = object2: The value of i returned will be zero.

Differences between Comparable and Comparator Interfaces

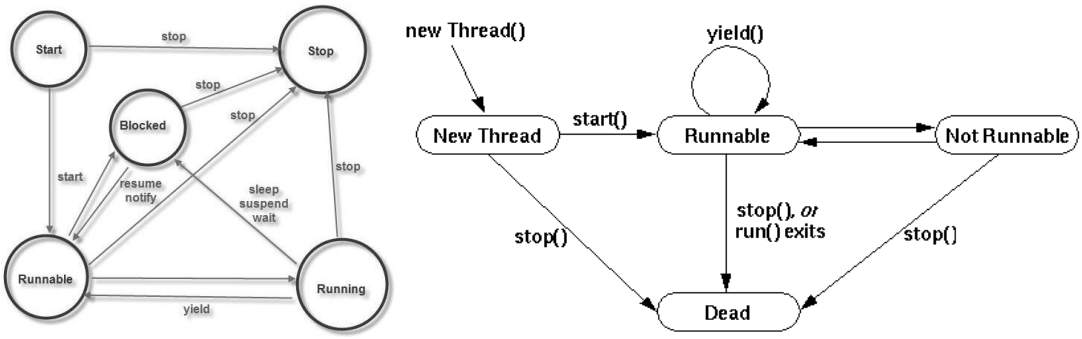
Comparable	Comparator
It uses the compareTo() method. int objectOne.compareTo(objectTwo).	It uses the compare() method. int compare(ObjOne, ObjTwo)
It is necessary to modify the class whose instance is going to be sorted.	A separate class can be created in order to sort the instances.
Only one sort sequence can be created.	Many sort sequences can be created.
It is frequently used by the API classes.	It used by third-party classes to sort instances.

Summary

Map	An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value
SortedMap	A Map that further provides a total ordering on its keys
NavigableMap	A SortedMap extended with navigation methods returning the closest matches for given search targets
ConcurrentMap	A Map providing additional atomic putIfAbsent, remove, and replace methods.
ConcurrentNavigableMap	A ConcurrentMap supporting NavigableMap operations, and recursively so for its navigable sub-maps.
List	An ordered collection
Set	A collection that contains no duplicate elements
SortedSet	A Set that further provides a total ordering on its elements
NavigableSet	A SortedSet extended with navigation methods reporting closest matches for given search targets
Queue	A collection designed for holding elements prior to processing
Dequeue	A linear collection that supports element insertion and removal at both ends
BlockingQueue	A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element
BlockingDequeue	A Deque that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element

Chapter 6 – Java Threading Model

Java Threads



Java Thread Lifecycle

An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:

Provide a Runnable object
The Runnable interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the HelloRunnable example:

```
public class HelloRunnable implements Runnable {
   public void run() {
      System.out.println("Hello from a thread!");
   }
   public static void main(String args[]) {
      (new Thread(new HelloRunnable())).start();
   }
}
```

Subclass Thread

The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run, as in the HelloThread example:

```
public class HelloThread extends Thread {
   public void run() {
      System.out.println("Hello from a thread!");
   }
   public static void main(String args[]) {
      (new HelloThread()).start();
   }
}
```

Notice that both examples invoke Thread.start in order to start the new thread. The first idiom, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread.

Concurrency and Multi-Threading in Java

It covers the concepts of parallel programming, immutability, threads, the executor framework (thread pools), futures, callables and the fork-join framework.

Concurrency

Concurrency is the ability to run several programs or several parts of a program in parallel. If a time consuming task can be performed asynchronously or in parallel, this improve the throughput and the interactivity of the program.

A modern computer has several CPU's or several cores within one CPU. The ability to leverage these multi-cores can be the key for a successful high-volume application.

Process versus Thread

A process runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process, e.g. memory and CPU time, are allocated to it via the operating system.

A thread is a so called lightweight process. It has its own call stack, but can access shared data of other threads in the same process. Every thread has its own memory cache. If a thread reads shared data it stores this data in its own memory cache. A thread can re-read the shared data.

A Java application runs by default in one process. Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.

Concurrency Issues

Threads have their own call stack, but can also access shared data. Therefore you have two basic problems, visibility and access problems.

A visibility problem occurs if thread A reads shared data which is later changed by thread B and thread A is unaware of this change.

An access problem can occur if several thread access and change the same shared data at the same time.

Visibility and access problem can lead to

Liveness failure: The program does not react anymore due to problems in the concurrent access of data, e.g. deadlocks.

Safety failure: The program creates incorrect data.