Java has two components - Runtime Environment and API
Java code is compiled by compiler into platform-independent bytecode (class file) which is then interpreted to machine language

Classloader: adds security by separating the package for the classes of the local file system from those that are imported from network sources.

Bytecode Verifier: checks the code fragments for illegal code that can violate access right to objects.

Security Manager: determines what resources a class can access such as reading and writing to the local disk.

Java does not use explicit pointers but uses implicitly for internal working

Automatic garbage collection

Architecture-neutral (same for 32 bit and 64 bit)

Multithreaded using common memory area

Java is always call by value

main() is static because it is executed by JVM and does not require any object for invocation

Valid main signatures:

1. public static void main(String[] args)
2. public static void main(String []args)
3. public static void main(String args[])
4. public static void main(String... args)
5. static public void main(String[] args)
6. public static final void main(String[] args)
7. final public static void main(String[] args)
8. final strictfp public static void main(String[] args)

Invalid signatures:

1. public void main(String[] args)

2. static void main(String[] args)
3. public void static main(String[] args)
4. abstract public static void main(String[] args)

At runtime,

Class file -> Class loader -> Bytecode verifier -> Interpreter -> Runtime -> Hardware

If class is not public, a java source file can be saved with a name other than the class name

Multiple classes can be present in a single java file

JVM is an abstract machine or a specification that provides runtime environment. JVM is not real. It has no form or physical existence. It is an idea. JVM is platform dependent. It needs to be specified for each platform. JVM loads, verifies, executes code and provides runtime environment. JVM provides definitions for memory area, class file format, register set, garbage-collected heap, fatal error reporting, etc.

JRE is the runtime environment and it is an implementation of JVM. It physically exists. JRE is a set of libraries and other files that JVM uses at runtime. JVM implementations are released by companies other that SUN. (3rd party). JRE is SUN's implementation of JVM. It is platform dependent.

JDK is java development kit. It is physical as well and contains JRE along with development tools. It is platform dependent.

When you run java command to run java class, an implementation of JVM (JRE instance gets created)

Heap memory area is used to store objects

Stack memory area is used to store local variables and functions.

Local variable belongs to a method.

Instance variable belongs to an object

Static variable belongs to a class

---

---

Widening does not need explicit casting

Narrowing needs explicit casting

Adding short or byte results in int which cannot be assigned back to short/byte without explicit casting

Failure in casting will result in compile time error

<< left shift multiplies by 2 (10<<2 = 10*2^2)

>> right shift divides by 2 (10>>2 = 10/2^2)

Java follows PEMDAS

For positive numbers, >> and >>> work same, while for negative number, it changes the MSB i.e. parity bit to 0

a+=b for short or byte does not need typecasting whereas a = a + b needs it

If break is not used with case statements, every case after the first match will be executed including default. This is called as fall-through.

We can have label for a for loop as

```
1. labelname:
2. for(initialization;condition;incr/decr){
3. //code to be executed
4. }
```

This label can be used with a break statement to break out of that particular loop. By default, breaks takes you out of the innermost loop.

for(;;) will start an infinite loop and is perfectly valid. while(true) does the same thing.

Object has state and behavior

Conventions:

| | |
|---|---|
| class name | should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc. |
| interface name | should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc. |
| method name | should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc. |
| variable name | should start with lowercase letter e.g. firstName, orderNumber etc. |
| package name | should be in lowercase letter e.g. java, lang, sql, util etc. |
| constants name | should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc. |

Objects are stored in heap memory while the reference variables are stored in stack memory.

Objects can be anonymous like new Calculation().fact(5);  These are good for one time use.

Objects can be created in a single statement like

Rectangle r1=new Rectangle(), r2=new Rectangle();//creating two objects

Constructor cannot have return type but it returns current class instance i.e. reference to the created object

Private constructors are valid

If no constructor is defined, compiler automatically inserts a default constructor with no arguments.

Constructors can be overloaded.

Objects can be initialized by manual assignment, using objects, or clone() method.

---

Static variables are assigned memory only once so they make the program memory efficient.

Static variables can be used for storing a property that is common to all objects like company name for all employees.

A static method belongs to the class and can be invoked without any object.

A static method can access only static variables and call other static methods, however instance methods can access static variables as well as instance variables.

This and super keyword cannot be used in static context since they refer to objects.

A static block can be used to initialize static variables just like a constructor is used to initialize instance variables.

Static block is executed before main()

This keyword refers to the current object

this() can be used to invoke constructor, method.

this() can be used for constructor chaining and must be the first statement in the calling constructor

Inheritance means IS-A

Aggregation means HAS-A

A class cannot inherit from multiple classes. To achieve this, interfaces will need to be used.

A class can implement multiple interfaces and another class can then extend it.

Method overloading is a way of achieving runtime polymorphism.

Methods cannot be overloaded on the basis of return type. It can only be done with the help of data type or number of arguments.

main() can be overloaded as well but the first one to get called will always be the one with string arguments and others can be called from that main or other methods.

byte/short are promoted to int automatically.

The arguments in a method will be promoted to the larger compatible type if the data types of the arguments are different.

```
1. private int sum (int a, long b) {}
2.  obj.sum(20,20);//now first int literal will be promoted to long
```

If both are int and int args are passed, int method will be invoked.

If both are long and long args are passed, long method will be invoked.

If one is long and other is int and int is passed, passed args will be promoted to long and long method will be called.

Method overriding is where the subclass provides a specific implementation of a method present in the parent.

Method overriding is also used to achieve runtime polymorphism.

To override a method, it must be identical to the one from parent class.

Static methods cannot be overridden.

Covariant return type is possible in method overriding.

Super keyword is used to refer to the immediate parent class object

Whenever an object of subclass is created, an object of superclass is automatically created.

super() is used to invoke parent class constructor.

super() is added to the constructor automatically as the first line

Instance initializer block can be used to initialize instance variables.

The compiler calls the instance initializer block after calling super() i.e. it places the call in the second line of the constructor.

A final variable can be initialized at the time of declarations or in the constructor block and nowhere else.

A blank final static variable can be initialized at the time of declaration or the static initializer block and nowhere else.

Once assigned, a final variable value cannot be changed and will be a constant.

A final method cannot be overridden, however, it can be inherited.

A final class cannot be extended.

A final argument value cannot be changed inside the method it is passed in to.

A constructor cannot be declared final.

Static method overloading is an example of compile time polymorphism.

Runtime polymorphism is also known as dynamic method dispatch.

```
1. class A{}
2. class B extends A{}
1. A a=new B();//upcasting
```

In case of overridden methods, the method that will be called depends on the type of object created and not the type of reference variable. For example, in above code, a.run() will call the run method in B since the object is of type B.

In case where same data member i.e. variable is present in both parent and child class, the one to be called will depend on the type of reference variable i.e here, the variable of A will be called.

When the type of object is determined at compile time, it is called as static binding or early binding.

If there is any private, static or final method in a class, it is static binding.

Example:

```
1.  class Dog{
2.   private void eat(){System.out.println("dog is eating...");}
3.
4.   public static void main(String args[]){
5.    Dog d1=new Dog();
6.    d1.eat();
7.   }
8.  }
```

When the type of object is determined at the runtime, it is called as dynamic binding or late binding.

Example:

```
1.  class Animal{
2.   void eat(){System.out.println("animal is eating...");}
3.  }
4.
5.  class Dog extends Animal{
6.   void eat(){System.out.println("dog is eating...");}
7.
8.   public static void main(String args[]){
9.    Animal a=new Dog();
10.   a.eat();
```

```
11. }
12. }
```

In above example, compiler does not know the type of object. It just knows that animal is eating. The fact that dog is eating can only be determined at runtime since the type of reference variable is of type animal.

Instanceof operator is used to test whether an object is an instance of the specified class, subclass or interface

E.g. dog is an instance of dog as well as animal

If the object is null, the instanceof operator returns null. Thus, it can be used to do a null check.

Downcasting is possible using instanceof operator.

E.g. Dog d1 = new Animal(); is not possible

Dog d1 = (Dog) new Animal(); is possible but will throw classcastexception at runtime.

E.g.

```
1.  class Animal { }
2.
3.  class Dog3 extends Animal {
4.    static void method(Animal a) {
5.      if(a instanceof Dog3){
6.        Dog3 d=(Dog3)a;//downcasting
7.        System.out.println("ok downcasting performed");
8.      }
9.    }
10.
11.  public static void main (String [] args) {
12.    Animal a=new Dog3();
13.    Dog3.method(a);
```

```
14.   }
15.
16. }
```

Downcasting without instanceof:

```
1.  class Animal { }
2.  class Dog4 extends Animal {
3.    static void method(Animal a) {
4.         Dog4 d=(Dog4)a;//downcasting
5.         System.out.println("ok downcasting performed");
6.    }
7.     public static void main (String [] args) {
8.      Animal a=new Dog4();
9.      Dog4.method(a);
10.   }
11. }
```

```
1.  Animal a=new Animal();
2.  Dog.method(a);
3.  //Now ClassCastException but not in case of instanceof operator
```

Instanceof operator can be used when two or more classes implement same interface and it is required to be decided at runtime as to which method is to be invoked.

```
1.  interface Printable{}
2.  class A implements Printable{
3.  public void a(){System.out.println("a method");}
4.  }
5.  class B implements Printable{
6.  public void b(){System.out.println("b method");}
7.  }
8.
```

```
class Call{
void invoke(Printable p){//upcasting
if(p instanceof A){
A a=(A)p;//Downcasting
a.a();
}
if(p instanceof B){
B b=(B)p;//Downcasting
b.b();
}

}
}//end of Call class

class Test4{
public static void main(String args[]){
Printable p=new B();
Call c=new Call();
c.invoke(p);
}
}
```

An abstract class can have abstract as well as non abstract methods

Abstraction helps you focus on what the object does instead of how it does it

Abstraction can be achieved using abstract class [0% - 100%] or interfaces [100%]

Abstract class is not real and hence cannot be instantiated. It needs to be extended and it's methods need to be implemented.

An abstract method does not have a body and is terminated with a semicolon.

A factory method is a method that returns an instance of the class.

An abstract class can have data member, abstract method, normal method with a body, constructor and even main() method.

If the class contains an abstract method, it needs to be declared abstract.

An abstract class can be used to provide some implementation of the interface so that the end user is not forced to override all the methods of the interface.

```
1. interface A{
2. void a();
3. void b();
4. void c();
5. void d();
6. }
7.
8. abstract class B implements A{
9. public void c(){System.out.println("I am C");}
10. }
11.
12. class M extends B{
13. public void a(){System.out.println("I am a");}
14. public void b(){System.out.println("I am b");}
15. public void d(){System.out.println("I am d");}
16. }
17.
18. class Test5{
19. public static void main(String args[]){
20. A a=new M();
21. a.a();
22. a.b();
23. a.c();
24. a.d();
25. }}
```

An interface is a blueprint of the class.
Interface achieves full abstraction.
Interface is an IS-A relationship.
Interface cannot be instantiated.
Interface helps to achieve multiple inheritance.
Interface is used to achieve loose coupling.

The java compiler adds public and abstract keywords to the interface methods and public, static, and final keywords to the data members. Thus, all the methods in an interface are public abstract and all the data members are constants.

An interface can extend another interface.
Class can implement interface but cannot extend it.
An interface can extend multiple interfaces at a time.

A marker or tagged interface is an interface that has no member i.e an empty interface. They are used to provide some essential information to the JVM so that it can perform some useful operation.

An interface can have another interface inside it.

| **Abstract class** | **Interface** |
|---|---|
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can have static methods, main method and constructor**. | Interface **can't have static methods, main method or constructor**. |
| 5) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 6) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |

| 7) **Example:**<br>public abstract class Shape{<br>public abstract void draw();<br>} | **Example:**<br>public interface Drawable{<br>void draw();<br>} |
|---|---|

A java package is used to group similar classed.
A package is used to achieve access protection and avoid naming collision.
A package can be imported into a class within another package to used the classes in it.
Importing a package does not import its subpackages.

Java has 4 access specifiers: public, default(package), protected, private

These can be used with classes, methods, constructors, instance variables, and static variables.

Public allows the visibility to be project wide
Default(no specifier mentioned) allows the visibility to be inside the current package
Protected allows the visibility in the current class as well as child classes
Private allows the visibility to be just in the current class

A private constructor indicates that you cannot instantiate the class object. This means no objects can be created using that constructor.

A class can be private or protected only if it is nested inside another class.

A protected method or variable can be available outside the package but only through inheritance.

| Access Modifier | within class | within package | outside package by subclass only | outside package |
|---|---|---|---|---|
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

If you are overriding a method, the access modifier for that method in the subclass should not be more restrictive than the access modifier of the same method in the parent class.

For example, if protected add() is declared in parent class, you cannot declare private add() in child class while overriding. You have to keep it protected or public.

Encapsulation is the approach of wrapping the data and the behavior together just like everything is wrapped inside a capsule combined.

Encapsulation allows you to make a class either fully read only or fully write only if you want to. This can be done by making all the instance variables private and use getters and setters. This approach also enables you to impose restrictions on the data setting.

Object class is the parent class of all the predefined classes in java. Everything in Java is an object.

You can refer an object with the reference variable of the type object if you do not know the type of the object via upcasting.

To clone an object, the class of the object must have implemented Cloneable interface.

Wrapper classes provide a mechanism to convert primitive to Objects and vice versa

Java is call by value. Call by reference does not exist in Java.

Java strictfp keyword stands for strict floating point. It ensures that you get the same precision for floating point on every platform.
It can be used with class, interface, and method.
It cannot be used with abstract methods, variables, and constructors.

The CharSequence interface is used to represent sequence of characters. It is implemented by String, StringBuffer and StringBuilder classes. It means, we can create string in java by using these 3 classes.

The java String is immutable i.e. it cannot be changed. Whenever we change any string, a new instance is created. For mutable string, you can use StringBuffer and StringBuilder classes.
The String equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

The = = operator compares references not values.

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is

returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

Java StringBuffer class is used to created mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized.

We can also create immutable class by creating final class that have final data members.

Java regex is used to check the formation of strings to see if they are fulfilling the criteria. This is mainly used in email and password constraint verification.

There are two ways to check this.
Pattern.compile("regex").Matcher.match("string").matches();
Pattern.matches("regex","string");

. (dot) - single character

Character Classes - which characters are allowed
[abc] = a, b, or c
[^abc] = any character except a, b, or c
[a-zA-Z] = any character within range a to z or A to Z
[a-d[m-p]] or [a-dm-p] similar as above in union form
[a-z&&[d-f]] = d, e, or f in intersection
[a-z&&[^bc]] = a to z except b, and c in subtraction
[a-z&&[^m-p]] = a to z excluding m to p in subtraction

Quantifiers - how many characters (occurrences) are allowed
A? = A occurs once or not at all
A+ = A occurs once or more than once
A* = A occurs zero or more times
A{n} = A occurs n time only
A{n,} = A occurs n or more times
A{m,n} = A occurs at least m times but less than n times

Metacharacters - short codes
. (dot) = any character
\d = any digit [0 - 9]

\D = any non digit [^0 - 9]
\s = any whitespace character [\t\n\x0B\f\r]
\S = any non-whitespace character [^\s]
\w = any word character [a-zA-Z0-9]
\W = any non word character [^\w]
\b = a word boundary
\B = a non word boundary

Exception handling should be performed to maintain the normal flow of the program in case when something unexpected happens that might disrupt the flow of programming. It is precautionary measure for the things that might go wrong.

Two types of throwable - exception and error

Exceptions can be checked or unchecked

Checked exceptions are the exceptions that are checked at compile-time. These are the classes that extend Throwable class except RuntimeException class, and Error class.
E.g. IOException, SQLException

Unchecked exceptions are the exceptions that are checked at the runtime. These are the classes that extend RuntimeException.
E.g ArithmeticException, NullPointerException

Errors are irrecoverable
E.g OutOfMemoryError, AssertionError

Keywords used for exception handling are - try, catch, finally, throw, throws

```
try
{
	// code that might cause a problem
}

catch
{
	// code that deals with the problem
}

finally
{
	// things to do finally despite of whether problem occurred or not
}
```

Valid mechanisms are try-catch or try-finally or try-catch-finally

Catch block should always be after try block in case catch block is present
Multiple catch blocks maybe present to handle different types of exceptions

At a time, only one type of exception is handled and only one catch block gets executed
Catch blocks should be in an order from most specific to most generic

Try blocks can be nested. There may exist a whole try catch mechanism inside a try block with its own catch block following it

Finally block is used to execute important code despite of whether exception occurred or not.
It is executed almost always.
It can follow either try block or catch block.
If an exception is not handled, JVM executed finally block before terminating the program.
There can be multiple catch blocks for a try block but only one finally block.
The finally block will not get executed of the program exits by calling System.exit(0) or a fatal error is caused which makes the processes abort.

A throw keyword is used to explicitly throw an exception
A checked or unchecked exception can be thrown using throw keyword
It is mainly used to throw custom exception

If an exception is not handled, it gets propagated to the bottom of the calling chain until they are handled.
By default, unchecked exceptions are propagated.
By default, checked exceptions are not propagated and cause compile time error is not handled at the origin.

The throws keyword is used to declare an exception.It is an alternative for using try catch block.
It is used to indicate to the programmer that an exception might occur and he needs to handle it.
This is mainly used when you are designing a library and need to provide clues to the user.

Only checked exceptions should be declared.
Unchecked exceptions are under your control.
Errors cannot be controlled.

Throws keyword allows the checked exceptions to be propagated in the calling chain.
It also guides the caller of the method about the possible exceptions.

If you are calling a method that declares an exceptions, you must either catch it or declare it using throws.

You can rethrow the exception from catch block.

If the superclass method does not declare the exception, the subclass overridden method cannot declare the checked exception but it can declare the unchecked exception.

If the superclass method declares the exception, the subclass overridden method can declare the same or lesser pr no exception but not the parent exception.

All custom exceptions need to extend the Exception class.

Java inner class or a nested class is a class declared inside another class or an interface.
It can access all the members of enclosing class including private data members and methods.
It is used to logically group classes and interfaces.

Nested class can be static and nonstatic (inner).

Nonstatic (inner) class can be member, anonymous, or local.
Member - within a class but outside a method
Anonymous - without a name
Local - inside a method

An interface can be nested too.

The java compiler creates two class files in case of inner class. The class file name of inner class is "Outer$Inner". If you want to instantiate inner class, you must have to create the instance of outer class. In such case, instance of inner class is created inside the instance of outer class.

Anonymous inner class can be created with an abstract parent class or an interface parent.

```
abstract class Person{
	abstract void eat();
}

Class Test{
	public static void main(String[] args)
	{
		Person p = new Person(){
			Void eat{
				System.out.println(“Eating”);
			}
		}

		p.eat();
	}
}
```

A local inner class is declared inside a method and to access the methods of the local inner class, you must instantiate it inside a method.

A local inner class cannot be invoked outside the method.

A local inner class cannot access non-final variables up to Java7. It can in Java 8.

A static nested class cannot access non-static data members and methods.
It can be accessed by outer class name.
It can access static data of outer class including private data.

An interface declared within another interface or class is called as nested interface.
It must be referred by outer interface and cannot be accessed directly.

Nested interface must be public if declared inside an interface but cannot have any access modifiers if declared inside a class.

Nested interfaces are implicitly static.

A class can be defined inside an interface and java makes it implicitly static.

Multithreading in java is the process of executing multiple threads simultaneously.
A thread is a lightweight subprocess, and the smallest unit of processing.
Multiprocessing and multithreading enable multitasking.

Threads share a common memory area. Thus, it saves memory.
This also allows fast context-switching between threads.

It is mostly used in games and animations.

Threads don’t block the user.
Multiple operations can be performed at the same time.
An exception in one thread does not affect other thread.

Multitasking can be thread based (multithreading) or process based (multiprocessing).

Multiprocessing:
Each process has its own memory area.
A process is heavyweight.
Interprocess communication is costly.
Context-switching takes time.

Multithreading:
Each thread shares the same memory area thus memory efficient.
A thread is lightweight.
Inter-thread communication is cheap.
Context-switching takes less time.

A thread cannot exist without a parent process.
It can be thought of as a subprocess.

A process can have multiple threads associated with it.

A thread has five states: new, runnable, running, non-runnable, terminated.

When a new thread is started with start(), it goes into runnable state (running state).
A running thread can be pushed into non-runnable state using sleep, block on IO, wait for lock, suspend, and wait.
A non-runnable thread (suspended/waited) state can be resumed to runnable (running) state using sleep done, IO complete, lock available, resume, notify.
A thread is terminated from runnable(running) state when run() method exits.

A thread is in new state after its instantiation from Thread class but before invocation of start() method.
A thread is in runnable state after invocation of start() method but the thread scheduler has not selected it to be the running thread.
A thread is in running state when the scheduler selects it for running.
A thread is in non-runnable or blocked state when the thread is still alive but is not currently eligible to run.
A thread is in terminated or dead state when it's run method exits.

Thread can be created by extending the Thread class or implementing Runnable interface.

run() contains the code to be executed i.e. the thread body.
start() starts the execution of the thread.
sleep() causes the current thread to sleep.
suspend() causes the thread to suspend.
resume() resumes the suspended thread.
stop() stops the current thread.
interrupt() interrupts the current thread.

Runnable is the interface a class needs to implement if its instance is intended to be executed by a thread.
Each thread has its own call stack but shared memory area.

The thread scheduler in Java is a part of the JVM that decides which threads should run when.
It can use preemptive scheduling or time slicing approaches to make this decision.

The sleep() method causes the thread to sleep for the specified amount of time.

Only one thread can run at a time.

If you try to start a thread when it is already started, it throws an IllegalThreadStateException.

If you call a run() method instead of start, execution will start on the current call stack instead of creating a new call stack. The thread will be treated as normal object and not a thread object.

join() method allows the thread it is called on to get completed before other threads start execution.

Daemon thread is a thread that provides service to all the user threads. It is terminated if no user thread is running.
Gc and finalizer are examples of daemon threads.
If you want to mark a user thread as daemon thread, do not start it. It is handled by JVM.

A thread pool is a collection of threads waiting for the job. A thread from thread pool is pulled out, assigned the job, and added back to the pool once it is done with the job. A thread pool can be created using ExecutorService.

A thread group can be used to group threads together so that operations can be performed on them at the same time.

A shutdown hook is used to perform cleanup tasks in case JVM shuts down unexpectedly.

A shutdown hook can be added using addShutdownHook() method of Runtime.getRuntime()
Shutdown sequence can be stopped using halt(int) method of Runtime class.

Garbage means unreferenced objects. Java garbage collector collects the unreferenced objects and frees the memory occupied by them. Objects can be unreferenced by assigning reference to null, assigning reference to another object, or using anonymous objects.

Finalize() method is invoked right before object is getting garbage collected. This method can be used to perform cleanup tasks.

Garbage collector cleanups only those objects which were created using new operator.

gc() method can be found in System and Runtime classes to request JVM for garbage collection. However, it is not guaranteed that the collection will be performed at the request.

Runtime class is used to interact with runtime environment.
Only one instance of Runtime class is available for one java application.
exec() method executes a program like notepad.
addShutdownHook() is used to add a shutdown hook.

Different operations like shutting down, restarting, getting free and total memory can be performed using Runtime instance.

Synchronization is used to control the access of shared resources by multiple threads.
It is used when we want only one thread to access the shared resource at a time.

There may be thread synchronization or process synchronization.

Thread synchronization can be mutually exclusive or inter-thread communication.

Mutually exclusive - synchronized method, synchronized block, static synchronization

A method, when declared synchronized, can be accessed by only one thread at a given time.

Synchronized block is used to declare a part of the code inside the method as synchronized. It can be seen as a subset of synchronized method.

If you declare a static method as synchronized, the lock will be obtained on the class and not the object.

Deadlock is a situation where threads are waiting for an object lock that are acquired by other threads and no thread can proceed.

Inter-thread communication or cooperation is a mechanism in which a running thread is paused in it's critical section and another thread is allowed to enter the same critical section to be executed.
This is implemented by wait(), notify(), notifyAll()

wait() causes current thread to release the lock and wait until another thread invokes notify or notifyAll() for this object, or a specified amount of time has been elapsed.

notify() wakes up a single thread that is waiting on it's object's monitor. If multiple threads are waiting on the object, one of them is chosen.

notifyAll() wakes up all the threads waiting on this object's monitor.

wait() releases the lock; sleep() does not release the lock
wait() belongs to object class; sleep() belongs to thread class
wait() is non-static; sleep() is static
wait() needs notify(), notifyAll(); sleep() is resumed after the duration

Calling interrupt() on a sleeping or waiting thread makes it come out of that state and throw an InterruptedException

Calling interrupt() on a running thread does nothing but sets the interrupt flag to true.

isinterrupted() returns the status of the interrupt flag.
Static method interrupted() returns the interrupted flag and sets it to false if it is true.

Java streams:
System.out - standard output stream
System.in - standard input stream
System.err - standard error stream

InputStream is used to read data from file, array, peripheral device, or socket
OutputStream is used to write data from file, array, peripheral device, or socket

OutputStream is the parent class of all the output stream classes like:

FileOutputStream
ByteArrayOutputStream
FilterOutputStream - DataOutputStream, BufferedOutputStream, PrintStream
PipedOutputStream
ObjectOutputStream

InputStream is the parent class of all the input stream classes like:
FileInputStream
ByteArrayInputStream
FilterInputStream - DataInputStream, BufferedInputStream, PushbackInputStream
PipedInputStream
ObjectInputStream

FileOutputStream is used to write to a file. It can be used to write a byte or byte array to the file.
FileInputStream is used to read from a file. It can be used to read a byte or byte array from a file.

BufferedOutputStream is used for greater performance.
OutputStream os = new BufferedOutputStream(new FileOutputStream("path"));
os.write(byteArray);

BufferedInputStream is used for greater performance.
InputStream is = new BufferedInputStream(new FileInputStream("path"));
Byte = is.read();

SequenceInputStream can be used to read data from multiple files at once.

ByteArrayOutputStream can be used to write same information to multiple files.
The information is written to byte array which can them be written to multiple streams.
The buffer automatically grows.
write() is used to write data to buffer
writeTo() is used to write data to stream

ByteArrayInputStream can be used to read byte array as input stream
The buffer automatically grows
Byte array can be populated from input stream and data can be read from that byte array

DataOutputStream allows an application to write primitive data types to the output stream in a machine independent way. This data can be read later using DataInputStream.
DataOutputStream dos = new DataOutputStream(new FileOutputStream("path"));

DataInputStream allows an application to read primitive data types from the input stream in a machine independent way.
DataInputStream dis = new DataInputStream(new FileInputStream("path"));

FilterOutputStream and FilterInputStream classes are rarely used as their subclasses BufferedOutputStream, DataOutputStream and BufferedInputStream, DataInputStream are used.

Console class is used for console operations.
System.console.readLine();

FilePermission class is used to read and grant permissions to files and directories.

FileWriter class is used to write character oriented data to a file.
FileReader class is used to read character oriented data from a file.

BufferedWriter is used to provide buffer for the FileWriter class.
BufferedWriter bw = new BufferedWriter(new FileWriter("path");

BufferedReader is used to provide buffer for the FileReader class.
BufferedReader br = new BufferedReader(new FileReader("path");

CharArrayWriter is used to write character array.
CharArrayReader is used to read character array.

Serialization is a mechanism of writing the state of an object to a byte stream.
Reading the state of an object from byte stream to restore the object is called as deserialization.

It is mainly used to travel object's state over network.

To make an object eligible for serialization, it has to implement the marker interface Serializable.

The String class and all the Wrapper classes implement Serializable by default.

ObjectOutputStream writeObject() can be used to write Objects to the byte stream.
Only objects implementing Serializable can be written to byte stream.

ObjectInputStream readObject() can be used to read Objects from the byte stream.
Only serialized objects can be read from the byte stream.

If a class implements Serializable, all its subclasses are also serializable.

If a class has reference of another class, the class must also be serializable in order of the main class to be serializable. I.e. All the objects within an object must be serialiazable.

If there is any static member in a class, the member won't be serialized.

In case of array or collection, all their objects must also be serializable.

If you don’t want to serialize a member of the class, you can mark it as transient.

Connection oriented - with acknowledgement (TCP)
Connectionless - without acknowledgement (UDP)

Socket programming can be used to exchange data between applications running on different JRE. Socket programming can be connectionless or connection oriented.

Connection oriented:
Socket
ServerSocket

Connectionless:
DatagramSocket
DatagramPacket

The client must know the ip address and the port number of the server.

A socket is simply an endpoint for communications between the machines.

Java URL class represents an URL and contains protocol, server name or ip address, port number, file or directory name.

URLConnection class represents a communication link between the URL and the application. This class can be used to read and write data to the specified resource referred by the URL.

openConnection() method returns the instance of the URLConnection class.

HttpUrlConnection class is the http specific URL connection. It works for http protocol only. You can get information on header information, status code, response code, etc.
It is a subclass of URLConnection class.

InetAddress class represents an IP address.
It provides methods to get ip of any hostname or vice-versa.

DatagramSocket and Datagram packet are used for connectionless socket programming and sending or receiving datagram packets. The arrival, arrival time, or the content of the datagram packet is not guaranteed.

Swing is a better alternative as compared to AWT. It is platform-independent, lightweight, and follows MVC architecture.

A frame can be created by making an instance of Frame class or extending it.

JFrame - create a window
JButton - create a button

JLabel - add text in window
JTextField - add editable text field in a window
JTextArea - add multiline text area in a window
JPasswordField -add a password field
JCheckBox - add a checkbox in a window
JRadioButton - add a radio button in a window
JComboBox - add a popup menu of choices
JTable - add a table in a window
JList - add a list of elements in a window
JOptionPane - add dialog boxes on event
JScrollBar - add horizontal and vertical scrollbar to window
JMenuBar, JMenu, JMenuItem - add menu elements to window
JPopupMenu - show a popup menu
JSeparator - add a separator between elements
JProgressBar -add a progress bar to a window
JTree - add a tree like structure to the window
JColorChooser - add a color chooser to window
JTabbedPane - add a tabbed pane to window
JSlider - add a slider to window
JSpinner - add a spinner to window
jDialog - add a pop-up dialog to window
JPanel - add a space inside a frame that can contain other elements
jFileChooser - open a file browsing window
JLayeredPane - add multiple panes inside a window

Create a jar using
jar -cvmf myfile.mf myjar.jar First.class

Java Reflection is a process of modifying runtime behavior of a class at run time.

Ways to get the object of Class class:
- forName()
- getClass()
- .class

forName() - load the class dynamically and get instance of the class
Cannot be used for primitive types

getClass() - return new instance of the class
It can be used with primitive types

.class - can be used to get the class instance by appending .class to the class name
Can be used with primitive types

isInterface() - determine if the class represents an interface type
isArray() - determine if the class represents an array type
isPrimitive() - determine if the class represents a primitive type

Class.newInstance() - used to create an instance of the class

A private method can be called from another class using java.lang.Class and java.lang.reflect.Method classes.

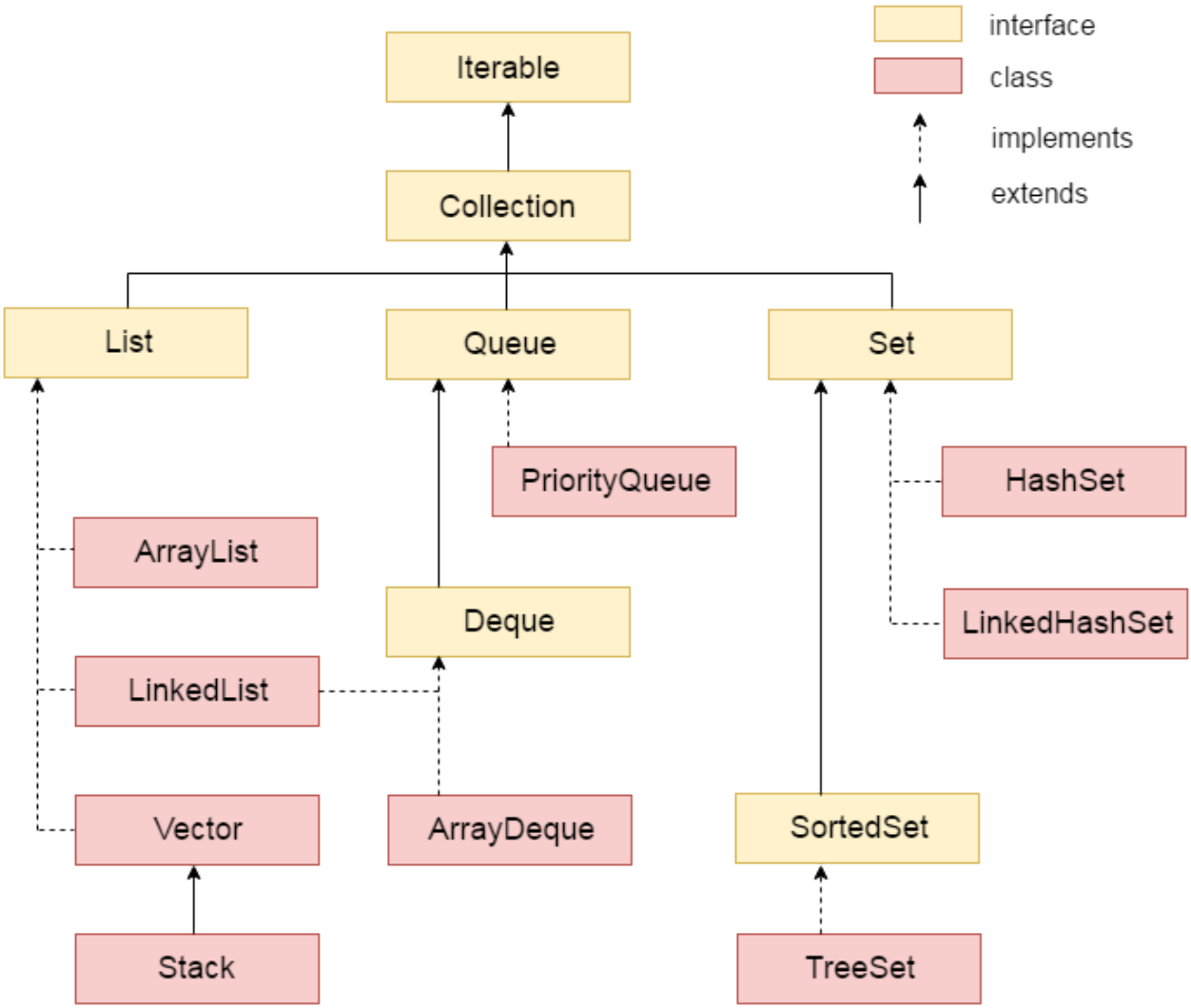
Method.setAccessible(boolean status) sets the accessibility of the method
invoke() is used to invoke the method

getDeclaredMethod() is used to get the method object of the method to be invoked. This object can then be set accessible and invoked

String to int - Integer.parseInt()
Int to String - String.valueOf() | Integer.toString() | append blank string to int
String to long - Long.parseLong()
Long to String - String.valueOf() | Long.toString() | append blank string to long
String to float - Float.parseFloat()
Float to String - String.valueOf() | Float.toString() | append blank string to float
String to double - Double.parseDouble()
Double to String - String.valueOf() | Double.toString() | append blank to double
String to date - DateFormat.parse(String date)

Collections in Java is a framework that provides an architecture to store and manipulate the group of objects.

Basic operations like insertion, deletion, searching, sorting, manipulation can be done using collections.




| | | |
|---|---|---|
| 1 | public boolean add(Object element) | is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection c) | is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | is used to delete an element from this collection. |

| | | |
|---|---|---|
| 4 | public boolean removeAll(Collection c) | is used to delete all the elements of specified collection from the invoking collection. |
| 5 | public boolean retainAll(Collection c) | is used to delete all the elements of invoking collection except the specified collection. |
| 6 | public int size() | return the total number of elements in the collection. |
| 7 | public void clear() | removes the total no of element from the collection. |
| 8 | public boolean contains(Object element) | is used to search an element. |
| 9 | public boolean containsAll(Collection c) | is used to search the specified collection in this collection. |
| 10 | public Iterator iterator() | returns an iterator. |
| 11 | public Object[] toArray() | converts collection into array. |
| 12 | public boolean isEmpty() | checks if collection is empty. |
| 13 | public boolean equals(Object element) | matches two collection. |
| 14 | public int hashCode() | returns the hashcode number for collection. |

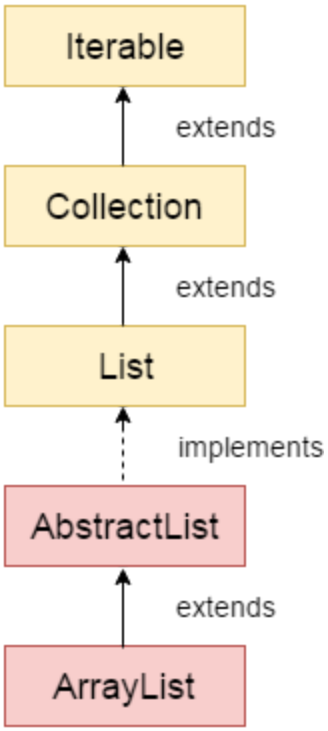
Iterator interfaces provides the capability of iterating over elements in forward direction only.

hasNext() returns true is the iterator has more elements.
next() returns the element and moves the cursor to the next element.

ArrayList:
Uses dynamic array for storing elements.

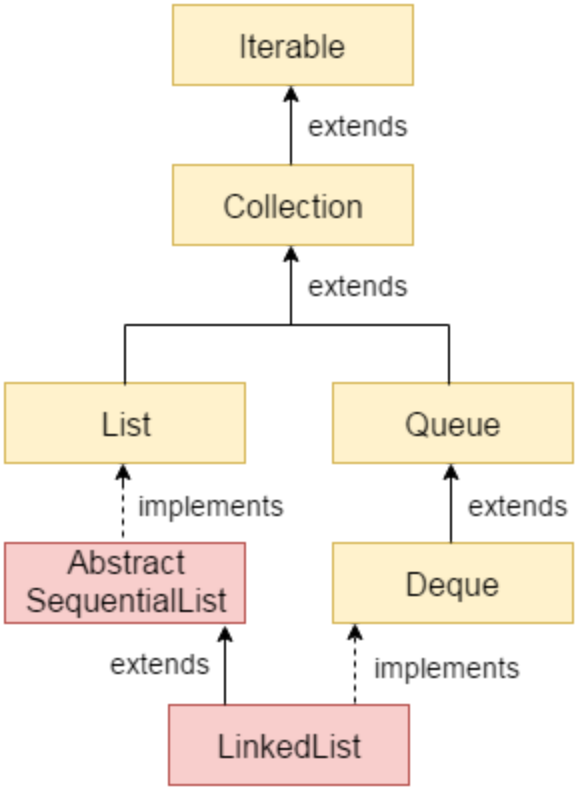
- Can contain duplicate elements
- Maintains insertion order
- Non synchronized
- Allows random access
- Manipulation is slow as shifting needs to be done on element removal

ArrayList can be iterated using iterator or using for-each loop




LinkedList:
Uses doubly linked list to store the elements.

- Can contain duplicate elements
- Maintains insertion order
- Non synchronized
- Manipulation is fast as no rearranging is necessary
- Can be used as a list, stack, or queue

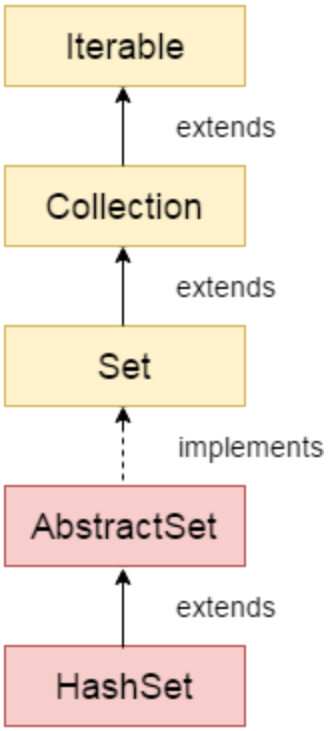



| ArrayList | LinkedList |
|---|---|
| 1) ArrayList internally uses **dynamic array** to store the elements. | LinkedList internally uses **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| 3) ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

HashSet:
Is is used to create a collection that uses hash table for data storage.

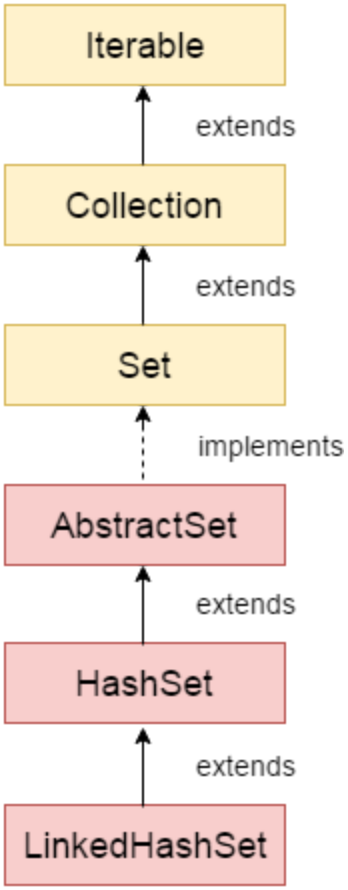
- Contains only unique elements
- Uses hashing to store the elements
- Does not necessarily maintain insertion order

A set can only contain unique elements whereas a list can contain duplicate elements.



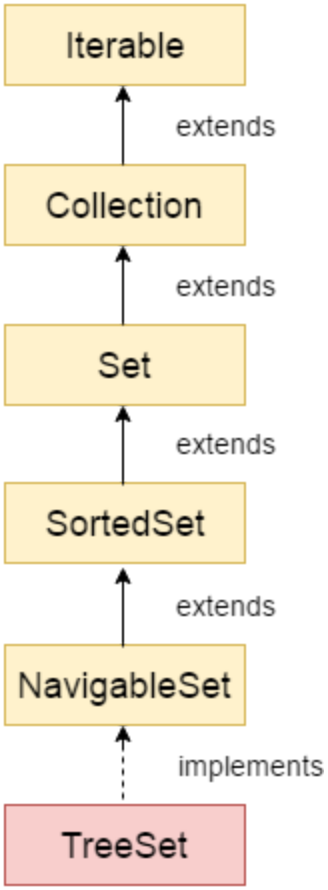

LinkedHashSet:

- Contains unique elements like HashSet
- Allows null elements
- Maintains insertion order




TreeSet:

- Contains unique elements like HashSet
- Access and retrieval times are quite fast
- Maintain ascending order




Queue:

- Orders the elements in FIFO manner

Priority Queue:

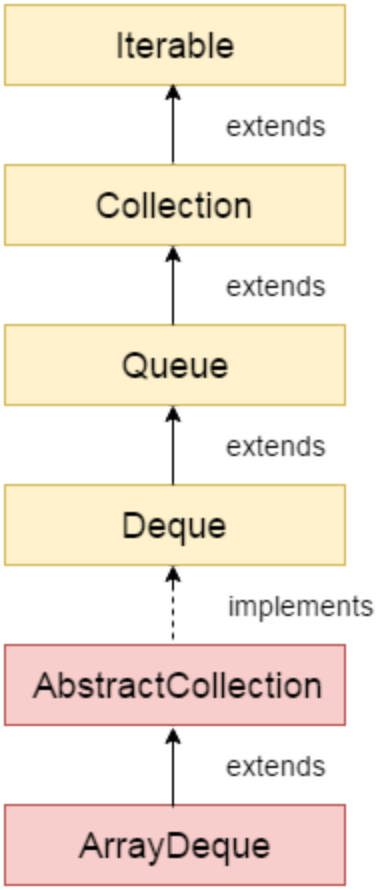
- The elements are not ordered in FIFO manner but ordered by priority
- The elements must be of comparable type.
- For user defined objects (other than String and wrapper classes), they need to implement comparable interface.

Deque:

- Stands for double ended queue

Array Deque:

- Can add or remove elements from both sides
- Null elements are not allowed
- Not thread safe
- Faster than linked list and stack
- No capacity restrictions

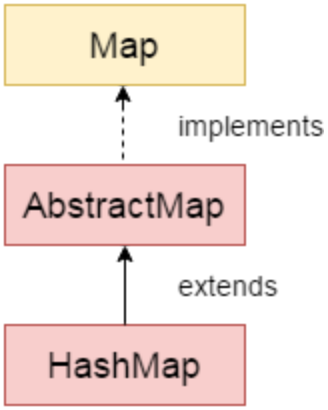



Map:

- Contains key-value pair
- Can contain only unique keys
- Is useful when you need to search, update, or delete on the basis of keys

HashMap:

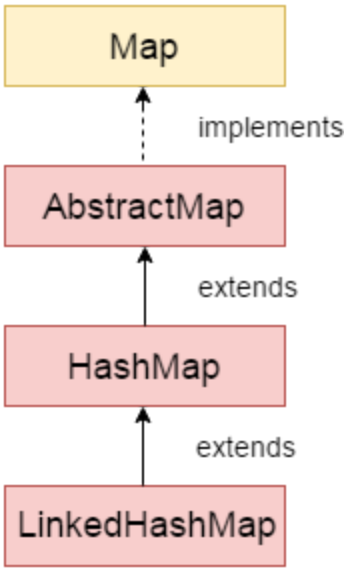
- It implements the map interface using Hash Table
- Maintains no order
- Can have one null key and multiple null values
- Contains unique elements




HashSet contains only values whereas HashMap contains key-value pairs.

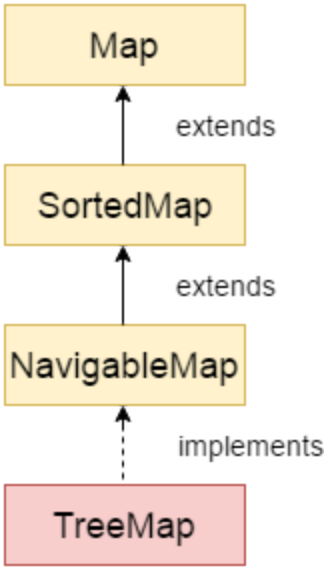
LinkedHashMap:

- Contains unique elements
- May have one null key and multiple null values
- Same as HashMap but maintains insertion order




TreeMap:

- Implements map interface using a tree
- Contains unique elements
- Cannot have null key but can have multiple null values
- Same as HashMap but maintains ascending order




HashTable:

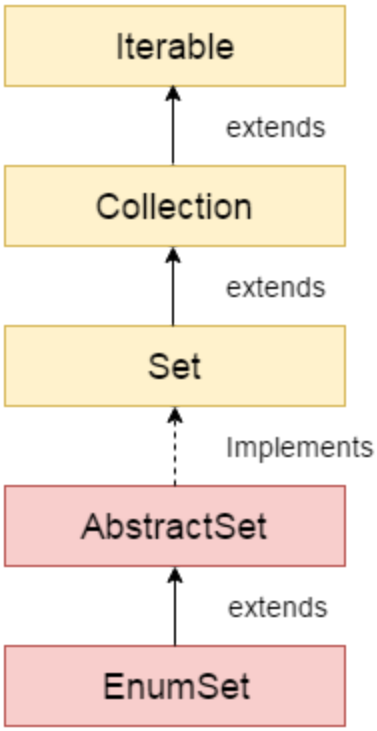
- A hash table is an array of list
- Each list is know as bucket
- The position of bucket is identified by calling hashCode() method of the Object
- Contains values based on key
- Contains unique elements
- Cannot have null key or value
- Is synchronized

| HashMap | Hashtable |
|---|---|
| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe and can be shared with many threads. |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |

| | |
|---|---|
| 5) We can make the HashMap as synchronized by calling this code<br>Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

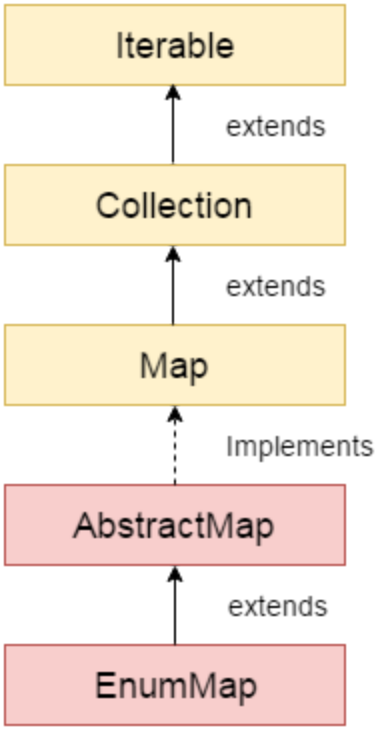
EnumSet:

- It is a specialized set implementation for enum types




EnumMap:

- It is a specialized map implementation for enum types




Collections can be sorted if they have elements as String, wrapper objects or user defined objects that implement Comparable interface

Comparator can be used for a more flexible approach where different comparators can be written for the same object and can be used interchangeably.

Properties class can be used to fetch from and store data to the properties file.
It has key and value both in the form of strings.
Recompilation is not required after changes.

| **ArrayList** | **Vector** |
|---|---|
| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| 2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity. | Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity. |

| | |
|---|---|
| 3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2. | Vector is a **legacy** class. |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object. |
| 5) ArrayList uses **Iterator** interface to traverse the elements. | Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also. |

Varargs:
- There can only be one vararg in method arguments
- A vararg must be the last in order in case of multiple arguments
- Can be accessed using for-each

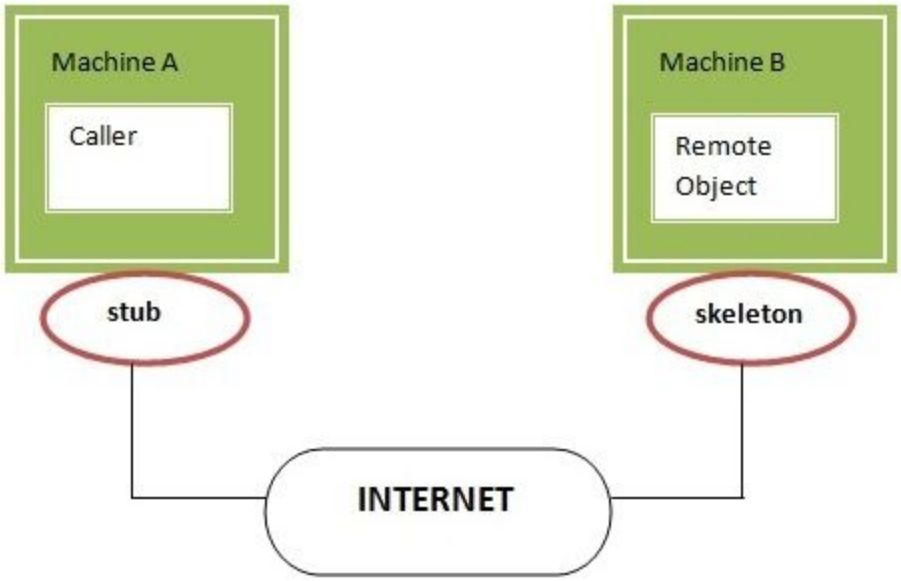
Enums:
- Convenient for fixed constants
- values() method is automatically added by compiler
- Values can be associated like WINTER(10)
- Can be used in a switch statement

Generics:
- Used for type-safety
- Custom generic class can be created using <T>
- T = type
- E = element
- K = key
- V = value
- N = number
- ? can be used as wildcard
- Example: ? extends Shape means any class that is a child of Shape

Java RMI stands for Remote Method Invocation. It provides a mechanism to created distributed applications in Java. RMI allows you to invoke a method executing in a different JVM.

Stub and skeleton are used in RMI.
A remote object is an object whose method is to be invoked.

Stub is an object that acts as a gateway on client side
Skeleton is an object that acts as a gateway on server side




Steps:

- Create remote interface by extending Remote interface
- import java.rmi.*;
- public interface Adder extends Remote{
- public int add(int x,int y)throws RemoteException;
- }
- Provide implementation of the remote interface
- import java.rmi.*;
- import java.rmi.server.*;
- public class AdderRemote extends UnicastRemoteObject implements Adder{
- AdderRemote()throws RemoteException{
- super();
- }
- public int add(int x,int y){return x+y;}
- }
- Create stub and skeleton objects using rmic tool
- rmic AdderRemote

- Start registry service using rmiregistry tool
- rmiregistry 5000
- Create and run server application
- import java.rmi.*;
- import java.rmi.registry.*;
- public class MyServer{
- public static void main(String args[]){
- try{
- Adder stub=new AdderRemote();
- Naming.rebind("rmi://localhost:5000/sonoo",stub);
- }catch(Exception e){System.out.println(e);}
- }
- }
- Create and run client application
- import java.rmi.*;
- public class MyClient{
- public static void main(String args[]){
- try{
- Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");
- System.out.println(stub.add(34,4));
- }catch(Exception e){}
- }
- }