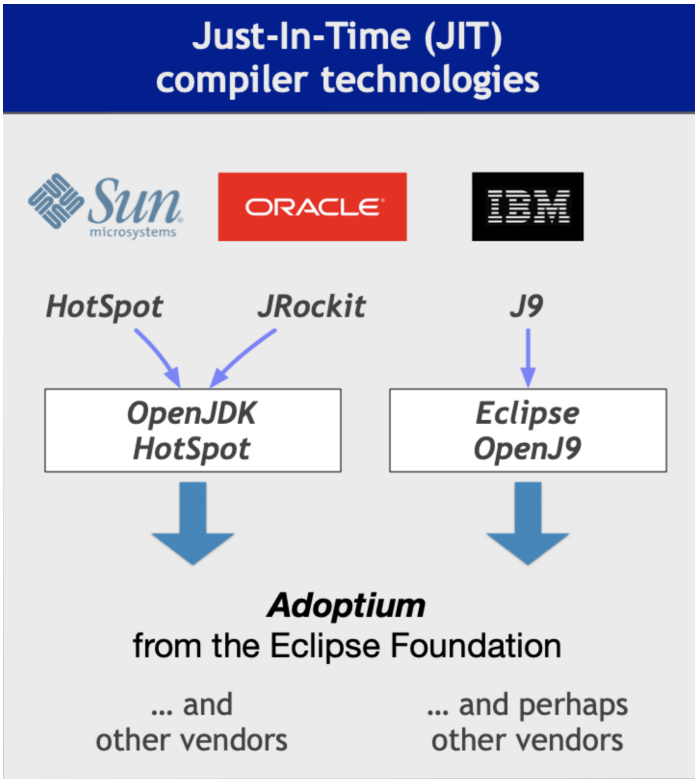
Java Virtual Machine

A Java Virtual Machine (JVM) is an imaginary computer that has never been physically built as hardware. A JVM runs programs compiled to its imaginary instruction set written to storage as an intermediate representation known as bytecode.

At runtime, the bytecode must be translated from the imaginary instruction set to the actual instruction set of the CPU of the host machine. This can be done on-the-fly by an interpreter. Or the bytecode can be fully compiled and cached, to run faster than through the interpreter, in a process known as Just-In-Time (JIT) compiling.

Over the decades, there have been many implementations of the JVM. Most have fallen away.



HotSpot is one implementation of JIT technology that starts by running interpreted, and watches the actual performance of the app. Parts of the app are then selected to be fully-compiled as native code and cached, for much faster execution. HotSpot was developed at Sun as a commercial product. After acquiring Sun, Oracle further evolved HotSpot by combining important parts of their competing product, JRockit. HotSpot is now open-sourced through the **OpenJDK** project, available free-of-charge.

Another such implementation in **OpenJ9**, developed by IBM, and now open-sourced through the Eclipse Foundation and available free-of-charge. Some JVM distributions built on OpenJDK replace HotSpot with OpenJ9 while still using the rest of OpenJDK such as the Java SE class libraries. For example, the pre-built distributions available at AdoptOpenJDK provide your choice of HotSpot or OpenJ9 on some hardware.

AdoptOpenJDK - open and reproducible build & test system for OpenJDK source across multiple platforms. AdoptOpenJDK provides rock-solid OpenJDK and Eclipse OpenJ9 (with OpenJDK class libraries) binaries for the Java ecosystem and also provides infrastructure as code, and a Build farm for builders of **OpenJDK** and Eclipse **OpenJ9** (with OpenJDK class libraries), on any platform.

HotSpot JVM

Hotspot provides:

- A Java Classloader
- A Java bytecode interpreter
- Client and Server virtual machines, optimized for their respective uses
- Several garbage collectors (including a very-low-pause-time ZGC)
- A set of supporting runtime libraries

The HotSpot JDK includes two flavors of the VM -- a **client**-side offering, and a VM tuned for **server** applications. These two solutions share the Java HotSpot runtime environment code base, but use different compilers that are suited to the distinctly unique performance characteristics of clients and servers. These differences include the compilation inlining policy and heap defaults.

Although the Server and the Client VMs are similar, the **Server VM has been specially tuned to maximize peak operating speed**. It is intended for executing long-running server applications, which need the fastest possible operating speed more than a fast start-up time or smaller runtime memory footprint.

The Client VM compiler serves as an upgrade for both the Classic VM and the just-in-time (JIT) compilers used by previous versions of the JDK. The Client VM offers improved run time performance for applications and applets. The Java **HotSpot Client VM has been specially tuned to reduce application start-up time and memory footprint**, making it particularly well suited for client environments. In general, the client system is better for GUIs.

The Client VM compiler does not try to execute many of the more complex optimizations performed by the compiler in the Server VM, but in exchange, it requires less time to analyze and compile a piece of code. This means the Client VM can start up faster and requires a smaller memory footprint.

The **Server VM contains an advanced adaptive compiler that supports many of the same types of optimizations** performed by optimizing C++ compilers, as well as some optimizations that cannot be done by traditional compilers, such as **aggressive inlining across virtual method invocations**. This is a competitive and performance advantage over static compilers. Adaptive optimization technology is very flexible in its approach, and typically outperforms even advanced static analysis and compilation techniques.

The 64-bit version of the Java SE Development Kit (JDK) currently ignores this option and instead uses the Server JVM.

Architecture	OS	Default client VM	if server-class, server VM; otherwise, client VM	Default server VM
SPARC 32-bit	Solaris		X	
i586	Solaris		X	
i586	Linux		X	
i586	Microsoft Windows	X		
SPARC 64-bit	Solaris	—		X
AMD64	Solaris	—		X
AMD64	Linux	—		X
AMD64	Microsoft Windows	—		X

Legend: **X** = default VM — = client VM not provided for this platform

Some JVM configuration properties:

- -Xms - set **initial heap size**
- -Xmx - set **maximum heap size**
- -Xss - set **thread stack size**
- -verbose:gc - logs garbage collector runs and how long they're taking.
- -XX:+PrintGCDetails - includes the data from -verbose:gc but also adds information about the size of the new generation and more accurate timings.
- -XX:+PrintGCTimeStamps - Print timestamps at garbage collection.
- -XX:+HeapDumpOnOutOfMemoryError - To trigger heap dump on out of memory
- -XX:+TraceClassLoading - to print logging information whenever classes loads into JVM
- -Xprof - Java Profiling is the process of monitoring various JVM levels parameters such as Method Executions, Thread Executions, Garbage Collections, and Object Creations

Memory

The Java Virtual Machine (JVM) divides memory between Java Heap Space and Java Stack Memory.

Java Heap Space

It is created by the Java Virtual Machine when it starts. The memory is used as long as the application is running. When an object is created, it is always created in Heap and has global access. That means all objects can be referenced from anywhere in the application.

Java String Pool

When we create a String variable and assign a value to it, the JVM searches the pool for a String of equal value. If found, the Java compiler will simply return a reference to its memory address, without allocating additional memory. If not found, it'll be added to the pool (interned) and its reference will be returned.

From Java 7 onwards, the Java String Pool is stored in the Heap space, which is garbage collected by the JVM. The advantage of this approach is the reduced risk of OutOfMemory error because unreferenced Strings will be removed from the pool, thereby releasing memory.

Java Integer Cache

Basically, the Integer class keeps a cache of Integer instances in the range of -128 to 127, and all autoboxing, literals and uses of Integer.valueOf() will return instances from that cache for the range it covers.

The purpose is mainly to save memory, which also leads to faster code due to better cache efficiency.

This is based on the assumption that these small values occur much more often than other ints and therefore it makes sense to avoid the overhead of having different objects for every instance (an Integer object takes up something like 12 bytes).

JVM Code Cache

JVM Code Cache is an area where JVM stores its bytecode compiled into native code. We call each block of the executable native code a nmethod. The nmethod might be a complete or inlined Java method.

The **just-in-time (JIT) compiler is the biggest consumer of the code cache area**. That's why some developers call this memory a JIT code cache.

Java Stack Memory

This is the temporary memory where variable values are stored when their methods are invoked. After the method is finished, the memory containing those values is cleared to make room for new methods.

When a new method is invoked, a new block of memory will be created in the Stack. This new block will store the temporary values invoked by the method and references to objects stored in the Heap that are being used by the method.

Any values in this block are only accessible by the current method and will not exist once it ends. When the method ends, that block will be erased. The next method invoked will use that empty block.

This “last in, first out” method makes it easy to find the values needed and allows fast access to those values.

Classloaders

Class loaders are responsible for **loading Java classes during runtime dynamically** to the JVM (Java Virtual Machine). Also, they are part of the JRE (Java Runtime Environment). Hence, the JVM doesn't need to know about the underlying files or file systems in order to run Java programs thanks to class loaders.

Also, these Java classes aren't loaded into memory all at once, but when required by an application. This is where class loaders come into the picture. They are responsible for loading classes into memory.

Java classes are loaded by an instance of java.lang.ClassLoader. However, class loaders are classes themselves. The java.lang.ClassLoader itself is loaded by bootstrap class loader.

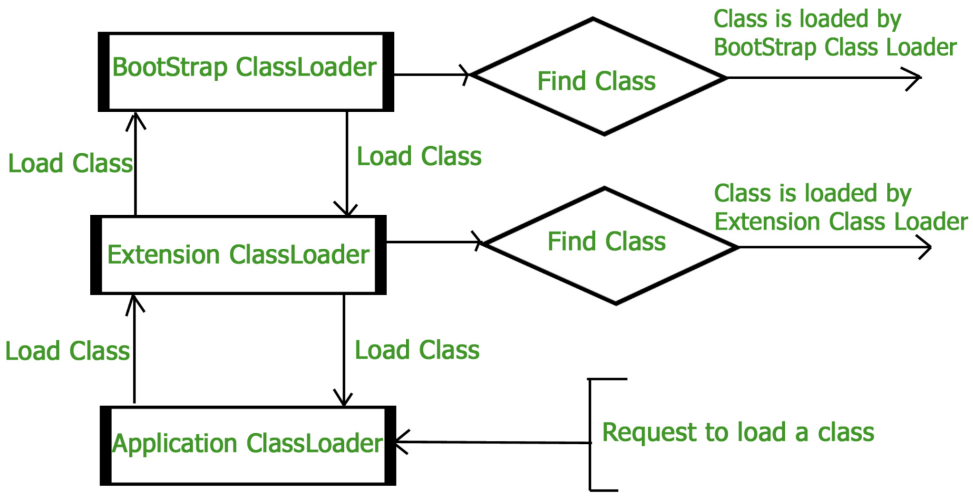
- **Bootstrap Class Loader**: it is mainly responsible for loading JDK internal classes, typically rt.jar and other core libraries located in $JAVA_HOME/jre/lib directory. Additionally, Bootstrap class loader serves as a **parent** of all the other ClassLoader instances.

  Bootstrap class loader is part of the core JVM and is written in native code. Different platforms might have different implementations of this particular class loader.

- **Extension Class Loader** - it is a child of the bootstrap class loader and takes care of loading the extensions of the standard core Java classes so that it's available to all applications running on the platform.

  Extension class loader loads from the JDK extensions directory, usually $JAVA_HOME/lib/ext directory or any other directory mentioned in the java.ext.dirs system property.

- **System Class Loader** - takes care of loading all the application level classes into the JVM. It loads files found in the classpath environment variable, -classpath or -cp command line option. Also, it's a child of Extension classloader.



Class loaders are part of the Java Runtime Environment. When the JVM requests a class, the class loader tries to locate the class and load the class definition into the runtime **using the fully qualified class name**.

The java.lang.ClassLoader.loadClass() method is responsible for loading the class definition into runtime. It tries to load the class based on a fully qualified name. If the class isn't already loaded, **it delegates the request to the parent class loader**. This process happens recursively.

Eventually, if the parent class loader doesn’t find the class, then the child class will call java.net.URLClassLoader.findClass() method to look for classes in the file system itself. If the last child class loader isn't able to load the class either, it throws java.lang.NoClassDefFoundError or java.lang.ClassNotFoundException.

The ClassLoader Delegation Hierarchy Model always functions in the order **Application ClassLoader->Extension ClassLoader->Bootstrap ClassLoader**. The Bootstrap ClassLoader is always given the higher priority, next is Extension ClassLoader and then Application ClassLoader.

Garbage Collection

Java garbage collection is the process by which Java programs perform **automatic memory management**. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

The garbage collection implementation lives in the JVM. Each JVM can implement its own version of garbage collection. However, it should meet the standard JVM specification of working with the objects present in the heap memory, marking or identifying the unreachable objects, and destroying them with compaction.

Garbage collectors work on the concept of Garbage Collection Roots (**GC Roots**) to identify live and dead objects:

- Classes loaded by system class loader (not custom class loaders)
- Live threads
- Local variables and parameters of the currently executing methods
- Local variables and parameters of JNI methods
- Global JNI reference
- Objects used as a monitor for synchronization
- Objects held from garbage collection by JVM for its purposes

The garbage collector traverses the whole object graph in memory, starting from those Garbage Collection Roots and following references from the roots to other objects.

A standard Garbage Collection implementation involves three phases:

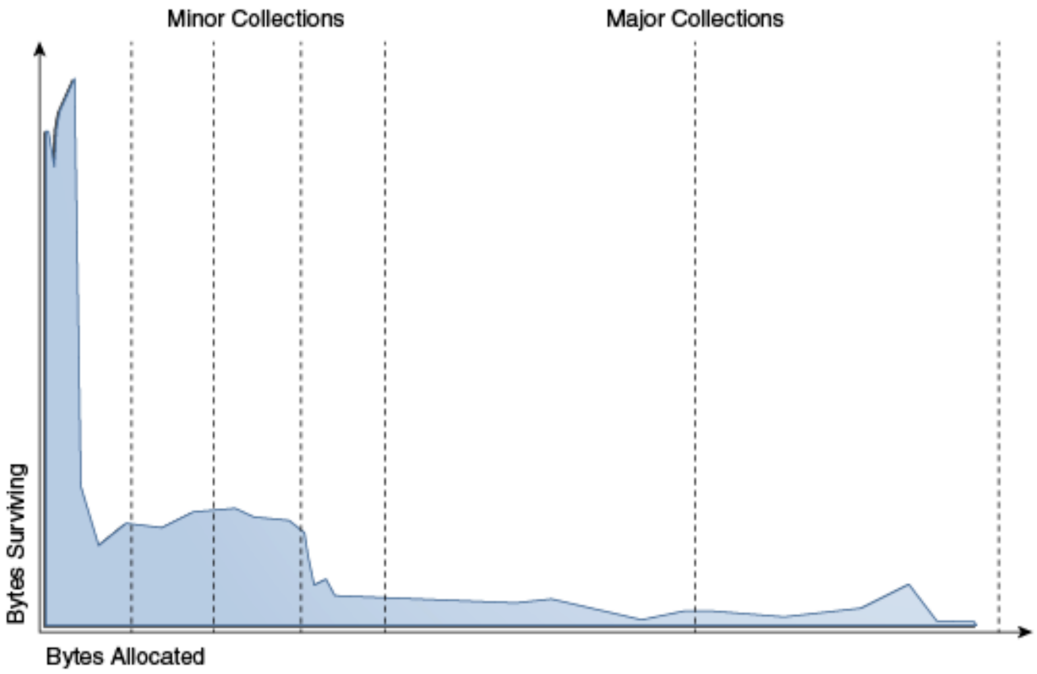
1. Mark objects as alive - In this step, the GC identifies all the live objects in memory by traversing the object graph.

2. Sweep dead objects - After marking phase, we have the memory space which is occupied by live (visited) and dead (unvisited) objects. The sweep phase releases the memory fragments which contain these dead objects.

3. Compact remaining objects in memory - The dead objects that were removed during the sweep phase may not necessarily be next to each other. Thus, you can end up having fragmented memory space. Memory can be compacted after the garbage collector deletes the dead objects, so that the remaining objects are in a contiguous block at the start of the heap. The compaction process makes it easier to allocate memory to new objects sequentially.

Generational Garbage Collection: Having to mark and compact all the objects in a JVM is inefficient. As more and more objects are allocated, the list of objects grows, leading to longer garbage collection times. Empirical analysis of applications has shown that most objects in Java survive for only a short period of time (**Weak generational hypothesis**)

Most of the garbage collector implementations distinguish between new objects and old objects, the “young generation” versus “old generation”. The goal is to make it cheaper for the programmer to create short-lived objects as they will be more quickly and efficiently disposed of to free up memory.



Description of "Figure 3-1 Typical Distribution for Lifetimes of Objects"

Young Generation - Newly created objects start in the Young Generation. When objects are garbage collected from the Young Generation, it is a **minor garbage collection** event.

Old Generation - Objects that are long-lived are eventually moved from the Young Generation to the Old Generation. This is also known as Tenured Generation, and contains objects that have remained in the survivor spaces for a long time. When objects are garbage collected from the Old Generation, it is a **major garbage collection** event.

Permanent Generation (replaced with **MetaSpace** starting with Java 8) - Metadata such as classes and methods are stored in the Permanent Generation. It is populated by the JVM at runtime based on classes in use by the application. Classes that are no longer in use may be garbage collected from the Permanent Generation.

Garbage Collectors are often benchmarked by:

- **Max Stop-The-World Time** (Stop-The-World means that the execution of the program is suspended for GC) - The VM will adjust the java heap size and other GC-related parameters in an attempt to keep GC-induced pauses shorter than nnn milliseconds. (-XX:MaxGCPauseMillis=nnn)
- **Throughput** - GC aims to size the heap so that the time spent in garbage collection is below the ratio determined by the -XX:GCTimeRatio. For example -XX:GCTimeRatio=19 sets a goal of 5% of the total time for GC and throughput goal of 95%. That is, the application should get 19 times as much time as the collector.
- **Heap Size** (memory footprint): -Xms and -Xmx

High load server-side apps generally seek short and predictable STW-time by means of server resources (HeapSize~RAM, Throughput~CPU).

Serial Garbage Collector

Serial - This is the simplest GC implementation, as it basically works with a single thread. As a result, this GC implementation freezes all application threads when it runs. Hence, it is not a good idea to use it in multi-threaded applications like server environments.

The Serial GC is the garbage collector of choice for most applications that do not have small pause time requirements and run on client-style machines. To enable Serial Garbage Collector, we can use the following argument:

```
java -XX:+UseSerialGC -jar Application.java
```

Parallel Garbage Collector

It's the default GC of the JVM and sometimes called Throughput Collectors. Unlike Serial Garbage Collector, this uses multiple threads for managing heap space. But it also freezes other application threads while performing GC.

If we use this GC, we can specify maximum garbage collection **threads** and **pause time**, **throughput**, and **footprint** (heap size):

```
-XX:ParallelGCThreads=<N>
-XX:MaxGCPauseMillis=<N>
-XX:GCTimeRatio=<N>
-Xmx<N>
```

To enable Parallel Garbage Collector, we can use the following argument:

```
java -XX:+UseParallelGC -jar Application.java
```

CMS Garbage Collector

The Concurrent Mark Sweep (CMS) implementation uses multiple garbage collector threads for garbage collection. As of Java 9, the CMS garbage collector has been deprecated, Java 14 completely dropped the CMS support.

G1 Garbage Collector

G1 (Garbage First) Garbage Collector is designed for applications running on multi-processor machines with large memory space. It's available since JDK7 Update 4 and in later releases.

G1 collector will replace the CMS collector since it's more performance efficient.

Unlike other collectors, G1 collector partitions the heap into a set of equal-sized heap regions, each a contiguous range of virtual memory. When performing garbage collections, G1 shows a concurrent global marking phase (i.e. phase 1 known as Marking) to determine the liveness of objects throughout the heap.

After the mark phase is completed, G1 knows which regions are mostly empty. It collects in these areas first, which usually yields a significant amount of free space (i.e. phase 2 known as Sweeping). It is why this method of garbage collection is called Garbage-First.

To enable the G1 Garbage Collector, we can use the following argument:

```
java -XX:+UseG1GC -jar Application.java
```

Shenandoah Garbage Collector

Shenandoah is a low-latency garbage collector that enables Java applications to operate quickly without changes. The feature was first introduced upstream in JDK 12 and has since been backported to the long-term support JDK 11.

Shenandoah’s key advance over G1 is to do more of its garbage collection cycle work concurrently with the application threads. G1 can evacuate its heap regions, that is, move objects, only when the application is paused, while Shenandoah can relocate objects concurrently with the application. To achieve the concurrent relocation, it uses what’s known as a Brooks pointer. This pointer is an additional field that each object in the Shenandoah heap has and which points back to the object itself.

To use Shenandoah in your application from Java 12 onwards, enable it with the following options:

```
-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC
```

ZGC Garbage Collector

The Z Garbage Collector, also known as ZGC, is a scalable low latency garbage collector designed to meet the following goals:

- Sub-millisecond max pause times
- Pause times do not increase with the heap, live-set or root-set size
- Handle heaps ranging from a 8MB to 16TB in size

At its core, ZGC is a concurrent garbage collector, meaning all heavy lifting work is done while Java threads continue to execute. This greatly limits the impact garbage collection will have on your application's response time.

It is a region-based collector, which means that the heap is divided into smaller regions and the compaction efforts will be focused on a subset of those regions, the ones with the most garbage. It is a single generation collector, it does not have young or old generations, yet.

One important thing to mention is that currently, pause times **do not increase with the heap or live size**, however, pause times **do increase with the root-set size** (number of java threads that your application is using).

ZGC intends to provide stop-the-world phases as short as possible. It achieves it in such a way that the duration of these pause times doesn't increase with the heap size. **These characteristics make ZGC a good fit for server applications, where large heaps are common, and fast application response times are a requirement**.

From JDK 15 in advance, you can use it just by specifying:

```
-XX:+UseZGC
```

Data Types

Types in Java are divided into two categories—primitive types and reference types.

Primitive Data Types

Type	Description	Wrapper class
byte	8-bit signed two's complement integer.	Byte
short	16-bit signed two's complement integer.	Short
int	32-bit signed two's complement integer.	Integer
long	64-bit two's complement integer.	Long
float	single-precision 32-bit IEEE 754 floating point.	Float
double	double-precision 64-bit IEEE 754 floating point.	Double
boolean	The boolean data type has only two possible values: true and false. Its "size" isn't something that's precisely defined.	Boolean
char	16-bit Unicode character.	Character

Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects. Sometimes you must use wrapper classes, for example when working with Collection objects, such as ArrayList, where primitive types cannot be used (the list can only store objects).

Reference Data Types

https://docs.oracle.com/javase/specs/jls/se15/preview/specs/class-terminology-jls.html

Reference types are the **class** types, the **interface** types, and the **array** types. An **enum** declaration is a kind of class declaration that introduces a special kind of class, an enum class. An **annotation** declaration is a kind of interface declaration that introduces a special kind of interface, an annotation interface.

The reference types are implemented by dynamically created objects that are either instances of classes or arrays. Many references to each object can exist. All objects (including arrays) support the methods of the class Object, which is the (single) root of the class hierarchy.

Class	It is a set of instructions. It describes the content of the object.
Array	It provides the fixed-size data structure that stores the elements of the same type.
Annotation	It provides a way to associate metadata with program elements.
Interface	It is implemented by Java classes.
Enumeration	It is a special kind of class that is type-safe. Each element inside the enum is an instance of that enum.

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

Object	clone()	Creates and returns a copy of this object.
boolean	**equals(Object obj)**	Indicates whether some other object is "equal to" this one.
void	finalize()	The finalization mechanism is inherently problematic. Deprecated.
Class<?>	**getClass()**	Returns the runtime class of this Object.
int	**hashCode()**	Returns a hash code value for the object. By default, integer value is mostly derived from **memory address** of the object in heap (but it's not mandatory always)
void	notify()	Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll()	Wakes up all threads that are waiting on this object's monitor.
String	**toString()**	Returns a string representation of the object.

void	wait()	Causes the current thread to wait until it is awakened, typically by being notified or interrupted.

We can compare the reference types in Java. Java provides two ways to compare reference types:

- By using the equal (==) operator - It compares the memory locations of the objects. If the memory address (reference) of both objects is the same, the objects are equal. Note that it does not compare the contents of the object.
- By using the Object.equals() Method - an equivalence relation on non-null object references:
    - It is **reflexive**: for any non-null reference value x, x.equals(x) should return true.
    - It is **symmetric**: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
    - It is **transitive**: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
    - It is **consistent**: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
    - For any non-null reference value x, x.equals(null) should return false.

HashCode contract:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

- It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

In Java there are **four types of references** differentiated on the way by which they are garbage collected: Strong References, Weak References, Soft References, Phantom References.

Strong references

This is the default type/class of Reference Object. Any object which has an active strong reference are not eligible for garbage collection. The object is garbage collected only when the variable which was strongly referenced points to null.

```
MyClass obj = new MyClass ();
```

Weak references

Suppose that the garbage collector determines at a certain point in time that an object is weakly reachable. At that time it will atomically clear all weak references to that object and all weak references to any other weakly-reachable objects from which that object is reachable through a chain of strong and soft references.

```
Object o = new Object();
WeakReference<Object> objectWeakReference = new WeakReference<>(o);
o = null;
```

This class provides the easiest way to harness the power of weak references. It is useful for implementing "registry-like" data structures, where the utility of an entry vanishes when its key is no longer reachable by any thread.

Soft references

In Soft reference, even if the object is free for garbage collection then also it is not garbage collected, until JVM is in need of memory badly. Softly referenced objects are collected as a last resort before an OutOfMemoryError is thrown. You can quite easily build a very simple and efficient cache using this type of reference.

```
Object o = new Object();
SoftReference<Object> softReference = new SoftReference<>(o);
o = null;
```

“*All soft references to softly-reachable objects are guaranteed to have been freed before the virtual machine throws an OutOfMemoryError*” - **that’s a lie**. It was true when soft references were first introduced in java 1.2, but from java 1.3.1 the jvm property -XX:SoftRefLRUPolicyMSPerMB was introduced. It defaults to 1000 (milliseconds), meaning that if there’s only 10MB available heap, the garbage collector will free references that have been used more than 10s ago. I.e. everything else will not be freed, leading to an OutOfMemoryError, breaking the guarantee from the javadoc. No problem, let’s just set it to -XX:SoftRefLRUPolicyMSPerMB=0 and the javadoc is suddenly true again.

When the GC figures that memory is running low and it better frees some softly referenced objects, **it will free all of them**. This will make our cache very inefficient, because it's expensive to recreate those objects. It would be better if the GC would only free a small portion of the available soft references.

Phantom references

The objects which are being referenced by phantom references are eligible for garbage collection. But, before removing them from the memory, JVM puts them in a queue called 'reference queue

```
Object o = new Object();
ReferenceQueue<Object> refQueue = new ReferenceQueue<>();
PhantomReference<Object> ref = new PhantomReference<>(o, refQueue);
o = null;
```

Arrays

In the Java programming language, arrays are objects, are dynamically created, and may be assigned to variables of type Object. All methods of class Object may be invoked on an array.

Array stores a fixed-size sequential collection of elements of the same type. It can contain primitives (int, char, etc.) as well as object (or non-primitive) references of a class depending on the definition of the array. In case of primitive data types, the actual values are stored in contiguous memory locations. In case of objects of a class, the actual objects are stored in heap segment.

Some other properties:

- In Java all arrays are dynamically allocated.(discussed below)
- Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++ where we find length using sizeof.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.
- Java array can also be used as a static field, a local variable or a method parameter.
- The size of an array must be specified by an int or short value and not long.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

Java array object uses default (Object) implementation for hashcode() and equals() methods:

- **equals**(Object obj): a method provided by java.lang.Object that indicates whether some other object passed as an argument is "equal to" the current instance. The default implementation provided by the JDK is based on memory location — two objects are equal if and only if they are stored in the **same memory address**.
- **hashcode**(): a method provided by java.lang.Object that returns an integer representation of the object memory address. By default, this method returns a **random integer** that is unique for each instance. This integer might change between several executions of the application and won't stay the same.

It is important to notice here that neither equals nor hashcode methods called on an array object reflect the inner state (content) of the array.

The **java.util.Arrays** class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

Some examples:

```
int[] arr = {1,2,3};

List<Integer> list = Arrays.asList(1, 2, 3);
int[] copy = Arrays.copyOf(arr, arr.length);
//Compares two int arrays lexicographically:
int compareResult = Arrays.compare(arr, new int[]{1, 2, 3});
```

```
boolean equals = Arrays.equals(arr, new int[]{1, 2, 3});
int searchResult = Arrays.binarySearch(arr, 2);
int hashCode = Arrays.hashCode(arr);
IntStream stream = Arrays.stream(arr);

Arrays.fill(arr, 0);
Arrays.sort(arr);
```

Java OOP

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about **creating objects that contain both data and methods**.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

There are four main OOP concepts in Java.

Abstraction

Abstraction means using simple things to represent complexity. We all know how to turn the TV on, but we don’t need to know how it works in order to enjoy it. In Java, abstraction means simple things like objects, classes, and variables represent more complex underlying code and data. This is important because it lets avoid repeating the same work multiple times.

Data abstraction is the process of hiding certain details and showing only essential information to the user. Abstraction can be achieved with either **abstract classes** or **interfaces**.

The abstract keyword is a non-access modifier, used for classes and methods:

- Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Difference Between Interface and Abstract Class:

- Type of methods: Interface can have only abstract methods. An abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.
- Final Variables: Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
- Type of variables: Abstract class can have final, non-final, static and non-static variables. The interface has only static and final variables.
- Implementation: Abstract class can provide the implementation of the interface. Interface can’t provide the implementation of an abstract class.

- Inheritance vs Abstraction: A Java interface can be implemented using the keyword “implements” and an abstract class can be extended using the keyword “extends”.
- Multiple implementations: An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
- Accessibility of Data Members: Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

Interface can provide default implementation using default methods which enable you to add new functionality to existing interfaces and **ensure binary compatibility** with code written for older versions of those interfaces. In particular, default methods enable you to add methods that accept lambda expressions as parameters to existing interfaces.

Encapsulation

Encapsulation is the practice of keeping fields within a class private, then providing access to them via public methods. It’s a protective barrier that keeps the data and code safe within the class itself. This way, we can reuse objects like code components or variables without allowing open access to the data system-wide.

Advantages of Encapsulation:

- Data Hiding: The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.

- Increased Flexibility: We can make the variables of the class as read-only or write-know that only depending on our requirement. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program.

- Reusability: Encapsulation also improves the re-usability and ease of change with new requirements.

- Testing code is easy: Encapsulated code is easy to test for unit testing.

Java provides a number of access modifiers to set access levels for classes, variables, methods, and constructors. The four access levels are:

- Visible to the class only (**private**).
- Visible to the package, the **default**. No modifiers are needed.
- Visible to the package and all subclasses (**protected**).
- Visible to the world (**public**).

	within class	within package	outside package by subclass	outside package

Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

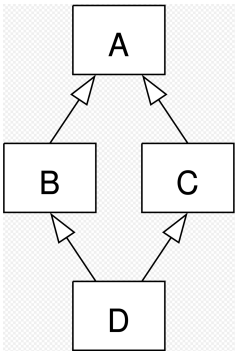
Inheritance

Inheritance lets programmers create new classes that share some of the attributes of existing classes. This lets us build on previous work without reinventing the wheel.

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface. The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

Why is multiple inheritance for classes not allowed?

The "**diamond problem**" is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?



Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of **default methods** which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

The default methods were introduced to provide **backward compatibility** so that existing interfaces can use the lambda expressions without implementing the methods in the implementation class. Default methods are also known as defender methods or virtual extension methods.

Until you don't create ambiguity by using default methods on interfaces, you are fine to use it. If it creates ambiguity, the compiler will alert you by throwing a compile-time error "inherits unrelated defaults". In case both the implemented interfaces contain default methods with the same method signature, the implementing class should **explicitly specify which default method** is to be used or it should override the default method.

```
public class Clazz implements IFace, IFace2 {
	@Override
	public int dom(int a) {
	   return IFace.super.dom(a);
	}
```

```
}

public interface IFace {
    default int dom(int a) { return 1;}
}
public interface IFace2 {
    default int dom(int a) {return 0; };
}
```

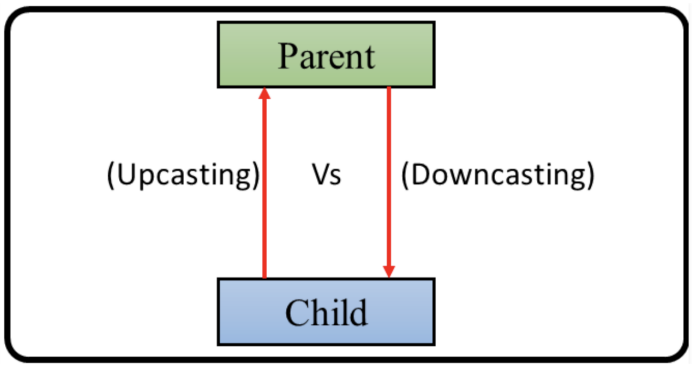
Polymorphism

The dictionary definition of **polymorphism** refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

Class Casting

Casting does not change the actual object type. Only the reference type gets changed.

If the reference variable of Parent class refers to the object of Child class, it is known as **upcasting**. Upcasting is always allowed, because the reference object of a subtype is always compatible with a supertype. Upcasting can be done implicitly.



Example:

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

Downcasting is casting to a subtype, downward to the inheritance tree. Example:

```
Object o = getSomeObject(),
String s = (String) o; // this is allowed because o could reference a String
```

The Java language provides the **instanceof** keyword to check the type of an object before casting. For example:

```
if (anim instanceof Cat) {
    Cat cat = (Cat) anim;
    cat.meow();
} else if (anim instanceof Dog) {
    Dog dog = (Dog) anim;
    dog.bark();
}
```

So if you are not sure about the original object type, use the instanceof operator to check the type before casting. This eliminates the risk of a ClassCastException thrown.

There's also another way to cast objects using the methods of Class:

```
public void whenDowncastToCatWithCastMethod_thenMeowIsCalled() {
    Animal animal = new Cat();
    if (Cat.class.isInstance(animal)) {
        Cat cat = Cat.class.cast(animal);
        cat.meow();
    }
}
```

Compile-time vs Runtime polymorphism

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.

Compile-time polymorphism is also known as static polymorphism. It is achieved by method overloading (or operator overloading, but Java doesn't support the Operator Overloading). In **method overloading**, a single method may perform different functions depending on the context in which it's called. That is, a single method name might work in different ways depending on what arguments are passed to it.

```
@Test
    void overloadingTest() {
        BaseClass baseClass = new BaseClass();

        System.out.println(baseClass.method1("1"));
        System.out.println(baseClass.method1((Object) "2"));
        System.out.println(baseClass.method1(new String[] {"3"}));
        System.out.println(baseClass.method1(new Object[] {"4"}));
    }

    public static class BaseClass {
        public String method1(String arg) {
            return "String:" + arg;
```

```
    }

    public String method1(String ... args) {
        return "String ... args:" + Arrays.stream(args)
                .collect(Collectors.joining());
    }

    public String method1(Object arg) {
        return "Object arg:" + arg.toString();
    }

    public String method1(Object ... args) {
        return "Object ... args:" + Arrays.stream(args)
                .map(Object::toString)
                .collect(Collectors.joining());
    }
}
```

Output:

```
String:1
Object arg:2
String ... args:3
Object ... args:4
```

Runtime polymorphism is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by **method Overriding**. It occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

```
@Test
    void overridingTest() {
        Parent cls = new Child();
        cls.say();
        System.out.println("Obtaining var externally: " + cls.var);
        System.out.println("Obtaining var externally using overriding: " + cls.getVar());
    }

    public static class Parent {
        public int var = 0;

        public void say() {
            System.out.println("I'm parent, var=" + var);
        }

        public int getVar() {
            return var;
        }
```

```
}

public static class Child extends Parent {
    public int var = 1;

    public void say() {
        System.out.println("I'm child, var=" + var);
    }

    @Override
    public int getVar() {
        return var; //we could access the parent var using super.var
    }
}
```

Hiding Fields: Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different. Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through super, which is covered in the next section. Generally speaking, we don't recommend hiding fields as it makes code difficult to read.

Java polymorphism from the C++ point of view:

By default, all the instance methods in Java are considered as the **Virtual function** except final, static, and private methods as these methods can be used to achieve polymorphism.

The virtual keyword is not used in Java to define the virtual function; instead, the virtual functions and methods are achieved using the following techniques:

- We can override the virtual function with the inheriting class function using the **same function name**. Generally, the virtual function is defined in the parent class and override it in the inherited class.
- The virtual function is supposed to be defined in the derived class. We can call it by referring to the derived class's object using the reference or **pointer of the base class**.
- A virtual function **should have the same name and parameters (method signature) in the base and derived class**.
- For the virtual function, an IS-A relationship is necessary, which is used to define the class hierarchy in inheritance.
- The Virtual function **cannot be private**, as the private functions cannot be overridden.
- A virtual function or method also **cannot be final**, as the final methods also cannot be overridden.
- Static functions are also cannot be overridden; so, a virtual function **should not be static**.
- By default, Every non-static method in Java is a virtual function.
- The virtual functions can be used to achieve oops concepts like **runtime polymorphism**.

Static, private, and final methods and variables are resolved using static binding which makes their execution fast because no time is wasted to find the correct method during runtime.

Final

While inheritance enables us to reuse existing code, sometimes we do need to set limitations on extensibility for various reasons; the final keyword allows us to do exactly that:

- Classes marked as final can’t be extended
- Methods marked as final cannot be overridden
- Variables marked as final can't be reassigned

Static

The **static variable** can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.

The static variable gets memory only once in the class area at the time of class loading. It makes your program memory efficient (i.e., it saves memory)

If you apply a **static** keyword with any method, it is known as a **static method**.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Why is the Java main method static? It is because the object is not required to call a static method. If the main() is allowed to be non-static, then while calling the main() method JVM has to instantiate its class. While instantiating it has to call the constructor of that class, There will be ambiguity if the constructor of that class takes an argument.

The main() method in Java must be declared public, static and void. If any of these are missing, the Java program will compile but a runtime error will be thrown.

Initialization order

1. Static fields and static blocks of ancestors. In each class they are initialized in order of appearance.
2. Static fields and static block of instantiated class.
3. Instance fields and initialization blocks of ancestors. In each class they are initialized in order of appearance.
4. Constructor of ancestor after initialization of its instance fields.
5. Instance fields and initialization blocks of the current class. Also in order of appearance.
6. Constructor of the current class.

```
public class InitOrder {

   //Print message and return given object
   static <T> T init(String name, T object) {
      System.out.printf("Initializing '%s' to '%s'%n", name, object);
      return object;
   }

   static class Base {
      private int i = init("Base instance i", 5);

      private static int x = init("Base static x", 1);

      private int ctorI;

      public Base() {
         ctorI = init("Base ctorI", 7);
      }

      private static int y = init("Base static y", 2);

      { // instance initialization block
         j = init("Base instance block j", 6);
      }

      private static int z;
      static { // static initialization block:
         z = init("Base.z static block", 3);
      }

      private int j;
   }

   static class Subclass extends Base {

      public Subclass() {
         ctorB = init("Subclass ctorB", 9);
      }
```

```
        int ctorB;

        int a = init("Subclass instance a", 8);

        static int foo = init("Subclass static foo", 4);
    }

    public static void main(String[] args) {
        System.out.println("First Subclass instance:");
        Base base = new Subclass();

        System.out.println("Another Subclass instance:");
        base = new Subclass();

        System.out.println("Instance of Base:");
        base = new Base();
    }
}
```

Output:

```
First Subclass instance:
Initializing 'Base static x' to '1'
Initializing 'Base static y' to '2'
Initializing 'Base.z static block' to '3'
Initializing 'Subclass static foo' to '4'
Initializing 'Base instance i' to '5'
Initializing 'Base instance block j' to '6'
Initializing 'Base ctorI' to '7'
Initializing 'Subclass instance a' to '8'
Initializing 'Subclass ctorB' to '9'

Another Subclass instance:
Initializing 'Base instance i' to '5'
Initializing 'Base instance block j' to '6'
Initializing 'Base ctorI' to '7'
Initializing 'Subclass instance a' to '8'
Initializing 'Subclass ctorB' to '9'

Instance of Base:
Initializing 'Base instance i' to '5'
Initializing 'Base instance block j' to '6'
Initializing 'Base ctorI' to '7'
```

Composition vs Inheritance

One of the advantages of an Object-Oriented programming language is code reuse. There are two ways we can do code reuse either by the implementation of **inheritance** (IS-A relationship), or object **composition** (HAS-A relationship). Although the compiler and Java virtual machine (JVM) will do a lot of work for you when you use inheritance, you can also get at the functionality of inheritance when you use composition.

In object-oriented programming, the concept of IS-A is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing". For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is **unidirectional**. For example, House is a Building. But Building is not a House.

It is a key point to note that you can easily identify the IS-A relationship. Wherever you see an **extends** keyword or **implements** keyword in a class declaration, then this class is said to have IS-A relationship.

Composition(HAS-A) simply mean the use of instance variables that are references to other objects. For example Maruti has Engine, or House has Bathroom.

Comparing Composition and Inheritance:

- It is easier to change the class implementing composition than inheritance. The change of a superclass impacts the inheritance hierarchy to subclasses.
- You can't add to a subclass a method with the same signature but a different return type as a method inherited from a superclass. Composition, on the other hand, allows you to change the interface of a front-end class without affecting back-end classes.
- Composition is dynamic binding (run-time binding) while Inheritance is static binding (compile time binding)
- It is easier to add new subclasses (inheritance) than it is to add new front-end classes (composition) because inheritance comes with polymorphism. If you have a bit of code that relies only on a superclass interface, that code can work with a new subclass without change. This is not true of composition unless you use composition with interfaces. Used together, composition and interfaces make a very powerful design tool.
- With both composition and inheritance, changing the implementation (not the interface) of any class is easy. The ripple effect of implementation changes remains inside the same class.
    - Don't use inheritance just to get code reuse If all you really want is to reuse code and there is no is-a relationship in sight, use composition.
    - Don't use inheritance just to get at polymorphism If all you really want is a polymorphism, but there is no natural is-a relationship, use composition with interfaces.

A good example of composition is Strategy Pattern:

```
public interface Flyable{
 public void fly();
}
```

```
public class Duck {
 Flyable fly;

 public Duck(){
  fly = new BackwardFlying();
 }
}
Thus we can have multiple classes which implement flying eg:

public class BackwardFlying implements Flyable{
  public void fly(){
    Systemout.println("Flies backward ");
  }
}
public class FastFlying implements Flyable{
  public void fly(){
    Systemout.println("Flies 100 miles/sec");
  }
}
```

To favor composition over inheritance is a design principle that gives the design **higher flexibility**. **It is more natural to build business-domain classes out of various components than trying to find commonality between them and creating a family tree**. For example, an accelerator pedal and a steering wheel share very few common traits, yet both are vital components in a car. What they can do and how they can be used to benefit the car is easily defined. Composition also provides a more stable business domain in the long term as it is less prone to the quirks of the family members. In other words, **it is better to compose what an object can do (HAS-A) than extend what it is (IS-A)**.

Initial design is simplified by identifying system object behaviors in separate interfaces instead of creating a hierarchical relationship to distribute behaviors among business-domain classes via inheritance. **This approach more easily accommodates future requirements changes that would otherwise require a complete restructuring of business-domain classes in the inheritance model**. Additionally, it avoids problems often associated with relatively minor changes to an inheritance-based model that includes several generations of classes. **Composition relation is more flexible as it may be changed on runtime**, while sub-typing relations are static and need recompilation in many languages.

Some languages, notably Go, use type composition exclusively.

Nested classes

The Java programming language allows you to define a class within another class. Such a class is called a nested class. Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are called static nested classes. Non-static nested classes are called inner classes.

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

A nested class is a member of its enclosing class. **As a member of the OuterClass, a nested class can be declared private, public, protected, or package private**. (Recall that **outer classes can only be declared public or package private**.)

Non-static nested classes (inner classes) have **direct access to other members of the enclosing class**, even if they are declared private. Also, because an inner class is associated with an instance, **it cannot define any static members itself**. A good example of utilizing Java inner class is a **Builder** pattern.

Static nested classes do not have access to other members of the enclosing class.

Why Use Nested Classes?

- **It is a way of logically grouping classes that are only used in one place**: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- **It increases encapsulation**: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- **It can lead to more readable and maintainable code**: Nesting small classes within top-level classes places the code closer to where it is used.

Playing with nested classes:

```
public class NestedClassTest {

    public int outerPublicField = 44;
    private int outerPrivateField = 10;
    private static String outerPrivateStaticField = "123";
```

```
    private void outerPrivateMethod() {
        System.out.println("outerPrivateMethod");
    }

    @Test
    void name() {
        InnerClass innerClass = new InnerClass();
        System.out.println(innerClass.innerPrivateField);
        innerClass.innerPublicMethod();
        innerClass.innerProtectedMethod();
        innerClass.innerDefaultMethod();
        innerClass.innerPrivateMethod();

        StaticNestedClass staticNestedClass = new StaticNestedClass();
        System.out.println(staticNestedClass.staticNestedPrivateField);
        System.out.println(StaticNestedClass.staticNestedPrivateStaticField);
        staticNestedClass.staticNestedPrivateMethod();
        staticNestedClass.staticNestedPublicMethod();
    }

    //As a member of the OuterClass, a nested class can be declared private, public, protected, or
package private
    private class InnerClass {
        private int innerPrivateField = 10;

        //static declarations in inner classes are not supported
        //private static void innerStaticMethod() {}

        //inner classes have access to other members of the enclosing class, even if they are declared
private
        public void innerPublicMethod() {
            System.out.println(outerPrivateField);
            outerPrivateMethod();
        }

        void innerDefaultMethod() {}
        private void innerPrivateMethod() {}
        private void innerProtectedMethod() {}
    }

    private static class StaticNestedClass {
        private static int staticNestedPrivateStaticField = 44;
        private int staticNestedPrivateField = 55;

        public void staticNestedPublicMethod() {
            System.out.println(outerPrivateStaticField);

            //Static nested classes DO NOT have access to other members of the enclosing class:
            //System.out.println(outerPrivateField);
```

```
            //System.out.println(outerPublicField);
        }
        private void staticNestedPrivateMethod() {

        }

        public static void staticNestedPublicStaticMethod() {
            System.out.println(outerPrivateStaticField);
        }
    }
}
```

Shadowing

If a declaration of a type (such as a member variable or a parameter name) in a particular scope (such as an inner class or a method definition) has the same name as another declaration in the enclosing scope, then the declaration shadows the declaration of the enclosing scope. You cannot refer to a shadowed declaration by its name alone. The following example, ShadowTest, demonstrates this:

```
public class ShadowTest {

    public int x = 0;

    class FirstLevel {

        public int x = 1;

        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
        }
    }

    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

The following is the output of this example:

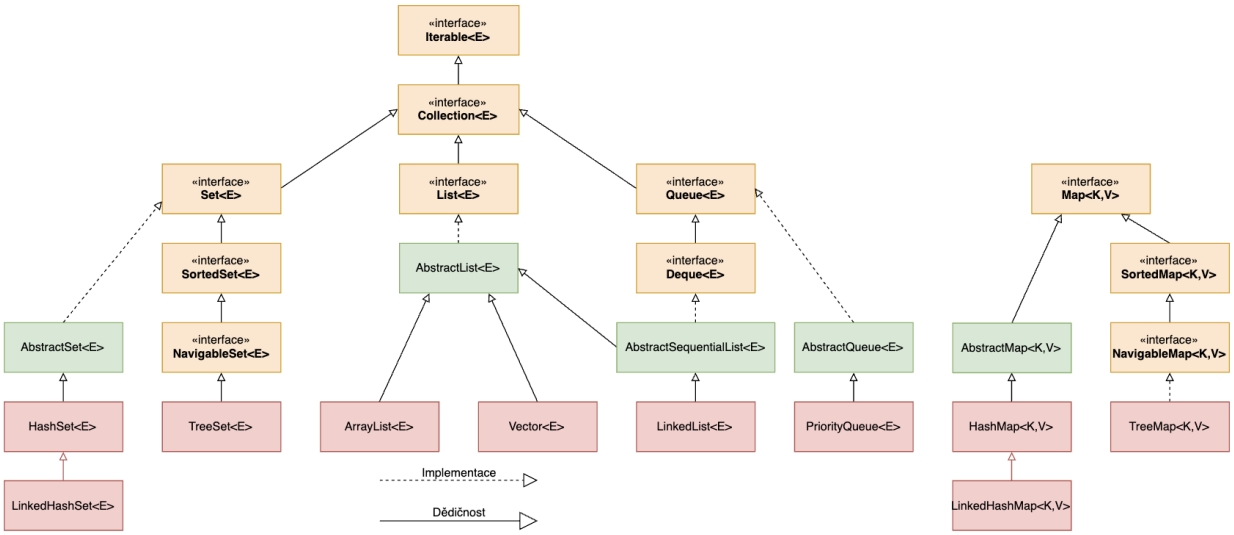
```
x = 23
this.x = 1
ShadowTest.this.x = 0
```

Collections

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

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

https://www.baeldung.com/java-collections



Collection

This is the root of the collection hierarchy. A collection represents a group of objects known as its elements. The Java platform doesn't provide any direct implementations of this interface.

The interface has methods to tell you how many elements are in the collection (**size, isEmpty**), to check whether a given object is in the collection (**contains**), to add and remove an element from the collection (**add, remove**), and to provide an iterator over the collection (**iterator**).

Collection interface also provides bulk operations methods that work on the entire collection – **containsAll, addAll, removeAll, retainAll, clear**.

The **toArray** methods are provided as a bridge between collections and older APIs that expect arrays on input.

Iterable

An Iterable represents a collection that can be traversed. Implementing the Iterable interface allows an object to make use of the for-each loop. It does that by internally calling the iterator() method on the object. For example, the following code only works as a List interface extends the Collection interface, and the Collection interface extends the Iterable interface.

- Iterator<T> **iterator**();
- default void **forEach**(Consumer<? super T> action) {...}
- default Spliterator<T> **spliterator**() {...}

Iterator

Iterator interface provides methods to iterate over any Collection. We can get an iterator instance from a Collection using the iterator method. Iterator takes the place of Enumeration in the Java Collections Framework. Iterators allow the caller to remove elements from the underlying collection during the iteration. Iterators in collection classes implement **Iterator Design Pattern**.

The contract for Iterable is that it should produce a new instance of an Iterator every time the iterator() method is called. This is because the Iterator instance maintains the iteration state, and things won't work as if the implementation returns the same Iterator twice.

- boolean **hasNext**();
- E **next**();
- default void **remove**() { throw new UnsupportedOperationException("remove"); }
- default void **forEachRemaining**(Consumer<? super E> action) {...}

ConcurrentModificationException

When you create an iterator, it starts to count the modifications that were applied on the collection. If the iterator detects that some modifications were made without using its method (or using another iterator on the same collection), it cannot guarantee anymore that it will not pass twice on the same element or skip one, so it throws this exception.

How does the iterator track down the structural modifications of the collection? AbstractList ancestor have a modCount member variable which gets incremented every time the structural change is made:

```
private void fastRemove(Object[] es, int i) {
        modCount++;
...
```

Every time a new iterator is instantiated, it remembers the current modCount value:

```
private class Itr implements Iterator<E> {
        int cursor;       // index of next element to return
```

```
        int lastRet = -1; // index of last element returned; -1 if no such
        int expectedModCount = modCount;
...
```

And it throws an exception if it detects a modification:

```
final void checkForComodification() {
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
        }
```

If you need to modify the collection you're iterating through, you should **remove items via iterator.remove** (and with only one iterator), or make a list of items to remove then **remove them after you finished iterating**.

List

An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

ArrayList

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list.

Complexity	Remove	Insert	Contains	Get by index
ArrayList	O(n)	O(n)	O(n)	O(1)

The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

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

LinkedList

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Complexity	Add first/last	Get	Insert	Remove
LinkedList	O(1)	O(n)	O(n)	O(n)

The Problem of Linked-List

The RAM-access speed is so slow that if CPU processes data directly on RAM, 99% of CPU speed will become a waste, and we can just make use of 1% of its power. Therefore, they invented CPU cache to solve this problem. The cache memory is very fast and is integrated directly on CPU. Its access speed is near the CPU speed. So now, instead of accessing data directly on RAM, **CPU will access data on RAM indirectly through L1 cache** (there are usually three levels of caches, and L1 cache is the fastest among them).

However, the CPU cache does not solve the problem completely because memory cache size is much smaller than RAM. We still need RAM as our main memory. And the CPU caches will just hold small pieces of data that are most likely to be needed by the CPU in the near future. Sometimes, the data that the CPU needs to access next is not already in L1 cache (nor in L2 or L3 cache), and it must be fetched from RAM, then the CPU will have to wait for several hundred cycles for the data to become available. This is what we call cache miss.

Therefore, to reduce cache miss, when the CPU wants to access data at address x in RAM, **it will not only fetch the data at address x, but also the neighborhood of address x.** Because we assume "if a particular memory location is referenced at a particular time, then **it is likely that nearby memory locations will be referenced in the near future**" This is what we call locality of reference. So, **if the data to be processed by the CPU is placed right next to each other, we can make use of locality of reference and reduce cache miss, which might cause huge performance overhead if it occurs often**.

Unlike array, which is a cache-friendly data structure because its elements are placed right next to each other, elements of linked-list can be placed anywhere in the memory. So **when iterating through linked-list, it will cause a lot of cache miss** (since we can't make use of locality of reference), and introduce lots of performance overheads.

Performance of Array vs Linked-List

When we perform random-insertion or random-deletion on an array, the subsequent elements need to be moved.

However, when it comes to a list of small elements (list of POD type, or list of pointers), **the cost of moving elements around is cheap, much cheaper than the cost of cache misses**. Therefore, most of the time, when working with a list of small data types (whatever type of operations it is: from iterating, random-insertion, random deletion to number crunching), the performance of array will be much better than linked-list.

But when we need to work with a list of large elements (> 32 bytes), the cost of moving elements around grows up to be higher than the cost of linked-lists "usual" cache misses. Then the linked list will have better performance than array.

In Java, we're forced to use a list of references that point to an object in the heap, instead of a list of objects, because reference types are placed on the heap by default and we can just make a reference to it. Since reference is just a managed pointer, its size is no more than 8 bytes.

Map

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. If the map previously contained a mapping for the key, the old value is replaced by the specified value.

Why Map interface does not extend Collection interface?

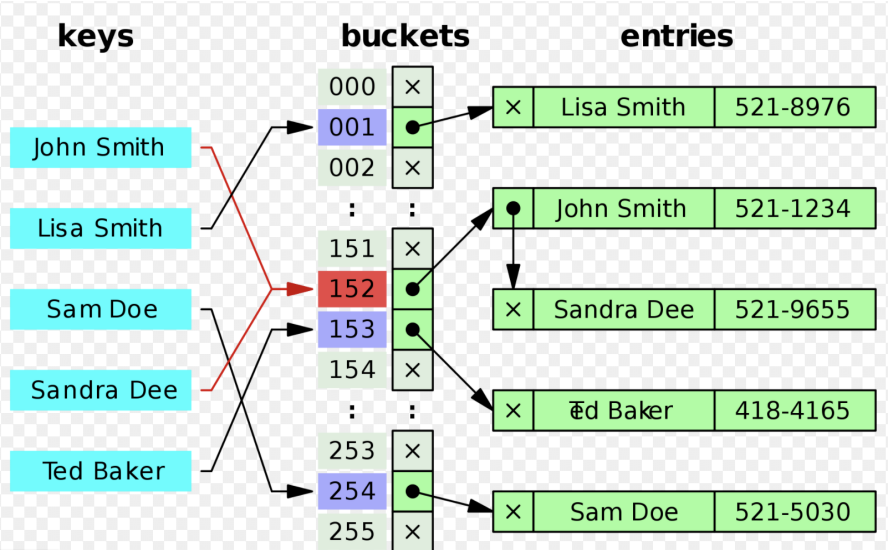
A good answer to this question is “because they are incompatible“. Collection has a method add(Object o). Map can not have such method because it need key-value pair. There are other reasons also such as Map supports keySet, valueSet etc. Collection classes does not have such views.

Due to such big differences, Collection interface was not used in Map interface, and it was build in separate hierarchy.

HashMap

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key.

When we put an object into a HashMap, it uses the key’s **hash code** value to determine if an element is not in the map already.



Each hash code value corresponds to a certain bucket location which can contain various elements, for which the calculated hash value is the same. But two objects with the same hashCode might not be equal. So, objects within the same bucket will be compared using the equals() method.

Complexity	Put	Remove	Get	ContainsKey
HashMap	O(1)	O(1)	O(1)	O(1)

In the **worst case scenario**, all the items in the map have the same hash code and are therefore stored in the same bucket. In this case, you'll need to iterate over all of them serially, which would be an **O(n)** operation.

Since different keys can be mapped to the same index, there is a chance of collision. If the number of collisions is very high (In this case, you'll need to iterate over all of the keys serially), the **worst case runtime is O(N)**, where N is the number of keys. However, we generally assume a good implementation that keeps collisions to a minimum, in which case the lookup time is O(1).

A good key object for HashMap fits requirements:

- Maintain the **hashcode contract** (between hashCode() and equals())
- Make the key object **immutable**
- Implement hashCode() method so that least number of hash collisions occurs and entries are **evenly distributed** across all the buckets.

For example, String is a good HashMap key candidate because it is immutable → its hash code is **computed and cached** once hashcode() is called.

The performance of a HashMap is affected mainly by two parameters – its **Initial Capacity**(number of buckets) and the **Load Factor**:

- The load factor describes what is the maximum fill level, above which, a set will need to be resized.
- A low initial capacity reduces space complexity but increases the frequency of rehashing which is an expensive process. On the other hand, a high initial capacity increases the cost of iteration and the initial memory consumption.

Some properties of HashMap:

- Java HashMap allows null key and null values.
- HashMap is not an ordered collection. You can iterate over HashMap entries through keys set but they are not guaranteed to be in the order of their addition to the HashMap.
- HashMap uses it's inner class Node<K,V> for storing map entries.
- HashMap stores entries into multiple singly linked lists, called buckets or bins. Default number of bins is 16 and it's always power of 2.
- HashMap uses hashCode() and equals() methods on keys for get and put operations. So HashMap key object should provide good implementation of these methods. This is the reason immutable classes are better suitable for keys, for example String and Integer.
- Java HashMap is not thread safe, for multithreaded environment you should use ConcurrentHashMap class or get synchronized map using Collections.synchronizedMap() method.

Fixed HashCode issue

You can implement your hashCode() method such that you always return a fixed value, for example like this:

```
@Override
public int hashCode() {
  return 1;
}
```

The above method satisfies all the requirements and is considered legal according to the hash code contract but it would not be very efficient. If this method is used, **all objects will be stored in the same bucket** i.e. bucket 1 and when you try to ensure whether the specific object is present in the collection, then it will always have to check the entire content of the collection.

Array-as-key issue

In the example below we created two similar-looking arrays. 'a' and 'b' look equal but their hash codes differ because hash code for an array is derived from the memory address of the object in the heap (not a mandatory rule though). The .equals() by default checks if two Object references refer to the same Object (which is false in our example).

Behavior for both **hashcode and equals** methods in the array object is **not related to the array content**.

```
@Test
void arrayAsHashMapKeyTest() {
	HashMap<Object, Object> objects = new HashMap<>();

	int[] a = new int[] {1, 2, 3};
	int[] b = new int[] {1, 2, 3};

	System.out.println("'a' hashcode: " + a.hashCode());
	System.out.println("'b' hashcode: " + b.hashCode());
	System.out.println("a.equals(b) = " + a.equals(b));

	objects.put(a, new Object());
	assertTrue(objects.containsKey(a));
	assertFalse(objects.containsKey(b));

	a[2] = 4;
	System.out.println("'a' hashcode after modifying: " + a.hashCode());
}
```

Output:

```
'a' hashcode: 1722021981
'b' hashcode: 1019153279
a.equals(b) = false
'a' hashcode after modifying: 1722021981
```

LinkedHashMap

Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order). Note that insertion order is not affected if a key is re-inserted into the map.

Complexity	Put	Remove	Get	ContainsKey
LinkedHashMap	O(1)	O(1)	O(1)	O(1)

TreeMap

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a **Comparator** provided at map creation time, depending on which constructor is used.

Complexity	Add	Remove	Get	ContainsKey
TreeMap	O(logN)	O(logN)	O(logN)	O(logN)

For example we would want to store the collection of strings ordered by string length:

```
TreeMap<String, String> map = new TreeMap<>(Comparator.comparingInt(String::length));

map.put("aaa", "aaa");
map.put("b", "b");
map.put("cccc", "cccc");
map.put("dddd", "dddd");

map.forEach((key,value) -> System.out.println(key + ":" + value));
```

The output may seem a little bit surprising:

```
b:b
aaa:aaa
cccc:dddd
```

But how is that possible? The map contract states that while inserting if the map previously contained a mapping for the key, the old **value** is replaced. It says nothing about key replacement, assuming that the comparator respects the **(compare(x, y)==0) == (x.equals(y))** rule. In the example above however this rule has been violated: “dddd”.length() == “cccc”.length(), but !“dddd”.equals(“cccc”).

JavaDoc for Comparator says:

It is generally the case, but not strictly required that (compare(x, y)==0) == (x.equals(y)). Generally speaking, any comparator that violates this condition should clearly indicate this fact.

JavaDoc for TreeMap says:
Note that the ordering maintained by a tree map, like any sorted map, and whether or not an explicit comparator is provided, must be consistent with equals if this sorted map is to correctly implement the Map interface. (See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Map interface is defined in terms of the equals operation, but a sorted map performs all key comparisons using its compareTo (or compare) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a sorted map is well-defined even if its ordering is inconsistent with equals; **it just fails to obey the general contract of the Map interface**.

Some methods useful for accessing elements in TreeMap:

- firstKey()/firstEntry() - Returns the first Entry in the TreeMap (according to the TreeMap's key-sort function). Returns null if the TreeMap is empty.
- lastKey()/lastEntry() - Returns the last Entry in the TreeMap (according to the TreeMap's key-sort function). Returns null if the TreeMap is empty.

- ceilingKey(K key)/ceilingEntry(K key) - Gets the entry corresponding to the specified key; if no such entry exists, returns the entry for the least key greater than the specified key; if no such entry exists (i.e., the greatest key in the Tree is less than the specified key), returns null.
- floorKey(K key)/floorEntry(K key) - Gets the entry corresponding to the specified key; if no such entry exists, returns the entry for the greatest key less than the specified key; if no such entry exists, returns null.

- higherKey(K key)/higherEntry(K key) - Gets the entry for the least key greater than the specified key; if no such entry exists, returns the entry for the least key greater than the specified key; if no such entry exists returns null.
- lowerKey(K key)/lowerEntry(K key) - Returns the entry for the greatest key less than the specified key; if no such entry exists (i.e., the least key in the Tree is greater than the specified key), returns null.

WeakHashMap

WeakHashMap is an implementation of the Map interface that stores only weak references to its keys. Storing only weak references allows a key-value pair **to be garbage collected when its key is no longer referenced outside of the WeakHashMap**. This class is intended primarily for use with key objects whose equals methods test for object identity using the == operator. Once such a key is discarded it can never be recreated, so it is impossible to do a look-up of that key in a WeakHashMap at some later time and be surprised that its entry has been removed.

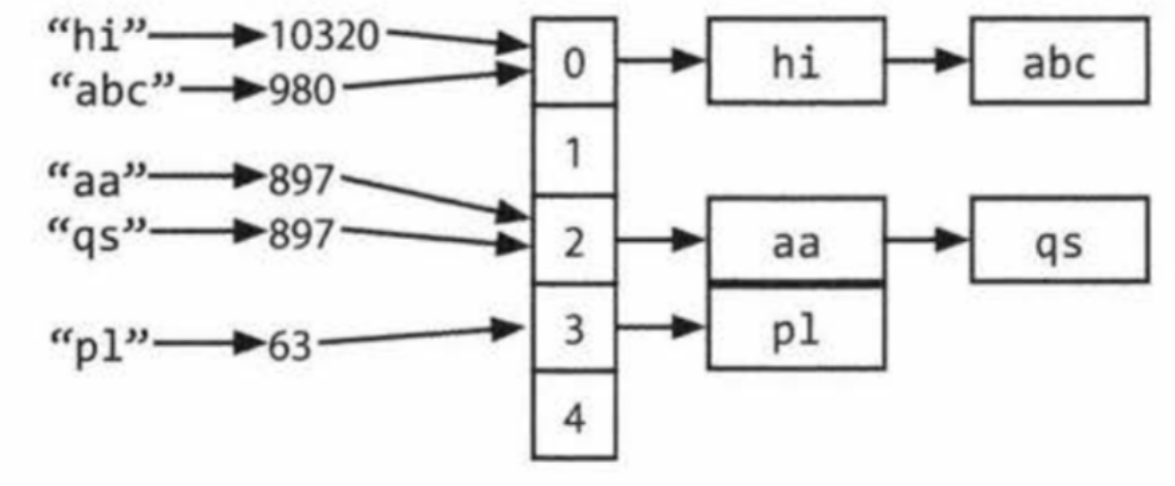
Set

A collection that contains no duplicate elements.

HashSet

The most important aspects of this implementation:

- It stores unique elements and permits nulls
- It's backed by a HashMap
- It doesn't maintain insertion order
- It's not thread-safe



Complexity	Add	Remove	Get	Contains
HashSet	O(1)	O(1)	O(1)	O(1)

In the **worst case scenario**, all the items in the set have the same hash code and are therefore stored in the same bucket. In this case, you'll need to iterate over all of them serially, which would be an **O(n)** operation.

In HashSet, the argument passed in add(Object) method serves as map key. Java internally associates dummy value for each value passed in add(Object) method:

```
// Dummy value to associate with an Object in the backing Map
private static final Object PRESENT = new Object();
...

public boolean add(E e) {
        return map.put(e, PRESENT)==null;
}
```

LinkedHashSet

Hash table and linked list implementation of the Set interface, with predictable iteration order. This implementation differs from HashSet in that **it maintains a doubly-linked list** running through all of its entries. This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (insertion-order). Note that insertion order is not affected if an element is re-inserted into the set. (An element e is reinserted into a set s if s.add(e) is invoked when s.contains(e) would return true immediately prior to the invocation.)

Complexity	Add	Remove	Get	Contains
LinkedHashSet	O(1)	O(1)	O(1)	O(1)

TreeSet

A NavigableSet implementation based on a TreeMap. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

Complexity	Add	Remove	Get	Contains
TreeSet	O(logN)	O(logN)	O(logN)	O(logN)

Queue

The **Queue** interface present in the java.util package and extends the Collection interface is used to hold the elements about to be processed in **FIFO**(First In First Out) order. It is an ordered list of objects with its use limited to insert elements at the end of the list and deleting elements from the start of the list, (i.e.), it follows the FIFO or the First-In-First-Out principle.

Being an interface the queue needs a concrete class for the declaration and the most common classes are the PriorityQueue, ArrayBlockingQueue, LinkedList in Java.

Characteristics of a Queue:

- The Queue is used to insert elements at the end of the queue and removes from the beginning of the queue. **It follows FIFO concept**.
- The Java Queue supports all methods of **Collection interface** including insertion, deletion etc.
- If any null operation is performed on BlockingQueues, NullPointerException is thrown.
- The Queues which are available in java.util package are Unbounded Queues.
- The Queues which are available in java.util.concurrent package are the Bounded Queues.
- All Queues except the Deques supports insertion and removal at the tail and head of the queue respectively. The Deques support element insertion and removal at both ends.

BlockingQueue

The Java BlockingQueue interface, java.util.concurrent.BlockingQueue, represents a queue which is **thread safe** to put elements into, and take elements out of from. In other words, multiple threads can be inserting and taking elements concurrently from a Java BlockingQueue, without any concurrency issues arising. Blocking Queue 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.

lockingQueue methods come in four forms, with **different ways of handling operations that cannot be satisfied immediately**, but may be satisfied at some point in the future: one throws an exception, the second returns a special value (either null or false, depending on the operation), the third blocks the current thread indefinitely until the operation can succeed, and the fourth blocks for only a given maximum time limit before giving up. These methods are summarized in the following table:

	Throws exception	Special value	Blocks	Times out
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	not applicable	not applicable

A BlockingQueue **does not accept null elements**. Implementations throw NullPointerException on attempts to add, put or offer a null. A null is used as a sentinel value to indicate failure of poll operations.

PriorityBlockingQueue

An unbounded blocking **priority** queue based on a priority heap. The elements of the priority queue are **ordered** according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used. A priority queue **does not permit null elements**. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in ClassCastException).

SynchronousQueue

A blocking queue in which **each insert operation must wait for a corresponding remove operation by another thread**, and vice versa. A synchronous queue does not have any internal capacity, not even a capacity of one. You **cannot peek** at a synchronous queue because an element is only present when you try to remove it; you **cannot insert an element (using any method) unless another thread is trying to remove it**; you **cannot iterate** as there is nothing to iterate. The head of the queue is the element that the first queued inserting thread is trying to add to the queue; if there is no such queued thread then no element is available for removal and poll() will return null. For purposes of other Collection methods (for example contains), a SynchronousQueue acts as an empty collection. **This queue does not permit null elements**.

Synchronous queues are well suited for **handoff designs**, in which an object running in one thread must sync up with an object running in another thread in order to hand it some information, event, or task.

ArrayBlockingQueue

A bounded blocking queue backed by an array. This queue orders elements FIFO (first-in-first-out). The head of the queue is that element that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

This is a classic "**bounded buffer**", in which a fixed-sized array holds elements inserted by producers and extracted by consumers. Once created, the **capacity cannot be changed**. Attempts to put an element into a full queue will result in the operation blocking; attempts to take an element from an empty queue will similarly block.

LinkedBlockingQueue

An optionally-bounded blocking queue based on **linked nodes**. This queue orders elements FIFO (first-in-first-out). The head of the queue is that element that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue. **Linked queues typically have higher throughput than array-based queues but less predictable performance in most concurrent applications.**

Thread-safe collections

If multiple threads access an unsynchronized collection concurrently, and at least one of the threads modifies the collection structurally, it must be synchronized externally.

Synchronized collections

Synchronized collections achieve thread-safety through intrinsic locking, and the entire collections are locked. Intrinsic locking is implemented via synchronized blocks within the wrapped collection's methods.

As we might expect, synchronized collections assure data consistency/integrity in multi-threaded environments. However, they might come with a penalty in performance, as only one single thread can access the collection at a time (a.k.a. synchronized access).

- Synchronization at Object level.
- Every read/write operation needs to acquire lock.
- **Locking the entire collection** is a performance overhead.
- This essentially gives access to only one thread to the entire map & blocks all the other threads.

```
Map<Object, Object> syncHashMap = Collections.synchronizedMap(new HashMap<>());
Map<Object, Object> syncLinkedMap = Collections.synchronizedMap(new LinkedHashMap<>());
SortedMap<Object, Object> syncSortedMap = Collections.synchronizedSortedMap(new
TreeMap<>());

Set<Object> syncHashSet = Collections.synchronizedSet(new HashSet<>());
Set<Object> syncLinkedHashSet = Collections.synchronizedSet(new LinkedHashSet<>());
SortedSet<Object> syncSortedSet = Collections.synchronizedSortedSet(new TreeSet<>());

List<Object> syncList = Collections.synchronizedList(new ArrayList<>());
```

Concurrent collections

Concurrent collections (e.g. ConcurrentHashMap), achieve thread-safety by dividing their data into segments. In a ConcurrentHashMap, for example, different threads can acquire locks on each segment, so multiple threads can access the Map at the same time (a.k.a. concurrent access). On the other hand, Collections.synchronized...() will lock all the data while updating, other threads can only access the data when the lock is released. **Concurrent collections are much more performant than synchronized collections**, due to the inherent advantages of concurrent thread access. **If there are many update operations and relative small amount of read operations, you should choose ConcurrentHashMap**.

Also one other difference is that ConcurrentHashMap will not preserve the order of elements in the Map passed in. It is similar to HashMap when storing data. There is no guarantee that the element order is preserved. For example, if you pass a TreeMap to ConcurrentHashMap, the elements order in the ConcurrentHashMap may not be the same as the order in the TreeMap, but Collections.synchronizedMap() will preserve the order.

- You should use ConcurrentHashMap when you need very high concurrency in your project.
- It is thread safe without synchronizing the whole map.
- Reads can happen very fast while write is done with a lock.
- There is no locking at the object level.
- The locking is at a much finer granularity at a hashmap bucket level.
- ConcurrentHashMap doesn't throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it.
- ConcurrentHashMap uses multitude of locks.

```
ConcurrentHashMap<Object, Object> concHashMap = new ConcurrentHashMap<>();
```

There's no built in type for ConcurrentHashSet because you can always derive a set from a map. Since there are many types of maps, you use a method to produce a set from a given map (or map class):

```
Set<Object> concHashSet = Collections.newSetFromMap(new ConcurrentHashMap<>());
//or using Guava (effectively the same):
Set<Object> concHashSet2 = Sets.newConcurrentHashSet();
```

CopyOnWriteArray

CopyOnWriteArrayList

A thread-safe variant of ArrayList in which all mutative operations (add, set, and so on) are implemented by **making a fresh copy of the underlying array**.

This is ordinarily too costly, but may be more efficient than alternatives when traversal operations vastly outnumber mutations, and is useful when you cannot or don't want to synchronize traversals, yet need to preclude interference among concurrent threads. The **"snapshot" style iterator** method uses a reference to the state of the array at the point that the iterator was created. **This array never changes during the lifetime of the iterator**, so interference is impossible and the iterator is guaranteed not to throw ConcurrentModificationException. **The iterator will not reflect additions, removals, or changes to the list since the iterator was created**. Element-changing operations on iterators themselves (remove, set, and add) are not supported. These methods throw UnsupportedOperationException.

CopyOnWriteArraySet

A Set that uses an internal CopyOnWriteArrayList for all of its operations. Thus, it shares the same basic properties:

- It is best suited for applications in which set sizes generally stay small, read-only operations vastly outnumber mutative operations, and you need to prevent interference among threads during traversal.
- It is thread-safe.
- **Mutative operations (add, set, remove, etc.) are expensive since they usually entail copying the entire underlying array.**
- Iterators do not support the mutative remove operation.
- Traversal via iterators is fast and cannot encounter interference from other threads. Iterators rely on unchanging snapshots of the array at the time the iterators were constructed.

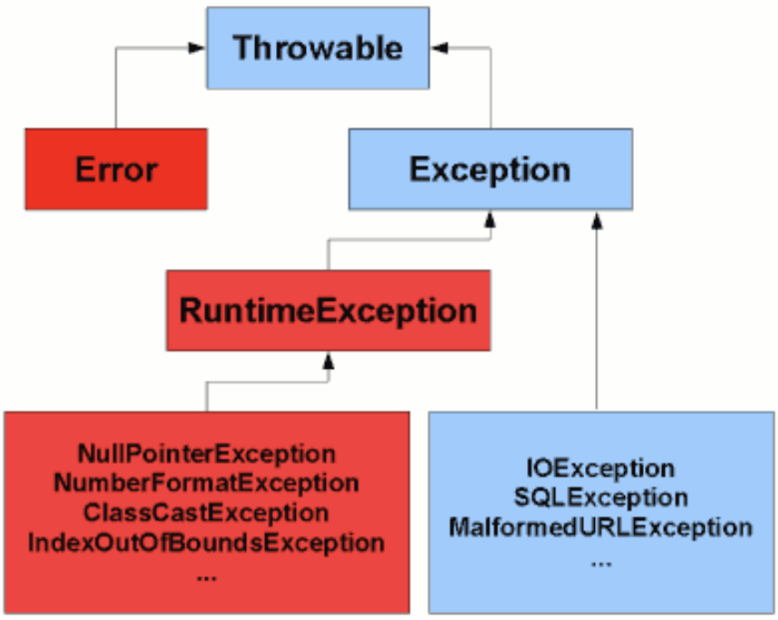
Exceptions

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

In Java, an exception is an object that wraps an error event that occurred within a method and contains:

- Information about the error including its type
- The state of the program when the error occurred
- Optionally, other custom information

Exception objects can be **thrown** and **caught**.



Checked vs Unchecked

These are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.

Unchecked are the exceptions that are not checked at compile time.

JavaDoc recommendation: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

Try-with-resources

The try-with-resources statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

Generics

Generic (parametrized) types are implicitly invariant in Java, meaning that different instantiations of a generic type are not compatible among each other.

```
Generic<SuperType> superGeneric;
Generic<SubType> subGeneric;
subGeneric = (Generic<SubType>)superGeneric; // type error
superGeneric = (Generic<SuperType>)subGeneric; // type error
```

Java Generics were introduced in JDK 5.0 with the aim of reducing bugs and adding an extra layer of abstraction over types.

Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define **Generic Methods**:

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type.
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Following example illustrates how we can print an array of different type using a single Generic method:

```
@Test
   void genericsTest() {

      printAll(new String[] {"a", "b", "c", "d"});
```

```
        printAll(new Integer[] {1, 2, 3, 4});
        printAll(new Object[] {new Object(){
            @Override
            public String toString() {
                return "custom object";
            }
        }});
    }

    public <X> void printAll(X[] elements) {
        for (X el : elements) {
            System.out.printf("%s", el);
        }
        System.out.println();
    }
```

Bounded Type Parameters

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses.

We can specify that a method accepts a type and all its subclasses (upper bound) or a type all its superclasses (lower bound).

To declare a bounded type parameter, list the type parameter's name, followed by the **extends** keyword, followed by its upper bound.

```
public <X extends Comparable<X>> X maximum(X x, X y) {
        if (x.compareTo(y) > 0) return x;
        else return y;
    }
```

Generic Classes

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

```
class Container<T> {
        private T t;
```

```
    public Container(T t) {
        this.t = t;
    }

    public T getT() {
        return t;
    }
}
```

Type Erasure

Generics were added to Java to ensure type safety and to ensure that generics wouldn't cause overhead at runtime, **the compiler applies a process called type erasure on generics at compile time**.

Type erasure removes all type parameters and replaces it with their bounds or with Object if the type parameter is unbounded. Thus the bytecode after compilation contains only normal classes, interfaces and methods thus ensuring that no new types are produced. Proper casting is applied as well to the Object type at compile time.

This is an example of type erasure:

```
public <T> List<T> genericMethod(List<T> list) {
    return list.stream().collect(Collectors.toList());
}
```

With type erasure, the unbounded type T is replaced with Object as follows:

```
// for illustration
public List<Object> withErasure(List<Object> list) {
    return list.stream().collect(Collectors.toList());
}

// which in practice results in
public List withErasure(List list) {
    return list.stream().collect(Collectors.toList());
}
```

If the type is bounded, then the type will be replaced by the bound at compile time:

```
public <T extends Building> void genericMethod(T t) {
    ...
}
```

would change after compilation:

```
public void genericMethod(Building t) {
    ...
}
```

Thus, the two method declarations must not occur together in an interface (or abstract class) definition:

```
void method(Generic<SuperType> p);
void method(Generic<SubType> p);
```

Wildcards

It is known that Object is the supertype of all Java classes, however, a collection of Object is not the supertype of any collection.

For example, a List<Object> is not the supertype of List<String> and assigning a variable of type List<Object> to a variable of type List<String> will cause a compiler error. This is to prevent possible conflicts that can happen if we add heterogeneous types to the same collection.

The Same rule applies to any collection of a type and its subtypes. Consider this example:

```
public static void paintAllBuildings(List<Building> buildings) {
    buildings.forEach(Building::paint);
}
```

if we imagine a subtype of Building, for example, a House, we can't use this method with a list of House, even though House is a subtype of Building. If we need to use this method with type Building and all its subtypes, then the bounded wildcard can do the magic:

```
public static void paintAllBuildings(List<? extends Building> buildings) {
    ...
}
```

Now, this method will work with type Building and all its subtypes. This is called an upper bounded wildcard where type Building is the upper bound.

Wildcards can also be specified with a lower bound, where the unknown type has to be a supertype of the specified type. Lower bounds can be specified using the super keyword followed by the specific type, for example, <? super T> means unknown type that is a superclass of T (= T and all its parents).

Wildcard Guidelines

https://docs.oracle.com/javase/tutorial/java/generics/wildcardGuidelines.html

- An "in" variable is defined with an upper bounded wildcard, using the extends keyword.
- An "out" variable is defined with a lower bounded wildcard, using the super keyword.
- In the case where the "in" variable can be accessed using methods defined in the Object class, use an unbounded wildcard.
- In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.

QA

Unbound wildcards <?> vs <Object> : The difference is List<?> will take any List with whatever declaration, but List<Object> will only take something that was declared as List<Object>, nothing else.

The last quote simply states, that List<?> is a list for which you literally don't know what type its items are. Because of that, you can not add anything to it other than null.

bounded wildcards or bounded type parameter?

```
instead of creating a method like this:
public void drawAll(List<? extends Shape> shapes){
    for (Shape s: shapes) {
        s.draw(this);
    }
}
```

This works fine too:

```
public <T extends Shape> void drawAll(List<T> shapes){
    for (Shape s: shapes) {
        s.draw(this);
    }
}
```

Which way should I use? Is wildcard useful in this case?

It depends on what you need to do. You need to use the bounded type parameter if you wanted to do something like this:

```
public <T extends Shape> void addIfPretty(List<T> shapes, T shape) {
    if (shape.isPretty()) {
       shapes.add(shape);
    }
}
```

Here we have a List<T> shapes and a T shape, therefore we can safely shapes.add(shape). If it was declared List<? extends Shape>, you can NOT safely add to it (**because you may have a List<Square> and a Circle**).

So by giving a name to a bounded type parameter, we have the option to use it elsewhere in our generic method. This information is not always required, of course, so if you don't need to know that much about the type (e.g. your drawAll), then just wildcard is sufficient.

```
public <T extends Comparable<T>> T wildcardMethod(List<T> c) {
        T max = c.get(0);   //we can use the type information
```

```
    for (T t : c) {
        if (t.compareTo(max) > 0) max = t;
    }
    return max;
}
```

Using wildcards:

```
public void wildcardMethod2(List<? extends Comparable<?>> in) {
    //no access to type

    Comparable<?> comparable = in.get(0);

    //does not compile
    //comparable.compareTo(in.get(1));
}
```

Even if you're not referring to the bounded type parameter again, a **bounded type parameter is still required if you have multiple bounds**.

A wildcard can have only one bound, while a type parameter can have several bounds. A wildcard can have a lower or an upper bound, while there is no such thing as a lower bound for a type parameter.

Wildcard bounds and type parameter bounds are often confused, because they are both called bounds and have in part similar syntax.

```
type parameter bound     T extends Class & Interface1 & … & InterfaceN

wildcard bound
   upper bound          ? extends SuperType
   lower bound          ? super   SubType
```

A wildcard can have only one bound, either a lower or an upper bound. A list of wildcard bounds is not permitted. A type parameter, in constrast, can have several bounds, but there is no such thing as a lower bound for a type parameter.

Example from Collections utils:

```
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
```

Explanation: Any subclass of T (including T) can be safely casted to any class Object...T (including T).

Summary:

- Do use bounded type parameters/wildcards, they increase flexibility of your API
- If the type requires several parameters, you have no choice but to use bounded type parameter
- if the type requires a lowerbound, you have no choice but to use bounded wildcard
- "Producers" have upperbounds, "consumers" have lowerbounds (PECS)
- Do not use wildcard in return types

There are many situations where you simply don't care what type you are referring to. In those cases, you may use ? without cluttering code with unused type parameter declarations.

By using a wildcard, you're explicitely saying: I don't care about the concrete type of the elements of the list, and what I do doesn't depend on this type. **If the method had to return a T, or pass the T to some other generic type, then you would use a typed method**.

Note that if the ? had the same semantics ("a List of objects of unknown type") but a different purpose ("the List would consume objects of unknown type") things would change very, very radically such that it may be inadvisable (or very difficult) to use a wildcard parameter (at least without a helper method to capture the object's type).

The ? states that the generic code does not require any reference to a type. This could be a class where the properties are independent of any type like the length of a collection.

Another reason for the ? is that it provides syntax for less ambiguity for the case when the generic code only needs behavior provided by class Object. As pointed out above, the parameter type <T> would work, but the fact that T was defined and then never used could suggest that the developer left out something.

So if <T> is dropped because of semantic ambiguity and <?> is not available, then the developer would be tempted to write <Object>. This however does not allow the generic code to be used for any type, just the Object type.

So, <?> is more precise. The <?> very clearly self-documents that the type is intentionally not used elsewhere.

Type parameter method is more powerful than the wildcard method: You give the parameter a name which you can reference. For example, consider a method that removes the first element of a list and adds it to the back of the list. With generic parameters, we can do the following:

```
static <T> boolean rotateOneElement(List<T> l){
    return l.add(l.remove(0));
}
```

With a wildcard, this is not possible since l.remove(0) would return capture-1-of-?, but l.add would require capture-2-of-?. I.e., the compiler is not able to deduce that the result of remove is the same type that add expects. This is contrary to the first example where the compiler can deduce that both is the same type T. This code would not compile:

```
static boolean rotateOneElement(List<?> l){
    return l.add(l.remove(0)); //ERROR!
}
```

So, what can you do if you want to have a rotateOneElement method with a wildcard, since it is easier to use than the generic solution? The answer is simple: Let the wildcard method call the generic one, then it works:

```
private static <T> boolean rotateOneElementImpl(List<T> l){   // Private implementation
    return l.add(l.remove(0));
}

static void rotateOneElement(List<?> l){  //Public interface
     rotateOneElementImpl(l);
}
```

The standard library uses this trick in a number of places. One of them is, IIRC, Collections.java

When you are declaring a type parameter, and using it only once, it essentially becomes a wildcard parameter. On the other hand, if you use it more than once, the difference becomes significant. e.g.

```
<E> void printObjectsExceptOne(List<E> list, E object) {
```

You might see that this case **enforces both types to be the same**.

As a result, if you are going to use a type parameter only once, it does not even make sense to name it. That is why java architects invented so called wildcard arguments (most probably).

Wildcard parameters avoid unnecessary code bloat and make code more readable. If you need two, you have to fall back to regular syntax for type parameters.

```
ArrayList<? extends CharSequence> cs = new ArrayList<>();
        //this works as any subclass of CharSequence can be safely upcasted to CharSequence (type
producing)
        CharSequence charSequence = cs.get(0);
        //below line won't compile, as 'cs' may be of any type other than String (type consuming)
        //cs.add("string");

        ArrayList<? super String> strs = new ArrayList<>();
        //this works as String can be safely upcasted to any type Object..String (type consuming)
        strs.add("234");
        //Object type available here (without explicit casting) because this list can be of any type between
Object..String
        Object s = strs.get(0);
```

Functional Programming

The biggest advantage of adopting functional programming in any language, including Java, is **pure functions and immutable states**. If we think in retrospect, most of the programming challenges are rooted in the side-effects and mutable state one way or the other. Simply getting rid of them makes our program easier to read, reason about, test, and maintain.

A **functional interface** in Java is an interface that only has one abstract method. By an abstract method is meant only one method which is not implemented. An interface can have multiple methods, e.g. default methods and static methods, both with implementations, but as long as the interface only has one method that is not implemented, the interface is considered a functional interface.

Some examples of functional interfaces:

Interface name	**Method**
Comparable	public int compareTo(T o)
Predicate	boolean test(T t);
Function	R apply(T t);
BiFunction	R apply(T t, U u);
UnaryOperator	same as function but T = R
BinaryOperator	same as bifunction but T = U = R
Consumer	void accept(T t);
Supplier	T get();
Runnable	public abstract void run();
Callable	V call() throws Exception;

Java Stream API

To perform a sequence of operations over the elements of the data source and aggregate their results, we need three parts: the **source**, **intermediate operation**(s) and a **terminal operation**.

We can instantiate a stream, and have an accessible reference to it, as long as only intermediate operations are called.

Stream creation

There are many ways to create a stream instance of different sources. Once created, the instance **will not modify its source**, therefore allowing the creation of multiple instances from a single source.

```
Stream<String> streamEmpty = Stream.empty();

Collection<String> collection = Arrays.asList("a", "b", "c");
Stream<String> streamOfCollection = collection.stream();

Stream<String> streamOfArray = Stream.of("a", "b", "c");

String[] arr = new String[]{"a", "b", "c"};
Stream<String> streamOfArrayFull = Arrays.stream(arr);

Stream<String> streamBuilder =
      Stream.<String>builder().add("a").add("b").add("c").build();

Stream<String> streamGenerated =
      Stream.generate(() -> "element").limit(10);

Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20);

IntStream intStream = IntStream.range(1, 3);
LongStream longStream = LongStream.rangeClosed(1, 3);
```

Stream transformation

Intermediate operations return a new modified stream.

Intermediate operations are lazy. This means that they will be invoked only if it is necessary for the terminal operation execution.

Intermediate operations which reduce the size of the stream should be placed before operations which are applying to each element. So we need to keep methods such as skip(), filter(), and distinct() at the top of our stream pipeline.

```
String result = IntStream.range(1, 100) //stream creation
        .filter(i -> i % 2 == 0)            //intermediate operation
        .mapToObj(String::valueOf)   //intermediate operation
        .collect(Collectors.joining());  //stream termination
```

Here is the list of intermediate operations:

- map(..)
- flatMap(..)
- sorted(..)
- peek(..)
- limit(..)
- skip(..)
- distinct()
- filter(..)

Another example:

```
Set<Integer> books = users.stream()
        .filter(User::isActive)
        .filter(u -> u.getAge() > 25)
        .map(User::getBooks)
        .filter(CollectionUtils::isNotEmpty)
        .flatMap(Collection::stream)
        .map(User.Book::getId)
        .collect(Collectors.toUnmodifiableSet());
```

Stream termination

A stream by itself is worthless; the user is interested in the result of the terminal operation, which can be a **value** of some type or an **action** applied to every element of the stream.

We can only use one terminal operation per stream: executing a terminal operation makes a stream inaccessible. An attempt to reuse the same reference after calling the terminal operation will trigger the IllegalStateException.

The API has many terminal operations which aggregate a stream to a type or to a primitive: **count**(), **max**(), **min**(), and **sum**(). However, these operations work according to the predefined implementation. So what if a developer needs to customize a Stream's reduction mechanism? There are two methods which allow us to do this, the **reduce**() and the **collect**() methods.

Reduce

Applying some scalar function to reduce all the elements to a single result.

```
<U> U reduce(U identity,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner);
```

- identity - initial value for the combiner function
- accumulator - function for incorporating an additional element into a result
- combiner - function for combining values produced by a combiner

Given a stream of type Integer, reduce its elements to a result of type String:

```
String joined = IntStream.range(0, 100).boxed()
            .reduce("", (res, i) -> res + i.toString(), (a, b) -> a + b);
```

Collect

Stream.collect(...) performs a mutable reduction operation on the elements of this stream:

```
<R> R collect(Supplier<R> supplier,
          BiConsumer<R, ? super T> accumulator,
          BiConsumer<R, R> combiner);
```

- supplier - function for combining values produced by a combiner
- accumulator - function for adding additional element to collection
- combiner - function for merging intermediate collections

Example of collecting stream elements to ArrayList:

```
List<Integer> collected = IntStream.range(0, 100).boxed().collect(
```

```
        () -> new ArrayList<>(),
        (list, el) -> list.add(el),
        (list1, list2) -> list1.addAll(list2));
```

Some predefined collectors:

- **toList()** :There are no guarantees on the type, mutability, serializability, or thread-safety of the List returned.
- **toMap** (keyMapper, valueMapper, mergeFunction, mapSupplier): If mapSupplier is not specified, the HashMap is used. For parallel stream pipelines, the combiner function operates by merging the keys from one map into another, which can be an expensive operation. If it is not required that results are inserted into the Map in encounter order, using **toConcurrentMap**(Function, Function) may offer better parallel performance.
- **toSet** (There are no guarantees on the type, mutability, serializability, or thread-safety of the Set returned)
- **toCollection** (Supplier<C> collectionFactory): Returns a Collector that accumulates the input elements into a new Collection, in encounter order.
- **mapping** (Function<? super T, ? extends U> **mapper**, Collector<? super U, A, R> **downstream**) - adapts a Collector accepting elements of type U to one accepting elements of type T by applying a mapping function to each input element before accumulation:

  ```
  return new CollectorImpl<>(downstream.supplier(),
                             (r, t) -> downstreamAccumulator.accept(r, mapper.apply(t)),
                             downstream.combiner(), downstream.finisher(),
                             downstream.characteristics());
  ```

  Example:

  ```
  Map<Integer, Set<String>> collectedToStringSet = IntStream.range(0, 100).boxed().collect(
          Collectors.groupingBy(i -> i % 3,
                  Collectors.mapping(String::valueOf, Collectors.toSet())));
  ```

- **partitioningBy** (Predicate<? super T> predicate, Collector<? super T, A, D> downstream): Returns a Collector which partitions the input elements according to a Predicate, reduces the values in each partition according to another Collector, and organizes them into a **Map<Boolean, D>** whose values are the result of the downstream reduction.

  Example:

  ```
  Map<Boolean, List<Integer>> collect = IntStream.range(0, 100)
          .boxed()
          .collect(Collectors.partitioningBy(i -> i % 2 == 0));
  ```

- **groupingBy** (Function<? super T, ? extends K> classifier,Supplier<M> mapFactory, Collector<? super T, A, D> downstream): Returns a Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector.

- <T> – the type of the input elements
- <K> – the type of the keys
- <D> – the result type of the downstream reduction
- <A> – the intermediate accumulation type of the downstream collector
- <M> – the type of the resulting Map

Example:

```
Map<City, Set<String>> namesByCity = people.stream().collect(groupingBy(
                    Person::getCity,
                    TreeMap::new,
                    mapping(Person::getLastName, toSet())));
```

- **joining** (CharSequence delimiter, CharSequence prefix, CharSequence suffix) - returns a Collector that concatenates the input elements, separated by the specified delimiter, with the specified prefix and suffix, in encounter order.

```
Stream.of("A", "B", "C")
            .collect(Collectors.joining("," ,"prefix","suffix"));
//prefixA,B,Csuffix
```

There are following collectors producing unmodifiable collections (by simply wrapping the resulting collection in ImmutableCollections implementation (List.of(..),Set.of(..),Map.of(..)) :

- toUnmodifiableList()
- toUnmodifiableSet()
- toUnmodifiableMap(keyMapper, valueMapper, mergeFunction)

Parallel Stream

Under the hood, Stream API automatically uses the **ForkJoin** framework to execute operations in parallel. By default, the **common thread pool** will be used and there is no way (at least for now) to assign some custom thread pool to it. This can be overcome by using a custom set of parallel collectors.

When using streams in parallel mode, avoid blocking operations. It is also best to use parallel mode when tasks need a similar amount of time to execute. If one task lasts much longer than the other, it can slow down the complete app's workflow.

```
Stream<Product> streamOfCollection = productList.parallelStream();
boolean isParallel = streamOfCollection.isParallel();
boolean bigPrice = streamOfCollection
  .map(product -> product.getPrice() * 12)
  .anyMatch(price -> price > 200);
```

Optional

Java SE 8 introduces a new class called java.util.Optional<T> that is inspired from the ideas of Haskell and Scala. The purpose of the class is to provide a type-level solution for representing optional values instead of null references.

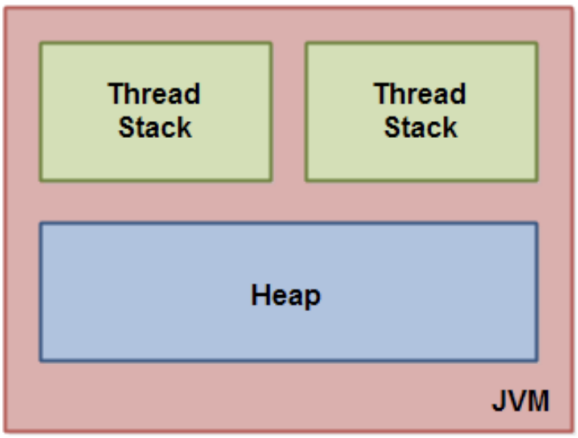
Some examples of using Optional:

```
Optional<String> notNullOptional = Optional.of("im not null");
Assertions.assertTrue(notNullOptional.isPresent());

Optional<Object> nullPassedOptional = Optional.ofNullable(null);
Assertions.assertTrue(nullPassedOptional.isEmpty());

Optional<Object> emptyOptional = Optional.empty();
Assertions.assertTrue(nullPassedOptional.isEmpty());

notNullOptional.map(s -> s.substring(0, 2))
        .ifPresent(System.out::println);

Object orElse = emptyOptional.orElse("default");
Object orElseGet = emptyOptional.orElseGet(this::loadFromSomewhere);

String value = notNullOptional
        .filter(s -> s.contains("im"))
        .map(s -> s.substring(0, 5))
        .orElseThrow(() -> new RuntimeException("value with 'im' is not present!"));
```

Concurrency

Java Memory Model

The Java memory model used internally in the JVM divides memory between thread stacks and the heap.

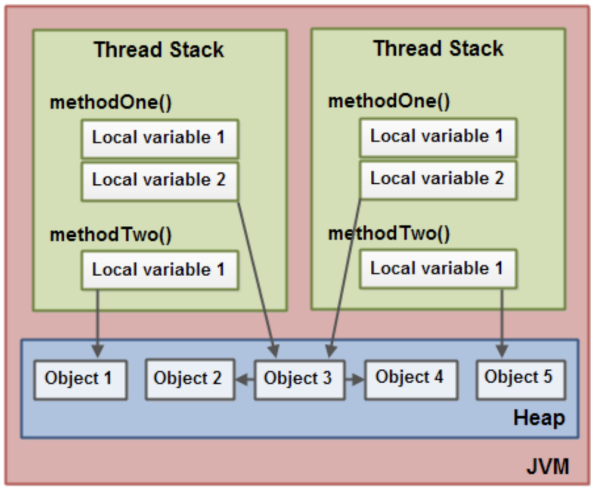


Each thread running in the Java virtual machine has its own thread stack. **The thread stack contains information about what methods the thread has called to reach the current point of execution**. I will refer to this as the "**call stack**". As the thread executes its code, the call stack changes.

The thread stack also contains all local variables for each method being executed (all methods on the call stack). A thread can only access it's own thread stack. Local variables created by a thread are invisible to all other threads than the thread who created it. Even if two threads are executing the exact same code, the two threads will still create the local variables of that code in each their own thread stack. Thus, **each thread has its own version of each local variable**.

All local variables of primitive types (boolean, byte, short, char, int, long, float, double) are fully stored on the thread stack and are thus not visible to other threads. One thread may pass a copy of a primitive variable to another thread, but it cannot share the primitive local variable itself.

The heap contains all objects created in your Java application, regardless of what thread created the object. This includes the object versions of the primitive types (e.g. Byte, Integer, Long etc.). It does not matter if an object was created and assigned to a local variable, or created as a member variable of another object, the object is still stored on the heap.



An **object's member variables are stored on the heap along with the object itself**. That is true both when the member variable is of a primitive type, and if it is a reference to an object.

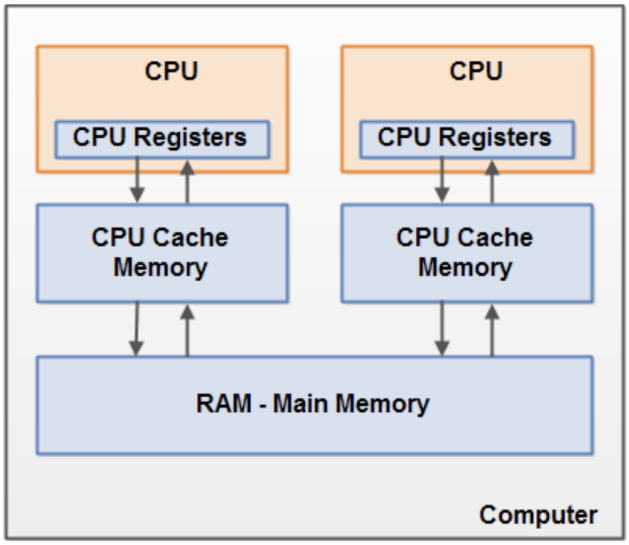
Static class variables are also stored on the heap along with the class definition.

Objects on the heap can be accessed by all threads that have a reference to the object. When a thread has access to an object, it can also get access to that object's member variables. If two threads call a method on the same object at the same time, they will

both have access to the object's member variables, but each thread will have its own copy of the local variables.

Hardware Memory Architecture

Each CPU contains a set of registers which are essentially in-CPU memory. The CPU can perform operations much faster on these registers than it can perform on variables in main memory. That is because the CPU can access these registers much faster than it can access main memory.

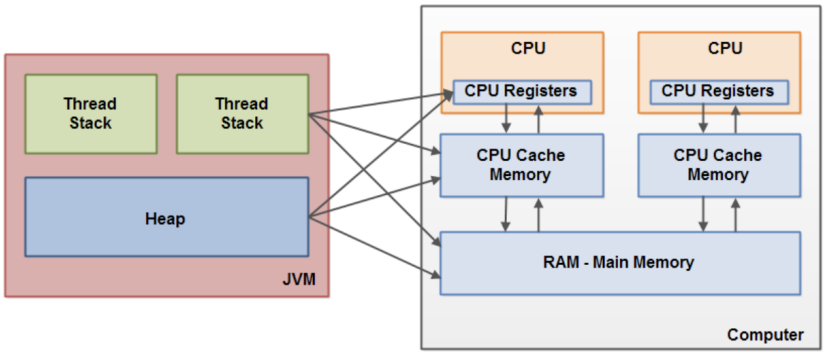


Each CPU may also have a CPU cache memory layer. In fact, most modern CPUs have a cache memory layer of some size. The **CPU can access its cache memory much faster than main memory**, but typically not as fast as it can access its internal registers. So, the CPU cache memory is somewhere in between the speed of the internal registers and main memory. Some CPUs may have multiple cache layers (Level 1 and Level 2), but this is not so important to know to understand how the Java memory model interacts with memory. What matters is to know that CPUs can have a cache memory layer of some sort.

Typically, when a CPU needs to access main memory it will **read part of main memory into its CPU cache**. It may even read part of the cache into its internal registers and then perform operations on it. When the CPU needs to write the result back to main memory it will flush the value from its internal register to the cache memory, and at some point **flush the value back to main memory**.

The values stored in the cache memory is typically flushed back to main memory when the CPU needs to store something else in the cache memory. The CPU cache can have data written to part of its memory at a time, and flush part of its memory at a time. **It does not have to read / write the full cache each time it is updated**. Typically the cache is updated in smaller memory blocks called "cache lines". One or more cache lines may be read into the cache memory, and one or more cache lines may be flushed back to main memory again.

The hardware memory architecture does not distinguish between thread stacks and heap. On the hardware, both the thread stack and the heap are located in main memory. Parts of the thread stacks and heap may sometimes be present in CPU caches and in internal CPU registers.



When objects and variables can be stored in various different memory areas in the computer, certain problems may occur. The two main problems are:

- Visibility of thread updates (writes) to shared variables (solution = volatile/synchronized guarantees)
- Race conditions when reading, checking and writing shared variables (solution = atomicity by means of synchronized/locks/CAS)

http://tutorials.jenkov.com/java-concurrency/java-memory-model.html

Thread

A **thread** is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a **single non-daemon thread** (which typically calls the method named main of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The **exit method of class Runtime has been called** and the security manager has permitted the exit operation to take place.
- **All threads that are not daemon threads have died**, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread. This subclass should override the run method of class Thread. An instance of the subclass can then be allocated and started. The other way to create a thread is to declare a class that implements the Runnable interface. That class then implements the run method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started.

Every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it.

Visibility and Happens-Before

https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5

Instruction reordering poses some challenges in a multithreaded, multi CPU system.

Example:

```
public class FrameExchanger  {

    private long framesStoredCount = 0:
    private long framesTakenCount  = 0;

    private volatile boolean hasNewFrame = false;

    private Frame frame = null;

        // called by Frame producing thread
    public void storeFrame(Frame frame) {
        this.frame = frame;
        this.framesStoredCount++;
        this.hasNewFrame = true;
    }

        // called by Frame drawing thread
    public Frame takeFrame() {
        while( !hasNewFrame) {
            //busy wait until new frame arrives
        }

        Frame newFrame = this.frame;
        this.framesTakenCount++;
        this.hasNewFrame = false;
        return newFrame;
    }
}
```

Now, when the hasNewFrame variable is set to true, the frame and frameStoredCount will also be synchronized to main memory. Additionally, every time the drawing thread reads the hasNewFrame variable in the while-loop inside the takeFrame() method, the frame and framesStoredCount will also be refreshed from main memory. Even framesTakenCount will get updated from main memory at this point.

Imagine if the Java VM reordered the instructions inside the storeFrame() method, like this:

```
    // called by Frame producing thread
    public void storeFrame(Frame frame) {
        this.hasNewFrame = true;
        this.framesStoredCount++;
```

```
    this.frame = frame;
}
```

Now the framesStoredCount and frame fields will get synchronized to main memory when the first instruction is executed (because hasNewFrame is volatile) - which is before they have their new values assigned to them!

This means, that the drawing thread executing the takeFrame() method may exit the while-loop before the new value is assigned to the frame variable. Even if a new value had been assigned to the frame variable by the producing thread, there would not be any guarantee that this value would have been synchronized to main memory so it is visible for the drawing thread!

As you can see, the reordering of the instructions inside storeFrame() method may make the application malfunction. This is where the volatile write happens before guarantee comes in - to put restrictions on what kind of instruction reordering is allowed around writes to volatile variables.

Very good explanation of the problem:
http://tutorials.jenkov.com/java-concurrency/java-happens-before-guarantee.html

volatile keyword

Java Volatile Happens Before Guarantee

A write to a non-volatile or volatile variable that happens before a write to a volatile variable is guaranteed to happen before the write to that volatile variable.

A read of a volatile variable will happen before any subsequent reads of volatile and non-volatile variables.

Java volatile Visibility Guarantee

The Java volatile keyword provides some visibility guarantees for when writes to, and reads of, volatile variables result in synchronization of the variable's value to and from main memory. This synchronization to and from main memory is what makes the value visible to other threads. Hence the term visibility guarantee.

When you write to a Java volatile variable the value is guaranteed to be written directly to main memory. Additionally, all variables visible to the thread writing to the volatile variable will also get synchronized to main memory.

```
this.nonVolatileVarA = 34;
this.nonVolatileVarB = new String("Text");
this.volatileVarC    = 300;
```

When the third instruction in the example above writes to the volatile variable volatileVarC, the values of the two non-volatile variables will also be synchronized to main memory - because these variables are visible to the thread when writing to the volatile variable.

When you read the value of a Java volatile the value is guaranteed to be read directly from memory. Furthermore, all the variables visible to the thread reading the volatile variable will also have their values refreshed from main memory.

```
c = other.volatileVarC;
b = other.nonVolatileB;
a = other.nonVolatileA;
```

Notice that the first instruction is a read of a volatile variable (other.volatileVarC). When other.volatileVarC is read in from main memory, the other.nonVolatileB and other.nonVolatileA are also read in from main memory.

Example of the visibility problem:

```
@Test
   void visibilityTest() {
      MySharedObject mySharedObject = new MySharedObject();

      new Thread(() -> mySharedObject.waitForFlag()).start();

      sleepUninterruptibly(1, TimeUnit.SECONDS);
      mySharedObject.flag = true;
      sleepUninterruptibly(10, TimeUnit.SECONDS);
   }

   class MySharedObject {
      volatile boolean flag = false;

      public void waitForFlag() {
         while (!flag) {}
         System.out.println("flag update has been read by another thread");
      }
   }
```

Having the flag variable not declared as volatile, the “flag update has been read by another thread” message is never going to appear.

Another visibility problem. Consider the problem below. What value is assigned to y?

```
static int x = 0;

public static void main(String[] args) {
   x = 1;
```

```
    Thread t = new Thread() {
        public void run() {
            int y = x;
        };
    };
    t.start();
}
```

Complex explanation: When a statement invokes Thread.start(), every statement that has a happens-before relationship with that statement also has a happens-before relationship with every statement executed by the new thread. The effects of the code that led up to the creation of the new thread are visible to the new thread.

When a thread terminates and causes a Thread.join() in another thread to return, then all the statements executed by the terminated thread have a happens-before relationship with all the statements following the successful join. The effects of the code in the thread are now visible to the thread that performed the join.

Simple explanation: The main thread has changed field x. Java memory model does not guarantee that this change will be visible to other threads if they are not synchronized with the main thread. But thread t will see this change because the main thread called t.start() and JLS guarantees that calling t.start() makes the change to x visible in t.run() so y is guaranteed to be assigned 1. In other words, **the newly started thread is guaranteed to see the changes made by the thread that started it**, NOT the ones made by other threads.

synchronized keyword

Java synchronized blocks provide visibility guarantees that are similar to those of Java volatile variables.

When a thread enters a synchronized block, all variables visible to the thread are refreshed from main memory.

When a thread exits a synchronized block, all variables visible to the thread are written back to main memory.

Race Condition

A race condition is a concurrency problem that may occur inside a critical section. A critical section is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.

When the result of multiple threads executing a critical section may differ depending on the sequence in which the threads execute, the critical section is said to contain a race condition. The term race condition stems from the metaphor that the threads are racing through the critical section, and that the **result of that race impacts the result of executing the critical section**.

Race conditions can occur when two or more threads read and write the same variable according to one of these two patterns:

- Read-modify-write
- Check-then-act

Here is the read-modify-write problem:

```
@Test
    void atomicityProblemTest() {
        MySharedObject mySharedObject = new MySharedObject();
        Thread thread = new Thread(mySharedObject);
        thread.start();

        for (int i = 0; i < 1000; i++) {
            mySharedObject.doReadWriteOnSharedResource();
        }

        Uninterruptibles.joinUninterruptibly(thread);
        System.out.println("Result: " + mySharedObject.cnt);
    }

    class MySharedObject implements Runnable {
        int cnt = 0;

        void doReadWriteOnSharedResource() {
            cnt++;
        }

        @Override
        public void run() {
            for (int i = 0; i < 1000; i++) {
                doReadWriteOnSharedResource();
            }
        }
    }
```

Explanation: Increment (i++) is probably not atomic in Java because atomicity is a special requirement which is not present in the majority of the uses. That requirement has a significant overhead: **there is a large cost in making an increment operation atomic**; it involves synchronization at both the software and hardware levels that need not be present in an ordinary increment.

Here is the solution based on synchronized block:

```
synchronized void doReadWriteOnSharedResource() {
    cnt++;
}
```

This problem can also be solved using Compare-And-Swap instruments as well as using Lock objects.

Compare-And-Swap

The compare-and-swap (CAS) instruction is an uninterruptible instruction that reads a memory location, compares the read value with an expected value, and stores a new value in the memory location when the read value matches the expected value. Otherwise, nothing is done.

AtomicInteger:

```
AtomicInteger cnt = new AtomicInteger(0);

void doReadWriteOnSharedResource() {
    cnt.incrementAndGet();
}
```

Java 5 offers this support via java.util.concurrent.atomic: a toolkit of classes used for lock-free, thread-safe programming on single variables: AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference. It also offers array versions of integer, long integer, and reference (AtomicIntegerArray, AtomicLongArray, and AtomicReferenceArray), markable and stamped reference classes for atomically updating a pair of values (AtomicMarkableReference and AtomicStampedReference), and more.

Locks

A lock is a tool for controlling access to a shared resource by multiple threads. Commonly, a lock provides **exclusive access to a shared resource**: only one thread at a time can acquire the lock and all access to the shared resource requires that the lock be acquired first. However, **some locks may allow concurrent access to a shared resource**, such as the read lock of a ReadWriteLock.

All Lock implementations must enforce the same memory synchronization semantics as provided by the built-in monitor lock, as described in Chapter 17 of The Java™ Language Specification:

- A successful lock operation has the same memory synchronization effects as a successful Lock action.

- A successful unlock operation has the same memory synchronization effects as a successful Unlock action.

ReentrantLock - A reentrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.

Example of using ReentrantLock:

```
int cnt = 0;
Lock lock = new ReentrantLock();

void doReadWriteOnSharedResource() {
    lock.lock();
    try {
        cnt++;
    } finally {
        lock.unlock();
    }
}
```

Note: All Lock implementations must enforce the same memory synchronization semantics as provided by the built-in monitor lock, as described in The Java Language Specification, Third Edition (17.4 Memory Model) https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/locks/Lock.html (that's why we don't need a **volatile** keyword for cnt variable)

To boost performance, ReentrantLock's synchronization is managed by a subclass of the abstract java.util.concurrent.locks.AbstractQueuedSynchronizer class. In turn, this class leverages the undocumented sun.misc.Unsafe class and its compareAndSwapInt() CAS method.

A **ReadWriteLock** maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive.

All ReadWriteLock implementations must guarantee that the memory synchronization effects of writeLock operations (as specified in the Lock interface) also hold with respect to the associated readLock. That is, a thread successfully acquiring the read lock will see all updates made upon previous release of the write lock.

A read-write lock allows for a greater level of concurrency in accessing shared data than that permitted by a mutual exclusion lock. It exploits the fact that while only a single thread at a time (a writer thread) can modify the shared data, in many cases any number of threads can concurrently read the data (hence reader threads). In theory, the increase in concurrency permitted by the use of a read-write lock will lead to performance improvements over the use of a mutual exclusion lock. In practice this increase in concurrency will only be fully realized on a multi-processor, and then only if the access patterns for the shared data are suitable.

Executors

Executor interface: An object that executes submitted Runnable tasks. This interface provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc.

```
void execute(Runnable command);
```

An Executor is normally used instead of explicitly creating threads. For example, rather than invoking new Thread(new RunnableTask()).start() for each of a set of tasks, you might use:

```
Executor executor = anExecutor();
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
...
```

However, the Executor interface does not strictly require that execution be asynchronous. In the simplest case, an executor can run the submitted task immediately in the caller's thread:

```
class DirectExecutor implements Executor {
  public void execute(Runnable r) {
    r.run();
  }
}
```

ExecutorService: An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.

An ExecutorService can be shut down, which will cause it to reject new tasks. Two different methods are provided for shutting down an ExecutorService. The **shutdown** method will allow previously submitted tasks to execute before terminating, while the **shutdownNow** method prevents waiting tasks from starting and attempts to stop currently executing tasks. Upon termination, an executor has no tasks actively executing, no tasks awaiting execution, and no new tasks can be submitted.

Method submit extends base method Executor.execute(Runnable) by creating and returning a Future that can be used to cancel execution and/or wait for completion.

ThreadPoolExecutor: An ExecutorService that executes each submitted task using one of possibly several pooled threads, normally configured using Executors factory methods.

Thread pools address two different problems: they usually provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead, and they provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks. Each ThreadPoolExecutor also maintains some basic statistics, such as the number of completed tasks.

ThreadPoolExecutor can be instantiated using constructor:

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
```

The pool consists of a fixed number of core threads that are kept inside all the time, and some excessive threads that may be spawned and then terminated when they are not needed anymore. The **corePoolSize** parameter is the number of core threads that will be instantiated and kept in the pool. When a new task comes in, if all core threads are busy and the internal queue is full, then the pool is allowed to grow up to **maximumPoolSize**.

The **keepAliveTime** parameter is the interval of time for which the excessive threads (instantiated in excess of the corePoolSize) are allowed to exist in the idle state. By default, the ThreadPoolExecutor only considers non-core threads for removal. In order to apply the same removal policy to core threads, we can use the allowCoreThreadTimeOut(true) method.

Let's create a thread pool executor with corePoolSize=maximumPoolSize=4, no keepAlive time, simple ArrayBlockingQueue and blocking RejectedExecutionHandler:

```
 @Test
   void executorTest() {
      ThreadPoolExecutor executor1 = new ThreadPoolExecutor(4, 4, 0, TimeUnit.SECONDS,
            new ArrayBlockingQueue<>(1000),
            Thread::new,
            (task, executor) -> Uninterruptibles.putUninterruptibly(executor.getQueue(), task));
   }
```

Most typical configurations are predefined in the Executors static methods:

```
@Test
   void predefinedExecutorsTest() {

      ExecutorService fixedThreadPoolExecutor = Executors.newFixedThreadPool(8);
      CompletableFuture.runAsync(() -> System.out.println("Handled by fixedThreadPoolExecutor"));

      ExecutorService cachedThreadPoolExecutor = Executors.newCachedThreadPool();
      CompletableFuture.runAsync(() -> System.out.println("Handled by
cachedThreadPoolExecutor"));

      ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(4);
      scheduledExecutorService.scheduleAtFixedRate(() -> System.out.println("Scheduled task"),
```

```
            0, 1, TimeUnit.SECONDS);

    ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
    CompletableFuture.runAsync(() -> System.out.println("Handled by a singleThreadExecutor"));

    Uninterruptibles.sleepUninterruptibly(10, TimeUnit.SECONDS);
    System.out.println("Shutting down...");
    fixedThreadPoolExecutor.shutdown();
    cachedThreadPoolExecutor.shutdown();
    scheduledExecutorService.shutdown();
    singleThreadExecutor.shutdown();

}
```

Coordination of threads

CountDownLatch

Essentially, by using a **CountDownLatch** we can cause a thread to block until other threads have completed a given task.

Simply put, a CountDownLatch has a counter field, which you can decrement as we require. We can then use it to block a calling thread until it's been counted down to zero.

If we were doing some parallel processing, we could instantiate the CountDownLatch with the same value for the counter as a number of threads we want to work across. Then, we could just call **countdown**() after each thread finishes, guaranteeing that a dependent thread calling **await**() will block until the worker threads are finished.

Phaser

The **Phaser** is a reusable synchronization barrier. It allows us to build logic in which threads need to wait on the barrier before going to the next step of execution.

We can coordinate multiple phases of execution, reusing a Phaser instance for each program phase. Each phase can have a different number of threads waiting for advancing to another phase. We'll have a look at an example of using phases later on.

To participate in the coordination, the thread needs to **register**() itself with the Phaser instance.

The thread signals that it arrived at the barrier by calling the **arriveAndAwaitAdvance**(), which is a blocking method. When the number of arrived parties is equal to the number of registered parties, the execution of the program will continue, and the phase number will increase. We can get the current phase number by calling the getPhase() method.

When the thread finishes its job, we should call the **arriveAndDeregister**() method to signal that the current thread should no longer be accounted for in this particular phase.

A Phaser may be used instead of a CountDownLatch to control a one-shot action serving a variable number of parties. The typical idiom is for the method setting this up to first register, then start all the actions, then deregister, as in:

```
void runTasks(List<Runnable> tasks) {
  Phaser startingGate = new Phaser(1); // "1" to register self
  // create and start threads
  for (Runnable task : tasks) {
    startingGate.register();
    new Thread(() -> {
      startingGate.arriveAndAwaitAdvance();
      task.run();
    }).start();
```

```
  }

  // deregister self to allow threads to proceed
  startingGate.arriveAndDeregister();
}
```