Java 5 (Code name: Tiger)

1. **Generics**: provides compile-time (static) type safety for collections and eliminates the need for most typecasts (type conversion).

2. **Metadata**: also called **annotations**; allows language constructs such as classes and methods to be tagged with additional data, which can then be processed by metadata-aware utilities.

3. **Auto-boxing/unboxing**: automatic conversions between primitive types (such as int) and primitive wrapper classes (such as Integer).

4. **Enumerations**: the enum keyword creates a type-safe, ordered list of values (such as Day.MONDAY, Day.TUESDAY). Previously this could only be achieved by non-type-safe constant integers or manually constructed classes.

5. **Varargs**: the last parameter of a method can now be declared using a type name followed by three dots; in the calling code any number of parameters of that type can be used and they are then placed in an array to be passed to the method, or alternatively the calling code can pass an array of that type.

6. **Enhanced for-each loop**: The FOR-loop syntax is extended with special syntax for iterating over each member of either an array or any Iterable, such as the standard Collection classes.

7. **Static imports**: This facility lets you avoid qualifying static members with class names without the shortcomings of the "Constant Interface antipattern".

Java 6 (Code name: Mustang)

1. These new collection interfaces are provided:
   - **Deque** - a double ended queue, supporting element insertion and removal at both ends. Extends the Queue interface.
   - **BlockingDeque** - a Deque with operations that wait for the deque to become non-empty when retrieving an element and wait for space to become available in the deque when storing an element. Extends both the Deque and BlockingQueue interfaces. (This interface is part of java.util.concurrent.)
   - **NavigableSet** - a SortedSet extended with navigation methods reporting closest matches for given search targets. A NavigableSet may be accessed and traversed in either ascending or descending order. This interface is intended to supersede the SortedSet interface.
   - **NavigableMap** - a SortedMap extended with navigation methods returning the closest matches for given search targets. A NavigableMap may be accessed and traversed in either ascending or descending key order. This interface is intended to supersede the SortedMap interface.
   - **ConcurrentNavigableMap** - a ConcurrentMap that is also a NavigableMap. (This interface is part of java.util.concurrent.)

2. The following concrete implementation classes have been added:
   - **ArrayDeque** - efficient resizable-array implementation of the Deque interface.
   - **ConcurrentSkipListSet** - concurrent scalable skip list implementation of the NavigableSet interface.

- **ConcurrentSkipListMap** - concurrent scalable skip list implementation of the ConcurrentNavigableMap interface.
- **LinkedBlockingDeque** - concurrent scalable optionally bounded FIFO blocking deque backed by linked nodes.
- **AbstractMap.SimpleEntry** - simple mutable implementation of Map.Entry
- **AbstractMap.SimpleImmutableEntry** - simple immutable implementation of Map.Entry

3. **Console** - Contains methods to access a character-based console device. The readPassword() methods disable echoing thus they are suitable for retrieval of sensitive data such as passwords. The method System.console() returns the unique console associated with the Java Virtual Machine.

4. Methods to retrieve disk usage information, in File Class:
   - **getTotalSpace()** returns the size of the partition in bytes.
   - **getFreeSpace()** returns the number of unallocated bytes in the partition.
   - **getUsableSpace()** returns the number of bytes available on the partition and includes checks for write permissions and other operating system restrictions.

5. Methods to set or query file permissions, in File Class:
   - Set the owner's or everybody's write permission:
     ```
     setWritable(boolean writable, boolean ownerOnly)
     setWritable(boolean writable)
     ```
   - Set the owner's or everybody's read permission:
     ```
     setReadable(boolean readable, boolean ownerOnly)
     setReadable(boolean readable)
     ```
   - Set the owner's or everybody's execute permission:
     ```
     setExecutable(boolean executable, boolean ownerOnly)
     setExecutable(boolean executable)
     ```
   - canExecute(), canRead(), canWrite().

6. **The Java API for XML Processing (JAXP):** enables applications to parse, transform, validate and query XML documents using an API that is independent of a particular XML processor implementation. JAXP provides a pluggability layer to enable vendors to provide their own implementations without introducing dependencies in application code. Using this software, application and tool developers can build fully-functional XML-enabled Java applications for e-commerce, application integration, and web publishing.

Java 7 (Code name: Dolphin)

1. New file I/O library adding support for multiple file systems, file metadata and symbolic links. The new packages are java.nio.file, java.nio.file.attribute and java.nio.file.spi.

2. **Binary integer literals:** In Java SE 7, the integral types (byte, short, int, and long) can also be expressed using the binary number system. To specify a binary literal, add the prefix 0b or 0B to the number. The following examples show binary literals:

```
// An 8-bit 'byte' value:
byte aByte = (byte)0b00100001;
```

```
// A 16-bit 'short' value:
short aShort = (short)0b1010000101000101;

// Some 32-bit 'int' values:
int anInt1 = 0b10100001010001011010000101000101;
int anInt2 = 0b101;
int anInt3 = 0B101; // The B can be upper or lower case.

// A 64-bit 'long' value. Note the "L" suffix:
long aLong =
0b1010000101000101101000010100010110100001010001011010000101000101L;
```

3. **Underscores in numeric literals:** In Java SE 7 and later, any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables you, for example, to separate groups of digits in numeric literals, which can improve the readability of your code.

```
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
```

4. Timsort is used to sort collections and arrays of objects instead of merge sort.

5. **The fork/join framework:** which is based on the ForkJoinPool class, is an implementation of the Executor interface. It is designed to efficiently run a large number of tasks using a pool of worker threads. A work-stealing technique is used to keep all the worker threads busy, to take full advantage of multiple processors.

6. **Multithreaded Custom Class Loaders:** In earlier releases, certain types of custom class loaders were prone to deadlock. The Java 7, modifies the locking mechanism to avoid deadlock. For backward compatibility, the Java 7 release ***continues to lock a class loader object unless it registers as parallel capable***.
   The function of a java.lang.ClassLoader is to locate the bytecode for a particular class, then transform that bytecode into a usable class in the runtime system. The CLASSPATH environment variable is one way to indicate to the runtime system where the bytecode is located.
   Custom class loaders will not run into deadlocks if they adhere to an acyclic class loader delegation model. In this model, every class loader has a parent (delegate). When a class is requested, the class loader first checks if the class was loaded previously. If the class is not found, the class loader asks its parent to locate the class. If the parent cannot find the class, the class loader attempts to locate the class itself.

7. **Using Non-Reifiable Formal Parameters with Varargs Methods:** Most parameterized types, such as ArrayList<Number> and List<String>, are non-reifiable types. A non-reifiable type is a type that is not completely available at runtime. At compile time, non-reifiable types undergo a process called type erasure during which the compiler removes information related to type parameters and type arguments. This ensures binary compatibility with Java libraries and applications that were created before generics. Because type erasure removes

information from parameterized types at compile-time, these types are non-reifiable.

Heap pollution occurs when a variable of a parameterized type refers to an object that is not of that parameterized type. This situation can only occur if the program performed some operation that would give rise to an unchecked warning at compile-time. An unchecked warning is generated if, either at compile-time or at runtime, the correctness of an operation involving a parameterized type cannot be verified. Consider the following example:

```
List l = new ArrayList<Number>();
List<String> ls = l;       // unchecked warning
l.add(0, new Integer(42)); // another unchecked warning
String s = ls.get(0);      // ClassCastException is thrown
```

During type erasure, the types ArrayList<Number> and List<String> become ArrayList and List, respectively.

The variable "ls" has the parameterized type List<String>. When the List referenced by "l" is assigned to "ls", the compiler generates an unchecked warning; the compiler is unable to determine at compile time, and moreover knows that the JVM will not be able to determine at runtime, if "l" refers to a List<String> type; it does not. Consequently, heap pollution occurs.

When the compiler encounters a varargs method, it translates the varargs formal parameter into an array. However, the Java programming language does not permit the creation of arrays of parameterized types. In the method ArrayBuilder.addToList, the compiler translates the varargs formal parameter "T..." elements to the formal parameter T[] elements, an array. However, because of type erasure, the compiler converts the varargs formal parameter to Object[] elements. Consequently, there is a possibility of heap pollution.

If you declare a varargs method that has parameterized parameters, and you ensure that the body of the method does not throw a ClassCastException or other similar exception due to improper handling of the varargs formal parameter (as shown in the ArrayBuilder.faultyMethod method), you can suppress the warning that the compiler generates for these kinds of varargs methods by using one of the following options:

- Add the following annotation to static and non-constructor method declarations: `@SafeVarargs`
  Unlike the @SuppressWarnings annotation, the @SafeVarargs annotation is a documented part of the method's contract; this annotation asserts that the implementation of the method will not improperly handle the varargs formal parameter.
- Add the following annotation to the method declaration:
  `@SuppressWarnings({"unchecked", "varargs"})`
  Unlike the @SafeVarargs annotation, the @SuppressWarnings("varargs") does not suppress warnings generated from the method's call site.
- Use the compiler option -Xlint:varargs.

8. **Strings in switch:** The switch statement compares the String object in its expression with the expressions associated with each case label as if it were using the String.equals() method; consequently, the comparison of String objects in switch statements is case sensitive. The Java compiler generates generally more efficient bytecode from switch statements that use String objects than from chained if-then-else statements.

```
int monthNameToDays(String s, int year){
        switch(s){
          case "April":
          case "June":
          return 30;
          ...
          default: ...
        }
}
```

9. **The try-with-resources statement**: The try-with-resources statement is a try statement that declares one or more resources. A resource is as 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 readFirstLineFromFileWithFinallyBlock(String path) throws IOException {
  BufferedReader br = new BufferedReader(new FileReader(path));
  try {
    return br.readLine();
  } finally {
    if (br != null) br.close();
  }
}
```

In this example, the resource declared in the try-with-resources statement is a BufferedReader. The declaration statement appears within parentheses immediately after the try keyword. The class BufferedReader, in Java 7 and later, implements the interface java.lang.AutoCloseable. Because the BufferedReader instance is declared in a try-with-resource statement, it will be closed regardless of whether the try statement completes normally or abruptly (as a result of the method BufferedReader.readLine() throwing an IOException).

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

However, in this example, if the methods readLine() and close() both throw exceptions, then the method readFirstLineFromFileWithFinallyBlock() throws the exception thrown from the finally block; the exception thrown from the try block is suppressed. In contrast, in the example readFirstLineFromFile(), if exceptions are thrown from both the try block and the try-with-resources statement, then the method readFirstLineFromFile() throws the exception thrown from the try block; the exception thrown from the try-with-resources block is suppressed. You can retrieve these ***suppressed exceptions*** by calling the Throwable.getSuppressed() method from the exception thrown by the try block.

10. **Catching multiple exception types:**

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

11. **More Precise Rethrow:** In Java 7, you can specify the exception types FirstException and SecondException in the throws clause in the rethrowException method declaration. The Java 7 compiler can determine that the exception thrown by the statement "throw e" must have come from the try block, and the only exceptions thrown by the try block can be FirstException and SecondException. Even though the exception parameter of the catch clause, "e", is type Exception, the compiler can determine that it is an instance of either FirstException or SecondException.

```
public void foo(String bar) throws FirstException, SecondException {
    try {
      // Code that may throw both
      // FirstException and SecondException
    }
    catch (Exception e) {
      throw e;
    }
}
```

12. **Improved type inference for generic instance creation**: You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context. This pair of angle brackets is informally called the diamond operator. For example, consider the following variable declaration:

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

Java 7: `Map<String, List<String>> myMap = new HashMap<>();`

13. **Garbage-First Collector:** The heap is partitioned into a set of equal-sized heap regions, each a contiguous range of virtual memory. G1 performs a concurrent global marking phase to determine the liveness of objects throughout the heap. After the mark phase completes, G1 knows which regions are mostly empty. It collects in these regions first, which usually yields a large amount of free space. ***This is why this method of garbage collection is called Garbage-First.*** As the name suggests, G1 concentrates its collection and compaction activity on the areas of the heap that are likely to be full of reclaimable objects, that is, garbage. G1 uses a pause prediction model to meet a user-defined pause time target and selects the number of regions to collect based on the specified pause time target. The regions identified by G1 as ripe for reclamation are garbage collected using evacuation. G1 copies objects from one or more regions of the heap to a single region on the heap, and in the process both compacts and frees up memory. This evacuation is performed in parallel on multi-processors, to decrease pause times and increase throughput. Thus, with each garbage collection, G1 continuously works to reduce fragmentation, working within the user defined pause times. This is beyond the capability of both the previous methods. CMS (Concurrent Mark Sweep) garbage collection does not do compaction. ParallelOld garbage collection performs only whole-heap compaction, which results in considerable pause times. It is important to note that G1 is not a real-time collector. It meets the set pause time target with high probability but not absolute certainty. Based on data from previous collections, G1 does an estimate of how many regions can be collected within the user specified target time. Thus, the collector has a reasonably accurate model of the cost of collecting the regions, and it uses this model to determine which and how many regions to collect while staying within the pause time target.

G1 is a generational, incremental, parallel, mostly concurrent, stop-the-world, and evacuating garbage collector which monitors pause-time goals in each of the stop-the-world pauses. Like other collectors, G1 splits the heap into (virtual) young and old generations. Space-reclamation efforts concentrate on the young generation where it is most efficient to do so, with occasional space-reclamation in the old generation

G1 reclaims space mostly by using evacuation: live objects found within selected memory areas to collect are copied into new memory areas, compacting them in the process. After an evacuation has been completed, the space previously occupied by live objects is reused for allocation by the application.

The Garbage-First collector is not a real-time collector. It tries to meet set pause-time targets with high probability over a longer time, but not always with absolute certainty for a given pause.

The downside is that the G1 doesn't perform well with small heaps.

Java 8 (Code name: Spider)

1. **Lambda Expression:** Lambda Expression are the way through which we can visualize functional programming in the java object-oriented world. Objects are the base of java programming language, and we can never have a function without an Object. Lambda Expressions syntax is **(argument) -> (body).**

```
Runnable r = new Runnable(){
        @Override
        public void run() {
                System.out.println("My Runnable");
}};

Lambda: Runnable r1 = () -> System.out.println("My Runnable");
```

If you have single statement in method implementation, we don't need curly braces also. For example "Interface1" anonymous class can be instantiated using lambda as follows:

```
Interface1 i1 = (s) -> System.out.println(s);
i1.method1("abc");
```

So, lambda expressions are means to create anonymous classes of functional interfaces easily. ***There are no runtime benefits of using lambda expressions.***

2. **The forEach() in Iterable interface:** Whenever we need to traverse through a Collection, we need to create an Iterator whose whole purpose is to iterate over and then we have business logic in a loop for each of the elements in the Collection. We might get ConcurrentModificationException if iterator is not used properly.

Java 8 has introduced forEach() method in java.lang.Iterable interface so that while writing code we focus on business logic only. The forEach() method takes java.util.function.Consumer object as argument, so it helps in having our business logic at a separate location that we can reuse.

```
//Consumer implementation that can be reused
class MyConsumer implements Consumer<Integer> {
        public void accept(Integer t) {
                System.out.println("Consumer impl Value::" + t);
}}
//traversing using Iterator
Iterator<Integer> it = myList.iterator();
while(it.hasNext()) {
```

```
        Integer i = it.next();
        System.out.println("Iterator Value::"+i);
}
//traversing through forEach() of Iterable with anonymous class
myList.forEach(new Consumer<Integer>() {
        public void accept(Integer t) {
                System.out.println("forEach anonymous Value::" + t);
        }
});
//traversing with Consumer interface implementation
MyConsumer action = new MyConsumer();
myList.forEach(action);
```

3. **Java Functional Interfaces:** An interface with exactly one abstract method is known as Functional Interface. A new annotation @FunctionalInterface has been introduced to mark an interface as Functional Interface. @FunctionalInterface annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces. It's optional but good practice to use it. Functional interfaces enable us to use lambda expressions to instantiate them.

4. **Extension Methods:** Java 8 enables us to add non-abstract method implementations to interfaces by utilizing the default keyword.
   Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes. Java interface default methods has bridge down the differences between interfaces and abstract classes. Java 8 interface default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.

5. **Static Methods in Interfaces:** The static methods in interface are like default method, so we need not implement them in the implementation classes. We can safely add them to the existing interfaces without changing the code in the implementation classes. Since these methods are static, we cannot override them in the implementation classes.
   Java interface static method is part of interface, we can't use it for implementation class objects. Java interface static methods are good for providing utility methods, for example null check, collection sorting etc. Java interface static method helps us in providing security by not allowing implementation classes to override them.

```
interface MyInterface {
    /* This is a default method so we need not implement this method in the
    implementation classes
    */
    default void newMethod() {
        System.out.println("Newly added default method");
    }
    /* This is a static method. Static method in interface is similar to default method
    except that we cannot override them in the implementation classes.
        * Similar to default methods, we implement these methods in implementation
    classes so we can add them to the existing interfaces.
        */
    static void anotherNewMethod() {
        System.out.println("Newly added static method");
    }
```

```
    /* Already existing public and abstract method. We must need to implement this
    method in implementation classes.
        */
    void existingMethod(String str);
}
```

6. **Method Reference:** is the shorthand syntax for a lambda expression that executes just ONE method.

```
class Numbers {
  public static boolean isMoreThanFifty(int n1, int n2) {
    return (n1 + n2) > 50;
  }
  public static List<Integer> findNumbers(
    List<Integer> l, BiPredicate<Integer, Integer> p) {
      List<Integer> newList = new ArrayList<>();
      for(Integer i : l) {
        if(p.test(i, i + 10)) {
          newList.add(i);
        }
      }
      return newList;
  }
}
```

We can call the findNumbers() method:

```
List<Integer> list = Arrays.asList(12,5,45,18,33,24,40);

// Using an anonymous class
findNumbers(list, new BiPredicate<Integer, Integer>() {
  public boolean test(Integer i1, Integer i2) {
    return Numbers.isMoreThanFifty(i1, i2);
  }
});

// Using a lambda expression
findNumbers(list, (i1, i2) -> Numbers.isMoreThanFifty(i1, i2));

// Using a method reference
findNumbers(list, Numbers::isMoreThanFifty);
```

- Reference to an instance method of a particular object.
  The syntax: **<containingObject>::<instanceMethodName>**
- Reference to an instance method of an arbitrary object of a particular type.
  The syntax: **<ContainingType>::<methodName>**
- Reference to a constructor. The syntax: **<ClassName>::<new>**

Syntax	Example	As Lambda
ClassName::new	String::new	() -> new String()
Class::staticMethodName	String::valueOf	(s) -> String.valueOf(s)
object::instanceMethodName	x::toString	() -> "hello".toString()
Class::instanceMethodName	String::toString	(s) -> s.toString()

7. **Constructor References**: Same concept as a method reference.

```
// Using a lambda expression
Factory<List<String>> f = () -> return new ArrayList<String>();
// Using a method reference
Factory<List<String>> f = ArrayList<String>::new;
```

8. **Concurrency API improvements:**
   - **ConcurrentHashMap** compute(), forEach(), forEachEntry(), forEachKey(), forEachValue(), merge(), reduce() and search() methods.
   - **CompletableFuture** that may be explicitly completed (setting its value & status).
   - **Executors newWorkStealingPool()** method to create a work-stealing thread pool using all available processors as its target parallelism level.

9. **Java IO improvements:**
   - **Files.list(Path dir)** that returns a lazily populated Stream, the elements of which are the entries in the directory.
   - **Files.lines(Path path)** that reads all lines from a file as a Stream.
   - **Files.find()** that returns a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.
   - **BufferedReader.lines()** that return a Stream, the elements of which are lines read from this BufferedReader.

10. Some new methods added in Collection API are:
    - **Iterator** default method **forEachRemaining(Consumer action)** to perform the given action for each remaining element until all elements have been processed or the action throws an exception.
    - **Collection** default method **removeIf(Predicate filter)** to remove all of the elements of this collection that satisfy the given predicate.
    - **Collection spliterator()** method returning Spliterator instance that can be used to traverse elements sequentially or parallel.
    - Map **replaceAll(), compute(), merge()** methods.

11. **The Hashmap performance improvements:** Until Java 7, java.util.Hashmap implementations always suffered with the problem of Hash Collision, i.e. when multiple hashCode() values end up in the same bucket, values are placed in a Linked List implementation, which ***reduces Hashmap performance from O(1) to O(n).***
Improve the performance of java.util.HashMap under high hash-collision conditions by using balanced trees rather than linked lists to store map entries. This will improve collision performance for any key type that implements Comparable. This change applies only to HashMap, LinkedHashMap, and ConcurrentHashMap.
The principal idea is that once the number of items in a hash bucket grows beyond a certain threshold (TREEIFY_THRESHOLD), that bucket will switch from using a linked list of entries to a balanced tree. In the case of high hash collisions, this will ***improve worst-case performance from O(n) to O(log n),*** and when they become too small (due to removal or resizing) they are converted back to Linked List.

```
static final int TREEIFY_THRESHOLD = 8;
static final int UNTREEIFY_THRESHOLD = 6;
```

Also note that in rare situations, this change could introduce a change to the iteration order of HashMap and HashSet. A particular iteration order is not specified for HashMap objects – any code that depends on iteration order should be fixed.

12. **From PermGen to Metaspace:** ***PermGen*** (Permanent generation) space stores the meta-data about the class. That involves information about the class hierarchy, information about the class like its name, fields, methods, bytecode. Run time constant pool that stores immutable fields that are pooled in order to save space like String are also kept in PermGen.
PermGen space is contiguous to heap space and the information stored in PermGen is relatively permanent in nature (not garbage collected as swiftly as in young generation like Eden). The space allocated to PermGen is controlled by the argument -XX:MaxPermSize and the default is 64M (30% higher in 64 bit JVM which means around 83M in a 64 bit JVM).
PermGen's size is Fixed at start-up and can't be changed dynamically which may lead to ***java.lang.OutOfMemoryError:PermGen*** error. As application grows the classes that are loaded will grow and the size of class metadata may become more than what was the size of the PermGen space. Once the permanent generation space is full OutOfMemoryError:PermGen Space error occurs.

Metaspace memory allocation model: Most allocations for the class metadata are now allocated out of native memory. The classes that were used to describe class metadata have been removed.
By default, class metadata allocation is limited by the amount of available native memory (capacity will of course depend if you use a 32-bit JVM vs. 64-bit along with OS virtual memory availability). A new flag is available (***MaxMetaspaceSize***), allowing you to limit the amount of native memory used for class metadata. If you don't specify this flag, the Metaspace will dynamically re-size depending on the application demand at runtime. Other flag options are ***MetaspaceSize***, ***MinMetaspaceFreeRatio*** and ***MaxMetaspaceFreeRatio***.
Garbage collection of the dead classes and classloaders is triggered once the class metadata usage reaches the "MaxMetaspaceSize". Proper monitoring & tuning of the Metaspace will obviously be required to limit the frequency or delay of such garbage collections. Excessive Metaspace garbage collections may be a symptom of classes, classloaders memory leak or inadequate sizing for your application.

PermGen	MetaSpace
It is removed from java 8.	It is introduced in Java 8.
PermGen always has a fixed maximum size.	Metaspace by default auto increases its size depending on the underlying OS.
Contiguous Java Heap Memory.	Native Memory(provided by underlying OS).
Inefficient garbage collection.	Efficient garbage collection.

13. **Sorting Map directly with Comparators:** Map interface added default methods which gives comparators like: comparingByKey(), comparingByValue().

```
Map<String, String> map = new HashMap<>();
map.put("C", "c");
map.put("B", "b");
map.put("Z", "z");
List<Map.Entry<String, String>> sortedByKey =
        map.entrySet().stream()
                .sorted(Map.Entry.comparingByKey())
                        .collect(Collectors.toList());
sortedByKey.forEach(System.out::println);
```

14. **Iterate over map easily with forEach():**

```
Map<String, String> map = new HashMap<>();
map.put("C", "c");
map.put("B", "b");
map.put("Z", "z");
map.forEach((k, v) -> S.out.println("Key: " +k+ " Value: " +v));
```

15. **Replace if-else condition with getOrDefault() method:** This method returns the value to which the specified key is mapped, otherwise returns the given *defaultValue* if this map contains no mapping for the key.

```
Map<String, String> map = new HashMap<>();
map.put("C", "c");
String val = map.getOrDefault("B", "Nah!");
System.out.println(val); // prints Nah!
```

16. **Replace() and Remove() utilities:** The replaceAll() can replace all the values in a single attempt. And replace(K key, V oldValue, V newValue) method replaces the entry for the specified key only if currently mapped to the specified value. In the same way you can use replace(), remove() methods to check by key and values pairs together.

```
Map<String, String> map = new HashMap<>();
map.put("C", "c");
map.put("B", "b");
map.replaceAll((k, v) -> "x"); // values is "x" for all keys.
```

17. **Do not override keys accidentally use putIfAbsent():**

```
Map<String, String> map = new HashMap<>();
map.put("C", "c");
map.put("B", "b");
map.putIfAbsent("B", "x");
System.out.println(map.get("B")); // prints "b"
```

18. **Operate directly on values:** Gone are the days when you needed to get the value for specific keys, process it and put them back. Now you can directly modify with help of compute method. Conditional computes are also available with **computeIfPresent(), computeIfAbsent()** methods.

```
Map<String, String> map = new HashMap<>();
map.put("C", "c");
map.put("B", "b");
map.compute("B", (k, v) -> v.concat(" - new "));
System.out.println(map.get("B")); // prints "b - new"
```

19. **To merge maps use merge() method:** If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. Otherwise, replaces the associated value with the results of the given remapping function, or removes if the result is null.

```
Map<String, String> map = new HashMap<>();
map.put("C", "c");
map.put("B", "b");
map.merge("B", "NEW", (v1, v2) -> v1 + v2);
System.out.println(map.get("B")); // prints bNEW
```

Java 8 Stream:

Major problem:

- We just want to know the sum of integers, but we would also have to provide how the iteration will take place, this is also called external iteration because client program is handling the algorithm to iterate over the list.
- Program is sequential in nature, there is no way we can do this in parallel easily.
- There is a lot of code to do even a simple task.

```
private static int sumIterator(List<Integer> list) {
        Iterator<Integer> it = list.iterator();
        int sum = 0;
        while (it.hasNext()) {
                int num = it.next();
                if (num > 10) {
                        sum += num;
                }
        }
        return sum;
}
```

To overcome we can use, Java Stream API to implement internal iteration, that is better because java framework is in control of the iteration. Internal iteration provides several features such as sequential and parallel execution, filtering based on the given criteria, mapping etc.

```
private static int sumStream(List<Integer> list) {
        return list.stream().
                filter(i -> i > 10).mapToInt(i -> i).sum();
}
```

Collections and Java Stream:

A collection is an in-memory data structure to hold values and before we start using collection, all the values should have been populated. Whereas a java Stream is a data structure that is computed on-demand.

Java Stream doesn't store data, it operates on the source data structure (collection and array) and produce pipelined data that we can use and perform specific operations. Such as we can create a stream from the list and filter it based on a condition. Java Stream operations use functional interfaces that makes it a very good fit for functional programming using lambda expression.

Java Streams are consumable, so there is no way to create a reference to stream for future usage. Since the data is on-demand, it's not possible to reuse the same stream multiple times. Java Streams support sequential as well as parallel processing, parallel processing can be very helpful in achieving high performance for large collections.

All the Java Stream API interfaces and classes are in the **java.util.stream** package. Since we can use primitive data types such as int, long in the collections using auto-boxing and these operations could take a lot of time, there are specific classes for primitive types – **IntStream, LongStream** and **DoubleStream**.

Functional Interfaces in Java 8 Stream:

- **Function and BiFunction:** Function represents a function that takes one type of argument and returns another type of argument. **Function<T, R>** is the generic form where T is the type of the input to the function and R is the type of the result of the function. For handling primitive types, there are specific Function interfaces – ToIntFunction, ToLongFunction, ToDoubleFunction,

ToIntBiFunction, ToLongBiFunction, ToDoubleBiFunction, LongToIntFunction, LongToDoubleFunction, IntToLongFunction, IntToDoubleFunction etc.
The Stream methods where **Function** or its primitive specialization is used are:

- <R> Stream<R> map(Function<? super T, ? extends R> mapper).
- IntStream mapToInt(ToIntFunction<? super T> mapper) – similarly for long and double returning primitive specific stream.
- IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper) – similarly for long and double.
- <A> A[] toArray(IntFunction<A[]> generator).
- <U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner).

- **Predicate and BiPredicate:** It represents a predicate against which elements of the stream are tested. This is used to filter elements from the java stream. Just like **Function**, there are primitive specific interfaces for int, long and double. The Stream methods where **Predicate** or **BiPredicate** specializations are used are:
  - Stream<T> filter(Predicate<? super T> predicate)
  - boolean anyMatch(Predicate<? super T> predicate)
  - boolean allMatch(Predicate<? super T> predicate)
  - boolean noneMatch(Predicate<? super T> predicate)

- **Consumer and BiConsumer:** It represents an operation that accepts a single input argument and returns no result. It can be used to perform some action on all the elements of the java stream. The Java 8 Stream methods where **Consumer, BiConsumer** or its primitive specialization interfaces are used are:
  - Stream<T> peek(Consumer<? super T> action)
  - void forEach(Consumer<? super T> action)
  - void forEachOrdered(Consumer<? super T> action)

- **Supplier:** Supplier represent an operation through which we can generate new values in the stream. Some of the methods in Stream that takes **Supplier** argument are:
  - public static<T> Stream<T> generate(Supplier<T> s)
  - <R> R collect(Supplier<R> supplier,BiConsumer<R, ? super T> accumulator,BiConsumer<R, R> combiner)

The java.util.Optional: Java Optional is a container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value. Stream terminal operations return Optional object.

- Optional<T> reduce(BinaryOperator<T> accumulator)
- Optional<T> min(Comparator<? super T> comparator)
- Optional<T> max(Comparator<? super T> comparator)
- Optional<T> findFirst()
- Optional<T> findAny()

The java.util.Spliterator: For supporting parallel execution in Java 8 Stream API, Spliterator interface is used. Spliterator trySplit() method returns a new Spliterator that manages a subset of the elements of the original Spliterator.

Java Stream Intermediate Operations: Java Stream API operations that returns a new Stream are called intermediate operations. Most of the times, these operations are lazy in nature, so they start producing new stream elements and send it to the next operation. Intermediate operations are never the final result producing operations. Commonly used intermediate operations are **filter** and **map**.

Java Stream Terminal Operations: Java 8 Stream API operations that returns a result or produce a side effect. Once the terminal method is called on a stream, it consumes the stream and after that we can't use stream. Terminal operations are eager in nature i.e., they process all the elements in the stream before returning the result. Commonly used terminal methods are **forEach(), toArray(), min(), max(), findFirst(), anyMatch(), allMatch()** etc. You can identify terminal methods from the return type, they will never return a Stream.

Java Stream Short-Circuiting Operations: An intermediate operation is called short circuiting, if it may produce finite stream for an infinite stream. For example **limit()** and **skip()** are two short circuiting intermediate operations. A terminal operation is called short circuiting, if it may terminate in finite time for infinite stream. For example **anyMatch(), allMatch(), noneMatch(), findFirst()** and **findAny()** are short circuiting terminal operations.

Creating Java Streams:

- We can create Java Stream of integers from a group of int or Integer objects.

```
Stream<Integer> stream = Stream.of(1,2,3,4);
```

- We can use Stream.of() with an array of Objects to return the stream.

```
Stream<Integer> stream = Stream.of(new Integer[]{1,2,3,4});
//works fine

Stream<Integer> stream1 = Stream.of(new int[]{1,2,3,4});
//Compile time error, Type mismatch: cannot convert from Stream<int[]> to Stream<Integer>
```

- We can use Collection stream() to create sequential stream and parallelStream() to create parallel stream.

```
List<Integer> myList = new ArrayList<>();
for(int i=0; i<100; i++) myList.add(i);
//sequential stream
Stream<Integer> sequentialStream = myList.stream();
//parallel stream
Stream<Integer> parallelStream = myList.parallelStream();
```

- We can use Stream.generate() and Stream.iterate() methods to create Stream.

```
Stream<String> stream1 = Stream.generate(() -> {return "abc";});
Stream<String> stream2 = Stream.iterate("abc", (i) -> i);
```

- Using Arrays.stream() and String.chars() methods.

```
LongStream is = Arrays.stream(new long[]{1,2,3,4});
IntStream is2 = "abc".chars();
```