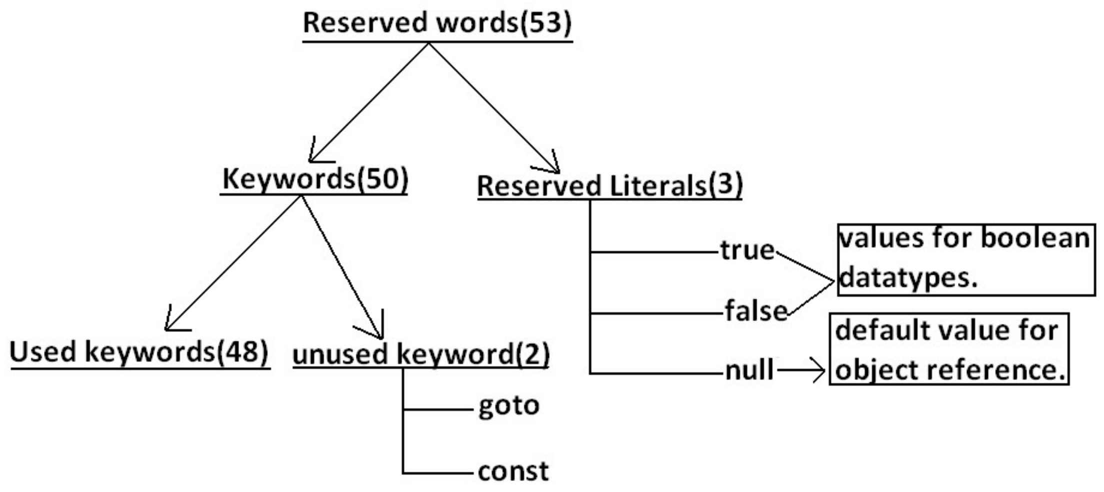
1. Reversed words (53)



1 Volatile vs synchronized

- volatile vs synchronized -volatile keyword used to ensure thread safe, and all writes and reads will be directly on the main memory instead of cpu cache. The effect of the volatile keyword is approximately that each individual read or write operation on that variable is made atomically visible to all threads.Notably, however, an operation that requires more than one read/write -- such as i++, which is equivalent to i = i + 1, which does one read and one write -- is not atomic, since another thread may write to i between the read and the write.   The Atomic classes, like AtomicInteger and AtomicReference, provide a wider variety of operations atomically, specifically including increment for AtomicInteger.

- Using synchronized prevents any other thread from obtaining the monitor (or lock) for the same object, thereby preventing all code blocks protected by synchronization on the same object from executing concurrently.

2 Transient

- transient is a variables modifier used in serialization. At the time of serialization, if we don't want to save value of a particular variable in a file, then we use transient keyword. When JVM comes across transient keyword, it ignores original value of the variable and save default value of that variable data type.

3 Native

- The native keyword in Java is applied to a method to indicate that the method is implemented in native code using JNI (Java Native Interface). The native keyword is a modifier that is applicable only for methods, and we can’t apply it anywhere else. The methods which are implemented in C, C++ are called native methods or foreign methods.
- To improve the performance of the system.
- To achieve machine level/memory level communication.
- To use already existing legacy non-java code.

4 Strictfp

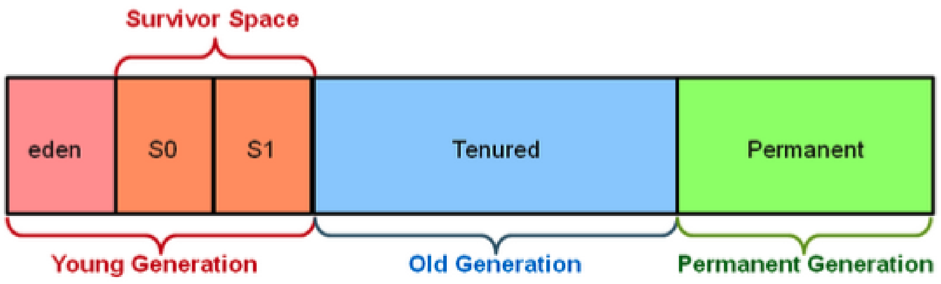
- strictfp is a modifier that stands for strict floating-point which was not introduced in the base version of java as it was introduced in Java version 1.2. It is used in java for restricting floating-point calculations and ensuring the same result on every platform while performing operations in the floating-point variable.
- When a class or an interface is declared with strictfp modifier, then all methods declared in the class/interface, and all nested types declared in the class, are implicitly strictfp.
- strictfp cannot be used with abstract methods. However, it can be used with abstract classes/interfaces.
- Since methods of an interface are implicitly abstract, strictfp cannot be used with any method inside an interface.

5 Garbage Collector

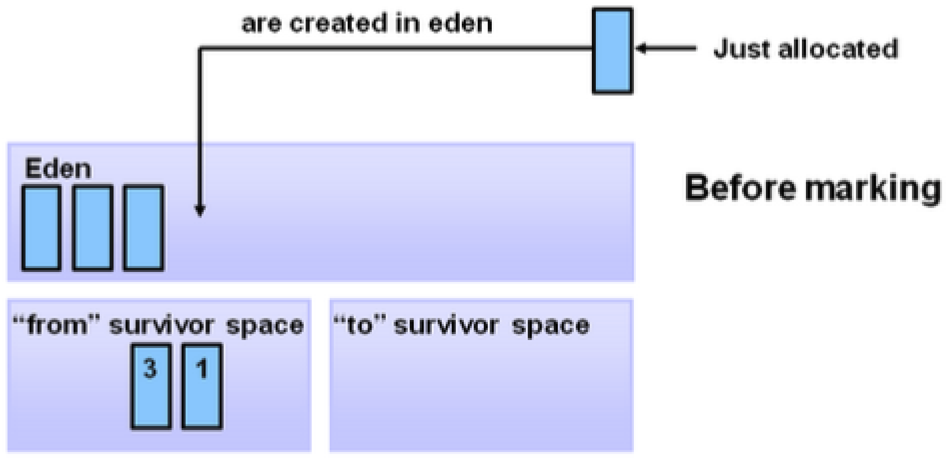
- ### 5.1 Serial collector:
    - uses a single thread to perform all garbage collection work, which makes it relatively efficient because there is no communication overhead between threads.
- ### 5.2 Parallel Collector
    - Also known as throughout collector, similar to the serial collector. The primary difference between the serial and parallel collectors is that the parallel has multiple threads that are used to speed up garbage collection.
- ### 5.3 G1 Collector
    - G1 garbage collector: This server-style collector is for multiprocessor machines with a large amount of memory. It meets garbage collection pause-time goals with high probability while achieving high throughput.

GC process

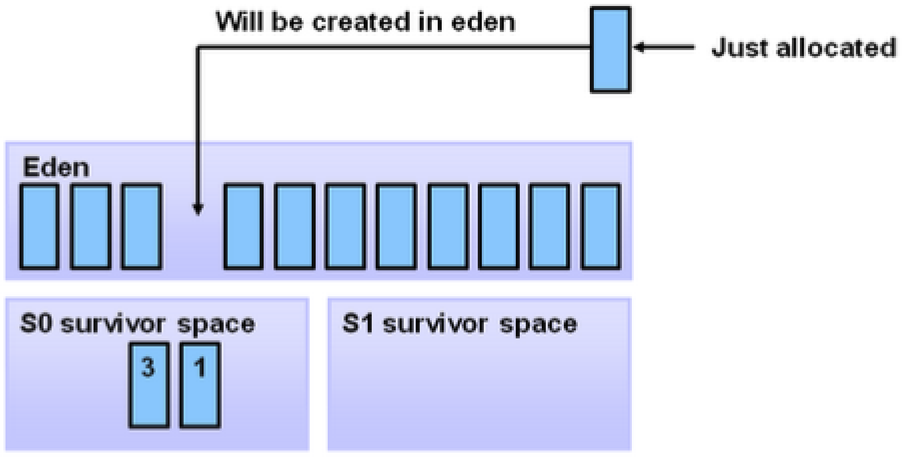
Hotspot Heap Structure



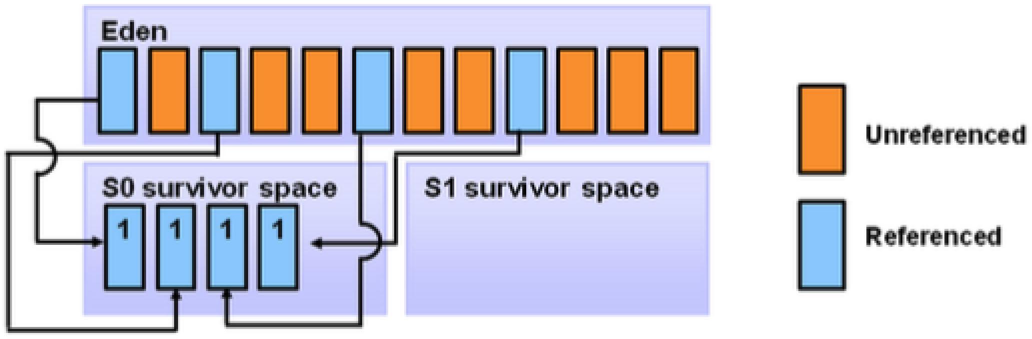
Object Allocation



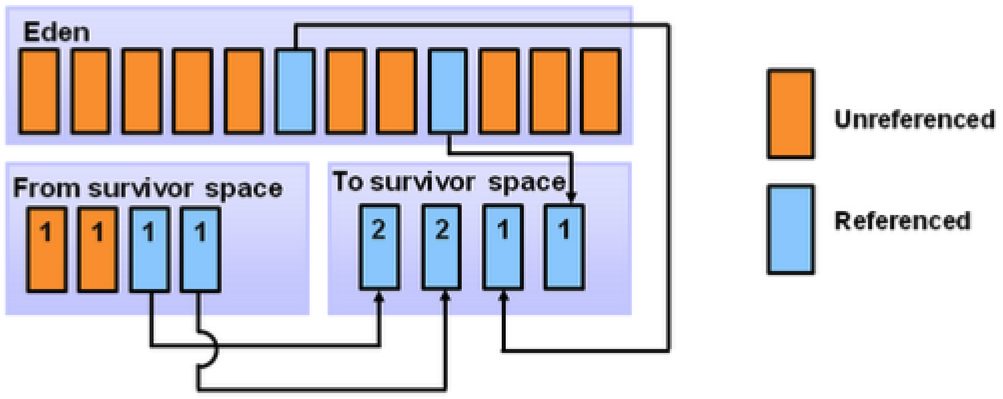
Filling the Eden Space



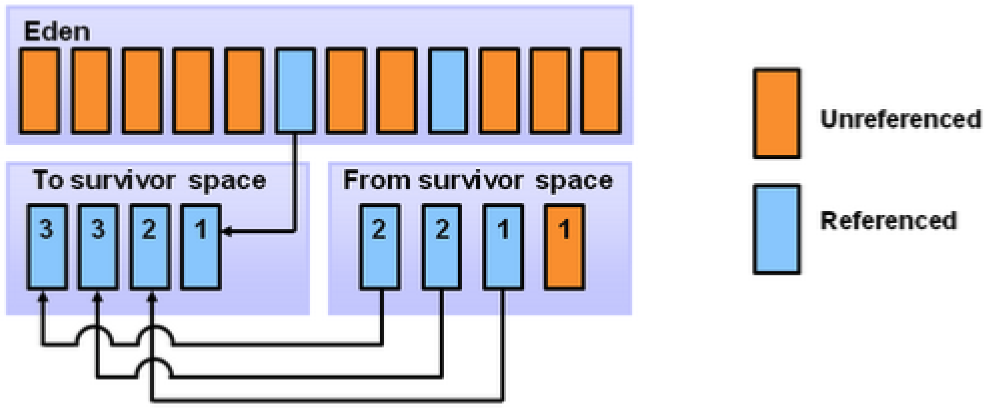
Copying Referenced Objects



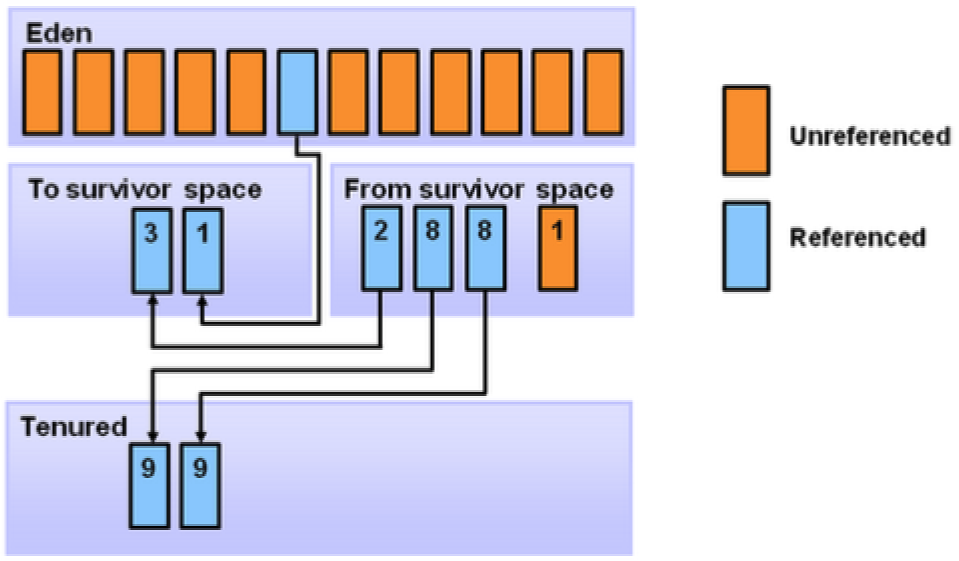
Object Aging



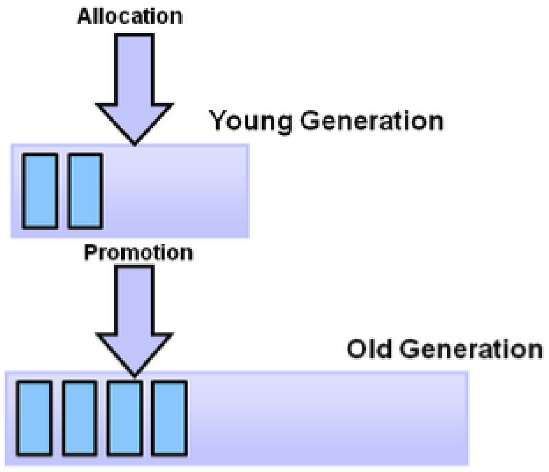
Additional Aging



Promotion



Promotion



Data Structures

Deque vs double LinkedList

- Deque is implemented in a circular array.

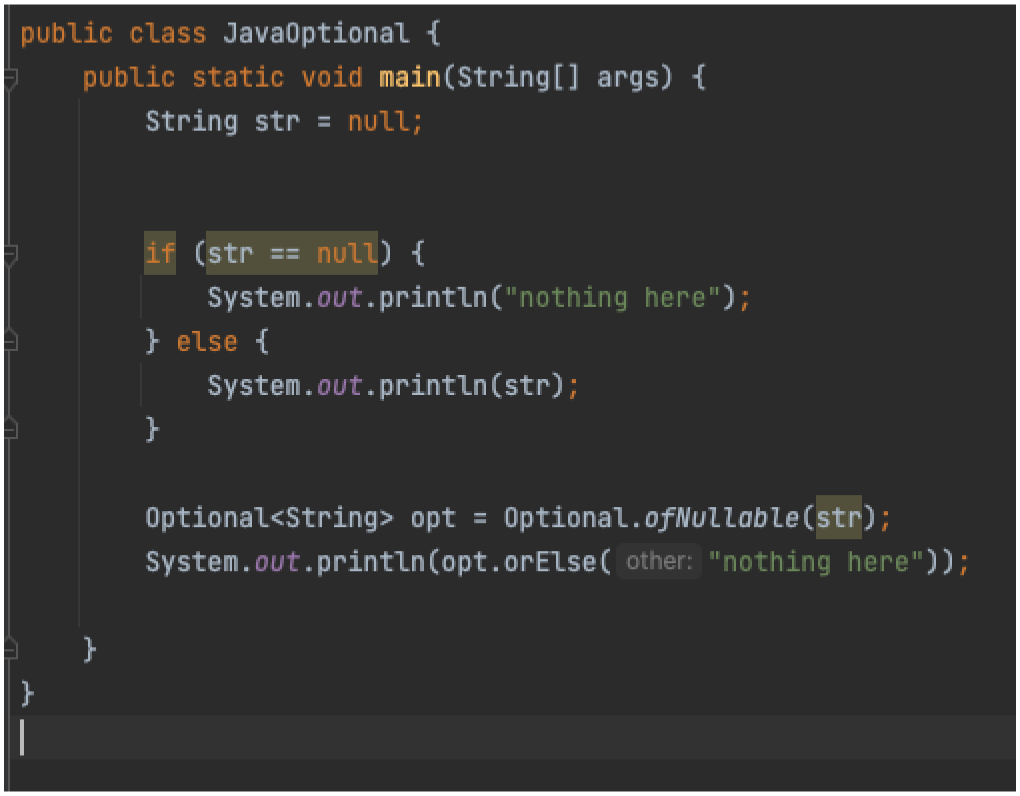
Generics

- Easier and less error-prone
- Enforce type correctness at compile time
- Without causing an extra overhead to your application.

Type Erasure

- Type erasure is a process in which the compiler replaces a generic parameter with an actual class or bridge method. In type erasure, the compiler ensures that no extra classes are created and there is no runtime overhead.
- Replace type parameters in generic type with their bound if bounded type parameters are used
- Replace type parameters in generic type with Object if unbounded type parameters are used
- Insert type casts to preserve type safety.
- Generate bridge methods to keep polymorphism in extended generic types.

Optional



```java
public class JavaOptional {
    public static void main(String[] args) {
        String str = null;


        if (str == null) {
            System.out.println("nothing here");
        } else {
            System.out.println(str);
        }

        Optional<String> opt = Optional.ofNullable(str);
        System.out.println(opt.orElse( other: "nothing here"));

    }
}
```

```
// Java program with Optional Class
import java.util.Optional;
public class OptionalDemo{
	public static void main(String[] args) {
		String[] words = new String[10];
		Optional<String> checkNull =
						Optional.ofNullable(words[5]);
		if (checkNull.isPresent()) {
			String word = words[5].toLowerCase();
			System.out.print(word);
		} else
			System.out.println("word is null");
	}
}
```

Stream APIs

A Stream should be operated on (invoking an intermediate or terminal stream operation) only once. A Stream implementation may throw IllegalStateException if it detects that the Stream is being reused.

- Stream.forEach()

```
List<Integer> values = new ArrayList<>();
for (int i = 1; i <= 100; i++) {
  values.add(i);
}
values.forEach(number -> System.out.println(number));
Stream<Integer> s = values.stream();
Stream<List<Integer>> s2 = Stream.of(values);

s.forEach(System.out::println); // will work
s.forEach(System.out::println); // throw exceptions

s2.forEach(System.out::println); //will work
s2.forEach(System.out::println); //throw exceptions.
```

- Stream.count()

count the total number of elements.

```
Stream<Integer> s = values.stream();
System.out.println(s.count()); //will print 100.
```

- Stream.allMatch()

Returns whether all elements of this stream match the provided predicate.

```
List<Integer> list = Arrays.asList(3, 4, 6, 12, 20);
boolean answer = list.stream().allMatch(n -> n % 3 == 0);
System.out.println(answer); //false.
```

- Stream.anyMatch()

Returns whether any elements of this stream match the provided predicate.

```
boolean answer = list.stream().anyMatch(n -> n % 3 == 0);
System.out.println(answer);//true
```

- ## Stream.distinct()

  Returns a stream consisting of the distinct elements (according to Object.equals(Object) of this stream.

```
List<Integer> list = Arrays.asList(1, 1, 2, 3, 3, 4, 5, 5);
list.stream().distinct().forEach(System.out::print); //12345
```

- ## Stream.limit(long maxSize)

  Return a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.

```
Stream.of(1,2,3,4,5,6,7,8,9,10).filter(i -> i%2 ==0)
.limit(2) //limit the size as two only
.forEach(i->System.out.print(i + " ");
//2 4
```

- ## Stream.skip(long maxSize)

```
Stream.of(1,2,3,4,5,6,7,8,9,10).filter(i -> i%2==0)
.skip(2)//skip the first two elements.
.forEach(i -> System.out.println(i + " ");
//6 8 10
```

- ## Stream.map()

  Returns a stream consisting of the results of applying the given function to the elements of this stream.

```
List<Integer> list = Arrays.asList(3,6,9,12,15);
list.stream().map(number -> number * 3).forEach(System.out::println);
```

```
9
18
27
36
45
```

- ## Stream.flatMap()

  Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each elements. For mapped stream is closed after its contents have been placed into this stream.

The list before flattening:
```
[[2,3,5],[7,11,13],[17,19,23]]
```
The list after flattening
```
[2,3,5,7,11,13,17,19,23]
```

- ## Difference between map() and flatMap()

map()	**flatMap()**
The function passed to map() operation returns a single value for a single input.	The function you pass to flatmap() operation returns an arbitrary number of values as the output.
One-to-one mapping occurs in map().	One too many mapping occurs in flatMap().
Only perform the mapping.	Perform mapping as well as flattening.
Produce a stream of value.	Produce a stream of stream value.
map() is used only for transformation.	flatMap() is used both for transformation and mapping.

- ## Java Method References

  Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you're using lambda expression to just referring a method, you can replace your lambda expression with method reference.