**1. Dependency:** Aggregation implies a relationship where the child **can exist independently** of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent. Example: Human and heart, heart don’t exist separate to a Human

**2. Type of Relationship:** Aggregation relation is **“has-a”** and composition is **“part-of”** relation.

**3. Type of association:**Composition is a **strong** Association whereas Aggregation is a **weak** Association.

**Aggregation**

public void setBs(List<B> b) *//To assign sportspeople*

{

this.b = b;

}

**Composition**

public void setStudentList() *//To set Students list*

{

//Create three Student objects

Student student1 = new Student("Abe");

Student student2 = new Student("lincoln");

Student student3 = new Student("Abraham");

ArrayList<Student> students = new ArrayList<Student>();

students.add(student1);

students.add(student2);

students.add(student3);

this.studentList = students;

}

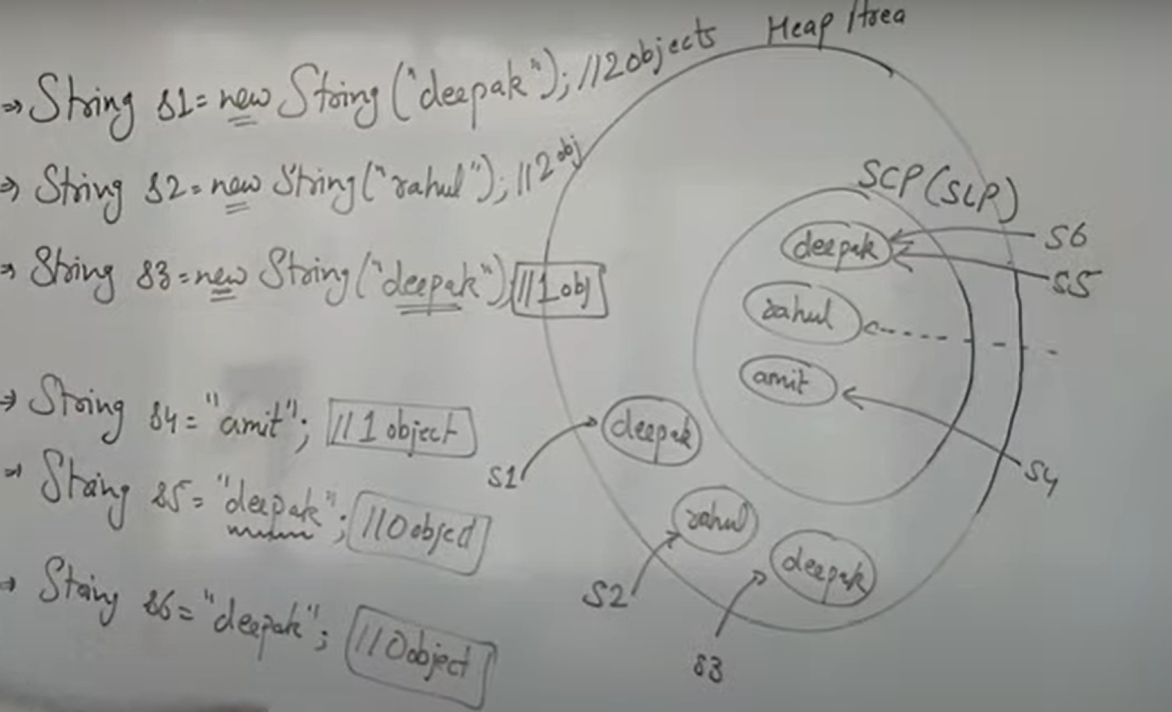
**Immutable class** in java means that once an object is created, we cannot change its content. In Java, all the [wrapper classes](https://www.geeksforgeeks.org/wrapper-classes-java/) (like Integer, Boolean, Byte, Short) and String class is immutable.

* **Make your class final :**  
  If you make your class final, no class will be able to extend it, hence will not be able override methods of this class.
* **Declare all instance variable with private and final :**  
  If you make instance variable private, no outside class will be able to access instance variables and if you make them final, you can not change it.
* **Say no to setter methods :**  
  Don’t create setter method for any instance variables, hence there will be no explicit way to change state of instance variables.
* **Initialize all variables in constructor :**  
  You can initialize variables in constructor. You need to take special care while working with mutable object. You need to do deep copy in case of immutable objects.
* **Perform cloning of mutable objects while returning from getter method:**  
  If you return clone of object from getter method, it won’t return original object, so your original object will remain intact.
* Immutable object’s hashcode won’t change, so it makes key retrieval faster as you can cache different hashcode for different keys. In case of mutable object, hashcode may be dependent on mutable fields, if any value for these field changes, it might change hashcode, so you might not be able to find your key in HashMap as hashcode is different now.
* Immutable classes are thread safe because you cannot change state of immutable objects, so even if two thread access immutable object in parallel, it won’t create any issue.

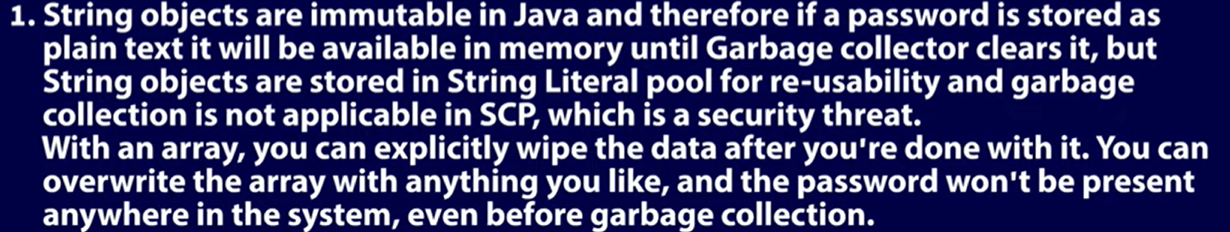
String immutable in Java is as follows-

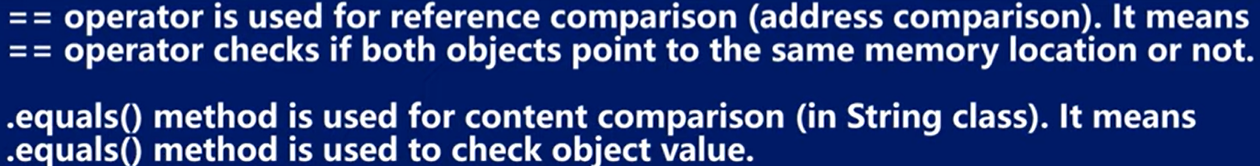
* **Security -** This is one of the major reasons for making String immutable. We need immutable string parameters for credentials, network connections, database connections. Such parameters should never be modified so these should be immutable.
* **Multithreading concurrency -** In a multithreaded application synchronization issues are resolved as Strings are immutable
* **Caching -** Since String is immutable, its hashcode is cached at the time of creation and it does not need to be calculated again. So it can be used as key in hashmap.
* **String Pool -** Since String is immutable, there is a concept of String pool in java. So strings can be reused.
* **Class loading -** Strings are used in java classloader and immutability provides security that correct class is getting loaded by Classloader.

String Constant/Literal Pool

****

**Why char Array is Better to Store Password Than String**

* 
* **If you print char array it will not show its content but string show, so we do not use string for password.**

****

****

**Unmodifiable collection**

public class UnmodifiableCollectionEG {

    public static void main(String[] args) {

        List x = getPersons();

        List unmodifiableX = Collections.unmodifiableList(x);

        System.out.println("before change unmodifiableX: " + unmodifiableX);

        x.remove(0);

        System.out.println("after change unmodifiableX: " + unmodifiableX);

    }

    private static List getPersons() {

        List list = new ArrayList(5);

        list.add(new Person("Person1", 20));

        list.add(new Person("Person2", 21));

        list.add(new Person("Person3", 22));

        list.add(new Person("Person4", 23));

        return list;

    }

}

class Person {

    public Person(String name, Integer age) {

        this.name = name;

        this.age = age;

    }

    private String name;

    private Integer age;

    //... getter - setter and toString methods here ...

}

Output

before change unmodifiableX: [Person{name=Person1, age=20}, Person{name=Person2, age=21}, Person{name=Person3, age=22}, Person{name=Person4, age=23}]

after change unmodifiableX: [Person{name=Person1, age=20}, Person{name=Person3, age=22}, Person{name=Person4, age=23}]

**Immutable collection**

public class ImmutableCollectionsEG {

    public static void main(String[] args) {

        List x = getPersons();

        List immutableX = List.copyOf(x);

        System.out.println("before change immutableX: " + immutableX);

        x.remove(0);

        System.out.println("after change immutableX: " + immutableX);

    }

    private static List getPersons() {

        List list = new ArrayList(5);

        list.add(new Person("Person1", 20));

        list.add(new Person("Person2", 21));

        list.add(new Person("Person3", 22));

        list.add(new Person("Person4", 23));

        return list;

    }

}

Output

before change immutableX: [Person{name=Person1, age=20}, Person{name=Person2, age=21}, Person{name=Person3, age=22}, Person{name=Person4, age=23}]

after change immutableX: [Person{name=Person1, age=20}, Person{name=Person2, age=21}, Person{name=Person3, age=22}, Person{name=Person4, age=23}]

**Archive Immutable collection in java 8 and previous**

List immutableX = Collections.unmodifiableList(new ArrayList(x));

Because instance variables CANNOT be overridden in Java. In Java, only methods can be overridden.

When you declare a field with the same name as an existing field in a superclass, the new field *hides* the existing field. The existing field from the superclass is still present in the subclass, and can even be used

# Auto-Widening:

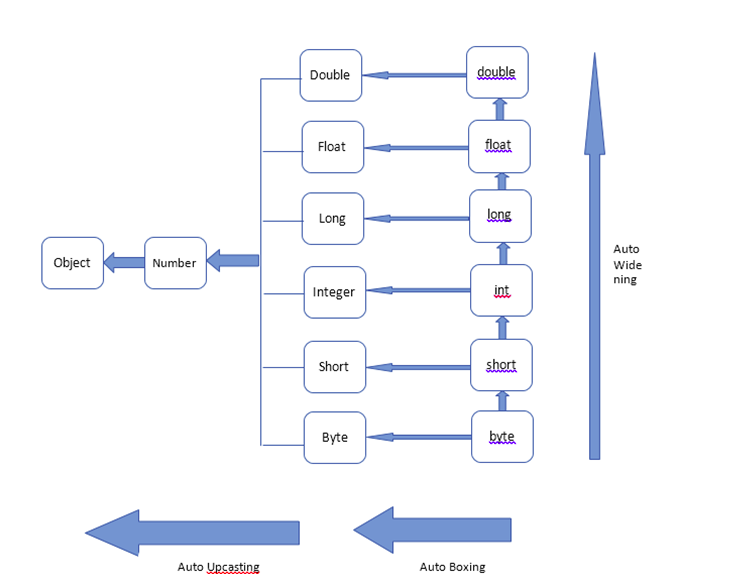
When the data is implicitly casted from small sized primitive type to big sized primitive type. This is called auto-widening. i.e The data is automatically casted from byte to short, short to int, int to long, long to float and float to double.

# ****Auto-Boxing:****

Auto-boxing occurs when primitive type is casted to corresponding wrapper class.The **wrapper class in Java** provides the mechanism to convert primitive into object and object into primitive.

# Auto-Up Casting:

Auto-up casting occurs when sub class type is casted to super class type.



pictorial representation

Now here are some of the important points you should know and also we will discuss them with examples and code.

1. If you are passing primitive data type as an argument to the method call, compiler first checks for a method definition which takes **same data type** as an argument.
2. If such method does not exist, then it checks for the method definition which takes big sized primitive data type than passed data type. i.e It tries to perform **auto-widening conversion** of passed data type.
3. If auto-widening conversion is not possible, then it checks for method definition which takes corresponding wrapper class type as an argument. i.e It tries to perform **auto-boxing conversion**.
4. If such method does not exist, then it checks for the method which takes super class type (Number or Object type) as an argument.
5. If such method also does not exist, then compiler gives compile time error.

* We can create your own custom marker interface in Java. Here is an example using custom Marker interface in Java. As we know marker [interfaces](https://www.netjstech.com/2015/05/interface-in-java.html) define a type, that can be used with [instanceof operator](https://www.netjstech.com/2017/03/instanceof-operator-in-java.html" \t "_blank) to test whether object is an instance of the specified type.
* **Abstract classes** can have constructors even when they are only called from their concrete subclasses.

**Order of execution of Initialization blocks and constructor in Java**

1. Static initialization blocks will run whenever the class is loaded first time in JVM
2. [Initialization blocks](https://www.geeksforgeeks.org/g-fact-26-the-initializer-block-in-java/) run in the same order in which they appear in the program.
3. [Instance Initialization blocks](https://www.geeksforgeeks.org/instance-initialization-block-iib-java/) are executed whenever the class is initialized and before constructors are invoked. They are typically placed above the constructors within braces.

**class** GFG {

GFG(**int** x) {

System.***out***.println("ONE argument constructor");

}

GFG() {

System.***out***.println("No argument constructor");

}

**static** {

System.***out***.println("1st static init");

}

{

System.***out***.println("1st instance init");

}

{

System.***out***.println("2nd instance init");

}

**static** {

System.***out***.println("2nd static init");

}

**public** **static** **void** main(String[] args) {

**new** GFG();

**new** GFG(8);

}

}

Output:

1st static init

2nd static init

1st instance init

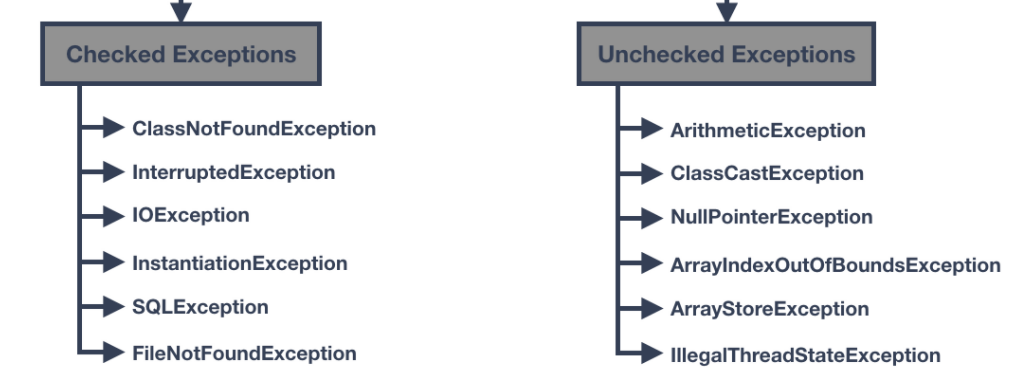
2nd instance init

No argument constructor

1st instance init

2nd instance init

ONE argument constructor

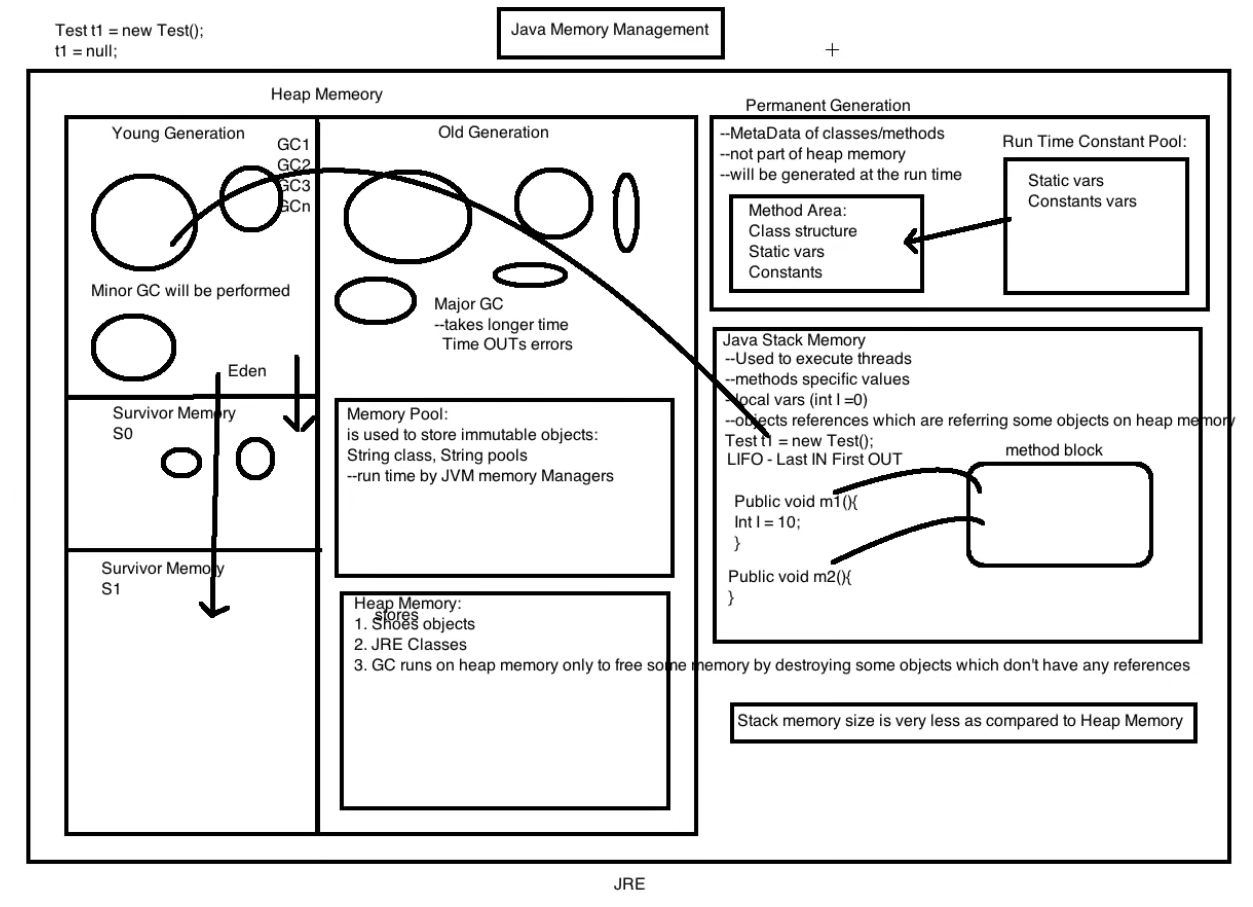
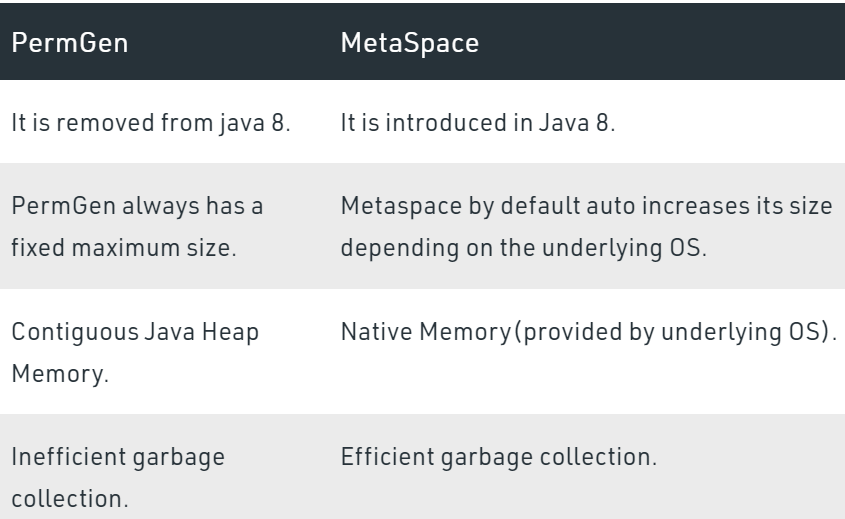
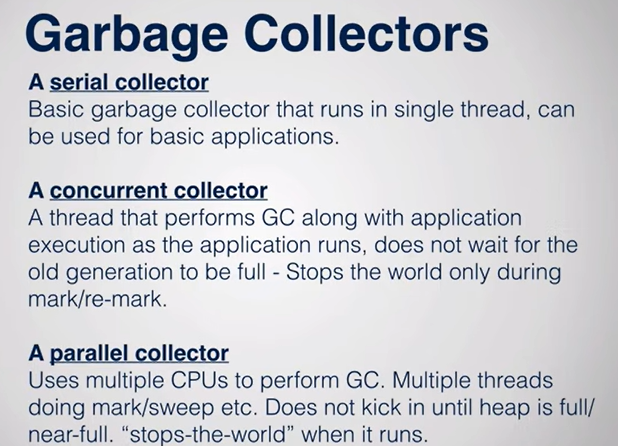
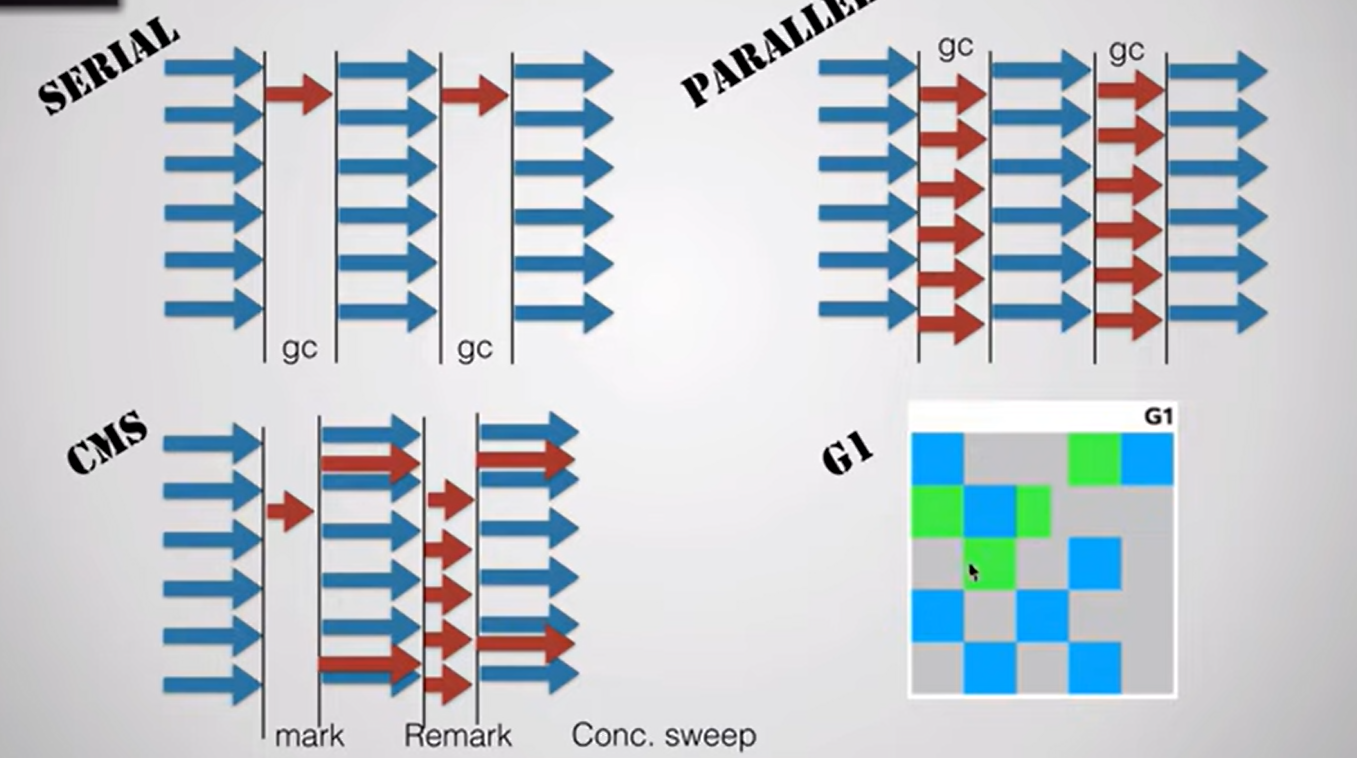
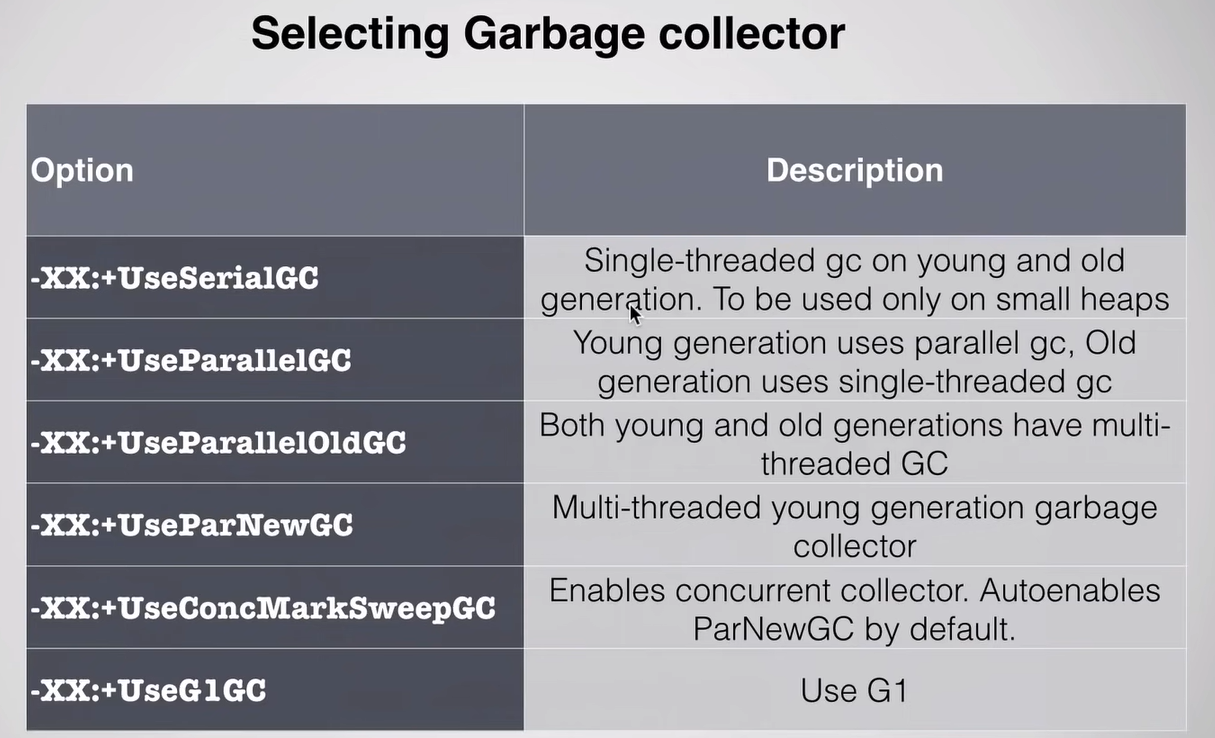
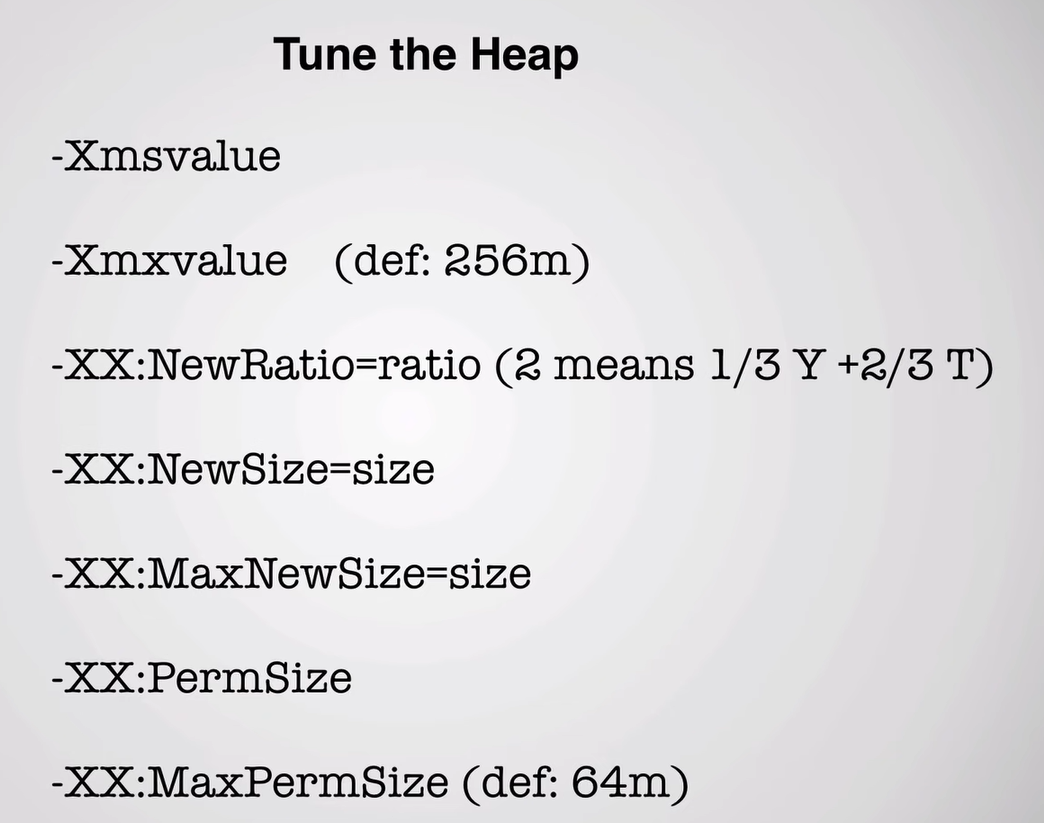
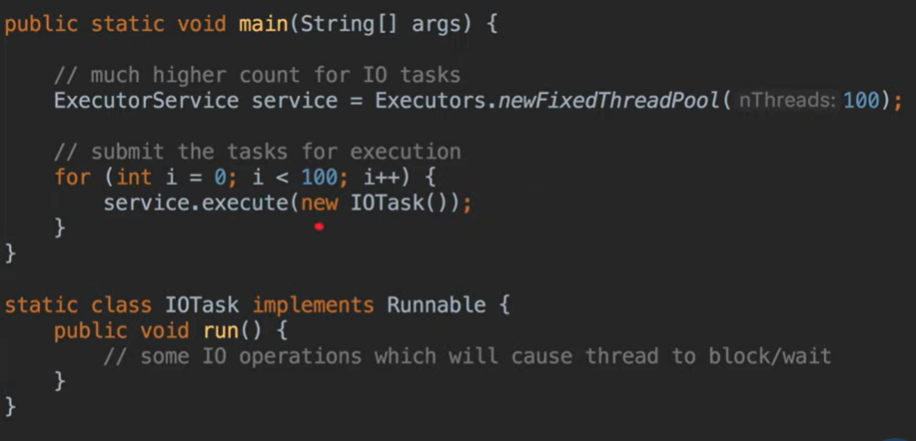
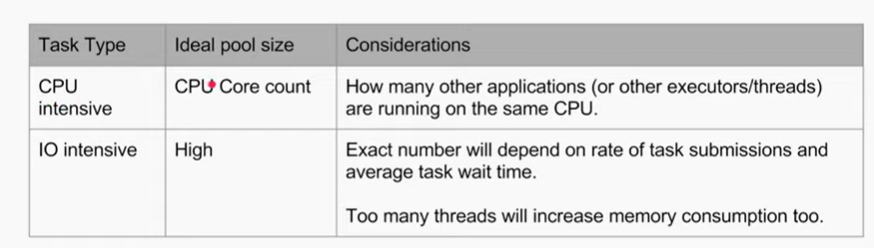
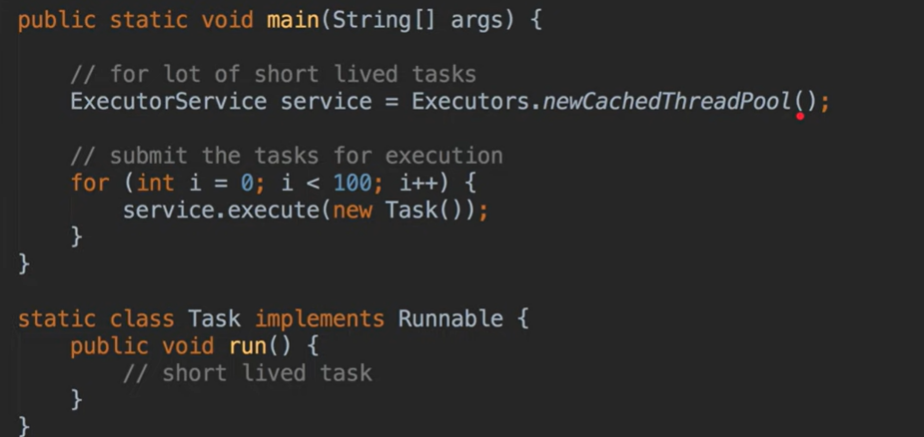
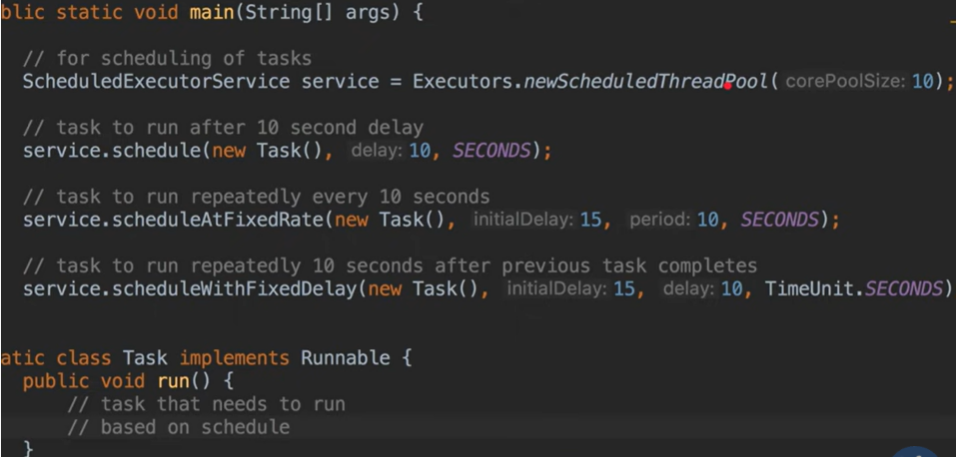
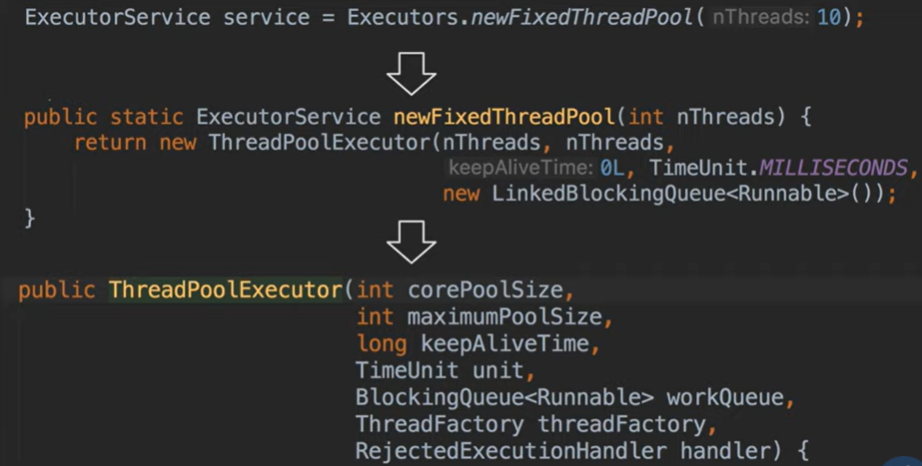
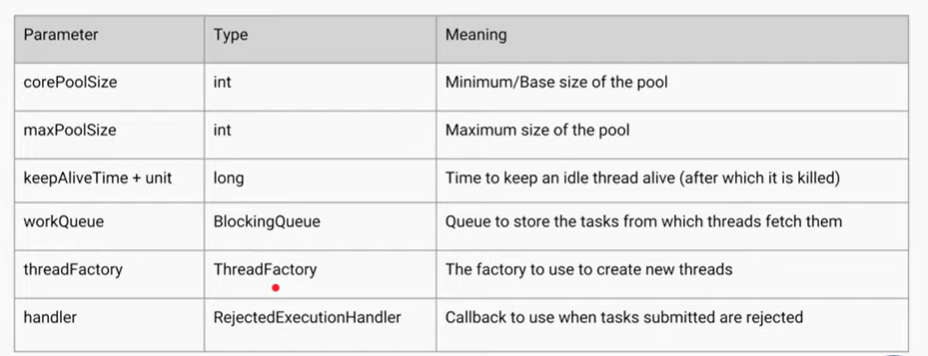
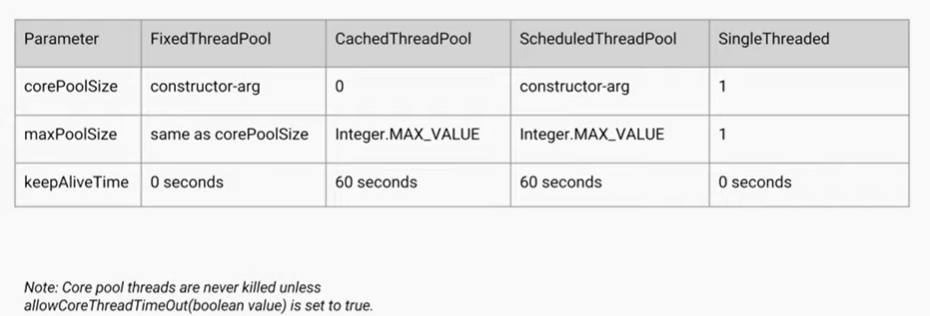
* A checked exception must be handled either by re-throwing or with a try catch block, a runtime isn’t required to be handled. An unchecked exception is a programming error and are fatal, whereas a checked exception is an exception condition within your codes logic and can be recovered or retried from.
* 
* Volatile keyword is used **to modify the value of a variable by different threads**. It is also used to make classes thread safe. It means that multiple threads can use a method and instance of the classes at the same time without any problem. The volatile keyword can be used either with primitive type or objects.
* The volatile keyword marks a variable as, well, *volatile*. By doing so, the JVM *guarantees* that each write operation's result isn't written in the local memory but rather in the main memory.

This means that any thread in the environment can access the shared variable with the newest, up-to-date value without any worry.

A similar, but not identical behavior, can be achieved with the [synchronized](https://stackabuse.com/synchronized-keyword-in-java/) keyword.

* Overriding the the hashCode() is generally necessary whenever equals() is overridden to maintain the general contract for the hashCode() method, which states that **equal objects must have equal hash codes**

<https://howtodoinjava.com/java/basics/java-hashcode-equals-methods/>

* If SuperClass does not declare an exception, then the SubClass can only declare unchecked exceptions, but not the checked exceptions.
* If SuperClass declares an exception, then the SubClass can only declare the same or child exceptions of the exception declared by the SuperClass and any new Runtime Exceptions, just not any new checked exceptions at the same level or higher.
* If SuperClass declares an exception, then the SubClass can declare without exception.
* Call to super() must be first statement in Derived(Student) Class constructor.
* If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.
* If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of Object. This, in fact, is the case. It is called *constructor chaining*.
* 
* 
* 
* 
* 
* 
* 
* 
* 
* 
* 
* 
* 
* 
* 