Object Oriented Programming (Java)

- Shashidhar Nagaral

Class.

Class is the template for the objects → logical construct.

Object is an instance of the class. → this is physical existance of a class.

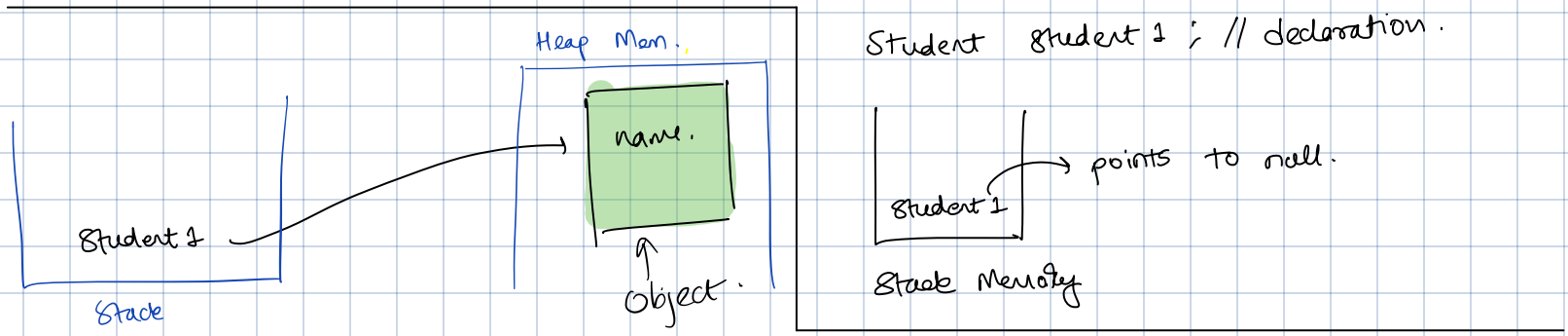
- Object
  - State : value
  - Identity : references.
  - Behaviour : methods / functions.

How to access instance variables?

```
Student student1 = new Student();
```
↑ reference variable

```
class Student {
    int rno;
    String name;
};
```
↑ instance variables

→ student1.name



Student student1; // declaration.

NOTE: Student S1; // declaration, S1 will have NULL by default.

S1 = new Student(); // created object.

⇓ dynamically allocation memory & returns the reference to it.

Dynamic Memory Allocation:

Student student1 = new Student();

Student student1 — Compile time

new Student(); — Runtime

↑ runtime means entire code is converted to byte code & it actually running

NOTE: (in Stack) Student 1 is not an object, it is a reference variable pointing to the object (in heap)

Constructors - it is special type of method.
- it is automatically called when the object is created, before the new operator completes.

Note: every class has a default constructor.

```
Box mybox1 = new Box();
```

⇒ new Box() is calling the Box() constructor

```
class Student {
    Student() {
    // default Constructor
    }
};
```

"this" Keyword: this refers to instance of the class. it is often used to differentiate b/w instance variables & parameters with same name

Why don't we use "new" Keyword for creating primitive data Types?

→ In Java, the primitive data types like int, float, char, boolean are not implemented as objects.

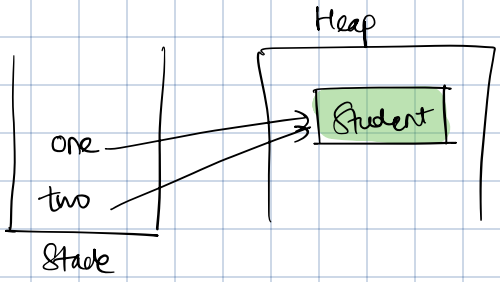
Primitives ⇒ NOT objects ⇒ Not stored in Heap but stored in Stack

Then why does java does have primitive data type?
→ To increase the performance, memory efficiency ....

Memory Allocation of "new" Keyword:

```
Student one = new Student();
Student two = one;
```

i.e one, two both are pointing to same instance of class present in the heap.



Wrapper Class:

```
int a = 10;      // a is primitive
Integer num = 10; // num is an object.
```

NOTE:

```
Integer num = new Integer(10); ✗ Not allowed from Java 9
Integer num = 10;              ✓
```

AUTOBOXING

Since Java 9, there has been feature Auto boxing that automatically converts primitive types to their corresponding class & vice versa;

Final used to define constant, (immutable) → for optimisation.

NOTE: final + primitive → we cannot update the value

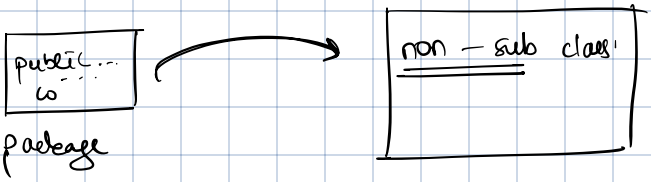
final + Non-Primitive → In objects, we cannot change the Reference but we can change the value

Packages: package are containers for classes
package is both a naming & a visibility control mechanism.

How does the java run-time system know where to look for packages that you create?

① By default, the JRT system uses the current working directory as its starting point.
② We can specify a dir. path or paths by setting the CLASSPATH environmental variables
③ You can use the -classpath option with java and javac to specify the path to your classes.

NOTE: when a package is imported, only those items within the package declared as public will be available to non-sub classes in the importing code.



Static: When a member is declared static, it can be accessed before any of its class instance are created.

most common example is main() method.
→ main() method is declared as static because it must be called before any objects exist.

→ Static method in Java is a method which belongs to the class & not to the object.

NOTE:
- A static method can access only static data.
- A static method can call other static method & cannot call a non-static method from it.

- A static method cannot refer to "this" or "super" keyword in anyway

Inner Class: Inner class act as a member to the outer class.

ex.
```
class A {                          // outer class
    private int num;
    class B {                      // inner class
        String name;
        B(String name) {
            this.name = name;
        }
        num = 10;
    }
    psvm() {
        B b = new B("a");          // ✗ this will give error bcz B is itself non static member of A and hence we can't access B without any instance of A.
        A a = new A();             // ✓
        A.B b = a.new B("a");
    }
}
```

we can access other member variables inside static inner class.

`new a.B("a");` ✗ bcz we can't access B with class

Static Inner class: Same as above code, but if the inner class is static then

① static tells that the inner class is not dependent on the instance of the outer class.

② we can't access other member variables inside static inner class.

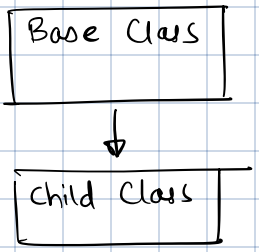
```
class A {
    static class B {
        String name;
        B(String name) {
            this.name = name;
        }
    }
    psvm
}
```

```
// main()
A.B b = new A.B();
```

A is just used to access the inner class

4 Pillars of OOP's : Inheritance, Polymorphism, Encapsulation, Abstraction.

Inheritance




```
class Parent {
    int length;
    int breadth;
}

class Child extends Parent {
    int price;
}
```

NOTE : • Java does not support multiple super class i.e only one super class for any class can be specified.

Private:

Subclass (child class) can access all of the members of its superclass, but cannot access members of the superclass that have been declared as private.

NOTE ** When a reference to a subclass object is assigned to a superclass reference variable; you will have access only to those parts of the object defined by superclass.

ex
```
class A {
    int a;
    int b;
}

class B extends A {
    int c;
    int d;
}

// main
A b_obj = new B();
```

super class reference type

subclass object

i.e b_obj.c ✗

b_obj.a ✓

i.e `SUPERCLASS ref = new SUBCLASS();`

Here ref can only access methods which are available in SUPERCLASS.

parent object

child reference type → `B a_obj = new A();` ✗ Error

this will show error bcz,

① Reference type is of child: this means parent object can access members of child class.

② Object is of Parent: but, instance is of parent class. & parent does not access to it child class.

"SUPER": whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword "super"
it can be used for:

① to call superclass constructor

② Access superclass members.

ex

child class. →

```
BoxWeight (double w, double h) {
    super(w, h);   ← constructor
    super.w;       → superclass member.
}
```

NOTE: A superclass variable can be used to reference any object derived from that class.

ex

```
class Box {
    double w;
    double h;
}
```

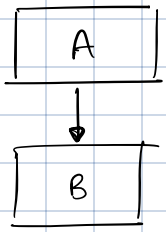
```
class BoxWeight extends Box {
    double weight;

    BoxWeight (BoxWeight ob) {
        super(ob);
        weight = ob.weight;
    }
}
```

we are passing object of type BoxWeight to the super of type Box

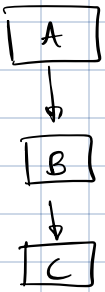
NOTE: If super() is not used in subclass's constructor, then default or parameterless constructor of each superclass will be executed.

Inheritance types

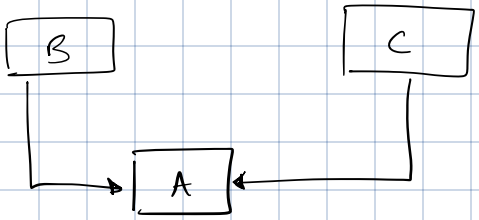
① Single Inheritance



② Multilevel Inheritance

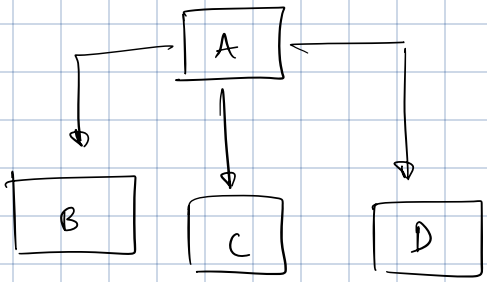


③ Multiple Inheritance ✗

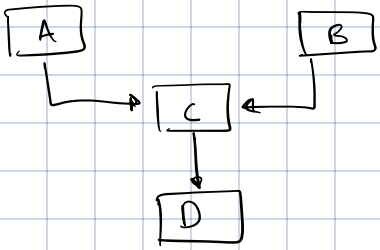


NOTE: Java does not support Multiple Inheritance

④ Hierarchical inheritance



✗ ⑤ Hybrid Inheritance: Single + Multiple Inheritance



NOTE: We can achieve multiple inheritance through INTERFACE

Polymorphism: polymorphism allows objects of different classes to be treated as objects of a common superclass during runtime.

Types of Polymorphism:

① Compile time / Static Polymorphism (Method Overloading):

- method overloading in Java allows a class to have multiple method with same name but different parameter list

Rules:

① Method names must be same.
② you can't overload a method just by changing its return type
③ The parameter lists must be different in terms of the number of parameter &/or their types.

② Runtime / dynamic polymorphism : (method overriding)

- Reference type is superclass
  Obj type is subclass.

NOTE: **Reference variable type** tells what all things are accessible

type of object tells which one of those to be seen.

```
Class Shape {
    void area() {
    }
}
Class Square extends Shape {
    void area() {   ← overrided the area() method of its superclass.
    }
}
```

How does overriding work?

```
Parent obj = new Child();
```

Here, which method will be called depends on the type of object,
this is known as **Upcasting**.

- Upcasting is the process of converting a reference variable of a subclass type to a reference variable of the superclass type.
- Upcasting is SAFE & implicit conversion because a subclass object inherently contains all the attributes & behaviors of its subclass.

Downcasting:
- Converting a reference variables of a superclass type to a reference variable of a subclass type.
- Not implicit, & requires an explicit cast.

- Downcasting is needed when you want to access the specific methods & fields defined in the subclass.

ex.
```
class Animal {
    void makeSound() {
    }
}
Class Dog extends Animal {
    void makeSound() {
    }
    void fetch() {
    }
}
```

```
// psvm
Animal animal = new Dog();
animal.fetch()  → since Ref type is Animal, we can't have access to fetch method.
```

soln: Downcasting

```
if (animal instanceof Dog) {
    Dog dog = (Dog) animal;
    dog.fetch();
}
```

How does java determines which members to be seen?

→

Dynamic Method Dispatch: also known as late binding, it allows the method invocation to be determined at runtime rather compile-time.

NOTE: Private, final and static methods and variables uses static binding & bonded by compiler while overridden methods are bonded during runtime based upon Type of runtime object.

Encapsulation: bundles the data (attributes) & methods (behaviors) that operate on the data within single unit called a class.

- main idea behind encapsulation is to hide the internal details of an object; this is often refered to as data hiding
- encapsulation is achieved using access modifiers & getter & setter methods.

Access Modifiers: control the visibility & accessibility of class members (fields & method)

Getter & setters: allow controlled access to the internal state of an object.

Abstraction: : it hides the internal implementation & shows only essential functionality to the user.

It can be achieved through INTERFACE & ABSTRACT classes.

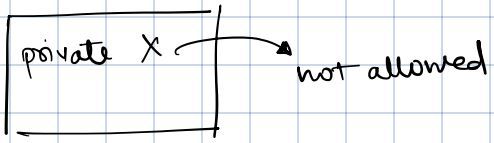
ex. CAR: pedal, if we press it → car speed will reduce
BUT HOW? this is abstracted to us.

- Increase security & confidentiality

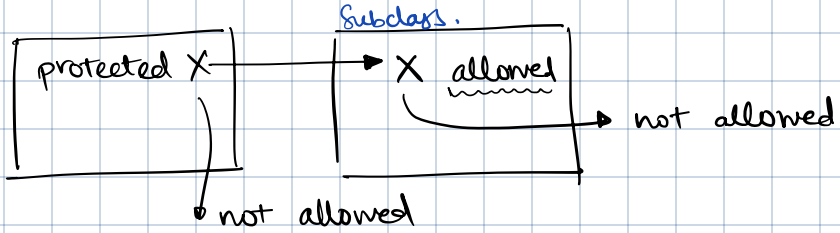
NOTE: Encapsulation : Implementation level issue.
Abstraction : Design level issue.

Access Control :

for sensitive data ← **private:** - not accessible outside the class.
- to manipulate we use getter & setter method.



outside package but only within subclass ← **protected:** - this makes the member to be accessed from the subclass also.



use when we want to hide from outside package ← **default / package private:** not accessible outside the package

we want to use everywhere ← **public:** Accessible everywhere

Access Modifier:	Private	Default / no-access (Package Private)	Protected	Public
Inside Class	✓	✓	✓	✓
Same package Class	✗	✓	✓	✓
Same package Subclass	✗	✓	✓	✓
Other package subclass	✗	✗	✓	✓
Other package class	✗	✗	✗	✓

Abstract Class: is a class that cannot be instantiated on its own.
it's meant to be done by subclass of by other class.

- It serves as a blueprint for creating derived classes

- Abstract classes are often used to define a common base for a group of related classes

abstract type name (parameter-list);

↳ these methods are also referred as subclass's responsibility.

NOTE: • Any class that contains one & more abstract methods must also be declared abstract.
• There can be no objects of an abstract class.

• You cannot declare abstract constructors, & abstract static methods.
• You can declare static method.

• Abstracts class can have concrete method.

why?
bcz if it were allowed then it would mean that we are calling an empty method (abstract) through classname bcz it is static.

Abstract Classes can have public constructor but still they cannot be called
then what is need to constructor inside abstracti class?
→ ① Abstract constructors will frequently be used to enforce class constraints & invariants such as minimum fields required to setup the class.

```
public abstract class Parent {
    // can we make class final and abstract both?
    // -> NO, since final doesn't allow extension of class, whereas abstract tells use to extend
    // and override method, both contradict
    abstract void run();

    // abstract static void build();// abstract and static is not possible because they contradict
    // each other, what I mean is
    // abstract tells the class to override and static methods cannot be overridden since they are
global to class

    private int x;

    public Parent(int x) {
        this.x = x;
    }

    // public abstract Parent(); // abstract constructor are not allowed

    // abstract class can have non-abstract methods (concrete methods) which can be called using
subclass
    void greeting() {
        System.out.println("Good Morning! This is parent.");
    }
    public int getX() {
        return x;
    }

    // NOTE: we cannot achieve multiple inheritance through abstract classes, why?
    // -> bcz, abstract class still allows concrete methods, which may cause ambiguity
}
```

```
public class Child extends Parent{

    public Child(int x1) {
        super(x1); // we can call the constructor of through child class

    }

    @Override
    void run() {
        System.out.println("Child is running");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Parent child = new Child(20);
        child.run(); // -> Child is running
        System.out.println(child.getX()); // -> 20

        // Parent parent = new Parent(); // cannot instantiate an abstract since it has undefined (only
declared) methods

        // But, only way to create object of parent is by overriding the methods;
        Parent parent = new Parent(10) {
            // overriding the run method while creating the object of the parent
            @Override
            void run() {
                System.out.println("Parent is running");
            }
        };
        parent.run(); // -> Parent is running
    }
}
```

Interface: is a reference type that defines a contract of methods that implementing class must adhere to

→ interface can have only abstract methods.
from java 8, it can have static & default method also ; why ? → backward compatibility, (future proof)

→ variables declared are static & final.

NOTE: Abstract class can provide the implementation of interface.
Interface can't provide the implementation of abstract class.

→ An interface can extend another Java interface only
→ Members of interface are by default public

```java
public class Sphere implements SolidShape {
    int r;

    public Sphere(int r) {
        this.r = r;
    }

    @Override
    public double volume() {
        return (4/3) * Shape.PI * Math.pow(this.r, 3);
    }

    @Override
    public double area() {
        return Shape.PI * Math.pow(this.r, 2);
    }
}
```

```java
public interface Shape {
  // static void fun(); // static interface method should always have a body

  public double area();

  double PI = Math.PI; // variable in interface are by default final and static

}
```

```java
// when we extend an interface then child interface will inherit all the parent abstract methods and variables

public interface SolidShape extends Shape{
    public double volume();
}
```

```java
public class Circle implements Shape, Color{
    private int r;
    public Circle(int r) {
        this.r = r;
    }

    @Override
    public double area() {
        return Shape.PI * Math.pow(this.r, 2);
    }
    @Override
    public void paint(COLORS color) {
        System.out.println("Circle is painted with " +color+" color");
    }
}
```

```java
public interface Color {
    enum COLORS{
        RED,
        ORANGE
    };

    public void paint(COLORS color);
    public default void greet() {
        System.out.println("this is color interface");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Circle circle = new Circle(10);
        System.out.println((circle.area())); // -> 314.1592653589793
        circle.paint(Color.COLORS.RED); // -> Circle is painted with RED color

        SolidShape sphere = new Sphere(15);
        System.out.println((sphere.area())); // -> 706.8583470577034
        System.out.println((sphere.volume())); // -> 10602.875205865552
    }
}
```

Java 8 and 9 features:

① Default Method (Java 8):

- Before Java 8, interface can have only abstract method. And all child class has to provide has to provide abstract method implementation.

  problem: what if we add new abstract method in the interface which is implemented by many classes!

  → we need to implement those in all those classes, even though the implementation is same.

  Solution:

  - we will allow default method implementation in the interface itself
  - all the classes can override these implementation or use them directly

  How they found this problem!

  → Stream() feature was introduced in Java 8.

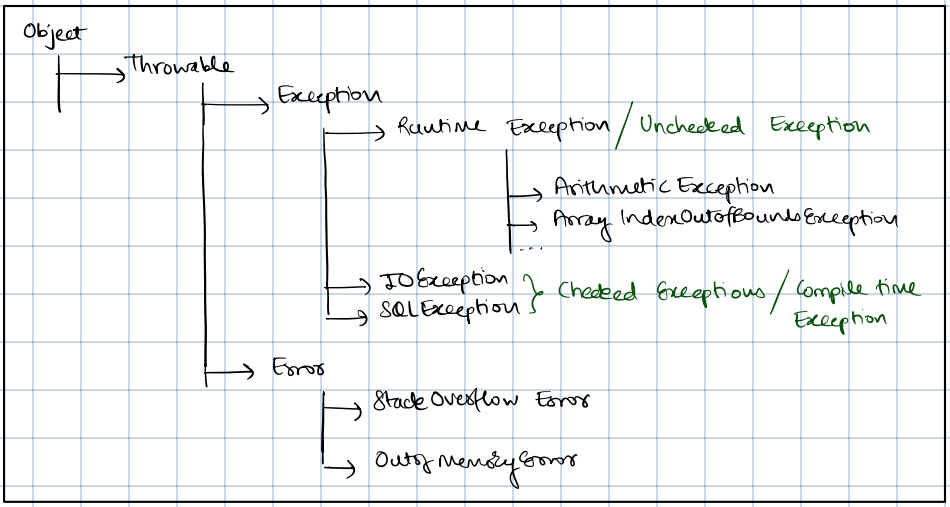
  → this is added into the Collection interface.

  → And around 200 classes implements the Collection interface.

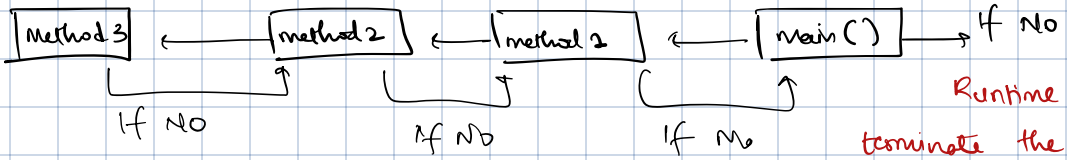
Exception Handling? is an event that disrupts the normal flow of a pgm's execution.

→ It creates the Exception Object, which contain information about the Error like
  → It type of Exception & Message
  → Stack Trace etc.

→ Runtime System use this Exception object & find the class which can handle it.



Can handle Except?



Runtime system will terminate the program abruptly & print Stack Trace.

Java exception handling is managed via five keywords:

1. **Try:** Program statement that you think can raise exception are contained within a try block.
   ex. file handling
2. **Catch:** the exception occured within try block are thrown, and to handle those we have catch block.
3. **finally:** the code which will be executed irrespect of whether the exception is thrown or not.
   ex. closing the file.
4. **Throw:** this is used to manually generate some exceptions.
5. **throws:** any exception that is thrown out of a method must be specified as such by a throws clause.

```
try {
    // block of code to monitor for errors
    // The code you think can raise an exception
} catch ( ExceptionType1 exOb ) {
    // exception handler for ExceptionType1
} catch ( ExceptionType2 exOb ) {
    // exception handler for ExceptionType2
}
//optional
finally {
    // block of a code to be executed after try block ends.
}
```

Sub class

Super class

Custom Exceptions: we can create our own custom exceptions by extending the 'Exception' & 'RuntimeException' classes

```
class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            throw new CustomException("This is a custom exception.");
        } catch (CustomException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

NOTE: when you perform the operation a/b where a is a double & b is 0, in floating-point arithmetic the result is considered mathematically undefined & is represented as "infinity". This is bcz floating-point arithmetic follows IEEE 754 Standard which defines special values like positive & negative infinity to handle such cases.

on the other hand, in integer arithmetic, divide by zero is not mathematically possible & is considered an error.

Object Cloning:

① **Copy Constructor**: it is a user-defined constructor that you implement in your class. You have full control over how the copying process is implemented. You can choose whether to perform a shallow & deep copy, or any other custom copy logic you need.

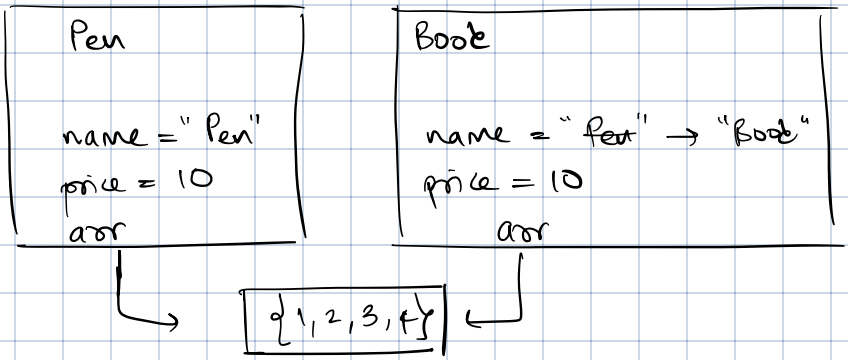
② **Cloning**: The clone method is a method provided by Java runtime in the Object class. You can override it to customize the copying behavior for your class.

If you want to achieve a deep copy, you need to implement the Cloneable interface & provide your own deep copying logic.

SHALLOW COPY.

```java
public class Product implements Cloneable{
    String name;
    int price;
    int arr[] = new int[]{1, 2, 3, 4};

    public Product(String name, int price) {
        this.name = name;
        this.price = price;
    }

    // copy constructor
//    public Product(Product other) {
//        this.name = other.name;
//        this.price = other.price;
//    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        // this is shallow copy
        return super.clone();
    }
}
```

```java
public class ShallowCopy {
    public static void main(String[] args) throws CloneNotSupportedException {
        Product pen = new Product("pen", 10);
        Product book = (Product) pen.clone();

        System.out.println(pen.name);
        System.out.println(book.name);

        // changing the name of pen object
        book.name = "book";

        // name attribute changed in book object and it has not reflected in pen object
        System.out.println(pen.name);
        System.out.println(book.name);

        System.out.println(Arrays.toString(pen.arr));
        System.out.println(Arrays.toString(book.arr));
        pen.arr[0] = 20;

        // BUT, in non-primitive data type like arr the changes in one object reflected in copied (book) object also
        // this is called SHALLOW COPY
        System.out.println(Arrays.toString(pen.arr));
        System.out.println(Arrays.toString(book.arr));
    }
}
```



both the object were still refering the same old non-primitive member.

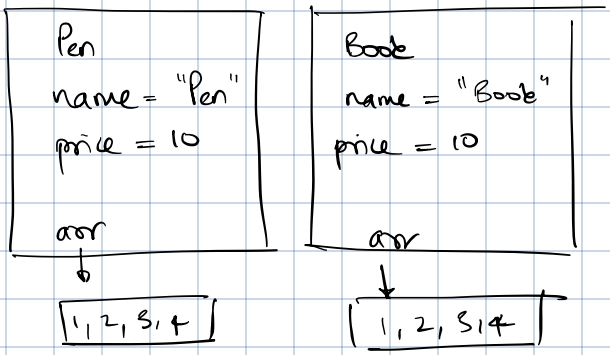
i.e shallow copy. does not create new instance of the non-primitive member.

DEEP COPY

```java
public class DeepProduct implements Cloneable{
    String name;
    int price;
    int arr[] = new int[]{1, 2, 3, 4};

    public DeepProduct(String name, int price) {
        this.name = name;
        this.price = price;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        // below is deep copy steps

        // 1. first we copy the object in another reference variable

        DeepProduct deepProduct = (DeepProduct) super.clone();

        deepProduct.arr = new int[this.arr.length]; // recreating new array for copied object

        // copying all the elements from old object arr
        for(int i=0; i<this.arr.length; i++) {
            deepProduct.arr[i] = this.arr[i];
        }

        return deepProduct;
    }
}
```

```java
public class DeepCopy {
    public static void main(String[] args) throws CloneNotSupportedException {
        DeepProduct pen = new DeepProduct("pen", 10);
        DeepProduct book = (DeepProduct) pen.clone();

        System.out.println(pen.name);
        System.out.println(book.name);

        // changing the name of pen object
        book.name = "book";

        // name attribute changed in book object and it has not reflected in pen object
        System.out.println(pen.name);
        System.out.println(book.name);

        System.out.println(Arrays.toString(pen.arr));
        System.out.println(Arrays.toString(book.arr));
        pen.arr[0] = 20;

        // Even after changing the value in arr member of pen object, there is
        // no change in the arr member of book object
        // this is DEEP COPY
        System.out.println(Arrays.toString(pen.arr));
        System.out.println(Arrays.toString(book.arr));
    }
}
```

→ we are first cloning the object to other new object, then we are creating new instance of arr to it filling the values



Enums:
- it is a class type.
- we can give them constructor, instance variables & methods, & even implement interfaces

Properties:

① An enumeration is a list of named constants (an object of type enum).

② Every enum constant is always implicitly public static final.
Since it is static, we can access it by using the enum name.
Since it is final, we can't create child enums.

```
enum Color {
    RED,
    BLUE
    GREEN
}
```

⟷

```
class Color {
    public static final Color Red = new Color();
    public static final Color Blue = new Color();
    public static final Color Green = new Color();
}
```

NOTE: - All enums implicitly extend java.lang.Enum class. As a class can only extend only one parent in Java, so an enum cannot extend anything else.

- toString() is overridden in java.lang.Enum Class which returns the enum constant name.

values(), ordinal(), & valueOf()

- values(): returns all values present inside enum. (array of all enum const)
- ordinal(): returns the index of position of the enum constant where it was declared inside enum.

- valueOf(): returns the enum constant of specified String value.

NOTE: - the constructor is executed seperatively for each enum constant at the time of enum class loading.
- we can't invoke invoke constructor directly.

NOTE: - Enum can contain concrete method.

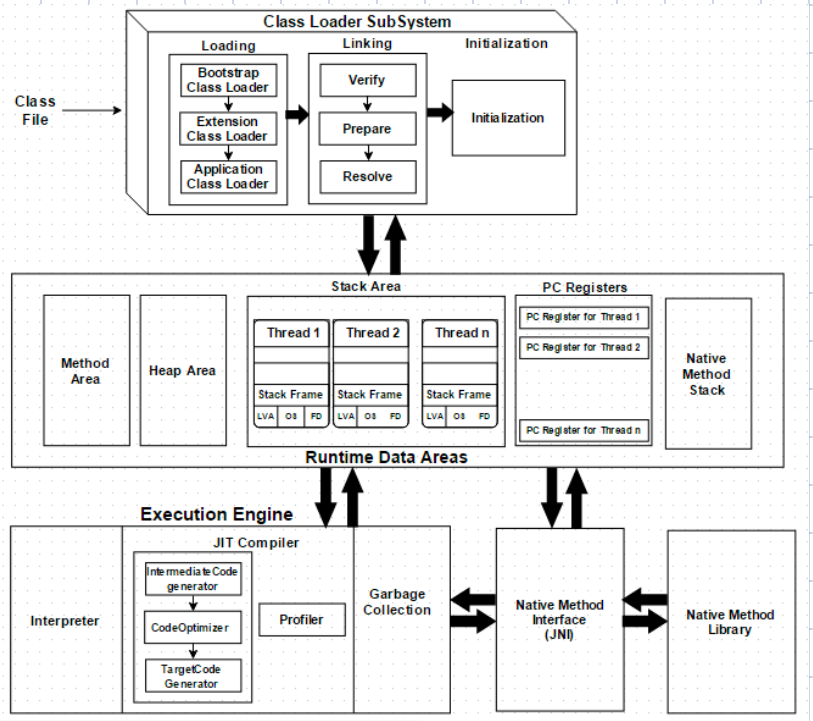
- Enum can have abstract methods, but they need to be defined for each constant.

```java
package l_enums;

public enum Color {
    RED(1) {

        // abstract methods are overridden and defined
        @Override
        public String greet() {
            return "Hey, this is " + this.name() + "color";
        }
    }, ORANGE(2) {
        @Override
        public String greet() {
            return "My color is"+this.name();
        }
    }, BLUE(3) {
        @Override
        public String greet() {
            return "I'm am of sky color";
        }
    };

    // enums can have variables
    int value;

    // constructor cannot be public

    // the constructor is called for all the constants present inside
    Color(int value) {
        System.out.println("Constructor for color" + value);
        this.value = value;
    }

    // enums can have concrete method
    public int getValue() {
        return value;
    }

    // enums can also have abstract methods, but then each instance of enum class must implement this method
    public abstract String greet();

}
```

JVM Architecture :

What is JVM ?

JVM (Java Virtual Machine) : It is an engine that provide runtime environment to lunch the Java application and responsible for converting the byte code (.class file) which generated by compiling the (.java file). JVM is a part of Java Runtime Environment(JRE).



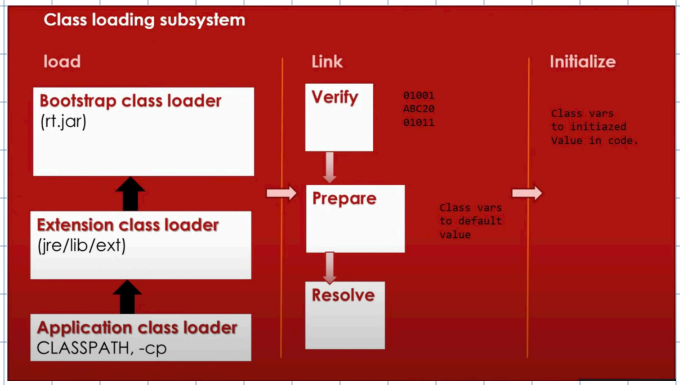
JVM is divided into three main subsystems:

1. ClassLoader
2. Runtime Data Area (Memory Area)
3. Execution Engine

1. ClassLoader:

The ClassLoader is a part of the Java Runtime Environment (JRE) that dynamically loads Java class files from file system, network or any other source into the Java Virtual Machine. It have three main phases :

i. Loading
ii. Linking
iii. Initialization



Loading: The Class loader reads the “.class” file, generate the corresponding binary data and save it in the method area. For each “.class” file, JVM stores the following information in the method area.

Linking: Performs verification, preparation, and (optionally) resolution.

Initialization: In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in the class hierarchy.

2. Runtime Data Area :

2.1 Method Area : The Java Virtual Machine has a method area that is shared among all Java Virtual Machine threads. The method area is analogous to the storage area for compiled code of a conventional language or analogous to the "text" segment in an operating system process. It stores per-class structures such as the run-time constant pool, field and method data, and the code for methods and constructors, including the special methods used in class and instance initialization and interface initialisation. If the memory in the method area cannot be made available to satisfy an allocation request, the Java Virtual Machine throws an OutOfMemoryError.

2.2 Heap : The Java Virtual Machine has a heap that is shared among all Java Virtual Machine threads. The heap is the run-time data area from which memory for all class instances and arrays is allocated. The heap is created on virtual machine start-up. Heap storage for objects is reclaimed by an automatic storage management system (known as a garbage collector); objects are never explicitly deallocated. The Java Virtual Machine assumes no particular type of automatic storage management system, and the storage management technique may be chosen according to the implementor's system requirements. The heap may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger heap becomes unnecessary. The memory for the heap does not need to be contiguous.

A Java Virtual Machine implementation may provide the programmer or the user control over the initial size of the heap, as well as, if the heap can be dynamically expanded or contracted, control over the maximum and minimum heap size.

The following exceptional condition is associated with the heap:

If a computation requires more heap than can be made available by the automatic storage management system, the Java Virtual Machine throws an OutOfMemoryError.

2.3 Stack :

Each Java Virtual Machine thread has a private Java Virtual Machine stack, created at the same time as the thread. A Java Virtual Machine stack stores frames . A Java Virtual Machine stack is analogous to the stack of a conventional language such as C: it holds local variables and partial results, and plays a part in method invocation and return. Because the Java Virtual Machine stack is never manipulated directly except to push and pop frames, frames may be heap allocated. The memory for a Java Virtual Machine stack does not need to be contiguous.

In the First Edition of The Java® Virtual Machine Specification, the Java Virtual Machine stack was known as the Java stack. This specification permits Java Virtual Machine stacks either to be of a fixed size or to dynamically expand and contract as required by the computation. If the Java Virtual Machine stacks are of a fixed size, the size of each Java Virtual Machine stack may be chosen independently when that stack is created.

A Java Virtual Machine implementation may provide the programmer or the user control over the initial size of Java Virtual Machine stacks, as well as, in the case of dynamically expanding or contracting Java Virtual Machine stacks, control over the maximum and minimum sizes.

The following exceptional conditions are associated with Java Virtual Machine stacks:

- If the computation in a thread requires a larger Java Virtual Machine stack than is permitted, the Java Virtual Machine throws a StackOverflowError.

If Java Virtual Machine stacks can be dynamically expanded, and expansion is attempted but insufficient memory can be made available to effect the expansion, or if insufficient memory can be made available to create the initial Java Virtual Machine stack for a new thread, the Java Virtual Machine throws an OutOfMemoryError.

2.4 PC Register : It stores the address of the of the instruction that currently executed.

2.5 Native Method Stack : Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

3. Execution Engine :

Execution Engine handles this by executing the code present in each class.However, before executing the program, the bytecode needs to be converted into machine language instructions. The JVM can use an interpreter or a JIT compiler for the execution engine.

No alt text provided for this image

It can be classified into three parts:

3.1 Interpreter : The interpreter reads and executes the bytecode instructions line by line. Due to the line by line execution, the interpreter is comparatively slower.

Another disadvantage of the interpreter is that when a method is called multiple times, every time a new interpretation is required.

3.2 Just-In-Time Compiler(JIT) : It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, thus efficiency is improved.

3.3 Garbage Collector: It destroys un-referenced objects. For more on Garbage Collector, refer Garbage Collector.

Java Native Interface (JNI) :

It is an interface that interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

Native Method Libraries :

It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine

https://medium.com/@kiranchowdhary/java-memory-management-and-garbage-collection-e29d3c313d05

Java Memory Management & Garbage Collection:

JVM divides memory into Stack & heap memory, both Stack & Heap are created by JVM and stored in RAM (primary memory).

① Stack Memory:

- Store temporary variables & seperate memory block for methods.
- Store Primitives data types.
- Store References of the heap objects.
  - ① Strong ref
  - ② Weak ref.
    - Soft ref.
- Each thread has its own Stack memory
- When a Stack memory goes full, its throws. "java.lang.StackOverflowError".

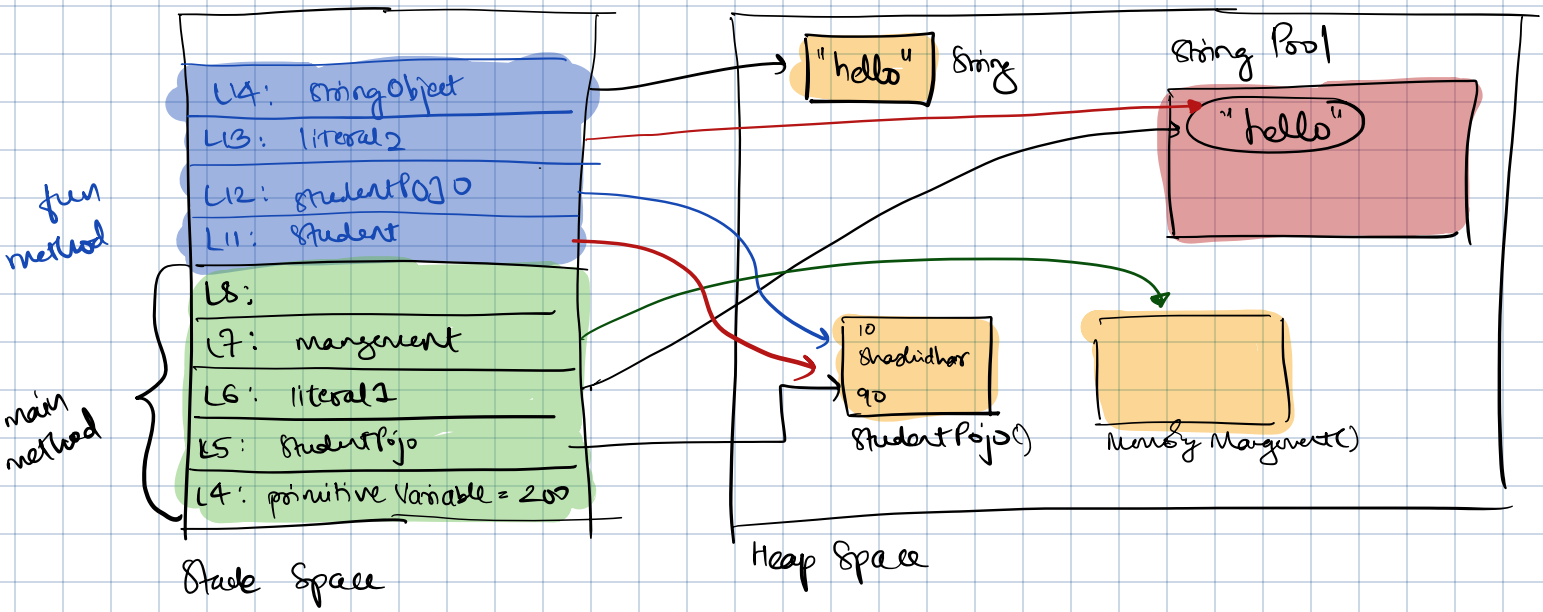
Key features of Stack Memory:

- Access to this memory is fast when compared to heap memory
- This memory is thread safe, as each thread operates in its own Stack.

② Heap Memory:

- used for the dynamically memory allocation of java objects & JRE classes at runtime.
- New objects are always created in heap space, & the references to these objects are stored in stack memory.
- This memory in contrast to stack, isn't automatically deallocated. It needs Garbage Collector to free up unused objects so as to keep the efficiency of the memory usage
- Unlike Stack, heap isn't thread safe & needs to be guarded by properly synchronizing the code.

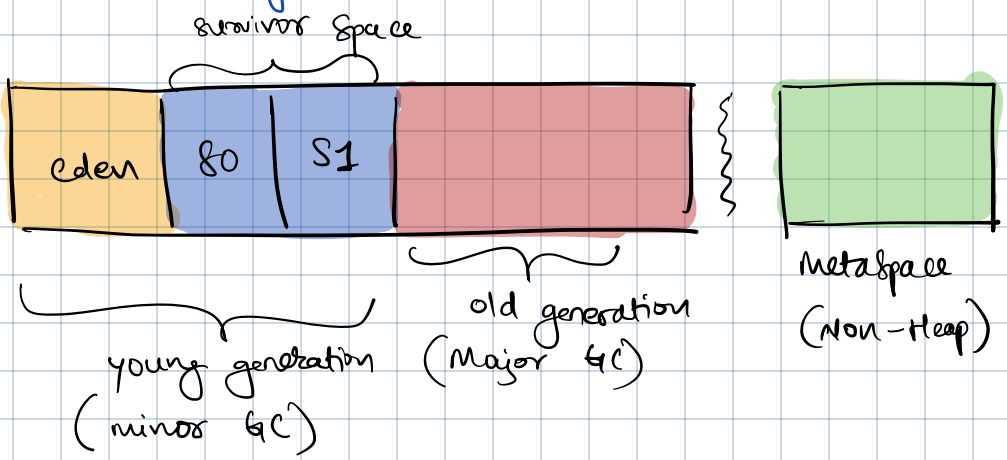
```java
public class MemoryManagement {

    public static void main(String[] args) {
        int primitiveVariable = 200;
        StudentPOJO studentPOJO = new StudentPOJO(10, "Shashidhar", 90);
        String literal1 = "hello";
        MemoryManagement management = new MemoryManagement();
        management.fun(studentPOJO);
    }

    void fun(StudentPOJO student) {
        StudentPOJO studentPOJO = student;
        String literal2 = "hello";
        String stringObject = new String("hello");
    }
}
```



What is Automatic Garbage Collection?

→ Automatic garbage collection is the process of looking at heap memory, identifying objects are in use & deleting the unreferenced objects.
An unused / unreferenced object, is no longer referenced by any part of your program, so the memory used by an unreferenced object can be reclaimed.

JVM Generations: Young Generation, Old Generation, Metaspace.



Young Generation: is where all new objects are allocated & aged

→ When Young Generation fills up, this causes a minor garbage collection.

→ A young generation full of dead objects is allocated very quickly. some surviving objects are aged & eventually move to the old generation.

NOTE: Minor GC & Major GC are "Stop The World" → all the applicatn threads are stopped until the operation completes.

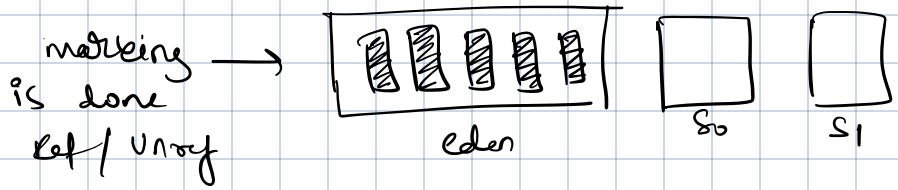
Old Generation: used to store long surviving objects, when age is met > threshold, objects gets moved to the old generation.

The old generation are collected. when filled up, and this is called major Garbage Collection.

Step 1: Any new object are allocated first to the eden space. Both the survivor start out empty.

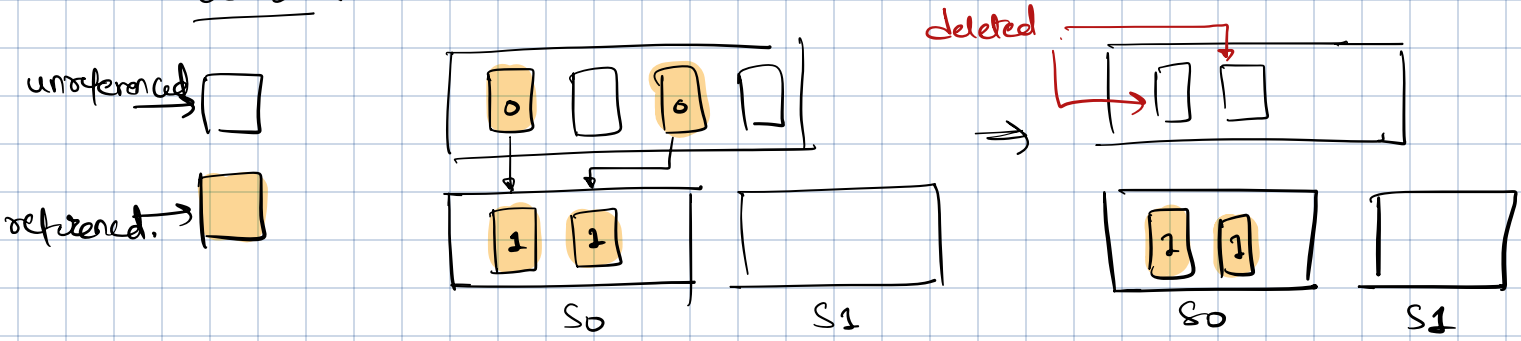
(minor GC)

Step 2: when eden space fills up, minor garbage collection is triggered.



Step 3:

- Referenced objects are moved to the first survivor place.
- Unreference objects are deleted when the eden space is cleared.



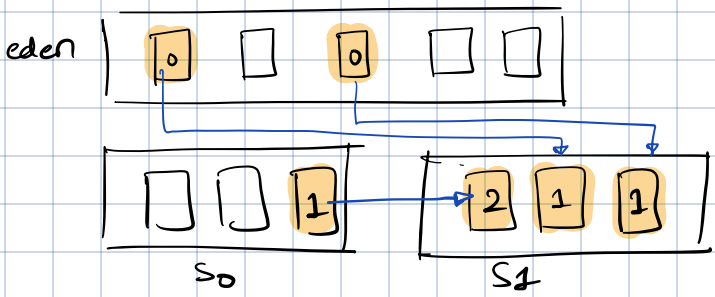
Step 4: In the next minor GC, (when eden space is filled)

1) All Ref object are marked and move to alternate survivor space (if prev. was S0 then now S1, if prev. was S1 then now S0)

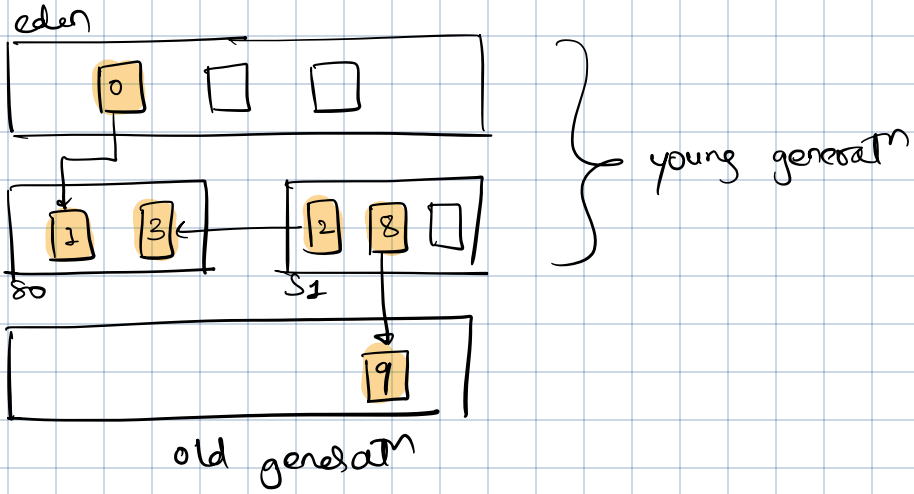
2) And also, all Ref object from other survivor (in this S0) are moved to S1 and their age is incremented.

3) Once all the survived objects are moved S1, both S0 & eden are cleared.

NOTE: S1 has differently aged object.



Step 5: This happens alternately until the age of the objects reaches to some threshold, they are promoted from young generation to old generation



Step 6: Eventually, a major GC will be performed on the old generation which cleans up & compacts that space

↓ deletion with compaction

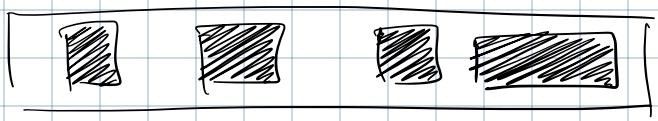
Marking Process:

- In this step, GC identifies which pieces of memory are in use & which are not.
- This step can be time consuming process if all objects in a system must be scanned.

Minor GC on young generation (Deletion without Compacting)

- Deletion removes unreferenced objects leaving referenced objects & pointers to free space.

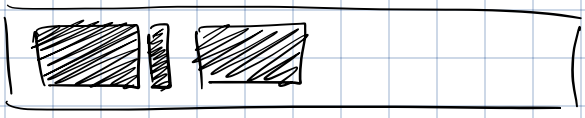
→ Memory allocators holds references to blocks of free space where new objects can be allocated.



After normal deletion.

Major GC on old generation (Deletion with compacting)

→ with compacting, this makes new memory allocatn much easier & faster, since there is no need to store the list of free space, only start ref is needed

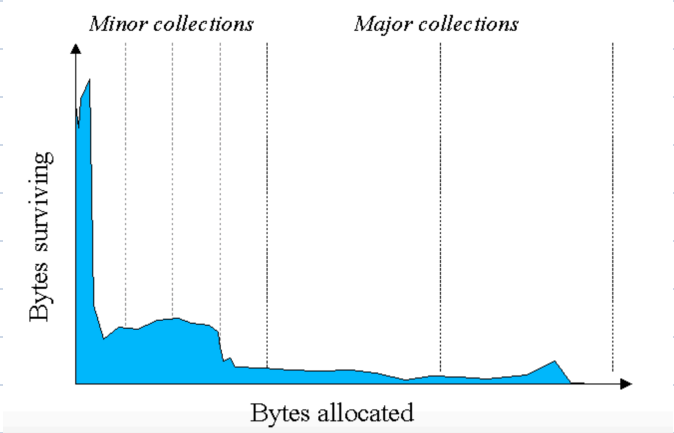


After normal delet^n with compacting

Why generational (generation-wise) Garbage Collection?

- mark & compact all the objects in a JVM is inefficient.
- As more objects get allocated → list of objects grows → longer GC time

→ However, empirical analysis application has shown that most objects are short lived.



no of bytes allocated

no. of bytes allocated over the time

Singleton Class: is a class that can have only one object at a time.
- if we try to create another instance, then the new ref variable will point to same instance present in the heap.

Purpose:
1. Global configuration Management: applicat^n parts modifys the same conf. inst.
2. Database connection Pooling: avoids unnecessary connections.
3. Logging Service: logs to same instance
4. Caching: to ensure all parts of the applicat^n use the same cache instance

Ask this question to yourself:

① How many instance we want?
→ ONE

② How to stop outside world from creating more than one instance?
→ An instance is created with construction being called with new keyword.

③ How to stop calling the constructor?
→ make the constructor private

④ Then how to access the instance of it?
→ create an static method to access it.

there are many ways to achieve singleton in the class:

①