Functions & Methods in Java

Lecture

- Functions & Return Types
- Scope of a Variables
- Call Stack
- Stack vs Heap Memory
- Primitives vs Object References
- Garbage Collection
- Problems

Methods

Methods

A method is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as functions.

Example 

```java
public class Main {

  static void sayHi() {
    System.out.println("Hi!");
  }

  public static void main(String[] args) {
    sayHi();
    sayHi();
    sayHi();
  }
}
```

Data can be passed to functions using one or more parameters. Parameters can have default values.

```java
public class Main {

  static void sayHi(String name) {
    System.out.println("Hi " + name);
  }

  public static void main(String[] args) {
    sayHi("Prateek");
    sayHi("Tarun");
    sayHi("Scaler");
  }
}
```

Data can be passed to functions using one or more parameters. Parameters can have default values.

```java
public class Main {

  static void sayHi(String name) {
    System.out.println("Hi " + name);
  }

  public static void main(String[] args) {
    sayHi("Prateek");
    sayHi("Tarun");
    sayHi("Scaler");
  }
}
```

Methods have Return Types

Data can be returned by functions by specifying the return type and having a return statement inside function.

```
1  public class Main {
2
3    static int areaOfSquare(int side) {
4      return side*side;
5    }
6
7    public static void main(String[] args) {
8          int area = areaOfSquare(5);
9      System.out.println(area);
10   }
11 }
```

Why Create Methods.

- Methods increase the reusability of code.
- Code looks more modular & organised.

Scope of Variable

Scope

In Java, variables are only accessible inside the region they are created. This is called scope.

Variables declared directly inside a method are available anywhere in the method following the line of code in which they were declared

Block Scope

```
1  public class Main {
2
3    public static void main(String[] args) {
4      int money = 100;
5      if(money>10){
6          int spends = 50;
7      }
8      System.out.println(spends); //error
9    }
10 }
```

Block Scope

A block of code may exist on its own or it can belong to an if, while or for statement. In the case of for statements, variables declared in the statement itself are also available inside the block's scope.

Function Scope

Variables declared directly inside a method are available anywhere in the method following the line of code in which they were declared

Problems-I 🚀

Write a method to find absolute value of a number.

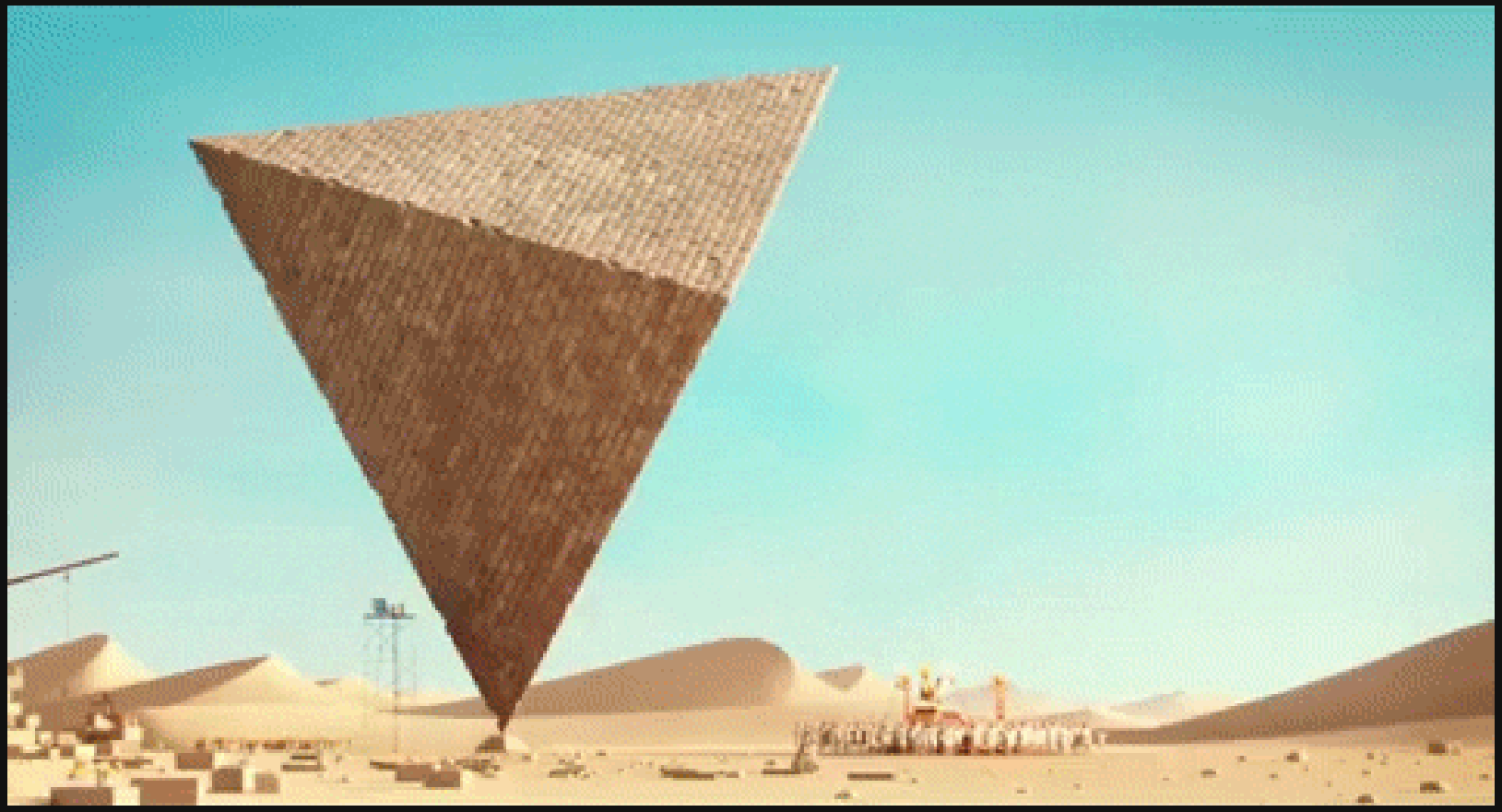
Problems-II 🚀

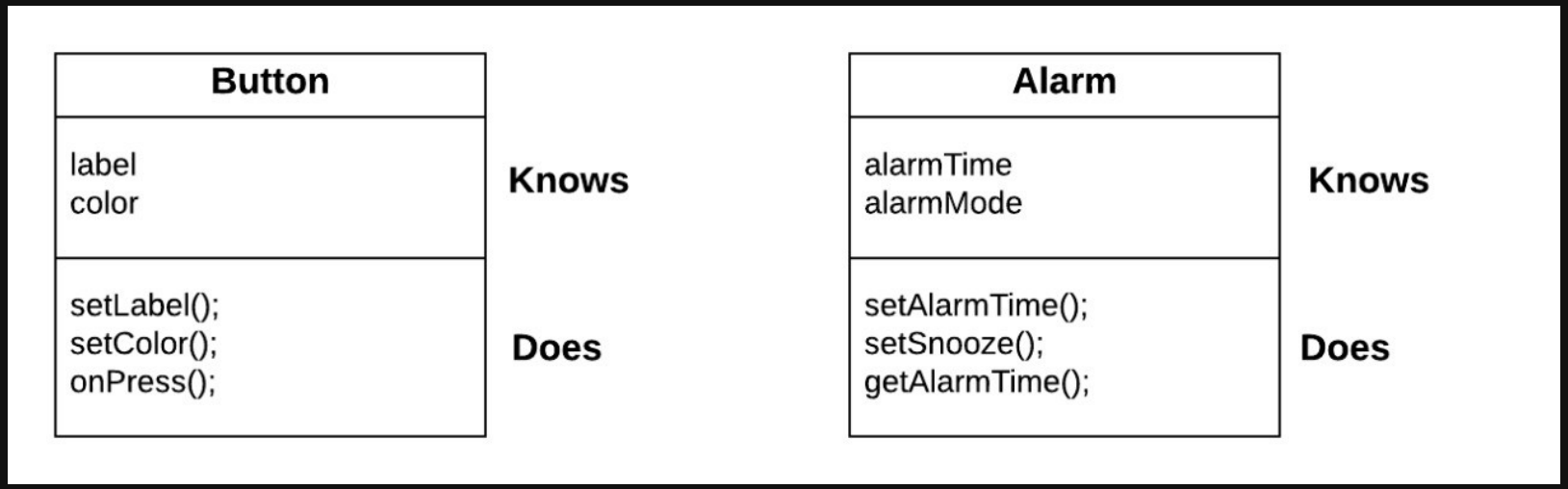
Write methods to convert

- decimal number to binary number.
- binary number to a decimal number

Primitives vs Objects

The Memory View




Button
label
color
Knows
setLabel();
setColor();
onPress();
Does
Alarm
alarmTime
alarmMode
Knows
setAlarmTime();
setSnooze();
getAlarmTime();
Does

Life and death on the heap

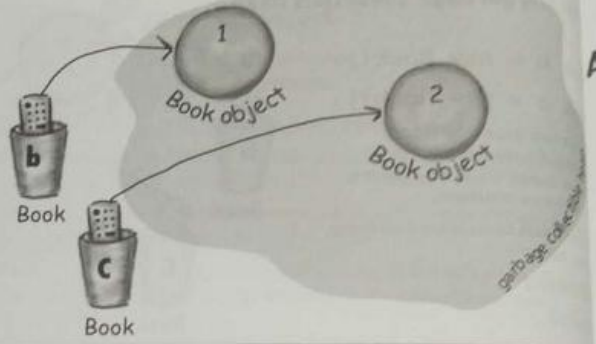
```
Book b = new Book();
Book c = new Book();
```

Declare two Book reference variables. Create two new Book objects. Assign the Book objects to the reference variables.

The two book objects are now living on the heap.

Active References: 2

Reachable Objects: 2




```
b = c;
```

Assign the value of variable **c** to variable **b**. The bits inside variable **c** are copied, and that new copy is stuffed into variable **b**. Both variables hold identical values.

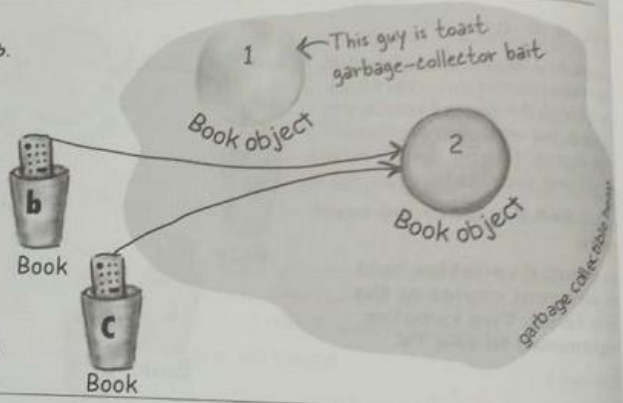
Both b and c refer to the same object. Object 1 is abandoned and eligible for Garbage Collection (GC).

Active References: 2

Reachable Objects: 1

Abandoned Objects: 1

The first object that **b** referenced, Object 1, has no more references. It's *unreachable*.




```
c = null;
```

Assign the value `null` to variable **c**. This makes **c** a *null reference*, meaning it doesn't refer to anything. But it's still a reference variable, and another Book object can still be assigned to it.

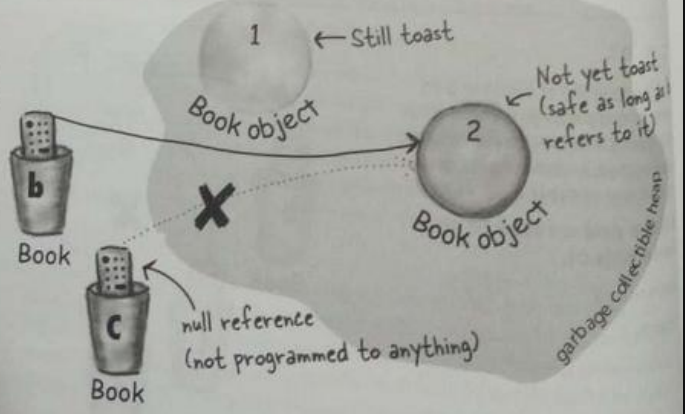
Object 2 still has an active reference (b), and as long as it does, the object is not eligible for GC.

Active References: 1

null References: 1

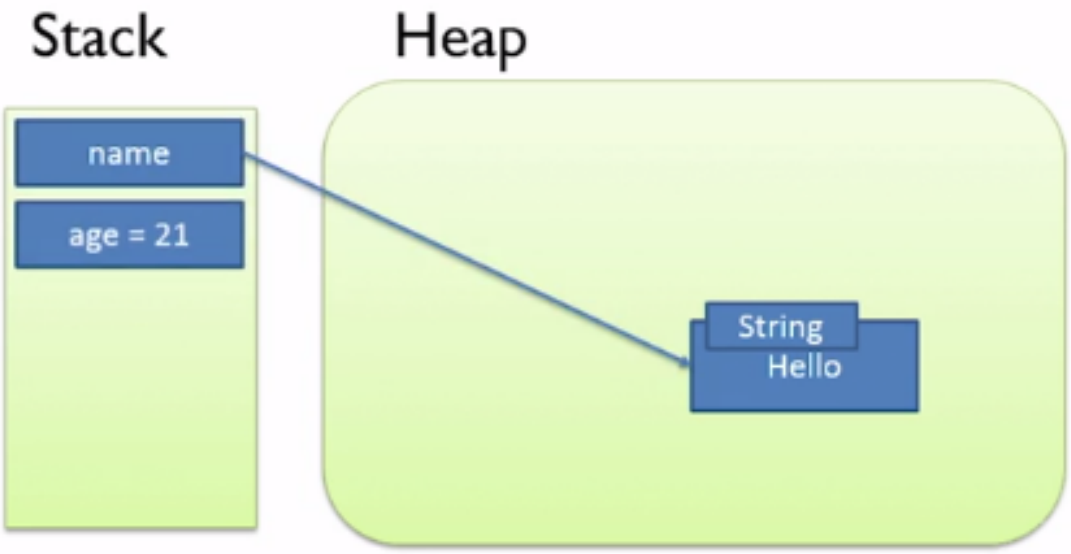
Reachable Objects: 1

Abandoned Objects: 1

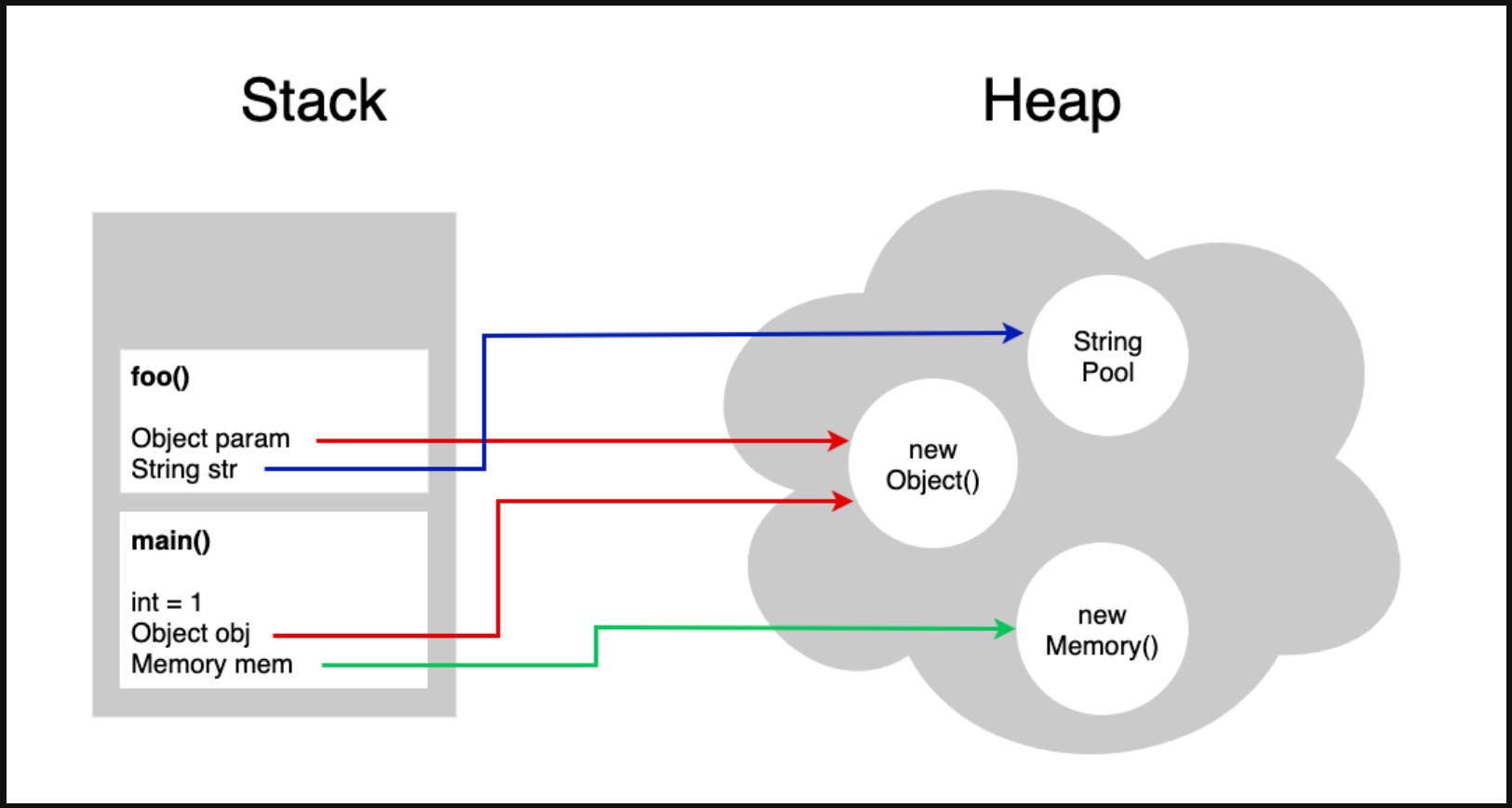


Java Memory

```
int age = 21;
String name = "Hello";
```



Objects live on the Heap!



Stack & Heap Memory

Memory

To run an application in an optimal way, JVM divides memory into stack and heap memory.

Whenever we declare new variables and objects, call a new method, declare a String, or perform similar operations, JVM designates memory to these operations from either Stack Memory or Heap Space.

Stack Memory

Stack Memory in Java is used for static memory allocation. It contains primitive values that are specific to a method and references to objects referred from the method that are in a heap.

Access to this memory is in Last-In-First-Out (LIFO) order. Whenever we call a new method, a new block is created on top of the stack.

When the method finishes execution, its corresponding stack frame is flushed, the flow goes back to the calling method, and space becomes available for the next method.

Key Features of Stack Memory

- It grows and shrinks as new methods are called and returned, respectively.
- Variables inside the stack exist only as long as the method that created them is running.
- It's automatically allocated and deallocated when the method finishes execution.
- If this memory is full, Java throws java.lang.StackOverFlowError.
- Access to this memory is fast when compared to heap memory.

Heap Memory

Heap space is used for the dynamic memory allocation of Java objects at runtime.

New objects are always created in heap space, and the references to these objects are stored in stack memory.

These objects have global access and we can access them from anywhere in the application

Heap Memory Features

- If heap space is full, Java throws java.lang.OutOfMemoryError.
- Access to this memory is comparatively slower than 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.

Garbage Collector!

Garbage Collection deals with finding and deleting the garbage from memory.

However, in reality, *Garbage Collection* tracks each and every object available in the JVM heap space and removes unused ones.

In simple words, GC works in two simple steps known as Mark and Sweep:

Mark – it is where the garbage collector identifies which pieces of memory are in use and which are not
Sweep – this step removes objects identified during the "mark" phase

Advantages

- No manual memory allocation/deallocation handling because unused memory space is automatically handled by *GC*
- No overhead of handling *Dangling Pointer*
- Automatic *Memory Leak* management (*GC* on its own can't guarantee the full proof solution to memory leaking, however, it takes care of a good portion of it)

Disadvantages

- Since JVM has to keep track of object reference creation/deletion, this activity requires more CPU power than the original application. It may affect the performance of requests which required large memory

- Programmers have no control over the scheduling of CPU time dedicated to freeing objects that are no longer needed

- Automatised memory management will not be as efficient as the proper manual memory allocation/deallocation

Prime Number Print

That's it!