CS2030 AY18/19 SEM 2

WEEK 10 | 29 MARCH 19

TA GAN CHIN YAO

DISCLAIMER

Slides are made by me, unofficial, optional
Available to download at bit.ly/cs2030_gan
Slides (if any) will be uploaded on
Friday weekly

Q1.

1. Given the following class A.

```
class A {
    int field;
    void method() {
        Function<Integer, Integer> func = x -> field + x;
    }
}
```

Model the execution of the program fragment:

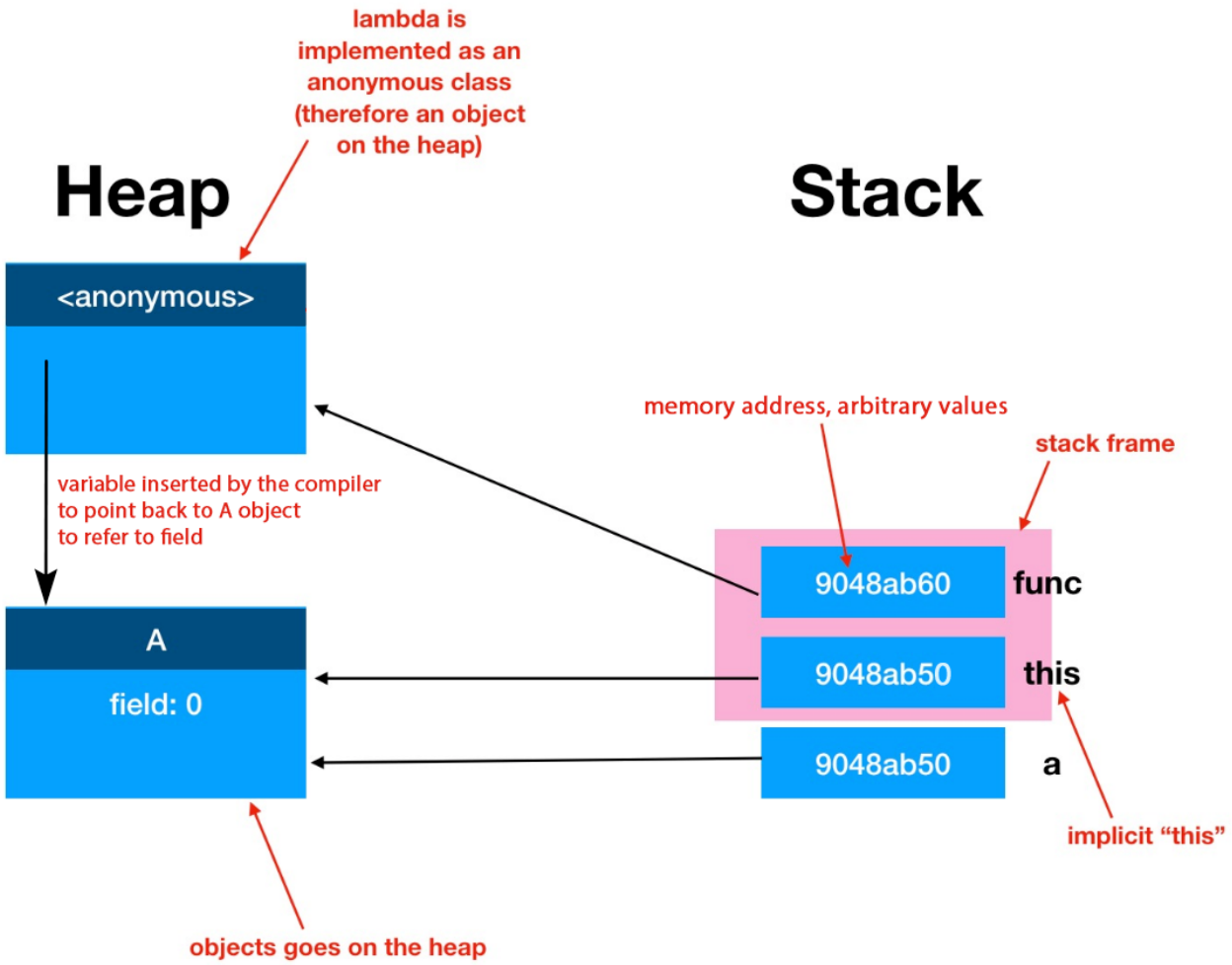
```
A a = new A();
a.method();
```

In particular, focus on the use of the *stack* and *heap* memory.

- A is a class so the instance field field would be on the heap.
- func is a local variable, so it would go on the stack.
- func refers to the lambda expression, which is internally implemented as an anonymous class, so it refers to an object on the heap.
- Finally, x is an argument to the lambda expression, so it is not stored anywhere.

Q1.

Ans:



1. Given the following class `A`.

```
class A {
    int field;
    void method() {
        Function<Integer, Integer> func = x -> field + x;
    }
}
```

Model the execution of the program fragment:

```
A a = new A();
a.method();
```

In particular, focus on the use of the *stack* and *heap* memory.

Q1.

More about Stack and Heap

- Stack and Heap are regions in memory.
- Partitioned memory for difference purposes
- Heap has a huge amount of memory. Stack only little amount
- If memory not enough -> error, programme will crash.
  E.g. StackOverflowError because Stack memory runs out

Stack

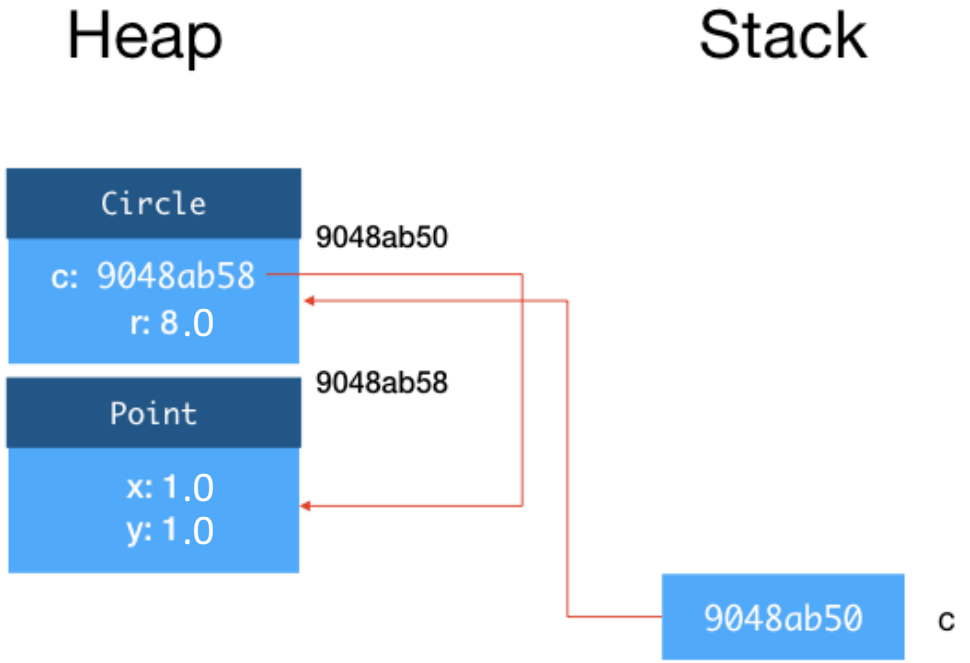
- all variables (including primitive types and object references) are allocated in and stored

Heap

- Where the actual object is stored

Q1. Example

```
1  Circle c;
2  c = new Circle(new Point(1, 1), 8);
```



Note: 9048ab50, 9048ab58 are arbitrary addresses in memory

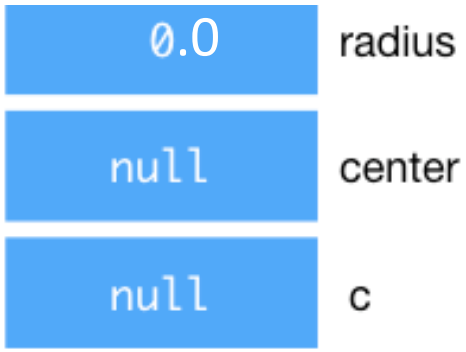
Q1.

Example

```
1   Circle c;
2   Point center;
3   double radius;
```

Heap

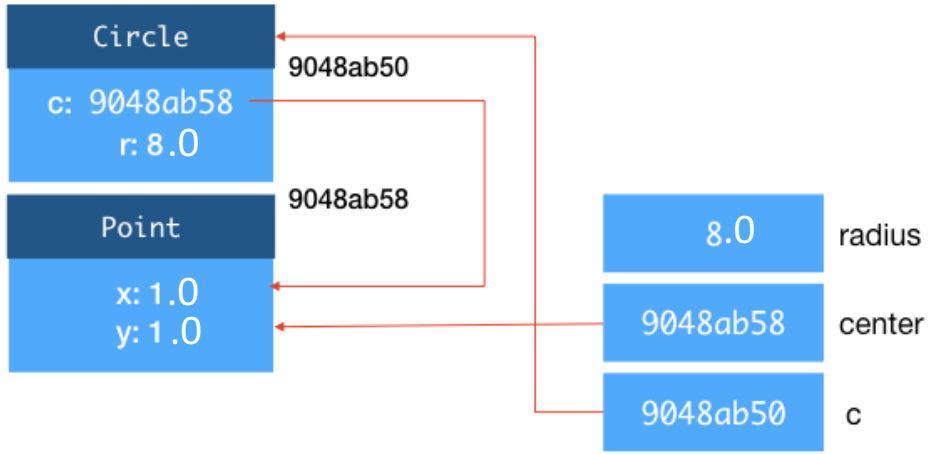
Stack



```
4   radius = 8;
5   center = new Point(1, 1);
6   c = new Circle(center, radius);
```

Heap

Stack



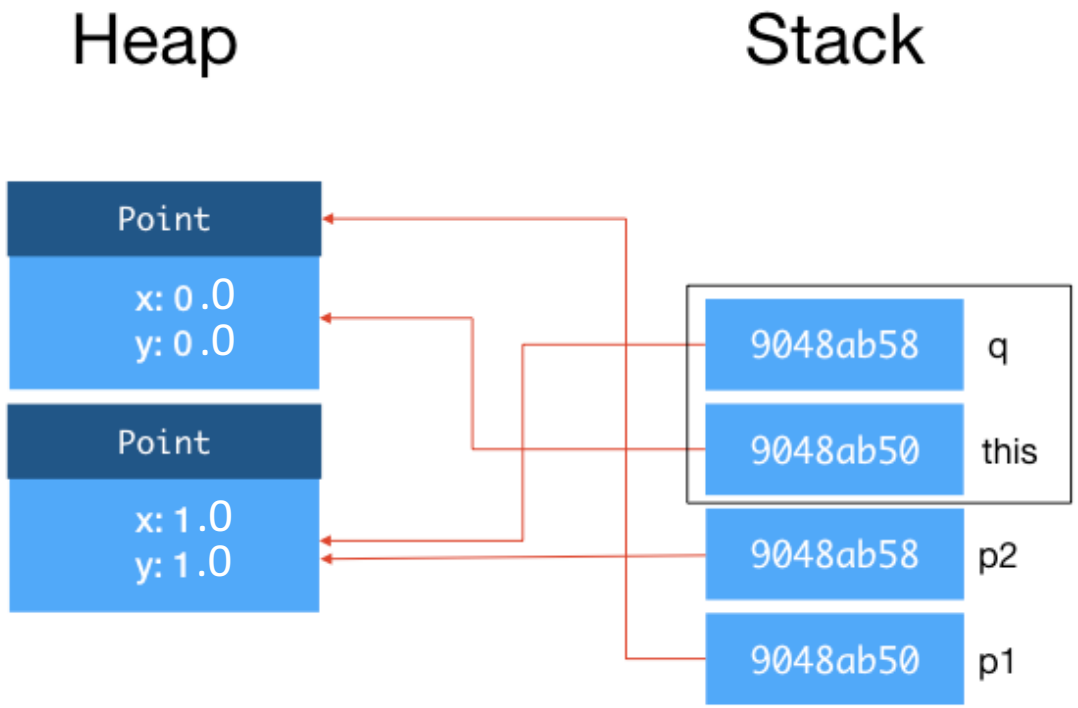
Primitive is stored as value, object is stored as address to actual object

Example


```
1  class Point {
2    private double x;
3    private double y;
4    public double distanceTo(Point q) {
5      return Math.sqrt((q.x - this.x)*(q.x - this.x)+(q.y - this.y)*(q.y - this.y
6    }
7  }
```

and the invocation:

```
1  Point p1 = new Point(0,0);
2  Point p2 = new Point(1,1);
3  p1.distanceTo(p2);
```



When distanceTo is called, JVM creates a *stack frame* for this instance method call. This stack frame is a region of memory that tentatively contains (i) the this reference, (ii) the method arguments, and (iii) local variables within the method, among other things[34]. When a class method is called, the stack frame does not contain the this reference.

Q2.
a.

2. Suppose we have the following lambda expression of type `Function<String, Integer>`:

```
str -> str.indexOf(' ')
```

(a) Write a `main` method to test the usage of the lambda expression above.

```java
public static void main(String[] args) {
    Function<String, Integer> f = str -> str.indexOf(' ');
    System.out.println(f.apply("hello world"));
}
```

Interface Function<T,R> API:

Modifier and Type	Method and Description
R	**apply**(T t) Applies this function to the given argument.

Q2.
b.

(b) Java implements lambda expressions as anonymous classes. Write the equivalent anonymous class for the lambda expression above.

Original Lambda:

```
Function<String, Integer> f = str -> str.indexOf(' ');
System.out.println(f.apply("hello world"));
```

Ans:

```
Function<String, Integer> f = new Function<>() {
    public Integer apply(String str) {
        return str.indexOf(' ');
    }
};
System.out.println(f.apply("hello world"));
```

Q3.

3. Complete the method `and` that takes in two `Predicate` objects `p1` and `p2` and returns a new `Predicate` object that evaluates to `true` if and only if both `p1` and `p2` evaluate to `true`.

```
Predicate<T> and(Predicate<T> p1, Predicate<T> p2) {
```

Interface Predicate<T> API:

Modifier and Type	Method and Description
`boolean`	**`test`**`(T t)` Evaluates this predicate on the given argument.

Q3.

Ans: Predicate<T> and(Predicate<T> p1, Predicate<T> p2) {

- Using lambda:

```
return x -> p1.test(x) && p2.test(x);
```

- Using anonymous class:

```
return new Predicate<T>() {
    public boolean test(T x) {
        return p1.test(x) && p2.test(x);
    }
}
```

- The following is wrong:

```
return p1.test(x) && p2.test(x);
```

It *eagerly* evaluates the predicates and returns a boolean.

Q4.

4. Write a method product that takes in two `List` objects `list1` and `list2`, and produce a `Stream` containing elements combining each element from `list1` with every element from `list2` using a `BiFunction`. This operation is similar to a Cartesian product.

```
public static <T,U,R> Stream<R> product(List<? extends T> list1,
        List<? extends U> list2,
        BiFunction<? super T, ? super U, R> func)
```

Interface BiFunction<T,U,R> API:

Modifier and Type	Method and Description
R	**apply**(T t, U u) Applies this function to the given arguments.

Q4.

For example, the following program fragment

```
List<Integer> list1 = new ArrayList<>();
List<Integer> list2 = new ArrayList<>();

Collections.addAll(list1, 1, 2, 3, 4);
Collections.addAll(list2, 10, 20);

product(list1, list2, (str1, str2) -> str1 + str2)
    .forEach(System.out::println);
```

gives the output

11
21
12
22
13
23
14
24

Q4.

Ans:

```
class Product {

    public static <T, U, R> Stream<R> product(
            List<? extends T> list1,
            List<? extends U> list2,
            BiFunction<? super T, ? super U, ? extends R> func) {

        return list1.stream()
            .flatMap(x ->
                    list2.stream()
                    .map(y -> func.apply(x,y)));
            }

    public static void main(String[] args) {
        List<Integer> list1 = new ArrayList<>();
        List<Integer> list2 = new ArrayList<>();

        Collections.addAll(list1, 1, 2, 3, 4);
        Collections.addAll(list2, 10, 20);

        product(list1, list2, (str1, str2) -> str1 + str2)
            .forEach(System.out::println);

    }
}
```

Q4.

5. Write a method that returns the first n Fibonacci numbers as a `Stream<BigInteger>`. The `BigInteger` class is used to avoid overflow.

   For instance, the first 10 Fibonacci numbers are $1, 1, 2, 3, 5, 8, 13, 21, 34, 55$.

   Hint: It would be useful to write a new `Pair` class that keeps two items around in the stream.

Fibonacci series

/fɪbəˈnɑːtʃi/

noun MATHEMATICS

a series of numbers in which each number (*Fibonacci number*) is the sum of the two preceding numbers. The simplest is the series 1, 1, 2, 3, 5, 8, etc.

Q4.

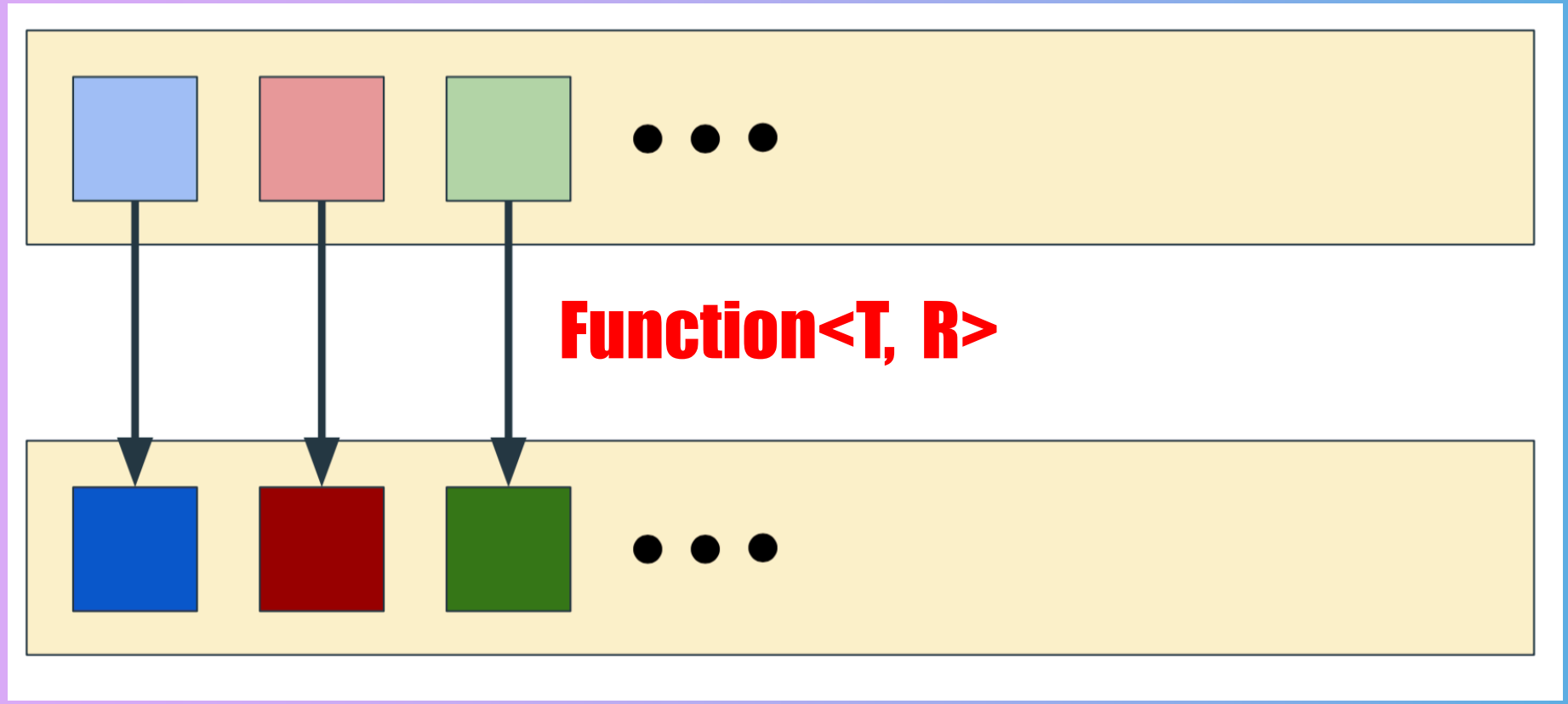
Ans:

```
Stream<BigInteger> fibonacci(int n) {
    return Stream.iterate(
                new Pair<>(BigInteger.ZERO, BigInteger.ONE),
                pr -> new Pair<>(pr.second, pr.first.add(pr.second)))
            .map(pr -> pr.second).limit(n);
}
```

SUMMARY CONCEPTS

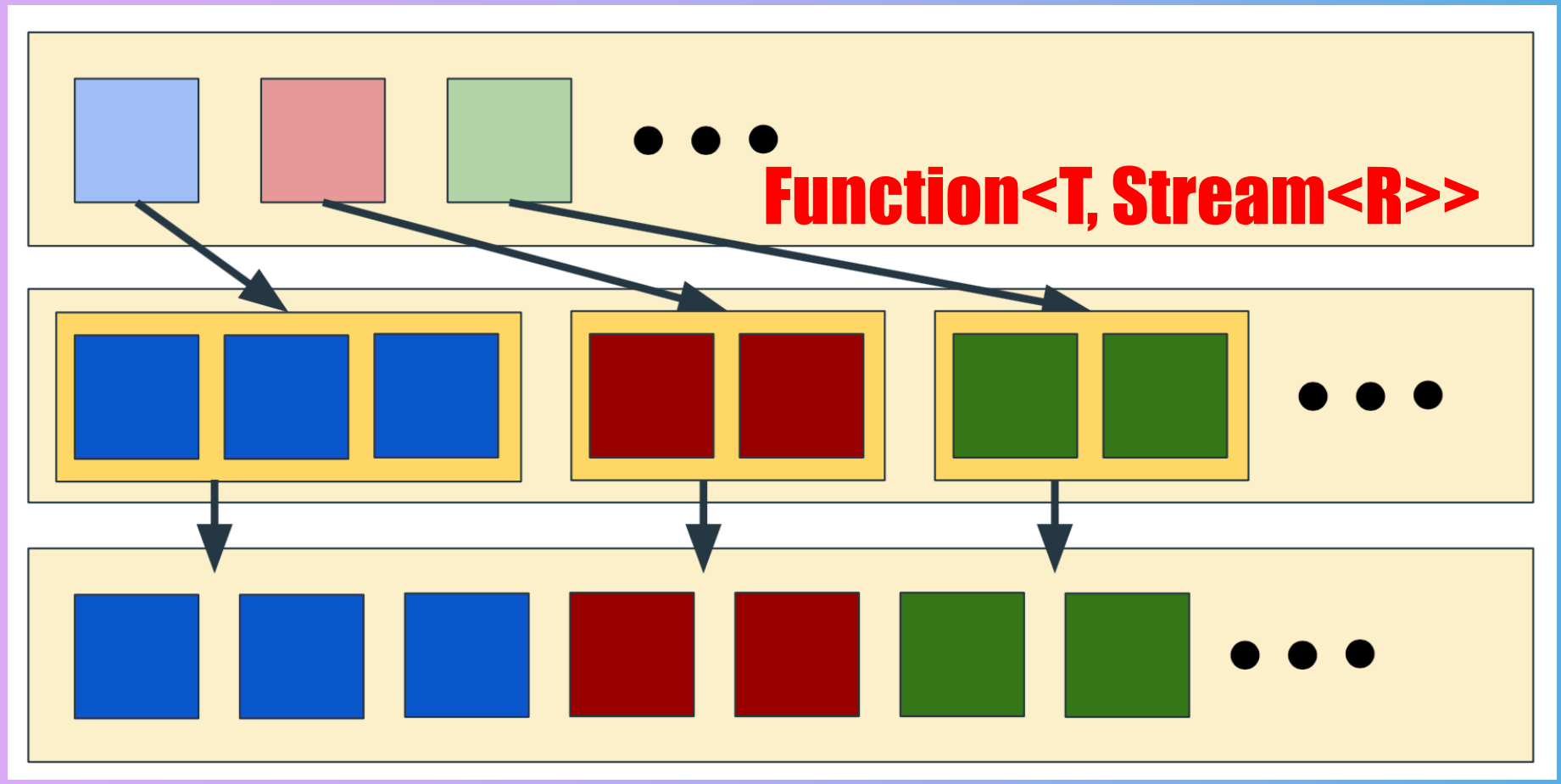
- Stack and heap
- How to draw stack and heap diagram
- How to use Function<T, R>
- How to use Predicate<T>
- How to use BiFunction<T, U, R>
- Map vs Flatmap

Map



If your Function returns a value R, map() will return Stream(values).

Flatmap



If your Function returns a Stream, map() will return Stream(Stream(values)). Flatmap() will return Stream(values)

QUESTIONS?