Nexthink Java Quiz

# Thank you very much for filling out this quiz to the best of your competences. The purpose of this quiz is to evaluate your level of expertise in technologies and techniques that we use daily @Nexthink.

# Good luck and thank you for your time!

# Question 1:

A producer thread periodically produces elements and puts them in a queue, and a consumer thread takes the elements from the queue and does some processing with them:

|  |  |
| --- | --- |
| **private final** List<Integer> **queue** = **new** LinkedList<>(); | |
| **Consumer**  **synchronized** (\_\_\_\_\_\_) {  **while** (\_\_\_\_\_\_) {  \_\_\_\_\_\_  }  **Integer** num = **queue**.remove(0);  \_\_\_\_\_\_  processElement(num); } | **Producer**  **synchronized** (\_\_\_\_\_\_) {  **int** maxSize = 1000L;  **while** (\_\_\_\_\_\_) {  \_\_\_\_\_\_  }  **queue**.add(**new** Random().nextInt());  \_\_\_\_\_\_ } |

Please write the missing instructions for the producer and consumer threads. Feel free to modify the given code structure, if needed.

Click here to enter text.

# Question 2:

Please complete the “sum” method in the following code (in a recursive way):

**public class** Payment {  
 **private final int amount**;  
  
 **public** Payment(**int** amount) {  
 **this**.**amount** = amount;  
 }  
  
 **public int** getAmount() {  
 **return amount**;  
 }  
}

**public class** Question3 {  
  
 **public static void** main(String[] args) {  
 List<Payment> payments = **new** ArrayList<>();  
 payments.add(**new** Payment(100));  
 payments.add(**new** Payment(50));  
  
 System.***out***.println(*sum*(…));  
 }  
  
 **private static int** sum(…) {**…**}  
}

Recursive sum:

Click here to enter text.

What can go wrong with this recursive function if the number of payments becomes too big?

Click here to enter text.

# Question 3:

Given the following code:

**public class** SafeLock {  
 **private** Lock **lock** = **new** ReentrantLock();  
  
 **public void** lock() **throws** InterruptedException {  
 LockingManager.*registerLock*(**this**);  
 **long** timeout = 1000;  
 **while** (!**lock**.tryLock(timeout, TimeUnit.***MILLISECONDS***)) {  
 **if** (LockingManager.*checkForCircularDependency*()) {  
 **throw new** ThreadLockException();  
 }  
 }  
 }  
  
 **public void** unlock() {  
 LockingManager.*unregisterLock*(**this**);  
 **lock**.unlock();  
 }  
}  
  
**public class** ThreadLockException **extends** RuntimeException {  
}

This class is designed to cope with a problematic situation when dealing with locks, what is it?

Click here to enter text.

What techniques can be used to avoid running into such a situation?

Click here to enter text.

What does checkForCircularDependency() possibly do?

Click here to enter text.

What particular characteristic do the static methods in LockingManager require to work properly?

Click here to enter text.

# Question 4:

When running the following program:

**class** Animal {  
 **private final int id**;  
 **private final** String **name**;  
  
 **public** Animal(**int** id, String name) {  
 **this**.**id** = id;  
 **this**.**name** = name;  
 }  
  
 @Override  
 **public boolean** equals(Object other) {  
 **if** (**this** == other) **return true**;  
 **if** (!(other **instanceof** Animal)) **return false**;  
 Animal otherAnimal = (Animal) other;  
 **return id** == otherAnimal.**id**;  
 }  
}

**public class** Question5 {  
 **public static void** main(String[] args) {  
 Animal lion1 = **new** Animal(1, **"lion"**);  
 Animal lion2 = **new** Animal(1, **"lion"**);  
 Set<Animal> animals = **new** HashSet<>();  
 animals.add(lion1);  
 animals.add(lion2);  
 System.***out***.println(**"Number of animals: "** + animals.size());  
 }  
}

The result is:

A description...

The result should instead be Number of animals: 1. What is missing in the Animal class? Please also provide an implementation of the missing part.

Click here to enter text.

**Question 5:**  
Consider the code presented in folder Question5 of the attached Java resources.

The program is a simple expression evaluator able to compute integer expressions bases on four operators (add, subtract, multiply and divide). The logic is implemented in class *Expression*, which offers the following features:

* Evaluating the expression to retrieve its Integer value;
* Pretty printing the expression;
* Exporting the expression to XML.

The goal of this exercise is to refactor the *Expression* class (and optionally the main program) to improve the code. You can change the code in any way you want, keeping in mind that:

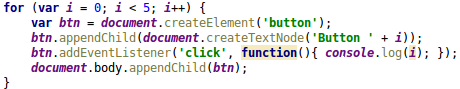
* In the main program, we still want to create expression programmatically (no need to implement a parser);
* The 3 features described above must behave like in the original code, specifically:
  + No matter how the API for constructing the expression is implemented, it must still take care of invalid input.
  + The output of pretty-printing the expression must be the same as now.
  + The output of XML serialization must be the same as now.
* You can add classes and change the API as you wish, as long as the previous points are not violated.

What we’re looking for is clean, solid, easy to understand, and correct code. Also think about which design pattern can help you make the code more readable and avoid code duplication.

Nexthink React Quiz

# Question 6:

Given the following code:



What is the result when you click on Button 3?

Click here to enter text.

Explain why and mechanisms that are behind?

Click here to enter text.

# Question 7:

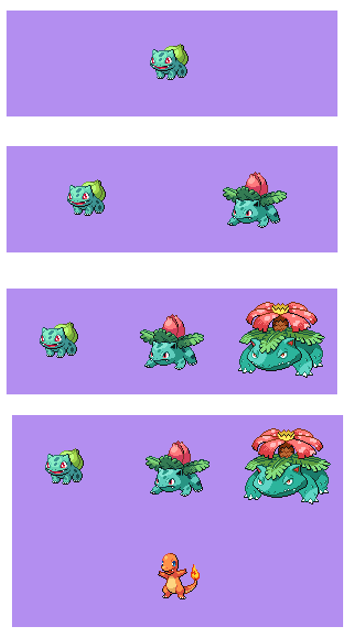
Create a React project which allows the user to catch and release Pokémons.

You will have two lists, **wild Pokémons** and **catched Pokémons.**

****

## Wild Pokémons

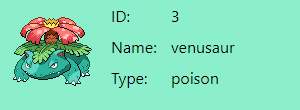
* Display a list of 0 to 151 Pokémons (ID from 1 to 151).
* A new Pokémon must appear in the list every half second until all 151 Pokémons are loaded.
* The items should be ordered by Pokémon Id.
* Pokémons will be displayed as follow (Here you can see them being loaded one by one).



* The list will display at maximum 3 Pokémons horizontally.
* The list can expand vertically as much as needed to fit all the Pokémons.
* Clicking on a wild Pokémon will remove it from the **wild Pokémons** list and put it in the **catched Pokémons** list.
* There can be at maximum 6 **catched Pokémons**, when the limit is reached, clicking on a wild Pokémon has no action.

## Catched Pokémons

* Display a list of 0 to 6 Pokémons. One item horizontally, displayed as followed.



* Clicking on a catched Pokémon will release it into the wild by adding it back to the wild list and removing it from the catched list.

## API

To get the Pokémons you will have to query an API. <https://pokeapi.co/docs/v2.html#pokemon>

You will need to use a GET query on the following.

https://pokeapi.co/api/v2/pokemon/id // id of pokemon is integer

With id being the id of the Pokémon. You’ll get a JSON response which you can parse as following:

id = data.id;

name = data.name;

sprite = data.sprites.front\_default; // Url of image

type = data.types[0].type.name;

## Guidelines

This project should be a static web page and could be executed from files, no need of persistence or any server.

* Use the library you prefer and explain your choices (what it brings to you, advantages or inconvenient)
* Explain how you can test this project
* Language can be Javascript or Typescript
* Gotta Catch 'Em All!