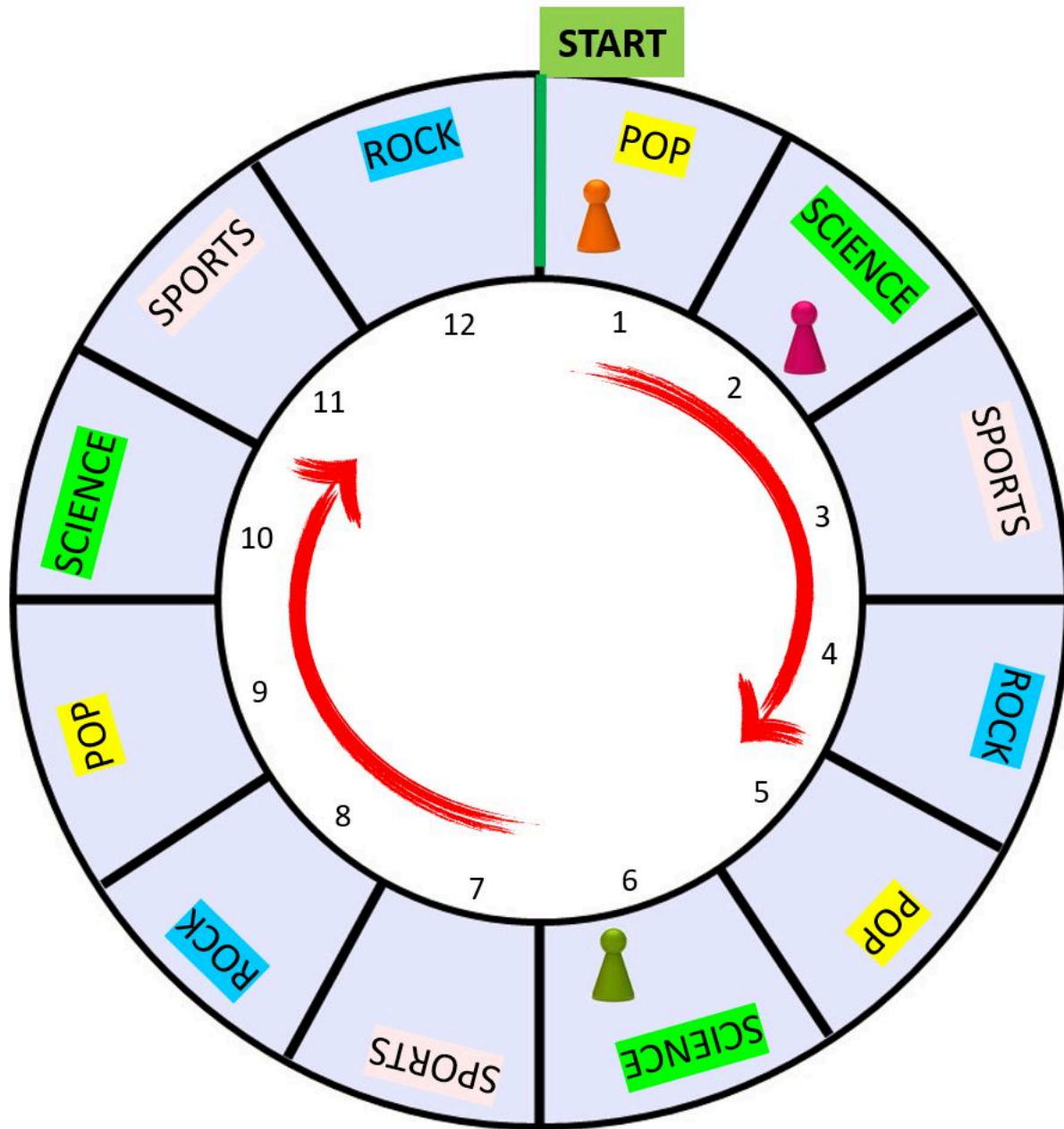



# Play with Trivia Refactoring Kata




by Bogdan Mihai Nicolae



## The problem

- Clone Victor Rentea's [kata-trivia-java](https://github.com/webtoknow/kata-trivia-java) repository to get started on your local machine! 

```
git clone https://github.com/webtoknow/kata-trivia-java.git
cd kata-trivia-java
```

- **Our mission:** Refactor Game.java , but keep an eye on the GameTest  to make sure nothing breaks! 

## Fixed sized arrays Java (static)

```
private String[] players = new String[6];  
private int[] places = new int[6];  
private int[] purses = new int[6];  
private boolean[] inPenaltyBox = new boolean[6];
```

## Learn More:

Take a look at [Why do some arrays in some languages have a fixed size, like in Java?](#)

## Array In javascript (dynamic)

- In JavaScript, arrays are dynamic! 🚀 They grow and shrink like magic! 🎩 ✨
- When your array gets too full, don't worry—it doubles in size! 💪

```
let players = [];  
players.push("John");  
players.push("Doe");  
players.push("Jane");
```

## Learn More:

Explore this topic: [Static vs Dynamic Arrays in JavaScript](#)

## Fixed sized arrays in JavaScript

```
let a = new Array(42);

if(Object.seal) {
  // fill array with some value because
  // empty slots can not be changed after calling Object.seal
  a.fill(undefined);

  Object.seal(a);
  // now a is a fixed-size array with mutable entries
}
```

### Learn More:

Check out this discussion: [Is it possible to create a fixed length array in javascript](#)

## 💡 Missing Types in the Codebase

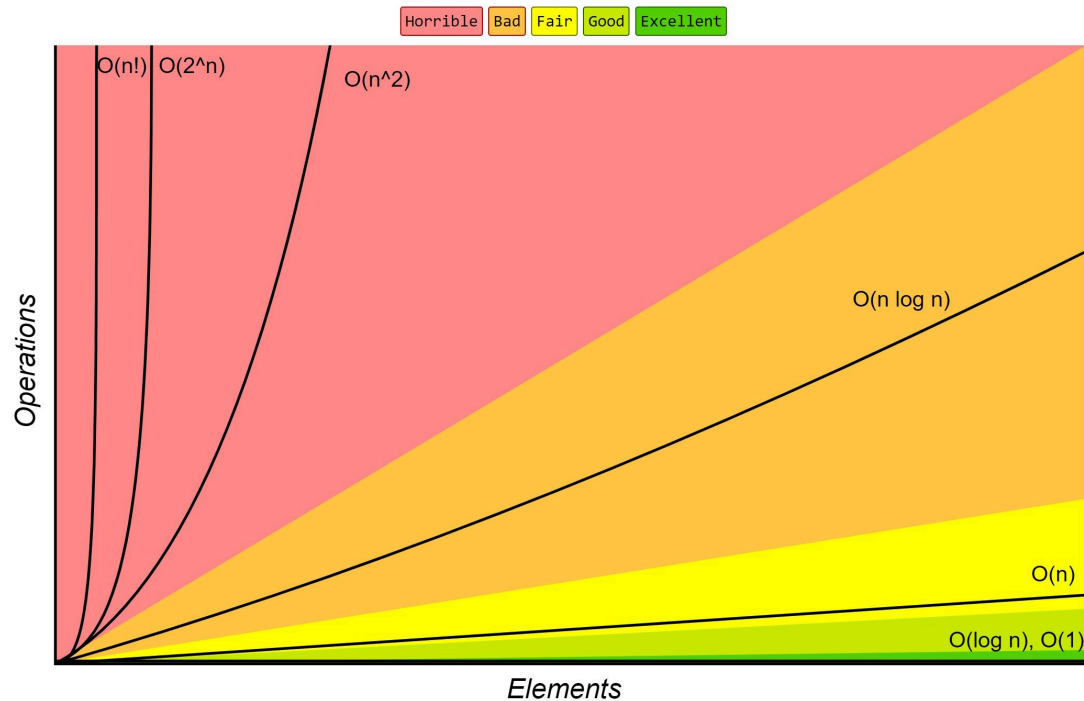
Working without proper types can lead to 🐛 runtime errors, 🤖 hard-to-read code, and 🚧 slower development cycles. Adding types ensures your code is robust and developer-friendly. 🛡️ ✨

### 🏆 Key Benefits:

- **Improves reliability:** Catch errors before runtime. ✅
- **Enhances developer experience:** Better autocompletion and IDE support. 💻 ✍️
- **Simplifies maintenance:** Types act as living documentation for your code. 📄 ✨



# Big O Complexity



## Learn More:

The cheat sheet for algorithmic complexity: [Big O Cheat Sheet](#)



## 💡 The DRY Principle


"Don't Repeat Yourself" 🛑🔄 ensures every piece of knowledge in your system has a single, definitive source.

### 🏆 Key Benefits:




- **Easier maintenance:** Fix a bug once and you're done! 🔍
- **Consistency guaranteed:** Fewer changes = fewer errors. ✅
- **Clean and scalable:** Your code stays readable and ready to grow. 🚀



## Single-Responsibility Principle (SRP) - The S in SOLID

Every class, module, or function should have a single responsibility—focus on doing one thing well! 

### Key Benefits:

- **Improved readability:** Clearer, more focused code. 
- **Easier debugging and testing:** Fewer responsibilities = fewer bugs. 
- **Enhanced maintainability:** Isolated changes don't ripple through the system. 




# The KISS Principle

"Keep It Simple, Stupid" 🧐✂️ (KISS), the goal is to keep systems as simple as possible—no overcomplicating things! 🎯






## Key Benefits:

- **Easier to debug and maintain:** Less code means less chaos when things go wrong! 🔧🔧
- **Faster onboarding for new developers:** Simplicity = less confusion, more learning! 📖🚀
- **Reduced risk of errors:** Fewer moving parts = fewer chances for mistakes! ⚡




















 **Flatten Functions** Flattening functions means simplifying them by breaking down complex, nested structures into smaller, more manageable pieces.  Keep each function short, clear, and focused on a single task. 



 **Key Benefits:**

- **Improved readability:** Easier for others (and future you!) to understand.  
- **Simpler debugging:** Smaller functions are easier to test and fix.  
- **Better reusability:** Clean, standalone functions can be used elsewhere. 

## Conclusion

- Refactoring is key : Enhancing maintainability, readability, and performance .
- Principles applied:
  - **SOLID** : Cleaner, modular design with a focus on single responsibility .
  - **DRY**  : Eliminated redundancy for better code reuse  and simpler maintenance .
  - **KISS**  : Streamlined the game logic to avoid unnecessary complexity .
  - **Big O Complexity** : Optimized for better performance and scalability .
- Final goal : Ensure a more robust  and flexible game structure that can easily accommodate new features  and growth .

## Further Reading & Watching

- Books :
  - [Clean Code: A Handbook of Agile Software Craftsmanship](#)
  - [Refactoring: Improving the Design of Existing Code](#)
- Video :
  - [Clean Code: The Next Chapter by Victor Rentea](#)
  - [Live-Refactoring a realistic codebase](#)
  - [Clean Code - Uncle Bob / Lesson 1](#)