Issue	Problem Description	How to solve

bug	Save and re-use this "Random". "Random" objects should be reused.
	
	Creating a new Random object each time a random value is needed is inefficient and may produce numbers which are not random depending on the JDK. For better efficiency and randomness, create a single Random, then store, and reuse it.
	
	The *Random()* constructor tries to set the seed with a distinct value every time. However there is no guarantee that the seed will be random or even uniformly distributed. Some JDK will use the current time as seed, which makes the generated numbers not random at all.
	
	This rule finds cases where a new Random is created each time a method **is invoked.**
		`‘’`
		`private Random rand = SecureRandom.getInstanceStrong();  // SecureRandom is preferred to Random`
		`‘’`
		`Use this everytime you need a new`

Security Hotspot	Make sure that returning this pseudorandom number generator is safe here in *generateRandomInt()*.	It is related to the problem upwards. The solution is the same.
Code Smell	Refactor the code in order to not assign to this loop counter from within the loop body. A for loop stop condition should test the loop counter against an invariant value (i.e. one that is true at both the beginning and ending of every loop iteration). Ideally, this means that the stop	I cannot not see compliant solution to this code smell that could not affect the application good’s functioning

	condition is set to a local variable just before the loop begins. Stop conditions that are not invariant are slightly less efficient, as well as being difficult to understand and maintain, and likely lead to the introduction of errors in the future. This rule tracks three types of non-invariant stop conditions: ● When the loop counters are updated in the body of the for loop ● When the stop condition depend upon a method call ● When the stop condition depends on an object property, since such properties could change during the execution of the loop.	
Code smell	Reorder the modifiers to comply with the Java Language Specification. static public EuromillionsDraw generateRandomDraw() The Java Language Specification recommends listing modifiers in the following order: 1. Annotations 2. public 3. protected 4. private 5. abstract 6. static 7. final 8. transient 9. volatile 10. synchronized 11. native 12. default	**Compliant Solution** public static void main(String[] args) {}

	13. strictfp Not following this convention has no technical impact, but will reduce the code’s readability because most developers are used to the standard order.	
Code Smell	public ArrayList<Dip> findMatches(CuponEuromillions playCuppon) The return type of this method should be an interface such as "List" rather than the implementation "ArrayList".	The purpose of the Java Collections API is to provide a well defined hierarchy of interfaces in order to hide implementation details. Implementing classes must be used to instantiate new collections, but the result of an instantiation should ideally be stored in a variable whose type is a Java Collection interface. This rule raises an issue when an implementation class: ● is returned from a public method. ● is accepted as an argument to a public method. ● is exposed as a public member.