

Homework 9

Q1)(C)

`String` does not allow appending. Each method we invoke on a `String` creates a new object and returns it. This is because `String` is immutable, it can't change its internal state. On the other hand `StringBuilder` is mutable. When we call `append()`, it alters the internal array, rather than creating a new `String` object. Thus it is more efficient.

Every immutable object in Java is thread safe, that implies `String` is also thread safe. `String` can not be used by two threads simultaneously. But `StringBuilder` class does not provide thread safety. We should use `StringBuffer` for thread safety. It is exactly the same as the `StringBuilder` class, except that it is thread-safe by virtue of having its methods synchronized. Both `String` and `StringBuilder` supports different languages and encoding.

Q2)(D)

A `String` can be created with usage of `String` literals, without a call to a constructor. Strings created via literals are interned, they are held inside a special place in heap called string pool. When we create a `String` object with a `String` literal, string pool is checked for already existing values, and if found, uses that object without creating a new one while the `new()` operator always creates a new `String` object. Strings are immutable, immutable objects once created, they can not be modified further. Hence they're essentially read-only. And maybe for some additional info related to question one, read-only things are always thread-safe, but if we want to modify something in a thread safe manner, we need exclusive lock.

Q3)(D)

First option creates an empty `StringBuilder` and appends "`clown`" to it via method chaining. Second option simply creates a `StringBuilder` with the value "`clown`" passed to constructor. Lastly the third option creates a `StringBuilder` with the value "`cl`" and then invokes `insert()` method on that newly created object to insert the "`own`" value starting from index 2, which means starting from the end of the value "`cl`" in that case.

Q4)(B)

A new `StringBuilder` Object is created with the value "`333`" and is assigned to `teams` variable. Then, the following two lines appends two values, invoking `append` method on the `StringBuilder` object.

The `append()` method is by far the most frequently used method in `StringBuilder`. It adds the given parameter to the `StringBuilder` and returns a reference to the current `StringBuilder`.

Q5)(B)

The `java.util.List` is a child interface of `Collection`. It is an ordered collection of objects in which the duplicate values can be stored. Since `List` preserves the insertion order, it allows positional access and insertion of elements. `List` interface is implemented by the classes of `ArrayList`, `LinkedList`, `Vector` and `Stack`.

The elements contained in a Java `List` can be inserted, accessed, iterated and removed according to the order in which they appear internally in the Java `List`. Coming back to the question, since `List` is not a class but an **interface**, we can not instantiate it. `Object` is not a compatible class since obviously it is the base class of all objects in Java and does not implements `List` interface. If there would have been a cast to the `List` type then the code would have compile although it would throw `ClassCastException`, but there isn't.

`ArrayList` is the only given viable option because it does implements the `List` interface as we have pointed out above.

Q6)(C)

Java `List add(E e)` method appends the element at the end of the list. Since `List` supports generics, the type of elements that can be added is determined when the list is created. In the question a new `List` is created with an `ArrayList` implementation, with a type parameter `String` used, which indicates the types of elements that the `List` is going to store. Then 3 elements were appended to the list. The `get()` method of `List` interface in Java is used to get the element present in this list at a given specific index, which is 1 in our case, corresponding to `"nail"` string object.

Q7)(C)

The code does not compile, because the `sb` variable is yet to be initialized where it is used. After the method calls are invoked reference is assigned to the variable. A new `StringBuilder` object is allocated, initialized and `insert` method is invoked on that newly created object and then the reference to the newly created object is assigned to the variable.

Q8)(A)

The code gives the output `"[Natural History, Science]"`. An array is created with an initial size of 1, and then 3 elements were added to the list, third element which has the index of 2 removed before printing the array.

An `ArrayList` is created with an initial capacity of 1. The constructor with the signature `ArrayList(int capacity)`, is used to build an arraylist with initial capacity being specified. The `List` interface in Java extends `Collection` and declares the behavior an ordered collection also known as a sequence. `ArrayList` is the resizable array implementation of the `List` interface. An `ArrayList` has an initial capacity which is simply the size of the array

used to store the elements in the list. As elements are added to `ArrayList` its capacity grows automatically. The initial capacity does not change the logical size of an `ArrayList` rather it reduces the amount of incremental reallocation. If we do not specify an initial capacity then an `ArrayList` object will be created containing an initial array of size 10.

Q9)(C)

The code's output is `"321"`. First a `StringBuilder` is created with the value `"12"`, `"3"` appended with the following line. The `reverse()` method was invoked on the `StringBuilder` object. The `reverse()` method causes this character sequence to be replaced by the reverse of the sequence, returning `StringBuilder` object after reversing the characters. Let n be the character length of this character sequence (not the length in char values) just prior to execution of the reverse method. Then the character at index k in the new character sequence is equal to the character at index $n-k-1$ in the old character sequence.

Q10)(D)

Lambda expression is an inline code that implements a functional interface without creating a concrete or anonymous class. A lambda expression is basically an anonymous method. Lambdas reduces the lines of code. It supports sequential and parallel execution by passing behaviour in methods with collection stream API. Using `Stream` API and lambda expression we can achieve higher efficiency with parallel execution in the case of bulk operations on collections. Another advantage of lambdas is deferred execution. Like most imperative programming languages, Java evaluates a method's arguments eagerly, but we should consider lazy alternative for avoiding needless expensive computation. Lambdas enable us for defining, passing, and storing blocks of code for later execution. For example, when logging, if the `Logger` interface has support for lambda expressions, client code can lazily log messages without explicitly checking if the requested log level is enabled.

Q11)(D)

The code outputs `"true 2"`. A new `StringBuilder` is created and initialized with the value `"-"`. The following line appends another `"-"` to the `StringBuilder`. The `append()` method, appends the specified string to `StringBuilder` character sequence. The characters of the `String` argument are appended, in order, increasing the length of sequence by the length of the argument. If `str` is `null`, then the four characters `"null"` are appended.

Q12)(D)

The `add()` and `get()` are both `List` interface methods, but `length()` method is a `String` method also present in the `StringBuilder` class. So, none of the given options are applicable types. Ignoring that fact, we would have used `ArrayList<String>` for the method parameter to be able to use `get()` method without casting, otherwise we would have to cast to `String` because `get()` method would have returned objects of type `Object`, not

String. Because `get()` is a generic method, we would have use `ArrayList<String>`, so the `get()` method can return `String` objects.

Q13)(D)

A functional interface is an interface which allows only one Abstract method within the Interface scope. There are some predefined functional interface in Java like `Predicate`, `Consumer` etc. When writing lambdas, we can omit the curly braces, and get rid of the `return` keyword. The parentheses can only be omitted if there is a single parameter and its type is not explicitly stated. Java does this because developers commonly use lambda expressions this way and they can do as little typing as possible. We can omit braces when we only have a single statement. We do this with if statements and loops. What is different here is that the rules change when we omit the braces. Java doesn't require us to type `return` or use semicolon when no braces are used. The syntax of lambdas is tricky because many parts are optional. These three lines of code fo the exact same thing:

```
Predicate<StringBuilder> p = (StringBuilder b) -> {return true;};
Predicate<StringBuilder> pr = (StringBuilder f) -> true;
Predicate<StringBuilder> prd = f -> true;
```

Q14)(A)

The `contains()` method is a `List` interface method which returns `true` if list object on which it's invoked contains the specified element. More formally, returns `true` if and only if the list contains at least one element `e` such that `Objects.equals(o, e)`. The `set(int index, Character element)` method is a `List` interface method that replaces the element at the specified position in the list on which it's invoked with the specified element. The code creates a list, then adds two character to it, replacing the second character with `'c'`, and then removes the first element in the list, leaving just a one element in the list which is `'c'`. And since we removed the character `'b'` form the list, `chars.contains('b')` returns `false` summing up to output `"1 false"`.

Q15)(D)

The code does not compile. String `b` is initialized with the value `"12"` and then using addition assignment operator appended with `"3"`. Then the `reverse()` method of `StringBuilder` is used to reverse the characters in the `StringBuilder` object. But the problem is `reverse()` method is a method of `StringBuilder` class as stated, also present in the `StringBuffer` class since it the thread safe version of the `StringBuilder`.

Q16)(A)

Syntax rules of lambdas is discussed in the earlier questions. Only one of them fails to compile because the parentheses can only be omitted if there is a single parameter and its type is not explicitly stated. The `String` type must be omitted to make the code compile. Other lambdas are essentially the same and all have correct syntax.

Q17)(A)

The code prints `true`. `Target` interface is a Functional interface since it has only one abstract method. The `prepare` static method accepts `Target` interface type as its second argument, so lambda expression can be used here, and as seen, a lambda was passed to it and it has a valid syntax. It matches the `needToAim` method defined in the `Target` interface, taking a `double` value and returning a `boolean` value. In the `prepare` method we see that the `45` was passed to `needToAim`, so the `45` is `d` in the lambda so to say. And the given expression evaluates to `true` according to the boolean condition given in lambda. Don't forget that, we are implementing the `needToAim` when we need it. Like anonymous classes lambdas are anonymous methods with a more concise and clear syntax.

Fundamentally, a lambda expression is just a shorter way of writing an implementation of a method for later execution. We can define for example, a `Runnable` which uses anonymous inner class syntax but it will clearly suffers from a "vertical problem", meaning that the code takes too many lines to express the basic concept. Functional interfaces are interfaces that require exactly one method to be implemented in order to satisfy the requirements of the interface. This is how the syntax achieves its brevity, because there is no ambiguity around which method of the interface the lambda is trying to define.

Q18)(A)

The code outputs `"694"`. The `concat` method of the `String` class concatenates the specified string to the end of this string. If the length of the argument string is 0, then this `String` object is returned. Otherwise, a `String` object is returned that represents a character sequence that is the concatenation of the character sequence represented by this `String` object and the character sequence represented by the argument string. The problem is since the `String` objects are immutable, every time some string is concatenated to a `String` object, a new `String` object is returned that represents the concatenated `String` object, so we are not actually changing the original object. We need to assign it to some variable to grab a hold of it, but as seen, in the code that is not the case.

Q19)(A)

`java.util` package contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes. `String`

class resides in the `java.lang` package while the `LocalDate` class is in the `java.time` package. Only the `ArrayList` is in the `java.util` package from the given options.

Q20)(C)

The `delete` method of `StringBuilder` class removes the characters in a substring of this sequence. The substring begins at the specified start and extends to the character at index `end - 1` or to the end of the sequence if no such character exists. If start is equal to end, no changes are made. In option B, `delete` method deletes everything after the character `'r'`. Consequent `append` and `insert` are adjusted to build up a `"radical robots"` `StringBuilder` object. Option D outputs the same character sequence too because it inserts the `"robots"` string starting at the end of the `StringBuilder` without replacing the space character residing at the end of it. Option C outputs `"radicalrobots"`, because the insertion is made at the index of the space character, resulting in replacement of the space character, hence the aforementioned output.

Q21)(A)

The code outputs true. The array variable is initialized with an array literal containing two string elements in it. Then a `List` representation of it is assigned to the `museums` variable. Then the element at index 0 is replaced with a `"Art"` value and since the `contains` method checks if the given parameter is in the `List`, `print` statement return true. The `asList(E... a)` method returns a fixed-size list backed by the specified array. Changes made to the array will be visible in the returned list, and changes made to the list will be visible in the array. Essentially it creates a wrapper that implements `List` interface, which makes the original array available as a list. Nothing is copied, only a single wrapper object is created. If some element is overwritten in the `List`, it gets overwritten in the original array. Some `List` operations aren't allowed on the wrapper, like adding or removing elements from the list, it's only allowed to read or overwrite the elements. `ArrayList` resulting from `Arrays.asList()` is not the type `java.util.ArrayList`, but `java.util.Arrays$ArrayList` which does not extend `java.util.ArrayList`, only extends `java.util.AbstractList`. It's a different kind of object. The `java.util.ArrayLists` have their own, internal array in which they store their elements and are able to resize the internal array. The wrapper does not have its own internal array, it only propagates operations to the array given to it.

Q22)(D)

Considering the methods invoked on the object, `s` should be a `String` object. The `contains()` method returns true if this string contains the specified sequence of char values. The `equals()` method compares string to the specified object. The result is true if and only if the argument is not null and is a `String` object that represents the same sequence of characters as the object. The `startsWith()` method of `String` class is used for checking prefix of a `String`. It returns a boolean value true or false based on whether the given string begins with the specified letter or word.

If some string contains the char sequence "abc", it does not necessarily mean that the string is equal to "abc". Also it does not mean that the aforementioned string starts with "abc". If some string starts with "abc" character sequence, it does not necessarily mean that the string consists merely of "abc", but it is guaranteed that the `contains()` will return `true`.

Q23)(D)

The code does not output anything because it does not compile. The `length()` method is undefined for the type `List`. It is defined for `String` objects, and can also be used for `StringBuilder` objects. We should use the `size()` method to get the intended result, which would be 1. The initial list size is 2, then one of the elements is replaced with some other character and then the first element of the list is removed, leaving only one element in the array. Hence the `size()` method would return 1.

Q24)(B)

The type of the argument `mystery` should be `String`. `replace` and `startsWith` methods are `String` methods, and are undefined for types `ArrayList` and `StringBuilder` while the `toString()` is inherited to all objects since it is defined in `Object` and all other classes extends that `Object` class. `StringBuilder` actually has a `replace()` method but the signature is different from the one used in the question.

Q25)(B)

Diamond operator aka diamond syntax was introduced in Java 7 as a new feature. Purpose of diamond operator is to simplify the use of generics when creating an object. Generics exist to provide compile-time protection against doing the wrong thing. The diamond operator is a nice feature as you don't have to repeat yourself. It makes sense to define the type once when you declare the type but just doesn't make sense to define it again on the right side. The diamond operator, allows the right hand side of the assignment to be defined as a true generic instance with the same type parameters as the left side, without having to type those parameters again.

If we insert the diamond operator at the position P, we need to specify the type parameter too, after that we don't need to specify the type parameter again in the right side inside the diamond operator. We can insert the diamond operator at position Q, without compilation error although it would give warnings because then, the left side is using raw types.

Q26)(D)

`Predicate` functional interface has one method with the signature `boolean test(T t)`. It has one argument of type `T`. We see that the type parameter has given as type `String` when declaring the `predicate` variable "pred". So there should be one parameter in lambda with the type `String` and the lambda should return a `boolean` value for the compiler to map the

given lambda expression to `boolean test(T t)`. The option C has the correct type but there already another variable with the same name in the enclosing scope which is the main method's parameter. Interesting and relevant note to this subject is lexical scoping. One thing that is new with the lambdas is how the compiler treats names (identifiers) within the body of a lambda compared to how it treated them in an inner class. Lambdas are lexically scoped, meaning that a lambda recognizes the immediate environment around its definition as the next outermost scope.

Q27)(A)

The code outputs `"false 1"`. Since the `Strings` are immutable, every method that looks like mutating the `strong` object is actually creates a new object. So the `anotherLine` variable is actually refers to a new object. Because of that `line == anotherLine` evaluates to `false` since they do not refer to same object. And `line.length()` returns `1` since we have already said, `strings` are immutable, `line` variable is left unchanged from the operation before.

Q28)(C)

The code does not compile. We get the compilation error the method `startsWith()` is undefined for the type `Object`. `Predicate` is a generic interface, but it's used as a raw type, without any type parameter given. But on the right hand side of the lambda expression (body of the lambda), we see that it is assumed by the programmer that the parameter `c` is a `string`. We pick that out from the invocation of `startsWith()` method on the `c` variable. But because no type parameter is given, the type of argument of the method `boolean test(T t)` is assumed by the compiler as type `Object`. So we cannot reasonably call a `String` method from a `Object` reference.

Q29)(B)

Java `LocalDate` class is an immutable class that represents `Date` with a default format of `yyyy-MM-dd`. Java `LocalDateTime` class is an immutable date-time object that represents a date-time, with the default format as `yyyy-MM-dd-HH-mm-ss.zzz`.

Java `LocalTime` class is an immutable class that represents time with a default format of `hour-minute-second`. There is no class such as `LocalTimeStamp` but `TimeStamp`.

Q30)(D)

The `charAt(int index)` method of `StringBuilder` returns the `char` value at the specified index. The `builder` variable is reassigned to substring starting from the index 4 which in this case is `"2"`. The `substring(int beginIndex)` method returns a `string` that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

There is no char value at the index **2**, because `builder`'s length is just **1**, so the maximum available index is **0**. Trying to access a non existing index throws `StringIndexOutOfBoundsException` at runtime.

Q31)(D)

When we are using braces to wrap the code, we need to explicitly `return` a value from the body of the lambda. We should also explicitly terminate the statement with semicolon as shown: `i -> { return i !=0; };`

We can omit the braces only if we have a single statement, like we do with the loops and if statements. But unlike them, if we have a single statement, java doesn't require us to type `return` or use a semicolon to terminate the statement.

Q32)(B)

`LocalDate` class static `LocalDate of(int n, int n2, int n3)` Obtains an instance of `LocalDate` from a year, month and day. This returns a `LocalDate` with the specified year, month and day-of-month. `plusDays(long daysToAdd)` Returns a copy of this `LocalDate` with the specified number of days added. Since the returned copy is not assigned to any reference type variable, the object to which `xmas` variable refers to is unchanged.

Q33)(A)

The code gives the output 'e'. The `StringBuilder` method `delete(int start, int end)` removes the characters in a substring of sequence. The substring begins at the specified start and extends to the character at index `end - 1` or to the end of the sequence if no such character exists. `deleteCharAt(0)` deletes the first character which is 'r' leaving "ed" in the character sequence. `delete(1, 2)` deletes the character starting from index **1** until the index **2** exclusive, which in this case corresponds to character 'd' in the remaining "ed" character sequence.

Q34)(B)

The code prints `false`. We are not going to get involved with some semo-detailed explanation since the relevant information is given in the answer to question twenty-eight. This code runs without any issue. The compiler treats the argument as type `Object`. Luckily we use a method that exists in the `Object` class, so although the compiler still complains about the usage of a raw type, allows invoking the `equals(Object obj)` since the `equals()` method is a `Object` class member method.

Q35)(C)

Most types at java which use indexes for accessing its elements use zero based indexes. These types also includes includes arrays and strings. `LocalTime` is an immutable date-time object that represents a time, often viewed as hour-minute-second. This class does not store or represent a date or time-zone. Instead, it is a description of the local time as seen on a wall clock, so the answer can't be option D.

`LocalDateTime` contains both a date and time but no time zone. One of the overloaded method signatures is as follows:

```
public static LocalDate of(int year, Month month, int dayOfMonth)
```

`Month` is an enum representing the 12 months of the year. In addition to the textual enum name, each month-of-year has an int value from 1 (January) to 12 (December). So the answer must be option C.

Q36)(C)

`Predicate` is a Functional interface which represents a predicate, boolean-valued function. Functional interfaces have only one abstract, unimplemented, method. A functional interface can contain default and static methods which do have an implementation, in addition to the single unimplemented method. `Predicate` functional interface has one abstract method which has the signature `boolean test(T t)`. Method has the return type `boolean`, and takes one argument.

Q37)(B)

`Period` class is used to define an amount of time with date-based values (years, months, days). The `Period` class provides various get methods, such as `getDays`, `getMonths` to extract the amount of time from the period. We can't chain methods when creating a `Period`. In the following example, only the last method is used because the `Period.ofX` methods are static methods.

```
// It represents just three days.
Period period1 = Period.ofWeeks(1).ofDays(3);
This code is somewhat equivalent to following:
Period wrong = Period.ofYears(1);
wrong = Period.ofWeeks(7);
```

Q38)(B)

The `DateTimeFormatter` class in java is used for parsing dates in different formats. We can use this class to format date in a specified format or we can use the predefined instance of `DateTimeFormatter`. In the question a format variable is defined with the format

"MM-dd-yyyy". Also a `Period` type variable with a period of 1 day if defined. Then this period is subtracted from `newYear LocalDate` object which is the date January 1, 2017. And then the resulting date is formatted with the formatter we defined, and printed as "12-31-2016" according to the format.

Q39)(C)

The `trim()` method returns a string whose value is the string object which it has invoked upon, with any leading and trailing whitespace removed. So, the new string after trimmed is " :) - (:". For the question string to be equal to `really` string, we need to pick the substring starting from the index 1, until the `length()-1` exclusive. The way we won't take the leading and trailing white spaces in the String, making it equals to the trimmed version `really`.

Q40)(C)

It is true that a `Period` class is typically used for adding or subtracting time from dates. `Period` class is used to define an amount of time with date-based values like year, month or day. The supported units of a period are YEARS, MONTHS and DAYS. Also, the `Period` class is immutable and thus thread-safe. For smaller units of time like minutes or seconds there is also a class named `Duration`.

Q41)(D)

The code prints the value '4'. First line in the main method creates a `StringBuilder` object with the "54312" character sequence. Second line returns a new string that contains a substring starting from index 2, which is the substring "312", but it is not assigned to any variable so the builder is left unchanged. Lastly, third line basically prints the character at index 1, which is the character '4'.

Q42)(B)

The code prints the List [3, 2]. Three values are added to the List `pennies`. Then the item at the index 2 is removed leaving the list with two items which are [3, 2]. There is another overloaded version of the method `remove(Object obj)` which removes the first occurrence of the given object in the list. Because of the autoboxing feature and our list holds `Integer` objects, `int` primitive type is also compatible with the method `remove(Object o)`. So, we could think that the item '2' will be removed from the list instead of the item at index 2. So, which method are we invoking actually? We are invoking the `remove(int index)` method, because it is an exact match so it runs instead.

Q43)(C)

The type of the argument must be `StringBuilder`. All of the invoked methods on the arguments are `StringBuilder` class member methods. We should note that `charAt` and `length` methods are also present in the `String` class and `ArrayList` has `size` method rather than the `length` method. But `String` class had no member method such as `insert`.

Q44)(C)

`LocalTime` is an immutable date-time object that represents a time, often viewed as hour-minute-second. Time is represented to nanosecond precision. For example, the value `"13:45.30.123456789"` can be stored in a `LocalTime`. This class does not store or represent a date or time-zone. With the `plusXXX()` and `minusXXX()` method we can return a copy of this `LocalTime` with the specified number of hours, minutes, seconds, and nanoseconds. So the smallest unit we can add to a `LocalTime` object is nanosecond.

Q45)(D)

The code compiles but throws `UnsupportedTemporalTypeException` at runtime with a useful stack trace information `"Unsupported field: MinuteOfHour"`. Patterns are based on a simple sequence of letters and symbols. A pattern is used to create a `Formatter` using the `ofPattern(String)` and `ofPattern(String, Locale)` methods. All letters `'A'` to `'Z'` and `'a'` to `'z'` are reserved as pattern letters. The problem in the code is that, in the pattern `"mm-dd-yyyy"`, `'m'` is a reserved pattern for minute-of-hour. But the `LocalDate` class does not store or represent time.

Q46)(D)

All of the three types can be passed as a parameter to the `replace()` method on the `String` class. There is two overloaded `replace` method with the signatures, `replace(char oldChar, char newChar)` and `replace(CharSequence target, CharSequence replacement)`. `CharSequence` is a interface that for representing objects as readable sequence of char values. This interface provides us uniform, read-only access to many different kinds of char sequences. `String` and `StringBuilder` both implements `CharSequence` interface and hence they are both `CharSequence` objects. Thus can be passed to `replace` method.

Q47)(C)

The code does not compile, thus no lines are output. Problem is the `print` method accepts a `Predicate` with the generic type parameter `Integer`, but the list contains `String` objects. We are passing these `String` objects to a method (`boolean test(T t)`, `T` is `Integer` in this case) which does expect `Integer` objects. Java does not allow this, so we encounter a compiler error. So we need to change the type parameter given to `Predicate` in the `print` method's signature to `String` type. But we should note an importing note, even if did that

the code would not compile still, Because then another compilation error would arise, the given lambda expression would be trying to compare a `String` with an `int`.

Q48)(D)

The code compiles but throws an `IndexOutOfBoundsException` at runtime. Two elements have been added to the `List` `magazines` and then they have been removed with the `clear()` method. Then yet another `String` has been added to the `magazines` list, and with the following line after that, `remove()` method has been invoked with the parameter `1`, which is the index of the element that is going to be removed. The problem is the list only contains 1 element now, thus only the index 0 is accessible. So it throws a runtime exception.

Q49)(C)

The code does not compile, complaining about a type mismatch. The `witch` reference variable is declared type `String` but it is being tried to be initialized with a `char` type literal. Having this error corrected, code prints the `String` `"black"`, since it basically concatenates the two strings and prints it.

Q50)(C)

The code does not compile. There is no method such as `setYear` for the class `LocalDate`. Since we have pointed out earlier somewhere in the questions, `LocalDate` class is immutable and thread-safe. It is reasonable to think that an immutable class not having setter methods for its fields.