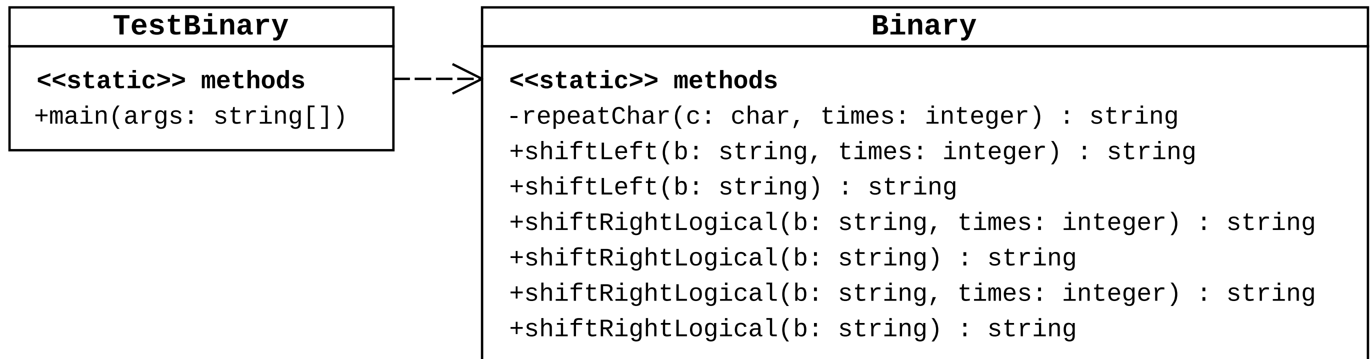
Project: Binary Strings and `class Binary` (*part 1*)

©2025 Chris Nielsen – *www.nielsenedu.com*

For this coding project, we will write a Java class that will manipulate binary numbers. Each number is to be stored in a Java `String`, and represented as a string of characters, each character being either a “`0`” or a “`1`”. For example, we might declare a `String` variable and initialize it to a string that represents a binary number like this:

```
String twentyOne = "10101";
```

Write all methods in a single class named `Binary`, and write code to test those methods in a separate class (or classes). Keeping the code for a particular task together, and separated from other code, is called ***encapsulation***. It’s an important vocabulary word for this course. In this case, we’ll be *encapsulating* the code related to manipulating binary strings in a class named `Binary`. Below is a *UML Class Diagram* of the classes you are to code for this assignment. You may name your test class whatever you please.



1. For the first part of the project, you will first use the Java `String` class methods to implement the methods: `shiftLeft`, `shiftRightLogical`, and `shiftRightArithmetic`.

a) `repeatChar`

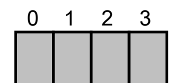
Before we write these methods, we’re going to write a “*helper*” method. A ***helper*** method is a method, which is usually `private`, that we can call to perform some task that will make coding our `public` methods easier. Methods that are `private` cannot be called from a different class.

The helper method we will write is to be named `repeatChar`, and it is to takes two parameters: a character (type `char`), and an integer that gives the number of times to repeat that character. It is to return a `String` that contains the character repeated the correct number of times. From the given information you should be able to deduce the method header, but it is given here to make sure you get it correct:

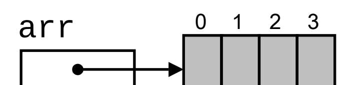
```
private static String repeatChar(char c, int times)
```

If, for example, we call the method with “`repeatChar('a', 5)`”, the value returned should be equal to “`aaaaa`”.

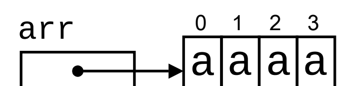
To implement this, you will need to first declare an array of the appropriate size. Recall that we can instantiate an empty array of characters with enough space to store four characters with “`new char[4]`”. This is shown diagrammatically immediately to the right of this text. The shading suggests the content has not been initialized.



To declare a variable named `arr` that contains a reference to an instantiation of an empty array of characters with enough space to store four characters, we use the statement: “`char[] arr = new char[4];`” This is shown diagrammatically immediately to the right of this text.



Once you have declared the empty array of characters, write a `for` loop to initialize each element of the array to the character given by the `char` parameter `c`. The final result is shown diagrammatically immediately to the right of this text.



We have constructed an array of characters with the desired values. To convert this array of characters to a `String`, use: “`new String(arr)`”. Finally, return this result.

To test this code, you can either temporarily make the method `public` and write some test code to test it from a different class, or you can create a `public` tester method within the `Binary` class, perhaps named `testRepeatChar`, write some code in it to test `repeatChar`, and call the tester method from another class. The `Binary` class should not contain a main method.

Guard Statements

Guards watch an area, protecting the objects inside the boundaries of a building or country. They ensure the safety of important things, such as people, buildings, treasure, or data.

In computer programming, a ***guard statement*** works in a similar way. It's a rule that checks whether something is true before allowing a program to do something. A guard decides if a particular person can enter based on the conditions; the *guard statement* decides if the program should enter a block of code to run it based on a condition. If the condition of the *guard statement* is true, the program continues; if it's false, it is prevented from continuing.

As an example, the following partial method code contains a *guard statement* to ensure the parameter named `s` is not equal to `null`. If the value of `s` is `null`, the guard statement prevents the program from executing the method by exiting the method, returning a value of `false`. If `s` is not `null`, the program continues to execute the remaining code inside the method.

```
public static boolean setValue(String s) {
    // Guard statement:
    if( s == null ) {
        return false;
    }
    // Remainder of method body not shown
}
```

b) `repeatChar` – error checking

In your test code for `repeatChar`, did you include a test to check what would happen if you called `repeatChar` and requested a negative number of characters? If you did not explicitly add a check to see if the method were called with a negative value for `times`, your method will likely crash with an exception called `NegativeArraySizeException` when the code tries to declare an array with a negative size. To fix this, add a *guard statement* that returns an empty string (`""`) if a negative value is passed in the parameter `times`. Actually, you can make the condition to include both negative values or zero, since the loop will return an empty string if a value of zero is passed – by returning an empty string when times is equal to `0`, we make the code ever so slightly more efficient.

Test your code to ensure it will return an empty string (and not crash) if a negative value is passed in the parameter named `times`.

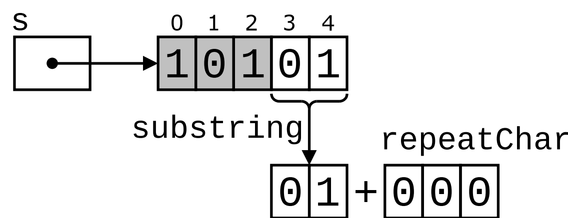
©2025 Chris Nielsen – *www.nielsenedu.com*

c) `shiftLeft`

Recall that when shifting a binary number left, the procedure is the same for both *unsigned* and *signed* numbers, thus the same for both *logical* and *arithmetic* shift. We will thus just implement a single `shiftLeft` method. For completeness, you could also create a `shiftLeftLogical` and `shiftLeftArithmetic`, and have those methods simply call the `shiftLeft` method. (You should generally not duplicated code when possible.)

Actually, there is to be two `shiftLeft` methods (overloaded methods). The first method is to take two parameters: a `String` parameter that contains a binary string to shift, and an integer parameter that is to contain the number of bit positions to shift the binary string. The method is to return the shifted binary string. The second method takes only a single parameter: a `String` parameter that contains the binary string to shift. This second method is to shift the binary string by one bit position. There is no reason to duplicate code – the second method can simply call the first method and specify the number of bit positions to shift to be `1`.

The body of the `shiftLeft` method must use the `String` class method `substring` to appropriately cut the `String` containing the binary number, simulating those bits shifted out of the number, and use the helper method `repeatChar` that you wrote in part (a) to generate a string of "`0`" characters to fill the spaces where the numbers were shifted out. This algorithm is shown diagrammatically below using the example of the binary string "`10101`" is shifted left by `3` bit positions. As can be seen in the diagram, the correct binary string is created when the result of `repeatChar` is concatenated to the end of the result of `substring`.



Example Test Code for `shiftLeft`

```
String original = "10101";
String shifted = Binary.shiftLeft(original, 3);

System.out.println(original + " shifted left by 3 is " + shifted );
```

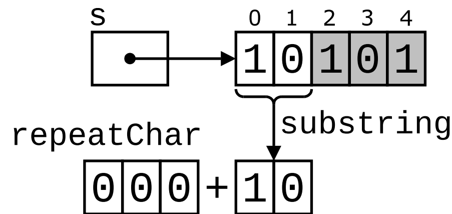
Output of Example Test Code for `shiftLeft`

```
10101 shifted left by 3 is 01000
```

©2025 Chris Nielsen – *www.nielsenedu.com*

d) `shiftRightLogical`

When performing a logical shift right, for each bit position shifted, the value shifted in on the left side is `0`. The code for this method will be very similar to that for `shiftLeft`, except the correct binary string is created when the result of `substring` is concatenated to the end of the result of the call to method `repeatChar`.



Example Test Code for `shiftRightLogical`

```
String original = "10101";
String shifted = Binary.shiftRightLogical(original, 3);

System.out.print(original + " logically shifted right by 3 is ");
System.out.println(shifted );
```

Output of Example Test Code for `shiftRightLogical`

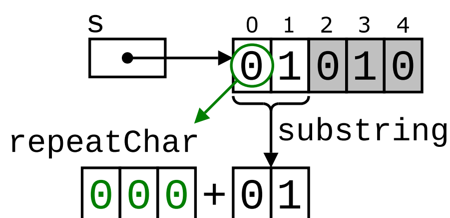
```
10101 logically shifted right by 3 is 00010
```

e) `shiftRightArithmetic`

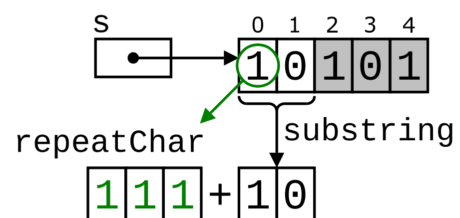
When performing an arithmetic shift right, for each bit position shifted, the value shifted in on the left side is equal to the *sign bit* (which is the *most significant bit*, or left-most bit). The code for this method will be very similar to that for `shiftLeftLogical`, except we need to determine the correct value, `0` or `1`, to pass to `repeatChar`. As with `shiftLeftLogical`, the correct binary string is created when the result of `substring` is concatenated to the end of the result of the call to method `repeatChar`.

The two diagrams below show arithmetic shifting `3` bit positions when the most significant bit is `0` (the result is the same as logical shifting `3` bit positions) and arithmetic shifting `3` bit positions when the most significant bit is `1`.

***Arithmetic shift by* `3` *bit positions when the MSB is* `0`**



***Arithmetic shift by* `3` *bit positions when the MSB is* `1`**



Example Test Code for `shiftRightArithmetic`

```
String original = "10101";
String shifted = BinaryString.shiftRightArithmetic(original, 3);

System.out.print(original);
System.out.print(" arithmetically shifted right by 3 is ");
System.out.println(shifted);

original = "01010";
shifted = BinaryString.shiftRightArithmetic(original, 3);

System.out.print(original);
System.out.print(" arithmetically shifted right by 3 is ");
System.out.println(shifted);
```

Output of Example Test Code for `shiftRightArithmetic`

```
10101 arithmetically shifted right by 3 is 11110
01010 arithmetically shifted right by 3 is 00001
```

f) You were asked to add a *guard statement* for `repeatChar` so that the program would not crash if the method were passed a negative number. If you did not already add a guard statement on your shift methods, consider whether they should have them or not, and if you were to add a guard statement, what value would you return if the user of the `Binary` class asked to shift by a negative number. Below I give you a few options, and it is up to you to define how your `Binary` class will react.

- Write a guard statement to return the value `null`, which means no `String` is returned. The user who called the shift method can check the return value, and they know the operation failed if `null` has been returned.
- Return the original `String` parameter without shifting. The user of the `Binary` class will need to verify if the value was shifted or not.
- Allow the program to crash – the programmer who uses the `Binary` class should never call the method asking to shift a negative number of times.
- Shift the binary string in the opposite direction: a `shiftRightLogical` by `-3` is the same as `shiftLeftLogical` by `+3`.

Whichever way you choose to implement your version of the `Binary` class, be prepared to explain and defend the reasons you chose to do it that way.