Checking for Style:

1. For all topics, we first check whether the student code has some minimum functionality that allows us to evaluate it for the style. If a code does not meet this, the style will be “cannot evaluate.” Here, “functionality” doesn’t mean that the code needs to compile (unless it is explicitly stated). When coding for style, “functionality” refers to the semantic meaning that the code most likely holds for the author, even if there are syntax errors that would prevent the code from compiling or runtime errors that would make the code crash.
2. Guidelines are for checking the style pattern that the code was meant to target - code can have an expert style for that pattern and a novice style for other issues.

Returning Boolean expressions (with method call or operator)

Writing Prompt, Boolean expression with method call	Writing Prompt, Boolean expression with operators
write a function that takes an int as input and returns a boolean. ● when input is 7 return true. ● otherwise return false.	write a function that takes a String as input and returns a boolean. ● when input starts with ”A”, return true. ● otherwise return false. **Hint:** word.startsWith(”x”) returns true when the String word starts with ”x”.

Expert style:

Directly returning Boolean expressions

Non-expert style examples:

1. if(condition) return true; else return false;
2. condition? true: false;
3. while (condition) return true; return false;

Functionality requirements to be evaluated for style:

1. Must compile with researchers’ minimum fixations.
2. Does not need to check the condition correctly (e.g., can compare the input to a number other than 7, or check if the string starts with something other than ‘A’). However, the condition must be formed properly (input for isSeven must be compared to a number, input for startsWithA needs to check if the first letter in the input is equal to another letter).

3. For the target pattern with an operator, must return the Boolean expression using an operator, and for the return Boolean with a method call, the code must use a method call

Example	Style Comments

`char input = word.charAt(0);`
`if(c == "A"){`
`return true;`
`}`
`else{`
`return false;`
`}`
	Novice Style: Although the variable names make the code not functional, students' usage of if-else statements (rather than returning the boolean expression) confirms that student used a novice style.

`if (word.startsWith("x") {`
`return true;`
`} else {`
`return false;`
`}`
	Novice Style: does not need to check correct condition

`if(word != null && word.charAt(0) == 'A')`
`return true;`
 	
`return false;`
	Novice Style for boolean returns. Expert styled for another topic: nested ifs vs. &&

`return word.startsWith("A");`	**Expert Style**
`if(word.startsWith("A")` `return true;`	**Cannot Evaluate**: returns true but not false

`if(num != 7)`
`return false;`
 	
`return true`
	Novice Style

`return num==7;`	**Expert Style**

`if (num == true)`
`return true`
`return false`
	Cannot Evaluate. Does not compare num to another number.

`return (num == 7 ? true : false);`	**Novice Style**

`public boolean isSeven(int num) {`
`return num.Equals(7);`
`}`
	Cannot evaluate for Boolean returns with operators. The response does not include an operator.

`public boolean isSeven(int num) {`
`return if (num == 7);`
`}`
	Cannot Evaluate

Repeating content in if and else (ishness)

Writing Prompt, unique vs. repeated code within an if and else
write a function that takes a String as input and returns a String. ● For input String that end in “sh”, concatenate the ending “ishness” and return a message saying how long the new word is. ● Write the function so that it would be easy for someone else to modify. **Hints**: word.endsWith(“sh”) will return true if the word ends with (“sh”) ● The message should be returned not printed. ● You may use + to concatenate strings. For example: String word2 = word1 + “ there”; If word1 is “hello”, word2 is “hello there”

Functionality requirements to be evaluated for style:

1. Must construct a string (printing and creating a string builder counts as constructing)
2. All final string constructions must include the original word
3. If the input string meets the condition, the constructed string must include **either** the original word with an ending (e.g., “<word>-ishness”) **or** the correct length of the modified word [pending: for ‘the correct length of the modified word’, the modified word doesn’t need to be the original word plus an ending as long as it’s based on the original word. E.g., it could be X letters from the original word plus some ending]
4. If the input string does not meet the condition, the constructed string must include a second instance of the original word **or** the correct length of the original word
5. All variables used in the constructed string must be assigned a value in both cases (when the input string meets the condition or does not meet the condition). If the variable is assigned a value but there is a scoping issue, ignore the scoping.
6. There must be a condition that checks for something. The ‘something’ can be super badly formed, but cannot be if(true) or if(false).

Functionalities that are NOT required for expert style:

1. Condition does not need to be correct, but must check if the input ends with some string (e.g., may check if the input ends with “h” or “elephant” rather than “sh”). The condition may be incorrectly formatted such that it would throw a runtime exception (e.g., index of string out of range error), or such that it would always be true or false due to a mistake (e.g., using the substring method to identify a part of a word, and then comparing that to “sh,” but the indices in the substring method were wrong so that the identified part of the word is only one letter long - this will always evaluate to false, but it's because of a reasonable mistake.)
2. Constructed string does not need to include both the modified word and length of the modified word
3. The modified word needs to have some ending, but doesn’t need to be “-ishness”

4. Constructed string does not need to be returned
5. Does not need to check if the String is null
6. Variables in the constructed string can be out of scope as long as they are assigned a value, no matter what the input is.

Cannot Evaluate:

- This is for code that doesn't fit into a style category because it doesn't have the minimum required functionality(e.g., the code doesn't repeat but we can't say it is expert-styled because the author left the code unfinished). This category is somewhat more subjective. Strong candidates for this category are:
    - Only one word or number is returned/printed for at least one of the cases (see examples below)
    - A full message is returned, and it looks similar to expert code because there is no or minimal repeated code and there is only one return, but it does not have the functionality of maintaining the original word in all cases and/or the modifications depending on the ending.

Cannot evaluate
 	
Does not maintain the original word
	`string endingToken = "sh";`
	`string addedToken = "-ishness";`
	`if(word.endsWith(endingToken))`
	`  word = word + addedToken;`
	`string messageString = "Your word was "+ word+ ". The length of " + word + " is " + word.length+ ".";`
	`return messageString;`

NOVICE
 	
Because the original word is maintained in the variable "word", even though it is printed next to "ishness".
	`***THIS WILL COUNT AS NOVICE***`
	
	`string suffix = "-ishiness";`
	
	`if (word.endswith("sh"){`
	`  return "Your word was " + word +suffix + ". The length of " + word +suffix " is " + word.length + suffix.length + ".";`
	`} else {`
	`  return "Your word was " + word + ". The length of " + word " is " + word.length + ".";`
	`}`

Cannot evaluate
 	
Mix of return and print.
	`if(word.endsWith("sh")){`
	`  System.out.println(word.length() + "-ishness".length());`
	`  return word + "-ishness";`
	`}`
	`System.out.println(word.length());`
	`return word;`

Cannot evaluate	`String newWord = null;` `if (word.charAt(word.length) == 'h') &&`

Only returns one word in each case	`(word.charAt(word.length-1) == 's') {`
	`  return word + "-ishness";`
	`} else {`
	`  return word;`
	`}`

Cannot evaluate
 	
only returns one word/number
	`if (word.Substring(word.Length - 2, 2).equals("sh")){`
	`  word += "-ishness";`
	`  return word;`
	`} else {`
	`  return word.Length.toString();`
	`}`

Expert styled:

- Max one return statement or print statement
- No hardcoding for calculating string length
- Can have slight repeated code (e.g, first example in the expert-styled code)
- Can have variables that are not initialized/declared

Expert
Expert - nonDestructive
	`public String ishness(String word) {`
	`String temp = word;`
	`if (word.endsWith("sh"))`
	`temp = word + "-ishness";`
	
	`return "Your word was " + word + ". The length of " + temp + " is " + temp.length() + ".";`
	`}`

Expert
Expert - nonDestructive
	`if(word.endsWith("sh"))`
	`wordEdit = word + "-ishness";`
	`int length = word.length();`
	`String sentence = "Your word was " + word + " . The length of " + wordEdit + " is " + length + "."`
	`return sentence;`

Expert
Expert - sequence
	`String output = "Your word was " + word + ". The length of " + word;`
	`if(word.endsWith("sh"))`
	`word += "-ishness"`
	`output += word + " is " + word.length;`
	`return output;`

Expert
Expert - nonDestructive [it is destructive of word, but preserves it in finalWord, so same idea]
	`String end = "sh";`
	`String add = "-ishness";`
	`String finalWord = word;`
	
	`if (word.endsWith(end))`
	`word = word + add;`
	`return "Your word was " + word +`

	". The length of " + finalWord + " is " + finalWord.size();

Expert
Expert - both (string constructed in a sequence and also not destructive)
	String ending = "sh";
	String output = "Your word was " + word + ". The length of ";
	String returnWord = word;
	if (word.endsWith(ending)){
	returnWord = returnWord + "-ishness";
	}
	return output + returnWord + " is " + returnWord.length() +
	".";

Middle-styled: (For the paper, we decided to consider this category as expert-styled)

- It is possible to calculate length once. Middle code is constructed in a way that requires length to be calculated twice. For example:
    - Hard coding the length calculation (leading to two different paths for calculating length when one path is possible)
    - Adding the ending on the string without modifying any variable, because it makes it impossible to calculate the length with one path
    - Calculating length twice

Middle-styled code examples

```
String end = (word.endsWith("sh")) ? "-ishness" : "";
return "Your word was " + word + ". The length of " word + end + " is " +
(word.length() + end.length()) + ".";
```

```
int len = word.Length;
 String wordd = "Your word was " + word + ". The length of " + word;
 if(word.Substring(word.Length - 2) == "sh"){
   wordd = wordd + "-ishness";
   len += 8;
 }
 wordd = wordd + " is " + len +".";
 return wordd;
```

```
string returnMessage = "You word was " + word + ". The length of " + word;
if (word.endsWidth("sh")
  returnMessage += "-ishness";
returnMessage += " is " + word.length + ".";
return returnMessage;
```

```
String ending = "";
if(word.endsWith("sh")) {
   ending = "-ishness";
}
return "Your word was " + word + ". The length of " + word + ending + " is "
+ word.length() + ".";
```

```
 String result = null;
 String temp = null;
 int size = word.length();
 if(word.endsWith("sh")){
  temp= word + "-ishness"
  size = size + 8;
 }
 result = "Your word was " + word + ". The length of " + temp + " is " +
size;
 return result;
```

```
 String original = "Your word was " + word + "." + " The length of " + word;
 int length = word.length();
 if (word.endsWith("sh")){
   String suffix = "-ishness";
   original += "-ishness";
   length += suffix.length();
 }
 original += " is " + length;
 return original;
```

```
 string lookFor = "sh";
 string endString = "-ishness"
 string repeatWord = "Your word was " + word + ". ";
 string giveLen = "The length of " + word;
 int length = word.Length();
 if (word.endsWith(endString)){
   length += 8;
   giveLen = giveLen + endString;
 }
 return repeatWord + giveLen + " is " + length + ".";
```

```
int length = word.length;
 String newWord = word;
 if(word.charAt(length-2) =="s" && word.charAt(length-1) == "h"){
 newWord = newWord+("-ishness");
 }
 length = newWord.length;
 String output = "Your word was " + word +". The length of " + newWord + "
is " + length.toString;
 return output;
```

calculating string length twice even though toReturn was modified, so could have just calculated length on to return rather than on word and toAppend.

```
string toLookFor = "sh";
string toAppend = "-ishness";
string toReturn = word;
int stringLength = word.Length;

if (word.endsWith(toLookFor))
 {
  toReturn = toReturn + toAppend;
  stringLength += toAppend.Length;
```

```
 }
 return "Your word was " + word + ". The length of " + toReturn + " is " +
stringLength;
```

```
String result = "Your word was " + word + "The length of " ;
if(word.endsWith("sh"){
  result = result + word + "-ishness ";
}else{
  result = result + word;
}
result = result + "is " + word.length;
return result;
```

Novice:

- More than one return
- Duplicate string construction

Novice-styled code examples

```
if(word.endsWith("sh")) {
   String word2 = word + " -ishness";
   String result = "Your word was"+ word +". The length of" + word2 "is " +
word2.length()+".";
 return result;
}else {
   String result = "Your word was"+ word +". The length of" + word "is " +
word.length()+"."
```

```
string wordEnding = "sh";
string output = "Your word was " + word + ". The length of " + word;

for (int i = 0 ; i > wordEnding.Length; i++) {
   if (wordEnding[0] != word[word.Length - 1 - i]) {
      return output + " is " + word.Length + ".";
   }
}
return output + "-" + wordEnding + " is " + (wordEnding.Length +
word.Length) + ".";
```

```
String result = "Your word was " + word + "The length of " ;
if(word.endsWith("sh"){
   result = result + word + "-ishness ";
}else{
   result = result + word;
}
result = result + "is " + word.length;
return result;
```

Style coding for salePrice

salePrice targets three patterns, however, we only talk about repeated code as it was one of the topics discussed in the paper.

- Nested ifs vs. &&:
    - `if (item.equals("socks")) {`
    - `  if (coupon) {`
    - `  return beginning + "since you have a coupon, "`
    - `  + ending + price * (1 - specialSale);`
    - `  }`
    - `}`
- Series of if-statements for exclusive cases instead of if-elses
- Repeated code

Each pattern should be coded separately.

Code editing prompt asked students to edit the style of following code:

```
public static String salePrice(String item, double price, boolean coupon) {
      double sale = .25;
      double specialSale = .5;
      double over50sale = .35;
      String beginning = "Your item, " + item + ", usually costs " + price
                        + ", but ";
      String ending = "you are getting it for ";
      if (item.equals("socks")) {
            if (coupon) {
                  return beginning + "since you have a coupon, "
                        + ending + price * (1 - specialSale);
            }
      }
      if (price > 50) {
            return beginning + "since it is over 50, "
                  + ending + price * (1 - over50sale);
      }
      if (price <= 50) {
            return beginning
                  + ending + price * (1 - sale);
      }
      return "Error.";
}
```

salePrice- repeated code

Minimum functionality required for evaluation:

- Code must have the potential to return at least two different string values, where the value of the string depends on the input parameters.

- The values for the strings must all have at least some part in common. (E.g., all start with the text “Your item”.

Expert style for this pattern:

- Must have only one statement that returns the desired string (for this pattern, can still have expert style if the line `return “error”;` is left in)
- All code that can be factored out is.
- Price calculation must be factored out

Good:

- Must have only one statement that returns the desired string (for this pattern, can still have expert style if the line `return “error”;` is left in)
- Price calculation does not need to be factored out
- Beginning, ending must be factored out
- No lines that are exact copy-paste between the blocks
- Has some duplication that prevents it from being Expert

Fair:

- Factored out some content but not enough for the Good category.
- It is not sufficient to factor out the return statement if the string construction is repeated within each block.

Novice style:

- Nothing is factored out
- If the return statement is factored out, it still counts as Novice if the string construction is repeated within each block.

Code	**Style**

`String beginning = "Your item, " + item + ", usually costs " + price + ", but ";`
`String reason;`
`String middle = "you are getting it for ";`
`String ending = "";`
 	
`double newPrice;`
`double amountSaved;`
 	
`if (item.equals("socks") && coupon)`
`{`
`newPrice = price * (1 - specialSale);`
`amountSaved = price - newPrice;`
`reason = "since you have a coupon, ";`
`}`
`else if (price > 50)`
`{`
	Fair. The line **amountSaved = price - newPrice;** is exactly copy-pasted between each block. It is not Novice because **beginning** and **middle** are factored out.

`  newPrice = price * (1 - over50sale);`
`  amountSaved = price - newPrice;`
`  reason = "since it is over 50, ";`
`  }`
`  else`
`  {`
`  newPrice = price * (1 - sale);`
`  amountSaved = price - newPrice;`
`  reason = "";`
`  }`
 	
`  return beginning + reason + middle + newPrice + ending;`	

`String beginning = "Your item, " + item + ", usually costs " + price`
`  + ", but ";`
`  String ending = "you are getting it for ";`
 	
`  if (item.equals("socks") && coupon) {`
`  savings = price * specialSale;`
`  return beginning + "since you have a coupon, "`
`  + ending + (price - savings);`
`  }`
`  if (price > 50) {`
`  savings = price * over50sale;`
`  return beginning + "since it is over 50, "`
`  + ending + (price - savings);`
`  }`
`  if (price <= 50) {`
`  savings = price * sale;`
`  return beginning`
`  + ending + (price - savings);`
`  }`
`  return "Error.";`
`}`
	Novice. Nothing that was in the original prompt is factored out.

`if (item.equals("socks")) {`
`  if (coupon) {`
`  customerSale = specialSale;`
`  saleMessage = "since you have a coupon, ";`
`  }`
`  }`
 	
`  if (price > 50) {`
	Expert style. One return statement puts together the string, so there's no redundancy in multiple lines combining the string. This code has a functionality bug where both ifs will execute, so customerSale from the first block will never be used, so strictly speaking the first assignment of customerSale is redundant,

`  customerSale = over50sale; // misspelled over50sale`
`  saleMessage = "since it is over 50, ";`
`  }`
`  else {`
`  customerSale = sale;`
`  }`
 	
`  return beginning + saleMessage + ending + price * (1 - customerSale);`
	but this is a separate issue from having repeated code.

`String output = beginning;`
`  double appliedSale = 0;`
 	
`  if (item.equals("socks")) {`
`  if (coupon) {`
`  output += "since you have a coupon, ";`
`  appliedSale = specialSale;`
`  }`
`  }`
`  if (price > 50) {`
`  output += "since it is over 50, " ;`
`  appliedSale = over50sale;`
`  }`
`  else {`
`  appliedSale = sale;`
`  }`
`  output += ending + price * (1 - appliedSale);`
`  return output;`
	Expert. The student builds output incrementally, but never has repeated cases of adding the exact same things on. As above, we ignore the fact that the line “appliedSale = specialSale” gets overwritten later.

`  if (item.equals("socks")) {`
`  if (coupon) {`
`  beginning = beginning + "since you have a coupon, ";`
`  endPrice = price * (1 - specialSale);`
`  }`
`  }`
`  else if (price > 50) {`
`  beginning = beginning + "since it is over 50, ";`
`  endPrice = price * (1 - over50sale);`
`  }`
`  else if (price <= 50) {`
`  endPrice = price * (1 - sale);`
`  }`
`  else {`
`  return "Error.";`
`  }`
	Good.The only thing that prevents it from being Expert is that the price calculation is not factored out.

`return beginning + ending + endPrice;`	