2020 AP Computer Science A - Summer

Packet Topics Table of Contents

1. logical operators, constructs & methods, pp. 2-9
2. arrays, for_loops & methods, pp. 10-27
3. Strings, String methods & user methods, pp. 28-51
4. Comprehensive Review of Java Syntax, Primitive Types, Operators, Arrays, Strings, and Programming Basics, pp. 52-63
5. Note that all the programs discussed in 1-3 above are CodingBat.com programs that can be run and tested by creating an account on CodingBat.com. So, it's not necessary (though desirable) to implement these programs in Eclipse. However, familiarity with what 'System.out.println("Hello World");' will print is necessary (see pp. 55-60).
6. Intensive study of these notes and working as many of the CodingBat.com Logic, Array & String problems as possible, should be very good preparation for the first APCS test which will be given in the second week of class. Good Luck and have a Great Summer!

Java Language: Some Basic Elements

Numeric Operators

+	-	*
/	%	++
--		

Numeric Comparators

<=	>=	==
<	>	!=

Grouping & Punct. Symbols

{	}	;
(	)	,
[	]	

Boolean Operators

true	false
&&	||
!	

Control Operators

if	else
for	while
return	break
continue	

Assignment Operators

=	+=
-=	

Data Types

int	double
long	String
char	boolean

Math Function Library

Math.sqrt( )	Math.pow( )
Math.abs( )	Math.PI
Math.sin( )	Math.cos( )
Math.exp( )	Math.log( )
Math.min( )	Math.max( )

String Operators & Methods

+	" "
length( )	substring( )
equals( )	equalsIgnoreCase( )
charAt( )	compareTo( )

Object Specifiers, Operators & Methods

class	static
public	private
new	.
main( )	equals( )
toString( )	

System Print Methods

System.out.print( )
System.out.println( )
System.out.printf( )

Array Operators & Methods

a[*i*]
new
length
clone( )
toString( )

Convert String to Number

Integer.parseInt( )
Double.parseDouble( )

Java Language: Logic 1

Problem 1. The parameter **weekday** is true if it is a weekday, and the parameter **vacation** is true if we are on vacation. We sleep in if it is not a weekday or we're on vacation. Construct a Java statement that defines a Java variable, **sleepIn**, that is true if we sleep in.

Note that parameters are variables (values) provided by other parts of the Java program. Boolean variables can have only two values: **true** or **false**. Here is the correct Java statement:

boolean sleepIn = (!weekday || vacation);

The Java statement above defines a variable of type boolean and assigns it's value to the value of the Java version of the statement: "not weekday or vacation". Here is a table with a complete breakdown of the Java language elements of this statement:

<table>
<tr><td>boolean</td><td>sleepIn</td><td>=</td><td>(</td><td>!</td><td>weekday</td><td>||</td><td>vacation</td><td>)</td><td>;</td></tr>
<tr><td>data type</td><td>variable name</td><td>assign. operator</td><td>grouping symbol</td><td>not operator</td><td>param. name</td><td>or operator</td><td>param. name</td><td>grouping symbol</td><td>punct. symbol</td></tr>
</table>

Problem 2. We have two monkeys, and we are in trouble if they are both smiling or if neither of them is smiling. The parameter **aSmile** is true if **monkey a** is smiling and **bSmile** is true if **monkey b** is smiling. Use these parameters to construct Java statements defining the variables **bothSmile** and **neithSmile**. Then use these two variables to construct a Java statement that defines the variable **inTrouble**.

boolean bothSmile = (aSmile && bSmile);

boolean	bothSmile	=	(	aSmile	&&	bSmile	)	;
data type	variable name	assign. operator	grouping symbol	param. name	and operator	param. name	grouping symbol	punct. symbol

boolean neithSmile = (!aSmile && !bSmile);

boolean	neithSmile	=	(	!	aSmile	&&	!	bSmile	)	;
data type	variable name	assign. oper.	group. symbol	not oper.	param. name	and oper.	not oper.	param. name	group. symbol	punct. symbol

So, to complete the problem we use the two variables defined above.

boolean inTrouble = (bothSmile || neithSmile);

Note that we also could have skipped the first two steps, and instead, defined **inTrouble** using the following statement.

boolean inTrouble = ((aSmile && bSmile) || (!aSmile && !bSmile));

Problem 3. We have a loud talking parrot. The parameter **talking** is true if the parrot is talking. The **hour** parameter is the current hour time in the range 1 to 24. We are in trouble if the parrot is talking and the hour is before 7 or after 20. Use these parameters to construct a Java statement defining the variable **inTrouble.**

boolean inTrouble = (talking && ((hour < 7) || (hour > 20));

Note that in this statement we also use the < comparator to compare the value of the **hour** parameter to the integer **7** and the > comparator to compare the value of the **hour** parameter to the integer **20**. Each of these component statements, namely **(hour < 7)** and **(hour > 20)** , will have boolean values of **true** or **false** depending on the integer value of the integer parameter **hour**. Remember, parameters are variables whose values are provided by some other part of the Java program.

Problem 4. We are given two integer parameters **a** and **b**. Use these parameters to construct a Java statement defining the variable **makesTen**, which is true if the value of either one of the parameters is 10, or if their sum is 10.

boolean makesTen = ((a == 10) || (b == 10) || (a + b == 10));

Notice the difference between the single equal sign and the double equal sign. In the statement **(a == 10)** the double equal sign tests whether or not the value of the parameter **a** is the integer **10**. The other two statements on the right also are logical tests that depend on the values of the parmeters **a** and **b**. These three statements are then combined using the operator **||** to create the complete logical test statement, which assigns the value **true** or **false** to the variable **makesTen**. So, the difference is that the ***single equal sign*** always ***assigns*** a value to a variable, but the ***double equal sign tests*** the value of an already existing variable (possibly more than one variable).
If we wanted to assign the value 10 to a new integer variable **a**, we would use the statement: **int a = 10;**

Homework. Here are some more problems for you to do on your own.

Problem 5. Given two temperatures (integer parameters) **temp1** and **temp2**, construct a Java statement defining the variable **icyHot**, which is true if one is less than 0 and the other is greater than 100.

Problem 6. Given an integer (parameter) **n**, construct a Java statement defining the variable **nearHundred**, which is true if **n** is within 10 of either 100 or 200. Note that **Math.abs(x)** returns the absolute value of a number, which can be used to simplify this problem.

Problem 7. Given two integer parameters **a** and **b**, construct a Java statement defining the variable **in1020**, which is true if either of them is in the range from 10 to 20 inclusive.

Problem 8. Let's say that a number is "teen" if it is in the range from 13 to 19 inclusive. Given three integer parameters **a**, **b**, and **c**, construct a Java statement defining the variable **hasTeen**, which is true if one or more of **a**, **b**, and **c** are teen.

Problem 9. As in Problem 8, a number is "teen" if it is in the range from 13 to 19 inclusive. Given two integer parameters **a**, and **b**, construct a Java statement defining the variable **loneTeen**, which is true if one or the other is teen, but not both.

Java Language: Logic 2

In this section we will extend our Java Logic development to include logical flow of control using **if**, **else**, and **return**. In Java parlance, a Java "method" is essentially a ***mathematical function*** or ***procedure*** that produces (returns) an answer based on given logical conditions and mathematical calculations. However, note that not all Java methods return an answer, e.g. a method may simply update an already existing internal variable such as an average or a total.

The problems in this section will address the construction of Java code fragments (statements) that could be used to define a Java method. These code fragments are not complete Java methods, but just part of them. Later, we will use these code fragments to create complete Java methods.

Problem 1. Given two integer parameters **a** and **b**, return their sum if they aren't equal, otherwise return double their sum. Construct a Java code fragment (statements) for a Java method that will return the correct answers under the given conditions.

general format	long way 1	long way 2	short way 1	short way 2

if (*condition*)
{
  ....
  ....
  return *ans1*;
}
else
{
  ....
  ....
  return *ans2*;
}
	int ans1 = a + b;
	int ans2 = 2*ans1;
	
	if (a != b)
	{
	  return ans1;
	}
	else
	{
	  return ans2;
	}
		int ans1 = a + b;
		int ans2 = 2*ans1;
		
		if (a == b)
		{
		  return ans2;
		}
		else
		{
		  return ans1;
		}
			if (a != b)
			  return (a + b);
			else
			  return 2*(a + b);
				if (a == b)
				  return 2*(a + b);
				else
				  return (a + b);

In the table above we have given examples of four different correct solutions. Notice that in the two long versions we have defined integer variables **ans1** and **ans2**, which strictly speaking isn't necessary here but often in more complex situations it will be very helpful to do so. Also, in the two long versions we have included the curly braces, which again strictly speaking isn't necessary here, since there is only one statement following the logical flow of control operators **if** and **else**. However, if there is more than one statement following one these logical operators then the curly braces must be included, otherwise the Java code won't be interpreted correctly and hence won't run correctly.

Problem 2. Given an integer parameter **n**, return the absolute difference between **n** and **21**, except return double the absolute difference if **n** is over **21**. Construct a Java code fragment (statements) for a Java method that will return the correct answers under the given conditions.

way 1	way 2

if (n > 21)
  return 2*Math.abs(n-21);
else
  return Math.abs(n-21);
	if (n <= 21)
	  return Math.abs(n-21);
	else
	  return 2*Math.abs(n-21);

Problem 3. Given two integer parameters **a** and **b**, and a boolean parameter **negative**, if **negative** is true then return true if **a** and **b** are both negative. If **negative** is false, return true if **a** and **b** have opposite signs. Otherwise, return false. Construct a Java code fragment (statements) for a Java method that will return the correct answers under the given conditions.

<table>
<tr><th>way 1</th><th>way 2</th></tr>
<tr><td>

```
if (negative)
{
    if (a < 0 && b < 0)
      return true;
    else
      return false;
}
else
{
    if (a*b < 0)
      return true;
    else
      return false;
}
```

</td><td>

```
if (negative)
  return (a < 0 && b < 0);
else
  return (a*b < 0);
```

</td></tr>
</table>

Notice that, since in each of the statements **if (a < 0 && b < 0)** and **if (a*b < 0)** we are testing a boolean (logical) condition and returning true or false based on whether or not the condition itself is true or false, we may simply return the boolean value of the logical condition itself ! As you can see, there are often several ways to write Java code that works. One of the most appealing and interesting aspects of Java programming is to try to find and create the most elegant (logically clear, concise) solution possible.

Homework. Here are some more problems for you to do on your own. In each one, construct a Java code fragment (statements) for a Java method that will return the correct answers under the given conditions.

Problem 4. Given three integer parameters **a**, **b**, and **c**, return the largest one.

Problem 5. Given two integer parameters **a** and **b**, return whichever value is nearest to the value 10, or return 0 in the event of a tie. Hint: Use **Math.abs(x)**.

Problem 6. Given two positive integer parameters **a** and **b**, return the larger value that is in the range from 10 to 20 inclusive, or return 0 if neither is in that range.

Problem 7. Given two integer parameters **a** and **b**, return false if one is negative. Return true if they aren't negative and have the same last digit, such as with 27 and 57. Otherwise, return false. Note that the "mod" operator, **%**, computes remainders, so **17 % 10** is **7**, i.e. 17 mod 10 is the remainder after dividing 17 by 10, which is 7.

Java Language: Logic 3

In this section we continue our Java Logic development to include logical flow of control using **if**, **else if**, **else**, and **return**. Again, the problems in this section will address the construction of Java code fragments (statements) that could be used to define a Java method. These code fragments are not complete Java methods, but just part of them. Later, we will use these code fragments to create complete Java methods.

Problem 1. You and your date are trying to get a table at a restaurant. The parameter **you** is the stylishness of your clothes and the parameter **date** is the stylishness of your date's clothes, both in the range from 0 to 10. You and your date get the table based on the following fashion assessment: if either of you is very stylish, 8 or more, then return 2 (yes) with the exception that if either of you has style 2 or less, then return 0 (no). Otherwise, return 1 (maybe).

<table>
<tr><th>general format</th><th>way 1</th><th>way 2</th></tr>
<tr>
<td>if (condition1)
{
  ....
  return ans1;
}
else if (condition2)
{
  ....
  return ans2;
}
else
{
  ....
  return ans3;
}</td>
<td>if (you <= 2 || date <= 2)
{
   return 0;
}
else if (you >= 8 || date >= 8)
{
   return 2;
}
else
{
   return 1;
}</td>
<td>if (you <= 2 || date <= 2)
   return 0;

if (you >= 8 || date >= 8)
   return 2;

return 1;</td>
</tr>
</table>

<table>
<tr><th>general format</th><th>way 3</th><th>way 4</th></tr>
<tr>
<td>int answer;
if (condition1)
{
  ....
  answer = ans1;
}
else if (condition2)
{
  ....
  answer = ans2;
}
else
{
  ....
  answer = ans3;
}
return answer;</td>
<td>int getTable;

if (you <= 2 || date <= 2)
{
   getTable = 0;
}
else if (you >= 8 || date >= 8)
{
   getTable = 2;
}
else
{
   getTable = 1;
}

return getTable;</td>
<td>int getTable = 1;

if (you >= 8 || date >= 8)
   getTable = 2;

if (you <= 2 || date <= 2)
   getTable = 0;

return getTable;</td>
</tr>
</table>

In the tables above we have given examples of four different correct solutions to Problem 1. Our purpose here is both to illustrate the use of the **if** .... **else if** .... **else** .... construction, and to show that there are many ways to write Java code that works in a given situation. Way 2 probably seems like the simplest solution for this specific problem. However, it is important to note that way 3 is a very important template that will be, in general, the most useful for more complex Java programs. Comparing way 3 and way 4 it is important to realize that way 4 is less efficient, since every lone **if** statement will be tested. In contrast, using the **if** .... **else if** .... **else** .... construction, once a condition has been found to be true nothing below it is tested. Furthermore, the logical structure of way 3 is much clearer than way 4. In fact, if we switch the two **if** statements in way 4, then the program is no longer correct (why?). On the other hand, if we use the logical structure in way 3, then when testing the condition in an **else if** or **else** statement the negation of all the conditions that occurred above is automatically built in.

Problem 2. Suppose we are given three integer parameters **a**, **b**, and **c**. We want to return their sum, except that if one of the values is the same as another of the values, it does not count towards the sum.

way 1	way 2

**if (a != b && a != c && b != c)
{
    return a + b + c;
}
else if (a != b && a != c)
{
    return a;
}
else if (a != b && b != c)
{
    return b;
}
else if (a != c && b != c)
{
    return c;
}
else
{
    return 0;
}**
	**if (a != b && a != c && b != c)
	    return a + b + c;
	
	if (a != b && a != c && b == c)
	    return a;
	
	if (a != b && a == c && b != c)
	    return b;
	
	if (a == b && a != c && b != c)
	    return c;
	
	return 0;**

The two solutions in the table above for Problem 2 further illustrate the nice properties of the **if** .... **else if** .... **else** .... construction. In particular, we see that using way 1, when testing the condition in an **else if** or **else** statement the negation of all the conditions that occurred above is automatically built in. This means that, in contrast to way 2, we can simplify the expression for the logical condition in each **else if** statement.

Homework. Here are some more problems for you to do on your own. In each one, construct a Java code fragment (statements) for a Java method that will return the correct answers under the given conditions.

Problem 3. Suppose we are given three integer parameters **a**, **b**, and **c**. We want to return their sum, except that if one of the values is 13 then it does not count towards the sum and values to its right do not count. So for example, if **b** is 13, then both **b** and **c** do not count.

Problem 4. Given two positive integer parameters **a** and **b**, return whichever value is nearest to 21 without going over. Return 0 if they both go over.

Problem 5. You are driving a little too fast, and a police officer stops you. Write code to return the type of ticket encoded as an integer: no ticket = 0, small ticket = 1, big ticket = 2. If the parameter **speed** is 60 or less, then return 0 (no ticket). If **speed** is between 61 and 80 inclusive, then return 1 (small ticket). If **speed** is 81 or more, then return 2 (big ticket). Unless the parameter **birthday** is true, then since it is your birthday, your speed can be 5 higher in all cases.

Problem 6. Your cell phone rings and you are to use the boolean parameters **isMorning**, **isMom**, and **isAsleep** to determine whether or not to answer it. Normally you answer, except in the morning you only answer if it is your mom calling. In all cases, if you are asleep, you do not answer. Return true or false based on these conditions.

Problem 7. Suppose you have a red lottery ticket and three integer parameters **a**, **b**, and **c** each representing values 0, 1, or 2 on the ticket. If they are all the value 2, then return the payoff 10. Otherwise, if they are all the same, then return the payoff 5. Otherwise, so long as both **b** and **c** are different from **a**, then return the payoff 1. Otherwise, return the payoff 0.

Problem 8. Suppose you have a green lottery ticket and three integer parameters **a**, **b**, and **c** each representing values on the ticket. If the numbers are all different from each other, then return the payoff 0. If all of the numbers are the same, then return the payoff 20. If two of the numbers are the same, then return the payoff 10.

Problem 9. Suppose you have a blue lottery ticket and three integer parameters **a**, **b**, and **c** each representing values on the ticket. This makes three pairs, which we will call ab, bc, and ac. Consider the sum of the numbers in each pair. If any pair sums to exactly 10, then return the payoff 10. Otherwise, if the ab sum is exactly 10 more than either bc or ac sums, then return the payoff 5. Otherwise, return the payoff 0.

Eclipse Programs: Arrays Part 1

Often a program must deal with a large amount of data. Fortunately, data can usually be organized and processed systematically. *Arrays* are very useful for organizing and storing data. This chapter will discuss arrays and give some examples of how they work.

Topics:

- *The idea of arrays.*
- *Array declaration.*
- *Array declaration and construction.*
- *Using arrays.*
- *Automatic bounds checking.*
- *Initializer lists.*

QUESTION 1:

Say that you are writing a program that reads and stores 100 numbers. Are you happy to write 100 input statements with 100 different variables?

A good answer might be:

Probably not. It would be nice to have some organized way of reading and storing the values.

Picture of an Array

	data
0	23
1	38
2	14
3	-3
4	0
5	14
6	9
7	103
8	0
9	-56

An *array* is an object that can be used to store a list of values. It is made out of a contiguous block of memory that is divided into a number of "slots." Each slot can hold a value, and all the values are of the same type (for example, primitive type *int*.) The picture shows an array.

The name of this array is *data*. The slots are indexed 0 through 9, and each slot holds an *int*. Each slot can be accessed by using its *index*. For example, `data[0]` is the slot which is indexed by zero (which contains the value 23), `data[5]` is the slot which is indexed by 5 (which contains the value 14).

Important Facts:

- Indexes always start at zero, and count up by one's until the last slot of the array.
- If there are ***n*** slots in an array, the indexes will be ***0*** through ***n-1***.

QUESTION 2:

What value is in `data[7]` ?

What value is in `data[7]` ?

Answer: 103

Using Arrays

index	data
0	23
1	38
2	14
3	-3
4	0
5	14
6	9
7	103
8	0
9	-56

Every slot of an array holds a value of the same type. So, for example, you can have an array of *int*, an array of *double*, and array of *String*, and so on.

The picture on the left shows an array of *int.* Every slot contains an *int*. A slot of this array can be used anywhere a variable of type *int* can be used. For example,

```
data[3] = 99 ;
```

works just like an assignment statement with an *int* variable on the left of the assignment operator. After it has been executed, the array will look like the picture on the right.

The value in slot 3 of the array has been changed.

index	data
0	23
1	38
2	14
3	99
4	0
5	14
6	9
7	103
8	0
9	-56

QUESTION 3:

What do you suppose is the value of the arithmetic expression:

```
data[2] + data[6]
```

What do you suppose is the value of the arithmetic expression:

```
data[2] + data[6]
```

A good answer is:

23 --- data[2] contains a 14 and data[6] contains a 9, the sum is 23

Arithmetic Expressions

index	data
0	23
1	38
2	14
3	99
4	0
5	14
6	9
7	103
8	0
9	-56

A slot of an array that contains a numeric type (such as *int*) can be used anywhere a numeric variable can be used. In an arithmetic expression such as the one in the question, the number in the designated slot of the array is used.

An arithmetic expression can contain a mix of literals, variables, and array slots. For example, if *x* contains a 10, then

```
(x + data[2]) / 4
```

evaluates to (10+14) / 4.

Here are some other legal statements:

```
data[0] = (x + data[2]) / 4 ;

data[2] = data[2] + 1;

x = data[3]++ ;

data[4] = data[1] / data[6]
```

QUESTION 4:

Assume that the array holds values as in the picture. What will be the result of executing the statement:

```
data[0] = data[6] + 8;
```

What will be the result of executing the statement:

```
data[0] = data[6] + 8;
```

A good answer is:

The value 17 is put into slot 0 of *data*.

Arrays are Objects

Array declarations look like this:

```
type[] arrayName;
```

This tells the compiler that *arrayName* will be used as the name of an array containing *type*. However, the actual array is not constructed by this declaration. Often an array is declared and constructed in one statetment like this:

```
type[] arrayName = new type[ length ];
```

This tells the compiler that *arrayName* will be used as the name of an array containing *type*, and *constructs an array object containing* length number of slots.

An array is an object, and like any other object in Java is constructed out of main storage as the program is running. The array constructor uses different syntax than most object constructors; *type*[*length*] names the type of data in each slot and the number of slots. Once an array has been constructed, the number of slots it has does not change.

For example, the example array we have been using might have been declared and constructed like this:

```
int[] data = new int[10];
```

This statement creates the array *data* and puts a zero into each slot.

QUESTION 5:

1. What is the length of the array *data*?
2. What are the indexes of *data*?

```
int[] data = new int[10];
```

A good answer is:

1. What is the length of the array *data*? ***10***
2. What are the indexes of *data*? ***0..9***

Bounds Checking

Recall that:

```
The length of an array is how many slots it has.
An array of length n has slots indexed 0..(n-1)
```

Indexes must be an integer type (since it makes no sense to speak of slot number 1.59, say.) It is OK to have spaces around the index of an array, for example `data[1]` and `data[ 1 ]` are exactly the same as far as the compiler is concerned.

Say that an array were declared:

```
int[] data = new int[10];
```

Then it is *not legal* to refer to a slot that does not exist:

- data[-1] ***illegal***
- data[10] ***illegal*** (given the above declaration)
- data[1.5] ***illegal***
- data[0] ***OK***
- data[9] ***OK***

If you have one of the above illegal expressions in your program, your program will not compile. Often the size of an array is not known to the compiler (since the array is constructed as the program is running, its length does not need to be known to the compiler.) However, if your running program tries to refer to a slot that does not exist, an exception will be thrown, and (unless another part of your program does something about it) your program be terminated.

QUESTION 6:

Here is a declaration of another array:

```
int[] scores = new double[25];
```

Which of the following are legal?

- scores[0]
- scores[1]
- scores[-1]
- scores[10]
- scores[25]
- scores[24]

```
int[] scores = new double[25];
```

Which of the following are legal?

A good answer is:

- scores[0] ***OK***
- scores[1] ***OK***
- scores[-1] ***illegal***
- scores[10] ***OK***
- scores[25] ***illegal***
- scores[24] ***OK***

More on Array Declaration

Lacking any other information, the slots of an array are initialized to the default value for their type, so each slot of a numeric array is initialized to zero.

Of course, the program can explicity place values into slots after the array has been constructed:

```
public class MainArray01 {

  public static void main ( String[] args )
  {
    int[] stuff = new int[5];

    stuff[0] = 23;
    stuff[1] = 38;
    stuff[2] = 7*2;

    System.out.println("stuff[0] has " + stuff[0] );
    System.out.println("stuff[1] has " + stuff[1] );
    System.out.println("stuff[2] has " + stuff[2] );
    System.out.println("stuff[3] has " + stuff[3] );
  }
}
```

(Create this program above using Eclipse)

QUESTION 7:

What does the above program write to the console window in Eclipse?

A good answer is:

```
stuff[0] has 23
stuff[1] has 38
stuff[2] has 14
stuff[3] has 0
```

Using a Variable as an Index

The index of an array is always an integer type. It does not have to be a literal. It can be any expression that evaluates to an integer. For example, the following are legal:

```
int values[] = new int[7];
int index;

index  = 0;
values[ index ] = 71;       // put 71 into slot 0

index  = 5;
values[ index ] = 23;       // put 23 into slot 5

index  = 3;
values[ 2+2 ] = values[ index-3 ];  // same as values[ 4 ] = values[ 0 ];
```

Using an expression for an array index is a very powerful tool. Often a problem is solved by organizing the data into arrays, and then processing that data in a systematic way using variables as indexes. Here is a further example:

```
public class MainArray02 {

  public static void main ( String[] args )
  {
    double[] val = new double[4];  //an array of doubles

    val[0] = 0.12;
    val[1] = 1.43;
    val[2] = 2.98;

    int j = 3;
    System.out.println( val[ j   ] );
    System.out.println( val[ j-1 ] );

    j = j-2;
    System.out.println( val[ j   ] );
   }
}
```

(Create this program above using Eclipse)

QUESTION 8:

What does the above program output to the console window in Eclipse?

A good answer is:

```
0.0
2.98
1.43
```

More Complicated Example

Here is a more complicated example of array subscripting:

```
public class MainArray03 {

  public static void main ( String[] args )
  {
    double[] val = new double[4];

    val[0] =  1.5;
    val[1] = 10.0;
    val[2] = 15.5;

    int j  = 3;
    val[j] = val[j-1] + val[j-2];   // same as val[3] = val[2] + val[1]

    System.out.println( "val[" + j + "] = " + val[j] );

   }
}
```

(Create this program above using Eclipse)

QUESTION 9:

What does the above program print out?

A good answer is:

```
val[3] == 25.5
```

Initializer Lists

You can declare, construct, and initialize the array all in one statement:

```
int[] data = {23, 38, 14, -3, 0, 14, 9, 103, 0, -56 };
```

This declares an array of *int* which will be named *data*, constructs an *int* array of 10 slots (indexed 0..9), and puts the designated values into the slots. The first value in the *initializer list* corresponds to index 0, the second value coresponds to index 1, and so on. (So in this example, data[0] gets the 23.)

You do not have to say how many slots the array has. The compiler will count the values in the initializer list and make that many slots. Remember that once an array has been constructed, the number of slots it has does not change. (But of course you can change what is in the slots.)

QUESTION 10:

Write a declaration for an array of *double* named "dvals" that is initialized to contain 0.0, 0.5, 1.5, 2.0, and 2.5.

A good answer is: `double[] dvals = { 0.0, 0.5, 1.5, 2.0, 2.5 };`

Several Arrays per Program

A program can have any number of arrays in it. Often values are copied back and forth between the various arrays. Here is an example program that uses two arrays:

```
public class MainArray04 {

  public static void main ( String[] args )
  {
    int[] valA = { 12, 23, 45, 56 };

    int[] valB = new int[4];

    __________ = ____________ ;

    __________ = ____________ ;

    __________ = ____________ ;

    __________ = ____________ ;


   }
}
```

QUESTION 11:

Fill in the blanks so that the values in *valA* are copied into the corresponding slots of *valB*.

Fill in the blanks so that the values in *valA* are copied into the corresponding slots of *valB*.

A good answer is:

```
public class MainArray04 {

  public static void main ( String[] args )
  {
    int[] valA = { 12, 23, 45, 56 };

    int[] valB = new int[4];

    valB[ 0 ]  = valA[ 0 ] ;
    valB[ 1 ]  = valA[ 1 ] ;
    valB[ 2 ]  = valA[ 2 ] ;
    valB[ 3 ]  = valA[ 3 ] ;

   }
}
```

(Create this program above using Eclipse)

Copying Values in Slots

In this example, the int in slot 0 of valA is copied to slot 0 of valB, the int in slot 1 of valA is copied to slot 1 of valB, and so on.

The ***following statements do NOT do the same thing*** as the statements in the program above:

```
int[] valA = { 12, 23, 45, 56 };

int[] valB = new int[4];

 valB = valA ;   // point to exactly the same memory location
```

Remember that ***arrays are objects***. The statement above will merely copy the object reference for the variable `valA` into the object reference for the variable `valB`, resulting in two ways to access the single array object, i.e. two variables that reference or point to ***exactly the same memory location*****.**
So, a change in the value of either variable will change the value of the other.

QUESTION 12:

Say that the statements above have been executed. What would the following two statements do?

```
valA[2] = 999;
System.out.println( valA[2] + "    " + valB[2] );
```

A good answer is:

Since *valA* and *valB* both refer to the same object (memory location), valA[2] and valB[2] are two ways to refer to the same slot which is changed for both variables, not just *valA*. The statements print out:

```
999    999
```

Eclipse Programs: Arrays Part 2

QUESTION 1:

How (in general) could you print out every element of an array with 100 elements?

A good answer is: This sounds like a good place for a counting loop.

Counting Loops and Arrays

Remember that in Java, the index of an array starts at 0 and counts up to one less than the number of elements in the array. This is, of course, exactly what counting loops can do. Here is a program that does that, except for a blank or two:

```
public class MainArray05 {

  public static void main ( String[] args )
  {
    int[] arr = { 2, 4, 6, 8, 10, 1, 3, 5, 7, 9 };

    for ( int index= __________ ; _______________ ; ____________ )
    {
      System.out.println( ___________________ );
    }
  }
}
```

QUESTION 2:

Can you fill in the blanks so that the program prints out every element, in order?

Fill in the blanks so that the program prints out every element, in order.

A good answer is:

```
public class MainArray05 {

  public static void main ( String[] args )
  {
    int[] arr = { 2, 4, 6, 8, 10, 1, 3, 5, 7, 9 };

    for ( int index= 0 ; index < 10 ; index++ )
    {
      System.out.println( arr[ index ] );
    }
  }
}
```

(Create this program above using Eclipse)

The *length* of an Array

It is annoying to have to count how many elements there are in an array. Worse, this might not be known when the program is being written. Array objects are created as the program is running and can be created with any number of elements. It is essential, sometimes, to write a program that can deal with an array who's size is not known until the program is running.

This can be done by asking the array how many elements it has. Remember that an array is an object. As an object, it has more in it than just the slots. An array object has a member *length* that is the number of slots (number of elements) it has. The `for` statement can be written like this:

```
for ( int index= 0 ; index < arr.length; index++ )
```

Lines of code similar to the above are *very* common in programs. One dimensional arrays are very common. Usually a program will use many of them. Almost always a program will "visit" each element of an array using a `for` loop such as the above.

QUESTION 3:

Can you fill in the blanks in this line of code so that the elements of the array are visited from the last element down to element 0?

```
for ( int index= ___________ ; ________________ ; _____________ )
```

Fill in the blanks in this line of code so that the elements of the array are visited from the last element down to element 0:

A good answer is:

```
for ( int index= arr.length-1 ; index >= 0 ; index-- )
```

Reading in Each Element

Here is a program that prompts the user and reads in each element. For now, the array is required to be five elements long. After the array is filled with data, the array is written to the monitor.

```
import java.io.* ;


public class MainArray06 {

  public static void main ( String[] args ) throws IOException
  {

    int[] array = new int[5];
    int   data;

    BufferedReader inData = new BufferedReader ( new InputStreamReader( System.in ) );

    // input the data
    for ( ___________ ; ________________ ; _____________ )
    {
      System.out.println( "enter an integer: " );
      data           = Integer.parseInt( inData.readLine() );
      array[ index ] = data ;
    }

    // write out the data
    for ( ___________ ; ________________ ; _____________  )
    {
      System.out.println( "array[ " + index + " ] = " + array[ index ] );
    }

  }
}
```

QUESTION 4:

Can you fill in the blanks so that the program works as described?

A good answer is:

The complete program below.

Complete Program

```
import java.io.* ;


public class MainArray06 {

  public static void main ( String[] args ) throws IOException
  {

    int[] array = new int[5];
    int   data;

    BufferedReader inData = new BufferedReader ( new InputStreamReader( System.in ) );

    // input the data
    for ( int index=0; index < array.length; index++ )
    {
      System.out.println( "enter an integer: " );
      data           = Integer.parseInt( inData.readLine() );
      array[ index ] = data ;
    }

    // write out the data
    for ( int index=0; index < array.length; index++ )
    {
      System.out.println( "array[ " + index + " ] = " + array[ index ] );
    }

  }
}
```

(Create this program above using Eclipse)

QUESTION 5:

The variable `data` is not really needed in this program. Can you mentally change the program so that this variable is not used?

The variable `data` is not really needed in this program. Can you mentally change the program so that this variable is not used?

A good answer is:

The two lines:

```
data           = Integer.parseInt( inData.readLine() );
array[ index ] = data ;
```

can be replaced by the single line:

```
array[ index ] = Integer.parseInt( inData.readLine() );
```

And then the declaration `int data;` should be removed.

Array Length Determined when the Program Runs

Because an array object is constructed as the program runs, its size can determined at run time. The programmer does not need to say how many elements the array has. The user picks the size when the program runs. The array constructor creates the array after the user has specified how large the array is to be. Here is the previous example, with some modifications:

```
import java.io.* ;


public class MainArray07

  public static void main ( String[] args ) throws IOException
  {
    BufferedReader inData = new BufferedReader ( new InputStreamReader( System.in ) );
    int[] array;

    // determine the array size and construct the array
    System.out.println( "What length is the array?" );
    int size  = Integer.parseInt( inData.readLine() );

    array = new int[ size ];

    // input the data
    for ( int index=0; index < array.length; index++)
    {
      System.out.println( "enter an integer: " );
      array[ index ] = Integer.parseInt( inData.readLine() );
    }

    // write out the data
    for ( int index=0; index < array.length; index++ )
    {
      System.out.println( "array[ " + index + " ] = " + array[ index ] );
    }

  }
}
```

(Create this program above using Eclipse)

Finding the Maximum of an Array

Here is the program. Carefully examine how the `if` statement is used to change `max`.

```
public class MainArray08

  public static void main ( String[] args )
  {

    int[] array =  { -20, 19, 1, 5, -1, 27, 19, 5 } ;
    int   max;

    // initialize the current maximum
    max = array[0];

    // scan the array
    for ( int index=0; index < array.length; index++ )
    {
      if ( array[ index ] > max )    // examine the current element
        max = array[ index ];        // if it is the largest so far, change max

    }

    System.out.println("The maximum of this array is: " + max );
  }
}
```

(Create this program above using Eclipse)

Finding the Minimum of an Array

As with the maximum-finding program, you should run this program with various sets of data and confirm that it works with all of them.

```
public class MainArray09

  public static void main ( String[] args )
  {

    int[] array =  { -20, 19, 1, 5, -1, 27, 19, 5 } ;
    int   min;

    // initialize the current minimum
    min = array[ 0 ];

    // scan the array
    for ( int index=0; index < array.length; index++ )
    {
      if ( array[ index ] < min )

        min = array[ index ] ;

    }

    System.out.println("The minimum of this array is: " + min );
  }
}
```

(Create this program above using Eclipse)

Finding the Total Sum of an Array

The variable *total* is declared to be of *double* type, since the sum of the doubles in the array will be a double. It is initialized to zero. Sums should be initialized to zero as a matter of course.

```
public class MainArray10

  public static void main ( String[] args )
  {
    double[] array =  { -47.39, 24.96, -1.02, 3.45, 14.21, 32.6, 19.42 } ;

    // declare and initialize the total
    double   total =   0.0 ;

    // add each element of the array to the total
    for ( int index=0; index < array.length; index++ )
    {
      total =  total + array[ index ]  ;

    }

    System.out.println("The total is: " + total );
  }
}
```

(Create this program above using Eclipse)

The program visits each element of the array, in order, adding each to the total. When the loop exits, *total* will be correct. The statement

```
total =  total + array[ index ]  ;
```

would not usually be used. It is more common to use the "+=" operator:

```
total += array[ index ]  ;
```

QUESTION 6:

If you know the sum of the elements in an array of numbers, and know how many elements there are, how can you compute the *average* of the elements?

A good answer is:

Divide the sum by the number of elements.

Finding the Average of an Array

Of course, this is assuming that there are more than zero elements. Dividing by zero will always result in a run-time error. Here is the program with some additional statements for computing the average of the elements:

```
public class MainArray11

  public static void main ( String[] args )
  {
    double[] array =  { -47.39, 24.96, -1.02, 3.45, 14.21, 32.6, 19.42 } ;

    // declare and initialize the total
    double  total = 0.0 ;

    // add each element of the array to the total
    for ( int index=0; index < array.length; index++ )
    {
       total =  total + array[ index ] ;
    }
     if ( array.length != 0 )
     {
        System.out.println("The total is:   " + total );
        System.out.println("The average is: " + total / array.length  );
     }
      else
        System.out.println("The array contains no elements." );

  }

}
```

(Create this program above using Eclipse)

It might look a little strange to test if *array* contains any elements, since it is obvious that it does. However, in a more realistic program the array will come from some external source, and sometimes the array will have length zero.

Eclipse Programs: Strings Part 1

Java String Introduction

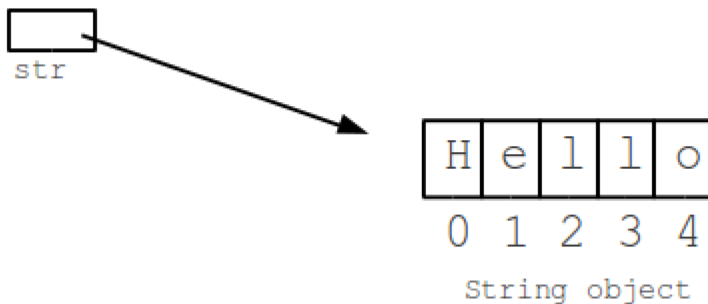
Strings are an incredibly common type of data in computers. This page introduces the basics of Java strings: chars, **+**, length(), and substring().

A Java string is a series of characters gathered together, like the word **"Hello"**, or the phrase "practice makes perfect". Create a string in the code by writing its chars out between double quotes.

- String stores text -- a word, an email, a book
- All computer languages have strings, look similar
- "In double quotes"
- Sequence of characters ("char")

```
String str = "Hello";
```

This picture shows the string object in memory, made up of the individual chars H e l l o. We'll see what the index numbers 0, 1, 2 .. mean later on.



String + Concatenation

Tthe **+** (plus) operator between strings puts them together to make a new, bigger string. The bigger string is just the chars of the first string put together with the chars of the second string.

```
String a = "kit" + "ten";  // a is "kitten"
```

Strings are not just made of the letters a-z. Chars can be punctuation and other miscellelaneous chars. For example in the string "hi ", the 3rd char is a space. This all works with strings stored in variables too, like this:

```
String fruit = "apple";
String stars = "***";
String a = fruit + stars;  //  a is "apple***"
```

CodingBat Practice> helloName

String-1 > helloName (MainString01.java)
Given a string name, e.g. "Bob", return a greeting of the form "Hello Bob!".

helloName("Bob") → "Hello Bob!"
helloName("Alice") → "Hello Alice!"
helloName("X") → "Hello X!"

```
public String helloName(String name) {

return "Hello " + name + "!";

}
```

Create and test this program above using Eclipse file: MainString01.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static String helloName(String name)
{
    return "Hello " + name + "!";

}
```

String Length

The "length" of a string is just the number of chars in it. So "hi" is length 2 and "Hello" is length 5. The length() method on a string returns its length, like this:

```
String a = "Hello";
int len = a.length();  // len is 5
```

String Index Numbers

- Index numbers -- 0, 1, 2, ...
- Leftmost char is at index 0
- Last char is at index length-1

H	e	l	l	o
0	1	2	3	4

The chars in a string are identified by "index" numbers. In "Hello" the leftmost char (H) is at index 0, the next char (e) is at index 1, and so on. The index of the last char is always one less than the length. In this case the length is 5 and 'o' is at index 4. Put another way, the chars in a string are at indexes 0, 1, 2, .. up through length-1. We'll use the index numbers to slice and dice strings with substring() in the next section.

String Substring

- str.substring(start)
- Chars beginning at index **start**
- Through the end of the string
- Later: more complex 2-arg substring()

H	e	l	l	o
0	1	2	3	4

The substring() method picks out a part of string using index numbers to identify the desired part. The simplest form, **substring(int start)** takes a start index number and returns a new string made of the chars starting at that index and running through the end of the string:

```
String str = "Hello";
String a = str.substring(1);  // a is "ello" (i.e. starting at index 1)
String b = str.substring(2);  // b is "llo"
String c = str.substring(3);  // c is "lo"
```

Above `str.substring(1)` returns "ello", picking out the part of "Hello" which begins at index 1 (the "H" is at index 0, the "e" is at index 1).

String-1 > extraEnd (MainString02.java)

Given a string, return a new string made of 3 copies of the last 2 chars of the original string. The string length will be at least 2.

extraEnd("Hello") → "lololo"
extraEnd("ab") → "ababab"
extraEnd("Hi") → "HiHiHi"

Create and test this program using Eclipse file: MainString02.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
// precondition: string length >= 2.
public static String extraEnd(String str)
{
   int n = str.length();

   String lastTwo = str.substring(n-2);

    return lastTwo + lastTwo + lastTwo;

 }
```

String Substring

- str.substring(start)
- str.substring(start, end)
- Chars beginning at **start**
- Up to but not including **end**

There is a more complex version of substring() that takes both start and end index numbers: **substring(int start, int end)** returns a string of the chars beginning at the start index number and running up to but not including the end index.

H	e	l	l	o
0	1	2	3	4

```
String str = "Hello";
String a = str.substring(2, 4);  // a is "ll" (not "llo")
String b = str.substring(0, 3);  // b is "Hel"
String c = str.substring(4, 5);  // c is "o" -- the last char
```

The c example above uses substring(4, 5) to grab the last char. The 5 is one more than the index of the last char. However, this does not go out of bounds because of the substring() "up to but not including" use of the end index. Incidentally, the length of the resulting substring can always be computed by subtracting (end - start) -- try it with the examples above.

String-1 > firstTwo (MainString03.java)

Given a string, return the string made of its first two chars, so the String "Hello" yields "He". If the string is shorter than length 2, return whatever there is, so "X" yields "X", and the empty string "" yields the empty string "". Note that str.length() returns the length of a string.

firstTwo("Hello") → "He"
firstTwo("abcdefg") → "ab"
firstTwo("ab") → "ab"

Create and test this program using Eclipse file: MainString03.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static String firstTwo(String str)
{
   int n = str.length();
   if (n >= 2)
      return str.substring(0,2);
   else
      return str;
}
```

String Index Errors: "String Index Out Of Bounds" or "String Index Out Of Range"

- Common mistake: index greater than length
- Index Out of Bounds Error
- If-statement check length first

It is very common to get little errors with the index numbers fed into substring(). The valid index numbers for substring are basically 0, 1, 2, ... str.length(), so code needs to be careful not to pass in numbers outside that range. Note that the last number, str.length(), is one beyond the end of the string. You need this number to fit the "up to but not including" way that substring() works. For the above "Hello" examples, the valid index numbers are always in the range 0..5 since the length of "Hello" is 5.

Often avoding substring() out of bounds errors involves adding logic to check the length of the string. For example, suppose we want to take the first 4 chars of a string, like this...

```
// Suppose we want the first 4 chars of str
String a = str.substring(0, 4);  // WRONG error sometimes
```

The problem with the above is .. what if the str length is less than 4? In that case, substring(0, 4) refers to non-existent chars and will fail wen run. One possible solution will add if-logic like this:

```
if (str.length() >= 4)
    a = str.substring(0, 4);
else
  // whatever you want to do when length is < 4 (see last example above)
```

String-1 > withoutEnd (MainString04.java)

Given a string, return a version without the first and last char, so "Hello" yields "ell". The string length will be at least 2.

withoutEnd("Hello") → "ell"
withoutEnd("java") → "av"
withoutEnd("coding") → "odin"

Create and test this program using Eclipse file: MainString04.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
//precondition: string length >= 2.
public static String withoutEnd(String str)
{
    int n = str.length();
    if (n == 2)
       return "";
    else
      return str.substring(1,n-1);

}
```

String-1 > right2 (MainString05.java)

Given a string, return a "rotated right 2" version where the last 2 chars are moved to the start. The string length will be at least 2.

right2("Hello") → "loHel"
right2("java") → "vaja"
right2("Hi") → "Hi"

Create and test this program using Eclipse file: MainString05.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
// precondition: n > 1.
public static String right2(String str)
{
    int n = str.length();
    return str.substring(n-2,n) + str.substring(0,n-2);
}
```

String-1 > theEnd (MainString06.java)

Given a string, return a string length 1 from its front, unless front is false, in which case return a string length 1 from its back. The string will be non-empty.

theEnd("Hello", true) → "H"
theEnd("Hello", false) → "o"
theEnd("oh", true) → "o"

Create and test this program using Eclipse file: MainString06.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
// precondition: str.length() >= 1;

public static String theEnd(String str, boolean front)
{
    int n = str.length();
    if (!front)
        return str.substring(n-1,n);
    else
        return str.substring(0,1);
}
```

String-1 > endsLy (MainString07.java)

Given a string, return true if it ends in "ly".
endsLy("oddly") → true
endsLy("y") → false
endsLy("oddy") → false

Create and test this program using Eclipse file: MainString07.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static boolean endsLy(String str)
{
    int n = str.length();
    if (n < 2 || !str.substring(n-2,n).equals("ly"))
        return false;
    else
        return true;
}
```

Java String Equals and Loops

- Compare two Strings:
- `a.equals(b)`
- **Do not** use ==
- Sadly == compiles, but does not work reliably .. a real trap
- In retrospect, an error in the design of Java

String Equals

Use the equals() method to check if 2 strings are the same. The equals() method is case-sensitive, meaning that the string "HELLO" is considered to be different from the string "hello". The == operator does not work reliably with strings. Use == to compare primitive values such as int and char. Unfortunately, it's easy to accidentally use == to compare strings, but it will not work reliably. Remember: use equals() to compare strings. There is a variant of equals() called equalsIgnoreCase() that compares two strings, ignoring uppercase/lowercase differences.

```
String a = "hello";
String b = "there";

if (a.equals("hello")) {
  // Correct -- use .equals() to compare Strings
}

if (a == "hello") {
  // NO NO NO -- do not use == with Strings
}

// a.equals(b) -> false
// b.equals("there") -> true
// b.equals("There") -> false
// b.equalsIgnoreCase("THERE") -> true
```

String-1 > hasBad (MainString08.java)

Given a string, return true if "bad" appears starting at index 0 or 1 in the string, such as with "badxxx" or "xbadxx" but not "xxbadxx". The string may be any length, including 0. Note: use .equals() to compare 2 strings.

hasBad("badxx")→true

hasBad("xbadxx")→true

hasBad("xxbadxx")→false

Create and test this program using Eclipse file: MainString08.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static boolean hasBad(String str) {

   int n = str.length();

   if ( (n >= 3 && str.substring(0,3).equals("bad")) ||
        (n >= 4 && str.substring(1,4).equals("bad"))   )
      return true;
  else
      return false;
}
```

String-1 > lastTwo (MainString09.java)
Given a string of any length, return a new string where the last 2 chars, if present, are swapped, so "coding" yields "codign".

lastTwo("coding") → "codign"
lastTwo("cat") → "cta"
lastTwo("ab") → "ba"

Create and test this program using Eclipse file: MainString09.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static String lastTwo(String str) {
  int n = str.length();

   if (n <= 1)
     return str;
   else
     // add a correct statement here to complete this method

}
```

String For Loop

- Super-common string for-loop
- Loop to hit each index number once:
- 0, 1, 2, ... length-1
- `for (int i = 0; i < str.length(); i++) {...}`

String-2 > countHi (MainString10.java)

Return the number of times that the string "hi" appears anywhere in the given string.

countHi("abc hi ho") → 1
countHi("ABChi hi") → 2
countHi("hihi") → 2

Create and test this program using Eclipse file: MainString10.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static int countHi(String str) {
 int n = str.length();
 int count = 0;
 if (n <= 1)
   return 0;
 else
 {
   for (int i = 0; i < n-1 ; i++)
   {
     if (// add a correct statement here to complete this method)
       count++;
   }
   return count;
 }
```

String-2 > catDog (MainString11.java)

Return true if the strings "cat" and "dog" appear the same number of times in the given string.

catDog("catdog") → true

catDog("catcat") → false

catDog("1cat1cadodog") → true

Create and test this program using Eclipse file: MainString11.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static boolean catDog(String str) {

 int n = str.length();
 int catCount = 0;
 int dogCount = 0;
 if (n > 2)
 {
   for (int i = 0; i < n-2; i++)
   {
     if (// add a correct statement here to complete this method) catCount++;
     if (// add a correct statement here to complete this method) dogCount++;
   }
 }
 return (catCount == dogCount);

}
```

String-2 > countCode (MainString12.java)

Return the number of times that the string "code" appears anywhere in the given string, except we'll accept any letter for the 'd', so "cope" and "cooe" count.

countCode("aaacodebbb") → 1

countCode("codexxcode") → 2

countCode("cozexxcope") → 2

Create and test this program using Eclipse file: MainString12.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static int countCode(String str) {

 int n = str.length();
 int count = 0;

 if (n > 3)
 {
   for (int i = 0; i < n-3; i++)
   {
     if (str.substring(i,i+2).equals("co") && // add a correct statement here to complete this method)
       count++;
   }
 }
 return count;
}
```

Java String indexOf

- str.indexOf(String target) -- searches left-to-right for target
- Returns index where found, or -1 if not found
- Use to find the first (leftmost) instance of target
- str.lastIndexOf(String target) -- searches right-to-left instead

The indexOf(String target) method searches left-to-right inside the given string for a "target" string. The indexOf() method returns the index number where the target string is first found or -1 if the target is not found. Like equals(), the indexOf() method is case-sensitive, so uppercase and lowercase chars are considered to be different.

So basically, if you were going to write a for-loop to iterate over a string and look for a string, indexOf() can just do it for you. Note that indexOf() works best if you want to find the **first** instance of the target. If you want to find all the instances, see the next section.

```
String str = "Here there everywhere";

int a = str.indexOf("there");  // a is 5
int b = str.indexOf("er");     // b is 1
int c = str.indexOf("eR");     // c is -1, "eR" is not found
```

Create allIndexes Method (Not a Codingbat problem)

allIndexes (MainString13.java)

This method finds all the indexes at which string1 occurs in string2.

allIndexes("ac","aaacodacbbb") → [2, 6]

allIndexes("e","decodexxcodde")→[1, 5, 12]

allIndexes("cop","cozexcozpex") → []

Create and test this program using Eclipse file: MainString13.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below. Note that we are using ArrayList<Integer> which we can add to and remove from after creation, i. e. the size does not have to be specified in advance and can be empty.

```
public static ArrayList<Integer> allIndexes(String str1, String str2){

  int n1 = str1.length();
  int n2 = str2.length();

  ArrayList<Integer> indexes = new ArrayList<Integer>();

  if (n1 > n2) return indexes;

  for (int i = 0; i < n2-n1+1; i++)
  {
   if (str2.substring(i,i+n1).equals(str1)) indexes.add(i);
  }

  return indexes;

}
```


CodingBat.com, Copyright 2012 Nick Parlante.

Eclipse Programs: Strings Part 2

String-2 > endOther (MainString14.java)
Given two strings, return true if either of the strings appears at the very end of the other string, ignoring upper/lower case differences (in other words, the computation should not be "case sensitive"). Note: str.toLowerCase() returns the lowercase version of a string.

endOther("Hiabc", "abc") → true

endOther("AbC", "HiaBc") → true

endOther("abc", "abXabc") → true

Create and test this program above using Eclipse file: MainString14.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static boolean endOther(String a, String b) {

 int n1 = a.length();
 int n2 = b.length();

 if (n1 == n2)
 {
     return a.toLowerCase().equals(b.toLowerCase());
 }
 else if (n1 < n2)
 {
     return a.toLowerCase().equals(b.toLowerCase().substring(n2-n1,n2));
 }
 else
 {
     return b.toLowerCase().equals(a.toLowerCase().substring(n1-n2,n1));
 }

}
```

String-2 > bobThere (MainString15.java)
Return true if the given string contains a "bob" string, but where the middle 'o' char can be any char.

bobThere("abcbob") → true

bobThere("b9b") → true

bobThere("bac") → false

Create and test this program above using Eclipse file: MainString15.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static boolean bobThere(String str) {

 int n = str.length();

 if (n <= 2)
 {

   return false;
 }
 else
 {
   for (int i = 0; i < n-2; i++)
   {

      if (str.substring(i,i+1).equals("b") &&
          str.substring(i+2,i+3).equals("b"))
        return true;

   }
 }

 return false;

}
```

String-2 > xyBalance (MainString16.java)
We'll say that a String is xy-balanced if for all the 'x' chars in the string, there exists a 'y' char somewhere later in the string. So "xxy" is balanced, but "xyx" is not. One 'y' can balance multiple 'x's. Return true if the given string is xy-balanced.

xyBalance("aaxbby") → true

xyBalance("aaxbb") → false

xyBalance("yaaxbb") → false

Create and test this program above using Eclipse file: MainString16.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static boolean xyBalance(String str) {

   int xIndex = maxIndex('x',str);   // call helper method
   int yIndex = maxIndex('y',str);

   if (xIndex == -1)     // x not in str
   {
       return true;
   }
   else
   {
       if (yIndex == -1)  // y not in str and x in str
         return false;
       else
         return (xIndex < yIndex);
}

// Create helper method which returns the max index at which the
// character ch occurs in the string str. Unless ch doesn't occur
// in str, in which case it returns -1.

public static int maxIndex(char ch, String str)
{
   int n = str.length();
   int max = -1;
   for (int i = 0; i < n; i++)
   {
       if (str.charAt(i) == ch) max = i;
   }
   return max;
}
```

String-2 > mixString (MainString17.java)

Given two strings, a and b, create a bigger string made of the first char of a, the first char of b, the second char of a, the second char of b, and so on. Any leftover chars go at the end of the result.

mixString("abc", "xyz") → "axbycz"

mixString("Hi", "There") → "HTihere"

mixString("xxxx", "There") → "xTxhxexre"

Create and test this program above using Eclipse file: MainString17.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static String mixString(String a, String b) {

  int n1 = a.length();
  int n2 = b.length();
  int n = Math.min(n1,n2);
  String mix = "";

  if (n1 == 0 || n2 == 0)
  {
      return a + b;
  }
  else
  {
      for (int i = 0; i < n; i++)
      {
          mix = mix + a.substring(i,i+1) + b.substring(i,i+1);
      }

      if (n1 == n2)
      {
         return mix;
      }
      else if (n1 < n2)
      {
         return mix + b.substring(n1,n2);
      }
      else
      {
         return mix + a.substring(n2,n1);
      }
   }
}
```

String-2 > repeatEnd (MainString18.java)
Given a string and an int n, return a string made of n repetitions of the last n characters of the string. You may assume that n is between 0 and the length of the string, inclusive.

repeatEnd("Hello", 3) → "llollollo"

repeatEnd("Hello", 2) → "lolo"

repeatEnd("Hello", 1) → "o"

Create and test this program above using Eclipse file: MainString18.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
// precondition: 0 <= n <= str.length()

public static String repeatEnd(String str, int n) {

  int m = str.length();

  if (m == 0 || n == 0)
     return "";
  else
  {
     String reps = "";
     for (int i = 0; i < n; i++)
     {
         reps += str.substring(m-n,m);
     }
     return reps;
   }
}
```

String-2 > repeatFront (MainString19.java)
Given a string and an int n, return a string made of the first n characters of the string, followed by the first n-1 characters of the string, and so on. You may assume that n is between 0 and the length of the string, inclusive (i.e. n >= 0 and n <= str.length()).

repeatFront("Chocolate", 4) → "ChocChoChC"

repeatFront("Chocolate", 3) → "ChoChC"

repeatFront("Ice Cream", 2) → "IcI"

Create and test this program above using Eclipse file: MainString19.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
// precondition: 0 <= n <= str.length()

public static String repeatFront(String str, int n) {

  int m = str.length();
  if (m == 0 || n == 0)
     return "";
  else
  {
     String reps = "";
     for (int i = 0; i < n; i++)
     {

        // insert your code here

     }
     return reps;
  }
}
```

String-2 > prefixAgain (MainString20.java)
Given a string, consider the prefix string made of the first N chars of the string. Does that prefix string appear somewhere else in the string? Assume that the string is not empty and that N is in the range 1..str.length().

prefixAgain("abXYabc", 1) → true

prefixAgain("abXYabc", 2) → true

prefixAgain("abXYabc", 3) → false

Create and test this program above using Eclipse file: MainString20.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static boolean prefixAgain(String str, int n) {

  int m = str.length();
  if (m == 0)
     return false;
  else
  {
     String pre = str.substring(0,n);
     int prelen = pre.length();

     for (int i = 1; i < m-prelen+1; i++)
     {

           // insert your code here

      }
      return false;
  }

}
```

String-2 > repeatSeparator (MainString21.java)
Given two String variables: ***word***, ***sep*** and an int variable: ***count***, return a big string made of ***count*** number of occurrences of ***word***, separated by the separator string ***sep***.

repeatSeparator("Word", "X", 3) → "WordXWordXWord"

repeatSeparator("This", "And", 2) → "ThisAndThis"

repeatSeparator("This", "And", 1) → "This"

Create and test this program above using Eclipse file: MainString21.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static String repeatSeparator(String word, String sep, int count) {

   String reps = "";
   for (int i = 0; i < count; i++)
   {
      if (i != count-1)
          reps += word + sep;
      else

          // insert your code here

   }

   // insert your code here


}
```

String-3 > countYZ (MainString22.java)
Given a string, count the number of words ending in 'y' or 'z' -- so the 'y' in "heavy" and the 'z' in "fez" count, but not the 'y' in "yellow" (not case sensitive). We'll say that a y or z is at the end of a word if there is not an alphabetic letter immediately following it. (Note: Character.isLetter(char) tests if a char is an alphabetic letter.)

countYZ("fez day") → 2

countYZ("day fez") → 2

countYZ("day fyyyz") → 2

Create and test this program above using Eclipse file: MainString22.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static int countYZ(String str)
{
    int count=0;
    int n=str.length();

    for(int i = 0; i < n;i++)
    {
        String str1 = "" + str.charAt(i);
        if((str1.equalsIgnoreCase("y") || str1.equalsIgnoreCase("z")))
        {

            if(i == n-1 || !Character.isLetter(str.charAt(i+1)))
              count++;
        }
    }
    return count;
}
```

String-3 > sumDigits (MainString23.java)
Given a string, return the sum of the digits 0-9 that appear in the string, ignoring all other characters. Return 0 if there are no digits in the string. (Note: Character.isDigit(char) tests if a char is one of the chars '0', '1', .. '9'. Integer.parseInt(string) converts a string to an int.)

sumDigits("aa1bc2d3") → 6

sumDigits("aa11b33") → 8

sumDigits("Chocolate") → 0

Create and test this program above using Eclipse file: MainString23.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static int sumDigits(String str)
{
      int n = str.length();
      int sum = 0;

      if (n > 0)
      {
           for (int i = 0; i < n; i++)
           {
                 char ch = str.charAt(i);
                 if (Character.isDigit(ch))
                    // insert your code here
            }
       }
       // insert your code here
}
```

String-3 > sameEnds(MainString24.java)
Given a string, return the longest substring that appears at both the beginning and end of the string without overlapping. For example, sameEnds("abXab") is "ab".

sameEnds("abXYab") → "ab"

sameEnds("xx") → "x"

sameEnds("xxx") → "x"

Create and test this program above using Eclipse file: MainString24.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static String sameEnds(String str)
{

      int n = str.length();

      if (n <= 1)
          return "";
      else
      {
            String pal = "";
            for (int i = 1; i < (n/2)+1; i++)
            {
                  String front = str.substring(0,i);
                  String back = str.substring(n-i,n);

                  if (front.equals(back))
                      // insert your code here
            }
            // insert your code here
      }

}
```

String-3 > maxBlock(MainString25.java)
Given a string, return the length of the largest "block" in the string. A block is a run of adjacent chars that are the same.

maxBlock("hoopla") → 2

maxBlock("abbCCCddBBBxx") → 3

maxBlock("") → 0

Create and test this program above using Eclipse file: MainString25.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static int maxBlock(String str)
{
      int n = str.length();
      if (n <= 1)
            return n;
      else
      {
             int max = 1;
             int temp = 1;

             for (int i = 1; i < n; i++)
             {
                      if ( str.substring(i-1,i).equals(str.substring(i,i+1)) )
                      {
                              temp++;
                              if (i == n-1)
                                  max = Math.max(max,temp);
                       }
                       else
                       {
                               // insert your code here
                               // insert your code here
                        }
              }
              // insert your code here
       }

}
```

String-3 > sumNumbers(MainString26.java)
Given a string, return the sum of the numbers appearing in the string, ignoring all other characters. A number is a series of 1 or more digit chars in a row. (Note: Character.isDigit(char) tests if a char is one of the chars '0', '1', .. '9'. Integer.parseInt(string) converts a string to an int.)

sumNumbers("abc123xyz") → 123

sumNumbers("aa11b33") → 44

sumNumbers("7 11") → 18

Create and test this program above using Eclipse file: MainString26.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

```
public static int sumNumbers(String str)
{
       int n = str.length();
       int sum = 0;
       String temp = "";

        for (int i = 0; i < n; i++)
        {
              if ( Character.isDigit(str.charAt(i)) )
              {
                     temp += "" + str.charAt(i);
                     if (i == n-1)
                         sum += Integer.parseInt(temp);
               }
               else
               {
                     if (!temp.equals(""))
                     {
                            // insert your code here
                           // insert your code here
                      }
                }
         }
         // insert your code here

}
```

String-3 > mirrorEnds(MainString27.java)
Given a string, look for a mirror image (backwards) string at both the beginning and end of the given string. In other words, zero or more characters at the very beginning of the given string, and at the very end of the string in reverse order (possibly overlapping). For example, the string "abXYZba" has the mirror end "ab".

mirrorEnds("abXYZba") → "ab"

mirrorEnds("abca") → "a"

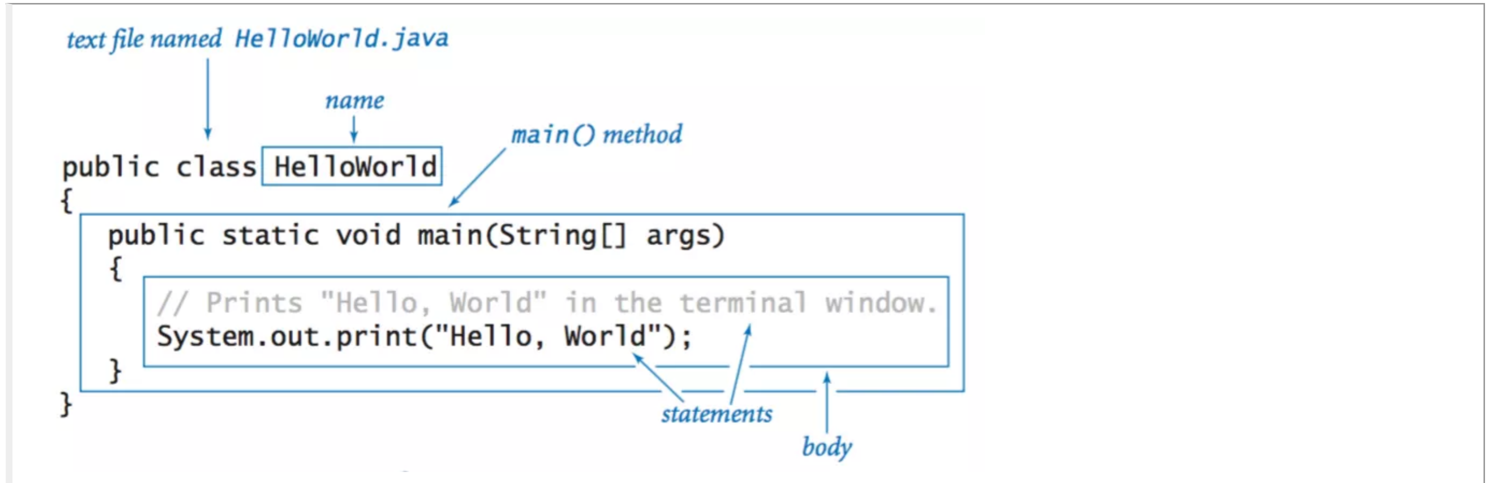
mirrorEnds("aba") → "aba"

Create and test this program above using Eclipse file: MainString27.java. Test the same three examples above. Also, make sure to include the static parameter as indicated below.

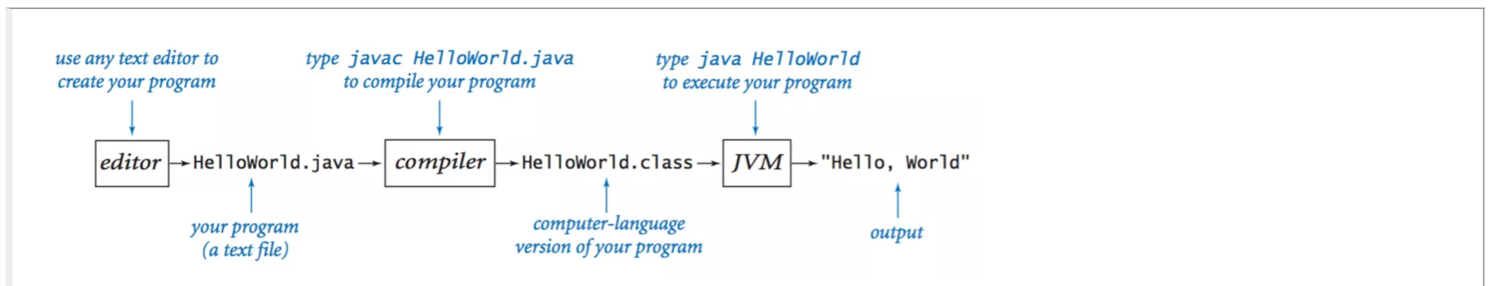
```
public static String mirrorEnds(String str)
{

      int n = str.length();

       if ( str.equals(reverseStr(str)) )  // str is a palindrome
           return str;
       else
       {
             String pal = "";
             for (int i = 1; i < (n/2)+1; i++)
             {
                   String front = str.substring(0,i);
                   String back = str.substring(n-i,n);

                    if ( front.equals(reverseStr(back)) )
                        // insert your code here
             }
             // insert your code here
        }

}

// helper method
 public static String reverseStr(String str)
 {
        int n = str.length();
        String reverse = "";

       for (int i = 0; i < n; i++)
       {
             reverse += str.substring((n-1)-i,n-i);
       }
        return reverse;
 }
```

Java Syntax, Primitive Types, Operators, Arrays, Strings, and Programming Basics

Hello, World.



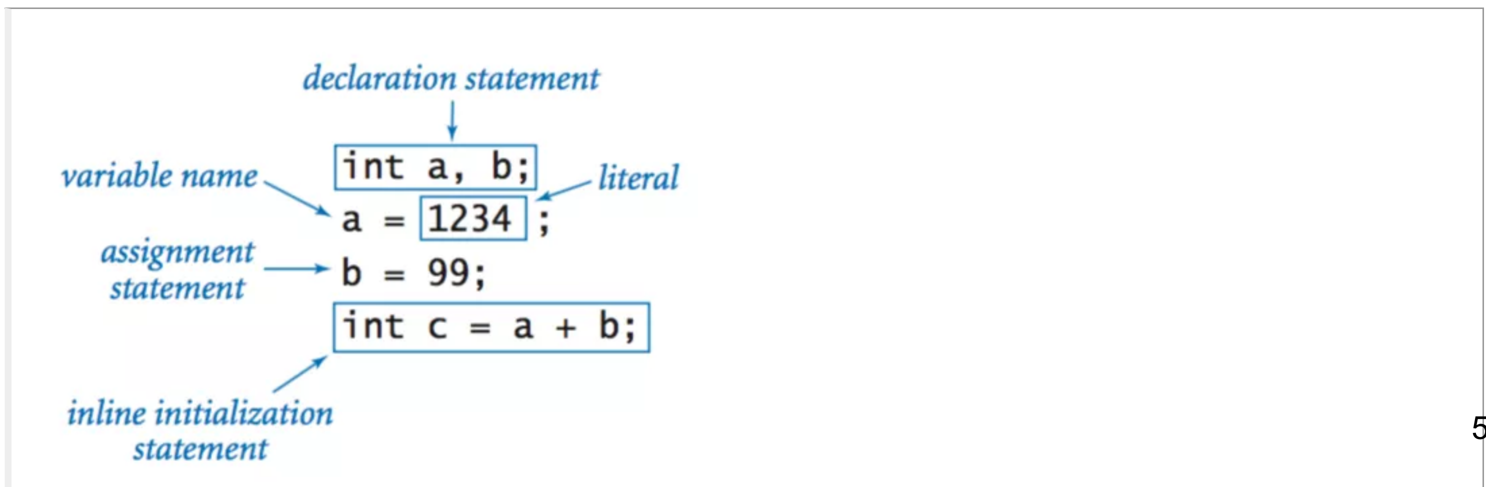
Editing, compiling, and executing.



Built-in data types.

type	set of values	common operators	sample literal values
int	integers	+ - * / %	99 12 2147483647
double	floating-point numbers	+ - * /	3.14 2.5 6.022e23
boolean	boolean values	&& || !	true false
char	characters		'A' '1' '%' '\n'
String	sequences of characters	+	"AB" "Hello" "2.5"

Declaration and assignment statements.



Integers.

values	integers between -2^{31} and $+2^{31}-1$					
typical literals	1234 99 0 1000000					
operations	*sign*	*add*	*subtract*	*multiply*	*divide*	*remainder*
operators	+ -	+	-	*	/	%

expression	*value*	*comment*
99	99	*integer literal*
+99	99	*positive sign*
-99	-99	*negative sign*
5 + 3	8	*addition*
5 - 3	2	*subtraction*
5 * 3	15	*multiplication*
5 / 3	1	*no fractional part*
5 % 3	2	*remainder*
1 / 0		*run-time error*
3 * 5 - 2	13	** has precedence*
3 + 5 / 2	5	*/ has precedence*
3 - 5 - 2	-4	*left associative*
(3 - 5) - 2	-4	*better style*
3 - (5 - 2)	0	*unambiguous*

Floating-point numbers.

values	real numbers (specified by IEEE 754 standard)			
typical literals	3.14159	6.022e23	2.0	1.4142135623730951
operations	*add*	*subtract*	*multiply*	*divide*
operators	+	-	*	/

expression	*value*
3.141 + 2.0	5.141
3.141 - 2.0	1.111
3.141 / 2.0	1.5705
5.0 / 3.0	1.6666666666666667
10.0 % 3.141	0.577
1.0 / 0.0	Infinity
Math.sqrt(2.0)	1.4142135623730951
Math.sqrt(-1.0)	NaN

10 % 3 = 1

Booleans.

values	*true or false*		
literals	true	false	
operations	and	or	not
operators	&&	||	!

a	*!a*
true	false
false	true

a	*b*	*a && b*	*a || b*
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Comparison operators.

op	meaning	true	false
==	equal	2 == 2	2 == 3
!=	not equal	3 != 2	2 != 2
<	less than	2 < 13	2 < 2
<=	less than or equal	2 <= 2	3 <= 2
>	greater than	13 > 2	2 > 13
>=	greater than or equal	3 >= 2	2 >= 3

non-negative discriminant?	`(b*b - 4.0*a*c) >= 0.0`
beginning of a century?	`(year % 100) == 0`
legal month?	`(month >= 1) && (month <= 12)`

Printing.

`void System.out.print(String s)`	*print s*
`void System.out.println(String s)`	*print s, followed by a newline*
`void System.out.println()`	*print a newline*

Parsing command-line arguments.

`int Integer.parseInt(String s)`	*convert s to an* `int` *value*
`double Double.parseDouble(String s)`	*convert s to a* `double` *value*
`long Long.parseLong(String s)`	*convert s to a* `long` *value*

Math library.

public class Math	
double abs(double a)	*absolute value of* a
double max(double a, double b)	*maximum of* a *and* b
double min(double a, double b)	*minimum of* a *and* b
double sin(double theta)	*sine of* theta
double cos(double theta)	*cosine of* theta
double tan(double theta)	*tangent of* theta
double toRadians(double degrees)	*convert angle from degrees to radians*
double toDegrees(double radians)	*convert angle from radians to degrees*
double exp(double a)	*exponential* (e^a)
double log(double a)	*natural log* ($\log_e a$, *or* $\ln a$)
double pow(double a, double b)	*raise* a *to the* b*th power* (a^b)
long round(double a)	*round* a *to the nearest integer*
double random()	*random number in* [0, 1)
double sqrt(double a)	*square root of* a
double E	*value of* e *(constant)*
double PI	*value of* π *(constant)*

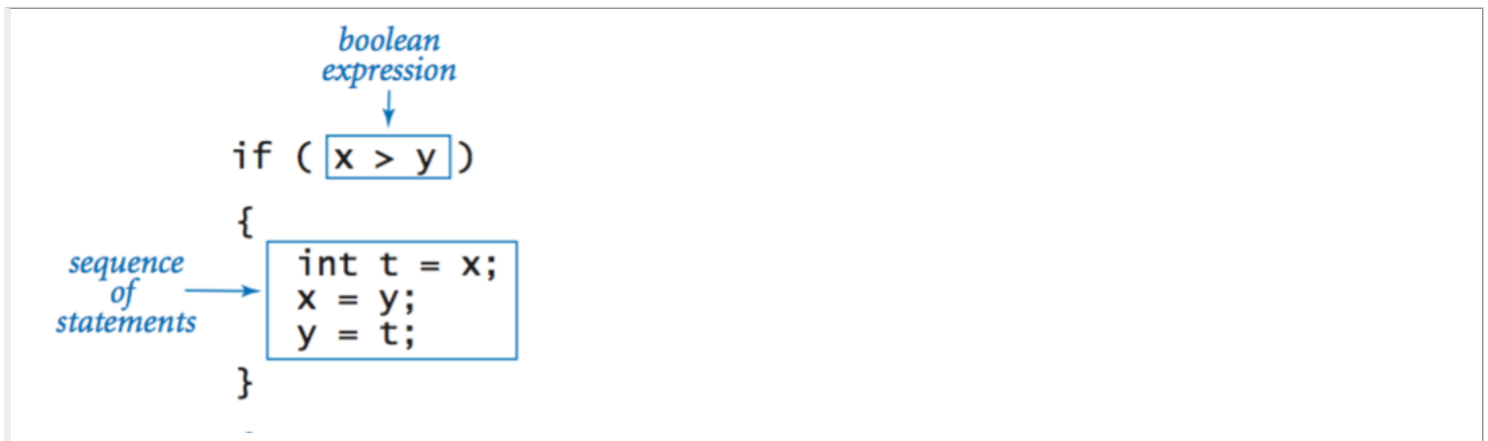
Java library calls.

method call	*library*	*return type*	*value*
Integer.parseInt("123")	Integer	int	123
Double.parseDouble("1.5")	Double	double	1.5
Math.sqrt(5.0*5.0 - 4.0*4.0)	Math	double	3.0
Math.log(Math.E)	Math	double	1.0
Math.random()	Math	double	*random in* [0, 1)
Math.round(3.14159)	Math	long	3
Math.max(1.0, 9.0)	Math	double	9.0

Type conversion.

expression	*expression type*	*expression value*
`(1 + 2 + 3 + 4) / 4.0`	`double`	`2.5`
`Math.sqrt(4)`	`double`	`2.0`
`"1234" + 99`	`String`	`"123499"`
`11 * 0.25`	`double`	`2.75`
`(int) 11 * 0.25`	`double`	`2.75`
`11 * (int) 0.25`	`int`	`0`
`(int) (11 * 0.25)`	`int`	`2`
`(int) 2.71828`	`int`	`2`
`Math.round(2.71828)`	`long`	`3`
`(int) Math.round(2.71828)`	`int`	`3`
`Integer.parseInt("1234")`	`int`	`1234`

Anatomy of an if statement.



If and if-else statements.

absolute value	`if (x < 0) x = -x;`

put the smaller value in x and the larger value in y	`if (x > y)`
	`{`
	`   int t = x;`
	`   x = y;`
	`   y = t;`
	`}`

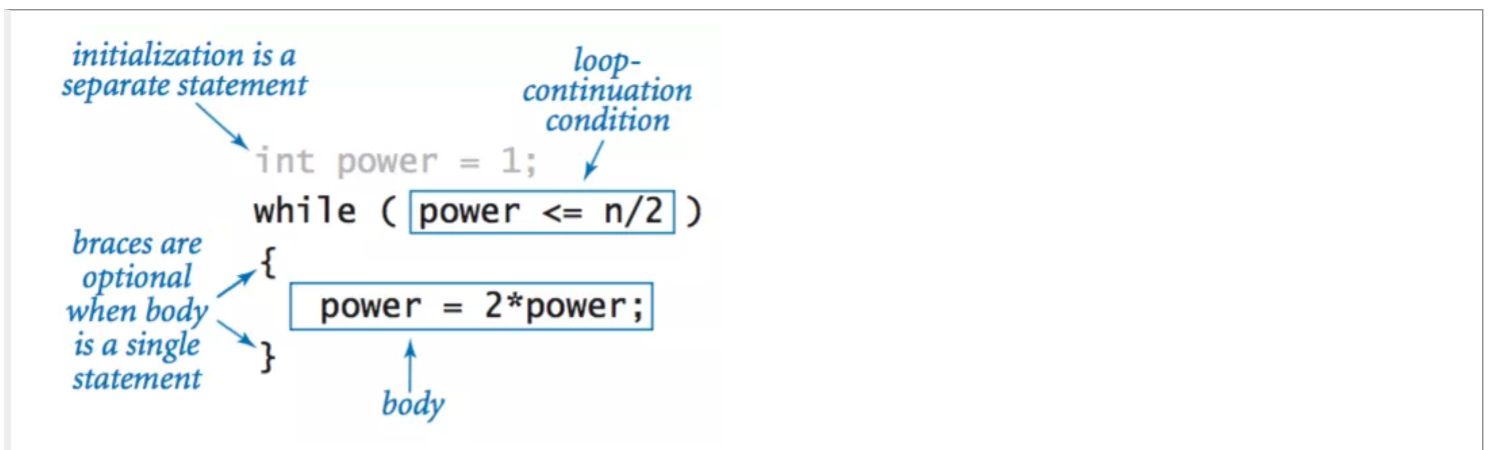
maximum of x and y	`if (x > y) max = x;` `else       max = y;`
error check for division operation	`if (den == 0) System.out.println("Division by zero");` `else           System.out.println("Quotient = " + num/den);`

error check for quadratic formula	`double discriminant = b*b - 4.0*c;`
	`if (discriminant < 0.0)`
	`{`
	`   System.out.println("No real roots");`
	`}`
	`else`
	`{`
	`   System.out.println((-b + Math.sqrt(discriminant))/2.0);`
	`   System.out.println((-b - Math.sqrt(discriminant))/2.0);`
	`}`

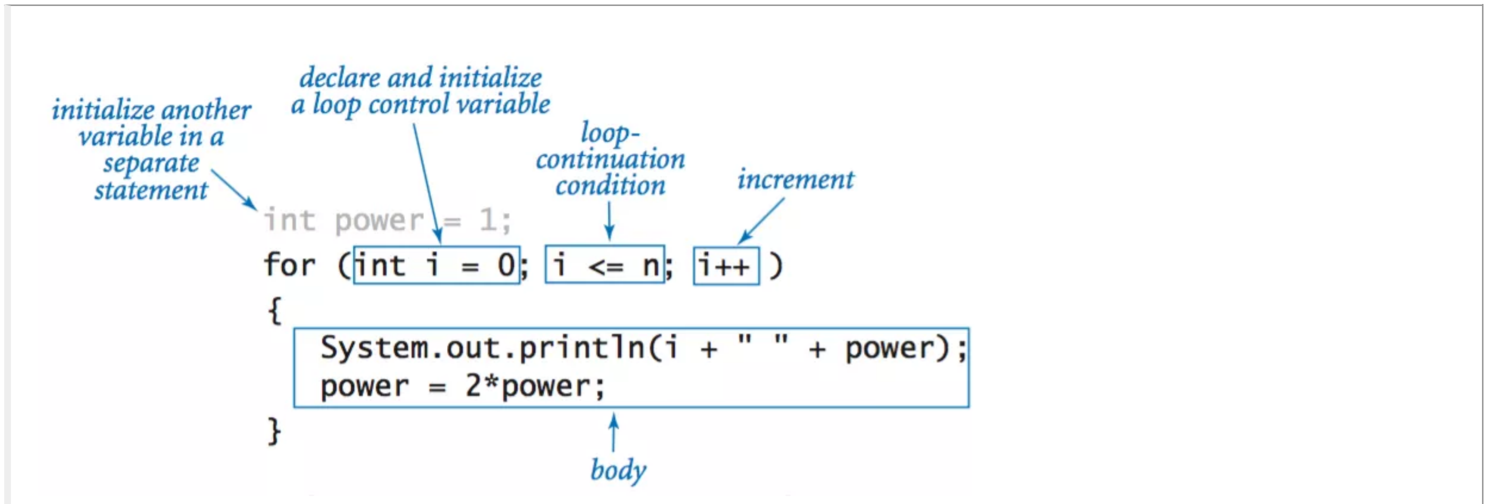
Nested if-else statement.

```
if      (income <      0) rate = 0.00;
else if (income <   8925) rate = 0.10;
else if (income <  36250) rate = 0.15;
else if (income <  87850) rate = 0.23;
else if (income < 183250) rate = 0.28;
else if (income < 398350) rate = 0.33;
else if (income < 400000) rate = 0.35;
else                      rate = 0.396;
```

Anatomy of a while loop.



Anatomy of a for loop.



Loops.

compute the largest power of 2 less than or equal to n	`int power = 1;`
	`while (power <= n/2)`
	`   power = 2*power;`
	`System.out.println(power);`

compute a finite sum (1 + 2 + … + n)	`int sum = 0;`
	`for (int i = 1; i <= n; i++)`
	`   sum += i;`
	`System.out.println(sum);`

compute a finite product (n! = 1 × 2 × … × n)	`int product = 1;`
	`for (int i = 1; i <= n; i++)`
	`   product *= i;`
	`System.out.println(product);`

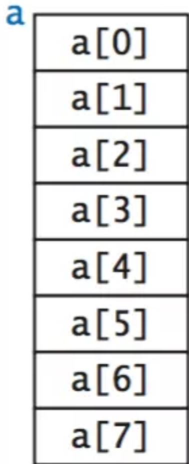
print a table of function values	`for (int i = 0; i <= n; i++)` `   System.out.println(i + " " + 2*Math.PI*i/n);`

compute the ruler function (see Program 1.2.1)	`String ruler = "1";`
	`for (int i = 2; i <= n; i++)`
	`   ruler = ruler + " " + i + " " + ruler;`
	`System.out.println(ruler);`

Break statement.

```
int factor;
for (factor = 2; factor <= n/factor; factor++)
   if (n % factor == 0) break;

if (factor > n/factor)
   System.out.println(n + " is prime");
```

Arrays.



Inline array initialization.

```
String[] SUITS = { "Clubs", "Diamonds", "Hearts", "Spades" };

String[] RANKS = {
   "2", "3", "4", "5", "6", "7", "8", "9", "10",
   "Jack", "Queen", "King", "Ace"
};
```

Typical array-processing code.

create an array with random values

```
double[] a = new double[n];
for (int i = 0; i < n; i++)
   a[i] = Math.random();
```

print the array values, one per line

```
for (int i = 0; i < n; i++)
   System.out.println(a[i]);
```

find the maximum of the array values

```
double max = Double.NEGATIVE_INFINITY;
for (int i = 0; i < n; i++)
   if (a[i] > max) max = a[i];
```

compute the average of the array values

```
double sum = 0.0;
for (int i = 0; i < n; i++)
   sum += a[i];
double average = sum / n;
```

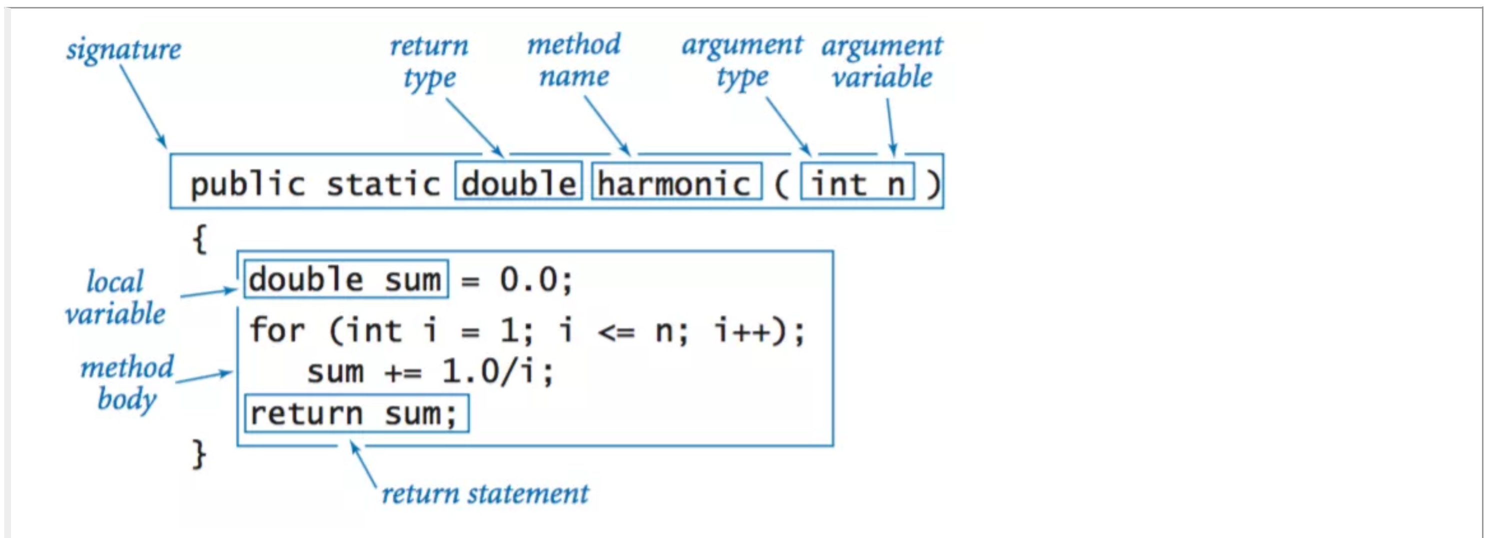
reverse the values within an array

```
for (int i = 0; i < n/2; i++)
{
   double temp = a[i];
   a[i] = a[n-1-i];
   a[n-i-1] = temp;
}
```

copy sequence of values to another array

```
double[] b = new double[n];
for (int i = 0; i < n; i++)
   b[i] = a[i];
```

Methods



absolute value of an `int` *value*	`public static int abs(int x)`
	`{`
	`   if (x < 0) return -x;`
	`   else        return  x;`
	`}`

absolute value of a `double` *value*	`public static double abs(double x)`
	`{`
	`   if (x < 0.0) return -x;`
	`   else          return  x;`
	`}`

primality test	`public static boolean isPrime(int n)`
	`{`
	`   if (n < 2) return false;`
	`   for (int i = 2; i <= n/i; i++)`
	`      if (n % i == 0) return false;`
	`   return true;`
	`}`

hypotenuse of a right triangle	`public static double hypotenuse(double a, double b)` `{  return Math.sqrt(a*a + b*b);  }`

harmonic number	`public static double harmonic(int n)`
	`{`
	`   double sum = 0.0;`
	`   for (int i = 1; i <= n; i++)`
	`      sum += 1.0 / i;`
	`   return sum;`
	`}`

uniform random integer in $[0, n)$	`public static int uniform(int n)` `{  return (int) (Math.random() * n);  }`

draw a triangle	`public static void drawTriangle(double x0, double y0,`
	`                                double x1, double y1,`
	`                                double x2, double y2 )`
	`{`
	`   StdDraw.line(x0, y0, x1, y1);`
	`   StdDraw.line(x1, y1, x2, y2);`
	`   StdDraw.line(x2, y2, x0, y0);`
	`}`

Using an object.

declare a variable (object name)

invoke a constructor to create an object

```
String s;
s = new String("Hello, World");
char c = s.charAt(4);
```

object name

invoke an instance method that operates on the object's value

Instance variables.

```
public class Charge
{
   private final double rx, ry;
   private final double q;
   .
   .
   .
}
```

instance variable declarations

access modifiers

Constructors.

access modifier

no return type

constructor name (same as class name)

parameter variables

```
public Charge ( double x0 , double y0 , double q0 )
{
   rx = x0;
   ry = y0;
   q = q0;
}
```

signature

instance variable names

body of constructor

Instance methods.

access modifier

return type

method name

paramater variables

```
public double potentialAt(double x, double y)
{
   double k  = 8.99e09;
   double dx = x - rx;
   double dy = y - ry;
   return k * q  / Math.sqrt(dx*dx + dy*dy) ;
}
```

signature

paraameter variable name

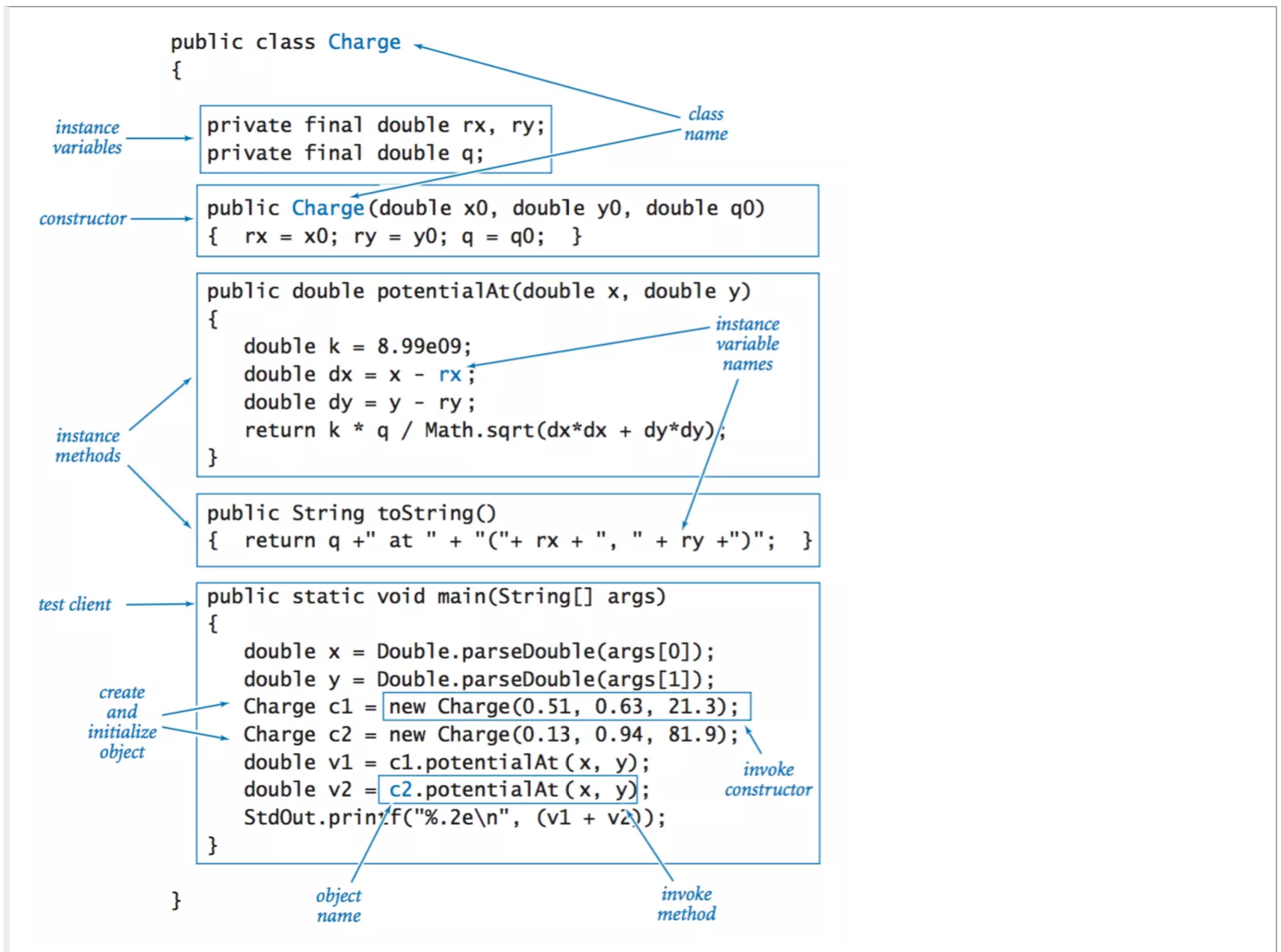
instance variable name

local variables

call on a static method

local variable name

Classes.



String Methods

```
String a = new String("now is");
String b = new String("the time");
String c = new String(" the");
```

The following declarations are also permitted:
String a = "now is";
String b = "the time";
String c = " the";

instance method call	*return type*	*return value*
`a.length()`	`int`	`6`
`a.charAt(4)`	`char`	`'i'`
`a.substring(2, 5)`	`String`	`"w i"`
`b.startsWith("the")`	`boolean`	`true`
`a.indexOf("is")`	`int`	`4`
`a.concat(c)`	`String`	`"now is the"`
`b.replace("t", "T")`	`String`	`"The Time"`
`a.split(" ")`	`String[]`	`{ "now", "is" }`
`b.equals(c)`	`boolean`	`false`