Nuances of working with variables

1. Constants

Many other programming languages have *constants*, that is, variables whose values cannot be changed. Usually, they are used for some kind of fundamental thing such as the number `Pi` or the number of days in the months of the year. That said, in principle, a programmer can make any variable a constant, if he or she decides that doing so is necessary.

So how do you declare an immutable variable (constant) in Java? There's a special keyword for this: `final`. Creating an immutable variable looks the same as creating an ordinary one. The only difference is that before the variable's type you need to write the word `final`, like this:

```
final Type name = value;
```

If you try to assign a different value to a `final` variable, then your program simply won't compile.

A `final` variable must be initialized (a value must be assigned to it) when it is declared. There is one exception to this rule: you can move initialization of a static class variable into a constructor. But you'll learn about this in Level 10.

To reduce the number of keywords, Java developers use the word `final` for more than just declaring constants. `final` can also apply to methods and even classes. Methods declared as `final` cannot be overridden, and a class declared as `final` cannot be inherited.

The `final` modifier may be added before any variables: local variables, method parameters, class fields, and static class variables.

Note that `final` before a variable name is just protection against any changes to that variable. If a variable stores a reference to an object, then the object can still be changed.

Example:

`final int[] data = {1, 2, 3, 4, 5, 6};`
 	
`data = {6, 7, 8, 9};`
 	
`data[0] = 0;`
`data[1] = 0;`
`data[2] = 0;`
	We create an array.
	
	This is not allowed: the `data` variable is declared as `final`.
	
	But you can do this.
	And also this.

Global constants

If you decide to declare global constants in your program, then you need to create *static class variables*, and make them `public` and `final`. There is a special style for the names of such variables: they are written in all capital letters, with an underscore character used to separate words.

Examples:

```
class Solution
{
   public static final String SOURCE_ROOT = "c:\\projects\\my\\";
   public static final int DISPLAY_WIDTH = 1024;
   public static final int DISPLAY_HEIGHT = 768;
}
```

2. Variable shadowing

As we said before, you cannot create several local variables with the same name in the same method. In different methods, you can.

But here's what you probably don't know: instance variables and local method variables can have the same name.

Example:

Code	Variable visibility

`public class Solution`
`{`
`  public int count = 0;`
`  public int sum = 0;`
 	
`  public void add(int data)`
`  {`
`  sum = sum + data;`
`  int sum = data * 2;`
`  count++;`
`  }`
`}`
	`count`
	`count, sum`
	`count, sum`
	`count, sum`
	`count, sum, data`
	`count, sum, data`
	`count, sum, data`
	`count, sum, data`
	`count, sum`

In the `add` method, we declared a local variable named `sum`. Until the end of the method, it shadows (or **masks**) the `sum` instance variable.

Okay, you say, that is to be expected in some sense. But that's not the end of the story. It turns out that if an instance variable is shadowed by a local variable, there is still a way to refer to the instance variable within the method. We do this by writing the `this` keyword before its name:

```
this.name
```

Here's an example where the name conflict is successfully resolved:.

Code	Variable visibility

`public class Solution`
`{`
`  public int count = 0;`
`  public int sum = 0;`
 	
`  public void add(int data)`
`  {`
`  int sum = data * 2;`
`  this.sum = this.sum + data;`
`  count++;`
`  }`
`}`
	`this.count`
	`this.count, this.sum`
	`this.count, this.sum`
	`this.count, this.sum`
	`this.count, this.sum, data`
	`this.count, this.sum, data, sum`
	`this.count, this.sum, data, sum`
	`this.count, this.sum, data, sum`
	`this.count, this.sum`

The `count` and `sum` variables are available everywhere with or without the `this` keyword. On lines where the `sum` local variable shadows the `sum` instance variable, the `sum` instance variable can only be accessed using the `this` keyword.

If a static class variable rather than an instance variable is shadowed, then you need to access it through the class name rather than the `this` keyword:

```
ClassName.name
```

Example:

Code	Variable visibility

`public class Solution`
`{`
`  public static int count = 0;`
`  public static int sum = 0;`
 	
`  public void add(int data)`
`  {`
`  int sum = data * 2;`
`  Solution.sum = Solution.sum + data;`
`  count++;`
`  }`
`}`
	`Solution.count, Solution.sum`
	`Solution.count, Solution.sum`
	`Solution.count, Solution.sum`
	`Solution.count, Solution.sum`
	`Solution.count, Solution.sum`
	`Solution.count, Solution.sum, data`
	`Solution.count, Solution.sum, data, sum`
	`Solution.count, Solution.sum, data, sum`
	`Solution.count, Solution.sum, data, sum`
	`Solution.count, Solution.sum`

You can access the `count` and `sum` static variables everywhere with or without using the class name `Solution` as a prefix. In those lines where the `sum` local variable shadows the `sum` instance variable, access to the `sum` instance variable is possible only when using `Solution` as a prefix.

3. Variables inside a `for` loop

And one more small but interesting fact.

There's also a place where a variable is declared in a special way — inside a **`for` loop**.

You may recall that a `for` loop typically has a counter variable in parentheses. And what will be the visibility of this variable? After all, it's not in the body of the loop. Is it the whole method? Or not?

The correct answer is: a variable declared in the header of a **`for` loop** is visible only in the **body of the loop** and in the **header of the `for` loop**.

Example:

Code	Variable visibility

`public static void main(String[] args)`
`{`
`  int a = 0;`
 	
`  for (int i = 0; i < 10; i++)`
`  {`
`  System.out.println(i);`
`  }`
 	
`  System.out.println("end");`
`}`
	
	a
	a
	a, i
	a, i
	a, i
	a
	a
	a

So, you can always write loops one after another in your code and use counter variables with the same name — that won't create any problems.

Example:

Code	Variable visibility

`public static void main(String[] args)`
`{`
`  int a = 0;`
 	
`  for (int i = 0; i < 10; i++)`
`  {`
`  System.out.println(i);`
`  }`
 	
`  for (int i = 0; i < 10; i++)`
`  {`
`  System.out.println(i);`
`  }`
 	
`  System.out.println("end");`
`}`
	
	a
	a
	a, i
	a, i
	a, i
	a
	a
	a, i
	a, i
	a, i
	a
	a
	a