

Naming Conventions



Naming is not a side issue. It is one of the most fundamental mechanisms. This section deals with good and bad naming in models and code.

This section is based on Stephen Kelvin Friedrich's article "What's in a name", published in JavaPro in October 2004. Elements not relevant to this course have been omitted. Stephen is Gentleware's former Chief Software Architect and permission has been granted.

What's in a Name?

Though a coding element will perform the way it was designed to perform, regardless of its name, it's good practice to choose semantics wisely.

Large projects can involve hundreds, if not thousands of classes. Working with models that are unknown makes one issue very pressing—names. We are not talking about syntactical issues, which are covered in almost every coding standard and generally agreed upon. Rather, we are talking about the relation of a name and the meaning or semantics of the concept it names.

Most of the designing, modelling and code writing capture some concepts that are grounded in real-world counterparts. For these concepts you need good names to easily establish the relationship from virtual to real. Other concepts are pure inventions without a real-life counterpart. These, even more so, need good names to visualize their meanings.

Is it hard to choose good names? Not really. Most modellers and programmers take a casual approach to this task. Being focused on getting the work done means that naming is perceived as an obstacle that gets in the way. Code will run just as well if you name a variable *a* or if you name it *averageGasConsumption*. Still, a name is meant to communicate something to those who will work with the model or code after you have finished it and that may even be you a week later. You will write that line of code only once, but chances are you and your team will be reading it a dozen times later on. Several factors contribute to poor naming habits.

Poor Naming Habits

Poor instruction

If your instructors spent more time on lessons of good naming than the occasional lecture to "choose expressive names," you are lucky. For sure, if you named those variables in your calculator homework assignment *a* and *b*, you would get back some bad remarks, but *result* and *plus* would have been dandy.

Bad examples

Most code examples you see are short and specific. That comes naturally, of course, with their intention to explain fundamental algorithms, APIs, or language features. So, examples typically neglect anything else, which may be justified in some areas like error checking and exception handling (those issues at least receive some attention separately). Yet good names would do nothing but help most examples be more easily understood.

No guidelines, no conventions

You already have your (company-)specific set of code style guidelines—if not, start with Sun's developers' site. It probably advocates variable names to start in lowercase, class names to start in uppercase, and continue in camel case. Maybe it also tells you to use nouns for class and variable names and verbs for methods. Chances are that besides those (important) syntactical instructions, further directions on choosing a good name are nonexistent.

Good and Bad

Imagine the names *secretHashMap* and *mef*. What does a variable named *mef* tell you about its purpose? Nothing. If, however, you already know the application area in some depth, you might guess that it stands for *ModelElementFactory*. But if you already know, who cares about the name?

The name should be helpful to the unknowing eye. A similar argument holds for *secretHashMap*. All that the uninitiated reader might infer from it is: "some hack going on here." Staying for a moment with the importance of semantic value, other examples come to my mind and all of these should almost always make you reconsider the concept you are trying to name: *Manager*, *Handler*, *System*, and *Process*. Like a university professor once said, "If you don't know what it is, call it a system. If you don't know what it does, call it a process."

Also, please let go of that infamous *i*. It is not the worst thing to use for an integer variable in a small loop, and it traditionally points to that very syntax (`int loop variable`). But then with modern IDE's code completion, it is not much more work to type *bookIndex* instead, which will be much nicer for future readers.

Think of *banana*. It's a fruit—yellow, tasty, peel, grocery—right? It could be, or it could be that large, plastic inflatable that's fun to ride on and pulled by a motorboat through the bay outside my holiday hotel. If a name leads you to think of one type of object, while in fact it refers to some different type of object altogether, you need to get more specific. (Something like *bananaBoat* would have provoked more accurate associations.) This issue is not about semantic value itself, but about using correct semantics.

This kind of misuse is common. For example, have a look at Swing's `JTable` implementation: `TableCellRenderer getCellRenderer(int row, int column)`; Come again? A *column* is an *int*? This complaint might seem nitpicky, as we all understand the method's purpose, but this bad practice is so widespread that the overall quality of code will improve a lot if you choose better names instead (which would be *rowIndex* and *columnIndex* in this case, of course). This is even more so, because you often see a variable without seeing its declaration also.

Better Names

Some practices are proposed that can improve models and code a lot and help achieve better quality. Naming is not a side issue. It is the most fundamental mechanism of your work. It helps you to think about what you want to do and to communicate that clearly. It speeds up development by making your code more solid and maintainable. Take your time to find a good name. Consequences of naming differ in importance, depending on the type of name.

Package names

Discussing these with your team can establish a common understanding of what is supposed to be in that package and how it relates to the system as a whole.

Class and interface names

Classes are the fundamental building blocks of OO software, and names are what give you a handle on classes. So a bad name can make somebody try to build a wall with a roof tile, which might work but will damage the overall architecture of your house. Put some effort into choosing a name that captures the concept that the class represents. Doing so will give you a better idea of that concept itself.

Public member names

(You don't have any members public but methods and constants, right?) These names represent the interface of a class, so bad names are an invitation for your co-workers to misuse the class and spend ages on debugging and rereading your code. A single important convention is not to name a method `get...()` if it changes the (externally visible) state of a class.

Other member names

These names will be read whenever somebody works on your model or code. If you are sloppy with naming, you are claiming that this is the perfect, bug-free, fast implementation that won't ever have to be touched again.

Local variables

A good name has value even here, yet the meaning can often be inferred quickly from the context. Don't lead readers down the wrong track: naming an *index* into a list of actors *i* doesn't tell the reader much, but naming it *modelElement* will obscure the code.

It is helpful to think like a reader, not like the author. When you want to introduce a name, you probably already have an idea of the concept. Better yet, try to think the other way around: Imagine that you have no idea of the concept, but you have a name only. Does it make you think of the right concept?

Ask your colleagues. Even if you try to take the reader's viewpoint, you still might fail, typically, because your knowledge in the area you are working on is deeper than anyone else's knowledge. Ask, "What do you think of when you hear *ProjectHandler*?" Check it against your concept of *ProjectHandler*. We came up with *ProjectReaderWriter* instead, which is a bit clumsy but at least it says what it is used for.

Say what it is. Have you ever met an *int* in real life? A tiny 3 hopping across your way without telling you whether it's three pieces of cake or three little dwarfs? Ask yourself what it is. Choose the appropriate suffix for your name: *index*, *weight*, *milliSeconds*, *count*, and so on. The same goes for variables of type string; don't name it *actor* if it is an *actorName*.

Avoid abbreviations, except when they are very common. Remember code completion. It saves you only a few keystrokes, but a proper name saves somebody else from guesswork. Using abbreviations also makes it harder to rename variables accordingly when you rename a class. Not even Idea can propose a new name for a variable *prh* of class *ProjectHandler* when you rename that class to *ProjectReaderWriter*.

Of course, you do want to use very well-known abbreviations such as *tutorialUrl*. If you decide to use an abbreviation, make only the first letter uppercase to show word separations clearly. Compare *HTTPURL* and *httpUrl*.



This section demonstrated the importance of naming different elements in models and (pseudo) code.