Chapter 2 - Meaningful Names

Names are everywhere in software. Files, directories, variables functions, etc. Because we do so much of it. We have better do it well.

Use Intention-Revealing Names

It is easy to say that names reveal intent. Choosing good names takes time, but saves more than it takes. So take care with your names and change them when you find better ones.

The name of a variable, function or class, should answer all the big questions. It should tell you why it exists, what it does, and how is used. **If a name requires a comment, then the name does not reveals its intent**.

Does not reveals intention	Reveals intention
`int d; // elapsed time in days`	`int elapsedTimeInDays`

Choosing names that reveal intent can make much easier to understand and change code. Example:

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

This code is simple, but create many questions:

1. What is the content of `theList`?
2. What is the significance of the item `x[0]` in the list?.
3. Why we compare `x[0]` vs 4?
4. How would i use the returned list?

The answers to these questions are not present in the code sample, but they could have been. Say that we're working in a mine sweeper game. We can refactor the previous code as follows:

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Now we know the next information:

1. `theList` represents the `gameBoard`
2. `x[0]` represents a cell in the board and `4` represents a flagged cell
3. The returned list represents the `flaggedCells`

Notice that the simplicity of the code has not changed. It still has exactly the same number of operators and constants, with exactly the same number of nesting levels. But the code has become much more explicit.

We can improve the code writing a simple class for cells instead of using an array of `ints`. It can include an **intention-revealing function** (called it `isFlagged`) to hide the magic numbers. It results in a new function of the function.

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Avoid Disinformation

Programmers must avoid leaving false clues that obscure the meaning of code. We should avoid words whose entrenched meaning vary from our intended meaning.

Do not refer to a grouping of accounts as an `accountList` unless it's actually a `List`. The word `List` means something specific to programmers. If the container holding the accounts is not actually a List, it may lead to false conclusions. So `accountGroup` or `bunchOfAccounts` or just plain `accounts` would be better.
Beware of using names which vary in small ways. How long does it take to spot the subtle difference between a `XYZControllerForEfficientHandlingOfStrings` in one module and, somewhere a little more distant, `XYZControllerForEfficientStorageOfStrings`? The words have frightfully similar shapes

Make Meaningful Distinctions

Programmers create problems for themselves when they write code solely to satisfy a compiler or interpreter. For example because you can't use the same name to refer two different things in the same scope, you might be tempted to change one name in an arbitrary way. Sometimes this is done by misspelling one, leading to the surprising situation where correcting spelling errors leads to an inability to compile. Example, you create the variable `klass`because the name `class` was used for something else.
In the next function, the arguments are noninformative, `a1` and `a2` doesn't provide clues to the author intention.

```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
```

```
}
```
We can improve the code selecting more explicit argument names:

```
public static void copyChars(char source[], char destination[]) {
  for (int i = 0; i < source.length; i++) {
    destination[i] = source[i];
  }
}
```
Noise words are another meaningless distinction. Imagine that you have a Product class. If you have another called `ProductInfo` or `ProductData`, you have made the names different without making them mean anything different. Info and Data are indistinct noise words like a, an, and the. Noise words are redundant. The word variable should never appear in a variable name. The word table should never appear in a table name.

Use Pronounceable Names

Imagine you have the variable `genymdhms` (Generation date, year, month, day, hour, minute and second) and imagine a conversation where you need talk about this variable calling it "gen why emm dee aich emm ess". You can consider convert a class like this:

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /* ... */
};
```
To

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /* ... */
};
```

Use Searchable Names

Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.

Avoid Encoding

We have enough encodings to deal with without adding more to our burden. Encoding type or scope information into names simply adds an extra burden of deciphering. Encoded names are seldom pronounceable and are easy to mis-type. An example of this, is the use of the Hungarian Notation or the use of member prefixes.

Interfaces and Implementations

These are sometimes a special case for encodings. For example, say you are building an ABSTRACT FACTORY for the creation of shapes. This factory will be an interface and will be implemented by a concrete class. What should you name them? `IShapeFactory` and `ShapeFactory`? Is preferable to leave interfaces unadorned.I don't want my users knowing that I'm handing them an interface. I just want them to know that it's a `ShapeFactory`. So if I must encode either the interface or the implementation, I choose the implementation. Calling it `ShapeFactoryImp`, or even the hideous `CShapeFactory`, is preferable to encoding the interface.

Avoid Mental Mapping

Readers shouldn't have to mentally translate your names into other names they already know.

One difference between a smart programmer and a professional programmer is that the professional understands that clarity is king. Professionals use their powers for good and write code that others can understand.

Class Names

Classes and objects should have noun or noun phrase names like `Customer`, `WikiPage`, `Account`, and `AddressParser`. Avoid words like `Manager`,`Processor`, `Data`, or `Info` in the name of a class. A class name should not be a verb.

Method Names

Methods should have verb or verb phrase names like `postPayment`, `deletePage` or `save`. Accessors, mutators, and predicates should be named for their value and prefixed with get, set, and is according to the javabean standard.
When constructors are overloaded, use static factory methods with names that describe the arguments. For example:

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

Is generally better than

```
Complex fulcrumPoint = new Complex(23.0);
```

Consider enforcing their use by making the corresponding constructors private.

Don't Be Cute

Cute name	Clean name
`holyHandGranade`	`deleteItems`

Cute name	Clean name
`whack`	`kill`
`eatMyShorts`	`abort`

Pick one word per concept

Pick one word for one abstract concept and stick with it. For instance, it’s confusing to have fetch, retrieve, and get as equivalent methods of different classes.

Don’t Pun

Avoid using the same word for two purposes. Using the same term for two different ideas is essentially a pun.

Example: in a class use `add` for create a new value by adding or concatenating two existing values and in another class use `add` for put a simple parameter in a collection, it's a better options use a name like `insert` or `append` instead.

Use Solution Domain Names

Remember that the people who read your code will be programmers. So go ahead and use computer science (CS) terms, algorithm names, pattern names, math terms, and so forth.

Use Problem Domain Names

When there is no “programmer-eese” for what you’re doing, use the name from the problem domain. At least the programmer who maintains your code can ask a domain expert what it means.

Add Meaningful context

There are a few names which are meaningful in and of themselves—most are not. Instead, you need to place names in context for your reader by enclosing them in well-named classes, functions, or namespaces. When all else fails, then prefixing the name may be necessary as a last resort

Variables like: `firstName`, `lastName`, `street`, `city`, `state`. Taken together it's pretty clear that they form an address, but, what if you saw the variable state being used alone in a method?, you could add context using prefixes like: `addrState` at least readers will understand that the variable is part of a large structure. Of course, a better solution is to create a class named `Address` then even the compiler knows that the variables belong to a bigger concept

Don’t Add Gratuitous Context

In an imaginary application called “Gas Station Deluxe,” it is a bad idea to prefix every class with GSD. Example: `GSDAccountAddress`
Shorter names are generally better than longer ones, so long as they are clear. Add no more context to a name than is necessary.