**Naming**

Variable names should always be meaningful.

**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.4

* Programmers must avoid leaving false clues that obscure the meaning of code.   
  Ex- 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.1 So accountGroup or bunchOfAccounts or just plain accounts would be better.
* 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.
* 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. How do you remember which method name goes with which class?

**Functions**

FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL.

THEY SHOULD DO IT ONLY.

The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification—and then shouldn’t be used anyway.

n the absence of specific conventions, the variable moneyAmount is indistinguishable from money, customerInfo is indistinguishable from customer, accountData is indistinguish-able from account, and theMessage is indistinguishable from message. Distinguish names in such a way that the reader knows what the differences offer.

* Anything that forces you to check the function signature is equivalent to a double-take. It’s a cognitive break and should be avoided.
* Either your function should change the state of an object, or it should return some information about that object. Doing both often leads to confusion.
* Prefer Exceptions to Returning Error Codes

**Comments**

1. Nothing can be quite so helpful as a well-placed comment. Nothing can clutter up a module more than frivolous dogmatic comments. Nothing can be quite so damaging as an old crufty comment that propagates lies and misinformation.
2. The proper use of comments is to compensate for our failure to express ourself in code.
3. if you find yourself wanting to mark your closing braces, try to shorten your functions instead.

**Formatting**

1. You should choose a set of simple rules that govern the format of your code, and then you should consistently apply those rules. If you are working on a team, then the team should agree to a single set of formatting rules and all members should comply.
2. There are blank lines that separate the package declaration, the import(s), and each of the functions. This extremely simple rule has a profound effect on the visual layout of the code. Each blank line is a visual cue that identifies a new and separate concept.
3. Concepts that are closely related should be kept vertically close to each other. closely related concepts should not be separated into different files unless you have a very good reason. If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible.

**OOPs**

1. Why, then, do so many programmers automatically add getters and setters to their objects, exposing their private variables as if they were public?   
   We should avoid this.
2. when we want to add new data types rather than new functions. For these cases objects and OO are most appropriate. On the other hand, there will also be times when we’ll want to add new functions as opposed to data types. In that case procedural code and data structures will be more appropriate. Mature programmers know that the idea that everything is an object is a myth. Sometimes you really do want simple data structures with procedures operating on them.
3. a method f of a class C should only call the methods of these:

• C

• An object created by f

• An object passed as an argument to f

• An object held in an instance variable of C

The method should not invoke methods on objects that are returned by any of the allowed functions. The following code3 appears to violate the Law of Demeter (among other things) because it calls the getScratchDir() function on the return value of getOptions() and then calls getAbsolutePath() on the return value of getScratchDir().

final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();

**Data Transfer Objects**

1. quite essential form of a data structure is a class with public variables and no functions. This is sometimes called a data transfer object, or DTO. DTOs are very useful structures, especially when communicating with databases or parsing messages from sockets, and so on. They often become the first in a series of translation stages that convert raw data in a database into objects in the application code.

**Active Records**

1. Active Records are special forms of DTOs. They are data structures with public (or beanaccessed) variables; but they typically have navigational methods like save and find. Typically these Active Records are direct translations from database tables, or other data sources.
2. to treat the Active Record as a data structure and to create separate objects that contain the business rules and that hide their internal data

**Don’t Pass Null**

1. Returning null from methods is bad, but passing null into methods is worse. Unless you are working with an API which expects you to pass null, you should avoid passing null in your code whenever possible.

**Clean Tests**

1. Instead of experimenting and trying out the new stuff in our production code, we could write some tests to explore our understanding of the third-party code. In learning tests we call the third-party API, as we expect to use it in our application. We’re essentially doing controlled experiments that check our understanding of that API.
2. having dirty tests is equivalent to, if not worse than, having no tests.
3. Clean Tests follow five rules :
4. FAST - When tests run slow, you won’t want to run them frequently. If you don’t run them frequently, you won’t find problems early enough to fix them easily
5. Independent - u should be able to run each test independently and run the tests in any order you like.
6. Repeatable - should be able to run the tests in the production environment, in the QA environment, and on your laptop while riding home on the train without a network. You’ll also find yourself unable to run the tests when the environment isn’t available
7. Self-Validating - Either they pass or fail. You should not have to manually compare two different text files to see whether the tests pass. If the tests aren’t self-validating, then running the tests can require a long manual evaluation.
8. Timely - The tests need to be written in a timely fashion. Unit tests should be written just before the production code that makes them pass. If you write tests after the production code, then you may find the production code to be hard to test.

**Classes**

1. **Classes** should be how small? - With classes we use a different measure. We count responsibilities. The Single Responsibility Principle (SRP)2 states that a class or module should have one, and only one, reason to change.
2. Classes should have a small number of instance variables. Each of the methods of a class should manipulate one or more of those variables. IA class in which each variable is used by each method is maximally cohesive. we would like cohesion to be high. When cohesion is high, it means that the methods and variables of the class are co-dependent and hang together as a logical whole
3. Classes should be open for extension but closed for modification. abstract classes, which represent concepts only. A client class depending upon concrete details is at risk when those details change. We can introduce interfaces and abstract classes to help isolate the impact of those details.

**System**