Readability

[Skrien §4.0] “Programs must be written for people to read, and only incidentally for machines to execute.”—Abelson & Sussman

Use a good set of coding conventions, such as the ones given in the Google Java Style Guide.

Pick a convention you deem worthy of note.

Several good coding practices are covered in the Skrien text.

Guideline: Give a variable the narrowest scope that you can.

Give an example of this principle.

Why is this a good principle?

Guideline: Using standard idioms, make code as concise as possible.

Example: In the following statement, b is a boolean variable:

```java
    if (b == true)
        return true;
    else
        return false;
```

This statement is far too verbose. An equivalent and much more readable statement is—

In most cases, this can be made even more readable. How?

Guideline: Variable names should be neither too short nor too long.

Consider a variable that controls whether a while-loop is exited.
while (variable) {
    ...
}

What is a good name for this variable?

Which should be shorter, in general? Variable names or names of constants?

In general, names that are used less frequently can be longer.

*Guideline:* Names should be descriptive of the entity they apply to. They should not be vague or overly general.

*Give an example* of a bad (variable, method, etc.) name you have encountered in code that you were refactoring or interfacing to.

*Here* are some examples from Expertiza.

*Guideline:* Names should not be redundant.

Suppose `course_controller.rb` contains a method called `create_course`. What should it be?

Suppose it contains a method called `create_section`. What should it be?

*Guideline:* Factor out duplicated code.

If a program has two places where the same sequence of instructions is being executed, it is almost always beneficial to move the duplicated code into a separate procedure.

*Example:* Suppose you are developing a class of objects one of whose responsibilities is to parse an input string, such as a complicated mathematical expression.

Part of the process of parsing involves checking that the input is valid. So the class might have a method like this:
public void parse(String expression) {
    ...do some parsing...
    if( ! nextToken.equals("+") ) {
        //error
        System.out.println("Expected +, but found " + nextToken);
        System.exit(1);
    }
    ...do some more parsing...
    if( ! nextToken.equals("*") ) {
        //error
        System.out.println("Expected *, but found " + nextToken);
        System.exit(1);
    }
    ...
}

How can we clean this code up?

private void handleError(String message) { 
    System.out.println(message);
    System.exit(1);
}

public void parse(String expression) {
    ...do some parsing...
    if( ! nextToken.equals("+") )
        ...do some more parsing...
    if( ! nextToken.equals("*") )
        ...
}

Besides being more readable, this code has another advantage. What?
**Guideline:** A method should do only one thing and do it well.

Here's an example of a method to avoid:

```java
void doThisOrThat(boolean flag) {
    if (flag) {
        /* twenty lines of code to do this... */
    }
    else {
        /* twenty lines of code to do that... */
    }
}
```

How should we change it?

**Using assertions for correctness**

How can we know that software is correct?

It is only correct if it does what it is supposed to do.

But how do we know what it is supposed to do? **We need a specification.** For example, is

\[ x = y + 5 \]

a correct statement? You need to know what the statement is supposed to do.

If the specification is,

"Make sure \( x \) and \( y \) have different values,"

then the statement is ________.

But if the specification is, "Make sure that \( x \) has a negative value,"

then the statement is ________.

So, software correctness is a relative notion.
Specifications are expressed through assertions. Not only do assertions help us determine whether the program is correct, they also—

- help document the program, and
- provide a basis for systematic testing and debugging.

A correctness formula is an expression

\[ \{ P \} \ S \{ Q \} \]

It means that any execution of \( S \), starting in a state where \( P \) holds, will terminate in a state where \( Q \) holds.

In this formula,

- \( P \) is called the precondition, and
- \( Q \) is called the postcondition.

This sounds simple, but the implications take some getting used to.

Here is a trivial correctness formula, which holds as long as \( x \) is an integer:

\[ \{ x \geq 9 \} \ x = x + 5 \ \{ x \geq 13 \} \]

It looks like this is a typo. After all, we could also strengthen the postcondition to read,

\[ \{ x \geq 14 \} \]

This postcondition is stronger than the original postcondition, because there are values of \( x \) which satisfy the original postcondition but do not satisfy it.

Given the original precondition, this is the strongest postcondition we can give and still have the program be correct.

Similarly, given the original postcondition, we can also weaken the precondition. What is the weakest precondition we can give involving \( x \) to have the program be correct?
From a formula that holds, you can always get another one that holds by strengthening the _______ or weakening the _______.

It is also interesting to consider what the strongest possible precondition would be.

It is possible to give a precondition strong enough that the program is correct regardless of what it does.

What would such a precondition be?

Similarly, we can make things easy for the program by weakening the postcondition as much as possible:

*Any* precondition and program will satisfy this.

Preconditions and postconditions can be introduced into programs by using *assertions*. Java has an *assert* statement (more on this later).

*Example: nth roots*  
[Skrien §4.6] Consider this method, and its preconditions and postconditions:

```java
//Precondition: n ≥ 0 and (n is odd or value ≥ 0)  
//Postcondition: The n-th root of the double  
//              value is returned.  
//              If n is even, the positive n-th root  
//              is returned.  
//              If n is odd, the n-th root will have  
//              the same sign as the value.  
public double nthRoot(double value, int n) {...}
```

Whose responsibility is it to ensure that the preconditions are satisfied before calling the method?
Suppose the preconditions are not satisfied. What can happen when the method is called?

However, it is usually better if clients of the routine know what could happen in all cases. What can we do to make this happen?

This leads to the following code:

```java
//Postcondition: The n-th root of the double value is returned.
// If n is even, the positive n-th root is returned.
// If n is odd, the n-th root will have the same sign as the value.
//
public double nthRoot(double value, int n) {...}
```

**Why is this better?**

Is it always good to check whether the (original) preconditions are satisfied?

It is a very good idea to write our code so that preconditions and postconditions can be checked. This is done with the Java `assert` statement.

The `assert` statement has the form

```
assert expression1;
```

or

```
assert expression1: expression2;
```

The first form simply raises an `AssertionError` if the assertion is not true.

The second form associates the value of `expression2` with the `AssertionError` if an error is triggered. The value of `expression2` should be something that will be useful in debugging.
We could code the postcondition checks at the end of the `nthRoot` method (assuming it returns `result`):

```java
assert Math.pow(result, (float) n) == value);
if (n % 2 == 0) assert result > 0;
else assert result * value > 0;
```

Because checking of assertions may be quite expensive, it can be enabled or disabled at run time:

- To enable assertions at various granularities, use the `-enableassertions`, or `-ea`, switch.

- To disable assertions at various granularities, use the `-disableassertions`, or `-da`, switch.

- You specify the granularity with the arguments that you provide to the switch. See Enabling and Disabling Assertions.

By default, assertions are disabled at run time.

Consider again the Liskov Substitution Principle:

Class `S` should be made a subclass of class `C` only if, for every method in `C`'s and `S`'s interface, `S`'s method

- accepts as input all the values that `C`'s method accepts (and possibly more) and
- does everything with those values that `C`'s method does (and possibly more).

How should we **rephrase it** in terms of preconditions and postconditions?

Class `S` should be made a subclass of class `C` only if, for every method with identical signatures in `C`'s and `S`'s interface,

- the preconditions for `C`'s method are no weaker than the ___ for `S`'s method, and
- the postconditions for `C`'s method are no stronger than the __________ for `S`'s method.
**Class invariants**

[Skrien §4.5] Preconditions and postconditions tell us what happens while a method of class $C$ is executing.

Class invariants tell us what is true while a method of class $C$ is *not* executing.

Stated another way, preconditions and postconditions express properties of individual routines.

We also need a way to express the global properties of instances of a class, properties that must be preserved by all routines.

Such properties will make up the class invariant, capturing the integrity constraints characterizing a class.

Consider the `MyStack` example from Week 10:

```java
class MyStack
{
    private java.util.Stack stack;
    public MyStack() { stack = new java.util.Stack(); }
    public void push(Object o) { stack.push(o); }
    public Object pop() { return stack.pop(); }
    public Object peek() { return stack.peek(); }
    public boolean isEmpty() { return stack.empty(); }
}
```

It uses a `java.util.Stack()`, which has *capacity* and *size* attributes.

The attributes of the class—Vector *representation* and integers *capacity* and *size*—constitute the representation of a stack.

Some properties cannot easily be expressed by preconditions and postconditions.

An example is

$$0 \leq size; size \leq capacity$$
A class invariant applies to every class instance, as a whole.

This is different from preconditions and postconditions, which characterize individual routines.

**Programming by contract**

By associating pre- and postcondition assertions with a method \( m \), the class tells its clients:

“If you promise to call \( m \) with \( pre \) satisfied then I, in return, promise to deliver a final state in which \( post \) is satisfied.

As in human relationships, any good contract between classes entails **obligations as well as benefits** for both parties (the “client” and the “supplier”).

- The precondition binds the ______. It defines the conditions under which a call to the routine is legitimate.
  
  It is an obligation for the ___ and a benefit for the ______ ___.

- The postcondition binds the ______. It defines the conditions that must be ensured by the routine on return.
  
  It is an obligation for the _________ and a benefit for the _____ ______.

Let us take a look at **one of the contracts** we specified in the \( n \)th-root method.

<table>
<thead>
<tr>
<th>( n )th root</th>
<th>Obligations</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>(Satisfy precondition)</td>
<td>(From postcond.)</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>(Satisfy postcond.)</td>
<td>(From precond.)</td>
</tr>
</tbody>
</table>

---

1 This discussion is taken from Bertrand Meyer’s *Object-Oriented Software Construction*, §11.6,
In case you haven’t noticed it, one of the contract rules goes against conventional software-engineering wisdom.

It is shocking to some, but is one of the method’s main contributions to software reliability.

*Recall:* If the precondition is not satisfied, the supplier doesn’t have to do anything at all.

It can go into an infinite loop, or even crash the system without violating the contract.

This implies that when you are writing a routine, it’s not your responsibility to check whether the preconditions are satisfied.

For example, if a square-root function has the precondition

\[ x \geq 0 \]

you don’t need to check whether the argument really is \( \geq 0 \).

Actually, it is not only useless, but actually harmful, to test whether the precondition is met.

This is Meyer’s *non-redundancy* principle:

Under no circumstances shall the body of a routine ever test for the routine’s precondition.

This rule is the reverse of “defensive programming” advocated by many software-engineering textbooks.

What is the rationale for this rule?

Design by contract invites you—

- to identify the consistency conditions that are necessary to the proper functioning of each client-supplier contract, and
- to specify, for each one of the conditions, whose responsibility it is to enforce it.
Meyer calls this “Zen and the art of software reliability: guaranteeing more by checking less.”

Who is the enforcer? OK, so the client and the supplier aren’t both supposed to check that the condition is satisfied.

Who should be the one to check it? One of the choices is “tolerant,” while the other is “demanding”:

- **Choice 1 (“demanding”):** The responsibility of checking is assigned to clients.
  
  Then where will the condition appear?

- **Choice 2 (“tolerant”):** The responsibility of checking is assigned to the supplier.
  
  Then where will the condition appear?

Which one of these approaches is “tolerant”?

Which approach is better, tolerant or demanding? Well, …

What is an advantage of the tolerant style?

What is an advantage of the demanding style?

A good routine author “does not try to outsmart his clients.” If he is not sure what to do in a specific case, he excludes it in the precondition.