Skip to main content

JavaHyperText and Data Structures

David Gries, with help from many

Home | Java HyperText | Style Guide | Eclipse | API and strings | Explain constructs | Exceptions | Abstract classes interfaces | Iterator & Iterable | Program correctness | Loop invariants | Stepwise refinement | Recursion Hashing | DFS BFS | Shortest path

Java Code Style Guidelines

One goal of a programming course is for you to learn to write programs that are not only correct but also understandable. These guidelines should help you toward that goal. They will give you a good basis for developing a style of your own as you become a more experienced programmer. This page includes guidelines for Java constructs that will be covered later in the semester. Skim these sections now and read them more carefully later, when the topics are discussed.

Some of these guidelines are standard Java practice, while others are intended to develop good coding habits. When in doubt, choose the alternative that makes the code most obviously correct:

> "There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. The other way is to make it so complicated that there are no obvious deficiencies." - C.A.R. Hoare.

1. Naming Conventions

We adhere to Java's naming conventions. These may seem arbitrary and bizarre if you're not used to them. Nonetheless, conforming to aesthetic standards of coding is something you'll need to do in just about any software development workplace, so it's good practice.

1.1 CamelCase

Java class, field, method, and variable names are written in *CamelCase*: Words are smashed together and the first letter of each word is capitalized. No word separator, like the underscore _, is used.

```
thisIsAnExample
```

There is one exception to CamelCase: constants. Constants are in all-caps with words separated by underscores. Constants should also have the keyword final, and they are usually static:

```
public static final double SPEED_OF_LIGHT= 299792458; // in m/s
```

1.2 Class and interface names

Class and interface names are generally noun or noun phrases and begin with a capital letter:

```
interface AqueousHabitat { ... }
class FishBowl implements AqueousHabitat { ... }
```

1.3 Method names

Method names generally begin with a lowercase letter. A call on a procedure is a statement to do something, so a procedure name is generally a verb phrase that is a command to do something.

```
public void setTitle(String t) { ... }
```

A function call yields a value, so a function name is generally a noun phrase that describes the value.

```
public double areaOfTriangle(int b, int c, int d) { ... }
```

However, a convention in Java is that a function that yields the value of a field (say title) is the name of the field preceded by "get". People are of two minds on this; not everyone thinks it's a good idea. The name "title" could also be used.

```
public String getTitle() { ... }
```

The name of a boolean function is often a verb phrase starting with "is", thus describing what the value means:

```
public boolean isEquilateralTriangle(int b, int c, int d) { ... }
```

1.4 Variable names

Variable names generally start with a lowercase letter.

Variable names should give the reader some hint about what the variable is used for. A well-chosen name, which gives a hint at the meaning of the variable, helps document a program, making it easier to understand. On the other hand, using the names of your friends or flowers that you like as variable names just annoys and makes the program harder to understand. Don't do that. Also, refrain from using vague names like *counter* or *var* or *data*; instead think about what the variable really is for and use a more concrete name.

There is a tension between writing long descriptive names and very short names. A long name may make it easy to remember what the variable is for, but its length may make the whole program seem long and complicated. It may be difficult to see what a very short name means, but it makes the program short. For example, consider these two statements, which look so different only because of the length of the variable name:

```
hypotenuseOfTriangle= 6 * (2 + hypotenuseOfTriangle) + 7 / hypotenuseOfTriangle
                    - hypotenuseOfTriangle;

x= 6 * (2 + x) + 7 / x - x;
```

A name can rarely be used to give a complete, precise definition of the entity it names, and a complete definition should always be given where the entity is defined.

Here is a compromise. We tend to use shorter names for parameters and local variables and longer names for fields and static variables. We explain why.

Parameter names. The specification of a method will name all parameters and give their meanings. The body of a method is usually fairly short ---at most 30-50 lines. Therefore, when reading the method body, the meanings of parameters are available without any scrolling. So parameter names may be short --even one letter long.

```
/** Return "lengths b, c, and d are the sides of an equilateral triangle" */
public boolean isEquilateralTriangle(int b, int c, int d) {
    return b == c && c == d;
}
```

Local variable names. A local variable is a variable that is declared in a method body. Its declaration should be placed as close to its first use as possible. The scope of a local variable is usually short, and its meaning is often obvious either from a comment on its declaration or from the short code in which it is used. Therefore, names of local variables may be short.

Fields and class (i.e. static) variables. The declarations of field and class variables may be far from their uses -- perhaps hundreds of lines away. Further, the meaning of fields and class variables are typically given as comments by the declarations, far from where the variables are used. Therefore, the names of field and class variables should be longer and as mnemonic as possible, giving the reader a good idea what the meaning are.

1.5 Package names

Package names are usually *all* lowercase and consist of nouns.

2. Format conventions

Use a formatting convention consistently within a class. For example, the position of open braces "{" should be the same throughout the program.

Put only one statement on a line (although there may be exceptional cases where more than one is better). Don't pack everything together, making a program hard to read.

Make sure that all lines can be read without the need for horizontal scrolling. A mximum of 80 characters per line is recommended.

2.1. Indentation and braces { }

Indentation is used to make the structure of a program clear. The basic rule is:

> Substatements of a statement or declaration should be indented.

For example, the declarations in a class, the body of a method, and the if-part and then-part of a conditional statement should be indented. We prefer indentation of 4 spaces, not 2. In Eclipse, you can set the amount to indent using menu item Preferences -> Java -> Code Style -> Formatter.

Eclipse makes it easy to indent a whole class consistently. Just select all lines (control-A or command-A) and press control-i or command-i.

We prefer putting an opening curly brace "{" at the end of a line, and not on a line by itself, as shown below. The closing brace "}" appears on its own line, indented as shown below.

```
if (x < y) {           if (x < y) {
    x= y;                  x= y;
    y= 0;                  y= 0;
}                         } else {
else {         (or)        x= 0;
    x= 0;                   y= y/2;
    y= y/2;            }
}
```

A second oft-used convention is to put the opening brace on a line by itself, as shown below. This has a disadvantage. A scarce resource is the number of lines that one can see on the monitor at one time, and this convention wastes that resource.

```
if (x < y)             if (x < y)
{                        {
    x= y;                   x= y;
    y= 0;                   y= 0;
}                        } else
else           (or)  {
{                              x= 0;
    x= 0;                   y= y/2;
    y= y/2;         }
}
```

If you are used to using the second convention, we suggest that you use the first convention for the remainder of the course. After that, you can choose either style for the rest of your life; you will be making an informed decision, based on experience with both.

2.2 Always use braces for control flow structures

The following is just begging for a bug:

```
if (flag) validate();
```

Can you spot the bug here?

```
if (flag) validate(); update();
```

Always use braces:

```
if (flag) {
    validate();
    update();
}
```

(This applies to other structures as well, like if-else statements for-loops, and while-loops)

That said, there *are* situations where the braces are best left out. You'll see that happening in a few places in this website.

Put a space between the keyword and the following parenthesis, to differentiate control structures from method calls:

```
//       Good if (username == null) { ... }
// Less Good if(username == null) { ... }
```

3. Documentation

There are two reasons for documenting a program well.

First, you, the programmer, need all the help you can get while writing and debugging the program. This means that documentation should be written *during* the coding/debugging process; it should not be put off until the program is considered finished. The tendency is to wait until the end, and then document. That just means you will make more errors, take more time programming, and end up with more bugs in your program. Get in the habit of documenting a field when you declare it and specifying a method *before* you write its body. If you decide to change what a method is supposed to do, change its specification first!

You document a program well so that others have an easier time understanding it. A professional programmer is well regarded if their code is easy to maintain by others. A grader of your assignment will be more likely to feel good about you when grading if your program is well documented. So, document your program well.

The Elements of Style, a famous little book on writing style by Cornell Professors W. Strunk, Jr., and E.B. White, contains several rules that are useful for programming as well as writing. Among them are:

> Omit needless words.
> Use the active voice.

Follow these rules when writing specifications of methods. For example, don't write the specification ``This function searches list x for a value y and ..." or ``Function isIn searches list x for a value y ...". Such specifications are too wordy and are not commands but descriptions. Instead, say the following.

```
/** Return (the value of the sentence) "y is in list x" */
boolean isIn(int y, List x)
```

3.1. Kinds of comments in Java, including Javadoc comments

Java has three kinds of comments. Comments that start with "//" are one-line comments: the comment ends at the end of the line. Comments that start with "/*" must end with "*/", but they may span many lines.

A Javadoc comment starts with "/**" (and therefore ends with "*/". Javadoc stands for "Java documentation". These comments have a special place in Java and also in Eclipse, when dealing with a Java program. Suppose a method is written like this:

```
/** Add e as this Butterfly's father. Precondition: ... */
public void addFather(Butterfly e) { ... }
```

If you now hover the mouse over a call of this function, e.g.

```
but.addFather(dad);
```

a pop-up window opens with the specification of the method in it! So you get to see preciely what the function call will do and what its preconditions is. Do you see how writing specifications during programming can help you?

Secondly, at any time, in Eclipse you can use menu item Project -> Generate Javadoc to extract from the program all properly written and placed javadoc comments into a bunch of webpages that document the class. Those pages are what others will use when using your program ---they will not look at the source program.

3.2 Don't over-comment

Some people have a tendency to put a "//" comment on almost every line. This is generally just noise, making it harder to read the program. Don't do it. Refrain from commenting lines like this:

```
i= i+1; // add one to i
```

Assume that the reader has experience with Java and understands basic statements. Generally speaking, the only kinds of comments needed are those that are mentioned in this section 3 on comments.

3.3 Class invariant

The class invariant is the collection of meanings of and constraints on fields of a class. The class invariant is generally placed in comments on each individual field declaration or placed as a single comment before the fields. Here are examples. Note the appearance of constraints, when necessary. We have made them javadoc comments so the Eclipse feature of showing the javadoc comments when an occurrence of the field name is moused over will work.

```
/** The hour of the day, in 0..23.*/
private int hr;

/** temps[0..numRecorded-1] are the recorded temperatures */
private double[] temps;
/** number of temperatures recorded */
private int numRecorded;
```

If you are not interested in using the javadoc facility in an IDE, since the fields are private, javadoc comments are not necessary. Therefore, you can write the class invariant in a readable style as one-line comments on the same line as the declaration:

```
private int hr; // The hour of the day, in 0..23.
```

```
private double[] temps; // temps[0..numRecorded-1] are the recorded temperatures
private int numRecorded; // number of temperatures recorded
```

The purpose of a constructor is to initialize all fields so that the class invariant is true. Then, when writing a method, you can assume that the class invariant is true when the method is called and write the method body so that it terminates with the class invariant true. Looking often at the class invariant can help prevent mistakes caused by forgetting just what a field means or what its constraints are.

3.4 Method specifications

Every method should be preceded by a blank line and then a Javadoc specification that describes what the method does, along with preconditions ---constraints on the arguments of a call. In order to do this properly, the specification must mention each parameter, indicating what it is for.

Java has certain conventions for describing the parameters and result of a method. For example, one can write this within a Javadoc specification to describe parameter b and the result

```
@param b one of the sides of the triangle
@return The area of the triangle
```

One can certainly use these. However, we have found that a specification can be written more compactly and clearly without using those, and we do not require them. The main guideline is to write the specification as a command to do something, naming all parameters in that command. Examples are given below.

Procedure specifications are generally best written as commands to do something, e.g.

```
/** Print the sum of a and b. */
public static void printSum(int a, int b) { ... }
```

Function specifications are generally written to indicate what to return, e.g.:

```
/** Return area of triangle whose side lengths are a, b, and c. */
public static double area(double a, double b, double c) { ... }
```

or

```
/** = area of triangle whose side lengths are a, b, and c. */
public static double area(double a, double b, double c) { ... }
```

The purpose of a constructor is to initialize all fields so that the class invariant is true. We favor writing a constructor specification as illustrated by the following:

```
/** Constructor: an instance with hour-of-day h and minute-of-the-hour m.
   * Precondition: h in 0..23 and m in 0..59. */
public Time(int h, int m) { ... }
```

3.4.1. Preconditions and postconditions

A precondition is an assertion --a true-false statement-- about the parameters of a method that must be satisfied if the method is to do its job properly. It is *not* the responsibility of the method to check that the precondition is true. It *is* the responsibiity of the programmer who writes a call on the method to ensure that the arguments of the call satisfy the precondition. The following example shows how we write preconditions:

```
/** Return the index of the rightmost occurrence of x in b.
    Precondition: b is sorted (in ascending order) and x is in b */
public static int binarySearch(int x, int[] b)
```

A call binarySearch(25, c) will work properly only if 25 is in array c and c is sorted. If not, the method can do *anything*.

If it doesn't cost too much, the method can be made more robust by checking that the precondition is true (using a Java assert statement), but it is not necessary.

Similarly, a postcondition is an assertion that will be true upon termination of the method.

3.5. Class specifications

Each public class is given in a separate file. The beginning of the file should contain a javadoc comment that explains what the class is for. This can be a simple, short summary. Often, one also puts here information concerning the author, the date of last modification, and so on. Here is an example.

```
/** An object of class Auto represents a car.
    Author: John Doe.
    Date of last modification: 25 August 1998 */
public class Auto { ... }
```

Inner classes and non-public classes that are defined in the same file along with a public class should be specified in a similar fashion.

3.6. Statement-comments

Just as the sentences of an essay are grouped in paragraphs, so the sequence of statements of the body of a method should be grouped into logical units. Often, the clarity of the program is enhanced by preceding a logical unit by a comment that explains what it does. This comment serves as the specification for the logical unit; it should say precisely what the logical unit does.

The comment for such a logical unit is called a *statement-comment*. It should be written as a command to do something. Here is an example.

```
// Truthify x >= y by swapping x and y if needed.
if (x < y) {
    int tmp= x;
    x= y;
    y= tmp;
}
```

A statement-comment should explain what the group of statements does, not how it does it. Thus, it serves the same purpose as the specification of a method: it allows one to skip the reading of the statements of the logical unit and just read the comment. With suitable statement-comments in the body of a method, one can read the method at several "levels of abstraction", which helps one scan a program quickly to find a section of current interest, much like on scans section and subsection headings in an article or book. But this purpose is served only if statement-comments are precise.

Statement comments must be complete. The comment

```
// Test for valid input
```

is not adequate. What happens if the input is valid? What if it isn't --is an error message written or is some flag set? Without this information, one must read the statements for which this statement-comment is a specification, and the whole purpose of the statement-comment is lost.

Use of blank lines. Place a blank line after the implementation of each statement-comment. In the following example,

```
// Eliminate whitespace from the beginning of t.
while (t.length() != 0 && isWhitespace(t.charAt(0))) {
    t= t.substring(1);
}

// If t is empty, print an error message and return.
if (t.length() == 0) {
    ...
    return false;
}

if (containsCapitals(t)) {
      ...
}

// Store the French translation of t in tFrench.
...
```

This sequence consists of 4 statements: (1) eliminate the whitespace ..., (2) print a message and return false if t is empty, (3) do something if t contains capitals, and (4) store the French translation. Three of these are statement-comments, and one can easily see, because of the blank lines, where their implementations end.

3.7 Assertions and the assert statement

An assertion is a true-false statement that is placed (as a comment) at some point in a method body to indicate what is true at that point. In order to distinguish it from a statement-comment, it is suggested to place it in braces { and } as shown in the following example --historically, that was how assertions were written. However, few people will take the trouble to put in the braces.

```
// { b[0..t-1] is sorted }
```

Place assertions wherever the code below depends on that assertion being true and the assertion helps the reader understand. One particular use of it can be in a function body that returns in several places. The following illustrates this:

```
/** Return the middle value of b, c, d */
public static int middle(int b, int c, int d) {
    if (b > c ) {
        int temp= b; b= c; c= temp;
    }
    // { b <= c }
    if (b <= d) {
       // { middle value is smaller of c and d }
       return Math.min(c, d);
    }
    // { d < b <= c }
    return b;
}
```

In place of the assertion as a comment, one could also use the Java assert statement:

```
assert <boolean expression> ;
```

If the <boolean expression> is true, nothing is done, but if it is false, execution throws an exception and, generally, the program terminates.

Using the assert statement instead of an assertion-comment has the advantage of catching errors that cause the assertion to be false. Some people advocate leaving assertions in a finished program, for just that reason, unless the boolean expression is so complicated that it ends up slowing down the program execution.

3.8 Loop invariants

A loop invariant is an assertion that is true before and after each iteration of a loop and is used to prove correctness of the loop (i.e. prove that execution of the loop achieves the desired postcondition). We do not explain loop invariants here. Our purpose is only to show where they are placed in the program.

A loop invariant is placed as a comment just before the loop. Below, we show a loop invariant in the same code written using a while loop and written using a for-loop.

```
// Sort b[0..]
k= 0;
// invariant: b[0..k-1] is sorted and b[0..k-1] <= b[k..]
while (k < b.length) {
    Swap b[k] with smallest of b[k..];
    k= k+1;
}


// Sort b[0..]
// invariant: b[0..k-1] is sorted and b[0..k-1] <= b[k..]
for (int k= 0; k < b.length; k= k+1) {
    Swap b[k] with smallest of b[k..];
}
```

4. Code organization

4.1 Field and class variable declarations go at the beginning of a class

Place all field and class (static) variables at the beginning of a class, before all the methods. That is where a reader will look for them.

4.2 Keep methods short

As a general rule of thumb, if a single method is getting beyond 20-50 lines, consider *lifting* some of its functionality into other methods. Rarely should a method be over 50 lines.

Similarly, if you find yourself pasting the exact same chunk of code into several places, consider creating a single method that does the job. The *Refactor* contextual menu in Eclipse can help with this.

It is possible to go overboard with this. Even if two blocks of code are structurally similar, don't merge them into the same method if they do different things. Each method should have only one purpose. Names like saveOrLoadAnimal() are a clue that you really need two methods: saveAnimal() and loadAnimal() .

4.3 Use statement-comments to organize a method

A long method body should generally be broken into a sequence of logical units, each performing some subtask. Each of these logical units needs a comment that describes *what* the unit does. The section on statement-

comments discusses this.

Often, the process of *step-wise refinement* of a program leads naturally to these logical units. For example, one might look at the specification of a method and decide that it can be implemented in three steps, each written in English:

```
Do b;
Do c;
Do d
```

They would become statement-comments, with their implementations underneath:

```
// Do b
implementation of "Do b"

// Do c
implementation of "Do c"

// Do d
implementation of "Do d"
```

4.4 Use returns to simplify method structure

You may hear from some people that a method should return only in the last statement of the method, at the end of the method body. In some cases, that can lead to a messy, incomprehensible method body. Use of return statements in several places, along with appropriate use of assertions, can lead to a far more comprehensible method. Further, this use of returns is developed naturally during stepwise refinement. However, one must be disciplined with the use of the returns and assertions.

As an example, we present a method that calculates the score for a bowling frame given the two objects for the next two frames. (Understanding this requires knowledge of bowling!) Note the use of assertions. We challenge you to write this method more simply, using only one return statement, or by using nested if-else statements, each part of which ends with a return statement.

```
/** = the score for this frame, given the two following frames f1 and f2.
    Pre: If this is the ninth frame, f2 is null.
         If this is the tenth frame, f1 and f2 are null*/
public int score(Frame f1, Frame f2) {
    if (isTenth) {
        return firstBall + secondBall + thirdBall;
    }

    // { it's not the tenth frame }
    // Return value if it is not a strike or spare
    if (firstBall + secondBall < 10) {
        return firstBall + secondBall;
    }

    // Return the value for a spare --in other than 10th frame
    if (firstBall < 10 && firstBall + secondBall == 10) {
        return firstBall + secondBall + f1.firstBall;
    }

    // { it's a strike and it's not the tenth frame }
    // Return a value for a strike in the ninth frame
    if (f2 == null) {
        return firstBall + f1.firstBall + f1.secondBall;
    }
```

```
        // Return a value for a strike in frame in the range 1..8
        if (f1.firstBall < 10) {
            return 10 + f1.firstBall + f1.secondBall;
        }
        // { The next frame is a strike }
        return firstBall + f1.firstBall + f2.firstBall;
    }
```

4.5 Put the shorter of the then-part and else-part first

One sometimes sees a segment that looks like this:

```
if (condition) {
   ...
   30 lines of code
    ...
} else single-statement
```

The long distance of the if-line from the else-part makes it difficult for the reader. Better is to structure it like this:

```
if (!condition) {
    single-statement
} else {
   ...
   30 lines of code
    ...
}
```

Further, if the single-statement is a return statement, simplify the structure by writing it like this:

```
if (!condition) {
    return ...;
// { ! condition }
...
30 lines of code
...
```

4.6 Declare local variables close to first use

A local variable is a variable declared within a method body.

The tendency is to declare all variables at the beginning of the method body, and to start off inserting there all the variables you think you might need. **Fight this tendency!** It leads to a proliferation of variables that often are not needed, and it forces the reader to look at and think about variables at the wrong time.

Principle: Declare a variable as close to its first use as possible.

Look at the small example below. Variable temp has been declared first. This causes the reader to ask, "What is temp for? Well, I don't know." The reader goes on and sees that there are some statements to swap b and c under certain circumstances and skips them because the reader knows how to write a swap. Well, the reader still doesn't know what temp is for!

```
/** Return middle value of b, c, d */
public static int middle(int b, int c, int d) {
    int temp;
```

```
        // Swap b, c to put smaller in b
        if (b > c) {
          temp= b;
          b= c;
          c= temp;
        }

        // { b <= c }
        if (d <= b) {
            return b;
        }
        // { b < d and b <= c }
        return Math.min(c, d);
    }
```

In the code below, temp is declared where it belongs, and the reader who doesn't want to read the swap implementation doesn't even have to know that there is a local variable temp.

This is a *small* example. In a longer method, with loops and such, the use of the principle to place local variables as close to their first use as possible becomes more important.

```
    /** Return middle value of b, c, d */
    public static int middle(int b, int c, int d) {
        // Swap b, c to put smaller in b
        if (b > c) {
            int temp= b;
            b= c;
            c= temp;
        }

        // { b <= c }
        if (d <= b) {
            return b;
        }
        // { b < d and b <= c }
        return Math.min(c, d);
    }
```

5. Public/private access modifiers

Java has four levels of access: public, private, protected, and package (the default; what you get when no modifier is specified). Here is how we generally use them:

public: public items can be used anywhere.

Methods are public if they are part of the outward "behavior" of the class, i.e. if the user should be able to use them. Otherwise, make them private.

Constants, i.e. static final fields, are usually public so that anyone can use them.

private: private items can be referred to only in code within the class.

Fields and class (i.e. static) variables should should be private unless there is a specific, good reason for them to be otherwise.

A method should be made private if the user should not be able to call it from outside the class.

protected: Protected components can be referenced within the class, within subclasses, and within any class defined in the same package.

package (the default, i.e. what one gets when no modifer is used): package components can be referenced within the class and within any class that is in the same package.

Modifier	Class	Package	Subclass	World
public	yes	yes	yes	yes
protected	yes	yes	yes	no
package	yes	yes	no	no
private	yes	no	no	no

Use of access modifiers

6. Creators/observers/mutators

Problems with the getter/setter terminology

Here is conventional terminology: a *getter method* is a function that returns the value of a field (and changes nothing); a *setter method* is a procedure that stores a value in a field.

This terminology has problems, and it is better to use an alternative terminology: creators, observers, and mutators. This pdf file explains it all in one page.

Immutable classes (or objects)

Classes that contain no mutator methods are called immutable (or rather their objects are immutable). Immutable means not capable of change, invariable, unalterable. Classes *String* and *Integer* are immutable. You cannot change the value of the string in a String object ---but you can create new String objects. The advantage of immutable classes is that their objects can be shared freely by different code modules.

7. Side effects

A side effect occurs when evaluation of an expression changes a value. For example, a side effect occurs if evaluation of the following loop condition, which is a call on function f, changes some variable:

```
while (f(...)) {...}
```

Generally, side effects should be avoided. This means, for example, that an assignment statement like

```
b= ...;
```

should change no variable other than b. A side-effect that changes something else could be disastrous, unless it is very well documented and the programmer is aware of what is happening. In fact, the theory of proving programs correct tells us that formal proofs of correctness when side effects may occur are far far more complex than when they may not.

Standard paradigms

One does see some standard paradigms that use side effects. For example, below is a loop that reads and prints lines of a *BufferedReader br*; the loop condition has the side effect of assigning to variable *line*.

```
String line;
while ((line= br.readLine()) != null) {
   System.out.println("line is " + line);
}
```

However, we would never write the loop like this, preferring to write it as follows, in a way that has no side effects, even though it is longer that the one above.

```
String line= br.readLine();
// invariant: all lines read in, except line, have been printed
while (line != null) {
   System.out.println("line is " + line);
   line= br.readLine();
}
```

Benevolent side effects

There is one class of side effect that is OK. A benevolent side effect is one that changes the state but not in a way that the user can observe. A benevolent side effect can be used to make a program more efficient. Here is an example.

Consider implementing a class that mantains a set of questions-and-answers. When someone asks a question, the corresponding answer is given. The questions-and-answers are implemented in an array, and one has to search the list in linear fashion to find a question. It is better if more frequently asked questions are at the front of the list, so they are more efficiently found. This can be achieved by the following: Whenever a question is asked, move it up one position in the array. This changes the representation of the set of questions-and-answers, but not the set itself; the user cannot notice the change. This is a benevolent side effect.

8. Code to interfaces instead of implementations

In most cases, the client code doesn't care about the implementation of the list ---it just wants to know what operations are supported. Specifying a linked list is needlessly providing a detail that makes future changes harder.

```
List<String> users= new LinkedList<String>(); // Good
LinkedList<String> users= new LinkedList<String>(); // Bad
```

9. Avoid deprecated patterns

Java has changed over the years, and as changes have been made, some methods and classes have been deprecated. Deprecated literally means "disapproved of", but a more accurate translation would be "retired". A deprecated method (or class) is still usable, but it is best not to use it. It will gradually be phased out. There is a new method to do the same thing.

So if you see that something is deprecated and you have the time, look for the better alternative and use it instead.
