Meaningful Names

All identifiers must have meaningful, human-readable, English names.

```
void display();
int countStudents();
Date dateStudentRegistered;
```

Exception: variables with very limited scope (<20 lines) or variables in loops may be shortened if the purpose of the variable is clear.

```
void swapCars(Person person1, Person person2) {
    Car tmp = person1.getCar();
    person1.setCar(person2.getCar());
    person2.setCar(tmp);
}
```

Naming Conventions

Constants must be all upper case, with multiple words separated by '_':

```
final int DAYS_PER_WEEK = 7;
```

Functions must use camelCase. Functions should be named in terms of an action:

```
double calculateTax();
```

Class names must start with an uppercase letter, and use CamelCase (PascalCase). Classes should be named in terms of a singular noun:

```
class VeryLargeCar;
```

Constant should have the most restrictive scope possible. For example, if it is used in only one class, then define the constant as private to that class. If a constant is needed in multiple classes, make it public.

Boolean variables should be named so that they make sense in an if statement:

```
if (isOpen) {
    ...
}
while (!isEndOfFile && hasMoreData()) {
    ...
}
```

Use named constants instead of literal numbers (magic numbers). It is often acceptable to use 0 and 1; however, it must be clear what they mean:

```
// OK:
int i = 0;
i = i + 1;

// Bad: What are 0 and 1 for?!?
someFunction(x, 0, 1);
```

Braces {...}

Opening brace is at the end of the enclosing statement; closing brace is on its own line, lined up with the start of the opening enclosing statement. Statements inside the block are indented one tab.

```
if (y > 500) {
→   ...
} else if (y == 0) {
→   ...
} else {
→   ...
}
if (someBigBooleanExpression
→   && !someOtherExpression) {
→   ...
}
```

Statements and Spacing

Declare each variable in its own definition, rather than together (`int i, j`).

```
int *p1;
int p2;
```

All binary (2 argument) operators (arithmetic, bitwise and assignment) and ternary conditionals (?:) must be surrounded by one space. Commas must have one space after them and none before. Unary operators (!, *, &, - (ex: -1), + (ex: +1), ++, --) have no additional space on either side of the operator.

```
i = 2 + (j * 2) + -1 + k++;
if (i == 0 || j < 0 || !k) {
    arr[i] = i;
}
myObj.someFunction(i, j + 1);
```

Add extra brackets in complex expressions, even if operator precedence will do what you want. The extra brackets increase readability and reduce errors.

```
if ((!isReady && isBooting)
    || (x > 10)
    || (y == 0 && z < (x + 1)))
{
    ...
}
```

However, it is often better to simplify complex expressions by breaking them into multiple sub-expressions that are easier to understand and maintain:

```
boolean isFinishedBooting = (isReady || !isBooting);
boolean hasTimedOut = (x > 10);
boolean isOldFirmware = (y == 0 && z < (x + 1));
if (!isFinishedBooting
    || hasTimedOut
    || isOldFirmware)
{
    ...
}
```

Classes

Inside a class, the constants must be on top of the fields, which must be at the top of the class, followed by the methods.

```
class Pizza {

    public static int MAX_NUMBER = 1231;

    private int toppingCount;

    public Pizza() {
        toppingCount = 0;
    }
    public int getToppingCount() {
        return toppingCount;
    }
    ...
}

class Topping {
    private String name;
    ...
    public String getName() {
        return name;
    }
}
```

Comments

Comments which are many lines long should use `/* ... */`.

Each class must have a descriptive comment before it describing the general purpose of the class. These comments should be in the JavaDoc format. Recommended format is shown below:

```
/**
 * Student class models the information about a
 * university student. Data includes student number,
 * name, and address. It supports reading in from a
 * file, and writing out to a file.
 */
class Student {
    ...
}
```

Other

Either post-increment or pre-increment may be used on its own:

```
i++;
++j;
```

All switch statements should include a `default` label. If the `default` case seems impossible, place an `assert false;` in it. Comment any intentional fall-throughs in `switch` statements:

```
switch(buttonChoice) {
case YES:
    // Fall through
case OK:
    System.out.println("It's all good.");
    break;
case CANCEL:
    System.out.println("It's over!");
    break;
default:
    assert false;
}
```

Use plenty of assertions. Any time you can use an assertion to check that some condition is true which "must" be true, you can catch a bug early in development. It is especially useful to verify function pre-conditions for input arguments or the object's state. Note that you must give the JVM the `-ea` argument (enable assertions) for it to correctly give an error message when an assertion fails.

Never let an assert have a side effect such as `assert i++ > 1;`. This may do what you expect during debugging, but when you build a release version, the asserts are removed. Therefore, the i++ won't happen in the release build.