A lot of what we define to be important for efficiency boils down to “don’t do anything unnecessary”. Knowing what’s necessary and what isn’t can be difficult, but following the way the spec phrases certain implementation details will lead you in the right direction. Unfortunately, there’s no hard and fast rule for finding every inefficiency. To find them, you will have to dedicate some time to reason about your code.

As a general rule, reduce or remove complicated expressions and logic where reasonably possible.

Creating new objects

Avoid making objects that you don’t need. This can be tricky to spot, but in particular you should review your code statements that instantiate `new` objects or call other methods that do so. Making sure that every object you instantiate is necessary (no other available methods can provide the same or reasonable behavior) and logical will give some sense of security here.

The first example makes a `String` just to discard it in the next line. When we set `words = anotherString.substring(1)` in the second line, the previous value of `words` is lost (unless it’s saved elsewhere). The second example doesn’t throw away any newly made objects.

Bad

```
String words = new String();
words = anotherString.substring(1);
```

Good

```
String words = anotherString.substring(1);
```

Recomputing values

Avoid recomputing complex expressions and method calls. If you have to use the value of a method call or a complicated expression more than once, you should store it in a

well-named variable. Using the value stored in this variable will give you instant access instead of recomputing the value by evaluating the method call or expression again. This will also clear up the clutter of your code by replacing generic symbols and expressions with a name to describe some context.

There are some methods that are fast enough that storing the value in a variable for the future doesn't save us anything. For example, the `size()` method for any data structure we use in this course will have the same instant access whether we store it or call the method. In such a case, efficiency is not an issue we have to worry about.

Factoring

If you have lines of repeated or very similar code that need to be executed in different places, group the code of the task into a private helper method. Putting the code into one place instead of several will make making changes easier, as well as make our code more readable.

Code inside if/else structure should represent different cases where the code is inherently different. This means duplicate lines of code or logic in an if/else structure should be factored out to come before or after, so that they are only written to happen once, unconditionally.

Bad

> Note that there are repeated lines of logic that actually always happen, instead of conditionally like how our structure is set up. We can factor these out to simplify and clean our code.

```
if (x % 2 == 0) {
    System.out.println("Hello!");
    System.out.println("I love even numbers too.");
    System.out.println("See you later!");
} else {
    System.out.println("Hello!");
    System.out.println("I don't like even numbers either.");
    System.out.println("See you later!");
}
```

Good

```
System.out.println("Hello!");
if (x % 2 == 0) {
    System.out.println("I love even numbers too.");
}  else {
    System.out.println("I don't like even numbers either.");
}
System.out.println("See you later!");
```

Tip

Review any if/else structures you write and make sure there are no duplicate lines of code or logic at the beginning or the end.

Redundant conditional logic

This section details issues with writing unnecessary if/else logic that can be omitted. We will refer to any control flow like if, else, etc. as conditional logic.

Loops

Good bounds and loop conditions will already deal with certain edge case behavior. Avoid writing additional conditional logic that loop bounds already generalize.

Bad

The `if` here is redundant. The loop won't produce any output (or even run) if n were 0 or negative in the first place, so we should remove the check.

```
if (n > 0) {
    for (int i = 0; i < n; i++) {
        System.out.println("I love loops < 3");
    }
}
```

Already known values

By using control flow (if/else/return) properly, you can guarantee some implicit facts about values and state without explicitly checking for them. The examples listed below add in redundant conditional logic that should be removed.

Bad

Because `someTest` was tested for alone in the first if, future branches of the if/else structure know `someTest` must be false, so the check for `!someTest` is redundant and can be omitted.

```
if (someTest) {
    ...
} else if (!someTest ...) {
    ...
}
```

Bad

At the second `if` statment, we know that `someTest` must have been false, so it's not necessary to check its value again.

```
if (someTest) {
    throw exception/return
}
if (!someTest) {
    ...
}
```

Boolean zen

Boolean zen is all about using boolean values efficiently and concisely.

The following code shows a tempting way to write an if statement based on boolean value.

Bad

```
if (test == true) {
    // do some work
}
```

Note that `test` itself is a `boolean` value. When it is `true`, we are asking whether `true == true`. `true == true` will evaluate to `true`, but remember that the `if` branch will execute as long as what is in its parentheses evaluates to true. So we can actually use `test` directly:

Good

```
if (test) {
    // do some work
}
```

Note

> To check for the opposite of `test`, don't check for `test == false`. Instead, use `!test` to check that the opposite of `test` evaluates to `true`.

Here's an example that uses what we learned in the previous section about simplifying boolean expressions.

Bad

```
if (test) {
    return true;
} else {
    return false;
}
```

There is actually a much more concise way to do this. If we want to return true when test is true and false when test is false, then we actually just want to return the value of `test`.

Good

```
return test;
```

In general, make your use of booleans to simplify conditional and boolean expressions. This is something to look out for whenever you have conditions or boolean values.

Bad

> Because this code always wants to do one or the other, (and doesn't involve a return or exception) we want to express this code more simply as an if/else.

```
if (someTest) {
    System.out.println("hello!");
}
if (!someTest) {
    System.out.println("I'm redundant");
}
```

Bad

Note that the behavior inside this if block is exactly the same behavior as in the other if block. Instead of rewriting the same code twice, we can combine the two if conditions with `||` and just write the behavior once.

```
if (max < result) {
    return max;
}
if (max == 0) {
    return max;
}
```

Bad

It doesn't matter if you think of conditions/cases or their negated versions, but after revising your code don't include empty condition blocks with no line of code inside. Instead, just flip the condition to have if (max >= 0), and no else.

```
if (max < 0) {
    // do nothing
} else {
    ...
}
```

Loop Zen

Loop Bounds

When writing loops, choose loop bounds or loop conditions that help generalize code the best. For example, the code before this `for` loop is unnecessary.

Bad

```
System.out.print("\*");
for (int i = 0; i < 4; i++) {
    System.out.print("\*");
}
```

Instead, why not just run the loop one extra time? This way we avoid writing duplicate behavior.

Good

```
for (int i = 0; i < 5; i++) {
    System.out.print("\*");
}
```

Only Repeated Tasks

If you have something that only happens once (maybe at the end of a bunch of repeated tasks), then don't put code for it inside of your loop.

Bad

```
for (int i = 0; i < 4; i++) {
    if (i != 3) {
        System.out.println("working hard!");
    } else {
        System.out.println("hardly working!");
    }
}
```

Good

```
for (int i = 0; i < 3; i++) {
    System.out.println("working hard!");
}
System.out.println("hardly working!");
```

Similarly, a loop that always runs one time is also a bad use of a loop. When reviewing the logic of your code, it might be helpful to check if a loop can be reduced to an if or just deleting the loop entirely.