http://www.yegor256.com/2015/04/02/class-casting-is-anti-pattern.html

Class Casting Is a Discriminating Anti-Pattern

2 April 2015 Yegor Bugayenko

Type casting is a very useful technique when there is no time or desire to think and design objects properly. Type casting (or class casting) helps us work with provided objects differently, based on the class they belong to or the interface they implement. Class casting helps us **discriminate** against the poor objects and **segregate** them by their race, gender, and religion. Can this be a good practice?

© Гадкий утенок (1956) by Владимир Дегтярёв

This is a very typical example of type casting (Google Guava is full of it, for example `Iterables.size()`:

```
public final class Foo {
```

```java
  public int sizeOf(Iterable items) {
    int size = 0;
    if (items instanceof Collection) {
      size = Collection.class.cast(items).size();
    } else {
      for (Object item : items) {
        ++size;
      }
    }
    return size;
  }
}
```

This `sizeOf()` method calculates the size of an iterable. However, it is smart enough to understand that if `items` are also instances of `Collection`, there is no need to actually iterate them. It would be much faster to cast them to `Collection` and then call method `size()`. Looks logical, but what's wrong with this approach? I see two practical problems.

First, there is a **hidden coupling** of `sizeOf()` and `Collection`. This coupling is not visible to the clients of `sizeOf()`. They don't know that method `sizeOf()` relies on interface `Collection`. If tomorrow we decide to change it, `sizeOf()` won't work. And we'll be very surprised, since its signature says nothing about this dependency. This won't happen with `Collection`, obviously, since it is part of the Java SDK, but with custom classes, this may and will happen.

The second problem is an inevitably **growing complexity** of method `sizeOf()`. The more special types it has to treat differently, the more complex it will become. This if/then forking is inevitable, since it has to check all possible types and give them special treatment. Such complexity is a result of a violation of the single responsibility principle. The method is not only calculating the size of `Iterable` but is also performing type casting and forking based on that casting.

What is the alternative? There are a few, but the most obvious is method overloading (not available in semi-OOP languages like Ruby or PHP):

```
public final class Foo {
  public int sizeOf(Iterable items) {
    int size = 0;
    for (Object item : items) {
      ++size;
    }
    return size;
  }
  public int sizeOf(Collection items) {
    return items.size();
  }
}
```

Isn't that more elegant?

Philosophically speaking, type casting is discrimination against the object that comes into the method. The object complies with the contract provided by the method signature. It implements the `Iterable` interface, which is a contract, and it expects equal treatment with all other objects that come into the same method. But the method discriminates objects by their types. The method is basically asking the object about its ... race. Black objects go right while white objects go left. That's what this `instanceof` is doing, and that's what discrimination is all about.

By using `instanceof` , the method is segregating incoming objects by the certain group they belong to. In this case, there are two groups: collections and everybody else. If you are a collection, you get special treatment. Even though you abide by the `Iterable` contract, we still treat some objects specially because they belong to an "elite" group called `Collection` .

You may say that `Collection` is just another contract that an object may comply with. That's true, but in this case, there should be another door through which those who work by that contract should enter. You announced that `sizeOf()` accepts everybody who works on the

`Iterable` contract. I am an object, and I do what the contract says. I enter the method and expect equal treatment with everybody else who comes into the same method. But, apparently, once inside the method, I realize that some objects have some special privileges. Isn't that discrimination?

To conclude, I would consider `instanceof` and class casting to be anti-patterns and code smells. Once you see a need to use them, start thinking about refactoring.