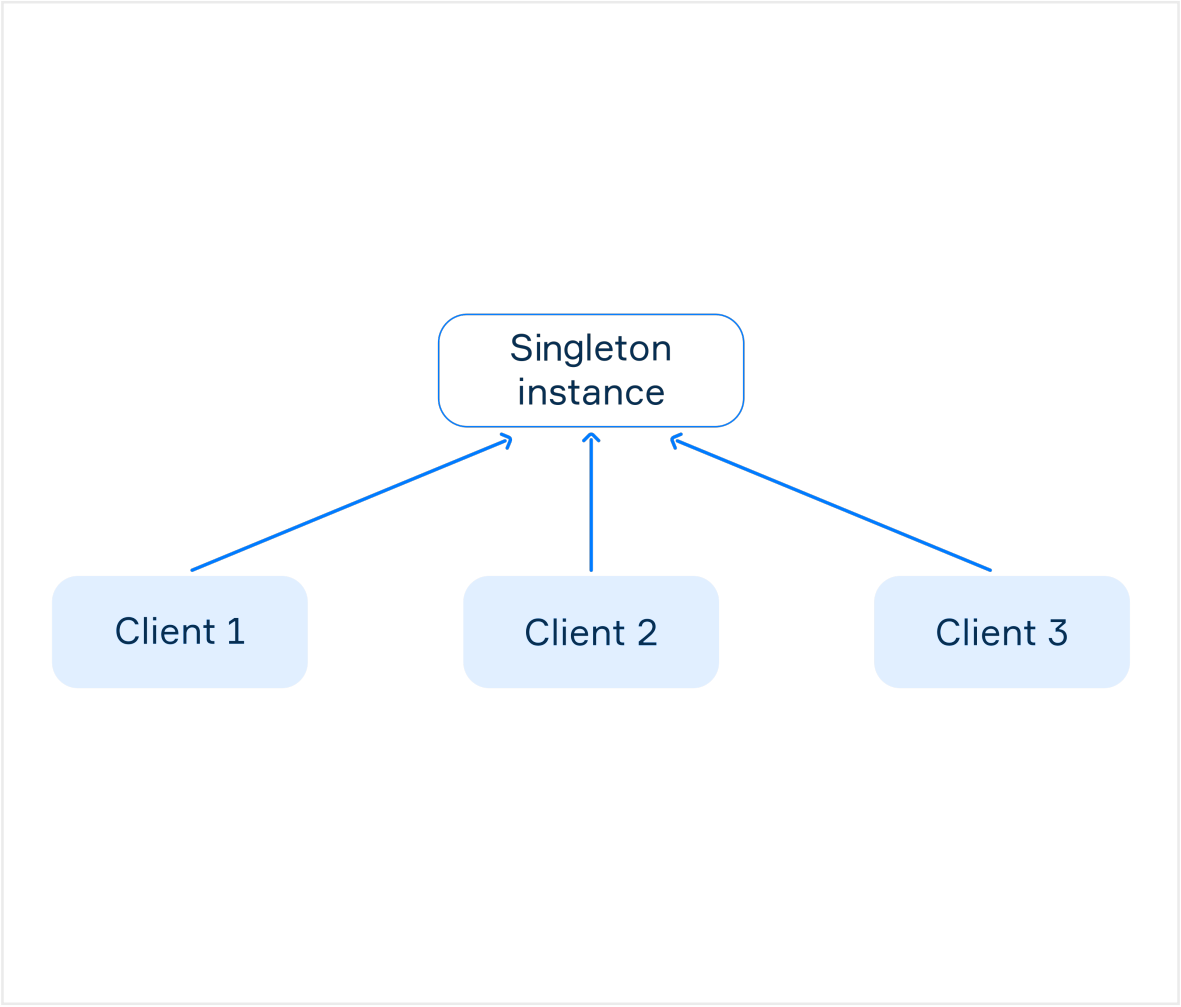
Singleton

Sometimes we need to have only a single instance of a class shared across an entire application.

There is a special design pattern called **Singleton** that restricts the instantiation of a class to one object.

The pattern ensures that there is only one instance of the class and provides global access to it for the outer world.



The same instance is used by different clients (classes, methods)

Singletons often control access to resources such as database connections or sockets.

For example, a class that keeps the connection to the database can be written as a singleton: creating a connection object time after time will be quite bulky in terms of memory. If it is a singleton, then your application will work better and faster.

Another example is a class that provides a universal logger for an application. It keeps the shared log-file as the resource.

Basic singleton in Java

The standard way to write a class according to the singleton pattern includes:

- a private constructor to defeat the creation of instances using the keyword new;
- a private static variable of the class that is the only instance of the class;
- a public static method to return the same instance of the class; this is a global access point to the instance.

The following code implements these concepts:

```
class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() { }

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

The class Singleton is a basic implementation of the considered pattern. The field INSTANCE stores only a single instance of the class. The constructor is declared as private to defeat instantiation. The static method getInstance returns the same instance of the class.

Let's create three variables, assign an instance to each of them and then compare the variables by references:

```
Singleton s1 = Singleton.getInstance();
Singleton s2 = Singleton.getInstance();
Singleton s3 = Singleton.getInstance();

System.out.println(s1 == s2); // true because s1 and s2 refer the same object
System.out.println(s2 == s3); // true because s2 and s3 refer the same object
```

As you can see, the variables s1s2, and s3 refer the same object that is stored in the static field of Singleton.

Usually, singletons have additional instance fields to store values (global variables) to share them across your application as well as methods to have a behavior.

Lazy initialization

he singleton above loads the instance when the class is loaded. But sometimes the initialization of a singleton can take much time: for example, to load the values of the fields from a file or a database.
The following implementation loads the instance only if it is needed by a client (who calls getInstance):

```
class Singleton {

    private static Singleton instance;

    private Singleton () { }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Even though the implementation of a singleton pattern contains few lines of code, it has an important feature: lazy initialization. It means that the singleton instance is not created until the getInstance method is called for the first time. This technique ensures that singleton instances are created only when needed.

Note that you should use the provided singleton class only in one-thread environments because it is prone to multithreading issues.

Singleton pattern in Java libraries

There are a lot of Singleton examples in the Java Class Library:

- java.lang.Runtime.getRuntime()
- java.awt.Desktop.getDesktop()
- java.lang.System.getSecurityManager()

To determine a singleton, see a static creational method which always returns the same instance of a class.
Popular frameworks for developing enterprise applications (such as Spring, Java EE) also use singletons.

Criticism

Some criticize the singleton pattern and consider it to be an anti-pattern: it is frequently used where it is not beneficial, it imposes unnecessary restrictions in some situations and introduces global state into an application. Hence the wisdom: use the pattern wisely when you really need a singleton.