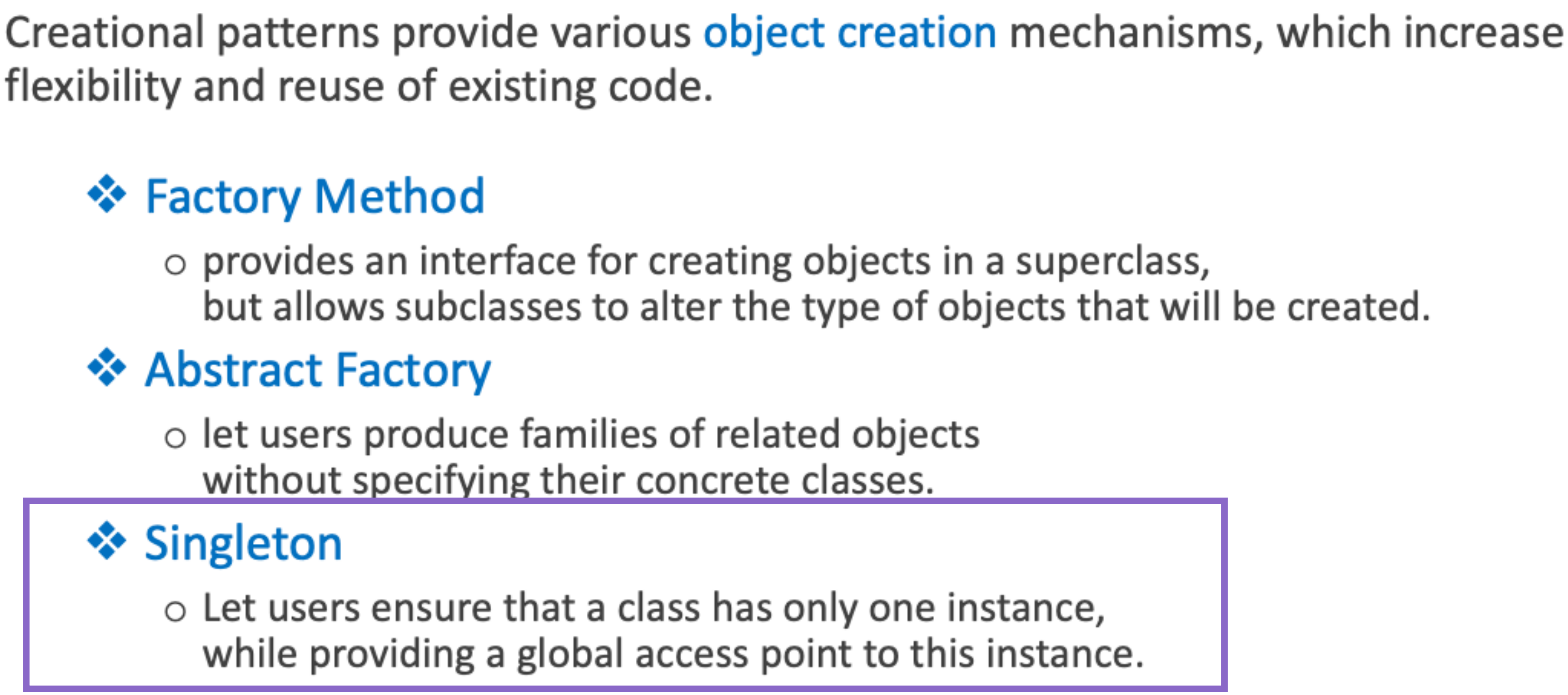
Singleton Pattern and Asynchronous Design

COMP2511, CSE, UNSW

Creational Pattern: Singleton Pattern



Creational patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

- **Factory Method**
  - provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
- **Abstract Factory**
  - let users produce families of related objects without specifying their concrete classes.
- **Singleton**
  - Let users ensure that a class has only one instance, while providing a global access point to this instance.

Singleton Pattern

Intent: Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

Problem: A client wants to,

- ensure that a class has just a single instance, and
- provide a global access point to that instance

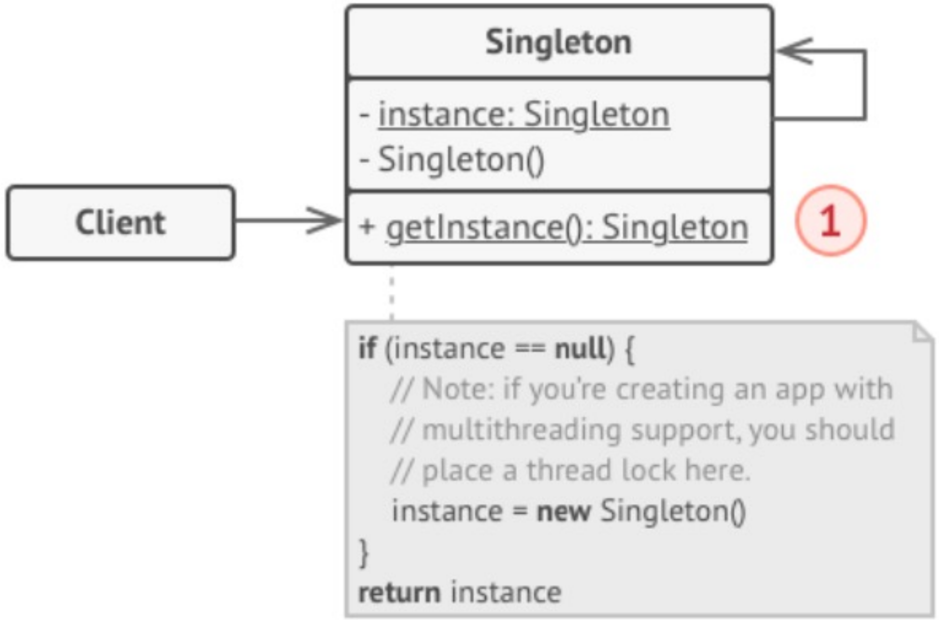
Solution:

All implementations of the Singleton have these two steps in common:

- Make the default constructor **private**, to prevent other objects from using the new operator with the Singleton class.
- Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.
- If your code has access to the Singleton class, then it's able to call the Singleton's static method.
- Whenever Singleton's static method is called, the same object is always returned.

Singleton: Structure

- The Singleton class declares the static method ***getInstance*** (1) that returns the same instance of its own class.
- The Singleton's constructor should be hidden from the client code.
- Calling the ***getInstance*** (1) method should be the only way of getting the Singleton object.



```
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```

Singleton: How to Implement

- Add a private static field to the class for storing the singleton instance.
- Declare a public static creation method for getting the singleton instance.
- Implement "lazy initialization" inside the static method.
  - It should create a new object on its first call and put it into the static field.
  - The method should always return that instance on all subsequent calls.
- Make the constructor of the class private.
  - The static method of the class will still be able to call the constructor, but not the other objects.
- In a client, call singleton's static creation method to access the object.

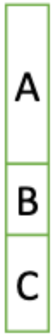
For more information, read:

https://refactoring.guru/design-patterns/singleton/java/example

Synchronous vs Asynchronous Software Design

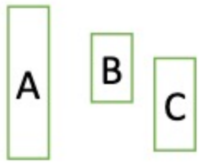
What is Synchronous programming?

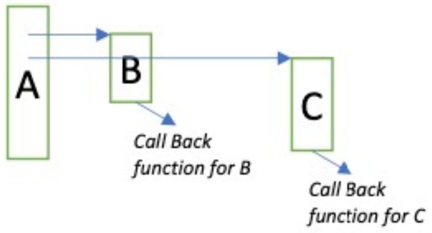
- In *synchronous* programming, operations are carried out in order.
- The execution of an operation is dependent upon the completion of the preceding operation.
- Tasks (functions) A, B, and C are executed in a sequence, often using one thread.



What is Asynchronous programming?

- In *asynchronous programming*, operations are carried out independently.
- The execution of an operation is not dependent upon the completion of the preceding operation.
- Tasks (functions) A, B, and C are executed independently, can use multiple threads/resources.

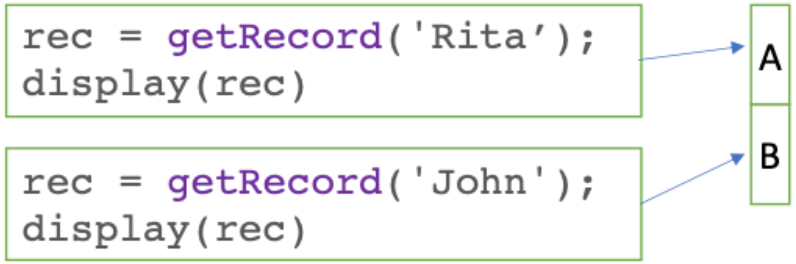




Example: Synchronous vs Asynchronous programming

Synchronous

```
function getRecord(key) {
    establish database connection
    retrieve the record for key
   return record;
}

function display(rec){
   display rec on the web page
}
```

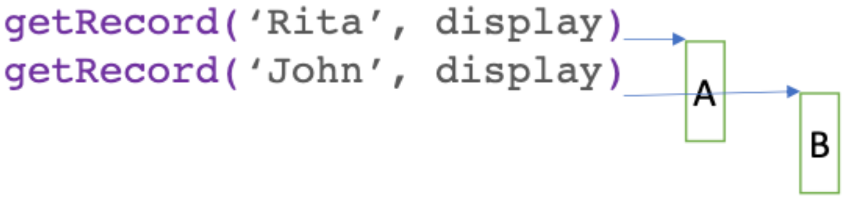


```
rec = getRecord('Rita’);
display(rec)
```

```
rec = getRecord('John');
display(rec)
```

Asynchronous

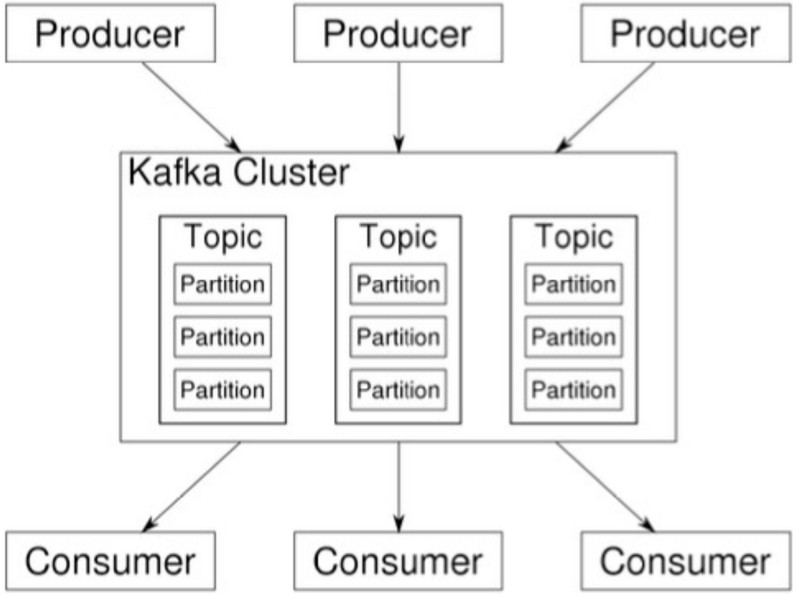
```
function getRecord(key, callback) {
    establish database connection
    retrieve the record for key
   callback(record);
}

function display(rec){
   display rec on the web page
}
```



```
getRecord(‘Rita’, display)
getRecord(‘John’, display)
```

Kafka: An Example of Asynchronous Software Design

- Today, streams of data records, including streams of events, are continuously generated by many online applications.
- A streaming platform enables the development of applications that can continuously and easily consume and process streams of data and events.
- Apache Kafka (Kafka) is a free and open-source distributed streaming platform useful for building, *real time* or *asynchronous*, event-driven applications.
- Kafka offers loose coupling between *producers* and *consumers*.
- Consumers have the option to either consume an event in real time or *asynchronously* at a later time.
- Kafka maintains the chronological order of records/events, ensuring fault tolerance and durability.
- To increase scalability, Kafka separates a topic and stores each partition on a different node.



- *Producer API* – Permits an application to *publish* streams of records/events.
- *Consumer API* – Permits an application to *subscribe* to topics and processes streams of records/events.

END