Understanding

Spring 5 Design Patterns

— Nandakumar Purohit

Spring 5 Design Patterns

Agenda of Topics

- POJO Pattern
- DI Pattern
- Decorator
- Proxy
- Template Method Design Pattern
- Singleton Pattern
- Factory Pattern
- MVC Pattern
- Caching Pattern
- Reactive Pattern

Spring 5 Design Patterns

What will you learn?

- Ability to map design patterns knowledge to Spring Framework
- Understanding few design patterns with practical use case exercises

Spring 5 Design Patterns

POJO Pattern

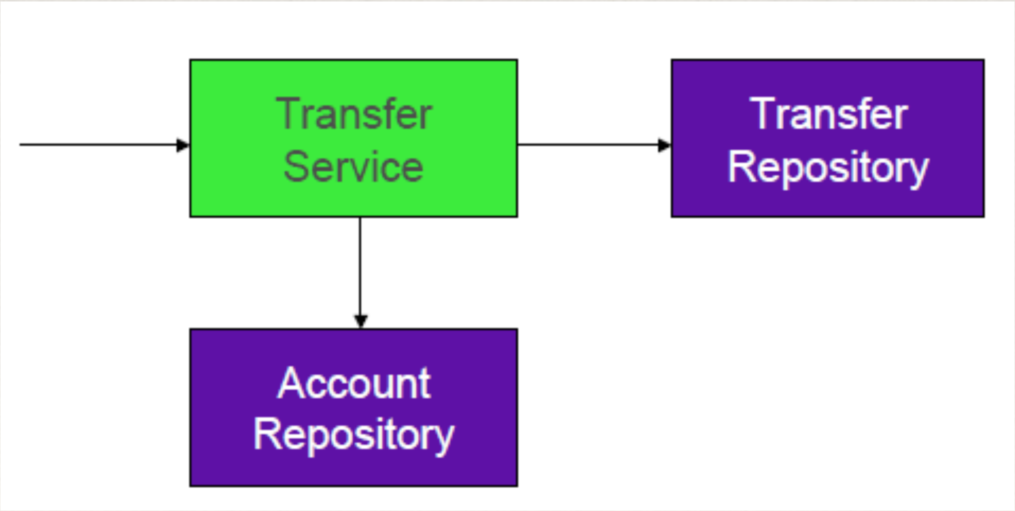
- All application classes are POJOs
- Spring uses the power of POJO pattern for lightweight and minimally invasive development of enterprise applications
- Non-invasive Programming Model
- Spring empowers POJOs by collaborating with other POJOs via Dependency Injection (DI) Pattern

```java
public class HelloWorld {
    public String hello() {
        return "Hello World";
    }
}
```

Spring 5 Design Patterns

Injecting dependencies between POJOs

- Many objects work together for a functionality
- Collaboration between objects is DI



Injecting dependencies between POJOs

```
public class TransferService {
    private AccountRepository 
accountRepository;

    public TransferService() {
        this.accountRepository = new 
AccountRepository();
    }

    public void transferMoney(Account 
a, Account b) {
        accountRepository.transfer(a, 
b);
    }
}
```

- The **TransferService** object needs an **AccountRepository** object
- Direct instantiation **increases coupling** and **scatters code**

Spring 5 Design Patterns

Factory Pattern for dependent components

```
public class TransferService {
    private AccountRepository accountRepository;

    public TransferService() {
        this.accountRepository =
AccountRepositoryFactory.getInstance("jdbc");
    }

    public void transferMoney(Account a, Account b) {
        accountRepository.transfer(a, b);
    }
}
```

- Centralizes the use of new keyword
- Creates objects based on business decisions
- Best practice is to use P2I (Program-2-Interface)

Spring 5 Design Patterns

Factory Pattern with P2I

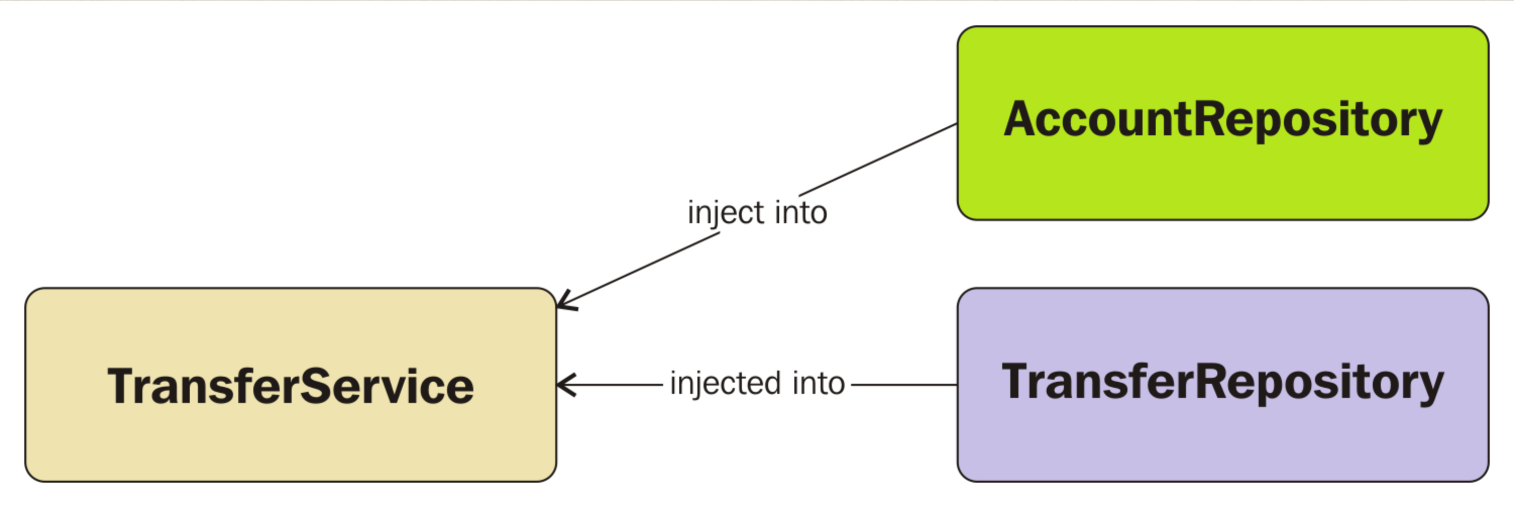
```java
public interface AccountRepository {
    void transfer();
    // other methods
}

public class JdbcAccountRepositry implements AccountRepositry {
    // ...implementation of methods defined in AccountRepositry
    // ...implementation of other methods
}
```

- Concrete class must implement an interface
- Introduces low-coupling (No direct dependency on implementation)
- We are still adding Factory classes to the business component

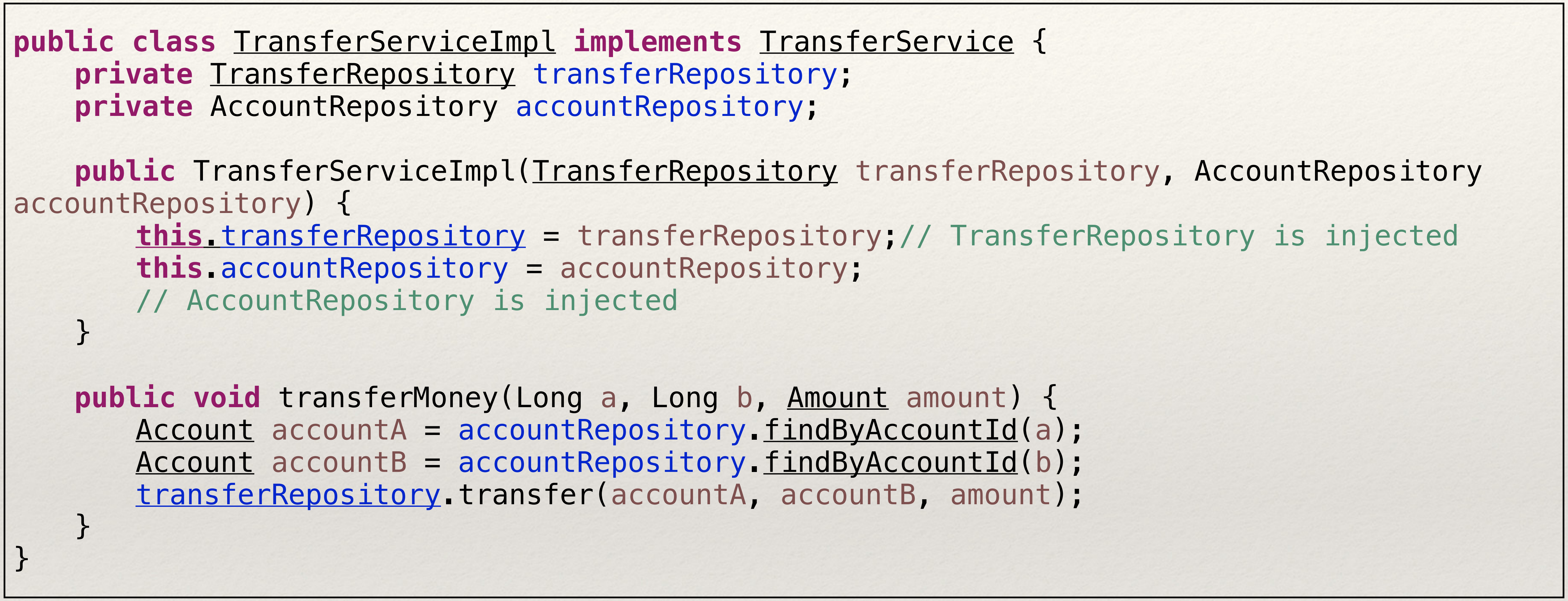
Spring 5 Design Patterns

Using DI Pattern for dependent components

- Dependent objects are **given their dependencies** at the time of **object creation** by Factory
- The factory ensures that **dependency object is not expected to create their dependencies**
- Focus on **Defining the Dependencies** rather than **Resolving Dependencies**



Using DI Pattern for dependent components



```
public class TransferServiceImpl implements TransferService {
    private TransferRepository transferRepository;
    private AccountRepository accountRepository;

    public TransferServiceImpl(TransferRepository transferRepository, AccountRepository
accountRepository) {
        this.transferRepository = transferRepository;// TransferRepository is injected
        this.accountRepository = accountRepository;
        // AccountRepository is injected
    }

    public void transferMoney(Long a, Long b, Amount amount) {
        Account accountA = accountRepository.findByAccountId(a);
        Account accountB = accountRepository.findByAccountId(b);
        transferRepository.transfer(accountA, accountB, amount);
    }
}
```

- **TransferService** has dependency with **AccountRepository** and **TransferRepository**
- We can either use **JdbcTransferRepository** or **JpaTransferRepository**
- **TransferServiceImpl** is **flexible** enough to take on any **TransferRepository** it's given

Spring 5 Design Patterns

Decorator Pattern

- Decorator Pattern allows you to add and remove behaviors for an individual object at runtime dynamically or statically, without changing the existing behavior of other associated objects from the same class.
- This design pattern does this without violating the **Single Responsibility Principle** or the **SOLID principle** of object-oriented programming
- This design pattern uses the **compositions over the inheritance** for objects associations
- It allows you to divide the functionality into **different concrete classes** with a unique area of concern.

Decorator Pattern - Benefits

- This pattern allows you to extend functionality dynamically and statically without altering the structure of existing objects
- By using this pattern, you could add a new responsibility to an object dynamically
- This pattern is also known as a Wrapper
- This pattern uses the compositions for object relationships to maintain SOLID principles
- This pattern simplifies coding by writing new classes for every new specific functionality rather than changing the existing code of your application

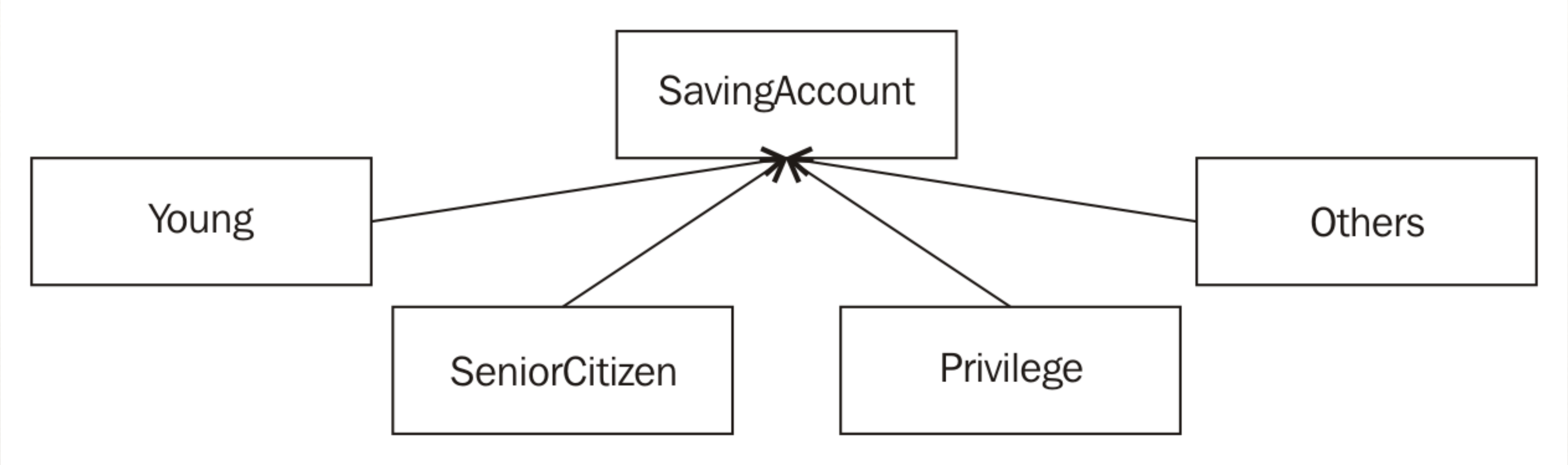
Spring 5 Design Patterns

Decorator Pattern - Solving common problems

- Consider that a bank offers multiple accounts with different benefits to customers.
- It divides the customers into three categories--**senior citizens, privileged, and young**.
- The bank launches a scheme on the savings account for **senior citizens**--if they open a savings account in this bank, they will be provided medical insurance of up to $1,000.
- Similarly, the bank also provides a scheme for the **privileged** customers as an accident insurance of up to $1,600 and an overdraft facility of $84.
- There is no scheme for the young.

Decorator Pattern - Solving common problems

Application design with inheritance (without Decorator)

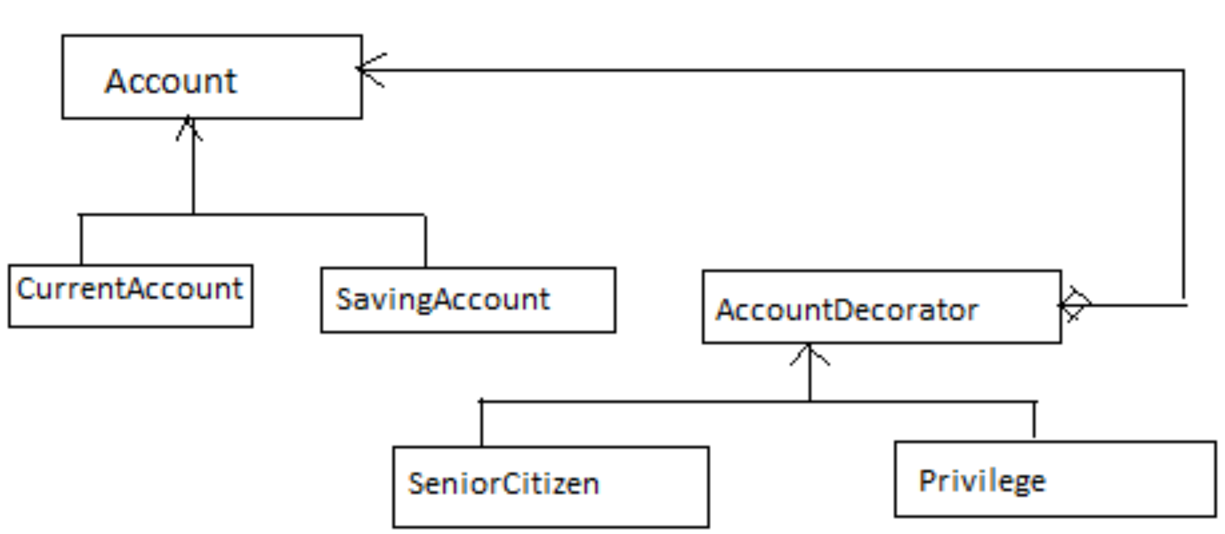


- Add new subclasses to SavingsAccount
- Each subclass represents SavingsAccount with additional benefits as decoration
- The **design is complex** as we add more benefits to SavingsAccount ONLY

Spring 5 Design Patterns

Decorator Pattern - Solving common problems

Application design with Decorator



- **IS-A relationship** between the **AccountDecorator** and **Account**, that is, **inheritance** for the correct type

- **HAS-A** relationship between the **AccountDecorator** and **Account**, that is, **composition** in order **to add new behavior** without changing the existing code

Spring 5 Design Patterns

Decorator Pattern in Spring Framework

- Weaving the **Advice** into the Spring application. It uses the Decorator pattern via the CGLib proxy. It works by generating a subclass of the target class at runtime.

- **org.springframework.beans.factory.xml.BeanDefinitionDecorator** : It is used to decorate the bean definition via applied custom attributes.

- **org.springframework.web.socket.handler.WebSocketHandlerDecorator**: It is used to decorate a WebSocketHandler with additional behaviors.

Spring 5 Design Patterns

Proxy Pattern

- Proxy design pattern provides an object of class that has the functionality of another class.
- Provide a surrogate or placeholder for another object to control access to it.
- The intent of this design pattern is to provide a different class for another class with its functionality to the outer world.

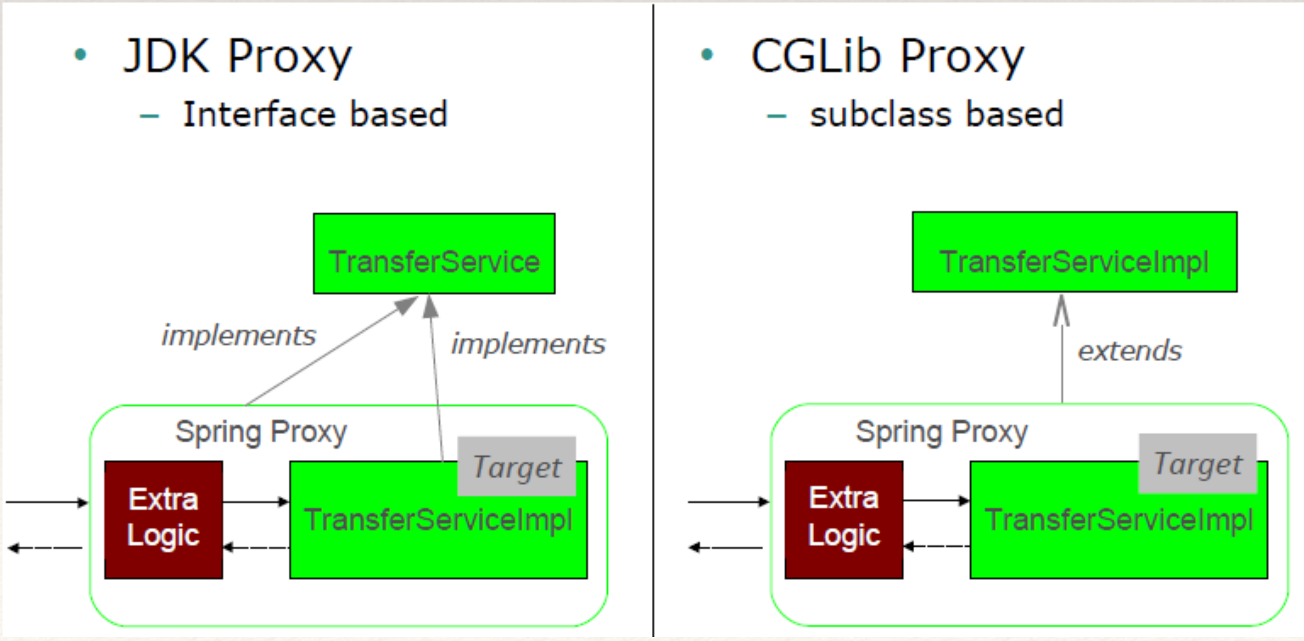
When to Use Proxy…?

- When we want a **simplified version** of a complex or heavy object
- When the original object is present in different address space, and we want to **represent it locally**
- When we want to add a **layer of security** to the original underlying object to provide **controlled access** based on access rights of the client

Spring 5 Design Patterns

Proxy Pattern in Spring Framework

- Spring provides two ways to create the proxy in the application.
  - CGLIB proxy
  - JDK proxy or dynamic proxy
- In Spring AOP, CGLIB is used to create the proxy in the application.
- CGLIB proxying works by generating a subclass of the target class at runtime.
- Spring configures this generated subclass to delegate method calls to the original target--the subclass is used to implement the Decorator pattern, weaving in the advice.



Proxy Pattern in Spring Framework

- Use Case Problem - Bank Accounts
- Create a custom service class to replace context.getBean() and implement @Before Advice using Proxy pattern when a method is invoked from SavingAccount class.

Spring 5 Design Patterns

Template Method Pattern

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure
- It reduces the boilerplate codes in the application by reusing code.
- This pattern creates a template or way to reuse multiple similar algorithms to perform some business requirements.

Spring 5 Design Patterns

Template Method Pattern

The Resource Management Problem

- *What happens when you order for a Pizza…?*
- **Pizza Order Application using JDBC**
  - Define the connection parameters.
  - Access a data source, and establish a connection.
  - Begin a transaction.
  - Specify the SQL statement.
  - Declare the parameters, and provide parameter values.
  - Prepare and execute the statement.
  - Set up the loop to iterate through the results.
  - Do the work for each iteration--execute the business logic.
  - Process any exception.
  - Commit or roll back the transaction.
  - Close the connection, statement, and ResultSet.

Spring 5 Design Patterns

Template Method Pattern in Spring Framework

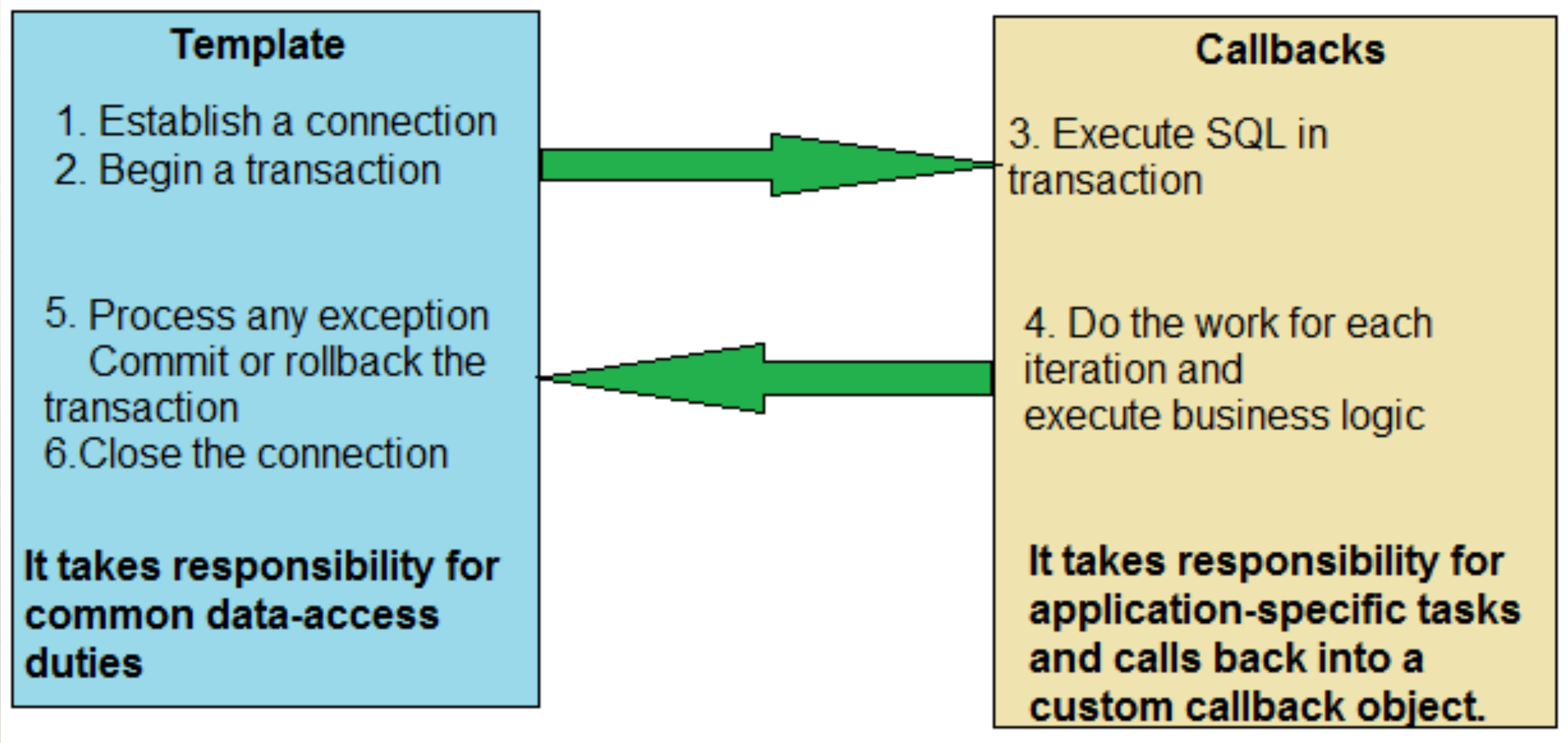
Pizza Order Application using JdbcTemplate

- **The Resource management from Spring**
  - Define the connection parameters.
  - **Access a data source, and establish a connection.**
  - **Begin a transaction.**
  - Specify the SQL statement.
  - Declare the parameters, and provide parameter values.
  - Prepare and execute the statement.
  - Set up the loop to iterate through the results.
  - Do the work for each iteration--execute the business logic.
  - **Process any exception.**
  - Commit or roll back the transaction.
  - **Close the connection, statement, and ResultSet.**

- **Fixed steps in the process**
  - Taking the Order
  - Preparing the Pizza
  - Adding the toppings
  - Delivering to customer

Spring 5 Design Patterns

Template Method Pattern in Spring Framework



Template

1. Establish a connection
2. Begin a transaction

5. Process any exception
   Commit or rollback the
transaction
6.Close the connection

It takes responsibility for common data-access duties

Callbacks

3. Execute SQL in
transaction

4. Do the work for each
iteration and
execute business logic

It takes responsibility for application-specific tasks and calls back into a custom callback object.

RestTemplate

JdbcTemplate

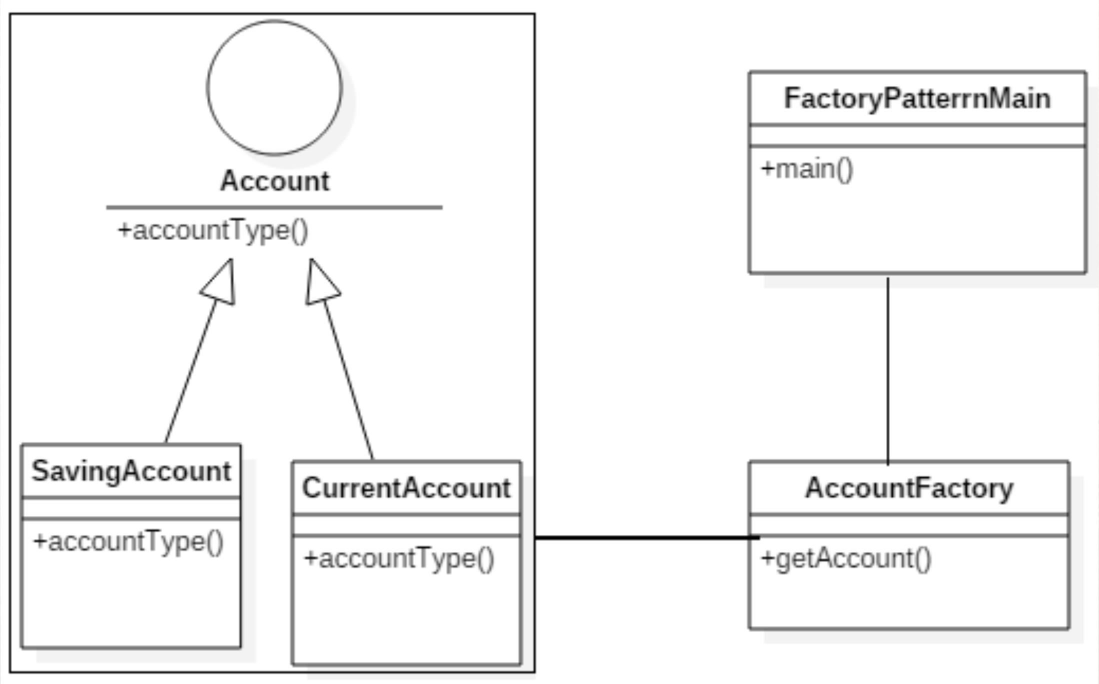
JmsTemplate

WebServiceTemplate

NamedParameterJdbcTemplate

Spring 5 Design Patterns

Factory Pattern



- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses.
- The Factory pattern promotes loose coupling between collaborating components or classes by using interfaces rather than binding application-specific classes into the application code
- Using this pattern, you can get an implementation of an object of classes that implement an interface at runtime
- The object life cycle is managed by the factory implemented by this pattern

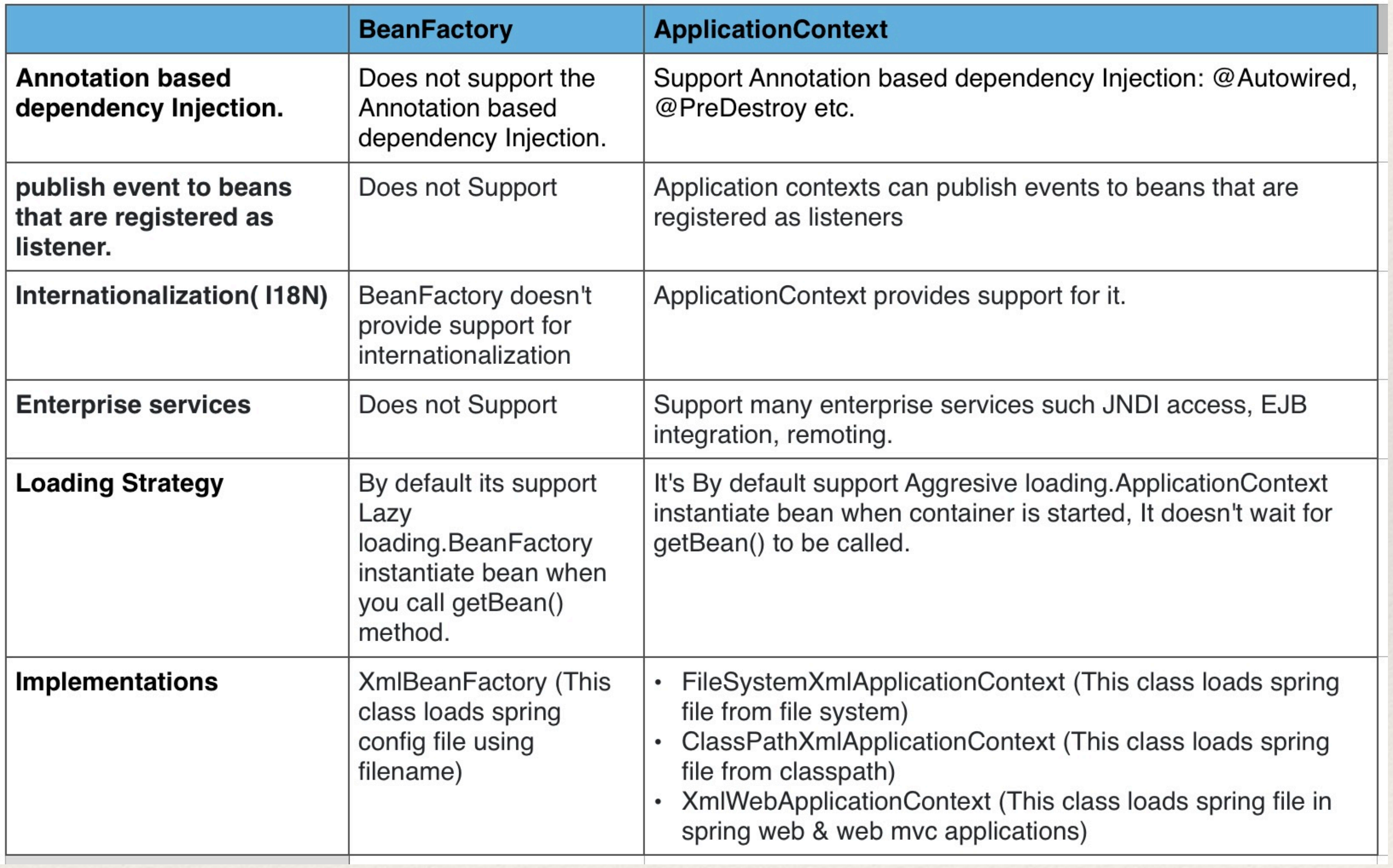
Spring 5 Design Patterns

Factory Pattern in Spring Framework

- Spring Framework transparently uses this Factory design pattern to implement Spring containers using **BeanFactory** and **ApplicationContext** interfaces.

- Spring's container works based on the Factory pattern to create spring beans for the Spring application and also manages the life cycle of every Spring bean.

- The ApplicationContext & BeanFactory are factory interfaces, and Spring has lots of implementing classes.

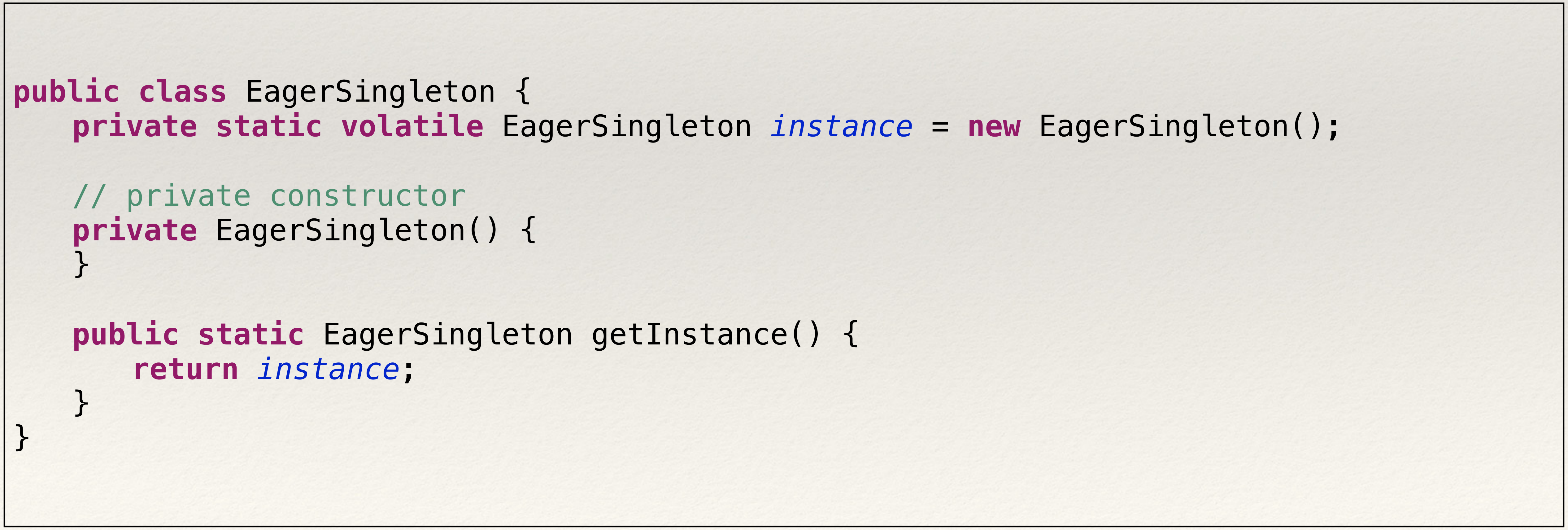


	BeanFactory	**ApplicationContext**
Annotation based dependency Injection.	Does not support the Annotation based dependency Injection.	Support Annotation based dependency Injection: @Autowired, @PreDestroy etc.
publish event to beans that are registered as listener.	Does not Support	Application contexts can publish events to beans that are registered as listeners
Internationalization(I18N)	BeanFactory doesn't provide support for internationalization	ApplicationContext provides support for it.
Enterprise services	Does not Support	Support many enterprise services such JNDI access, EJB integration, remoting.
Loading Strategy	By default its support Lazy loading.BeanFactory instantiate bean when you call getBean() method.	It's By default support Aggresive loading.ApplicationContext instantiate bean when container is started, It doesn't wait for getBean() to be called.
Implementations	XmlBeanFactory (This class loads spring config file using filename)	• FileSystemXmlApplicationContext (This class loads spring file from file system) • ClassPathXmlApplicationContext (This class loads spring file from classpath) • XmlWebApplicationContext (This class loads spring file in spring web & web mvc applications)

Spring 5 Design Patterns

Singleton Pattern - Eager

- This is a design pattern where an instance of a class is created much before it is actually required.
- It is done on system startup.
- In an eager initialization singleton pattern, the singleton instance is created irrespective of whether any other class actually asked for its instance or not.

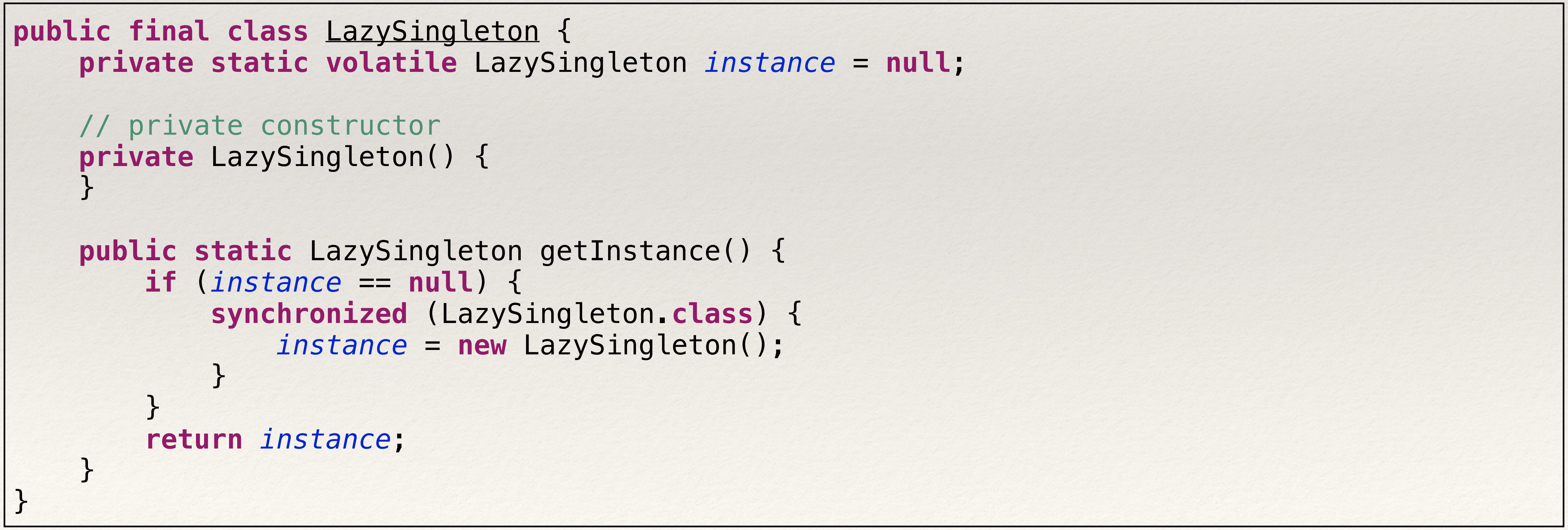


```java
public class EagerSingleton {
    private static volatile EagerSingleton instance = new EagerSingleton();

    // private constructor
    private EagerSingleton() {
    }

    public static EagerSingleton getInstance() {
        return instance;
    }
}
```

Spring 5 Design Patterns

Singleton Pattern - Lazy

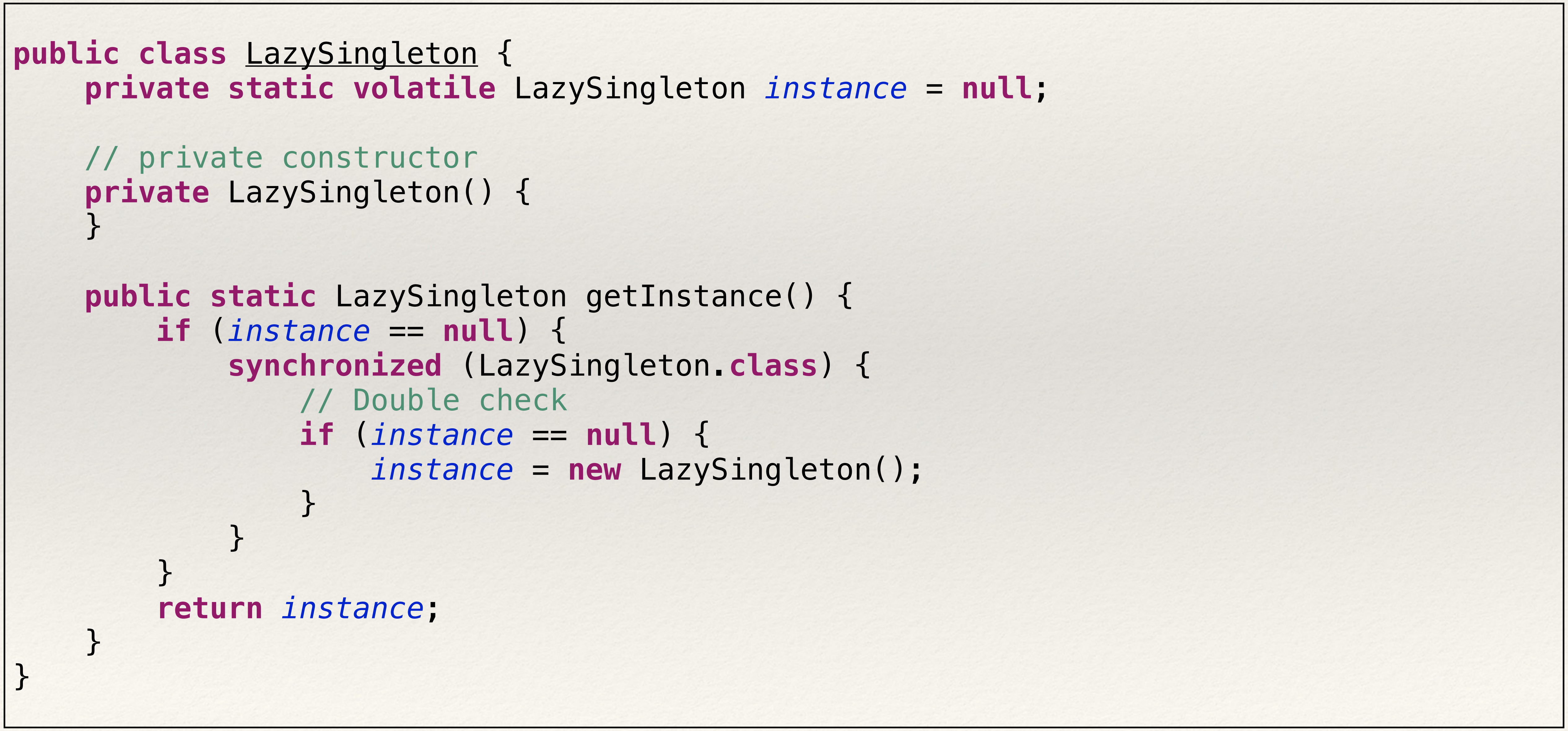
- In computer programming, lazy initialization is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process, until the first time it is needed.
- In a singleton pattern, it restricts the creation of an instance until it is requested for first time.



```java
public final class LazySingleton {
    private static volatile LazySingleton instance = null;

    // private constructor
    private LazySingleton() {
    }

    public static LazySingleton getInstance() {
        if (instance == null) {
            synchronized (LazySingleton.class) {
                instance = new LazySingleton();
            }
        }
        return instance;
    }
}
```

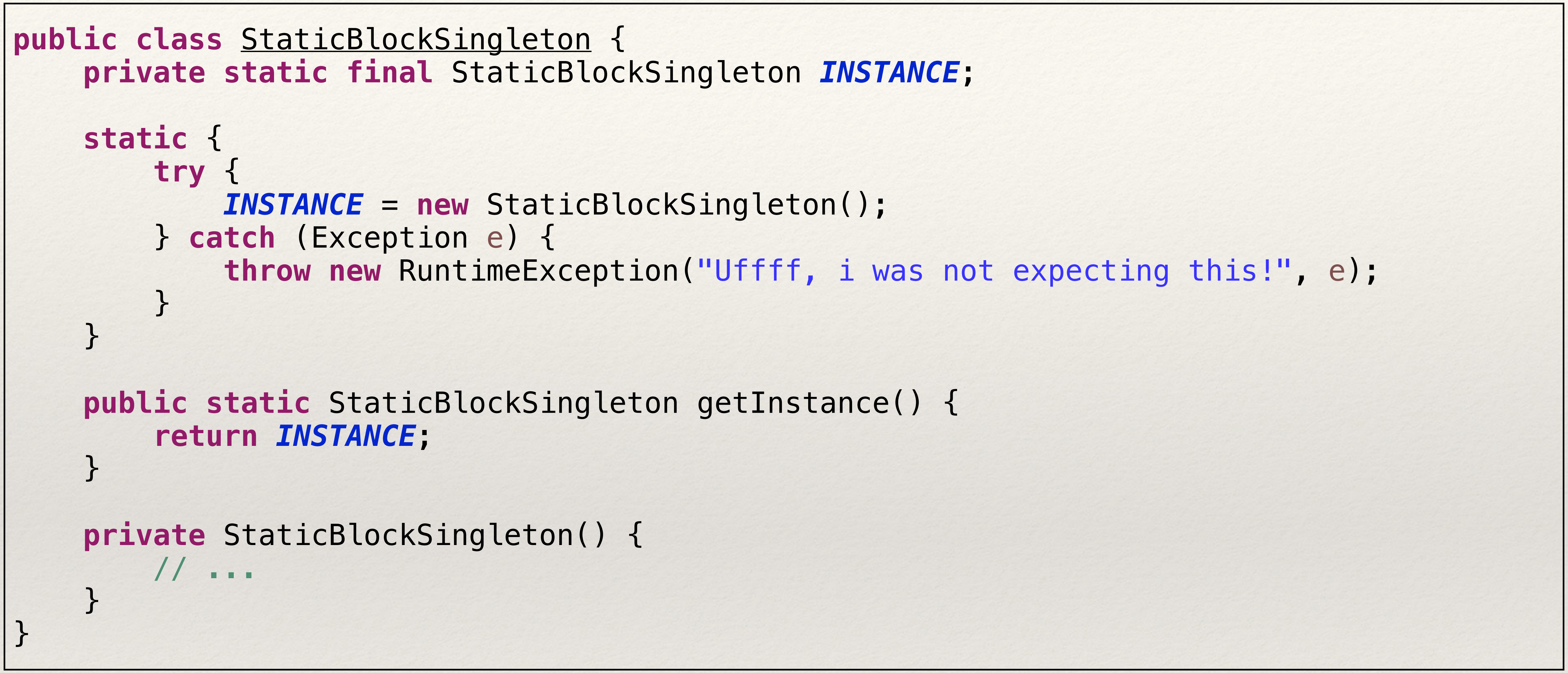
Singleton Pattern - Double-checked locking

- This principle tells us to recheck the instance variable again in the synchronized block



```java
public class LazySingleton {
    private static volatile LazySingleton instance = null;

    // private constructor
    private LazySingleton() {
    }

    public static LazySingleton getInstance() {
        if (instance == null) {
            synchronized (LazySingleton.class) {
                // Double check
                if (instance == null) {
                    instance = new LazySingleton();
                }
            }
        }
        return instance;
    }
}
```

Singleton Pattern - Static Block Initialization



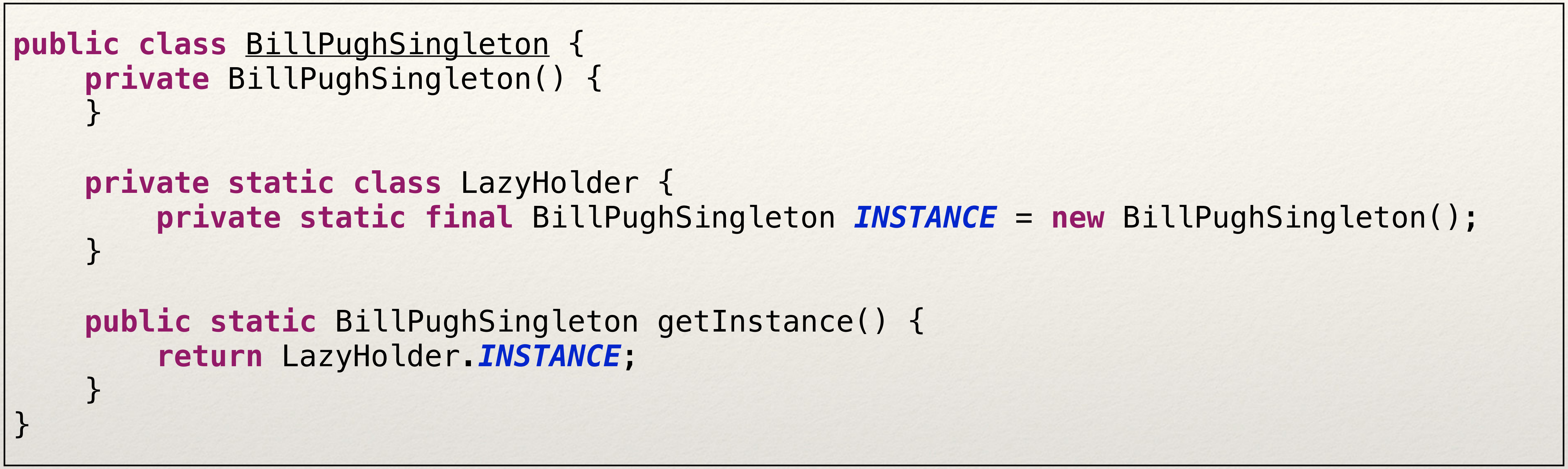
```java
public class StaticBlockSingleton {
    private static final StaticBlockSingleton INSTANCE;

    static {
        try {
            INSTANCE = new StaticBlockSingleton();
        } catch (Exception e) {
            throw new RuntimeException("Uffff, i was not expecting this!", e);
        }
    }

    public static StaticBlockSingleton getInstance() {
        return INSTANCE;
    }

    private StaticBlockSingleton() {
        // ...
    }
}
```

Drawback

Even if not needed, an object is created during class loading

Spring 5 Design Patterns

Singleton Pattern - Static Block Initialization



```
public class BillPughSingleton {
    private BillPughSingleton() {
    }

    private static class LazyHolder {
        private static final BillPughSingleton INSTANCE = new BillPughSingleton();
    }

    public static BillPughSingleton getInstance() {
        return LazyHolder.INSTANCE;
    }
}
```

- **Bill Pugh** was main force behind the **java memory model** changes.

- His principle **"Initialization-on-demand holder idiom"** also uses the static block idea, but in a different way.

- "As you can see, until we need an instance, the LazyHolder class will not be initialized until required and you can still use other static members of BillPughSingleton class. *This is the solution, i will recommend to use. I have used it in my all projects."*

Spring 5 Design Patterns

Singleton Pattern in Spring Framework

- According to the Singleton pattern, a scoped bean in the Spring Framework means a single bean instance per container and per bean.
- If you define one bean for a particular class in a single Spring container, then the Spring container creates one and only one instance of the class defined by that bean definition.

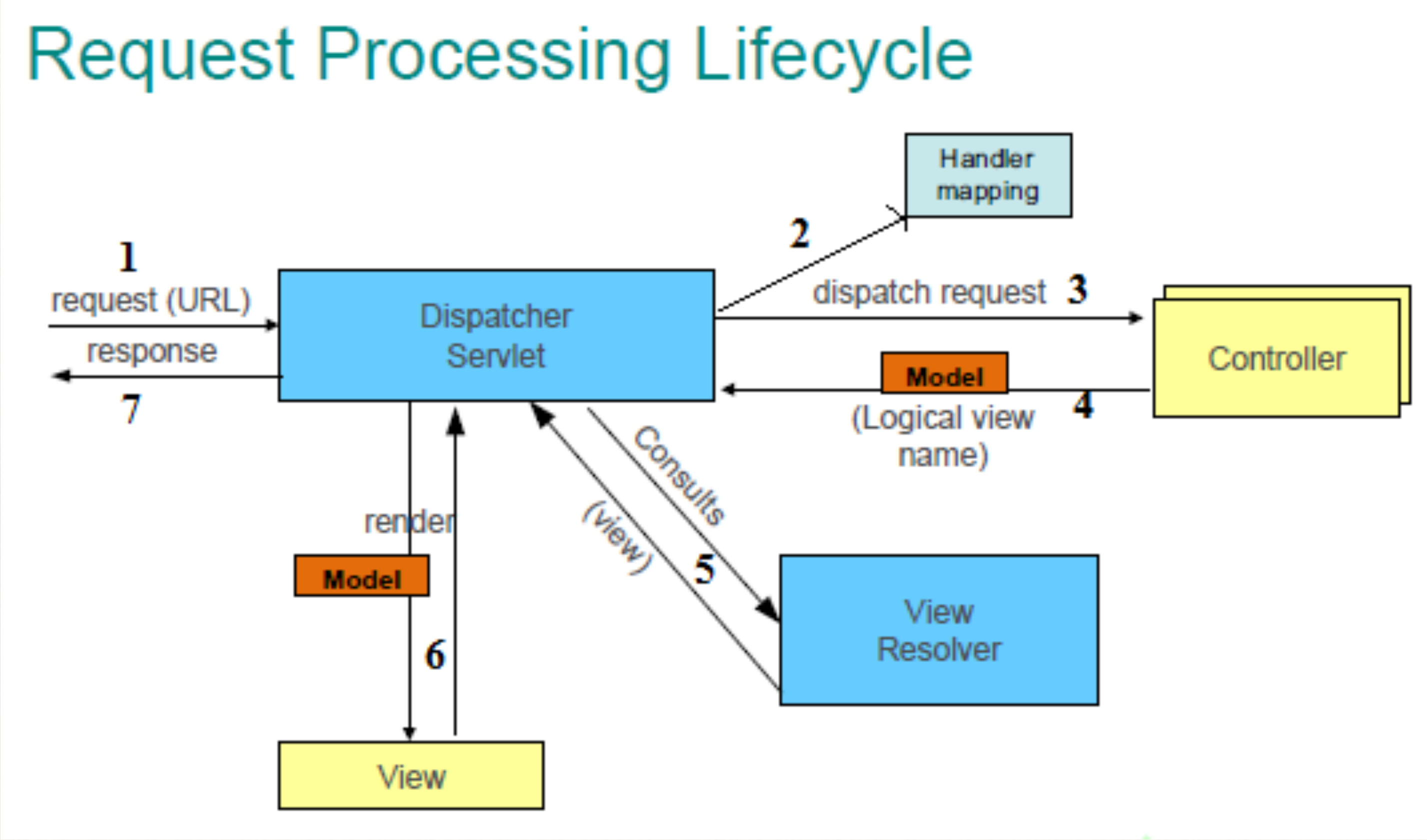
Spring 5 Design Patterns

MVC Pattern in Spring Framework

The Front Controller Pattern

The Front Controller handles following issues of MVC Model 1

- Too many controllers are required to handle too many requests. It is difficult to maintain and reuse them.
- Each request has its own point of entry in the web application; it should be a single point of entry for each request.
- JSP and Servlet are the main components of the Model 1 MVC pattern, so, these components handle both action and view, violating the Single Responsibility principle.



Spring 5 Design Patterns

Caching Pattern

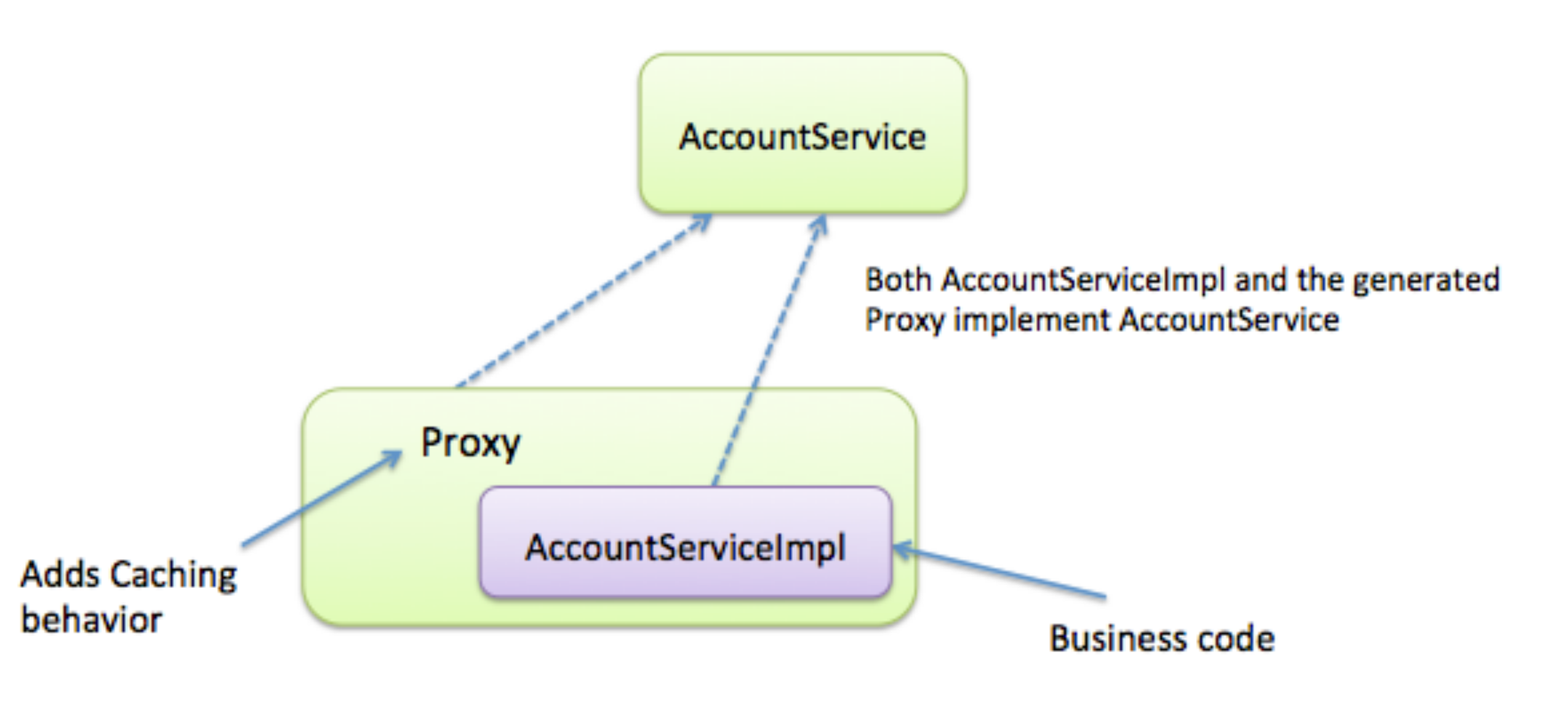
- Cache Providers
  - Redis
  - Memcached
  - OrmLiteCacheClient
  - InMemory Cache
  - AWS DynamoDB Cache Client
  - Azure Cache Client
  - EHCache
  - JBoss Cache
- Caching declaration: Recognize those methods in the application that need to be cached, and annotate these methods either with caching annotations, or you can use XML configuration by using Spring AOP
- Cache configuration: This means that you have to configure the actual storage for the cached data--the storage where the data is stored and read from

- **Caching declaration:** Recognize those methods in the application that need to be cached, and annotate these methods either with caching annotations, or you can use XML configuration by using Spring AOP
- **Cache configuration:** This means that you have to configure the actual storage for the cached data--the storage where the data is stored and read from

Spring 5 Design Patterns

Caching Pattern - Enable Caching in Spring

Enable Caching using Proxy Pattern


AccountService
Both AccountServiceImpl and the generated Proxy implement AccountService
Proxy
AccountServiceImpl
Adds Caching behavior
Business code

- Spring applies caching to Beans methods by using AOP
- Spring applies Proxy around the Spring Beans where methods need to be cached
- Use **@EnableCaching**

Pre-requisite before Reactive Pattern

It is highly important to understand how event notifications are generated and broadcasted in real-time applications.

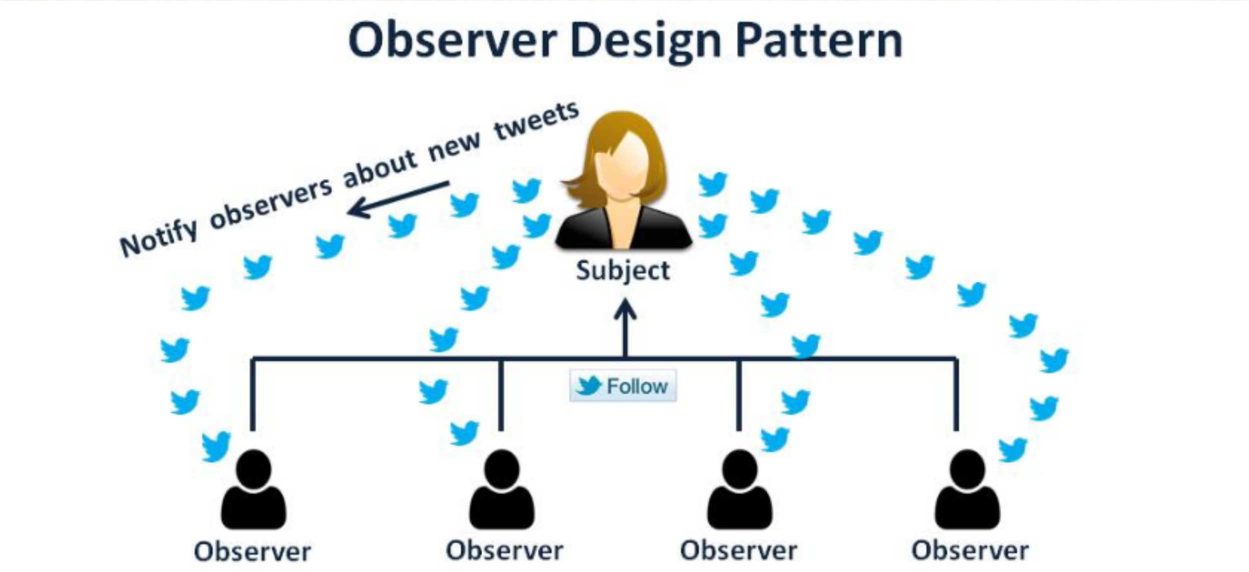
So, lets understand **OBSERVER** Design Pattern before we dive into Reactive Scenarios

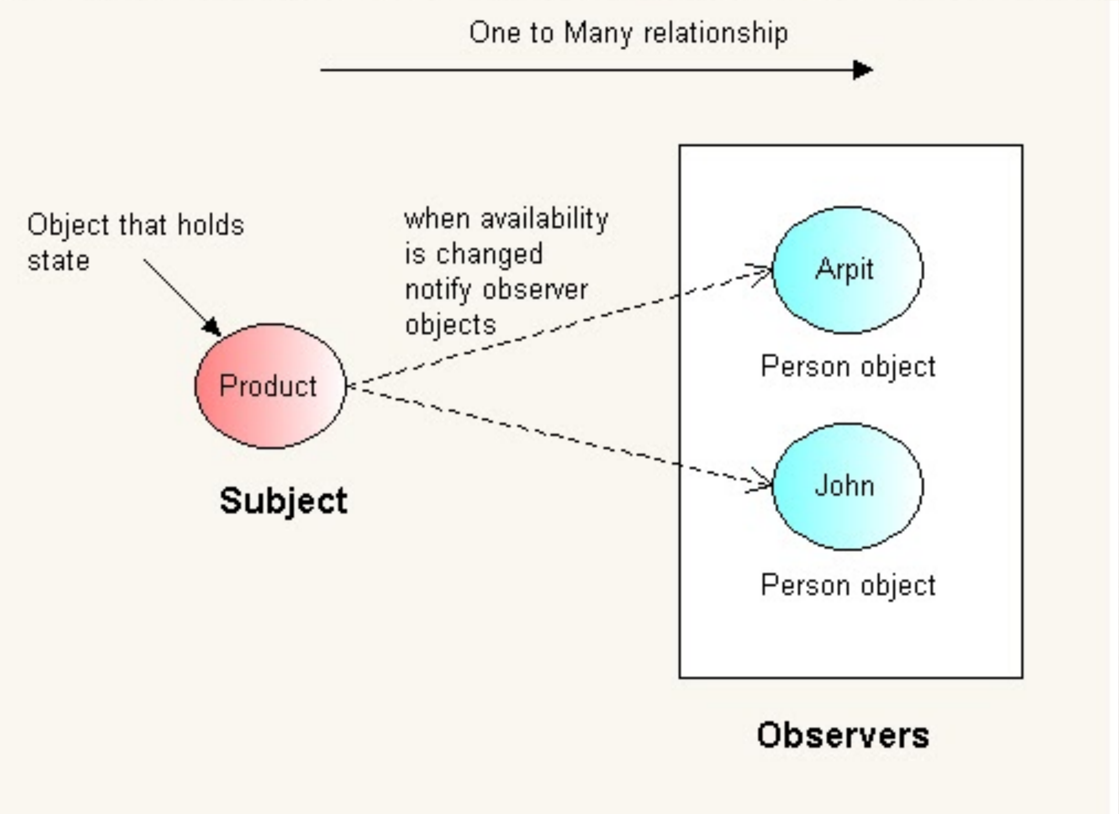
Observer Design Pattern

- *“The* ***observer*** *pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.”*
- The object which is being watched is called the ***subject***.
- The objects which are watching the state changes are called ***observers*** or ***listeners***.

Spring 5 Design Patterns

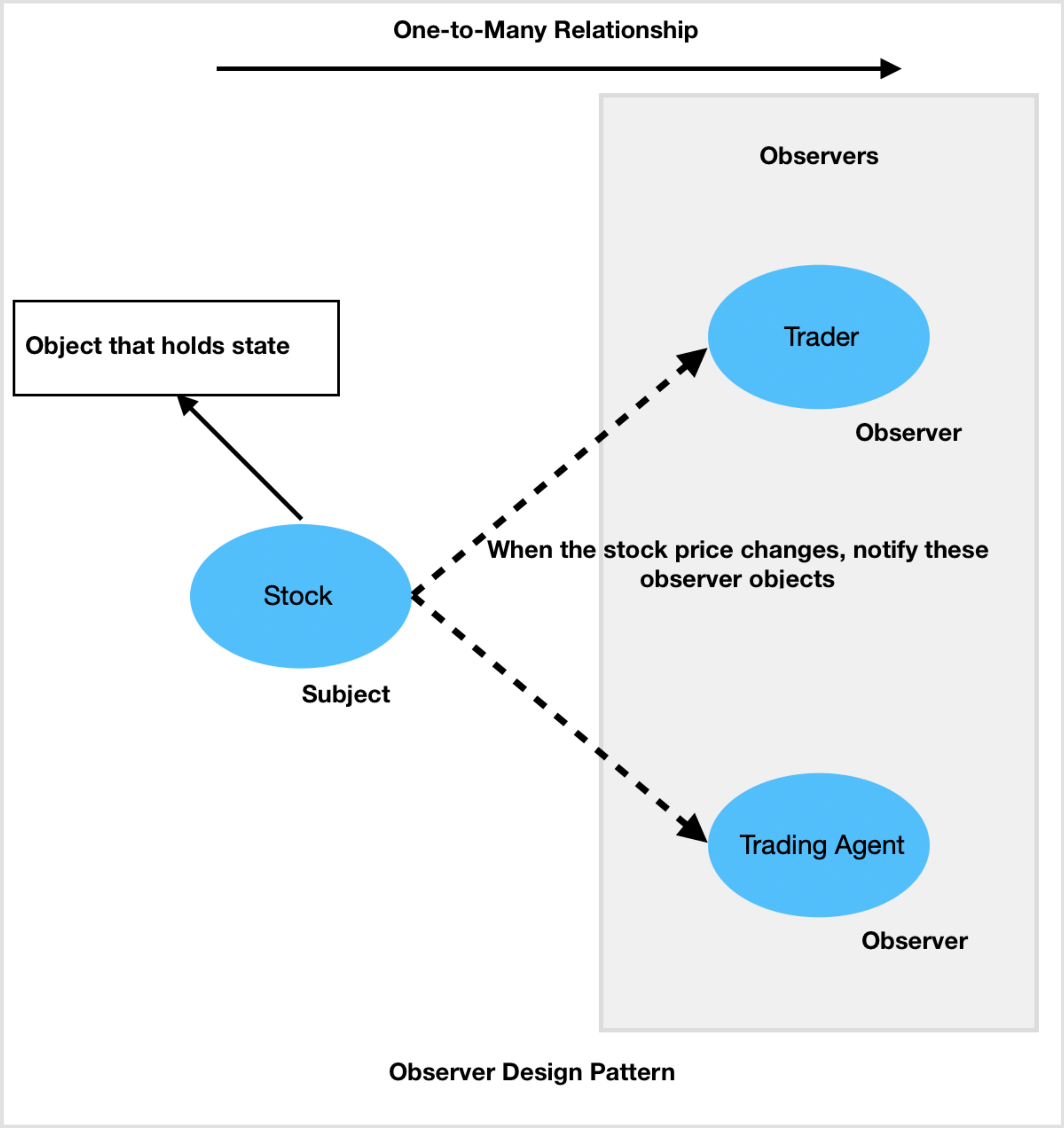
Observer Design Pattern





Spring 5 Design Patterns

Observer Design Pattern



Spring 5 Design Patterns

Reactive Pattern

Requirements	Now	15 Years ago
Server nodes	More than 1000 nodes required.	Ten nodes were enough.
Response times	Takes milliseconds to serve requests, and send back responses.	Took seconds to response.
Maintenance downtimes	Currently, there is no or zero maintenance downtime required.	Took hours of maintenance downtime.
Data volume	Data for the current application that increased to PBs from TBs.	Data was in GBs.

Spring 5 Design Patterns

Reactive Pattern - Modern Applications

Current Expectations

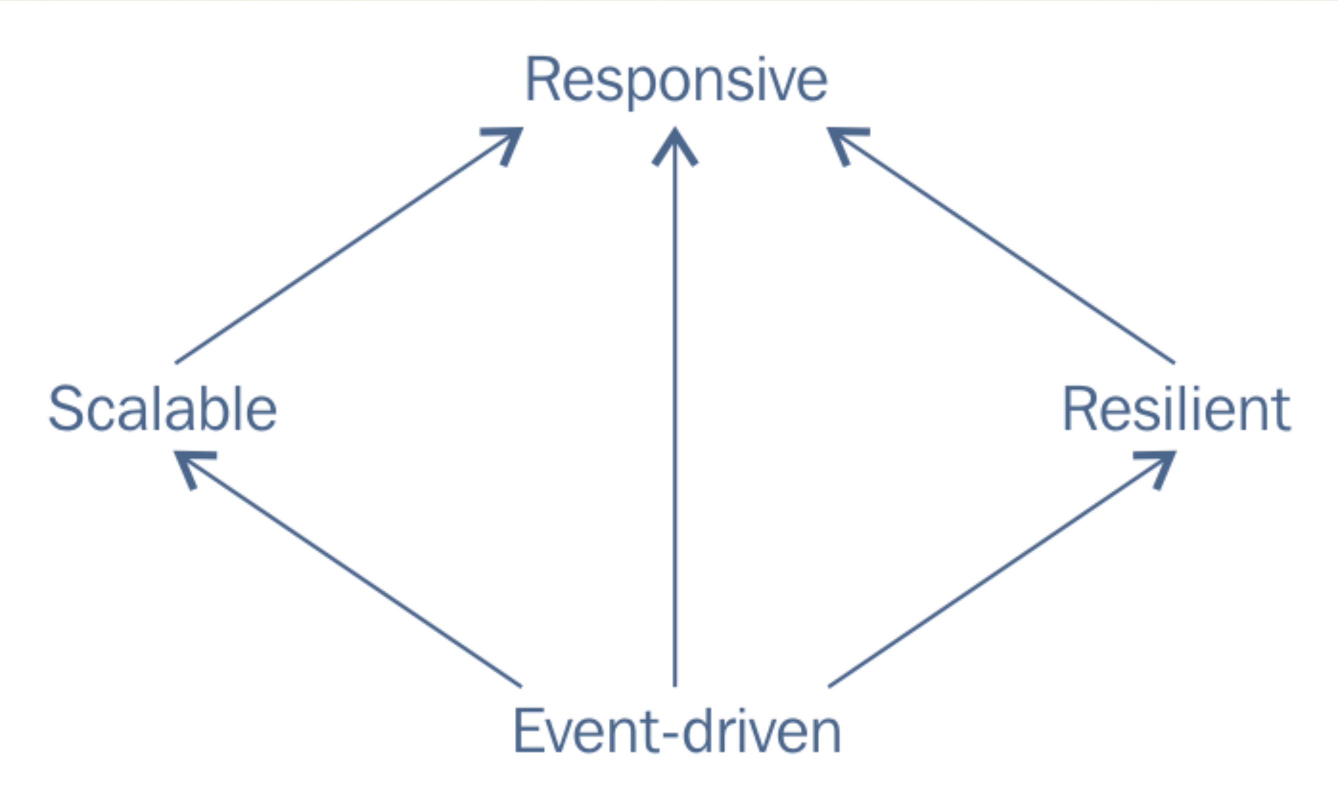
- Robust
- Resilient
- Flexible
- Better positioned
- Application level
- System level

Reactive Pattern Traits

- Must be Responsive
- Must be Resilient
- React to Variable loads
- Must not be overloaded
- React to Events

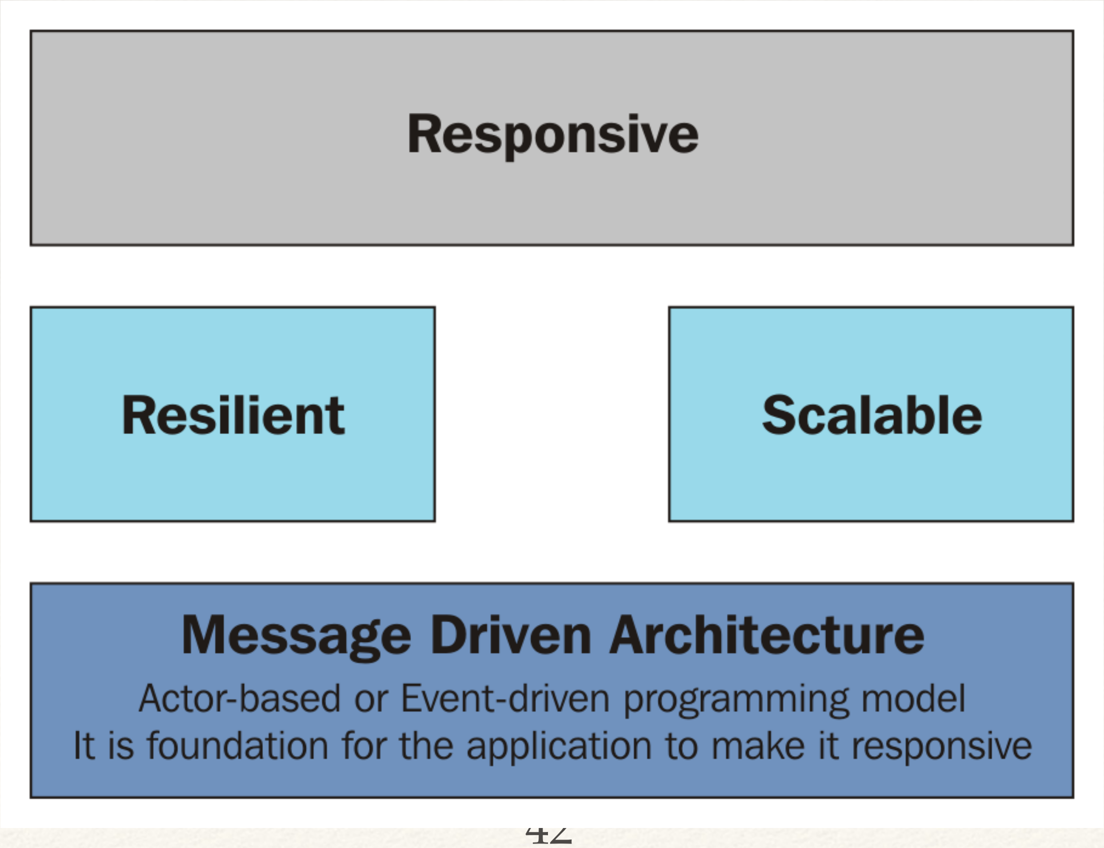
Reactive Pattern - Traits

- **Responsive:** This is the goal of each application today.
- **Resilient:** This is required to make an application responsive.
- **Scalable:** This is also required to make an application responsive; without resilience and scalability, it is impossible to achieve responsiveness.
- **Message-driven:** A message-driven architecture is the base of a scalable and resilient application, and ultimately, it makes a system responsive. Message-driven either based on the event-driven or actor-based programming model.



Reactive Pattern - Responsiveness

- It means that the application or system **responds quickly** to all users in a given time in all conditions, and that is in good condition as well as bad. It ensures a consistent positive user experience.
- Required for a system for usability and utility
- Upon system failure, the failures are detected quickly, and dealt with effectively
- A user must not face any failure
- It must deliver a consistent quality of service to the user



Responsive

Resilient

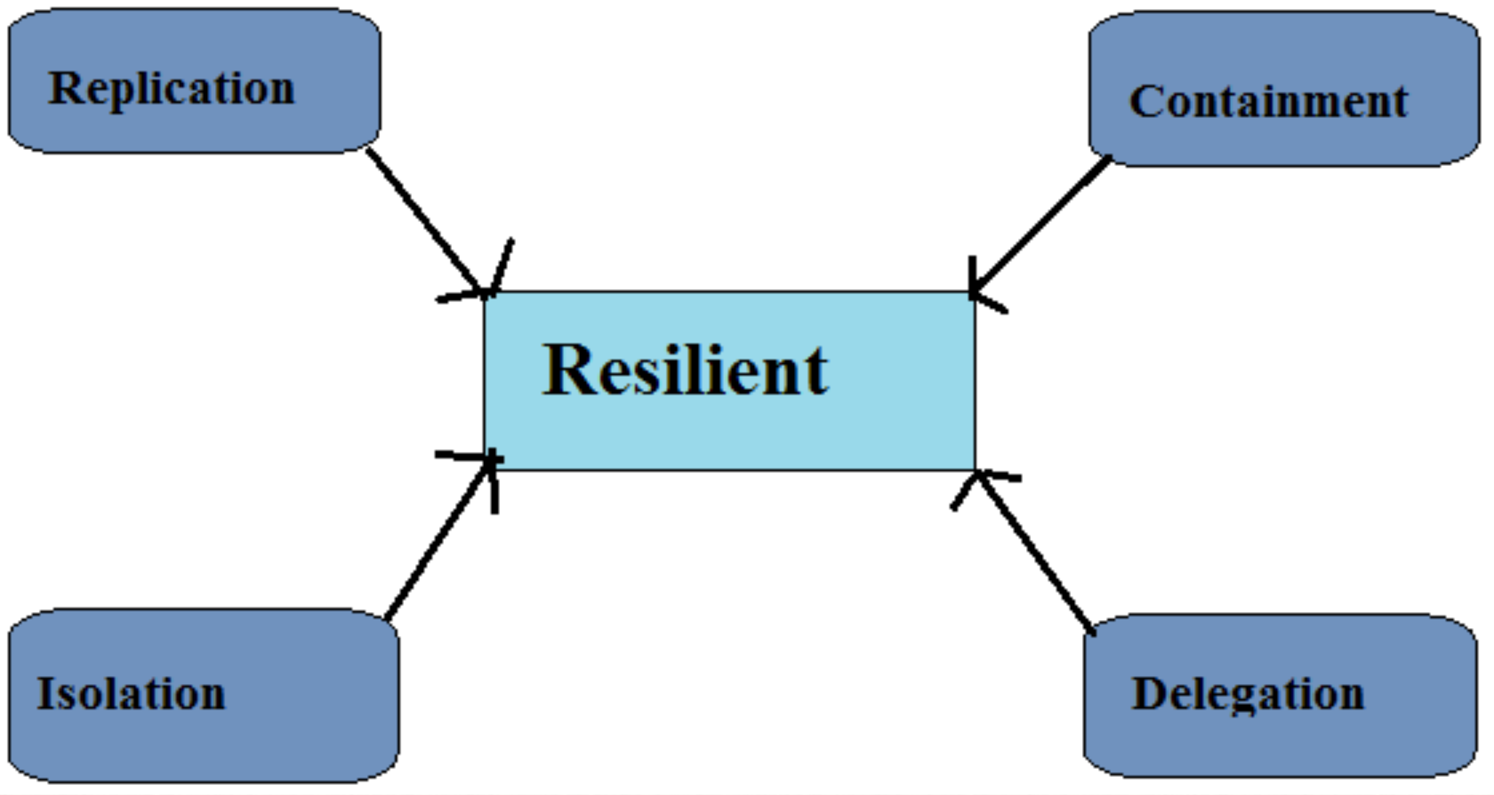
Scalable

Message Driven Architecture

Actor-based or Event-driven programming model
It is foundation for the application to make it responsive

Reactive Pattern - Resilience

- Any major application failures result in downtime, data loss & causes bad reputation in market. Enabling application to be responsive in all the conditions is called as a **Resilience**
- Each **component must be isolated** from each other
- Recovery of a component is via **Replication**



Spring 5 Design Patterns

Reactive Pattern - Resilience

- **Replication:** This ensures high-availability, where necessary, at the time of component failure.
- Isolation: This means that the failure of each component must be isolated, which is achieved by decoupling the components as much as possible. **It enables self-heal & performance measure**
- Containment: The result of decoupling is containment of the failure. It helps avoid failure in the system as a whole.
- Delegation: After failure, the recovery of each component is delegated to another component. It is possible only when our system is composable.

Reactive Pattern - Scalable

A scalable system or an elastic system can easily be upgraded under a varying workload. Resiliency and scalability together make a system consistently responsive.

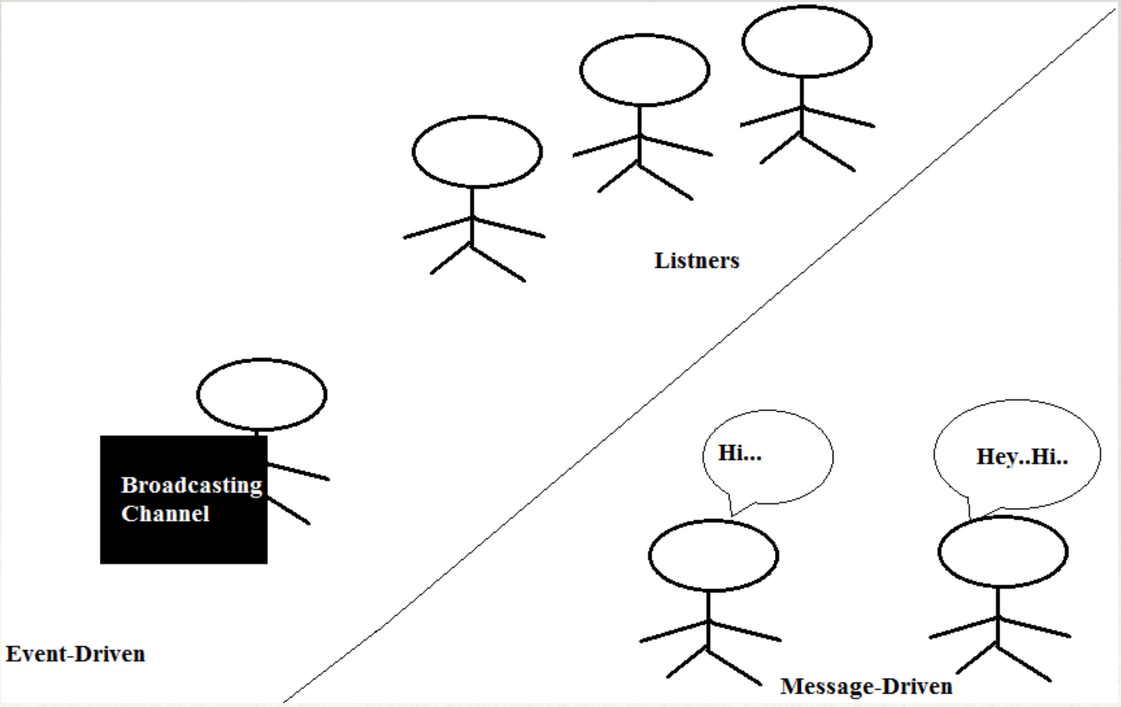
- **scale-up:** It makes use of parallelism in multi-core systems.
- **scale-out:** It makes use of multi-server nodes. Location transparency and resilience are important for this.

> **Elasticity** and **Scalability** are both the same! Scalability is all about the efficient use of resources already available, while elasticity is all about adding new resources to your application on demand when the needs of the system changed. So, eventually, the system can be made responsive anyway--by either using the existing resources of the system or by adding new resources to the system.

Spring 5 Design Patterns

Reactive Pattern - Message-driven Architecture

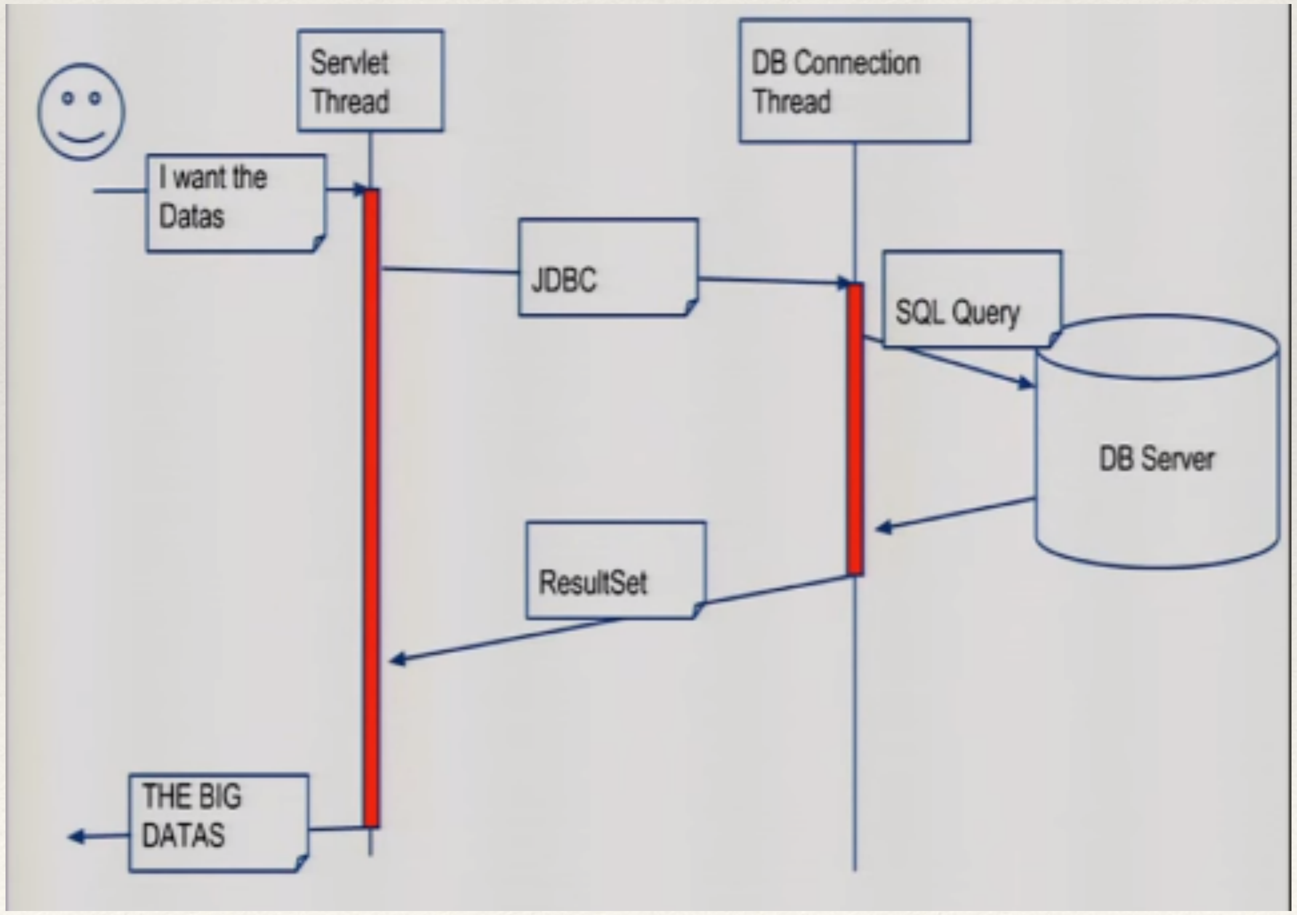
- A message-driven architecture is the base of a responsive application.
- Can be an event-driven and actor-based application.
- It can also be a combination of both the architectures
    - Event-driven architecture
    - Message-driven architecture.



- Loose coupling
- Isolation
- Location Transparency
- Isolation depends on Loose coupling
- Events can be handled asynchronously

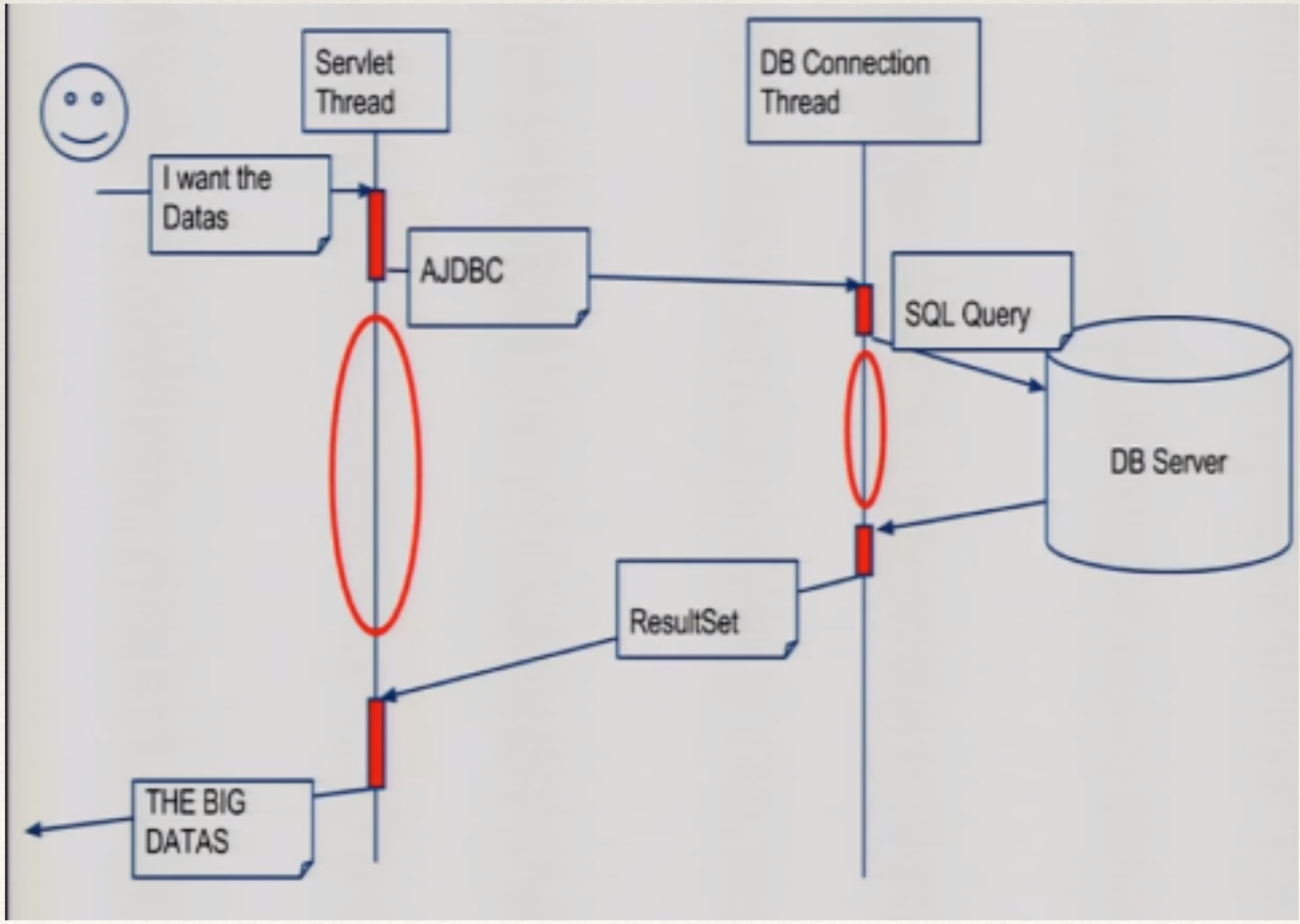
Blocking Calls

- A call may be holding the resources while other calls wait for the same resources.
- Blocking a call means some operations in the application or system that take a longer time to complete, such as file I/O operations and database access using blocking drives.



Non-Blocking Calls

- A thread competes for a resource without waiting for it.
- If the resources are not available at the time of calling, then it moves to other work rather than waiting for the blocked resources. The system is notified when the blocked resources are available.



APIs in Spring Framework with Design Patterns

CREATIONAL	STRUCTURAL	BEHAVIORAL
Singleton	Composite	Command
Prototype	Decorator, Facade	**CoR** • Authentication • Authorization
Factory	**BeanDefinitionDecorator** • WebSocketHandlerDecorator	**Interpreter** • Spring EL
Abstract Factory • ProxyFactoryBean • JndiFactoryBean • LocalSessionFactoryBean • LocalContainerEntityManagerFactoryBean	**Proxy** • AOP • RMI • HttpInvoker	**Iterator** • CompositeIterator
Builder • EmbeddedDatabaseBuilder • AuthenticationManagerBuilder • UriComponentsBuilder • BeanDefinitionBuilder • MockMvcWebClientBuilder	**Adapter** • JpaVendorAdapter • HibernateJpaVendorAdapter • HandlerInterceptorAdapter • MessageListenerAdapter • SpringContextResourceAdapter • ClassPreProcessorAgentAdapter • RequestMappingHandlerAdapter • AnnotationMethodHandlerAdapter • WebMvcConfigurerAdapter	**Observer** • ApplicationContext • ApplicationEvent • ApplicationListener