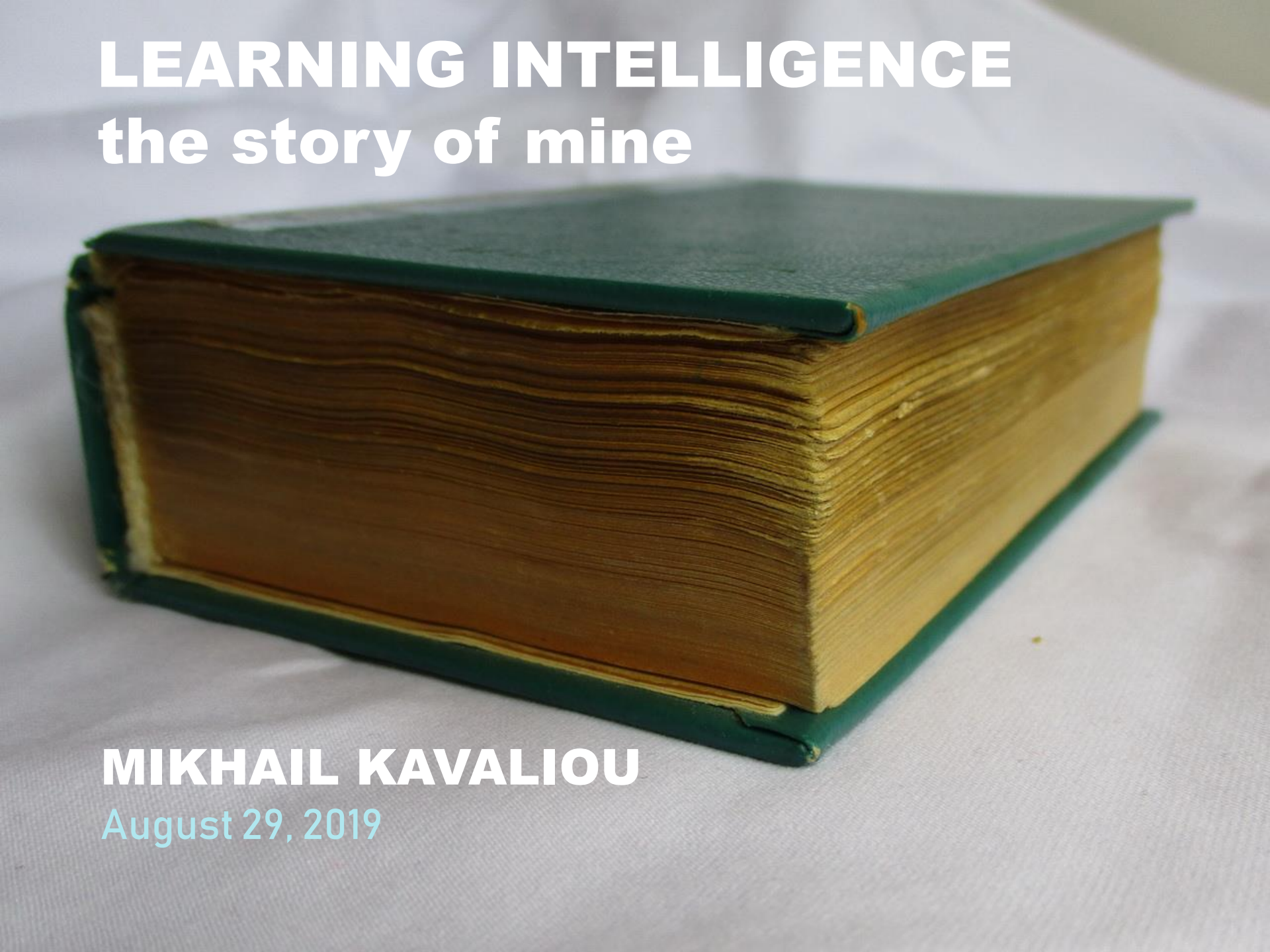
#Miniq18

VITEBSK

communities.by

LEARNING INTELLIGENCE
the story of mine



MIKHAIL KAVALIOU

August 29, 2019

Mikhail Kavaliou

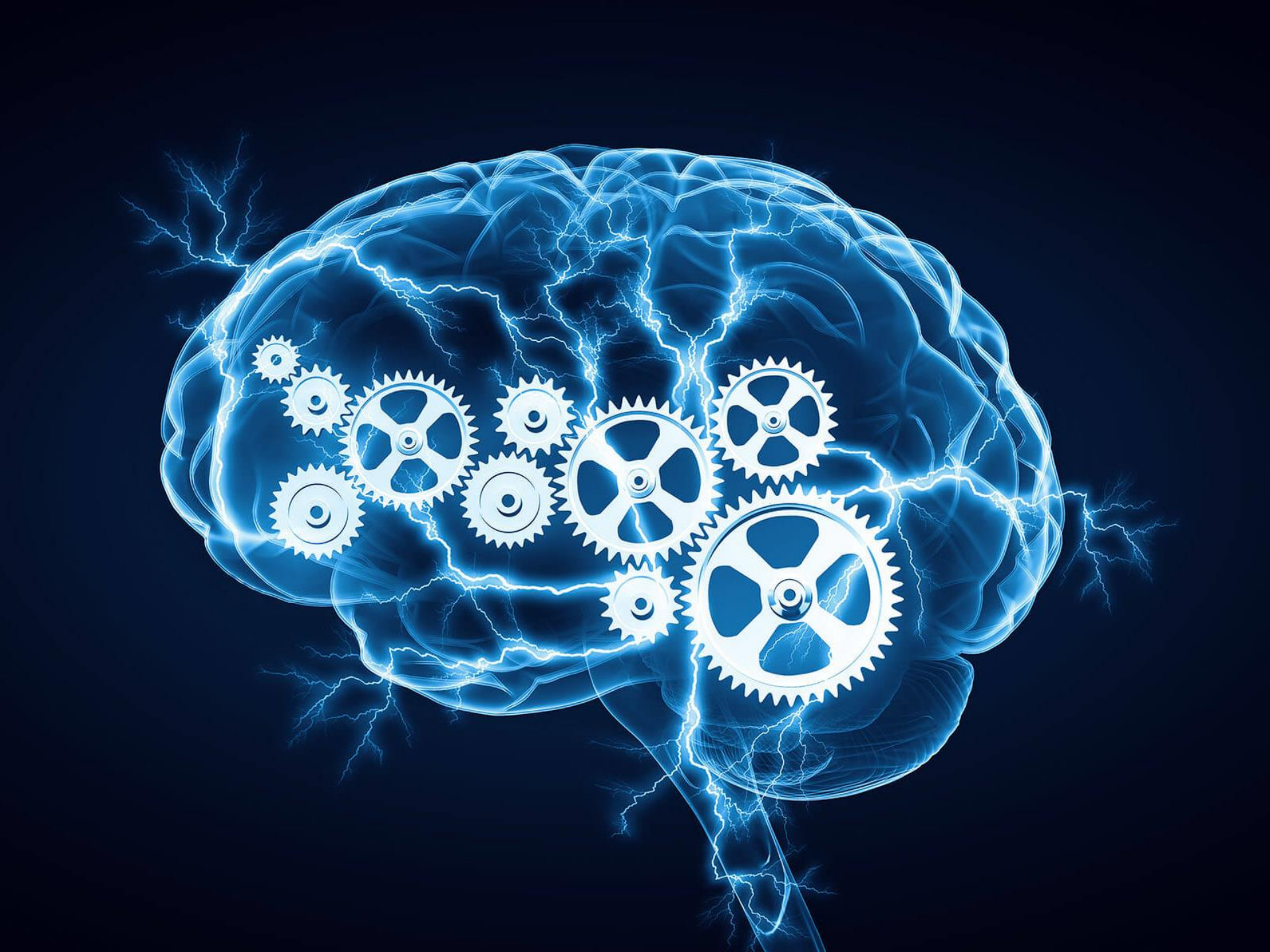
Senior Java Developer

6+ years in Java
3+ years in EPAM

MinuteForShop.com

AGENDA

- A story
- Timing and motivation
- Perception types
- Free learning



20 min

5 min

5 min

docker

20 min

Learning Styles

Audio

Visual

Kinaesthetic

container. The `BeanFactory` interface provides an advanced configuration mechanism capable of managing any type of object. `ApplicationContext` is a sub-interface of `BeanFactory`. It adds:

- Easier integration with Spring's AOP features
- Message resource handling (for use in internationalization)
- Event publication
- Application-layer specific contexts such as the `WebApplicationContext` for use in web applications.

In short, the `BeanFactory` provides the configuration framework and basic functionality, and the `ApplicationContext` adds more enterprise-specific functionality. The `ApplicationContext` is a complete superset of the `BeanFactory` and is used exclusively in this chapter in descriptions of Spring's IoC container. For more information on using the `BeanFactory` instead of the `ApplicationContext`, see The `BeanFactory`.

In Spring, the objects that form the backbone of …ur application and th… …are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, …semb… …rw… …e m…naged by a Spring IoC container. Otherwise, a bean is simply one of many objects in your applicati…. Bea… …p…nden…es among them, are reflected in the configuration metadata used by a container.

1.2. Container Overview

The `org.springframework.context.ApplicationContext` …presents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It lets you express the objects that compose your application and the rich interdependencies between those objects.

Several implementations of the `ApplicationContext` interface are supplied with Spring. In stand-alone applications, it is common to create an instance of `ClassPathXmlApplicationContext` or `FileSystemXmlApplicationContext`. While XML has been the traditional format for defining configuration metadata, you can instruct the container to use Java annotations or code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.

In most application scenarios, explicit user code is not required to instantiate one or more instances of a Spring IoC container. For example, in a web application scenario, a simple eight (or so) lines of boilerplate web descriptor XML in the `web.xml` file of the application typically suffices (see Convenient ApplicationContext Instantiation for Web Applications). If you use the Spring Tool Suite (an Eclipse-powered development environment), you can easily create this boilerplate configuration with a few mouse clicks or keystrokes.

There's no single Spring container. Spring comes with several container implementations that can be categorized into two distinct types. *Bean factories* (defined by the `org.springframework.beans.factory.BeanFactory` interface) are the simplest of containers, providing basic support for DI. *Application contexts* (defined by the `org.springframework.context.ApplicationContext` interface) build on the notion of a bean factory by providing application framework services, such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.

Although it's possible to work with Spring using either bean factories or application contexts, bean factories are often too low-level for most applications. Therefore, application contexts are preferred over bean factories. We'll focus on working with application contexts and not spend more time talking about bean factories.

1.2.1 Working with an application context

Spring comes with several flavors of application context. The three that you'll most likely encounter are

- `ClassPathXmlApplicationContext`—Loads a context definition from an XML file located in the classpath, treating context definition files as classpath resources.
- `FileSystemXmlApplicationContext`—Loads a context definition from an XML file in the file system.
- `XmlWebApplicationContext`—Loads context definitions from an XML file contained within a web application.

We'll talk more about `XmlWebApplicationContext` in chapter 7 when we discuss web-based Spring applications. For now, let's simply load the application context from the file system using `FileSystemXmlApplicationContext` or from the classpath using `ClassPathXmlApplicationContext`.

Loading an application context from the file system or from the classpath is similar to how you load beans into a bean factory. For example, here's how you'd load a `FileSystemXmlApplicationContext`:

```
ApplicationContext context = new
        FileSystemXmlApplicationContext("c:/foo.xml");
```

Similarly, you can load an application context from within the application's classpath using `ClassPathXmlApplicationContext`:

```
ApplicationContext context = new
        ClassPathXmlApplicationContext("foo.xml");
```

The difference between using `FileSystemXmlApplicationContext` and `ClassPathXmlApplicationContext` is that `FileSystemXmlApplicationContext` will look for foo.xml in a specific location within the file system, whereas `ClassPathXmlApplicationContext` will look for foo.xml anywhere in the classpath (including JAR files).

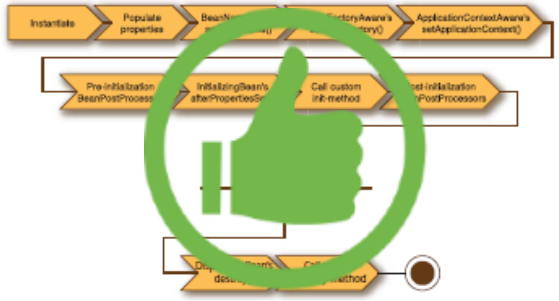


Figure 1.5 A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.

With an application context in hand, you can retrieve beans from the Spring container by calling the context's `getBean()` method.

Now that you know the basics of how to create a Spring container, let's take a closer look at the lifecycle of a bean in the bean container.

1.2.2 A bean's life

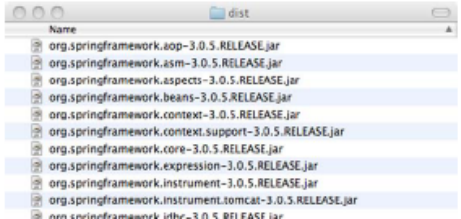
In a traditional Java application, the lifecycle of a bean is simple. Java's `new` keyword is used to instantiate the bean (or perhaps it's deserialized) and it's ready to use. Once the bean is no longer in use, it's eligible for garbage collection and eventually goes to the big bit bucket in the sky.

In contrast, the lifecycle of a bean within a Spring container is more elaborate. It's important to understand the lifecycle of a Spring bean, because you may want to take advantage of some of the opportunities that Spring offers to customize how a bean is created. Figure 1.5 shows the startup lifecycle of a typical bean as it's loaded into a Spring application context.

As you can see, a bean factory performs several setup steps before a bean is ready to use. Breaking down figure 1.5 in more detail:

1. Spring instantiates the bean.
2. Spring injects values and bean references into the bean's properties.
3. If the bean implements `BeanNameAware`, Spring passes the bean's ID to the `setBeanName()` method.
4. If the bean implements `BeanFactoryAware`, Spring calls the `setBeanFactory()` method, passing in the bean factory itself.

5. If the bean implements `ApplicationContextAware`, Spring will call the `setApplicationContext()` method, passing in a reference to the enclosing application context.
6. If any of the beans implement the `BeanPostProcessor` interface, Spring calls their `postProcessBeforeInitialization()` method.
7. If any beans implement the `InitializingBean` interface, Spring calls their `afterPropertiesSet()` method. Similarly, if the bean was declared with an `init-method`, then the specified initialization method will be called.
8. If there are any beans that implement `BeanPostProcessor`, Spring will call their `postProcessAfterInitialization()` method.



Подробнее о `XmlWebApplicationContext` будет рассказываться в главе 8 вместе с обсуждением веб-приложений на основе фреймворка Spring. А пока просто загрузим контекст приложения из файловой системы, используя `FileSystemXmlApplicationContext`, или из библиотеки классов (classpath), используя `ClassPathXmlApplicationContext`.

Загрузка контекста приложения из файловой системы или из библиотеки классов похожа на загрузку компонентов с использованием фабрики компонентов. Например, ниже показано, как можно использовать `FileSystemXmlApplicationContext`:

```
ApplicationContext context = new
        FileSystemXmlApplicationContext("c:/foo.xml");
```

Аналогично выполняется загрузка контекста приложения из библиотеки классов приложения с помощью `ClassPathXmlApplicationContext`:

```
ApplicationContext context = new
        ClassPathXmlApplicationContext("foo.xml");
```

Рис. 1.5. От создания до уничтожения в контейнере Spring компонент преодолевает несколько этапов. Spring позволяет настроить выполнение каждого из этапов

Различие между `FileSystemXmlApplicationContext` и `ClassPathXmlApplicationContext` состоит в том, что `FileSystemXmlApplicationContext` будет искать файл `foo.xml` в определенном месте, внутри файловой системы, тогда как `ClassPathXmlApplicationContext` будет искать `foo.xml` по всей библиотеке классов (включая JAR-файлы).

После загрузки контекста приложения извлекать компоненты из контейнера Spring можно с помощью метода `getBean()` контекста.

Теперь, после знакомства с основами создания контейнера Spring, познакомимся поближе с жизненным циклом компонента в контейнере.

1.2.2. Жизненный цикл компонента

В традиционных Java-приложениях жизненный цикл компонента довольно прост. Сначала компонент создается с помощью ключевого слова `new` (при этом, возможно, выполняется его десериализация), после чего он готов к использованию. Когда компонент перестает использоваться, он утилизируется сборщиком мусора и в конечном счете попадает в большой «битоприемник» на небесах.

Напротив, жизненный цикл компонента внутри контейнера Spring намного сложнее. Иметь представление о жизненном цикле компонента в контейнере Spring очень важно, потому что в этом случае появляются новые возможности управления процессом создания компонента. Рисунок 1.5 иллюстрирует начальные этапы жизненного цикла типичного компонента, которые он минует, прежде чем будет готов к использованию.

Как показано на рисунке, фабрика компонентов выполняет несколько подготовительных операций, перед тем как компонент будет готов к использованию. Исследуем рис. 1.5 более детально.

1. Spring создает экземпляр компонента.
2. Spring внедряет значения и ссылки на компоненты в свойства данного компонента.
3. Если компонент реализует интерфейс `BeanNameAware`, Spring передает идентификатор компонента методу `setBeanName()`.
4. Если компонент реализует интерфейс `BeanFactoryAware`, Spring вызывает метод `setBeanFactory()`, передавая ему саму фабрику компонентов.
5. Если компонент реализует интерфейс `ApplicationContextAware`, Spring вызывает метод `setApplicationContext()`, передавая ему ссылку на вмещающий контекст приложения.

Евгений Борисов — Spring-потрошитель, часть 2

69 289 просмотров

1,2 ТЫС. 3 ПОДЕЛИТЬСЯ СОХРАНИТЬ

JUG .ru
Опубликовано: 3 июл. 2014 г.

ПОДПИСАТЬСЯ 27 ТЫС.

Java-конференция Joker 2019:
25-26 октября, Санкт-Петербург.
Подробности и билеты: http://bit.ly/2uO0mzJ
ЕЩЁ

27 комментариев УПОРЯДОЧИТЬ

Оставьте комментарий

Igor Petrov 3 года назад
"Какая разница между Spring, String и Swing?")))
29 ОТВЕТИТЬ
Показать ответ

Phasers

A Phaser provides functionality similar to the CyclicBarrier and CountDow Lat nizers. It provides the following features:

Phaser is reusable.

The number of parties to synchronize on a Phaser can change dynamically. In a CyclicBarrier, the number of parties is fixed at the time the barrier is created.

A Phaser has an associated phase number, which starts at zero. When all registered parties arrive at a Phaser, the Phaser advances to the next phase and the phase number is incremented by one. The maximum value of the phase number is Integer.MAX_VALUE. After its maximum value, the phase number restarts at zero.

A Phaser has a termination state. All synchronization methods called on a Phaser in a termination state return immediately without waiting for an advance.

A Phaser has three types of parties count: a registered parties count, an arrived parties count, and an unarrived parties count.

The registered parties count is the number of parties that are registered for synchronization. The arrived parties count is the number of parties that have arrived at the current phase of the phaser.

The unarrived parties count is the number of parties that have not yet arrived at the current phase of the phaser.

When the last party arrives, the phaser advances to the next phase.

Optionally, a Phaser lets you execute a phaser action when all registered parties arrive at the

- **int arriveAndDeregister()** — сообщает о завершении всех фаз стороной и снимает ее с регистрации. Возвращает номер текущей фазы;
- **int awaitAdvance(int phase)** — если phase равно номеру текущей фазы, приостанавливает вызвавший его поток до её окончания. В противном случае сразу возвращает аргумент.

```
arriveAndAwaitAdvance();
arrive();
awaitAdvance(i);
arriveAndDeregister();
register();
```

phase = i

parties = 5

arrived = 0

Phaser

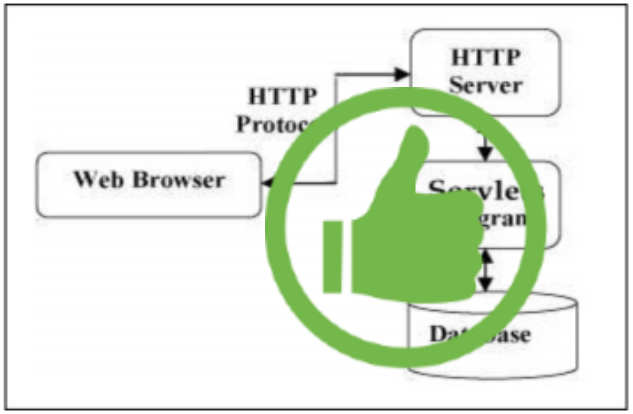
Using Servlets, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.

Java Servlets often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). But Servlets offer several advantages in comparison with the CGI.

- Performance is significantly better.
- Servlets execute within the address sp ce of a eb se r. It is not necessary to create a separate process to handle each clie reques
- Servlets are platform-independent b caus itte in Java.
- Java security manager on the ser er se of restrictions to protect the resources on a server machine. So serv ts are trusted
- The full functionality of the Java class libraries is available to a servlet. It can communicate with applets, databases, or other software via the sockets and RMI mechanisms that you have seen already.

Servlets Architecture

The following diagram shows the position of Servlets in a Web Application.

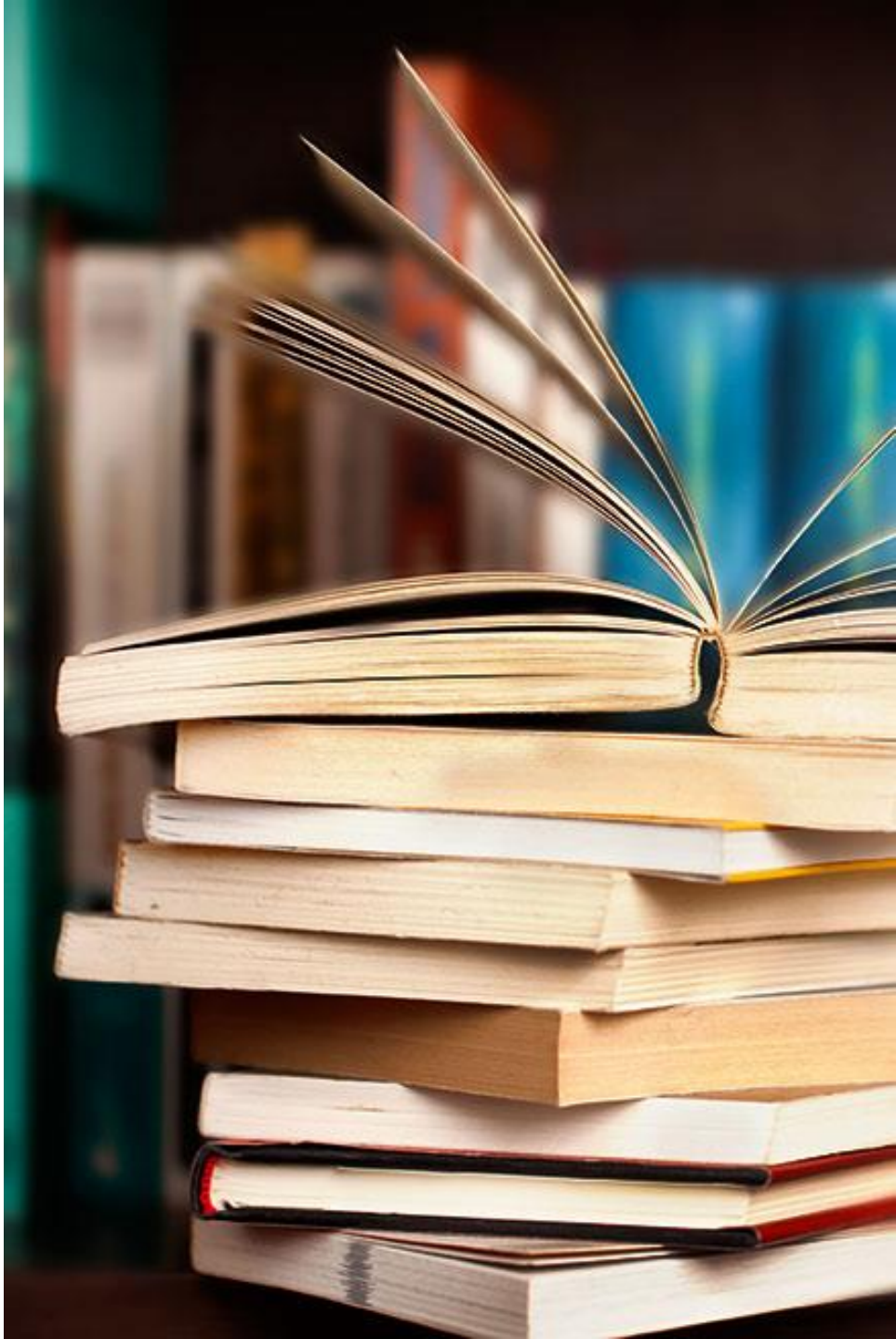


Servlets Tasks

Servlets perform the following major tasks −

SUMMARY

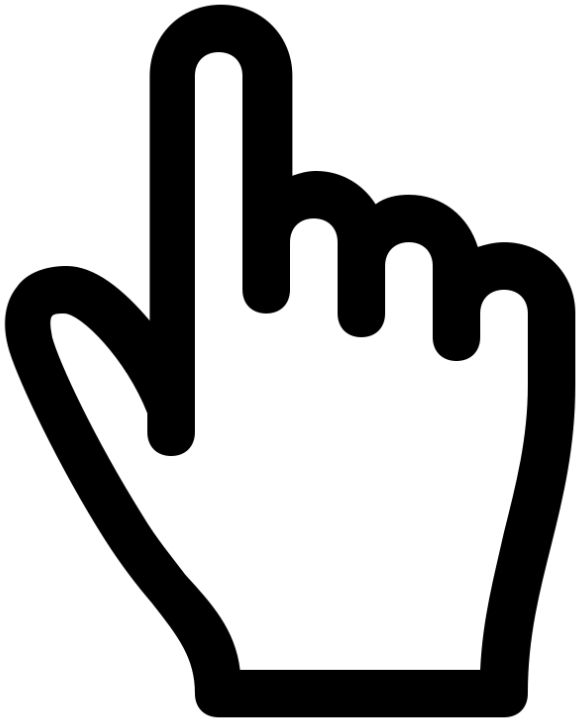
1. Timing
2. Perception
3. Resources
4. Priorities



USED RESOURCES

- https://spring.io/docs
- https://www.manning.com/books/spring-in-action-fifth-edition
- https://www.ozon.ru/context/detail/id/31239365/
- https://www.youtube.com/watch?v=BmBr5diz8WA
- https://habr.com/ru/post/277669/
- https://www.tutorialspoint.com

https://github.com/MikhailKavaliou/LearningIntelligence




Never
stop
learning