# 11. Validation, Formatting, and Type Conversion

-org.springframework.core.convert: convert any Java types within Spring application.

## 11.1 Converting String Values using PropertyEditors

-Record:

A close up of words

AI-generated content may be incorrect.

-Spring Configuration with Error:

A computer screen shot of a computer code

AI-generated content may be incorrect.

A screen shot of a computer code

AI-generated content may be incorrect.

-To fix this, We need to tell Spring how to do the conversion of text representations of calendar dates to LocalDate. We can do by an extension of PropertyEditorSupport:

A screen shot of a computer code

AI-generated content may be incorrect.

-Registry custom property editor:

A computer code with black text

AI-generated content may be incorrect.

A computer code with black text

AI-generated content may be incorrect.

## 11.2 Introduce Spring Type Conversion

### 11.2.1 Implement a Custom Converter

-Implement Converter<S,T>



A black text on a white background

AI-generated content may be incorrect.

-To use this converter instead of PropertyEditor, we need to configure ConversionService interface:

A computer screen shot of a program

AI-generated content may be incorrect.

+Declare conversionService bean with class ConversionServiceFactoryBean. This type of bean groups multiple conversion services: if no conversions service bean is defined, Spring will use PropertyEditor-based system.

-By default, type conversion service supports conversions between common types: string, number, enum, collection, map.

### 11.2.2 Converting between Arbitrary Types

A screenshot of a computer program

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

-Spring MVC makes heavy use of conversion service (and Formatter SPI):

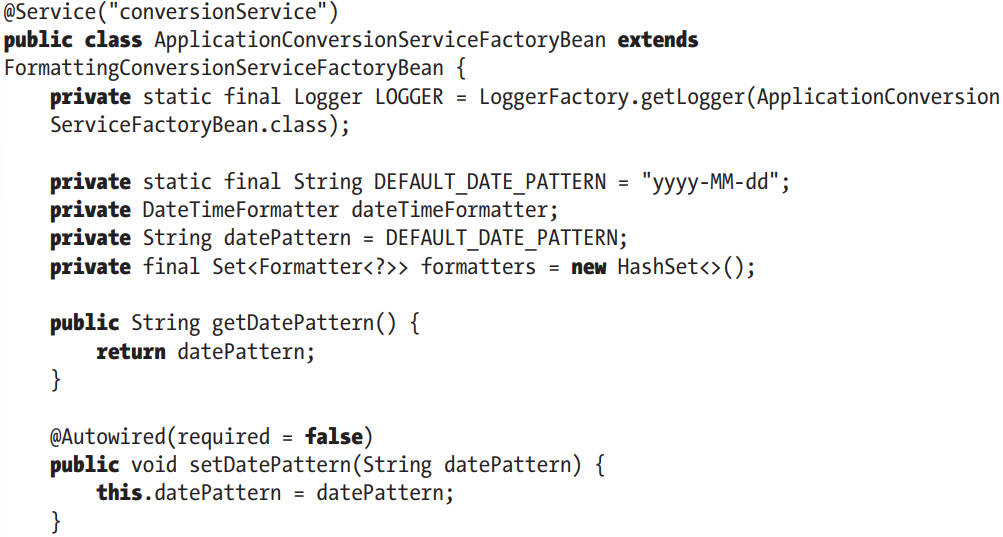
+@EnableWebMvc automatically register all default converters (StringToArrayConverter, StringToBooleanConverter, StringToLocaleConverter... in org.springframework.core.convert.support) and formatter (CurrencyStyleFormatter, DataFormatter, AbstractNumberFormatter in org.springframework.format)

## 11.3 Field Formatting in Spring

-Formatter SPI: configure filed-formatting aspects

-The main interface is Formatter<T>. It provides implementations: CurrencyStyleFormatter, DateFormatter, AbstractNumberFormatter, PercentStyleFormatter

-We extend FormattingConversionServiceFactoryBean and provide custom formatter. This is a factory class that provides access to FormattingConversionService



A screenshot of a computer code

AI-generated content may be incorrect.

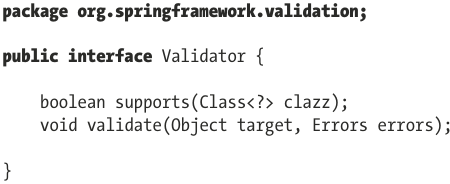
## 11.4 Validation in Spring

-Validation rules applied on domain objects ensure all business data is well structured and fulfills all business definitions.

-The ideal case is that all validation rules are maintained in centralized location, and the same set of rules are applied to same type of data, no matter which source the data comes from.

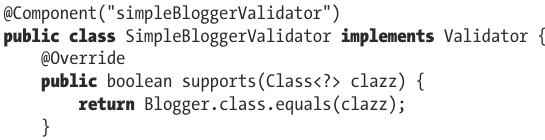
-**data binding**: extract data from source and perform conversion from sth to desired type. When data binding complete and domain object constructed, validation will be applied to object, and any errors will be returned and displayed to user. If validation succeeds, the object will be persisted to database

-Spring supports 2 main types of validation: Validator interface and via Spring’s support of JSR-349



### 11.4.1 Use Spring Validator in Spring

-Implement Validator

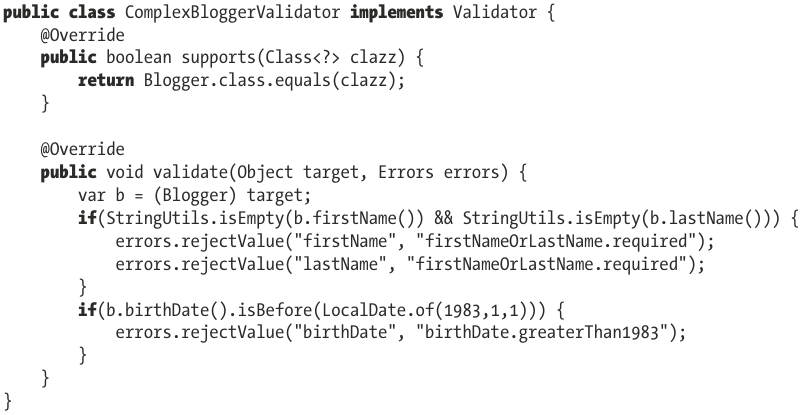




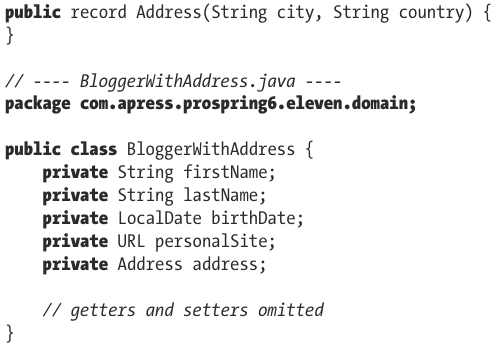
-Note: Converters, formatters and validators are necessary components for applications that handle user-provided data.

-More complex:

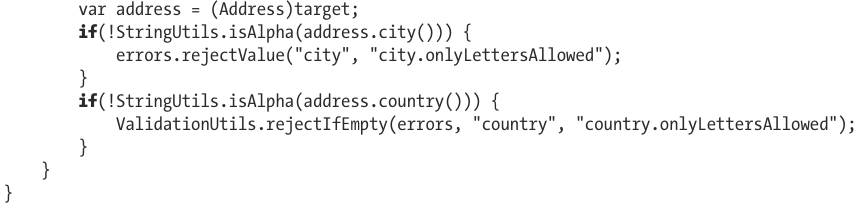


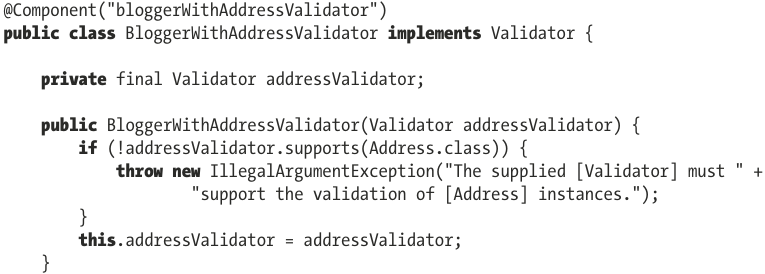


-Validator interface can be implemented to validate complex objects by reusing validation logic for nested objects









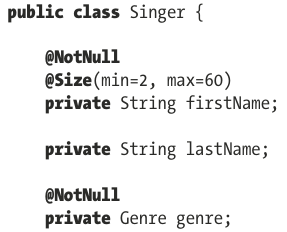


### 11.4.2 Use JSR-349/Jakarta Bean Validation

### 11.4.3 Dependencies

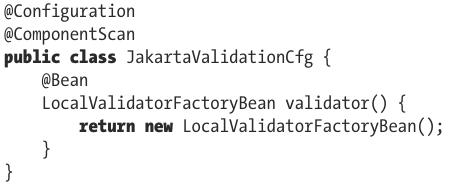
-hibernate-validator

### 11.4.4 Define Validation Constraints on Domain Object Properties



### 11.4.5 Configure Bean Validation Support in Spring

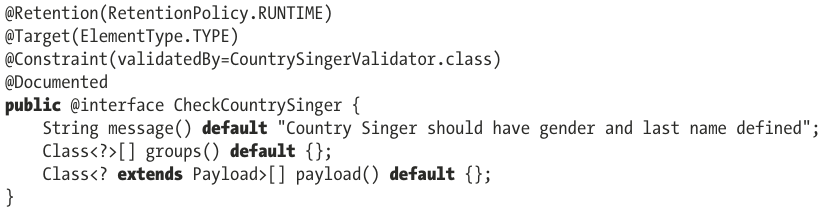
-To configure support of Bean Validation API in ApplicationContext, we define a LocalValidatorFactoryBean, then register all validation services needed.





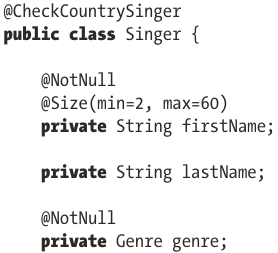
### 11.4.6 Create a Custom Validator

-Annotation interface for Custom Validator



-Validation logic:





### 11.4.7 Use AssertTrue for Custom Validation

-See in book

### 11.4.8 Decide which Validation API to use

-See in book

### 11.4.9 Configure Validation in Spring Boot App

-spring-boot-starter-validation

# 12. Task Execution and Scheduling

-Task scheduling is composed mainly of 3 parts:



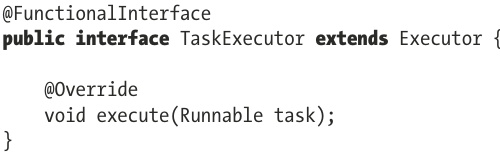
-Java app is described by code that JVM can run on one or multiple threads, and one of threads is non-daemon thread that calls main() of main class.

-In Java, the class to model an execution thread is java.lang.Thread. It can be created by extending this class and overriding run(). The resulting instance models an execution thread must be started by start()

-Another way to create threads: implement **Runnable**. It provides a common protocol for objects that wish to execute code. This means Runnable instances can be created and passed to some components (executors) that execute code in the way they were configured: sequentially, parallel, using threads provided by a thread pool.

-In Java app, a task is any instance of type Runnable

-**Executor** interface represents abstraction for **asynchronous task execution**. In **Spring**, there is an interface that extends this one: **TaskExecutor**. This interface is necessary for backward compatibility with JDK in Spring.



## 12.1 Task Executing in Java

-In Java there are a few Executor implementations.

-**Thread pools** provide improved performance when executing large number of asynchronous tasks. They provide a way of bounding and managing the resources, including threads when executing tasks.

-ThreadPoolExecutor maintains basic statistics as the number of active and completed tasks

-Example: sorting tasks

A diagram of a computer program

AI-generated content may be incorrect.

+Running implementation in parallel managed by **ThreadPoolExecutor** instance. This class is an implementation of **ExecutorService** that executes each submitted task using one of several pooled threads, configured using **Executors** factory methods. The **size** of **thread pool** and its **maximum capacity** are provided as **arguments**. The **tasks** are provided as an argument of type that implements **BlockingQueue<Runnable>** interface, which avoid the same task being submitted more than once for execution, by being thread-safe.

+ThreadPoolMonitor is a custom extension of Thread to print statistics for a running ThreadPoolExecutor

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer code

AI-generated content may be incorrect.

-Example: Generate array containing 100000 elements with values between 0 and 500000 that is handed over to be sorted in parallel by tasks. These tasks are managed and executed by a ThreadPoolExecutor instance, which is monitored by ThreadPoolMonitor.

A screen shot of a computer code

AI-generated content may be incorrect.

A screen shot of a computer code

AI-generated content may be incorrect.

A close up of text

AI-generated content may be incorrect.

A close up of black text

AI-generated content may be incorrect.

## 12.2 Task Executing in Spring

-Spring’s **TaskExecutor** interface.

A close up of text

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

-Example:

A computer code with black text

AI-generated content may be incorrect.

+Provide a declaration for TaskExecutor bean:

A computer code with black text

AI-generated content may be incorrect.

A computer screen shot of a program code

AI-generated content may be incorrect.

+@EnableAsync: enables Spring’s asynchronous method execution capability. Spring will search for an associated thread pool definition: a unique TaskExecutor bean in context or Executor bean named taskExecutor. If none is found, SimpleAsyncTaskExecutor will be used to process async method invocations.

## 12.3 Task Scheduling in Spring

-There are many ways to trigger the execution of a task in Spring app. One way is to trigger a job externally from a scheduling system that already exists in application deployment environments. Example: commercial system (Control-M, CA AutoSys), Linux/Unix platform (crontab scheduler). The job triggering can be done by sending a RESTful-WS request to Spring and having Spring MVC controller trigger the task.

-Another way is **task scheduling support**. Spring provides 3 options of task scheduling:

A close-up of a white background

AI-generated content may be incorrect.

### 12.3.1 Introduce Spring TaskScheduler Abstraction

-TaskScheduler abstraction provides methods for scheduling tasks to run at some point and has 3 participants:

A text on a white background

AI-generated content may be incorrect.

A close-up of a computer screen

AI-generated content may be incorrect.

-Relationship between Trigger, TaskScheduler and task that implement Runnable:

A diagram of a task scheduler

AI-generated content may be incorrect.

### 12.3.2 Explore a Sample Task

-Service:

A black text on a white background

AI-generated content may be incorrect.

-Configuration enable task scheduling:

A black text on a white background

AI-generated content may be incorrect.

**-@EnableScheduling**: enable detection of @Scheduled on any bean. This annotation look for: unique TaskScheduler bean in context or a TaskScheduler bean named taskScheduler or a ScheduledExecutorService bean. If none is found, a local single-threaded default scheduler will be created an used to execute scheduled tasks.

A computer screen shot of a program

AI-generated content may be incorrect.

+We search for scheduler bean named taskScheduler (value of DEFAULT\_TASK\_SCHEDULER\_BEAN\_NAME)

-Scheduling task in Spring is implemented by **proxying**. @EnableScheduling adds SheduledAnnotationBeanPostProcessor to context that picks up @Scheduled methods. These methods are invoked by TaskSheduler according to fixedRate, fixedDelay, cron.. @Scheduled methods are executed in a thread pool.

-2 ways to **configure a task scheduler**: make TaskSchedulingConfig class **implement** **SchedulingConfigurer** and **override configureTasks()** to set up a custom task scheduler, or **declare** a **TaskScheduler** custom **bean**.

A close up of a computer screen

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

+Declare TaskScheduler bean directly:

A computer screen shot of a program

AI-generated content may be incorrect.

### 12.3.3 Asynchronous Task Execution in Spring

# 13. Spring Remoting

## 13.1 Communication via HTTP using Spring REST

## 13.2 Use JMS in Spring

### 13.2.1 Working with Apache ActiveMQ Artemis

## 13.3 Use Spring for Apache Kafka