10. 스프링 타입 컨버터

#1.인강/5. 스프링 MVC 2/강의#

프로젝트 생성

스프링 부트 스타터 사이트로 이동해서 스프링 프로젝트 생성

https://start.spring.io

- 프로젝트 선택
  - Project: Gradle Project
  - Language: Java
  - Spring Boot: 2.4.x
- Project Metadata
  - Group: hello
  - Artifact: typeconverter
  - Name: typeconverter
  - Package name: **hello.typeconverter**
  - Packaging: **Jar**
  - Java: 11

- Dependencies: **Spring Web**, **Lombok** , **Thymeleaf**

build.gradle

```
plugins {
    id 'org.springframework.boot' version '2.4.5'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
    useJUnitPlatform()
}
```

- 동작 확인
  - 기본 메인 클래스 실행(`TypeconverterApplication.main()`)
  - http://localhost:8080 호출해서 Whitelabel Error Page가 나오면 정상 동작

스프링 타입 컨버터 소개

문자를 숫자로 변환하거나, 반대로 숫자를 문자로 변환해야 하는 것 처럼 애플리케이션을 개발하다 보면 타입을 변환해야 하는 경우가 상당히 많다.

다음 예를 보자.

HelloController - 문자 타입을 숫자 타입으로 변경

```
package hello.typeconverter.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.servlet.http.HttpServletRequest;

@RestController
public class HelloController {

    @GetMapping("/hello-v1")
    public String helloV1(HttpServletRequest request) {
        String data = request.getParameter("data"); //문자 타입 조회
        Integer intValue = Integer.valueOf(data); //숫자 타입으로 변경
        System.out.println("intValue = " + intValue);
        return "ok";
    }
}
```

실행

http://localhost:8080/hello-v1?data=10

분석

`String data = request.getParameter("data")`

HTTP 요청 파라미터는 모두 문자로 처리된다. 따라서 요청 파라미터를 자바에서 다른 타입으로 변환해서 사용하고 싶으면 다음과 같이 숫자 타입으로 변환하는 과정을 거쳐야 한다.

`Integer intValue = Integer.valueOf(data)`

이번에는 스프링 MVC가 제공하는 `@RequestParam`을 사용해보자.

HelloController - 추가

```
@GetMapping("/hello-v2")
public String helloV2(@RequestParam Integer data) {
    System.out.println("data = " + data);
    return "ok";
```

```
}
```

실행

http://localhost:8080/hello-v2?data=10

앞서 보았듯이 HTTP 쿼리 스트링으로 전달하는 `data=10` 부분에서 10은 숫자 10이 아니라 문자 10이다.
스프링이 제공하는 `@RequestParam`을 사용하면 이 문자 10을 `Integer` 타입의 숫자 10으로 편리하게 받을 수 있다.
이것은 스프링이 중간에서 타입을 변환해주었기 때문이다.

이러한 예는 `@ModelAttribute`, `@PathVariable`에서도 확인할 수 있다.

@ModelAttribute 타입 변환 예시

```
@ModelAttribute UserData data

class UserData {
  Integer data;
}
```

`@RequestParam`와 같이, 문자 `data=10`을 숫자 10으로 받을 수 있다.

@PathVariable 타입 변환 예시

```
/users/{userId}
@PathVariable("userId") Integer data
```

URL 경로는 문자다. /users/10 ➜ 여기서 10도 숫자 10이 아니라 그냥 문자 "10"이다. data를 `Integer` 타입으로 받을 수 있는 것도 스프링이 타입 변환을 해주기 때문이다.

스프링의 타입 변환 적용 예

- 스프링 MVC 요청 파라미터
  - `@RequestParam`, `@ModelAttribute`, `@PathVariable`
- `@Value` 등으로 YML 정보 읽기
- XML에 넣은 스프링 빈 정보를 변환
- 뷰를 렌더링 할 때

스프링과 타입 변환

이렇게 타입을 변환해야 하는 경우는 상당히 많다. 개발자가 직접 하나하나 타입 변환을 해야 한다면, 생각만 해도 괴로울 것이다.
스프링이 중간에 타입 변환기를 사용해서 타입을 `String` ➜ `Integer`로 변환해주었기 때문에 개발자는 편리하게 해

당 타입을 바로 받을 수 있다. 앞에서는 문자를 숫자로 변경하는 예시를 들었지만, 반대로 숫자를 문자로 변경하는 것도 가능하고, `Boolean` 타입을 숫자로 변경하는 것도 가능하다. 만약 개발자가 새로운 타입을 만들어서 변환하고 싶으면 어떻게 하면 될까?

컨버터 인터페이스

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {
  T convert(S source);
}
```

스프링은 확장 가능한 컨버터 인터페이스를 제공한다.
개발자는 스프링에 추가적인 타입 변환이 필요하면 이 컨버터 인터페이스를 구현해서 등록하면 된다.
이 컨버터 인터페이스는 모든 타입에 적용할 수 있다. 필요하면 X ➜ Y 타입으로 변환하는 컨버터 인터페이스를 만들고, 또 Y ➜ X 타입으로 변환하는 컨버터 인터페이스를 만들어서 등록하면 된다.
예를 들어서 문자로 `"true"` 가 오면 `Boolean` 타입으로 받고 싶으면 `String` ➜ `Boolean` 타입으로 변환되도록 컨버터 인터페이스를 만들어서 등록하고, 반대로 적용하고 싶으면 `Boolean` ➜ `String` 타입으로 변환되도록 컨버터를 추가로 만들어서 등록하면 된다.

> **참고**
> 과거에는 `PropertyEditor` 라는 것으로 타입을 변환했다. `PropertyEditor` 는 동시성 문제가 있어서 타입을 변환할 때 마다 객체를 계속 생성해야 하는 단점이 있다. 지금은 `Converter` 의 등장으로 해당 문제들이 해결되었고, 기능 확장이 필요하면 `Converter` 를 사용하면 된다.

실제 코드를 통해서 타입 컨버터를 이해해보자.

타입 컨버터 - Converter

타입 컨버터를 어떻게 사용하는지 코드로 알아보자.

타입 컨버터를 사용하려면 `org.springframework.core.convert.converter.Converter` 인터페이스를 구현하면 된다.

> **주의**
> `Converter`라는 이름의 인터페이스가 많으니 조심해야 한다.
> `org.springframework.core.convert.converter.Converter`를 사용해야 한다.

컨버터 인터페이스

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {
    T convert(S source);
}
```

먼저 가장 단순한 형태인 문자를 숫자로 바꾸는 타입 컨버터를 만들어보자.

StringToIntegerConverter - 문자를 숫자로 변환하는 타입 컨버터

```
package hello.typeconverter.converter;

import lombok.extern.slf4j.Slf4j;
import org.springframework.core.convert.converter.Converter;

@Slf4j
public class StringToIntegerConverter implements Converter<String, Integer> {

    @Override
    public Integer convert(String source) {
        log.info("convert source={}", source);
        return Integer.valueOf(source);
    }
}
```

`String` ➜ `Integer`로 변환하기 때문에 소스가 `String`이 된다. 이 문자를 `Integer.valueOf(source)`를 사용해서 숫자로 변경한 다음에 변경된 숫자를 반환하면 된다.

IntegerToStringConverter - 숫자를 문자로 변환하는 타입 컨버터

```
package hello.typeconverter.converter;

import lombok.extern.slf4j.Slf4j;
import org.springframework.core.convert.converter.Converter;
```

```
@Slf4j
public class IntegerToStringConverter implements Converter<Integer, String> {
    @Override
    public String convert(Integer source) {
        log.info("convert source={}", source);
        return String.valueOf(source);
    }
}
```

이번에는 숫자를 문자로 변환하는 타입 컨버터이다. 앞의 컨버터와 반대의 일을 한다. 이번에는 숫자가 입력되기 때문에 소스가 `Integer`가 된다. `String.valueOf(source)`를 사용해서 문자로 변경한 다음 변경된 문자를 반환하면 된다.

테스트 코드를 통해서 타입 컨버터가 어떻게 동작하는지 확인해보자.

ConverterTest - 타입 컨버터 테스트 코드

```
package hello.typeconverter.converter;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.*;

class ConverterTest {

    @Test
    void stringToInteger() {
        StringToIntegerConverter converter = new StringToIntegerConverter();
        Integer result = converter.convert("10");
        assertThat(result).isEqualTo(10);
    }

    @Test
    void integerToString() {
        IntegerToStringConverter converter = new IntegerToStringConverter();
        String result = converter.convert(10);
        assertThat(result).isEqualTo("10");
    }

}
```

사용자 정의 타입 컨버터

타입 컨버터 이해를 돕기 위해 조금 다른 컨버터를 준비해보았다.

`127.0.0.1:8080`과 같은 IP, PORT를 입력하면 IpPort 객체로 변환하는 컨버터를 만들어보자.

IpPort

```
package hello.typeconverter.type;

import lombok.EqualsAndHashCode;
import lombok.Getter;

@Getter
@EqualsAndHashCode
public class IpPort {

    private String ip;
    private int port;

    public IpPort(String ip, int port) {
        this.ip = ip;
        this.port = port;
    }

}
```

롬복의 `@EqualsAndHashCode`를 넣으면 모든 필드를 사용해서 `equals()`, `hashcode()`를 생성한다. 따라서 모든 필드의 값이 같다면 `a.equals(b)`의 결과가 참이 된다.

StringToIpPortConverter - 컨버터

```
package hello.typeconverter.converter;

import hello.typeconverter.type.IpPort;
import lombok.extern.slf4j.Slf4j;
import org.springframework.core.convert.converter.Converter;

@Slf4j
public class StringToIpPortConverter implements Converter<String, IpPort> {
```

```
    @Override
    public IpPort convert(String source) {
        log.info("convert source={}", source);
        String[] split = source.split(":");
        String ip = split[0];
        int port = Integer.parseInt(split[1]);

        return new IpPort(ip, port);
    }
}
```

`127.0.0.1:8080` 같은 문자를 입력하면 `IpPort` 객체를 만들어 반환한다.

IpPortToStringConverter

```
package hello.typeconverter.converter;

import hello.typeconverter.type.IpPort;
import lombok.extern.slf4j.Slf4j;
import org.springframework.core.convert.converter.Converter;

@Slf4j
public class IpPortToStringConverter implements Converter<IpPort, String> {

    @Override
    public String convert(IpPort source) {
        log.info("convert source={}", source);
        return source.getIp() + ":" + source.getPort();
    }
}
```

`IpPort` 객체를 입력하면 `127.0.0.1:8080` 같은 문자를 반환한다.

ConverterTest - IpPort 컨버터 테스트 추가

```
@Test
void stringToIpPort() {
    StringToIpPortConverter converter = new StringToIpPortConverter();
    String source = "127.0.0.1:8080";
    IpPort result = converter.convert(source);
    assertThat(result).isEqualTo(new IpPort("127.0.0.1", 8080));
}
```

```java
@Test
void ipPortToString() {
    IpPortToStringConverter converter = new IpPortToStringConverter();
    IpPort source = new IpPort("127.0.0.1", 8080);
    String result = converter.convert(source);
    assertThat(result).isEqualTo("127.0.0.1:8080");
}
```

타입 컨버터 인터페이스가 단순해서 이해하기 어렵지 않을 것이다.
그런데 이렇게 타입 컨버터를 하나하나 직접 사용하면, 개발자가 직접 컨버팅 하는 것과 큰 차이가 없다.
타입 컨버터를 등록하고 관리하면서 편리하게 변환 기능을 제공하는 역할을 하는 무언가가 필요하다.

> **참고**
> 스프링은 용도에 따라 다양한 방식의 타입 컨버터를 제공한다.
>
> `Converter` ➜ 기본 타입 컨버터
> `ConverterFactory` ➜ 전체 클래스 계층 구조가 필요할 때
> `GenericConverter` ➜ 정교한 구현, 대상 필드의 애노테이션 정보 사용 가능
> `ConditionalGenericConverter` ➜ 특정 조건이 참인 경우에만 실행
>
> 자세한 내용은 공식 문서를 참고하자.
> https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#core-convert

> **참고**
> 스프링은 문자, 숫자, 불린, Enum등 일반적인 타입에 대한 대부분의 컨버터를 기본으로 제공한다. IDE에서 `Converter`, `ConverterFactory`, `GenericConverter`의 구현체를 찾아보면 수 많은 컨버터를 확인할 수 있다.

컨버전 서비스 - ConversionService

이렇게 타입 컨버터를 하나하나 직접 찾아서 타입 변환에 사용하는 것은 매우 불편하다. 그래서 스프링은 개별 컨버터를 모아두고 그것들을 묶어서 편리하게 사용할 수 있는 기능을 제공하는데, 이것이 바로 컨버전 서비스(`ConversionService`)이다.

ConversionService 인터페이스

```
package org.springframework.core.convert;

import org.springframework.lang.Nullable;

public interface ConversionService {

    boolean canConvert(@Nullable Class<?> sourceType, Class<?> targetType);
    boolean canConvert(@Nullable TypeDescriptor sourceType, TypeDescriptor
targetType);

    <T> T convert(@Nullable Object source, Class<T> targetType);
    Object convert(@Nullable Object source, @Nullable TypeDescriptor sourceType,
TypeDescriptor targetType);

}
```

컨버전 서비스 인터페이스는 단순히 컨버팅이 가능한가? 확인하는 기능과, 컨버팅 기능을 제공한다.

사용 예를 확인해보자.

ConversionServiceTest - 컨버전 서비스 테스트 코드

```
package hello.typeconverter.converter;

import hello.typeconverter.type.IpPort;
import org.junit.jupiter.api.Test;
import org.springframework.core.convert.support.DefaultConversionService;

import static org.assertj.core.api.Assertions.*;

public class ConversionServiceTest {

    @Test
    void conversionService() {
        //등록
        DefaultConversionService conversionService = new
DefaultConversionService();
        conversionService.addConverter(new StringToIntegerConverter());
        conversionService.addConverter(new IntegerToStringConverter());
        conversionService.addConverter(new StringToIpPortConverter());
        conversionService.addConverter(new IpPortToStringConverter());

        //사용
        assertThat(conversionService.convert("10",
```

```
Integer.class)).isEqualTo(10);
        assertThat(conversionService.convert(10, String.class)).isEqualTo("10");

        IpPort ipPort = conversionService.convert("127.0.0.1:8080",
IpPort.class);
        assertThat(ipPort).isEqualTo(new IpPort("127.0.0.1", 8080));

        String ipPortString = conversionService.convert(new IpPort("127.0.0.1",
8080), String.class);
        assertThat(ipPortString).isEqualTo("127.0.0.1:8080");

    }
}
```

`DefaultConversionService`는 `ConversionService` 인터페이스를 구현했는데, 추가로 컨버터를 등록하는 기능도 제공한다.

등록과 사용 분리

컨버터를 등록할 때는 `StringToIntegerConverter` 같은 타입 컨버터를 명확하게 알아야 한다. 반면에 컨버터를 사용하는 입장에서는 타입 컨버터를 전혀 몰라도 된다. 타입 컨버터들은 모두 컨버전 서비스 내부에 숨어서 제공된다. 따라서 타입을 변환을 원하는 사용자는 컨버전 서비스 인터페이스에만 의존하면 된다. 물론 컨버전 서비스를 등록하는 부분과 사용하는 부분을 분리하고 의존관계 주입을 사용해야 한다.

컨버전 서비스 사용

`Integer value = conversionService.convert("10", Integer.class)`

인터페이스 분리 원칙 - ISP(Interface Segregation Principle)

인터페이스 분리 원칙은 클라이언트가 자신이 이용하지 않는 메서드에 의존하지 않아야 한다.

`DefaultConversionService`는 다음 두 인터페이스를 구현했다.

- `ConversionService`: 컨버터 사용에 초점
- `ConverterRegistry`: 컨버터 등록에 초점

이렇게 인터페이스를 분리하면 컨버터를 사용하는 클라이언트와 컨버터를 등록하고 관리하는 클라이언트의 관심사를 명확하게 분리할 수 있다. 특히 컨버터를 사용하는 클라이언트는 `ConversionService`만 의존하면 되므로, 컨버터를 어떻게 등록하고 관리하는지는 전혀 몰라도 된다. 결과적으로 컨버터를 사용하는 클라이언트는 꼭 필요한 메서드만 알게된다. 이렇게 인터페이스를 분리하는 것을 `ISP`라 한다.

ISP 참고: https://ko.wikipedia.org/wiki/%EC%9D%B8%ED%84%B0%ED%8E%98%EC%9D%B4%EC%8A%A4_%EB%B6%84%EB%A6%AC_%EC%9B%90%EC%B9%99

스프링은 내부에서 `ConversionService`를 사용해서 타입을 변환한다. 예를 들어서 앞서 살펴본 `@RequestParam` 같은 곳에서 이 기능을 사용해서 타입을 변환한다.
이제 컨버전 서비스를 스프링에 적용해보자.

스프링에 Converter 적용하기

웹 애플리케이션에 `Converter`를 적용해보자.

WebConfig - 컨버터 등록

```
package hello.typeconverter;

import hello.typeconverter.converter.IntegerToStringConverter;
import hello.typeconverter.converter.IpPortToStringConverter;
import hello.typeconverter.converter.StringToIntegerConverter;
import hello.typeconverter.converter.StringToIpPortConverter;
import org.springframework.context.annotation.Configuration;
import org.springframework.format.FormatterRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        registry.addConverter(new StringToIntegerConverter());
        registry.addConverter(new IntegerToStringConverter());
        registry.addConverter(new StringToIpPortConverter());
        registry.addConverter(new IpPortToStringConverter());
    }
}
```

스프링은 내부에서 `ConversionService`를 제공한다. 우리는 `WebMvcConfigurer`가 제공하는 `addFormatters()`를 사용해서 추가하고 싶은 컨버터를 등록하면 된다. 이렇게 하면 스프링은 내부에서 사용하는

`ConversionService`에 컨버터를 추가해준다.

등록한 컨버터가 잘 동작하는지 확인해보자.

HelloController - 기존 코드

```
@GetMapping("/hello-v2")
public String helloV2(@RequestParam Integer data) {
    System.out.println("data = " + data);
    return "ok";
}
```

실행

http://localhost:8080/hello-v2?data=10

실행 로그

```
StringToIntegerConverter    : convert source=10
data = 10
```

`?data=10`의 쿼리 파라미터는 문자이고 이것을 `Integer data`로 변환하는 과정이 필요하다.
실행해보면 직접 등록한 `StringToIntegerConverter`가 작동하는 로그를 확인할 수 있다.

그런데 생각해보면 `StringToIntegerConverter`를 등록하기 전에도 이 코드는 잘 수행되었다. 그것은 스프링이 내부에서 수 많은 기본 컨버터들을 제공하기 때문이다. 컨버터를 추가하면 추가한 컨버터가 기본 컨버터 보다 높은 우선순위를 가진다.

이번에는 직접 정의한 타입인 `IpPort`를 사용해보자.

HelloController - 추가

```
@GetMapping("/ip-port")
public String ipPort(@RequestParam IpPort ipPort) {
    System.out.println("ipPort IP = " + ipPort.getIp());
    System.out.println("ipPort PORT = " + ipPort.getPort());
    return "ok";
}
```

실행

http://localhost:8080/ip-port?ipPort=127.0.0.1:8080

실행 로그

```
StringToIpPortConverter    : convert source=127.0.0.1:8080
ipPort IP = 127.0.0.1
ipPort PORT = 8080
```

`?ipPort=127.0.0.1:8080` 쿼리 스트링이 `@RequestParam IpPort ipPort`에서 객체 타입으로 잘 변환 된 것을 확인할 수 있다.

처리 과정

`@RequestParam`은 `@RequestParam`을 처리하는 `ArgumentResolver`인 `RequestParamMethodArgumentResolver`에서 `ConversionService`를 사용해서 타입을 변환한다. 부모 클래스와 다양한 외부 클래스를 호출하는 등 복잡한 내부 과정을 거치기 때문에 대략 이렇게 처리되는 것으로 이해해도 충분하다. 만약 더 깊이있게 확인하고 싶으면 `IpPortConverter`에 디버그 브레이크 포인트를 걸어서 확인해보자.

뷰 템플릿에 컨버터 적용하기

이번에는 뷰 템플릿에 컨버터를 적용하는 방법을 알아보자.

타임리프는 렌더링 시에 컨버터를 적용해서 렌더링 하는 방법을 편리하게 지원한다.

이전까지는 문자를 객체로 변환했다면, 이번에는 그 반대로 객체를 문자로 변환하는 작업을 확인할 수 있다.

ConverterController

```
package hello.typeconverter.controller;

import hello.typeconverter.type.IpPort;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;


@Controller
public class ConverterController {

    @GetMapping("/converter-view")
    public String converterView(Model model) {
        model.addAttribute("number", 10000);
        model.addAttribute("ipPort", new IpPort("127.0.0.1", 8080));
```

```
        return "converter-view";
    }

}
```

`Model`에 숫자 `10000`와 `ipPort` 객체를 담아서 뷰 템플릿에 전달한다.

`resources/templates/converter-view.html`

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<ul>
    <li>${number}: <span th:text="${number}" ></span></li>
    <li>${{number}}: <span th:text="${{number}}" ></span></li>
    <li>${ipPort}: <span th:text="${ipPort}" ></span></li>
    <li>${{ipPort}}: <span th:text="${{ipPort}}" ></span></li>
</ul>

</body>
</html>
```

타임리프는 `${{...}}`를 사용하면 자동으로 컨버전 서비스를 사용해서 변환된 결과를 출력해준다. 물론 스프링과 통합 되어서 스프링이 제공하는 컨버전 서비스를 사용하므로, 우리가 등록한 컨버터들을 사용할 수 있다.

변수 표현식 : `${...}`
컨버전 서비스 적용 : `${{...}}`

실행

http://localhost:8080/converter-view

실행 결과

- ${number}: 10000
- ${{number}}: 10000
- ${ipPort}: hello.typeconverter.type.IpPort@59cb0946
- ${{ipPort}}: 127.0.0.1:8080

실행 결과 로그

```
IntegerToStringConverter    : convert source=10000
IpPortToStringConverter     : convert
source=hello.typeconverter.type.IpPort@59cb0946
```

- `${{number}}` : 뷰 템플릿은 데이터를 문자로 출력한다. 따라서 컨버터를 적용하게 되면 `Integer` 타입인 `10000` 을 `String` 타입으로 변환하는 컨버터인 `IntegerToStringConverter` 를 실행하게 된다. 이 부분은 컨버터를 실행하지 않아도 타임리프가 숫자를 문자로 자동으로 변환히기 때문에 컨버터를 적용할 때와 하지 않을 때가 같다.
- `${{ipPort}}` : 뷰 템플릿은 데이터를 문자로 출력한다. 따라서 컨버터를 적용하게 되면 `IpPort` 타입을 `String` 타입으로 변환해야 하므로 `IpPortToStringConverter` 가 적용된다. 그 결과 `127.0.0.1:8080` 가 출력된다.

폼에 적용하기

이번에는 컨버터를 폼에 적용해보자.

ConverterController - 코드 추가

```
package hello.typeconverter.controller;

import hello.typeconverter.type.IpPort;
import lombok.Data;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;


@Controller
public class ConverterController {

    @GetMapping("/converter-view")
    public String converterView(Model model) {
        model.addAttribute("number", 10000);
        model.addAttribute("ipPort", new IpPort("127.0.0.1", 8080));
        return "converter-view";
    }

    @GetMapping("/converter/edit")
```

```
    public String converterForm(Model model) {

        IpPort ipPort = new IpPort("127.0.0.1", 8080);
        Form form = new Form(ipPort);

        model.addAttribute("form", form);
        return "converter-form";
    }

    @PostMapping("/converter/edit")
    public String converterEdit(@ModelAttribute Form form, Model model) {
        IpPort ipPort = form.getIpPort();
        model.addAttribute("ipPort", ipPort);
        return "converter-view";
    }

    @Data
    static class Form {
        private IpPort ipPort;

        public Form(IpPort ipPort) {
            this.ipPort = ipPort;
        }
    }

}
```

`Form` 객체를 데이터를 전달하는 폼 객체로 사용한다.

- `GET /converter/edit`: `IpPort`를 뷰 템플릿 폼에 출력한다.
- `POST /converter/edit`: 뷰 템플릿 폼의 `IpPort` 정보를 받아서 출력한다.

`resources/templates/converter-form.html`

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<form th:object="${form}" th:method="post">
```

```
    th:field <input type="text" th:field="*{ipPort}"><br/>
    th:value <input type="text" th:value="*{ipPort}">(보여주기 용도)<br/>
    <input type="submit"/>
</form>

</body>
</html>
```

타임리프의 `th:field`는 앞서 설명했듯이 `id`, `name`를 출력하는 등 다양한 기능이 있는데, 여기에 컨버전 서비스도 함께 적용된다.

실행

http://localhost:8080/converter/edit

- `GET /converter/edit`
  - `th:field`가 자동으로 컨버전 서비스를 적용해주어서 `${{ipPort}}`처럼 적용이 되었다. 따라서 `IpPort` ➜ `String`으로 변환된다.
- `POST /converter/edit`
  - `@ModelAttribute`를 사용해서 `String` ➜ `IpPort`로 변환된다.

포맷터 - Formatter

`Converter`는 입력과 출력 타입에 제한이 없는, 범용 타입 변환 기능을 제공한다.
이번에는 일반적인 웹 애플리케이션 환경을 생각해보자. 불린 타입을 숫자로 바꾸는 것 같은 범용 기능 보다는 개발자 입장에서는
문자를 다른 타입으로 변환하거나, 다른 타입을 문자로 변환하는 상황이 대부분이다.
앞서 살펴본 예제들을 떠올려 보면 문자를 다른 객체로 변환하거나 객체를 문자로 변환하는 일이 대부분이다.

웹 애플리케이션에서 객체를 문자로, 문자를 객체로 변환하는 예

- 화면에 숫자를 출력해야 하는데, `Integer` ➜ `String` 출력 시점에 숫자 `1000` ➜ 문자 `"1,000"` 이렇게 1000 단위에 쉼표를 넣어서 출력하거나, 또는 `"1,000"`라는 문자를 `1000`이라는 숫자로 변경해야 한다.
- 날짜 객체를 문자인 `"2021-01-01 10:50:11"`와 같이 출력하거나 또는 그 반대의 상황

Locale

여기에 추가로 날짜 숫자의 표현 방법은 `Locale` 현지화 정보가 사용될 수 있다.

이렇게 객체를 특정한 포멧에 맞추어 문자로 출력하거나 또는 그 반대의 역할을 하는 것에 특화된 기능이 바로 포맷터(`Formatter`)이다. 포맷터는 컨버터의 특별한 버전으로 이해하면 된다.

Converter vs Formatter

- `Converter`는 범용(객체 ➡ 객체)
- `Formatter`는 문자에 특화(객체 ➡ 문자, 문자 ➡ 객체) + 현지화(Locale)
  - `Converter`의 특별한 버전

포맷터 - Formatter 만들기

포맷터(`Formatter`)는 객체를 문자로 변경하고, 문자를 객체로 변경하는 두 가지 기능을 모두 수행한다.

- `String print(T object, Locale locale)` : 객체를 문자로 변경한다.
- `T parse(String text, Locale locale)` : 문자를 객체로 변경한다.

Formatter 인터페이스

```
public interface Printer<T> {
    String print(T object, Locale locale);
}

public interface Parser<T> {
    T parse(String text, Locale locale) throws ParseException;
}

public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

숫자 `1000`을 문자 `"1,000"`으로 그러니까, 1000 단위로 쉼표가 들어가는 포맷을 적용해보자. 그리고 그 반대도 처리해주는 포맷터를 만들어보자.

MyNumberFormatter

```
package hello.typeconverter.formatter;

import lombok.extern.slf4j.Slf4j;
import org.springframework.format.Formatter;
```

```
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;

@Slf4j
public class MyNumberFormatter implements Formatter<Number> {

    @Override
    public Number parse(String text, Locale locale) throws ParseException {
        log.info("text={}, locale={}", text, locale);
        NumberFormat format = NumberFormat.getInstance(locale);
        return format.parse(text);
    }

    @Override
    public String print(Number object, Locale locale) {
        log.info("object={}, locale={}", object, locale);
        return NumberFormat.getInstance(locale).format(object);
    }
}
```

`"1,000"` 처럼 숫자 중간의 쉼표를 적용하려면 자바가 기본으로 제공하는 `NumberFormat` 객체를 사용하면 된다. 이 객체는 `Locale` 정보를 활용해서 나라별로 다른 숫자 포맷을 만들어준다.

`parse()` 를 사용해서 문자를 숫자로 변환한다. 참고로 `Number` 타입은 `Integer`, `Long` 과 같은 숫자 타입의 부모 클래스이다.
`print()` 를 사용해서 객체를 문자로 변환한다.

잘 동작하는지 테스트 코드를 만들어보자.

MyNumberFormatterTest

```
package hello.typeconverter.formatter;

import org.junit.jupiter.api.Test;

import java.text.ParseException;
import java.util.Locale;

import static org.assertj.core.api.Assertions.*;
```

```java
class MyNumberFormatterTest {

    MyNumberFormatter formatter = new MyNumberFormatter();

    @Test
    void parse() throws ParseException {
        Number result = formatter.parse("1,000", Locale.KOREA);
        assertThat(result).isEqualTo(1000L); //Long 타입 주의
    }

    @Test
    void print() {
        String result = formatter.print(1000, Locale.KOREA);
        assertThat(result).isEqualTo("1,000");
    }
}
```

`parse()`의 결과가 `Long`이기 때문에 `isEqualTo(1000L)`을 통해 비교할 때 마지막에 `L`을 넣어주어야 한다.

실행 결과 로그

```
MyNumberFormatter - text=1,000, locale=ko_KR
MyNumberFormatter - object=1000, locale=ko_KR
```

> **참고**
> 스프링은 용도에 따라 다양한 방식의 포맷터를 제공한다.
> `Formatter` 포맷터
> `AnnotationFormatterFactory` 필드의 타입이나 애노테이션 정보를 활용할 수 있는 포맷터
>
> 자세한 내용은 공식 문서를 참고하자.
> https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#format

포맷터를 지원하는 컨버전 서비스

컨버전 서비스에는 컨버터만 등록할 수 있고, 포맷터를 등록할 수 는 없다. 그런데 생각해보면 포맷터는 객체 ➔ 문자, 문자 ➔ 객체로 변환하는 특별한 컨버터일 뿐이다.

포맷터를 지원하는 컨버전 서비스를 사용하면 컨버전 서비스에 포맷터를 추가할 수 있다. 내부에서 어댑터 패턴을 사용해서 `Formatter`가 `Converter`처럼 동작하도록 지원한다.

`FormattingConversionService`는 포맷터를 지원하는 컨버전 서비스이다.
`DefaultFormattingConversionService`는 `FormattingConversionService`에 기본적인 통화, 숫자 관련 몇가지 기본 포맷터를 추가해서 제공한다.

```
package hello.typeconverter.formatter;

import hello.typeconverter.converter.IpPortToStringConverter;
import hello.typeconverter.converter.StringToIpPortConverter;
import hello.typeconverter.type.IpPort;
import org.junit.jupiter.api.Test;
import org.springframework.format.support.FormattingConversionService;

import static org.assertj.core.api.Assertions.assertThat;

public class FormattingConversionServiceTest {

    @Test
    void formattingConversionService() {

        DefaultFormattingConversionService conversionService = new
DefaultFormattingConversionService();
        //컨버터 등록
        conversionService.addConverter(new StringToIpPortConverter());
        conversionService.addConverter(new IpPortToStringConverter());
        //포맷터 등록
        conversionService.addFormatter(new MyNumberFormatter());

        //컨버터 사용
        IpPort ipPort = conversionService.convert("127.0.0.1:8080",
IpPort.class);
        assertThat(ipPort).isEqualTo(new IpPort("127.0.0.1", 8080));
        //포맷터 사용
        assertThat(conversionService.convert(1000,
String.class)).isEqualTo("1,000");
        assertThat(conversionService.convert("1,000",
Long.class)).isEqualTo(1000L);
    }
}
```

DefaultFormattingConversionService 상속 관계

`FormattingConversionService`는 `ConversionService`관련 기능을 상속받기 때문에 결과적으로 컨버터도 포맷터도 모두 등록할 수 있다. 그리고 사용할 때는 `ConversionService`가 제공하는 `convert`를 사용하면 된다.

추가로 스프링 부트는 `DefaultFormattingConversionService`를 상속 받은 `WebConversionService`를 내부에서 사용한다.

포맷터 적용하기

포맷터를 웹 애플리케이션에 적용해보자.

WebConfig - 수정

```
package hello.typeconverter;

import hello.typeconverter.converter.IntegerToStringConverter;
import hello.typeconverter.converter.IpPortToStringConverter;
import hello.typeconverter.converter.StringToIntegerConverter;
import hello.typeconverter.converter.StringToIpPortConverter;
import hello.typeconverter.formatter.MyNumberFormatter;
import org.springframework.context.annotation.Configuration;
import org.springframework.format.FormatterRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addFormatters(FormatterRegistry registry) {

        //주석처리 우선순위
        //registry.addConverter(new StringToIntegerConverter());
        //registry.addConverter(new IntegerToStringConverter());
        registry.addConverter(new StringToIpPortConverter());
        registry.addConverter(new IpPortToStringConverter());

        //추가
        registry.addFormatter(new MyNumberFormatter());
    }
}
```

주의 `StringToIntegerConverter`, `IntegerToStringConverter`를 꼭 주석처리 하자.
`MyNumberFormatter`도 숫자 ➜ 문자, 문자 ➜ 숫자로 변경하기 때문에 둘의 기능이 겹친다. 우선순위는 컨버터가 우선하므로 포맷터가 적용되지 않고, 컨버터가 적용된다.

실행 - 객체 ➜ 문자

http://localhost:8080/converter-view

- ${number}: 10000
- ${{number}}: 10,000

컨버전 서비스를 적용한 결과 `MyNumberFormatter`가 적용되어서 `10,000` 문자가 출력된 것을 확인할 수 있다.

실행 - 문자 ➜ 객체

http://localhost:8080/hello-v2?data=10,000

실행 로그

```
MyNumberFormatter : text=10,000, locale=ko_KR
data = 10000
```

`"10,000"`이라는 포맷팅 된 문자가 `Integer` 타입의 숫자 10000으로 정상 변환 된 것을 확인할 수 있다.

스프링이 제공하는 기본 포맷터

스프링은 자바에서 기본으로 제공하는 타입들에 대해 수 많은 포맷터를 기본으로 제공한다.
IDE에서 `Formatter` 인터페이스의 구현 클래스를 찾아보면 수 많은 날짜나 시간 관련 포맷터가 제공되는 것을 확인할 수 있다.
그런데 포맷터는 기본 형식이 지정되어 있기 때문에, 객체의 각 필드마다 다른 형식으로 포맷을 지정하기는 어렵다.

스프링은 이런 문제를 해결하기 위해 애노테이션 기반으로 원하는 형식을 지정해서 사용할 수 있는 매우 유용한 포맷터 두 가지를 기본으로 제공한다.

- `@NumberFormat`: 숫자 관련 형식 지정 포맷터 사용, `NumberFormatAnnotationFormatterFactory`
- `@DateTimeFormat`: 날짜 관련 형식 지정 포맷터 사용, `Jsr310DateTimeFormatAnnotationFormatterFactory`

예제를 통해서 알아보자.

FormatterController

```
package hello.typeconverter.controller;

import lombok.Data;
import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.format.annotation.NumberFormat;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;

import java.time.LocalDateTime;

@Controller
public class FormatterController {

    @GetMapping("/formatter/edit")
    public String formatterForm(Model model) {

        Form form = new Form();
        form.setNumber(10000);
        form.setLocalDateTime(LocalDateTime.now());

        model.addAttribute("form", form);
        return "formatter-form";
    }

    @PostMapping("/formatter/edit")
    public String formatterEdit(@ModelAttribute Form form) {
        return "formatter-view";
    }

    @Data
    static class Form {

        @NumberFormat(pattern = "###,###")
        private Integer number;
```

```
        @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
        private LocalDateTime localDateTime;
    }
}
```

templates/formatter-form.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<form th:object="${form}" th:method="post">
    number <input type="text" th:field="*{number}"><br/>
    localDateTime <input type="text" th:field="*{localDateTime}"><br/>
    <input type="submit"/>
</form>

</body>
</html>
```

templates/formatter-view.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<ul>
    <li>${form.number}: <span th:text="${form.number}" ></span></li>
    <li>${{form.number}}: <span th:text="${{form.number}}" ></span></li>
    <li>${form.localDateTime}: <span th:text="${form.localDateTime}" ></span></
li>
    <li>${{form.localDateTime}}: <span th:text="${{form.localDateTime}}" ></
span></li>
</ul>

</body>
```

```
</html>
```

실행

http://localhost:8080/formatter/edit

실행해보면 지정한 포맷으로 출력된 것을 확인할 수 있다.

결과

```
• ${form.number}: 10000
• ${{form.number}}: 10,000
• ${form.localDateTime}: 2021-01-01T00:00:00
• ${{form.localDateTime}}: 2021-01-01 00:00:00
```

> 참고
> `@NumberFormat`, `@DateTimeFormat`의 자세한 사용법이 궁금한 분들은 다음 링크를 참고하거나 관련 애노테이션을 검색해보자.
> https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#format-CustomFormatAnnotations

정리

컨버터를 사용하든, 포맷터를 사용하든 등록 방법은 다르지만, 사용할 때는 컨버전 서비스를 통해서 일관성 있게 사용할 수 있다.

주의!
메시지 컨버터(`HttpMessageConverter`)에는 컨버전 서비스가 적용되지 않는다.
특히 객체를 JSON으로 변환할 때 메시지 컨버터를 사용하면서 이 부분을 많이 오해하는데,
`HttpMessageConverter`의 역할은 HTTP 메시지 바디의 내용을 객체로 변환하거나 객체를 HTTP 메시지 바디에 입력하는 것이다. 예를 들어서 JSON을 객체로 변환하는 메시지 컨버터는 내부에서 Jackson 같은 라이브러리를 사용한다. 객체를 JSON으로 변환한다면 그 결과는 이 라이브러리에 달린 것이다. 따라서 JSON 결과로 만들어지는 숫자나 날짜 포맷을 변경하고 싶으면 해당 라이브러리가 제공하는 설정을 통해서 포맷을 지정해야 한다. 결과적으로 이것은 컨버전 서비스와 전혀 관계가 없다.

컨버전 서비스는 `@RequestParam`, `@ModelAttribute`, `@PathVariable`, 뷰 템플릿 등에서 사용할 수 있다.