Chap05. Bean

1. Bean

아래 코드는 이번 차시의 테스트에 공통적으로 사용할 `Product` , `Beverage` , `Bread` , `ShoppingCart` 클래스이다.

1. Product(abstract class)

```
public abstract class Product {

  private String name;  //상품명
  private int price;    //상품가격

  public Product() {}

  public Product(String name, int price) {
    super();
    this.name = name;
    this.price = price;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public int getPrice() {
    return price;
  }

  public void setPrice(int price) {
    this.price = price;
  }

  @Override
  public String toString() {
    return name + " " + price;
  }
}
```

2. Beverage

```
public class Beverage extends Product {

  private int capacity; //용량

  public Beverage() {
    super();
  }

  public Beverage(String name, int price, int capacity) {
    super(name, price);
    this.capacity = capacity;
  }
```

```
  public int getCapacity() {
    return this.capacity;
  }

  public void setCapacity(int capacity) {
    this.capacity = capacity;
  }

  @Override
  public String toString() {
    return super.toString() + " " + this.capacity;
  }
}
```

3. Bread

```
public class Bread extends Product {

  private java.util.Date bakedDate; //생산시간

  public Bread() {
    super();
  }

  public Bread(String name, int price, java.util.Date bakedDate) {
    super(name, price);
    this.bakedDate = bakedDate;
  }

  public java.util.Date getBakedDate() {
    return this.bakedDate;
  }

  public void setBakedDate(java.util.Date bakedDate) {
    this.bakedDate = bakedDate;
  }

  @Override
  public String toString() {
    return super.toString() + " " + this.bakedDate;
  }
}
```

4. ShoppingCart

```
public class ShoppingCart {

  private final List<Product> items;  //쇼핑카트에 담긴 상품들

  public ShoppingCart() {
    items = new ArrayList<>();
  }

  public void addItem(Product item) {
    items.add(item);
  }

  public List<Product> getItem() {
    return items;
  }
}
```

1-1. Bean Scope

`bean scope`란 스프링 빈이 생성될 때 생성되는 인스턴스의 범위를 의미한다. 스프링에서는 다양한 bean scope를 제공한다.

Bean Scope	Description
Singleton	하나의 인스턴스만을 생성하고, 모든 빈이 해당 인스턴스를 공유하여 사용한다.
Prototype	매번 새로운 인스턴스를 생성한다.
Request	HTTP 요청을 처리할 때마다 새로운 인스턴스를 생성하고, 요청 처리가 끝나면 인스턴스를 폐기한다. 웹 애플리케이션 컨텍스트에만 해당된다.
Session	HTTP 세션 당 하나의 인스턴스를 생성하고, 세션이 종료되면 인스턴스를 폐기한다. 웹 애플리케이션 컨텍스트에만 해당된다.

1-1-1. singleton

> 💡 Spring Framework에서 Bean의 기본 스코프는 `singleton`이다. `singleton`은 애플리케이션 내에서 하나의 인스턴스만을 생성하고, 모든 빈이 해당 인스턴스를 공유하여 사용한다. 이를 통해 메모리 사용량을 줄일 수 있으며, 성능 향상을 기대할 수 있다.

```
@Configuration
public class ContextConfiguration {

  @Bean
  public Product carpBread() {

    return new Bread("붕어빵", 1000, new java.util.Date());
  }

  @Bean
  public Product milk() {

    return new Beverage("딸기우유", 1500, 500);
  }

  @Bean
  public Product water() {

    return new Beverage("지리산암반수", 3000, 500);
  }

  @Bean
  @Scope("singleton")   //기본값
  public ShoppingCart cart() {

    return new ShoppingCart();
  }
}
```

ContextConfiguration 설정 파일에 `Bread` 타입의 붕어빵과 `Beaverage` 타입의 딸기우유와 지리산암반수, `ShoppingCart` 타입의 쇼핑카트를 bean으로 등록 한다. `@Scope` 어노테이션에 기본 값에 해당하는 `singleton` 설정도 기재하였다.

```
/* 빈 설정 파일을 기반으로 IoC 컨테이너 생성 */
ApplicationContext context = new AnnotationConfigApplicationContext(ContextConfiguration.class);

/* 붕어빵, 딸기우유, 지리산 암반수 등의 빈 객체를 반환 받는다. */
Product carpBread = context.getBean("carpBread", Bread.class);
Product milk = context.getBean("milk", Beverage.class);
Product water = context.getBean("water", Beverage.class);

/* 첫 번째 손님이 쇼핑 카트를 꺼낸다. */
ShoppingCart cart1 = context.getBean("cart", ShoppingCart.class);
cart1.addItem(carpBread);
cart1.addItem(milk);

/* 붕어빵과 딸기우유가 담겨있다. */
System.out.println("cart1에 담긴 내용 : " + cart1.getItem());

/* 두 번째 손님이 쇼핑 카트를 꺼낸다. */
ShoppingCart cart2 = context.getBean("cart", ShoppingCart.class);
cart2.addItem(water);

/* 붕어빵과 딸기우유와 지리산암반수가 담겨있다. */
System.out.println("cart2에 담긴 내용 : " + cart2.getItem());

/* 두 카드의 hashcode를 출력해보면 동일한 것을 볼 수 있다. */
System.out.println("cart1의 hashcode : " + cart1.hashCode());
System.out.println("cart2의 hashcode : " + cart2.hashCode());
```

▼ 실행 결과

```
cart1에 담긴 내용 : [붕어빵 1000 Thu Jun 08 21:58:27 KST 2023, 딸기우유 1500 500]
cart2에 담긴 내용 : [붕어빵 1000 Thu Jun 08 21:58:27 KST 2023, 딸기우유 1500 500, 지리산암반수 3000 500]
cart1의 hashcode : 696933920
cart2의 hashcode : 696933920
```

이 예제에서 손님 두 명이 각각 쇼핑 카트를 이용해 상품을 담는다고 가정했지만 `singleton`으로 관리되는 `cart`는 사실 `하나의 객체`이므로 두 손님이 동일한 카트에 물건을 담는 상황이 발생한다. 상황에 따라서 기본 값인 singleton 스코프가 아닌 prototype 스코프가 필요할 수 있다.

1-1-2. prototype

💡 `prototype` 스코프를 갖는 Bean은 매번 새로운 인스턴스를 생성한다. 이를 통해 의존성 주입 등의 작업에서 객체 생성에 대한 부담을 줄일 수 있다.

```
@Configuration
public class ContextConfiguration {

  @Bean
  public Product carpBread() {

    return new Bread("붕어빵", 1000, new java.util.Date());
  }

  @Bean
```

```
  public Product milk() {

    return new Beverage("딸기우유", 1500, 500);
  }

  @Bean
  public Product water() {

    return new Beverage("지리산암반수", 3000, 500);
  }

  @Bean
  @Scope("prototype")   //기본 값에서 변경
  public ShoppingCart cart() {

    return new ShoppingCart();
  }

}
```

이전 예제와 동일하게 ContextConfiguration 설정 파일에 `Bread` 타입의 붕어빵과 `Beaverage` 타입의 딸기우유와 지리산암반수, `ShoppingCart` 타입의 쇼핑카트를 bean으로 등록 한다. 단, `ShoppingCart`의 경우 `@Scope` 어노테이션에 기본 값이 아닌 해당하는 `prototype` 설정도 기재하였다.

```
/* 빈 설정 파일을 기반으로 IoC 컨테이너 생성 */
ApplicationContext context = new AnnotationConfigApplicationContext(ContextConfiguration.class);

/* 붕어빵, 딸기우유, 지리산 암반수 등의 빈 객체를 반환 받는다. */
Product carpBread = context.getBean("carpBread", Bread.class);
Product milk = context.getBean("milk", Beverage.class);
Product water = context.getBean("water", Beverage.class);

/* 첫 번째 손님이 쇼핑 카트를 꺼낸다. */
ShoppingCart cart1 = context.getBean("cart", ShoppingCart.class);
cart1.addItem(carpBread);
cart1.addItem(milk);

/* 붕어빵과 딸기우유가 담겨있다. */
System.out.println("cart1에 담긴 내용 : " + cart1.getItem());

/* 두 번째 손님이 쇼핑 카트를 꺼낸다. */
ShoppingCart cart2 = context.getBean("cart", ShoppingCart.class);
cart2.addItem(water);

/* 지리산암반수가 담겨있다. */
System.out.println("cart2에 담긴 내용 : " + cart2.getItem());

/* 두 카드의 hashcode를 출력해보면 다른 것을 볼 수 있다. */
System.out.println("cart1의 hashcode : " + cart1.hashCode());
System.out.println("cart2의 hashcode : " + cart2.hashCode());
```

▼ 실행 결과

```
cart1에 담긴 내용 : [붕어빵 1000 Thu Jun 08 21:58:40 KST 2023, 딸기우유 1500 500]
cart2에 담긴 내용 : [지리산암반수 3000 500]
cart1의 hashcode : 1946988038
cart2의 hashcode : 1990519794
```

ShoppingCart의 bean scope를 `prototype`으로 설정하자 `getBean`으로 인스턴스를 꺼내올 때 마다 새로운 인스턴스를 생성하게 된다. 따라서 이번 예제에서는 손님 두 명이 각각 쇼핑 카트를 이용해 상품을 담는 상황이 잘 연출되었다.

1-1-3. xml 설정

위의 예제에서는 모두 Java 빈 객체 설정을 통해 확인해보았다. 만약 XML 파일에 `<bean>` 태그를 이용한다면 다음과 같이 속성을 기재할 수 있다.

```
<!-- singleton 설정 -->
<bean id="cart" class="패키지명.ShoppingCart" scope="singleton"/>
```

```
<!-- prototype 설정 -->
<bean id="cart" class="패키지명.ShoppingCart" scope="prototype"/>
```

XML 설정에 대한 별도의 실행 예제는 생략한다.

1-2. init, destroy method

> 💡 스프링 빈은 `초기화(init)`와 `소멸화(destroy)`의 `라이프 사이클`을 가지고 있다. 이 라이프 사이클을 이해하면 빈 객체가 생성되고 소멸될 때 추가적인 작업을 수행할 수 있다.

`init-method` 속성을 사용하면 스프링이 빈 객체를 생성한 다음 초기화 작업을 수행할 메소드를 지정할 수 있다. 이 메소드는 빈 객체 생성자가 완료된 이후에 호출된다. `init-method` 속성으로 지정된 메소드는 일반적으로 빈의 초기화를 위해 사용된다.

`destroy-method` 속성을 사용하면 빈 객체가 소멸될 때 호출할 메소드를 지정할 수 있다. 이 메소드는 `ApplicationContext`의 `close()` 메소드가 호출되기 전에 빈 객체가 소멸될 때 호출된다. `destroy-method` 속성으로 지정 된 메소드는 일반적으로 사용하던 리소스를 반환하기 위해 사용된다.

1-2-1. java

java 설정 방식으로 `init-method`, `destroy-method` 를 테스트 하기 위해 `Owner` 라는 클래스를 추가한다.

```
public class Owner {

  public void openShop() {
    System.out.println("사장님이 가게 문을 열었습니다. 이제 쇼핑을 하실 수 있습니다.");
  }

  public void closeShop() {
    System.out.println("사장님이 가게 문을 닫았습니다. 이제 쇼핑을 하실 수 없습니다.");
  }

}
```

`Owner` 타입의 빈 객체를 설정 파일에 등록한다. `@Bean` 어노테이션에는 `initMethod`, `destoryMethod` 라는 속성이 있는데 해당 속성을 Owner 클래스에 정의했던 `openShop`, `closeShop` 메소드로 설정한다.

```
@Configuration
public class ContextConfiguration {

  ...생략

  /* init-method로 openShop 메소드를 설정하고 destory-method로 closeShope 메소드를 설정한다. */
  @Bean(initMethod = "openShop", destroyMethod="closeShop")
  public Owner owner() {

    return new Owner()
  }

}
```

실행 파일에서 빈 설정 파일을 기반으로 IoC 컨테이너를 생성하고 쇼핑 카트에 상품을 담는 동일한 코드의 끝에 컨테이너를 종료하는 코드를 추가하여 실행해본다.

```
/* 빈 설정 파일을 기반으로 IoC 컨테이너 생성 */
ApplicationContext context = new AnnotationConfigApplicationContext(ContextConfiguration.class);

...생략

/* init 메소드는 빈 객체 생성 시점에 동작하므로 바로 확인할 수 있지만
 * destroy 메소드는 빈 객체 소멸 시점에 동작하므로 컨테이너가 종료 되지 않을 경우 확인할 수 없다.
 * 가비지 컬렉터가 해당 빈을 메모리에서 지울 때 destroy 메소드가 동작하게 되는데
 * 메모리에서 지우기 전에 프로세스는 종료되기 때문이다.
 * 따라서 아래와 같이 강제로 컨테이너를 종료시키면 destroy 메소드가 동작할 것이다.
 * */
((AnnotationConfigApplicationContext) context).close();
```

▼ 실행 결과

```
사장님이 가게 문을 열었습니다. 이제 쇼핑을 하실 수 있습니다.
cart1에 담긴 내용 : [붕어빵 1000 Thu Jun 08 21:58:53 KST 2023, 딸기우유 1500 500]
cart2에 담긴 내용 : [지리산암반수 3000 500]
사장님이 가게 문을 닫았습니다. 이제 쇼핑을 하실 수 없습니다.
```

`openShop` 메소드가 `Owner` 객체의 생성 시점에 호출 되고, `closeShop` 메소드가 `Owner` 객체의 소멸 시점에 호출 되었음을 확인할 수 있다. init 메소드는 빈 객체 생성 시점에 동작하므로 바로 확인할 수 있지만 destroy 메소드는 빈 객체 소멸 시점에 동작하므로 컨테이너가 종료 되지 않을 경우 확인할 수 없다. 따라서 컨테이너를 종료시키면 destroy 메소드의 동작까지 확인 할 수 있다.

1-2-2. annotataion

annotation 방식으로 `init-method`, `destroy-method` 를 테스트 하기 위해 `Owner` 라는 클래스를 추가한다.

```
@Component
public class Owner {

  /* init-method와 같은 설정 어노테이션이다. */
  @PostConstruct
  public void openShop() {
```

```
    System.out.println("사장님이 가게 문을 오픈하셨습니다. 이제 쇼핑을 하실 수 있습니다.");
  }

  /* destroy-method와 같은 설정 어노테이션이다. */
  @PreDestroy
  public void closeShop() {
    System.out.println("사장님이 가게 문을 닫으셨습니다. 이제 쇼핑을 하실 수 없습니다.");
  }

}
```

컴포넌트 스캔을 통해 빈 등록 하기 위해 `@Componet` 어노테이션을 클래스 위에 설정한다. `javax.annotation` 의 `@PostConstruct` , `@PreDestroy` 어노테이션은 `@Bean` 어노테이션에 `init-method`, `destroy-method` 속성을 설정하는 것과 같은 역할을 한다. 단, 해당 어노테이션을 사용할 수 있도록 라이브러리 의존성이 `build.gradle.kts` 파일에 추가 되어 있어야 한다.

```
dependencies {
  ...생략
  implementation 'javax.annotation:javax.annotation-api:1.3.2'
}
```

`Owner` 타입의 빈 객체를 컴포넌트 스캔을 통해 읽어 등록할 수 있도록 빈 설정 파일에 컴포넌트 스캔 경로를 설정한다. 그 외의 상품과 쇼핑 카트의 빈 등록 코드는 그대로 사용한다.

```
@Configuration
@ComponentScan("com.ohgiraffers.section02.initdestroy.subsection02.annotation")
public class ContextConfiguration {
  ...생략
}
```

실행 파일에서 빈 설정 파일을 기반으로 IoC 컨테이너를 생성하고 쇼핑 카트에 상품을 담는 동일한 코드의 끝에 컨테이너를 종료하는 코드를 추가하여 실행해본다.

```
/* 빈 설정 파일을 기반으로 IoC 컨테이너 생성 */
ApplicationContext context = new AnnotationConfigApplicationContext(ContextConfiguration.class);

...생략

/* init 메소드는 빈 객체 생성 시점에 동작하므로 바로 확인할 수 있지만
 * destroy 메소드는 빈 객체 소멸 시점에 동작하므로 컨테이너가 종료 되지 않을 경우 확인할 수 없다.
 * 가비지 컬렉터가 해당 빈을 메모리에서 지울 때 destroy 메소드가 동작하게 되는데 메모리에서 지우기 전에 프로세스는 종료되기 때문이다.
 * 따라서 아래와 같이 강제로 컨테이너를 종료시키면 destroy 메소드가 동작할 것이다.
 * */
((AnnotationConfigApplicationContext) context).close();
```

▼ 실행 결과

```
사장님이 가게 문을 열었습니다. 이제 쇼핑을 하실 수 있습니다.
cart1에 담긴 내용 : [붕어빵 1000 Thu Jun 08 21:59:07 KST 2023, 딸기우유 1500 500]
cart2에 담긴 내용 : [지리산암반수 3000 500]
사장님이 가게 문을 닫았습니다. 이제 쇼핑을 하실 수 없습니다.
```

`openShop` 메소드가 `Owner` 객체의 생성 시점에 호출 되고, `closeShop` 메소드가 `Owner` 객체의 소멸 시점에 호출 되었음을 확인할 수 있다.

1-2-3. xml

xml 설정 방식으로 `init-method`, `destroy-method` 를 테스트 하기 위해 `Owner` 라는 클래스를 추가한다.

```
public class Owner {

  public void openShop() {
    System.out.println("사장님이 가게 문을 열었습니다. 이제 쇼핑을 하실 수 있습니다.");
  }

  public void closeShop() {
    System.out.println("사장님이 가게 문을 닫았습니다. 이제 쇼핑을 하실 수 없습니다.");
  }

}
```

bean configuration file에 `<bean>` 태그를 통해 상품, 쇼핑카트 그리고 `Owener` 에 대한 빈 등록을 설정한다. `Owener` 에는 `init-method`, `destroy-method` 를 설정한다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
                      http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="carpBread" class="com.ohgiraffers.common.Bread">
    <constructor-arg name="name" value="붕어빵"/>
    <constructor-arg name="price" value="1000"/>
    <constructor-arg name="bakedDate" ref="today"/>
  </bean>

  <bean id="today" class="java.util.Date" scope="prototype"/>

  <bean id="milk" class="com.ohgiraffers.common.Beverage">
    <constructor-arg name="name" value="딸기우유"/>
    <constructor-arg name="price" value="1500"/>
    <constructor-arg name="capacity" value="500"/>
  </bean>

  <bean id="water" class="com.ohgiraffers.common.Beverage">
    <constructor-arg name="name" value="지리산암반수"/>
    <constructor-arg name="price" value="3000"/>
    <constructor-arg name="capacity" value="500"/>
  </bean>

  <bean id="cart" class="com.ohgiraffers.common.ShoppingCart" scope="prototype"/>

  <bean id="owner" class="com.ohgiraffers.section02.initdestroy.subsection03.xml.Owner"
        init-method="openShop" destroy-method="closeShop"/>
</beans>
```

실행 파일에서 빈 설정 파일을 기반으로 IoC 컨테이너를 생성하고 쇼핑 카트에 상품을 담는 동일한 코드의 끝에 컨테이너를 종료하는 코드를 추가하여 실행해본다.

```
/* 빈 설정 파일을 기반으로 IoC 컨테이너 생성 */
ApplicationContext context
```

```
  = new GenericXmlApplicationContext("section02/initdestroy/subsection03/xml/spring-context.xml");

...생략

/* init 메소드는 빈 객체 생성 시점에 동작하므로 바로 확인할 수 있지만
 * destroy 메소드는 빈 객체 소멸 시점에 동작하므로 컨테이너가 종료 되지 않을 경우 확인할 수 없다.
 * 가비지 컬렉터가 해당 빈을 메모리에서 지울 때 destroy 메소드가 동작하게 되는데
 * 메모리에서 지우기 전에 프로세스는 종료되기 때문이다.
 * 따라서 아래와 같이 강제로 컨테이너를 종료시키면 destroy 메소드가 동작할 것이다.
 * */
((GenericXmlApplicationContext) context).close();
```

▼ 실행 결과

```
사장님이 가게 문을 열었습니다. 이제 쇼핑을 하실 수 있습니다.
cart1에 담긴 내용 : [붕어빵 1000 Thu Jun 08 22:00:09 KST 2023, 딸기우유 1500 500]
cart2에 담긴 내용 : [지리산암반수 3000 500]
사장님이 가게 문을 닫았습니다. 이제 쇼핑을 하실 수 없습니다.
```

`openShop` 메소드가 `Owner` 객체의 생성 시점에 호출 되고, `closeShop` 메소드가 `Owner` 객체의 소멸 시점에 호출 되었음을 확인할 수 있다.

1-3. Properties

1-3-1. Properties

`Properties` 는 키/값 쌍으로 이루어진 간단한 파일이다. 보통 소프트웨어 설정 정보를 저장할 때 사용된다. 스프링에서는 `Properties` 를 이용하여 빈의 속성 값을 저장하고 읽어올 수 있다.

Properties 파일의 각 줄은 다음과 같은 형식으로 구성된다. 주석은 `#`으로 시작하며, 빈 줄은 무시된다.

```
# 주석
key=value
```

`product-info.properties` 라는 이름의 파일을 생성하고 Product 타입의 값이 될 value를 적절한 key를 설정하여 정의한다.

```
bread.carpbread.name=붕어빵
bread.carpbread.price=1000
beverage.milk.name=딸기우유
beverage.milk.price=1500
beverage.milk.capacity=500
beverage.water.name=지리산암반수
beverage.water.price=3000
beverage.water.capacity=500
```

상품들을 빈으로 등록할 설정 파일을 작성한다. 이 때 `product-info.properties` 파일에 기재한 값으로 상품들의 값을 초기화 하려고 한다. properties 파일을 읽어올 때 `@PropertySource` 어노테이션에 경로를 기재하여 읽어올 수 있으므로 읽어올 properties 파일의 경로를 작성한다.

```
@Configuration
/* resources 폴더 하위 경로를 기술한다. 폴더의 구분은 슬러쉬(/) 혹은 역슬러쉬(\\)로 한다. */
@PropertySource("section03/properties/subsection01/properties/product-info.properties")
public class ContextConfiguration {

}
```

빈 설정 파일 내부에 `@Value` 어노테이션을 사용하여 properties의 값을 읽어온다. `@Value` 어노테이션은 빈의 속성 값을 자동으로 주입받을 수 있는 어노테이션이다.

```
@Configuration
/* resources 폴더 하위 경로를 기술한다. 폴더의 구분은 슬러쉬(/) 혹은 역슬러쉬(\\)로 한다. */
@PropertySource("section03/properties/subsection01/properties/product-info.properties")
public class ContextConfiguration {

  /* 치환자(placeholder) 문법을 이용하여 properties에 저장된 key를 입력하면 value에 해당하는 값을 꺼내온다.
   * 공백을 사용하면 값을 읽어오지 못하니 주의한다.
   * : 을 사용하면 값을 읽어오지 못하는 경우 사용할 대체 값을 작성할 수 있다.
   **/
  @Value("${bread.carpbread.name:팥붕어빵}")
  private String carpBreadName;

  /* 값은 여러 번 불러올 수 있다. */
//  @Value("${bread.carpbread.name:슈크림붕어빵}")
//  private String carpBreadName2;

  @Value("${bread.carpbread.price:0}")
  private int carpBreadPrice;

  @Value("${beverage.milk.name:}")
  private String milkName;

  @Value("${beverage.milk.price:0}")
  private int milkPrice;

  @Value("${beverage.milk.capacity:0}")
  private int milkCapacity;

  @Bean
  public Product carpBread() {

      return new Bread(carpBreadName, carpBreadPrice, new java.util.Date());
  }

  @Bean
  public Product milk() {

      return new Beverage(milkName, milkPrice, milkCapacity);
  }

  @Bean
  public Product water(@Value("${beverage.water.name:}") String waterName,
                       @Value("${beverage.water.price:0}") int waterPrice,
                       @Value("${beverage.water.capacity:0}") int waterCapacity) {

      return new Beverage(waterName, waterPrice, waterCapacity);
  }

  @Bean
  @Scope("prototype")
  public ShoppingCart cart() {

      return new ShoppingCart();
  }
}
```

필드 또는 파라미터에 `@Value` 어노테이션을 사용할 수 있으며 해당 어노테이션에 `${key}` 와 같이 치환자(placeholder) 문법을 이용하여 properties에 저장된 key를 입력하면 value에 해당하는 값을 꺼내와 필드 또는 파라미터에 주입한다. key 뒤에 `:` 을 이용하여 해당 key 값이 없을 경우에는 주입할 기본 값을 입력할 수 있다.

```
/* 빈 설정 파일을 기반으로 IoC 컨테이너 생성 */
ApplicationContext context = new AnnotationConfigApplicationContext(ContextConfiguration.class);

...생략
```

▼ 실행 결과

```
cart1에 담긴 내용 : [붕어빵 1000 Thu Jun 08 23:07:58 KST 2023, 딸기우유 1500 500]
cart2에 담긴 내용 : [지리산암반수 3000 500]
cart1의 hashcode : 716487794
cart2의 hashcode : 987249254
```

해당 빈 설정 파일을 읽어 IoC 컨테이너를 생성하고 컨테이너에서 상품 객체들을 꺼내 쇼핑 카트에 담아 출력하는 코드를 실행해보면 `Properties` 파일의 값이 잘 주입 된 것을 확인할 수 있다.

1-3-2. 국제화

💡 국제화(Internationalization 또는 i18n)란, 소프트웨어를 다양한 언어와 문화권에 맞게 번역할 수 있도록 디자인하는 과정이다. Spring Framework에서 i18n은 `MessageSource` 인터페이스와 property 파일을 이용하여 구현된다. 각 언어별로 property 파일을 정의하면, Spring은 사용자의 로케일에 맞게 적절한 파일을 선택하여 애플리케이션 텍스트를 올바르게 번역할 수 있다.

먼저 `resources` 폴더에 `한국어` 와 `영어` 버전의 에러 메세지를 properties 파일로 정의한다. properties 파일에 `{인덱스}` 의 형식으로 문자를 입력하면 값을 전달하면서 value를 불러올 수 있다. 각 파일의 끝에는 `Locale` 이 적절하게 입력 되어야 한다. 다음은 일반적으로 사용되는 `Locale` 코드들의 목록이다.

언어	국가	코드
한국어	대한민국	ko_KR
영어	미국	en_US
영어	영국	en_UK
일본어	일본	ja_JP
스페인어	스페인	es_ES

```
# 한국어 버전
# message_ko_KR.properties
error.404=페이지를 찾을 수 없습니다!
error.500=개발자의 잘못입니다. 개발자는 누구? {0} 입니다. 현재시간 {1}
```

```
# 영어 버전
# message_en_US.properties
error.404=Page Not Found!!
error.500=something wrong! The developer''s fault. who is devloper? It''s {0} at {1}
```

빈 설정 파일에 `MessageSource` 타입의 빈을 등록한다. 지정 된 명칭이므로 컨테이너에 등록 되는 빈 이름(메소드명)을 정확하게 작성하도록 한다.

```
@Configuration
public class ContextConfiguration {

  @Bean
  public ReloadableResourceBundleMessageSource messageSource() {

    /* 접속하는 세션의 로케일에 따라 자동 재로딩하는 용도의 MessageSource 구현체 */
    ReloadableResourceBundleMessageSource messageSource = new ReloadableResourceBundleMessageSource();

    /* 다국어메세지를 읽어올 properties 파일의 파일 이름을 설정한다. */
    messageSource.setBasename("section03/properties/subsection02/i18n/message");
    /* 불러온 메세지를 해당 시간 동안 캐싱한다. */
    messageSource.setCacheSeconds(10);
    /* 기본 인코딩 셋을 설정할 수 있다. */
    messageSource.setDefaultEncoding("UTF-8");

    return messageSource;
  }
}
```

`ReloadableResourceBundleMessageSource` 은 `MessageSource` 인터페이스의 구현체 중 한 종류로 접속하는 세션의 로케일에 따라 자동 재로딩하는 기능을 가지고 있다. `basename` 속성에 다국어 메세지를 읽어올 properties 파일의 이름을 설정한다. 그 외에도 `MessageSource` 에 대해 다양한 설정을 할 수 있다.

```
ApplicationContext context = new AnnotationConfigApplicationContext(ContextConfiguration.class);

String error404MessageKR = context.getMessage("error.404", null, Locale.KOREA);
String error500MessageKR = context.getMessage("error.500",
  new Object[] {"여러분", new Date()}, Locale.KOREA);

System.out.println("I18N error.404 메세지 : " + error404MessageKR);
System.out.println("I18N error.500 메세지  : " + error500MessageKR);
```

▼ 실행 결과

```
I18N error.404 메세지 : 페이지를 찾을 수 없습니다!
I18N error.500 메세지 : 개발자의 잘못입니다. 개발자는 누구? 여러분 입니다. 현재시간 23. 6. 8. 오후 11:31
```

빈 설정 파일을 읽어와 IoC 컨테이너를 구동시키고 `getMessage` 메소드를 통해 읽어올 메세지의 key 값과 `Locale.KOREA` 라고 하는 Locale을 전달한다. `ReloadableResourceBundleMessageSource` 가 기능하여 한국어 메세지를 로드해오는 것을 출력을 통해 확인할 수 있다. 이 때 `getMessage` 메소드의 두 번째 인자는 전달 값으로 배열로 전달 시 properties 파일에 `{인덱스}` 로 작성했던 영역에 인덱스 순서대로 채워지는 것을 확인할 수 있다.

```
ApplicationContext context = new AnnotationConfigApplicationContext(ContextConfiguration.class);

String error404MessageUS = context.getMessage("error.404", null, Locale.US);
```

```
String error500MessageUS = context.getMessage("error.500",
  new Object[] {"you", new Date()}, Locale.US);

System.out.println("The I18N message for error.404 : " + error404MessageUS);
System.out.println("The I18N message for error.500 : " + error500MessageUS);
```

▼ 실행 결과

```
The I18N message for error.404 : Page Not Found!!
The I18N message for error.500 : something wrong! The developer's fault. who is devloper?
It's you at 6/8/23, 11:31 PM
```

다시 한 번 빈 설정 파일을 읽어와 IoC 컨테이너를 구동시키고 `getMessage` 메소드를 통해 읽어올 메세지의 key 값과 `Locale.US` 라고 하는 Locale을 전달한다. `ReloadableResourceBundleMessageSource` 가 기능하여 영어 메세지를 로드해오는 것을 출력을 통해 확인할 수 있다. `getMessage` 메소드의 두 번째 인자로 전달 된 값도 함께 잘 출력 되는 것을 확인할 수 있다.