의존성 주입

Spring for OOP

- **Exam**

```
public interface MemberPrint {
    void print();
}
```

```
public class KoreanMemberPrint implements MemberPrint{
    public void print() {
        System.out.println("김");
    }
}
```

```
public class EnglishMemberPrint implements MemberPrint{
    public void print() {
        System.out.println("kim");
    }
}
```

Spring for OOP

- **Exam**

```
public class PrintInfo {
    private MemberPrint memberPrint = new KoreanMemberPrint();
}
```

부품 A

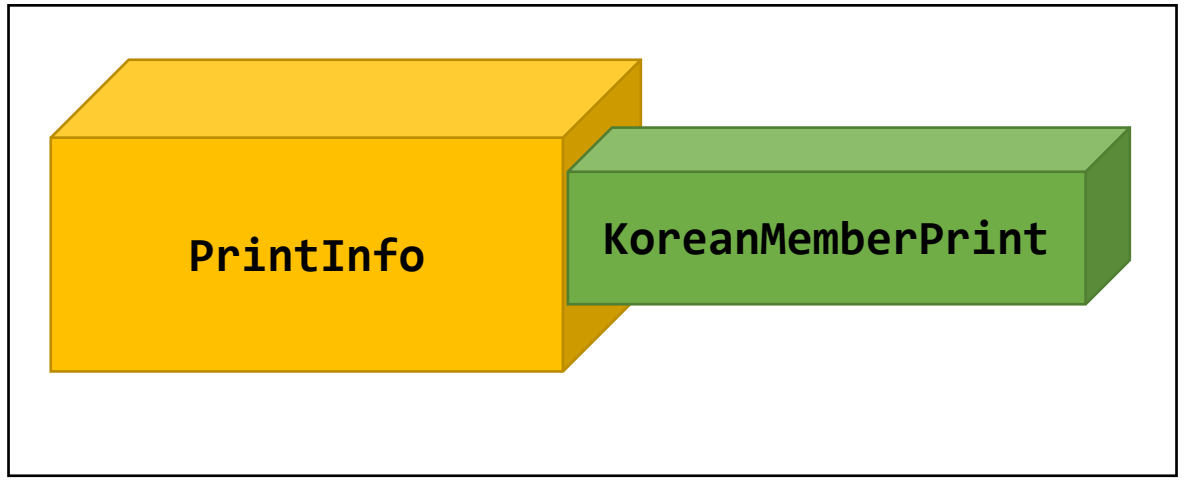

```
public class PrintInfo {
    //private MemberPrint memberPrint = new KoreanMemberPrint();
    private MemberPrint memberPrint = new EnglishMemberPrint();
}
```

부품 B

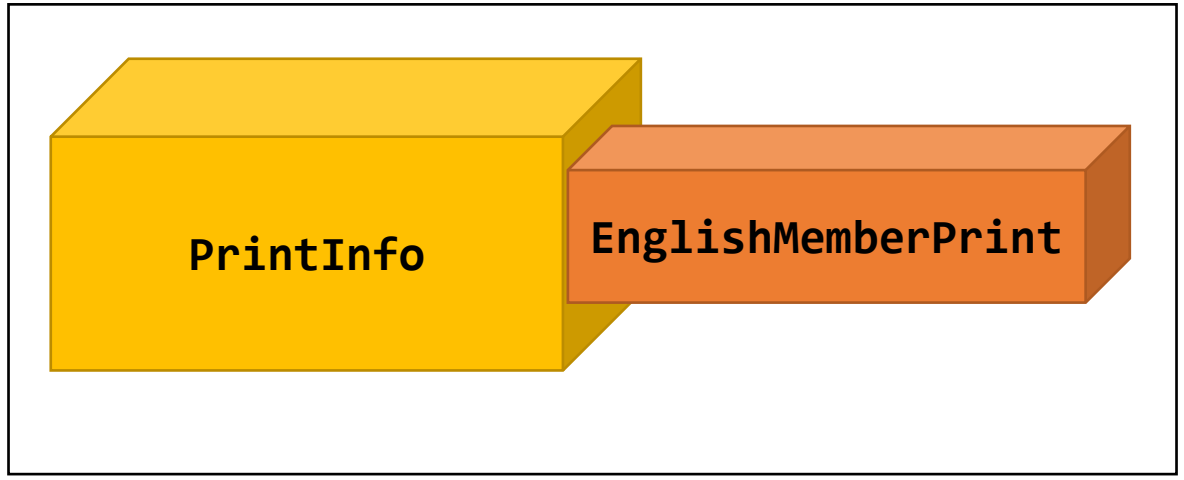
The program codes are modified(the dependency is changed) → build, test, deployment…

Spring framework is solution

의존성 주입(DI)



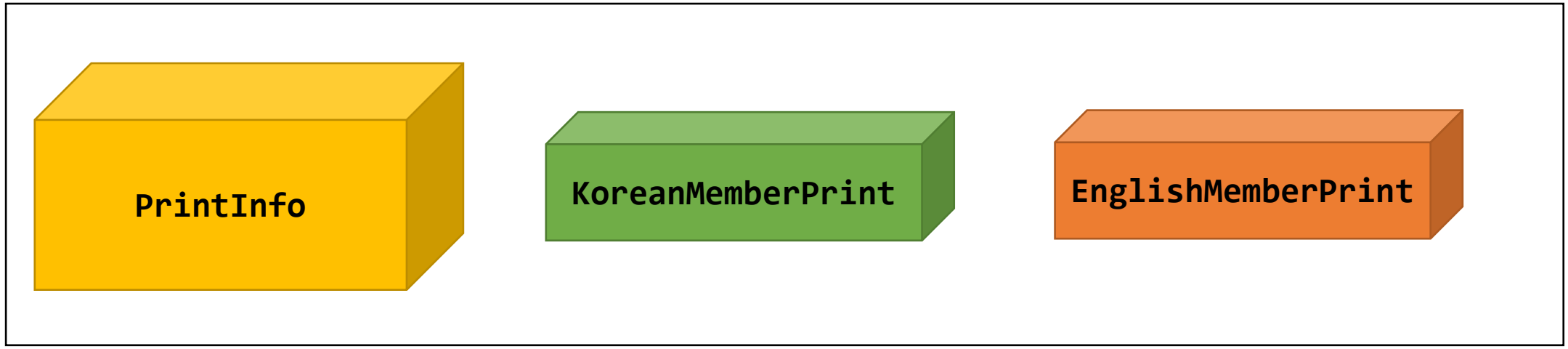
부품 A



부품 B

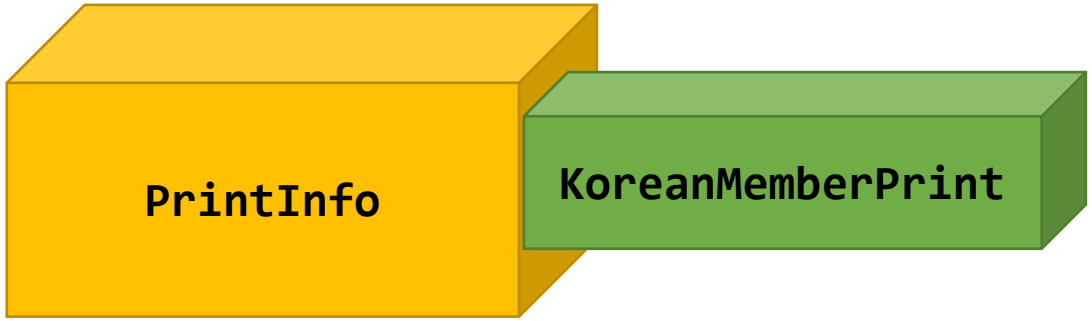
두 컴포넌트가 결합되어 있는 상태
Why? Concrete 클래스가 내부에 포함되어 있다

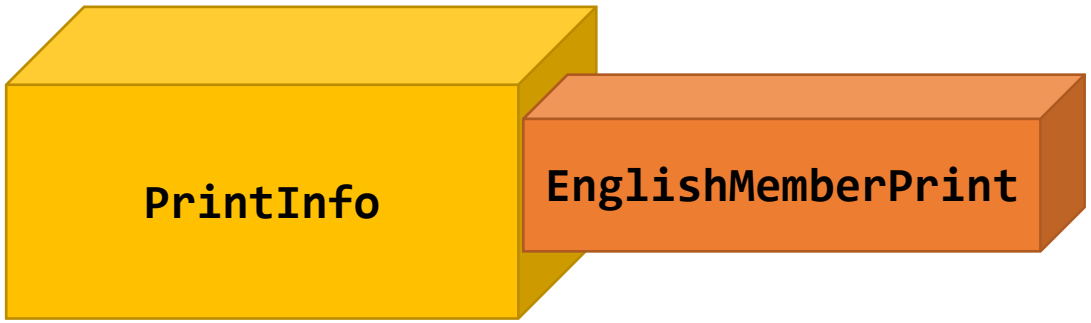
의존성 주입(DI)



부품 조립도
PrintInfo + KoreanMemberPrint

부품 조립도 수정
PrintInfo + EnglishMemberPrint

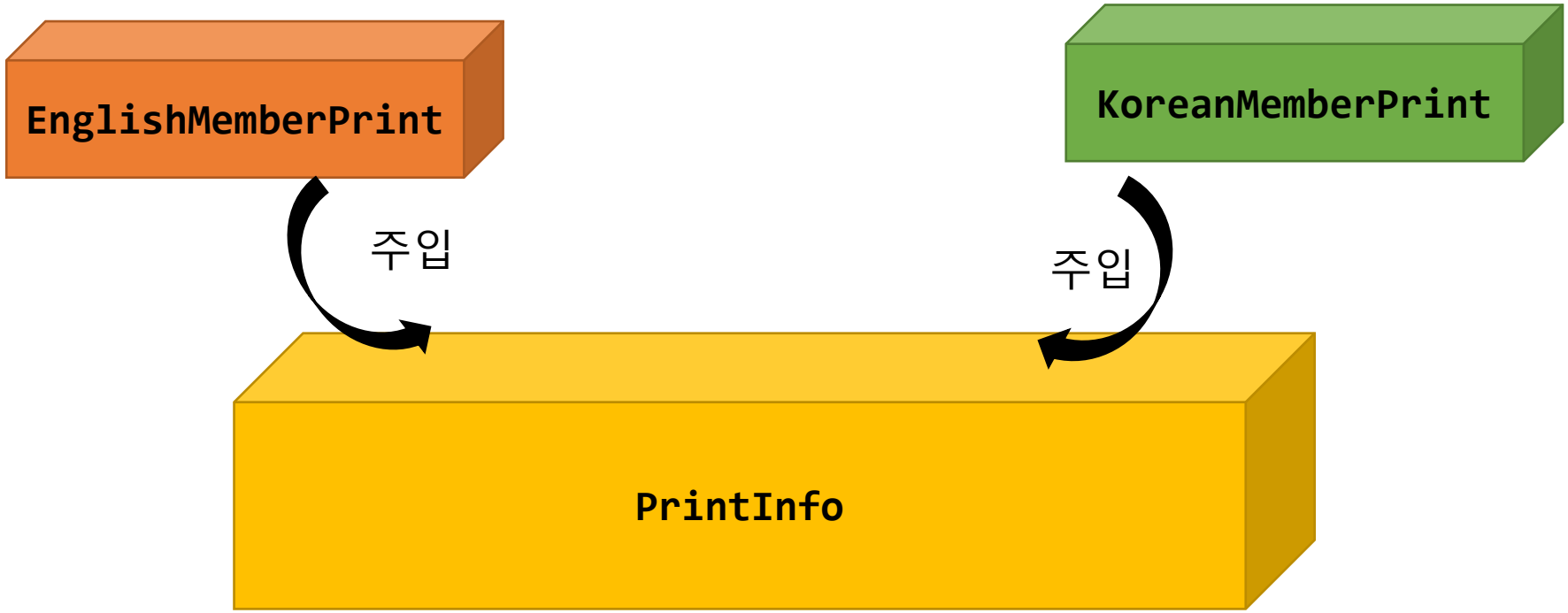




동작하는 클래스와 별개로 객체들의 의존성을 설정하는 외부 클래스(조립도)가 필요

의존관계 주입

- **의존관계 주입 방법(Autowired의 위치에 따른 주입 방법 구분)**
  - 생성자 주입(가장 추천되는 방법)
  - 수정자 주입
  - 필드 주입



의존관계 주입

- **의존관계 주입 방법(생성자 주입)**
  - **@Autowired 생략이 가능한 경우가 있음**
    - 생성자가 하나일 경우, Autowired는 생략 가능(단, 스프링 빈으로 등록된 객체)

```
@Autowired
public PrintInfo(MemberPrint memberPrint) {
    this.memberPrint= memberPrint;
}
```

```
public PrintInfo(MemberPrint memberPrint) {
    this.memberPrint= memberPrint;
}
```

O

```
public PrintInfo(){

}
@Autowired
public PrintInfo(MemberPrint memberPrint) {
    this.memberPrint = memberPrint;
}
```

O

```
public PrintInfo(){

}
public PrintInfo(MemberPrint memberPrint) {
    this.memberPrint = memberPrint;
}
```

X

의존관계 주입

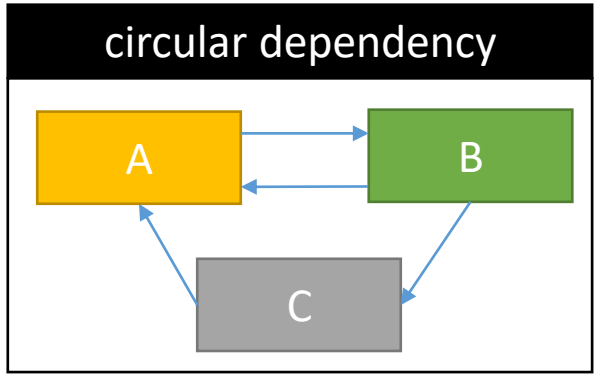
- **의존관계 주입 방법(수정자(setter) 주입)**
  - setter를 이용하여 의존관계 주입
    - 선택, 변경 가능성이 있는 의존관계에 사용

```
@Autowired
public void setMemberPrint(MemberPrint memberPrint) {
    this.memberPrint = memberPrint;
}
```

- **의존관계 주입 방법: 필드(field) 주입**
  - 필드에 @Autowired 어노테이션을 표기
    - 가장 간단한 방법으로 테스트 시 간편하게 사용 가능

```
@Autowired
private MemberPrint memberPrint;
```

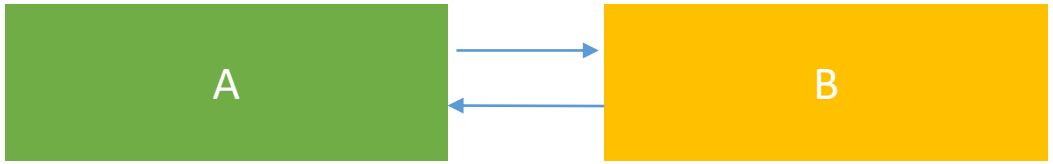
의존관계 주입



- **생성자 주입을 써라**
  - immutable(final 키워드)
    - final 키워드 사용 가능
    - setter를 public으로 지정할 필요 없음
  - 테스트 코드 작성이 편리
    - 프레임워크의 독립적으로 인스턴스화가 가능한 POJO(Plain Old Java Ojbect) 여야 함
    - 테스트: 프레임워크 도움 없는 단위 테스트, 프레임워크 기반 통합 테스트
  - 생성자 주입을 사용할 때 많은 의존성 참조가 발생한다면 클래스 설계가 잘못되었다는 signal로 해석할 수 있음(하나의 객체는 하나의 책임을 가져야 함)
  - 순환 참조를 방지
    - 필드/수정자 주입은 runtime시에, 생성자 주입은 컴파일 시에 발견

의존관계 주입

순환 참조 모습



의존 관계: A → B → C
생성 순서: C → B → A

- **의존 관계를 설정하는 시기**
  - **수정자 주입, 필드 주입**
    - 객체를 만들고 이후에 의존 관계 설정
  - **생성자 주입**
    - 자신의 생성과 연관관계 설정이 동시에 발생
    - A는 B가 필요하고 B는 A가 필요
    - A를 만들기 위해 B를 생성하러 감→B를 생성하러 가니 A가 필요하여 A를 생성하러 감 → …

의존관계 주입

- **Possible Solution**
  - Refactor the design to remove circular dependency
  - Use @Lazy autowiring
  - Use Setter Injection instead of constructor injection

@Lazy autowiring

```
public class Bean1 {

    private Bean2 bean2;

    @Autowired
    public Bean1(@Lazy Bean2 bean2) {
        this.bean2 = bean2;
    }
}
```

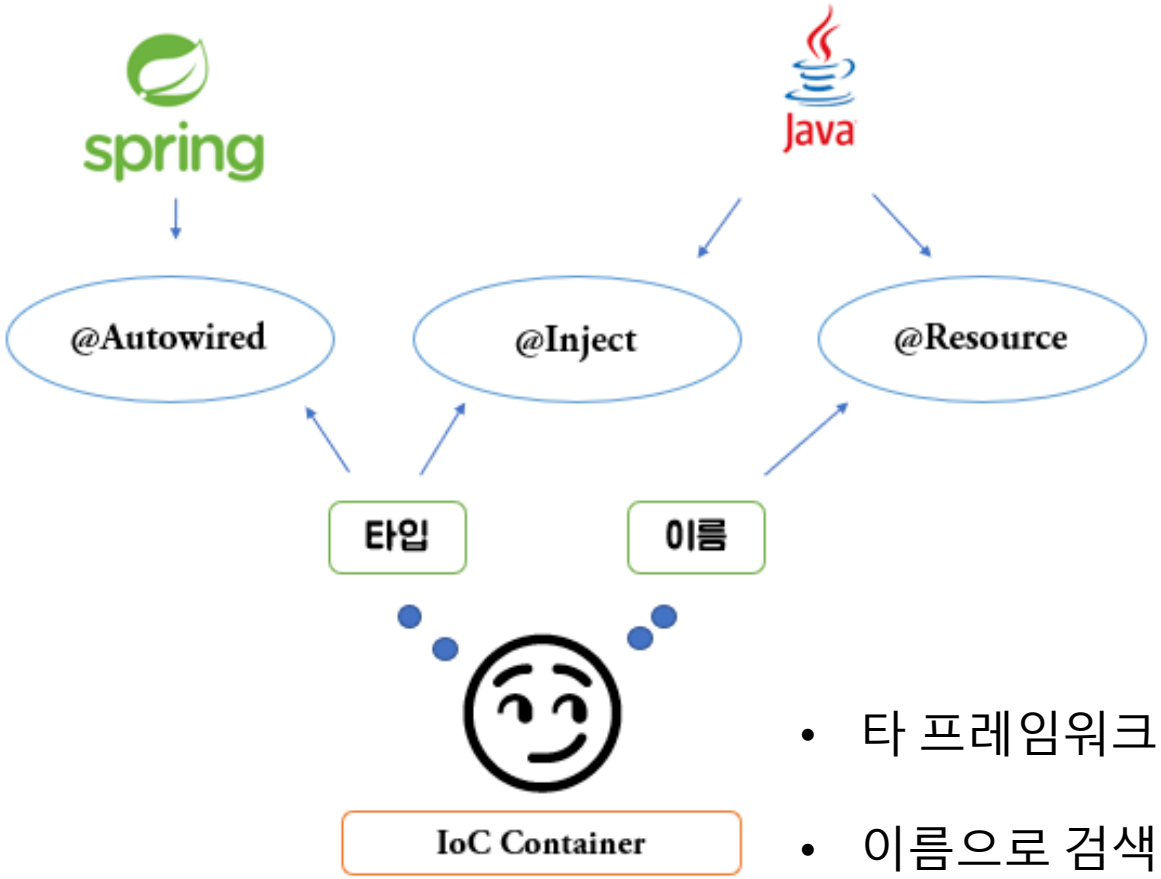
```
public class Bean1 {

    @Lazy
    @Autowired
    private Bean2 bean2;
}
```

의존관계 주입

- **의존관계 주입을 위해 사용하는 어노테이션**
  - @Autowired
    - 이름을 통해 빈을 검색
    - Spring에만 존재하기 때문에 타 프레임워크에서 사용 불가
    - @Autowired는 기본적으로 특정 빈을 찾지 못하면 예외를 던짐(단, required 속성으로 처리 가능)
  - @Resource
    - 자바 진영에서 @Autowired를 참고해 만든 어노테이션
    - 이름을 통한 검색 방식 → POJO가 여럿일 때 대상이 모호하지 않고, 명확
    - @Autowired + @Qualifier
  - @Inject
    - 자바 진영에서 @Autowired를 참고해 만든 어노테이션
    - 타입을 통한 검색방식
    - 타입이 같은 POJO가 여럿일 때 커스텀 어노테이션(custom annotation)을 작성

의존관계 주입

- **의존관계 주입을 위해 사용하는 어노테이션**



	@Autowired	@Resource	@Inject
지원	스프링 프레임워크	자바 (javax.annotation)	자바 (javax.annotation)
검색 방식	타입	이름	이름

- 타 프레임워크로도 호환을 원한다면 **@Resource, @Inject**
- 이름으로 검색→@Resource, 타입으로 검색 → @Autowired, @Inject

Lombok 이용

- **Lombok + 생성자 주입**

```
@RequiredArgsConstructor
```

  - @RequiredArgsConstructor: 초기화 되지 않은 final 필드와 @NonNull 어노테이션이 붙은 필드(반드시 값이 채워져 있어야하는 부분)에 대한 생성자를 생성
    - 의존 객체를 final로 정의

컴포넌트 스캔

https://www.baeldung.com/spring-component-scanning

Component Scan

- **개요**
  - 스프링 컨테이너에서 관리될 빈들을 찾음
  - 클래스를 빈으로 등록하는 방법은 다양하지만 최근에는 어노테이션으로 빈으로 관리될 클래스를 지정
  - @ComponentScan 어노테이션은 스프링에게 스프링 컴포넌트를 찾아야 할 위치를 명시적으로 알려줌
  - @ComponentScan은 @Configuration과 함께 사용됨

Component Scan

```
@Configuration
@ComponentScan
public class SpringComponentScanApp {
   private static ApplicationContext applicationContext;

   @Bean
   public ExampleBean exampleBean() {
      return new ExampleBean();
   }

   public static void main(String[] args) {
      applicationContext =
        new
AnnotationConfigApplicationContext(SpringComponentScanAp
p.class);

      for (String beanName :
applicationContext.getBeanDefinitionNames()) {
         System.out.println(beanName);
      }
   }
}
```

```
package com.baeldung.componentscan.springapp.animals;
// ...
@Component
public class Cat {}
```

```
package com.baeldung.componentscan.springapp.animals;
// ...
@Component
public class Dog {}
```

```
package com.baeldung.componentscan.springapp.flowers;
// ...
@Component
public class Rose {}
```

```
springComponentScanApp
cat
dog
rose
exampleBean
```

Component Scan

- **@ComponentScan for Specific Packages**
  - 모든 클래스를 전부 확인하는 것은 큰 오버헤드
  - ComponentScan 대상의 위치를 지정 가능
    - "집 안에 차키가 있다" 보다는 "안방 화장대에 차키가 있다" 라고 친절하게 전달하는 것이 좋음

```
@ComponentScan(basePackages = "com.baeldung.componentscan.springapp.animals")
@Configuration
public class SpringComponentScanApp {
  // ...
}
```

```
springComponentScanApp
cat
dog
exampleBean
```

Component Scan

- **SpringBoot에서의 Component Scan**
  - @SpringBootApplication
    - @Configuration
    - @EnableAutoConfiguration
    - @ComponentScan

- **@SpringBootApplication이 붙은 클래스의 위치를 기준으로 하위 클래스 스캔 대상이 됨**
- **따라서, @SpringBootApplication이 붙은 클래스를 상단에 두는 것이 필요**