1
디자인 패턴

조장 : 김우철
이정아
구슬기
전연규
장민봉

Index

01 템플릿 메소드

템플릿 메소드란?

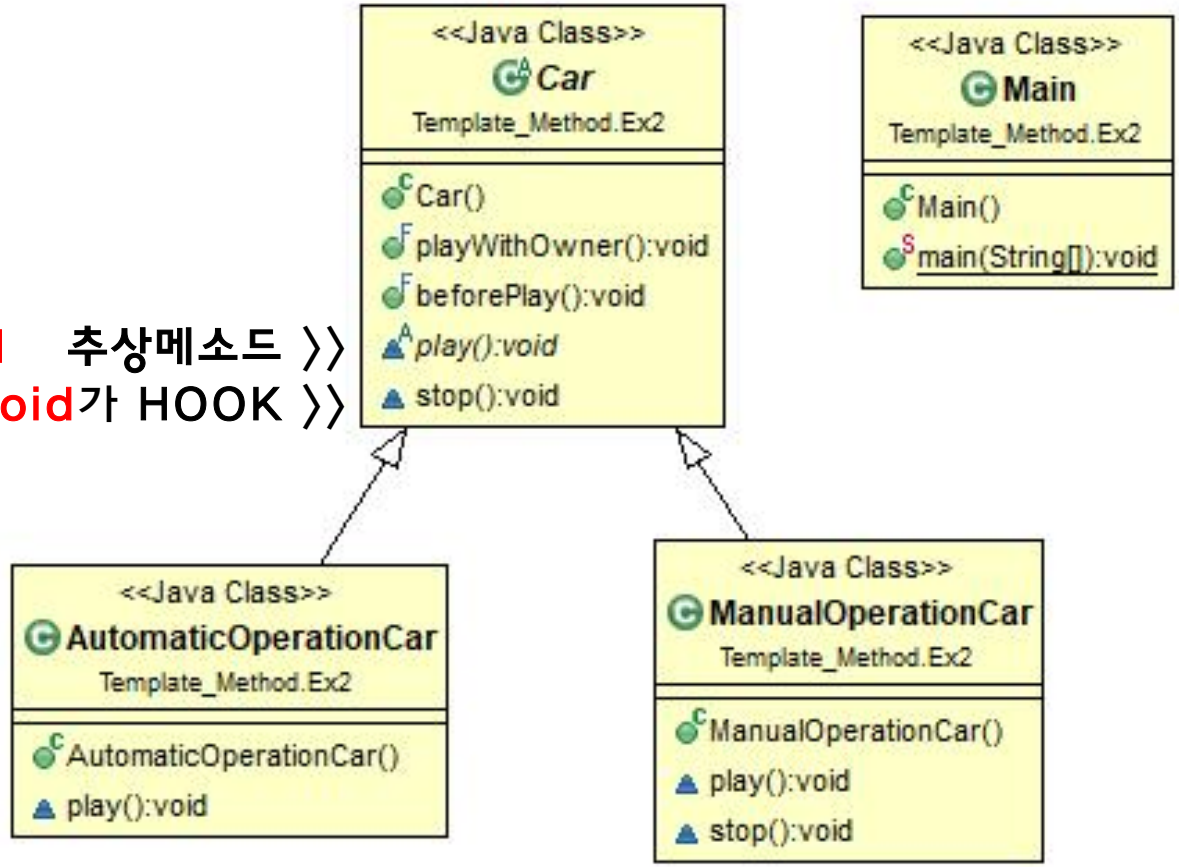
프로그램의 '뼈대'를 정의하는 **행위 디자인 패턴**

동일한 기능을 상위클래스에서 정의하면서
확장/변화가 필요한 부분만 서브 클래스에서 재정의

※ 행위 디자인 패턴 :
객체나 클래스 사이의 알고리즘이나 책임 분배에 관련된 패턴
ex) 옵저버, 스테이트, 전략 패턴 등

템플릿 메소드 클래스 다이어 그램

Play():void 추상메소드 〉〉
stop():void가 HOOK 〉〉



템플릿 메소드 적용 코드 – 추상 클래스

```java
package Template_Method.Ex2;

public abstract class Car {
    // 템플릿 메소드(오버라이딩 불가)
    // 전체적인 알고리즘의 틀을 제공
    public final void playWithOwner() {
        beforePlay();
        play();
        stop();
    }

    public final void beforePlay() {
        System.out.println("시동 켜기");
        System.out.println("사이트 브레이크 해제");
    }

    // 추상 메서드(강제성O)
    abstract void play();

    // Hook메서드(일반 메서드)
    // 알고리즘에서 필수적이지 않은 부분으로서 재정의해도 되고 안해도 됨.
    // 공백이거나 기본 행동을 정의
    // 서브 클래스에서 사용하고 싶을 때만 오버라이딩 (강제성 X)
    void stop() {
        System.out.println("브레이크");
    }

}

```

템플릿 메소드 적용 코드 – 서브 클래스1(HOOK 재정의X)

```java
package Template_Method.Ex2;

public class AutomaticOperationCar extends Car{

    @Override
    void play() {
        System.out.println("Drive D에 기어 놓기");
        System.out.println("자동 기어 변속");
    }

}//Hook은 사용하지 않음

```

템플릿 메소드 적용 코드 – 서브 클래스2(HOOK 재정의)

```java
package Template_Method.Ex2;

public class ManualOperationCar extends Car{

    @Override
    void play() {
        System.out.println("클러치한 상태에서 2단 넣기");
        System.out.println("기어 수동 조작");
    }

    //Hook 메소드 재정의
    void stop() {
        System.out.println("빽빽하게 브레이크~!!");
    }

}
```

템플릿 메소드 적용 코드 – Main

```java
//https://limkydev.tistory.com/81
//채택

package Template_Method.Ex2;

public class Main {
    public static void main(String[] args) {
        Car labo = new ManualOperationCar();
        Car bmw = new AutomaticOperationCar();

        System.out.println("[AutomaticOperationCar]");
        bmw.playWithOwner();
        System.out.println("\n");
        System.out.println("[ManualOperationCar]");
        labo.playWithOwner();

    }

}
```

템플릿 메소드 적용 코드 – 결과

[AutomaticOperationCar]
시동 켜기
사이드 브레이크 해제
Drive D에 기어놓기 〉〉 추상메소드로 재정의
자동 기어 변속 〉〉 추상메소드로 재정의
브레이크

[ManualOperationCar]
시동 켜기
사이드 브레이크 해제
클러치한 상태에서 2단 넣기 〉〉추상메서드로 재정의
기어 수동 조작 〉〉추상메서드로 재정의
뻑뻑하게 브레이크~!! 〉〉HOOK

템플릿 메소드 적용 코드

```
1  package Template_Method.Ex2;
2
3  public class AutomaticOperationCar{
4
5  public void playWithOwner(){
6  System.out.println("시동 켜기");
7  System.out.println("사이드 브레이크 해제");
8
9  System.out.println("Drive D에 기어 놓기");
10 System.out.println("자동 기어 변속");
11
12 System.out.println("브레이크");
13
14 }
15 }
```

》코드 중복
》코드 수정 시 클래스를 일일이 고쳐야 한다

```
1  package Template_Method.Ex2;
2
3  public class ManualOperationCar{
4
5  public void playWithOwner(){
6  System.out.println("시동 켜기");
7  System.out.println("사이드 브레이크 해제");
8
9  System.out.println("클러치한 상태에서 2단 넣기");
10 System.out.println("기어 수동 조작");
11
12 System.out.println("브레이크");
13
14 }
15 }
```

02 팩토리 메소드

팩토리 메소드란?

객체를 생성하기 위한 추상 클래스를 정의한 후
객체를 만들어 내는 부분을
서브 클래스에 위임하는 생성 디자인 패턴

※ 생성 패턴 : 인스턴스를 만드는 절차를 추상화하는 패턴
ex) 싱글톤, 빌더, 추상 팩토리 등

[Template vs Factory]

- 템플릿 메소드 :객체의 행위를 동작하는 공통된 메소드를 만드는 것
- 팩토리 메소드 :객체의 생성을 리턴하는 메소드를 만드는 것

팩토리 메소드를 사용하는 이유

– 객체 간의 결합도를 낮춰 유지보수를 용이 〉〉 다형성 이용

〉〉다형성이란 하나의 타입으로
여러 타입의 객체를 참조할 수 있는 것!

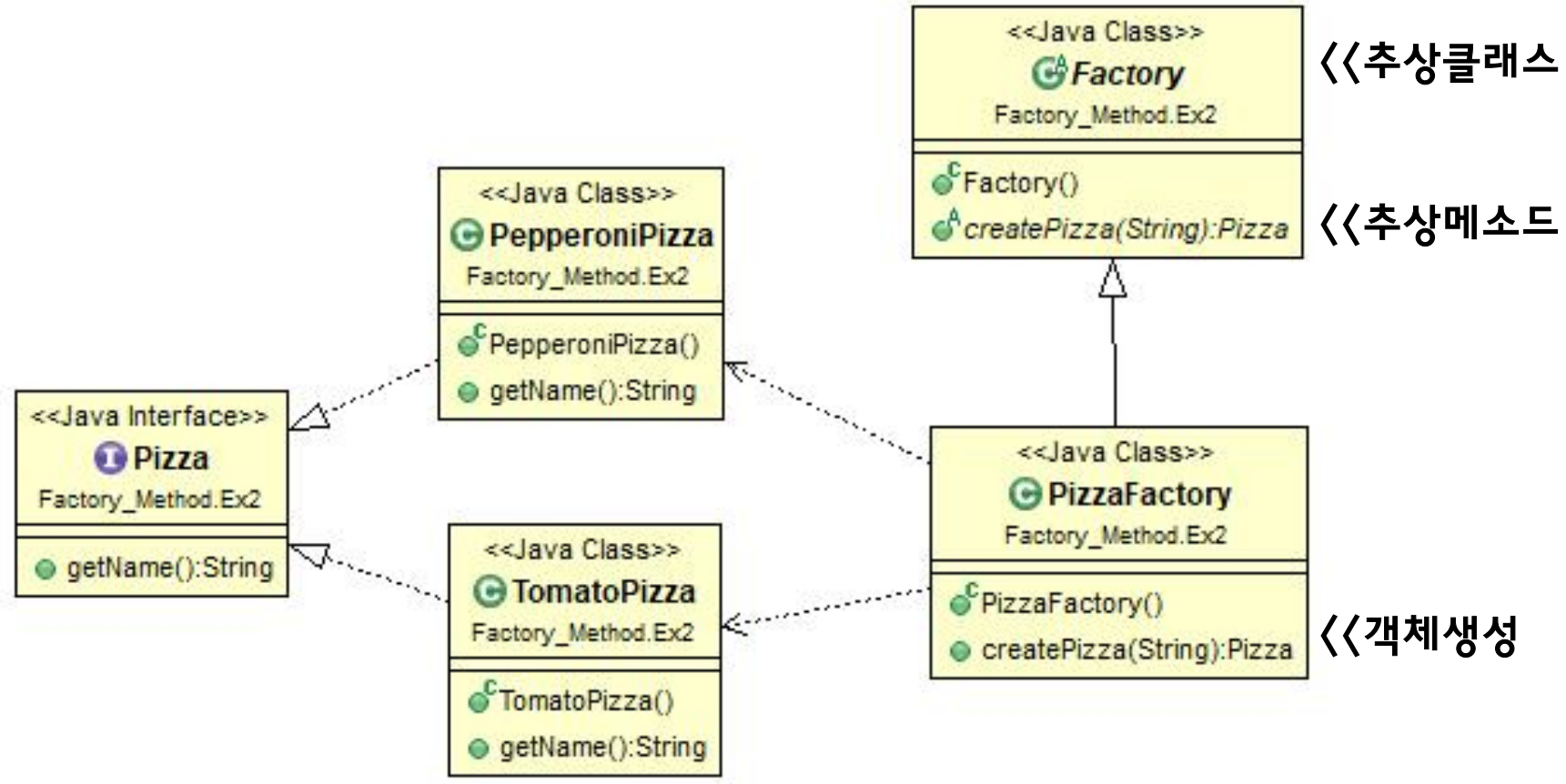
〈'인터페이스'를 이용한 다형성〉

– 인터페이스를 정의해놓고 실제 메소드의 구현은 서브 클래스에서 구현
〉〉 표준화된 규격(메소드)을 해놓음으로서 유지보수 용이

〉〉인터페이스를 통해 인터페이스 구현체와 연결하면,
이후 다른 인터페이스 구현체로의 변경이나 확장이 용이
(List를 구현한 ArrayList, LinkedList, Stack 등)

```
ex) List list = new ArrayList();
    List list = new LinkedList();
    List list = new Stack();
```

팩토리 메소드 클래스 다이어 그램



팩토리 메소드 적용 코드 – 인터페이스

```
1  package Factory_Method.Ex2;
2
3  public interface Pizza {
4      public String getName();
5  }
```

팩토리 메소드 적용 코드 – 서브 클래스1(Interface)

```java
package Factory_Method.Ex2;

public class TomatoPizza implements Pizza {
    @Override
    public String getName() {
        return "TomatoPizza";
    }
}
```

팩토리 메소드 적용 코드 – 서브 클래스2(추상)

```
1  package Factory_Method.Ex2;
2
3  public class PepperoniPizza implements Pizza {
4      @Override
5      public String getName() {
6          return "PepperoniPizza";
7      }
8  }
```

팩토리 메소드 적용 코드 – 추상 클래스

```
1  package Factory_Method.Ex2;
2
3  public abstract class Factory {
4      public abstract Pizza createPizza(String name);
5  }
```

팩토리 메소드 적용 코드 – 서브 클래스3(추상)

```java
package Factory_Method.Ex2;

public class PizzaFactory extends Factory {
    public Pizza createPizza(String name) {
        switch (name) {
        case "Tomato":
            return new TomatoPizza();
        case "Pepperoni":
            return new PepperoniPizza();

        }
        return null;
    }
}
```

객체 생성의 캡슐화 = '정보은닉'

〉〉 추상클래스에는 외부에서 보고 접근가능한 정보만 공개(createPizza 메소드명)하고 실제구현은 내부로 숨기는 것

코드 예시)

```
List list = newArrayList(
);
list.add(1);
```

실생활 예시) 자동차 브레이크의 자세한 과정, 차가 움직일때 엔진의 원리

Factory 추상클래스를 상속받은 후, 객체생성 메소드 내부 로직을 구현

객체 생성은 오직 하나의 메소드에서만 담당

〉〉 한 곳에서만 관리하면 되므로 객체 생성에 관한 확장 용이

〉〉 코드의 중복 방지

〉〉 리턴값으로 서로 다른 객체를 반환 〉 객체 선택의 유연함

팩토리 메소드 적용 코드 – Main

```java
package Factory_Method.Ex2;

import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        Factory pizzaFactory = new PizzaFactory();
        Pizza pizza1 = pizzaFactory.createPizza("Tomato");
        Pizza pizza2 = pizzaFactory.createPizza("Pepperoni");

        System.out.println(pizza1.getName() + "\n");
        System.out.println(pizza2.getName() + "\n");

    }
}
```

[출력결과]

```
TomatoPizza

PepperoniPizza
```

〉〉 필요한 객체를 new를 통해 직접 생성하지 않고 팩토리 메서드 클래스에 요청

〉〉 팩토리 메서드 클래스에서 생성한 객체를 반환 받아 사용

〉〉 구상 클래스(Tomato, Pepperoni)에 의존하지 않고 객체 생성을 추상 메소드에 의존, 유지보수 용이 및 결합도 감소

03 추상 팩토리 메소드

추상 팩토리란?

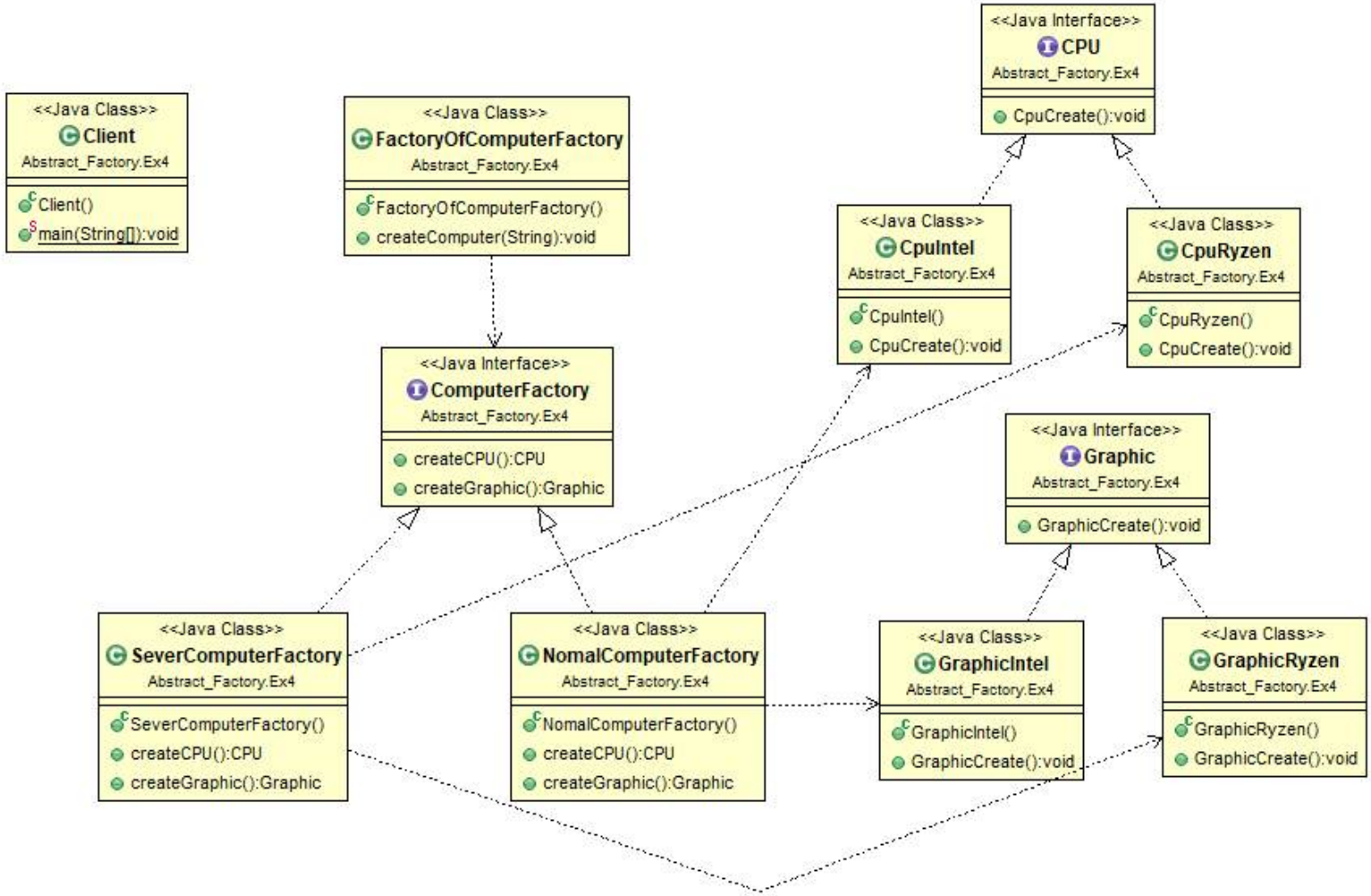
관련성 있는 여러 종류의 객체를 특정 그룹으로 묶어서
팩토리 클래스로 만들고,
이들 팩토리를 조건에 따라 생성 하도록 다시 팩토리를 만들어서
객체를 생성

〉〉 팩토리 메서드 패턴을 좀 더 캡슐화 한 방식

[팩토리 메소드 vs 추상 팩토리]

- 팩토리 메소드 : 하나의 메소드가 여러 종류의 객체를 생성
- 추상 팩토리 : 객체 별 한 종류의 객체를 생성
  즉, "객체의 집합"을 생성

추상 팩토리 클래스 다이어그램



추상 팩토리 적용 코드 – CPU

```java
package Abstract_Factory.Ex4;

public interface CPU {
    public void CpuCreate();
}
```

```java
package Abstract_Factory.Ex4;

public class CpuIntel implements CPU {
    @Override
    public void CpuCreate() {
        System.out.println("인텔 CPU 생성");
    }
}
```

```java
package Abstract_Factory.Ex4;

public class CpuRyzen implements CPU {
    @Override
    public void CpuCreate() {
        System.out.println("라이젠 CPU 생성");
    }
}
```

추상 팩토리 적용 코드 – Graphic

```java
package Abstract_Factory.Ex4;

public interface Graphic {
    public void GraphicCreate();
}
```

```java
package Abstract_Factory.Ex4;

public class GraphicIntel implements Graphic {
    public void GraphicCreate(){
        System.out.println("인텔 그래픽 카드 생성");
    }
}
```

```java
package Abstract_Factory.Ex4;

public class GraphicRyzen implements Graphic {
    public void GraphicCreate() {
        System.out.println("라이젠 그래픽 카드 생성");
    }
}
```

추상 팩토리 적용 코드 – Computer Factory

```
1  package Abstract_Factory.Ex4;
2
3  public interface ComputerFactory {
4      public CPU createCPU();
5      public Graphic createGraphic();
6  }
```

추상 팩토리 적용 코드 – Normal Computer Factory

```
1  package Abstract_Factory.Ex4;
2
3  public class NomalComputerFactory implements ComputerFactory {
4      public CPU createCPU() {
5          return new CpuIntel();
6      };
7      public Graphic createGraphic() {
8          return new GraphicIntel();
9      };
10 }
```

추상 팩토리 적용 코드 – Server Computer Factory

```
1  package Abstract_Factory.Ex4;
2
3  public class SeverComputerFactory implements ComputerFactory {
4      public CPU createCPU() {
5          return new CpuRyzen();
6      };
7
8      public Graphic createGraphic() {
9          return new GraphicRyzen();
10     };
11 }
```

추상 팩토리 적용 코드 – 컨슈머 클래스

```
1  package Abstract_Factory.Ex4;
2
3  public class FactoryOfComputerFactory {
4      public void createComputer(String type) {
5          ComputerFactory computerFactory = null;
6          switch (type) {
7          case "일반":
8              computerFactory = new NomalComputerFactory();
9              break;
10
11         case "서버":
12             computerFactory = new SeverComputerFactory();
13             break;
14
15         }
16
17         computerFactory.createCPU().CpuCreate();
18         //CPU cpu = computerFactory.createCPU();
19         //cpu.CpuCreate();
20         computerFactory.createGraphic().GraphicCreate();
21         //Graphic graphic = computerFactory.createGraphic();
22         //graphic.GraphicCreate();
23     }
24
25 }
```

추상 팩토리 적용 코드 – Main

```
1  package Abstract_Factory.Ex4;
2  public class Client {
3      public static void main(String[] args) {
4          FactoryOfComputerFactory factoryOfComputerFactory = new FactoryOfComputerFactory();
5          System.out.println("[일반 PC]");
6          factoryOfComputerFactory.createComputer("일반");
7          System.out.println("--------------------------------");
8          System.out.println("[서버용 PC]");
9          factoryOfComputerFactory.createComputer("서버");
10     }
11 }
```

[출력문]〉〉

```
[일반 PC]
인텔 CPU 생성
인텔 그래픽 카드 생성
--------------------
[서버용 PC]
라이젠 CPU 생성
라이젠 그래픽 카드 생성
```