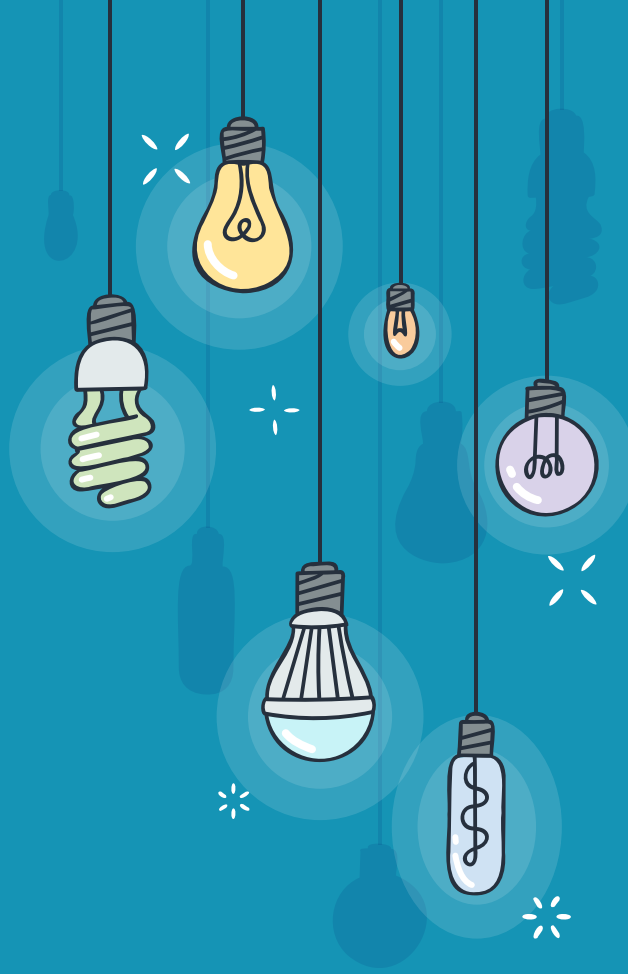
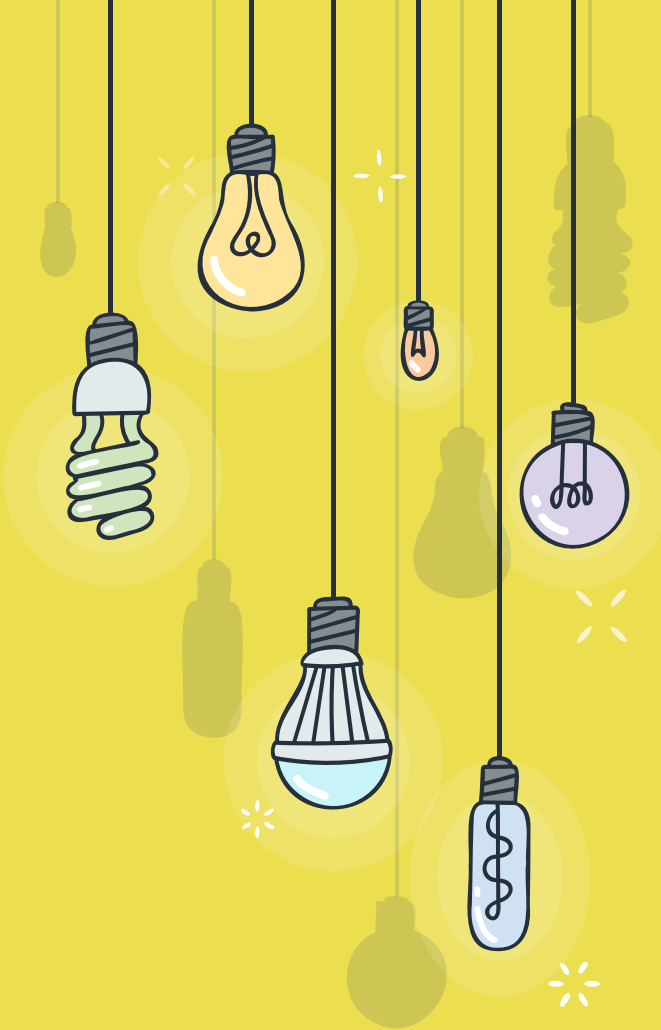
JAVA

1. 객체지향언어
2. 클래스와 객체
3. 필드
4. 메서드
5. 생성자
6. 정적멤버와 인스턴스멤버
7. 패키지와 제어자



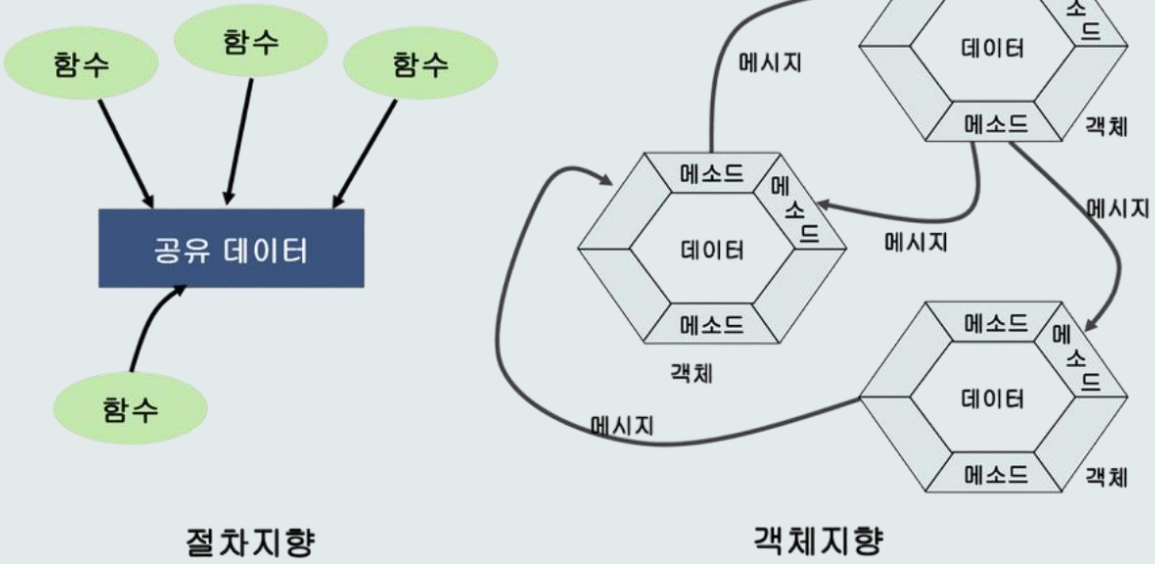
1 객체지향언어



객체지향언어의 역사

- 1960년대 최초의 객체지향언어 Simula탄생
  - 노르웨이에서 개발된 최초의 객체 지향 언어
  - 객체, 클래스, 상속 등의 개념을 처음으로 도입
- 1980년대
  - Smalltalk, C++등
  - 사용자 폭이 넓지 못함
- 자바(Java) 탄생(1995년)
  - 인터넷의 발전과 함께 크게 유행
    - 빠르게 변화하는 사용자들의 요구 상황을 절차적인 언어로 극복하기 어려움을 느낌
  - 객체지향언어는 언어의 주류로 자리잡게 됨

객체지향 언어 VS 절차 지향 언어

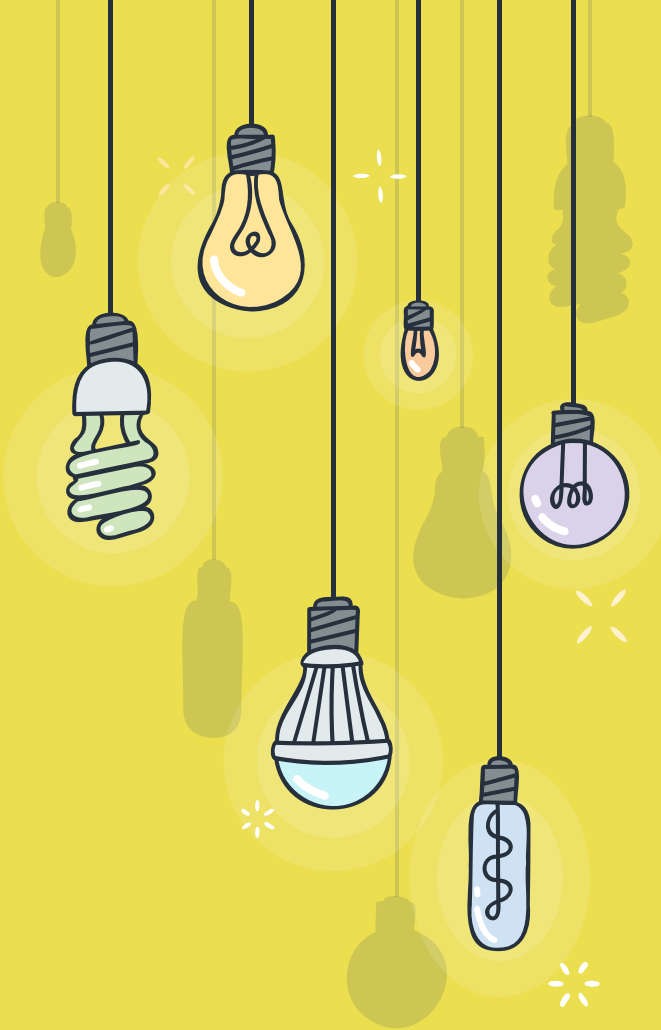


절차지향 객체지향

객체지향 언어의 특징

- 코드의 재사용성이 높다.
  - 한 번 작성한 클래스나 객체를 다른 프로그램에서도 쉽게 재사용할 수 있다
- 코드의 관리가 용이하다.
  - 코드를 객체 단위로 분할하고 모듈화하여 관리하기 쉽게 만든다
- 신뢰성이 높은 프로그래밍을 가능하게 한다.
  - 객체 지향 언어는 캡슐화, 상속, 다형성 등의 개념을 통해 신뢰성이 높은 프로그래밍을 가능하게 한다

2 클래스와 객체



클래와 객체의 개념

- 현실세계 : 설계도 → 건물
- 자　　바 : 클래스 → 객체

- 객체
  - 클래스로부터 만들어진 클래스의 인스턴스(instance)
  - 하나의 클래스로부터 여러 개의 인스턴스를 만들 수 있음

개발자

클래스

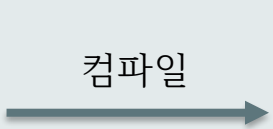
객체

클래스 선언

- 소스 파일당 하나의 클래스를 선언하는 것이 관례
  - 두 개 이상의 클래스도 선언 가능
  - 소스 파일 이름과 동일한 클래스만 public으로 선언 가능
  - 선언한 개수만큼 바이트 코드 파일이 생성

Car.java

```
public class Car {
  }

class Tire {
  }
```



Car.class
Tire.class

클래스 선언

- 클래스 이름 작성 규칙
  - 숫자로 시작 불가
  - 특수기호는 _와 $만 사용 가능(권장하지 않음)
  - 자바 키워드는 사용 불가
  - 첫 글자는 대문자로 시작하는 것이 관례

클래스의 용도

- 라이브러리(API: Application Program Interface) 용
  - 자체적으로 실행되지 않음
  - 다른 클래스에서 이용할 목적으로 만든 클래스

- 실행용
  - main() 메소드를 가지고 있는 클래스로 실행할 목적으로 만든 클래스

애플리케이션 = 1개의 실행클래스 + n개의 API클래스

객체의 구성요소

- 객체는 속성과 기능으로 이루어져 있음
  - 속성 = 필드
  - 기능 = 메서드
  - 생성자 : 필드의 초기화

속성	기능
• 회사명	• 전원 켜기
• 모델명	• 전원 끄기
• 색상	• 음량 올리기
• 음량 등	• 음량 내리기 등

```
Class phone {
    String compony;
    String model;
    String color;
    int volume;
    boolean power

    void power() { power = !power }
    void volumeUp() { volume += 1 }
    void volumeDown() { volume -= 1 }
}
```

속성

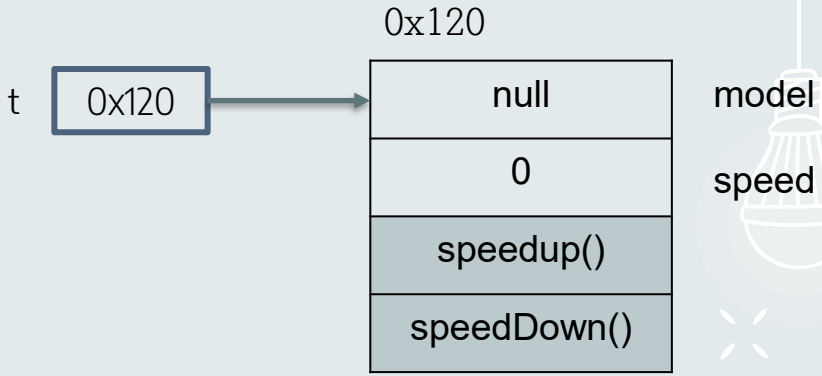
기능

인스턴스 생성과 사용

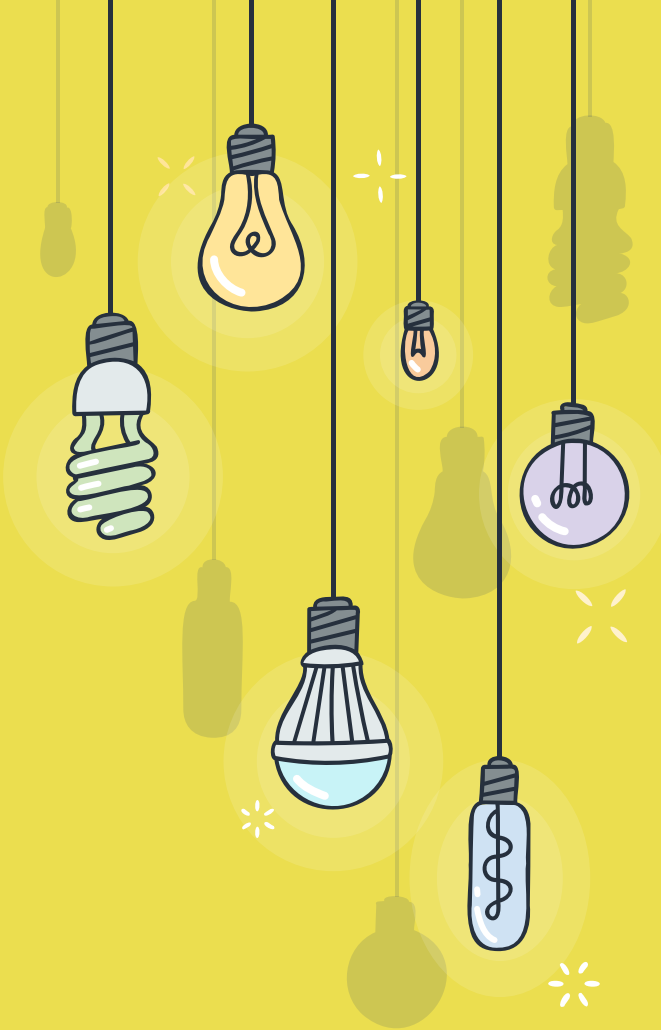
클래스로 인스턴스 생성

```
클래스명 참조변수명;
참조변수명 = new 클래스명();

클래스명 참조변수명 = new 클래스명
```

```
Tv t;
t = new Tv();

Tv t = new Tv();
```



3 필드



필드

- 필드
  - 객체의 고유 데이터
  - 클래스 블록 내부에 선언

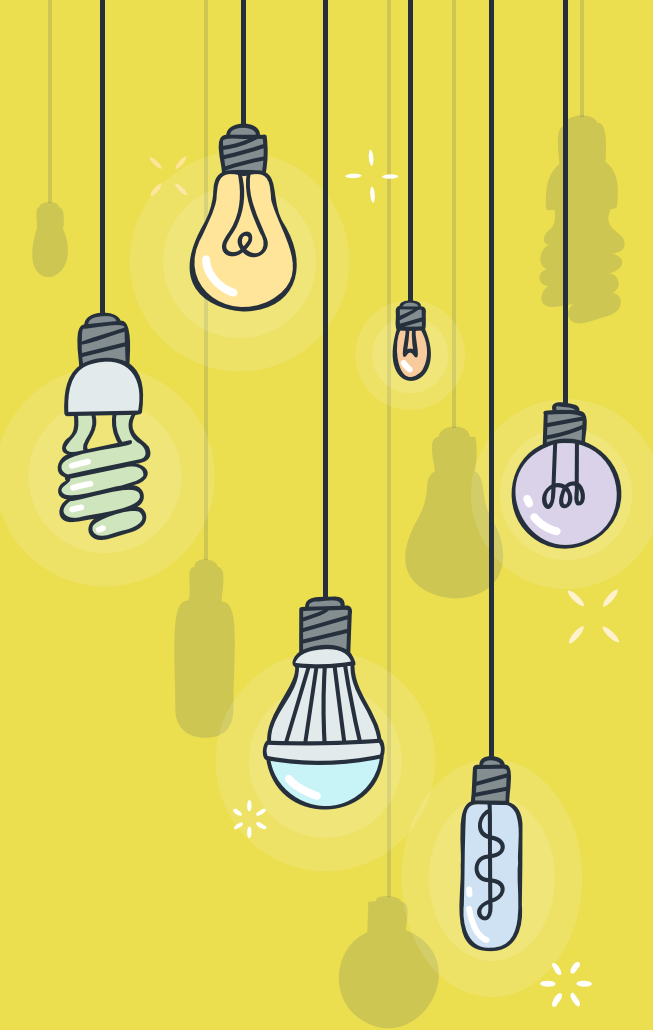
```
타입 필드명;
타입 필드명 = 값;
```

필드의 초기화

- 변수는 초기값을 지정할 수도 있고
- 초기값을 지정하지 않고 객체 생성시 자동으로 기본 초기값을 가질 수 있다

분류		타입	초기화
기본타입	정수타입	byte, short, int	0
		char	\u0000(빈공백)
		long	0L
	실수타입	float	0.0F
		double	0.0
	논리타입	boolean	false
참조타입		배열, String, 인터페이스, 클래스	null

4 메서드



메서드

- 메서드란
  - 특정 기능을 정의한 코드들의 집합
- 메서드의 장점
  - 중복 코드의 사용 줄임
  - 손쉬운 유지보수
- 메서드의 선언

선언부 / 구현부

```
접근제어자 리턴타입 메소드명(매개변수 선언) {
    // 실행할 코드 작성
    return 반환값;
}
```

메서드

- 접근제한자
  - 메소드에 접근할 수 있는 범위 지정
  - 종류
    - public : 모든 범위에서 사용가능
    - protected : 같은 패키지 또는 자식 클래스에서 사용가능
    - default : 같은 패키지안의 모든 클래스에서 사용 가능(생략시)
    - private : 클래스 내부에서만 사용가능

메서드

- return문
  - 현재 실행 중인 메서드를 종료하고 호출한 메서드로 되돌아간다
  - 메서드의 정상 종료
    - 메서드의 블럭{}의 끝에 도달했을 때
    - 메서드의 블럭{}을 수행 도중 return문을 만났을 때
  - 리턴타입
    - 메소드가 모든 작업을 마치고 반환하는 결과의 데이터 타입 명시
    - 결과값이 없을 때 반환타입에 void 작성

```
리턴타입 메소드명(매개변수 선언){
    // 실행할 코드 작성
    return 반환값;
}
```

메서드

- 매개변수(parameters)
  - 메소드 호출 시 전달되는 인수의 값을 저장할 변수
  - 메소드 수행에 필요한 입력값 저장
  - 여러 개의 입력 값 저장 가능
    - 입력 시 반드시 자료형과 개수가 맞아야 됨

```
리턴타입 메소드명(매개변수 선언) {
    // 실행할 코드 작성
    return 반환값;
}
```

메서드 오버로딩(OVERLOADING)

- 메서드 오버로딩
  - 같은 이름의 메서드를 여러 개 선언
  - 반드시 매개변수의 타입 또는 개수가 달라야 함

```
int plus(int x, int y) {
    return x + y;
}

double plus(int x, double y) {
    return x + y;
}
```

가변인자

- 매개변수를 동적으로 지정
  - 가변인자는 반드시 맨 마지막 매개변수여야 함

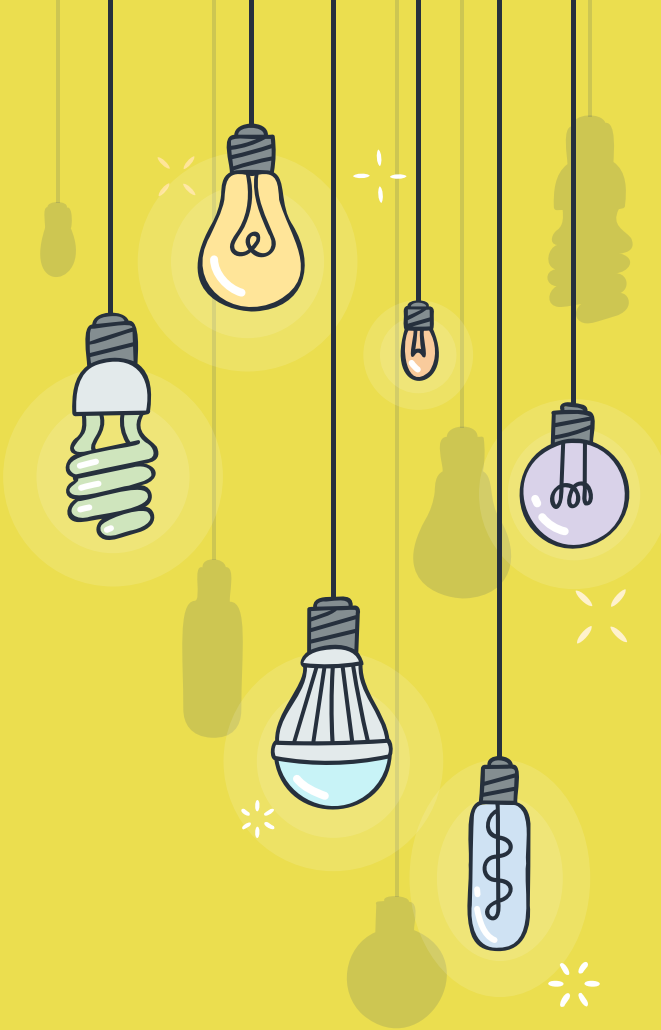
```
void mothod(String v1, int v2, String... str ) {  ... }
```

  - 오버로딩 사용시 중복 코드 줄임 효과

```
void mothod(String v1) {  ... }
void mothod(String v1, String v2) {  ... }
void mothod(String v1, String v2, String v3) {  ... }
```

```
void mothod(String... str) {  ... }
```

5 생성자



생성자(CONSTRUCTOR)

- 생성자(constructor)
  - 객체 생성시 단 한번 호출되어 필드의 초기화 담당

- 기본 생성자(default constructor)
  - 클래스에 생성자를 선언하지 않아도 객체 생성시 기본생성자(빈 생성자)를 컴파일러가 자동 추가해줌

```
Person person1 = new Person();
```

Person.java

```
public class Person {

}
```

Person.class

```
public class Person {
        public Person() { }     // 자동추가
}
```

생성자(CONSTRUCTOR)

- 생성자 선언
  - 매개변수는 생략할 수도 있고 여러 개 선언할 수도 있음

```
public class Person {
    Person(String name, int age, String phone) {
        .....
    }
}
```

  - 객체 생성시 반드시 선언된 생성자를 호출하여 객체 생성

```
Person person1 = new Person("홍길동", 25, "010-1234-5678");
```

생성자(CONSTRUCTOR)

- 생성자로 필드 초기화

```
public class Person {
    String name;
    int age;
    String phone;

    Person(String n, int a, String p) {
        name = n;
        age = a;
        phone =p;
    }
}
```

```
Person person1 = new Person("홍길동", 25, "010-1234-5678");
Person person12= new Person("아무개", 23, "010-2345-6789");
```

THIS

- this
  - 생성자의 매개변수 이름은 필드 이름과 유사하거나 동일하게 사용하는 것을 권장
  - 매개변수 이름과 필드 이름이 동일 할 경우 this.필드로 표현

```
public class Person {
    String name;
    int age;
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

생성자 오버로딩

- 생성자 오버로딩
  - 생성자의 매개변수의 타입이나 개수를 다르게 선언

```
public class Person {
    Person() { ..... }
    Person(String name) { ..... }
    Person(String phone) { ..... }  // 매개변수 타입과 개수 같음
    Person(String name, int age) { ..... }
    Person(String name, int age, String phone) { ..... }
}
```

생성자 안에서의 this()

- 생성자 안에서 this()
  - 다른 생성자 호출
  - this() 사용시 반드시 생성자 첫 줄에 표기
  - 초기화의 중복코드를 효과적으로 줄여줌

```
public class Person {
    Person(String name) {
        this(name, 20, “010-1111-2222”);  // 다른 생성자 호출
        …
    }
}
```

생성자 안에서의 this()

```
public class Person {
    Person(String name) {
        this.name = name;
        this.age = 20;
        this.phone = "010-1111-2222";
    }
    Person(String name, int age) {
        this.name = name;
        this.age = age;
        this.phone = "010-1111-2222";
    }
    Person(String name, int age, String phone) {
        this.name = name;
        this.age = age;
        this.phone = phone;
    }
}
```

중복코드

중복코드

중복코드

```
public class Person {
    Person(String name) {
        this(name, 20, "010-1111-2222");
    }
    Person(String name, int age) {
        this(name, age, "010-1111-2222");
    }
    Person(String name, int age, String phone) {
        this.name = name;
        this.age = age;
        this.phone = phone;
    }
}
```

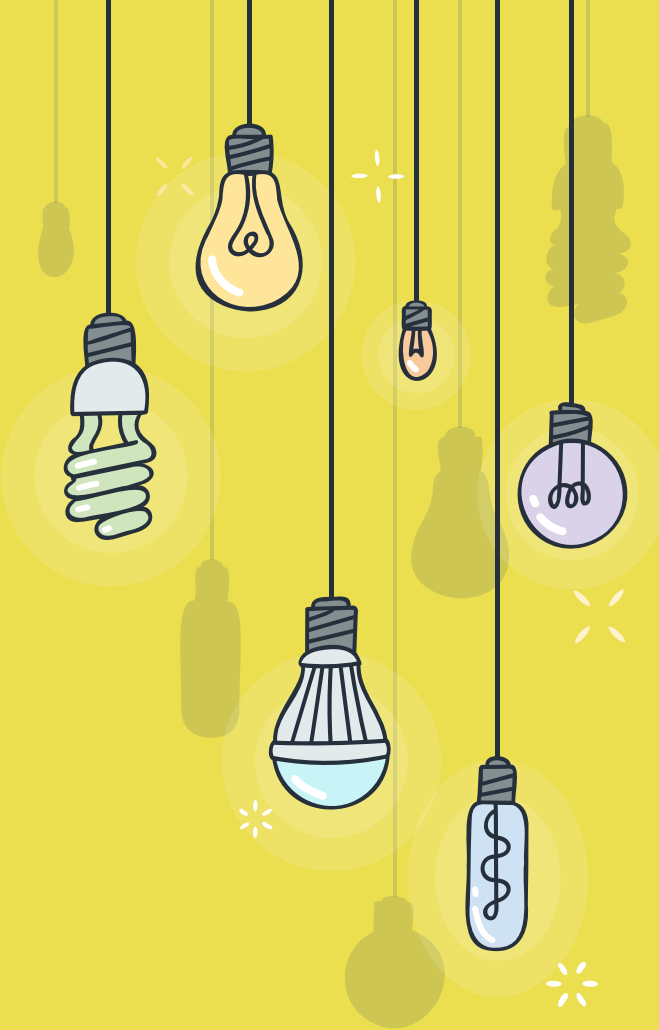
생성자 호출

생성자 호출

공통 실행 코드

6

정적 멤버와 인스턴스 멤버



정적 멤버와 인스턴스 멤버

- 인스턴스 멤버
  - 객체마다 가지고 있는 멤버
    - 인스턴스 필드
      - 힙 영역의 객체 마다 가지고 있는 멤버
      - 객체마다 다른 데이터 저장
    - 인스턴스 메소드
      - 객체가 있어야 호출 가능한 메소드
- 정적 멤버
  - 객체와 상관없는 멤버
    - 정적 필드 및 상수
      - 객체 없이 클래스만으로 사용 가능한 필드
    - 정적 메소드
      - 객체 없이 클래스만으로 호출 가능한 메소드

필드의 종류

종류	선언위치	사용
정적 필드	클래스 영역	클래스이름.클래스변수명
인스턴스 필드		참조변수.인스턴스변수명
지역변수	메서드 영역	변수명 (메서드 블록을 벗어나면 소멸하여 사용불가)

클래스 영역

메소드 영역

```
class Test {
  int a;
  static int b;

  void method() {
    int a = 1;
  }
}
```

인스턴스 필드

정적 필드

지역 변수

```
Test test = new Test();

test.a;  // 인스턴스 필드 호출

Test.b;  // 정적 필드 호출
```

필드의 종류

- 정적 필드
  - 같은 클래스의 모든 인스턴스들이 공유하는 필드
  - 클래스가 로딩될 때 생성되고 프로그램이 종료될 때 소멸
- 인스턴스 필드
  - 각 인스턴스의 개별적인 저장공간
  - 인스턴스마다 다른 값 저장가능
  - 인스턴스를 생성할 때 생성, 참조변수가 없을 때 가비지컬렉터에 의해 자동 제거됨
- 지역 변수
  - 메서드 내에 선언되며, 메서드의 종료와 함께 소멸

필드 초기화 시기와 순서

- 정적필드 초기화 시점 : 클래스가 처음 로딩될 때 한번
- 인스턴스필드 초기화 시점 : 객체가 생성될 때 마다

정적 필드 초기화		인스턴스 필드 초기화		
기본값	명시적 초기화	기본값	명시적 초기화	생성자
sf 0	sf 1	sf 1 insf 0	sf 1 insf 1	sf 1 insf 2
1	2	3	4	5

```
Test test = new Test();
```

```
class Test {
    static int sf = 1;
    int insf = 1

    Test() {
        insf = 2;
    }
}
```

메소드의 종류

종류	선언	사용
정적 메소드	static 반환형 메소드명 (매개변수)	클래스명.메소드명(인자)
인스턴스 메소드	반환형 메소드명(매개변수)	참조변수.메소드명(인자)

```
class Test {
  static void method1() {   // 정적 메소드
    int s = 1;
  }
  void method2() {     // 인스턴스 메소드
    int i = 2;
  }
}
```

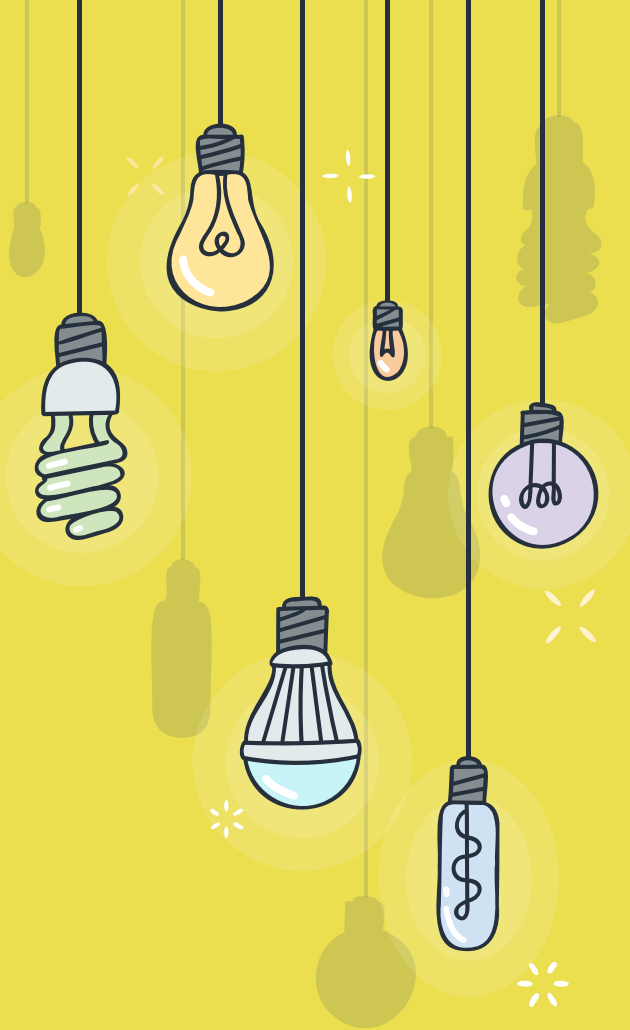
```
Test test = new Test();

Test.method1();  // 정적 메소드 호출

test.method2();  // 인스턴스 메소드 호출
```

정적 메소드 주의사항

- 정적 메소드 선언 시 주의 사항
  - 정적 메소드 선언 시 인스턴스 필드와 인스턴스 메소드 호출 불가

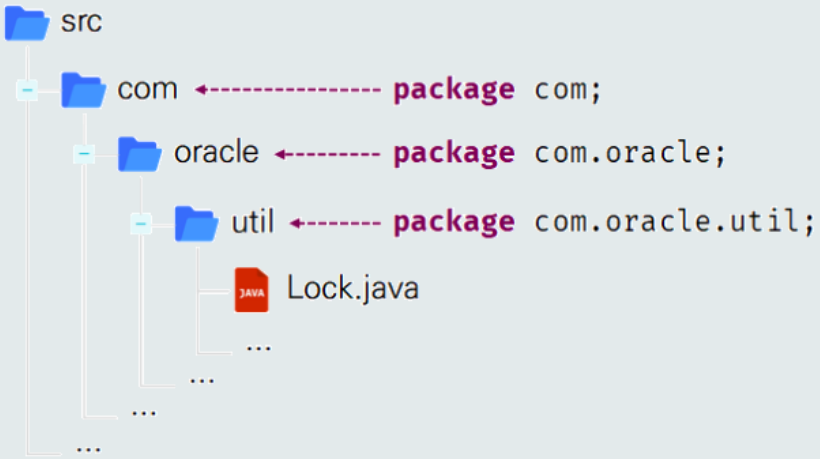
```
class Abc {
   int field1;
   static int field2;

   void method1() { … }
   static void method2() {
      field1 = 1;       // 사용불가
      method1();        // 사용불가
   }
}
```

7 패키지와 제어자



패키지

- 패키지
  - 서로 관련된 클래스들의 물리적인 폴더의 형태
  - 클래스 이름이 동일하더라도 패키지가 다르면 다른 클래스로 인식
  - 클래스의 전체 이름 : 상위패키지.하위패키지.클래스



패키지 선언

- 패키지 선언
  - 클래스 작성 시 해당 클래스 가 어떤 패키지에 속할 것인지 선언

```
package 상위패키지.하위패키지;

public class Test() {
        ...
}
```

- 패키지 이름 규칙
  - 숫자로 시작 불가
  - 특수기호는 _와 $만 사용 가능(권장하지 않음)
  - 소문자로 작성하는 것이 관례

IMPORT문

- import문
  - 다른 패키지의 클래스나 인터페이스 사용시 컴파일러에 통지
  - 패키지 선언과 클래스 선언 사이에 작성
  - import를 사용하지 않을 시 매번 클래스 전체이름을 사용해야 함

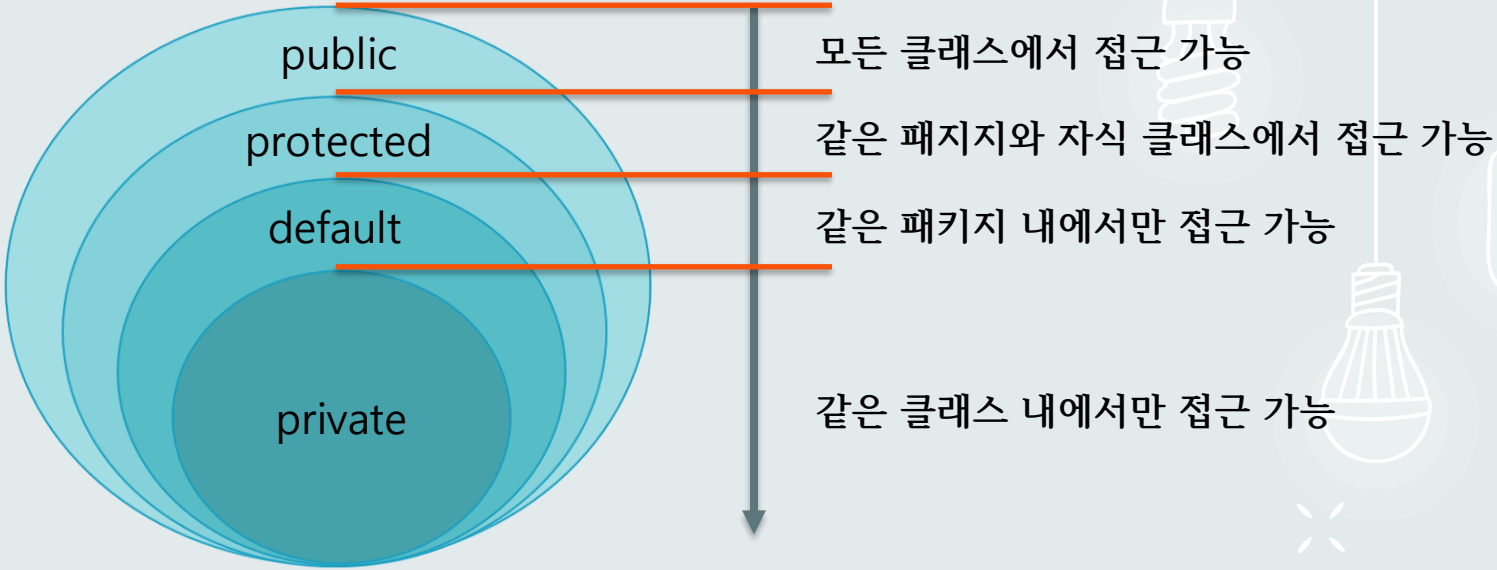
```
package 상위패키지.하위패키지;

import java.util.ArrayList;    // 패키지 안의 명시한 클래스만 사용
import java.io.*;              // java.io패키지 안의 모든 클래스 사용

public class Test() {
        ...
}
```

제어자(MODIFIER)

- 클래스, 메소드, 변수 등에 적용되어 해당 요소의 동작이나 접근 범위 등을 제어하는 키워드
- 종류
  - 접근 제어자(Access Modifiers)
    - 필드 또는 클래스에 사용
    - 외부로부터 접근 제한하여 데이터 보호
  - 그 외의 제어자(Non-Access modifiers)
    - 동작 특성, 상속 여부 등을 지정하거나 제한하는 키워드

접근 제어자(Access Modifier)

- 접근 제어자가 사용되는 곳
  - 클래스, 인스턴스 변수, 메서드, 생성자



GETTER/SETTER

- 일반적으로 인스턴스 필드의 접근 제한자를 private로 했을 때 사용
  - 보통 메소드 이름을 필드명과 동일하게
  - 첫글자만 대문자로 함
- Getter
  - 인스턴스 필드 값을 외부로 리턴하는 메소드

```
String getName() { return name };
```

- Setter
  - 외부에서 인자값을 받아 필드의 값을 변경하는 메소드

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

그 외의 제어자(Non-Access modifiers)

제어자	사용되는 곳	설명
static	인스턴스 필드, 메서드	정적멤버
final	클래스	상속 불가
	메서드	오버라이딩 불가
	멤버 필드, 지역 변수	상수
abstract	클래스, 메서드	추상화
synchronized	메서드, 코드 블록	스레드 동기화 (멀티스레딩 환경)
volatile	변수 값이 캐시 되지 않고 항상 메인 메모리에서 읽힘	
transient	직렬화에서 제외됨	
native	자바가 아닌 다른 언어(C/C++)로 구현된 메소드	
strictfp	IEEE 754 표준의 정확한 float/double 규칙에 따르도록 강제 계산 결과가 플랫폼에 관계없이 일관되게 보장	

제어자의 조합

1. 메서드에서 static과 abstract을 동시 사용 안됨
2. 클래서에서 abstract와 final을 동시 사용 안됨
3. abstract메서드의 접근 제어자가 private이면 안됨
4. 메서드에 private과 final을 같이 사용할 필요 없음

사용하는 곳	사용 가능한 제어자
클래스	public, default, final, abstract
메서드	접근 제어자, final, abstract, static
인스턴스 필드	접근 제어자, final, static
지역변수	final

Thanks!

- Any questions?