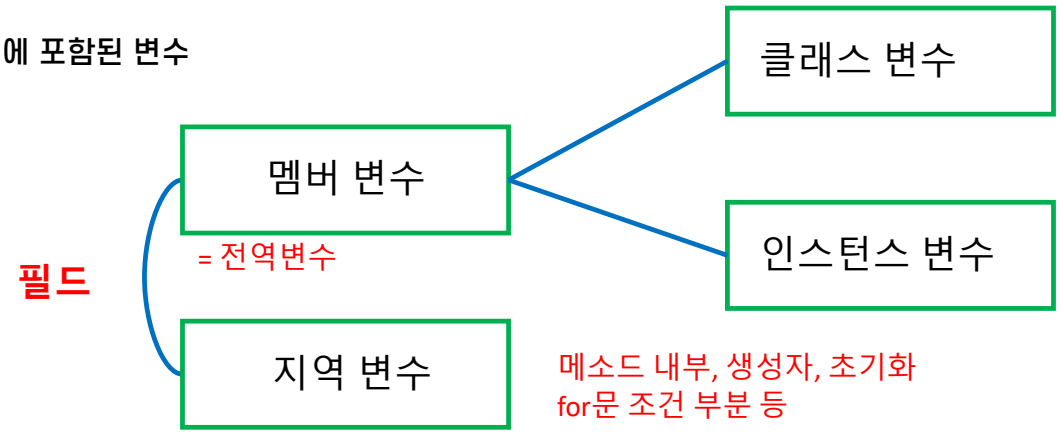
Java의 정석

Chapter 6.3~6.6

변수와 메소드

❑ **변수**

- **필드: 클래스에 포함된 변수**



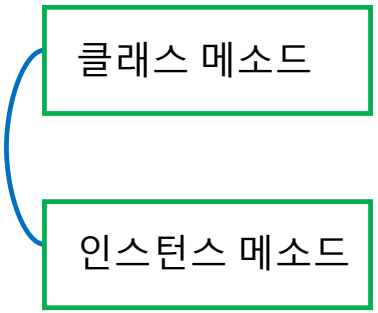
- Static을 앞에 붙인다
- 메모리를 따로 저장한다
- 동일한 클래스로 만든 객체가 2개 있다고 가정할 때, 하나의 객체에서 클래스 변수를 바꿀 경우 다른 객체에서도 값이 바뀐다
- 값을 바꾸지 않고 고정시키고 싶다면 앞에 final

```
class Car {
   String color;  // 필드 : 인스턴스 변수
   int speed;     // 필드 : 인스턴스 변수
}
```

변수와 메소드

❑ 메소드(=함수)

- 클래스의 '기능'에 해당하는 부분을 만든다
- 중복 최소화, 재사용, 구조화 목적 = 작업 단위를 잘개 쪼개어 여러번 사용 가능하도록 도와줌



- Static을 앞에 붙인다

- **Return 값 있을 때:** 메소드 선언 시 return 값 데이터형 작성
- **Return 값 없을 때:** void 작성
- 클래스 내부에서 메소드를 만들 때의 입장에서 들어오는 값들을 **매개변수(parameter)**라고 한다
- 우리가 만드는 객체 입장에서 메소드에 보내주는 값들을 **인자(argument)**라고 한다.
- **매개변수와 인자는 동일**한 것이지만 관점의 차이이다.

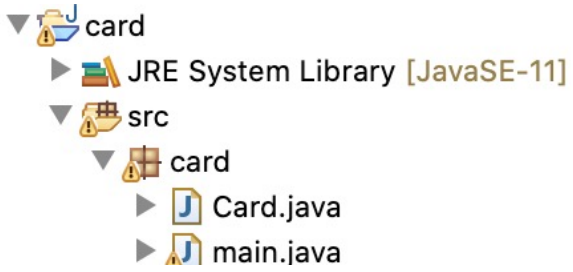
변수와 메소드

❑ Static (클래스 변수/메소드)에 대하여

- 모든 인스턴스가 공통된 저장공간(변수)을 공유하게 됨
- 인스턴스를 생성하지 않고 언제라도 바로 사용 가능

변수와 메소드

❑ Card 클래스




```java
package card;

public class Card {
    String kind;
    int number;
    final static int width = 100, height= 250;

    static void shuffle() {
        System.out.println("카드를 섞습니다");
    }

    Card draw() {
        System.out.println("카드 한개를 뽑습니다");
        Card c = new Card();
        c.kind = "Clover";

        // 0~9사이 랜덤한 숫자 출력
        c.number = (int) Math.round(Math.random()*10);
        return c;
    }

    int number() {
        return number;
    }
    String kind() {
        return kind;
    }
}
```

- Draw() 메소드는 참조형 반환타입이다. (p.268)
- 같은 클래스에 속한 멤버(변수/메소드) 간에는 별도의 인스턴스 생성 없이 서로 참조(사용)이 가능하다 (p.280)
- 단, 클래스멤버가 인스턴스 멤버를 참조/호출하기 위해서는 객체가 필요하다(12~20줄)
- “Shuffle() 메소드 안에 number() 메소드 쓸 수 없다”

변수와 메소드

❑ 실제 실행할 메인문

```
3  public class main {
4
5      public static void main(String[] args) {
6          // TODO Auto-generated method stub
7          Card card = new Card();
8
9  //      static 메소드이므로 클래스.shuffle() / 객체이름.shuffle() 다 가능함
10         Card.shuffle();
11         System.out.println(); // 띄어쓰기 목적
12
13
14         // 카드 한장 뽑기. return 값은 카드 객체
15         Card first_card = card.draw();
16         System.out.println();
17
18         System.out.println(first_card.number());
19         System.out.println(first_card.kind());
20         System.out.println();
21
22         // card 객체에 값 입력해주기
23         card.kind = "Spade";
24         card.number = 7;
25
26         // 인스턴스 메소드이므로 클래스.numer()와 같이 출력 불가
27         System.out.println(card.number());
28         System.out.println(card.kind());
29     }
30
31 }
```

JVM 메모리 구조

❑ **메소드 영역 / 스태틱 메모리 영역**

- 프로그램 실행 중 어떤 클래스가 사용되면 해당 클래스에 대한 정보가 이곳에 저장됨
- Class와 “static”이 들어간 모든 것들은 여기 속한다고 보면 된다
- 프로그램이 종료될 때 메모리에서 사라진다
- 가비지 콜렉터의 관리를 받지 않는다

❑ **힙(heap) 영역**

- 인스턴스(객체)가 생성되는 공간
- new로 만드는 애들, String으로 만든 애들은 힙 영역에 메모리가 들어간다고 보면 된다
- 가비지 콜렉터의 관리를 받는다

❑ **호출스택(call stack / execution stack) 영역**

- 메소드가 호출되었을 때, 메소드 내부 연산들을 시행하는 과정에서 필요한 메모리가 저장되는 공간
- 메소드 연산 중 나오는 지역변수, 매개변수, 중간 연산들 모두 여기에 해당
- 메소드 작업 완료 시 메모리가 비워진다

JVM 메모리 구조

- 파이썬과 달리 자바, C 등의 언어의 강점은 메모리 관리를 통한 빠른 속도이다.
- 그렇다면 앞의 메모리 개념을 생각해볼 때 어떤 점들을 신경써서 관리 해야할까?

① 메소드(스태틱 메모리) 영역. ② 힙 영역 ③ 호출스택 영역

JVM 메모리 구조

❑ 우리가 주목할 부분은 크게 2가지이다.

1) 스태틱 메모리 영역 중 "static"

- 우리는 코딩 과정에서 필요한 많은 클래스들을 만들고 가져다가 쓸 것이다. 하지만 이것은 필요한 부분이므로 메모리를 절감시키기 어렵다.
- 한편, static 변수 및 메소드의 경우 여러 객체를 만들 때 동일한 변수/메소드를 여러 번 만들며 메모리 낭비하는 행위를 막을 수 있다는 장점은 있다. 하지만, 한번 선언할 경우 프로그램이 종료될 때까지 메모리를 무조건적으로 차지한다는 단점이 있다.
- 따라서, 우리는 이 trade-off 관계에서 적당한 상황에서만 static을 쓰도록 고민해야한다.

2) 힙 영역

- 힙 영역은 가비지 콜렉터에 의해 관리 받는다. 하지만 가비지 콜렉터가 모든 불필요한 객체를 지워주는 것은 아니다.
- 따라서 지속적으로 객체를 만드는 행위는 하지 않도록 해야한다. (무한히 객체를 만들어내는 구조)
- 이러한 객체 또한 프로그램 종료까지 메모리를 먹기 때문이다.

가비지 컬렉션(Garbage Collection)

❑ 가비지 컬렉션

- 힙 영역의 객체 중 stack에서 도달 불가능한 객체들이 삭제되는 대상이 된다.
- https://yaboong.github.io/java/2018/06/09/java-garbage-collection/
- https://mangkyu.tistory.com/47

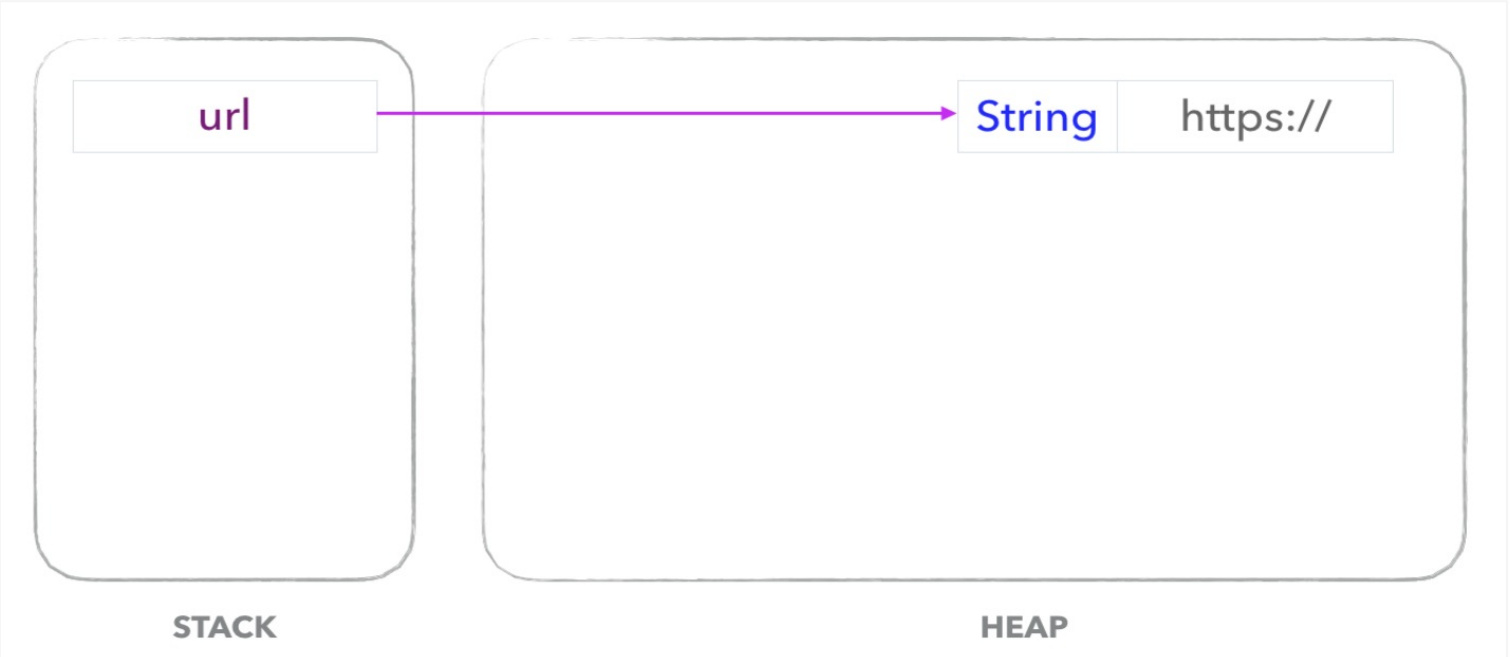
예시1) String 클래스의 경우

```
public class Main {
    public static void main(String[] args) {
        String url = "https://";
        url += "yaboong.github.io";
        System.out.println(url);
    }
}
```

가비지 컬렉션(Garbage Collection)

```
String url = "https://";
```

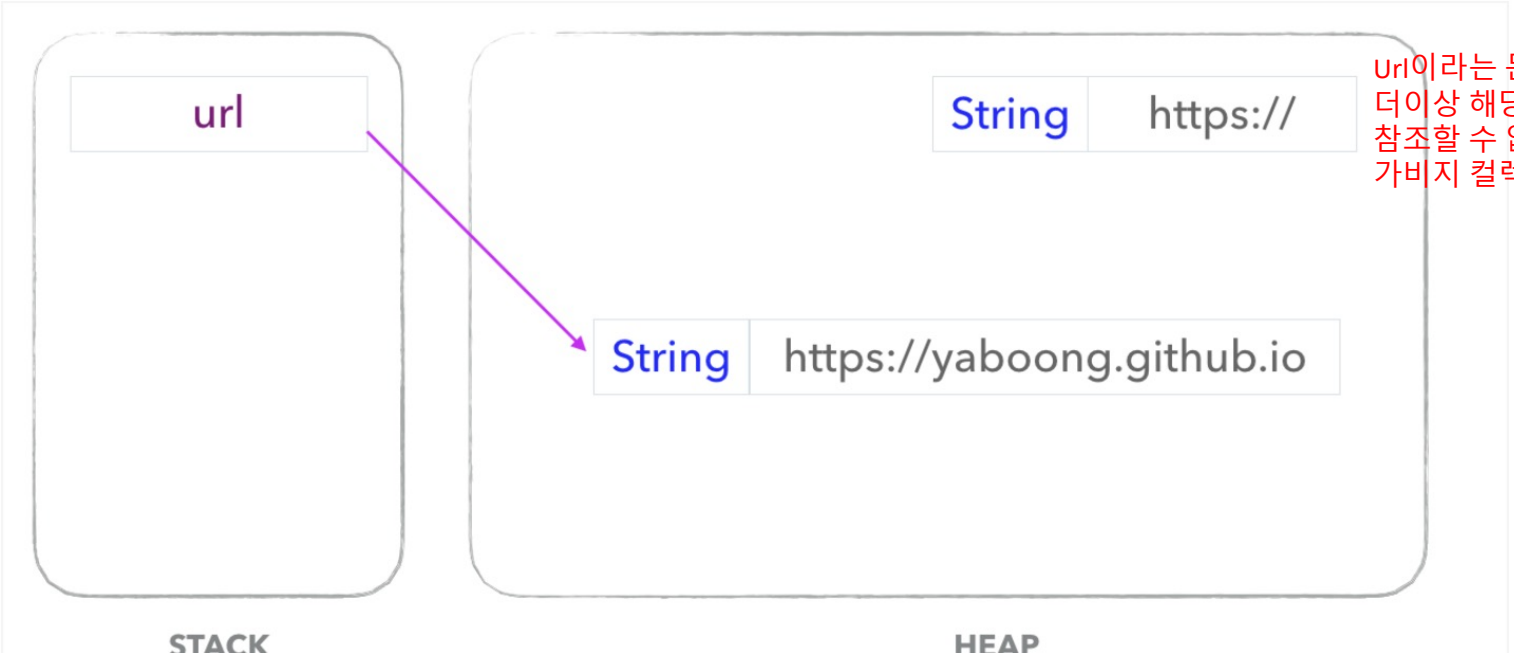
구문이 실행된 뒤 스택과 힙은 아래와 같다.



가비지 컬렉션(Garbage Collection)

다음 구문인

```
url += "yaboong.github.io";
```



Url이라는 문자열은
더이상 해당 문자열을
참조할 수 없으므로
가비지 컬렉터에 의해 삭제됨

가비지 컬렉션(Garbage Collection)

예시2) 다른 객체를 넣어줄 때

```
public class MadPlay {
    public static void main(String[] args) {
        MadMan madMan1 = new MadMan("Kim");
        MadMan madMan2 = new MadMan("Taeng");

        /* madMan2가 가리키던 객체는 가비지가 된다. */
        madMan2 = madMan1;
```

예시3) 안쓰는 객체를 null 처리할 때

```
public class MadPlay {
    public static void main(String[] args) {
        String testVar1 = new String("MadPlay");
        String testVar2 = new String("MadLife");
        String testVar3 = new String("Kimtaeng");
        String testVar4 = null;

        testVar1 = null;
        testVar4 = testVar3;
        testVar3 = null;
    }
}
```

결론적으로, 불가피하게 많은 객체를 계속만들어서 써야하는 구조라면 안쓰는 객체를 null 처리하도록 하는 것이 중요하다

재귀호출

- 고등학교 때 배운 factorial을 생각해보자.
- 똑같은 메소드를 계속 호출하며 연산하는 과정을 재귀호출이라고 한다.

```java
public class ex_0628 {

    public static void main(String[] args) {
        int result = factorial(4);
        System.out.println(result);
    }

    static int factorial(int n) {
        if (n==1) return 1;
        return n*factorial(n-1);
    }

}
```

오버로딩

❑ 같은 기능을 하는데 매개변수가 다를 경우(데이터타입, 개수 등) 메소드 이름을 달리 여러 개 만들지 않고 오버로딩 방식을 사용한다.

```
1  package javaStudy;
2
3  public class ex_0628 {
4
5      public static void main(String[] args) {
6          My_math m = new My_math();
7          System.out.println(m.add(2,5));
8          System.out.println(m.add(2.2, 5.3));
9
10
11     }
12 }
13
14 class My_math{
15     int add(int x, int y) {
16         return x+y;
17     }
18     double add(double x, double y) {
19         return x+y;
20     }
21 }
```

가변인자(varargs)

- 메소드에 들어갈 매개변수 개수를 자유롭게 하고 싶을 때 사용
- “(타입)... (변수명)”과 같이 작성
- 배열을 이용하는 것이다 -> 아래 코드의 for문 방식 잘 활용할 것(or lambda, stream)

```java
public class ex_0628 {

    public static void main(String[] args) {
        System.out.println(concatenate(" ", "I", "am", "a", "boy" ));

    }

    static String concatenate(String a, String... args) {
        String result="";

        for (String str: args) {
            result += str + a;
        }
        return result;
    }

}
```

```
Problems  Javadoc  Declaration  Console
<terminated> ex_0628 [Java Application] /Library/Java/JavaVirtualMachines/jdk-11.0.11.jdk/Contents/Home/bin
I am a boy
```

생성자 & 초기화

❑ 생성자: 생성자는 인스턴스가 생성될 때 호출되는 ' 인스턴스 초기화 메소드'이다.

- 쉽게 생각하면, 클래스에 해당하는 객체 처음 만들 때 초기값을 바로 입력 받을 수 있도록 도와준다.
- 오버로딩 기법을 많이 사용한다.
- this: 인스턴스 멤버에 사용한다
- this(): 클래스명 대신 사용한다

❑ 초기화: 아무것도 입력받지 않았을 때 가장 첫번째로 갖고 있을 값을 작성하는 것을 의미한다.

- 명시적: 변수 설정 시 바로 작성하는 것
- 초기화 블럭: 초기값 설정을 위해 연산이 필요하거나 복잡할 때 사용한다(for문 등 식 사용 가능)
  1) 클래스 초기화 블럭: 클래스가 메모리 처음 로딩될 때 한번만 수행
  2) 인스턴스 초기화 블럭: 인스턴스를 생성할 때마다 수행
- 초기화 순서: 명시적 -> 클래스 초기화 -> 인스턴스 초기화 -> 생성자