4차시 정리

```
// 실세계 객체마다 고유한 특성과 행동을 가지고, 서로 상호작용을 하면서 존재

// 객체 지향 특성 : 캡슐화
// 캡슐화 : 객체를 캡슐로 싸서 내부를 볼 수 없게 하는것, 외우의 접근으로 부터 객체를 보호하기 위해

// 클래스(class): 객체 모양을 선언한 틀(캡슐화)
// 메소드(멤버 함수)와 필드(멤버 변수)는 모두 클래스 내에 구현
// 객체를 만들어 내기 위한 설계도 혹은 틀

// 객체
// 클래스 모양대로 생성된 인스턴스(instance)
// 객체 내 데이터에 대한 보호, 외부 접근 제한
// 클래스의 모양 그대로 찍어낸 실체
// 메모리 공간을 갖는 구체적인 실체
// 인스턴스라고도 부름
// 객체들은 클래스에 선언된 동일한 속성을 가지지만 객체마다 서로 다른 고유한 값으로 구분됨

// 객체 지향의 특성 : 상속
// 상속 : 상위 개체의 속성이 하위 개체에 물려져서 하위 객체가 상위 개체의 속성을 모두 가지는 관계
// 예) 사람은 동물의 속성을 다 가지고 있지만 나무는 동물의 속성을 가지고 있지 않다.
// 생물 -> 동물 -> 사람, 생물 -> 식물 -> 나무
// 부모 클래스 : 수퍼클래스
// 하위 클래스 : 서브클래스, 수퍼 클래스를 재사용하고 새로운 특성 추가

// 객체 지향의 특성 : 다형성
// 다형성 : 같은 이름의 메소드가 클래스나 객체에 따라 다르게 동작하도록 구현, 메소드 오버로딩과 메소드 오버라이딩이 해당
// 메소드 오버로딩 : 같은 이름이지만 다르게 작동하는 여러 메소드
// 메소드 오버라이딩 : 슈퍼클래스의 메소드를 서브 클래스마다 다르게 구현, 부모클래스의 함수를 수정하여 사용하는 방법, 재정의

// 객체 지향 언어의 목적
// 소프트웨어의 생산성 향상 : 컴퓨터의 산업 발전에 따라 생명 주기 단축, 객체 지향 언어
// 소프트웨어를 빠른 속도로 생산할 필요성이 커지며,
// 재사용을 위한 여러 장치(상속, 다형성, 객체, 캡슐화)가 생기므로 다시 만드는 부담이 줄고 생산성 향상됨.
// 실세계에 대한 쉬운 모델링 : 컴퓨터 초기시대의 프로그래밍, 현대 프로그래밍, 객체 지향 언어
// 옛날에는 처리 과정, 계산 절차에 초점을 두었다면 현재는 객체등의 상호작용을 묘사하는 것이 더 중요해졌다.
// 실세계의 일을 보다 쉽게 프로그래밍하기 위한 객체 중심적 언어

// 절차 지향 프로그래밍
// 순서로 표현, 함수들의 집합으로 작성

// 객체 지향 프로그래밍
// 객체들간 상호작용으로 표현, 클래스 혹은 객체들의 집합으로 작성
```

```
// public 다른 클래스들에서 Circle 클래스를 사용하거나 접근할 수 있음을 선언
public class Circle1 {

	int radius; // 원의 반지름 멤버변수(필드)
	String name; // 원의 이름 멤버변수(필드)

	// 원의 생성자, 멤버 함수
	// 생성자 : 객체가 생성될때 초기화를 위해 실행되는 메소드
	// 기본 생성자 : 매개 변수 없고 아무 작업 없이 단순 리턴하는 생성자(디폴트 생성자라고 부르기도 함)
	// 클래스에 생성자가 하나라도 작성된 경우 기본생성자는 자동 삽입되지 않음.
	public Circle1() { }  // 생성자 메소드(멤버 함수)

	public double getArea() { // 원의 면적 계산 메소드(멤버 함수)
		return 3.14*radius*radius;
	}

	// static 때문에 따른 파일에서 불러올시 참조를 못함
	// public 다른 모든 클래스의 접근을 허용
	public static void main(String[] args) {

		// 반드시 new로 레퍼런스 Circle 타입의 객체(변수)선언
		// 객체를 선언할 때 반드시 new 키워드 이용
		// 객세 생성과정 : 객체에 대한 레퍼런스변수(주소변수) 선언, 메모리 할당, 객체내 생성자코드 실행
		Circle1 pizza;
		pizza = new Circle1();
		pizza.radius = 10;
		pizza.name = "자바피자";
		double area = pizza.getArea();
		System.out.println(pizza.name + "의 면적은 " + area);

		// 반드시 new로 레퍼런스 Circle 타입의 객체(변수)선언
		// 객체를 선언할 때 반드시 new 키워드 이용
		Circle1 donut = new Circle1(); // 반드시 new로 레퍼런스 Circle 타입의 객체(변수)선언
		donut.radius = 2;
		donut.name = "자바도넛";
		area = donut.getArea();
		System.out.println(donut.name + "의 면적은 " + area);
	}
}

// 출력
// 자바피자의 면적은 314.0
// 자바도넛의 면적은 12.56
```

```
import java.util.Scanner;

public class Rectangle {

	int width; // 멤버 변수(필드)
	int height; // 멤버 변수(필드)

	public int getArea() { // 멤버 함수(메소드)
		return width*height;
	}

	// static 때문에 따른 파일에서 불러올시 참조를 못함
	// public 다른 모든 클래스의 접근을 허용
	public static void main(String[] args) {

		Scanner scanner = new Scanner(System.in);
		Rectangle rect = new Rectangle(); // 반드시 new로 선언

		System.out.print("면적을 구하려는 사각형의 너비와 높이를 입력해주세요. >> ");

		rect.width = scanner.nextInt();
		rect.height = scanner.nextInt();

		System.out.println("사각형의 면적은 " + rect.getArea());

		// 사용자 입력 객체 종료
		scanner.close();
	}
}

// 출력
// 면면적을 구하려는 사각형의 너비와 높이를 입력해주세요. >> 4 5
// 사각형의 면적은 20
```

```
public class Circle2 {
	int radius;
	String name;

	// Circle2의 생성자, 멤버 함수, 메소드
	// 생성자 : 객체가 생성될 때 초기화를 위해서 자동으로 한 번 호출(실행)되는 메소드
	// 생성자는 new를 통해 객체를 생성되며, 생성될 때, 반드시 객체당 한 번 호출.
	// 리턴 타입을 따로 생성할 수 없다.
	// 기본 생성자 : 매개 변수 없고 아무 작업 없이 단순 리턴하는 생성자(디폴트 생성자라고 부르기도 함)
	// 클래스에 생성자가 하나도 선언되지 않은 경우, 컴파일러에 의해 자동으로 삽입
	// 클래스에 생성자가 하나라도 작성된 경우 기본생성자는 자동 삽입되지 않음.
	// 함수의 오버로딩 (메소드 인자의 개수나 자료형에 따라 생성자를 여러 개 선언하는 것)
	public Circle2() { // 매개 변수 없는 생성자, 기본생성자
		radius = 1; // radius의 초기값은 1
		name = "";
	}

	public Circle2(int r, String n) { // 메소드 인자가 2개가 있는 생성자, 함수의 오버로딩 실현
		radius = r;
		name = n;
	}

	public double getArea() {
		return 3.14 * radius * radius;
	}

	// static 때문에 따른 파일에서 불러올시 참조를 못함
	// public 다른 모든 클래스의 접근을 허용
	public static void main(String[] args) {

		Circle2 pizza = new Circle2(10, "자바피자"); // Circle 객체 생성, 반지름 10
		pizza.name = "시험싫다";
		double area = pizza.getArea();

		System.out.println(pizza.name + "의 면적은 " + area);

		Circle2 donut = new Circle2(); // Circle 객체 생성, 반지름 1
		donut.name = "도넛피자";
		area = donut.getArea();

		System.out.println(donut.name + "의 면적은 " + area);
	}
}

// 출력
// 시험싫다의 면적은 314.0
// 도넛피자의 면적은 3.14
```

```
// 생성자 : 객체가 생성될 때 초기화를 위해서 자동으로 한 번 호출(실행)되는 메소드
// 생성자는 new를 통해 객체를 생성되며, 생성될 때, 반드시 객체당 한 번 호출.
// 리턴 타입을 따로 생성할 수 없다.
// 기본 생성자 : 매개 변수 없고 아무 작업 없이 단순 리턴하는 생성자(디폴트 생성자라고 부르기도 함)
// 클래스에 생성자가 하나도 선언되지 않은 경우, 컴파일러에 의해 자동으로 삽입
// 클래스에 생성자가 하나라도 작성된 경우 기본생성자는 자동 삽입되지 않음.
// 함수의 오버로딩 (메소드 인자의 개수나 자료형에 따라 같은 이름의 생성자를 여러 개 선언하는 것)
// 생성자의 리턴 타입은 오버로딩과 관련이 없음

public class Book {
	String title; // 필드(멤버 변수)
	String author; // 필드(멤버 변수)

	// 기본 생성자는 없는 상태
	// 함수의 오버로딩 (메소드 인자의 개수나 자료형에 따라 다른게 선언하는 것)
	public Book(String t) { // 메소드 인자가 1개가 있는 생성자
		title = t;
		author = "작자미상";
	}

	public Book(String t, String a) { // 메소드 인자가 2개가 있는 생성자
		title = t;
		author = a;
	}

	public static void main(String[] args) {

		// 메소드 인자가 2개인 2번째 오버로딩 함수를 호출
		Book littlePrince = new Book("어린왕자", "생텍쥐페리");

		// 메소드 인자가 1개인 1번째 오버로딩 함수를 호출
		Book loveStory = new Book("춘향전");

		System.out.println(littlePrince.title + " " + littlePrince.author);
		System.out.println(loveStory.title + " " + loveStory.author);
	}
}

// 출력
// 어린왕자 생텍쥐페리
// 춘향전 작자미상
```


```
// 생성자 : 객체가 생성될 때 초기화를 위해서 자동으로 한 번 호출(실행)되는 메소드
// 생성자는 new를 통해 객체를 생성되며, 생성될 때, 반드시 객체당 한 번 호출.
// 리턴 타입을 따로 생성할 수 없다.
// 기본 생성자 : 매개 변수 없고 아무 작업 없이 단순 리턴하는 생성자(디폴트 생성자라고 부르기도 함)
// 클래스에 생성자가 하나도 선언되지 않은 경우, 컴파일러에 의해 자동으로 삽입
// 클래스에 생성자가 하나라도 작성된 경우 기본생성자는 자동 삽입되지 않음.
// 함수의 오버로딩 (메소드 인자의 개수나 자료형에 따라 생성자를 여러 개 선언하는 것)
```

```
public class Book2 {

	String title;
	String author;

	void show() {
		System.out.println(title + " " + author);
	}

	// 함수의 오버로딩 (메소드 인자의 개수나 자료형에 따라 다른게 선언하는 것)
	public Book2() { // 메소드 인자가 없는 생성자
		// this : 객체 자신에 대한 래퍼런스, this.멤버변수 는 클래스이름.멤버변수로 생각할 수 있다.
		// this가 필요한 경우
		// 멤버변수와 메소드변수의 이름이 같은 경우나, 객체 자신의 레퍼런스를 전달하여 다른 메소드를 호출하거나,
		// 메소드가 객체 자신의 레퍼런스(주소)를 반환할 때 this를 사용
		// this.변수1 -> class내 지역변수 (변수1)를 호출.
		// this(인자들) -> class내 인자들의 숫자와 같은 생성자를 호출
		// 반드시 생성자 코드의 제일 처음에 수행, 생성자 내에서만 사용 가능
		// this로 메소드 인자가 2개인 2번째 오버로딩 함수를 호출
		this("", "");
		System.out.println("기본생성자 호출됨");
		// this("", ""); # 이 문장은 this함수가 생성자 코드 제일 첫 번째에 작성되지 않아서 오류이다.
	}
	public Book2(String title) { // 메소드 인자가 2개가 있는 생성자
		// this로 메소드 인자가 2개인 2번째 오버로딩 함수를 호출
		this(title, "작자미상");
		System.out.println("생성자1 호출됨");
	}
	public Book2(String title, String author) { // 메소드 인자가 2개가 있는 생성자
		this.title = title; this.author = author;
		System.out.println("생성자2 호출됨");
	}
	public static void main(String[] args) {
		// 메소드 인자가 2개인 2번째 오버로딩 함수를 호출
		Book2 littlePrince = new Book2("어린왕자", "생텍쥐페리");

		// 메소드 인자가 1개인 1번째 오버로딩 함수를 호출
		// 함수의 순서에 따라 다음으로, 메소드 인자가 2개인 2번째 오버로딩 함수를 호출
		Book2 loveStory = new Book2("춘향전");

		// 메소드 인자가 없는 기본 생성자를 호풀
		// 함수의 순서에 따라 다음으로, 시스템 출력 후 메소드 인자가 2개인 2번째 오버로딩 함수를 호출
		Book2 emptyBook = new Book2();

		// 출력
		loveStory.show();
	}
}
// 출력
// 생성자2 호출됨
// 생성자2 호출됨
// 생성자1 호출됨
// 생성자2 호출됨
// 기본생성자 호출됨
// 춘향전 작자미상
```

```
// 객체의 치환은 개체가 복사되는 것이 아니며 레퍼런(주소)가 복사된다.

class Circle {
	int radius;
	// public : 패키지 관계 없이 모든 클래스에게 접근 허용
	// 동일 클래스 내에만 접근 허용, 상속 받은 서브 클래스에서 접근 불가
	public Circle(int radius) {
		// this : class 객체 자신에 대한 래처런스
		// 반드시 생성자 코드의 제일 처음에 수행, 생성자 내에서만 사용 가능
		this.radius = radius;
	}

	public double getArea() {
		return 3.14 * radius * radius;
	}
}

public class CircleArray {
	public static void main(String[] args) {
		// Circle[] c = new Circle[5]; 로도 선언가능, 컴퓨터에 생성과 레퍼런스 매칭이 동시에 진행된다.
		Circle[] c; // 레퍼런스 변수 c 생성. Circle 객체인 c가 여러개 만들어진 준비를 한다.
		c = new Circle[5]; // 레퍼런스 배열생성. Circle 객체인 c를 5개 만들고 각 객체에 레퍼런스를 매칭한다.

		for (int i = 0; i < c.length; i++)
			c[i] = new Circle(i); // 각 객체에 생성자를 호출하기 위해서 new를 사용해야한다.

		for (int i = 0; i < c.length; i++)
			System.out.print((int) (c[i].getArea()) + " "); // 실수부분을 잘라내서 정보 누락 생김
	}
}

// 출력
// 0 3 12 28 50
```

```
import java.util.Scanner;
class Book {
	String title, author;

	public Book(String title, String author) {
		// this : class 객체 자신에 대한 래처런스
		// 반드시 생성자 코드의 제일 처음에 수행, 생성자 내에서만 사용 가능
		this.title = title;
		this.author = author;
	}
}

public class BookArray2 {
	// 접근 지정자 : public
	// 리턴 타입 : void
	// 메소드 이름 : main
	// 메소드 인자 : String[] args
	// 그리고 모든 객체에 대해 공유가 된다.
	public static void main(String[] args) {

		// Book 객체인 book가 여러 개 만들어진 준비를 한다.
		// new를 작성하면서 Book 객체인 book를 2개 만들고 각 객체에 레퍼런스를 매칭한다.
		Book[] book = new Book[2]; // Book 배열 선언 및 레퍼런스 배열 생성
		// Book[] book; : 레퍼런스 변수 book생성
		// book = new Book[2] : 레퍼런스 배열 생서

		// 사용자 입력 공간 선언
		Scanner scanner = new Scanner(System.in);

		for (int i = 0; i < book.length; i++) {

			System.out.print( (i+1) + "번 책 제목>>");
			String title = scanner.nextLine();

			System.out.print( (i+1) + "번 책 저자>>");
			String author = scanner.nextLine();

			// 배열의 각 객체에 생성자로 값을 대입할때 new를 사용하여 배열 원소 객체를 생성한다.
			book[i] = new Book(title, author); // 배열의 각 원소 객체 생성
		}

		for (int i = 0; i < book.length; i++) // 배열의 각 원소 객체 사용
			System.out.print( (i+1) + "번책 : (" + book[i].title + ", " + book[i].author + ")\n");

		scanner.close();
	}
}

// 출력
// 1번 책 제목>> 홍길동전
// 1번 책 저자>> 허균
// 2번 책 제목>> 노인과 바다
// 2번 책 저자>> 누구더라...
// 1번책 : ( 홍길동전,  허균)
// 2번책 : ( 노인과 바다,  누구더라...)
```

```
// 메소드 형식
// 메소드는 클래스 멤버의 함수이고, c의 함수와 동일하다.
// 자바의 모든 메소드는 반드시 클래스 안에 있어야 한다.(캡슐화 원칙)

// 메소드의 구성 형식
// 접근지정자 리턴타입 메소드이름 (메소드인자들){...}

// 접근지정자 : public, protected, 디폴트, private
// 리턴타입 : 메소드가 반환하는 값의 데이터 타입

// 자바에서의 인자 전달 방법
// 기본 타입의 경우 값은 값을 복사
// 그 외 객체, 배열들은 레퍼런스(주소) 전달, 메소드의 매개변수와 호출한 실인자 객체나 배열 공유
// 배열이 전달 되는 경우 배열 전체가 전달 되는 것이 아니라 레퍼런스(주소)만 전달되서 공유가 되는 것이다.

public class ArrayParameterEX {
	// 접근 지정자 : 멤버를 보호하는 캡슐화의 정체에 묶인 보호를 일부 해제할 목적으로 생성
	// 접근 지정자의 종류 : public, protected, 디폴트, private
	// 패키지 : 관련 있는 클래스 파일( .class)을 저장하는 디렉터리
	// public : 모든 패키지의 모든 클래스의 접근을 허용한다.
	// protected : 같은 패키지의 자식 클래스에 허용.
	// 디폴트(생략, package-private라고도 불림) : 같은 패키지의 클래스에만 허용.
	// private : 같은 클래스 내에서만 가능(패키지 이동 불가) 외부로부터 완벽차단

	// non-static 멤버(인스턴스 멤버)
	// 공간적 특성 : 멤버들은 객체마다 독립적으로 별도 존재
	// 시간적 특성 : 필드와 메소드는 객체 생성 후 비로소 사용가능
	// 비공유 특성 : 멤버들은 다른 객체에 의해 공유되지 않고 배타적
	// static 멤버(클래스 멤버): 클래스당 하나만 생성되면서 객체를 생성하지 않고 사용할 수 있다.
	// 공간적 특성 : static 멤버들은 클래스 당 하나만 객체 내부가 아닌 클래스 코드가 적재되는 곳에 별도 생성
	// 시간적 특성 : static 멤버들은 클래스가 로딩될 때 공간 할당. 즉 객체 생성 전에 이미 생성되고 사용 가능.
	//              : 객체가 사라져도 멤버는 사라지지 않고, 프로그램이 종료될 때 사라짐.
	// 공유 특성 : static 멤버들은 동일한 클래스의 모든 객체에 의해 공유
	// static는 전역 변수와 전역 함수, 공유 멤버를 작성할 때 활용.

	static void replaceSpace(char a[]) {
		for (int i = 0; i < a.length; i++)
			if (a[i] == ' ')
				a[i] = ','; // 공간이 하나밖에 없으므로 문자 하나만 대입가능
	}
	static void printCharArray(char a[]) {
		for (int i = 0; i < a.length; i++)
			System.out.print(a[i]);
		System.out.println(); // 줄나눔을 하기 위해 선언
	}
	public static void main(String[] args) {
		// 단순 객체(정수, 실수, 문자 등)는 값으로 복사, 그외 배열, 리스트, 튜플 등은 주소로 매개변수로 전달한다.
		char c[] = { 'T', 'h', 'i', 's', ' ', 'i', 's', ' ', 'a', ' ', 'p', 'e', 'n', 'c', 'i', 'l', '.' };
		printCharArray(c);
		replaceSpace(c);
		printCharArray(c);
	}
}
// 출력
// This is a pencil.
// This,is,a,pencil.
```

```
// 객체의 소멸과 가비지 컬렉션

// 객체 소멸 : new에 의해 할당된 객체 메모리를 자바 가상 기계의 가용메모리로 되돌려 주는 행위
// 자바 응용프로그램에서 임의로 객체 소멸을 할 수 없음. 객체 소멸은 자바 가상 세계의 고유한 역활이며, 개발자 입장에선 다행임.

// 가비지 : 가치키는 레퍼런스가 하나도 없는 객체
// 가비지 컬렉션
// 자바 가상 기계의 가비지 컬렉터가 자동으로 가비지 수집 반환, 자바에서 가비지를 자동 회수하는 과정
// 사용 메모리로 반환이 되며, 가비지 컬렉션 스레드에 의해 수행

// 개발자에 의한 강제 가비지 컬렉션
// System 또는 Runtime 객체의 gc() 메소드 호출,
// 예) System.gc(); -> 강력한 가비지 컬렉션 요청

// 접근지정자 이해
// 패키지 : 관련있는 클래스 파일(.class)을 저장하는 디렉터리, 자바 응용프로그램은 하나 이상의 패키지로 구성
// 접근 지정자 : 멤버를 보호하는 캡슐화의 정체에 묶인 보호를 일부 해제할 목적으로 생성
// 접근 지정자의 종류 : public, protected, 디폴트, private
// public : 모든 패키지의 모든 클래스의 접근을 허용한다.
// protected : 같은 패키지 또는 다른 패키지여도 자식 클래스에도 접근 허용.
// 디폴트(생략, package-private라고도 불림) : 같은 패키지의 클래스에만 허용.
// private : 같은 클래스 내에서만 가능(같은 패키지내 클래스 이동 불가) 외부로부터 완벽차단

public class GarbageEx {
	public static void main(String[] args) {

		String a = new String("Good");
		String b = new String("Bad");
		String c = new String("Normal");
		String d, e;

		// 가비지 : 주소(레퍼런스)는 생성되었는데 가리키는 변수가 없는 상황.
		a = null; // 가비지 발생, 매칭되고 있는 a의 주소는 그대로 있지만 가리키는 화살표가 사라지므로.
		d = c;
		c = null; // 가비지가 생성 안됨. 이전에 d가 주소를 가리키기 때문에.

		// 가비지 컬렉션 : 가비지가 있는 경우 자동으로 가비지를 자동 회수하는 과정
		// System 객체의 gc() 메소드를 호출하여 강제로 사용
		System.gc(); // 가비지 컬렉션 작동 요청
	}
}

// 출력
// (없음)
```

// 접근지정자 이해
// 패키지 : 관련있는 클래스 파일(.class)을 저장하는 디렉터리, 자바 응용프로그램은 하나 이상의 패키지로 구성
// 접근 지정자 : 멤버를 보호하는 캡슐화의 정체에 묶인 보호를 일부 해제할 목적으로 생성
// 접근 지정자의 종류 : public, protected, 디폴트, private
// public : 모든 패키지의 모든 클래스의 접근을 허용한다.
// protected : 같은 패키지 또는 다른 패키지여도 자식 클래스에도 접근 허용.
// 디폴트(생략, package-private라고도 불림) : 같은 패키지의 클래스에만 허용.
// private : 같은 클래스 내에서만 가능(같은 패키지내 클래스 이동 불가) 외부로부터 완벽차단

멤버에 접근하는 클래스	멤버의 접근 지정자			
	private	디폴트	protected	public
같은 패키지의 클래스	X	O	O	O
다른 패키지의 클래스	X	X	X	O
접근 가능 영역	클래스 내	동일 패키지 내	동일 패키지와 자식 클래스	모든 클래스

```
class Sample {
	public int a;
	private int b;
	int c;

	public void Sample_print() {
		System.out.println("a :" + a +", b :"+b+", c :"+c);
	}
}

public class AccessEx {
	public static void main(String[] args) {

		Sample aClass = new Sample();

		// Sample 클래스의 a와 c는 각각 public, default 지정자로 선언이 되었으므로,
		// 같은 패키지에 속한 AccessEx 클래스에서접근 가능
		// b는 private으로 선언이 되었으므로 AccessEx 클래스에서 접근 불가능
		aClass.a = 10;
		// aClass.b = 10; // 접근 불가
		aClass.c = 10;

		aClass.Sample_print();
	}
}

// 출력 -> ,가 아닌 +로 문자열을 이어줘야함
// a :10, b :0, c :10
```

// non-static 멤버(인스턴스 멤버)

// 공간적 특성 : 멤버들은 객체마다 독립적으로 별도 존재
// 시간적 특성 : 필드와 메소드는 객체 생성 후 비로소 사용가능
// 비공유 특성 : 멤버들은 다른 객체에 의해 공유되지 않고 배타적

// static 멤버(클래스 멤버): 클래스당 하나만 생성되면서 객체를 생성하지 않고 사용할 수 있다.
// 공간적 특성 : static 멤버들은 클래스 당 하나만 객체 내부가 아닌 클래스 코드가 적재되는 곳에 별도 생성
// 시간적 특성 : static 멤버들은 클래스가 로딩될 때 공간 할당. 즉 객체 생성 전에 이미 생성되고 사용 가능.
// : 객체가 사라져도 멤버는 사라지지 않고, 프로그램이 종료될 때 사라짐.
// 비공유 특성 : static 멤버들은 동일한 클래스의 모든 객체에 의해 공유

	non-static 멤버	static 멤버
선언		
공간적 특성	멤버는 객체마다 별도 존재 - 인스턴스 멤버라고 부름	멤버는 클래스강 하나 생성 - 멤버는 객체 내부가 아닌 별도의 공간(클래스 코드가 적제되는 메모리)에 생성 - 클래스 멤버라고 부름
시간적 특성	객체 생성 시에 멤버 생성됨 - 객체가 생길 때 멤버도 생성 - 객체 생성 후 멤버 사용가능 - 객체가 사라지면 멤버도 사라짐	클래스 로딩시에 멤버 생성 - 객체가 생기기 전에 이미 생성 - 객체가 생기기 전에도 사용가능 - 객체가 사라져도 멤버는 사라지지 않음 - 멤버는 프로그램이 종료될 때 사라짐
공유의 특성	공유되지 않음 - 멤버는 객체 내에서 각각 공간유지	동일한 클래스의 모든 객체들에 의해 공유됨

// static는 전역 변수와 전역 함수, 공유 멤버를 작성할 때 활용

```
class StaticSample{
	public int n;
	public void g() {
		m = 20;
	}
	public void h() {
		m = 30;
	}
	public static int m;
	public static void f() {
		m = 5;
	}
}

public class Ex {
	public static void main(String[] args) {
		StaticSample.m = 8; // static 필드를 클래스 이름으로 접근하는 방법
		System.out.println("StaticSample.m : " + StaticSample.m );

		StaticSample s1, s2;
		s1 = new StaticSample();
		s1.n = 5;
		s1.m = 50; // static 필드를 객체의 멤버로 접근하는 방법
		System.out.println("s1.m : " + s1.m );
		// System.out.println("s2.m : " + s2.m ); -> 오류, 아직 s2가 선언되지 않아서
		s1.g(); // static 메서드를 객체의 멤버로 접근하는 방법

		s2 = new StaticSample();
		System.out.println("s2.m : " + s2.m );
		s2.n = 8;
		s2.h(); // static 메서드를 객체의 멤버로 접근하는 방법
		System.out.println("s1.m : " + s1.m );
		s2.g(); // static 메서드를 객체의 멤버로 접근하는 방법
		System.out.println("s1.m : " + s1.m );

		StaticSample.f(); // static 메서드를 클래스 이름으로 접근하는 방법
		System.out.println("StaticSample.m : " + StaticSample.m );
	}
}

// 출력
// StaticSample.m : 8
// s1.m : 50
// s2.m : 20
// s1.m : 30
// s1.m : 20
// StaticSample.m : 5
```

```
// non-static 멤버(인스턴스 멤버)
// 공간적 특성 : 멤버들은 객체마다 독립적으로 별도 존재
// 시간적 특성 : 필드와 메소드는 객체 생성 후 비로소 사용가능
// 비공유 특성 : 멤버들은 다른 객체에 의해 공유되지 않고 배타적

// static 멤버(클래스 멤버): 클래스당 하나만 생성되면서 객체를 생성하지 않고 사용할 수 있다.
// 공간적 특성 : static 멤버들은 클래스 당 하나만 객체 내부가 아닌 클래스 코드가 적재되는 곳에 별도 생성
// 시간적 특성 : static 멤버들은 클래스가 로딩될 때 공간 할당. 즉 객체 생성 전에 이미 생성되고 사용 가능.
//           : 객체가 사라져도 멤버는 사라지지 않고, 프로그램이 종료될 때 사라짐.
// 비공유 특성 : static 멤버들은 동일한 클래스의 모든 객체에 의해 공유

// static는 전역 변수와 전역 함수, 공유 멤버를 작성할 때 활용

// static 멤버를 가진 클래스 사례 : Math 클래스(java.lang.Math)
// 모든 필드와 메서드가 public static으로 선언. 즉 다른 모든 클래스에서 사용할 수 있음.
// 잘못된 사용법
// Math m = new Math(); -> Math() 생성자는 private
// int n = m.abs(-5);
// 바른 사용법
// int n = Math.abs(-5)

class Calc {

	// static 멤버(클래스 멤버): 클래스당 하나만 생성되면서 객체를 생성하지 않고 사용할 수 있다.
	public static int abs(int a) {
		return a > 0 ? a : -a;
	}

	public static int max(int a, int b) {
		return (a > b) ? a : b;
	}

	public static int min(int a, int b) {
		return (a > b) ? b : a;
	}
}

public class CalcEx {
	public static void main(String[] args) {
		// 각 함수는 Calc의 static 객체이므로 따로 new 호출 필요 없이 사용 가능하다.
		// static는 non-static과 this에서는 사용 불가능하다.
		System.out.println(Calc.abs(-5));
		System.out.println(Calc.max(10, 8));
		System.out.println(Calc.min(-3, -8));
	}
}

// 출력
// 5
// 10
// -8
```

```
// static 제약조건 1
// static 메소드는 non-static 멤버 접근을 할 수 없음.
// 객체가 생성되기 전에도 static는 사용가능하므로 non-static 멤버 접근 불가
// 반대로 non-static 메소드는 static 멤버 사용가능

// static 제약조건 2
// static 메소드는 this 사용불가
// 객체가 생성되기 전에도 static는 사용가능하므로 현재 객체를 가리키는 this 레버런스 사용 불가

public class StaticAndNonAndThis {
	int n;
	static int m;

	void f1(int x) { n = x; } // 정상, non-static 메소드는 non-static 필드 사용가능 (n)
	void f2(int x) { m = x; } // 정상, non-static 메소드는 static 필드 사용가능 (n)
	//static void s1(int x) { n = x; } // 컴파일 오류, static 메소드는 non-static 필드 사용불가 (n)
	//static void s2(int x) { f1(3); } // 컴파일 오류, static 메소드는 non-static 메소드 사용불가 (f1)

	void g1(int x) { this.n = x; } // 정상, non-static 메소드에서는 non-static this 레퍼런스 사용 가능
	void g2(int x) { this.m = x; } // 정상, non-static 메소드에서는 static this 레퍼런스 사용 가능
	//static void s1(int x ) {this.n = x;} // 컴파일 오류. static 메소드는 non-static this 레퍼런스 사용 불가
	//static void s2(int x ) {this.m = x;} // 컴파일 오류. static 메소드는 static this 레퍼런스 사용 불가

	public static void main(String[] args) {}
}
// 출력
// (없음))
```

```
// static를 이용한 환율 계산기

// final 클래스와 메소드
// final 클래스는 상속이 불가. 사용 예) final class FinalClass{... }
// final 메소드는 오버라이딩(재정의)가 불가. 사용 예) public final int finalMethid( ... )

// final 필드 , 상수 선언
// 상수를 선언할때 사용을 하며. 사용 예) public static final double PI = 3.14
// 상수  필드는 선언시에 초기 값을 지정해야 한다. 재정의가 불가능 하므로
// 상수 필드는 실행중에 값을 변경할 수 없다. 또한 재정의도 불가능 하다.

package staticMenber;

import java.util.Scanner;

class CurrencyConverter {
	private static double rate; // 한국 원화에 대한 환율

	public static double toDollar(double won) {
		return won / rate; // 한국 원화를 달러로 변환
	}

	public static double toKWR(double dollar) {
		return dollar * rate; // 달러를 한국 원화로 변환
	}

	public static void setRate(double r) {
		rate = r; // 환율 설정. KWR/$1
	}
}

public class staticMenber {
	public static void main(String[] args) {

		Scanner scanner = new Scanner(System.in);

		System.out.print("환율(1달러)>> ");
		double rate = scanner.nextDouble();

		CurrencyConverter.setRate(rate); // 미국 달러 환율 설정

		System.out.println("백만원은 $" + CurrencyConverter.toDollar(1000000) + "입니다.");
		System.out.println("$100는 " + CurrencyConverter.toKWR(100) + "원입니다.");
		scanner.close();
	}
}

// 출력
// 환율(1달러)>> 1455
// 백만원은 $687.2852233676975입니다.
// $100는 145500.0원입니다.
```