

### 3. 자바 프로그래밍 리뷰



## 클래스

- 클래스(class): 객체를 만드는 설계도(추후에 학습)
- 자바 프로그램은 클래스들로 구성된다.

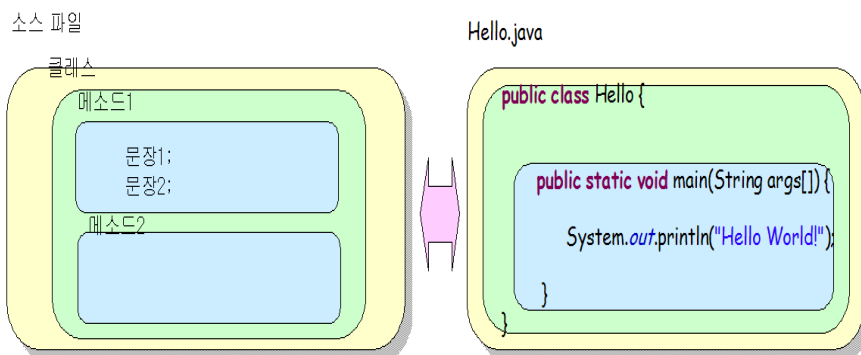
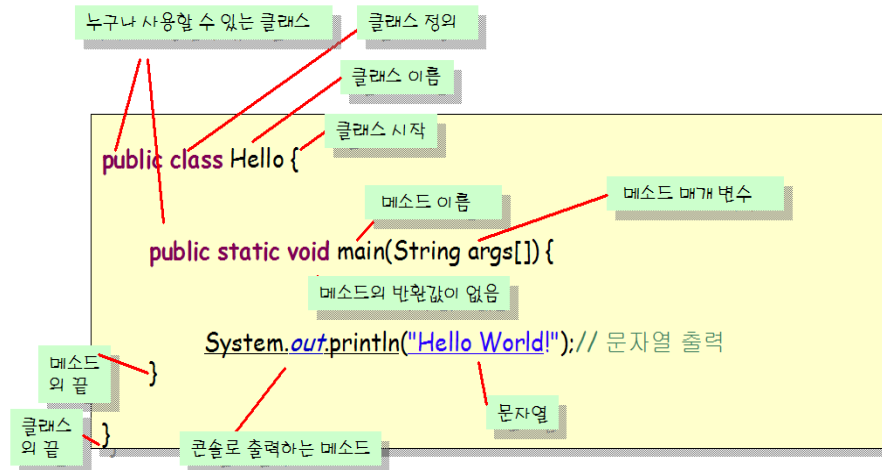


그림 4.2 자바 프로그램의 구조

## 용어 설명



## 문장은 순차적으로 실행

Welcome.java

// 문장의 순차적인 실행 예제

```
public class Welcome {  
    public static void main(String args[]) {  
        System.out.println("Welcome to");  
        System.out.println("Java");  
    }  
}
```

문장들을 순차적으로 실행된다.

실행결과

```
Welcome to  
Java
```

## Add 예제


```
Hello.java
/**
 * 두수의 합을 계산하는 애플리케이션
 */
public class Add {
    public static void main(String args[]) {
        int x; // 첫번째 정수를 저장할 변수
        int y; // 두번째 정수를 저장할 변수
        int sum; // 두 정수의 합을 저장하는 변수

        x = 100;
        y = 200;
        sum = x + y;

        System.out.println(sum);
        return;
    }
}
```

실행결과

```
300
```



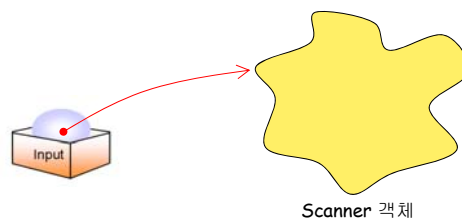
## import 문장

- `import java.util.Scanner;` // Scanner 클래스 포함
- Scanner 클래스를 포함시키는 문장
- Scanner는 자바 클래스 라이브러리(Java Class Library)의 일종
- Scanner는 입력을 받을 때 사용

## 객체 생성

```
Scanner input = new Scanner(System.in);
```

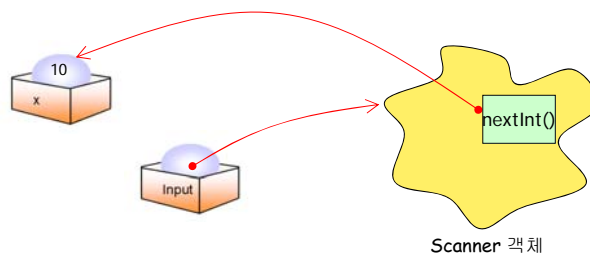
- `input`은 타입이 `Scanner`인 변수
- `new Scanner(System.in)`은 `Scanner` 클래스의 객체(object)를 생성
- `input`은 생성된 객체를 가리킨다.
- 상세한 설명은 차후에....
- 일단 입력을 받으려면 이 문장이 필요하다고 알아두자.



## 사용자로부터 입력

```
x = input.nextInt(); // 사용자로부터 첫 번째 정수를 읽는다.
```

- `Scanner` 객체인 `input`을 이용하여 사용자로부터 정수를 읽어 들이는 문장
- `input`을 통하여 `nextInt()`라고 하는 메소드를 호출하게 된다.
- `nextInt()`에서 반환된 값은 변수 `x`에 대입된다.
- 추후에 자세히....



### Salary.java

```
// 저축액을 계산하는 프로그램
import java.util.Scanner; // 입력 보조 클래스

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

        int salary; // 월급
        int deposit; // 저축액
        Scanner input = new Scanner(System.in);

        System.out.print("월급을 입력하시오: "); // 입력 안내 출력
        salary = input.nextInt();

        deposit = 10 * 12 * salary;
        System.out.printf("10년 동안의 저축액: %d\n", deposit);
    } // end method main
} // end class Salary
```

## 자료형

- 자료형(data type)은 자료의 타입
- **기초형**과 **참조형**으로 나눈다.

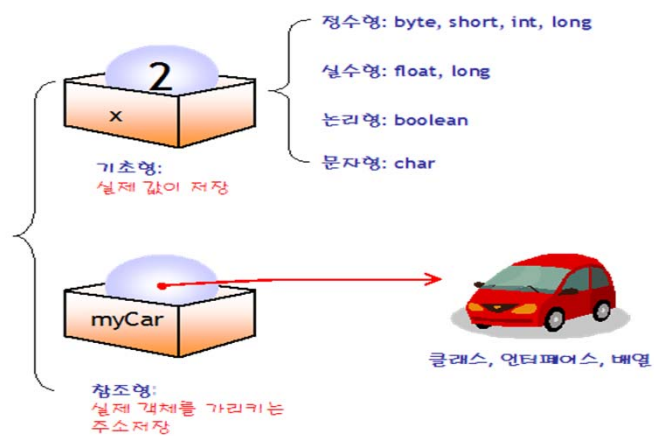
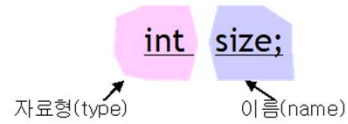


그림 5.4 기초형과 참조형

## 변수의 선언과 초기화

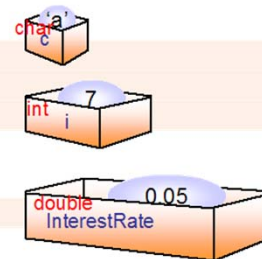


```
char c;  
int i;  
double interestRate;
```

```
char c = 'a';  
int i = 7;  
double interestRate = 0.05;
```

하나의 문장에서 변수를 여러 개 선언할 수도 있다.

```
int index, total = 0;
```



## 예제

*IsHangulOK.java*

```
public class IsHangulOK {  
    public static void main(String args[]) {  
        int 인덱스 = 0;  
        인덱스++;  
        System.out.println(인덱스);  
    }  
}
```

한글 변수  
이름도  
가능합니  
다.

실행결과

1

## 기초형

데이터형	설 명	크기 (비트)	기본값	최소값	최대값
byte	부호있는 정수	8비트	0	-128	127
short	부호있는 정수	16 비트	0	-32768	32767
int	부호있는 정수	32비트	0	-2147483648	2147483647
long	부호있는 정수	64비트	0L	-9223372036854775808	9223372036854775807
float	실수	32	0.0f	약 $\pm 3.4 \times 10^{-38}$ (유효숫자 7개)	약 $\pm 3.4 \times 10^{+38}$ (유효숫자 7개)
double	실수	64	0.0d	약 $\pm 1.7 \times 10^{-308}$ (유효숫자 15개)	약 $\pm 1.7 \times 10^{+308}$ (유효숫자 15개)
char	문자(유니코드)	16	null	'\u0000'(0)	'\uFFFF'(65535)
boolean	true 또는 false	8	false	해당없음	해당없음

## 기호 상수

- 상수에 이름을 주어서 변수처럼 사용

**final** double PI = 3.141592;

- 숫자보다 이해하기 쉽고, 값의 변경이 용이하다.

## 자바에서 지원하는 연산자

표 5.4 산술 연산자의 종류

연산자	우선 순위
후위 증감	expr++ expr--
단항	++expr --expr +expr -expr ~ !
곱셈	* / %
덧셈	+ -
이동	<< >>>>
관계	< > <= >= instanceof
동등	== !=
비트별 AND	&
비트별 XOR	^
비트별 OR	
논리적 AND	&&
논리적 OR	
조건	? :
대입	= += -= *= /= %= &= ^=  = <<= >>= >>>=

## 관계 연산자

표 5.9 관계 연산자

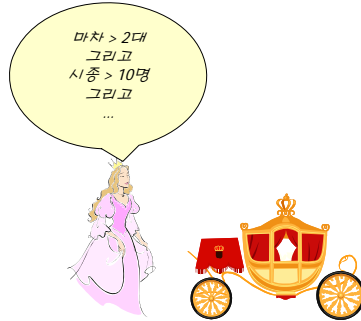
연산자 기호	의미	사용예
==	x와 y가 같은가?	x == y
!=	x와 y가 다른가?	x != y
>	x가 y보다 큰가?	x > y
<	x가 y보다 작은가?	x < y
>=	x가 y보다 크거나 같은가?	x >= y
<=	x가 y보다 작거나 같은가?	x <= y



## 논리 연산자

연산자 기호	사용예	의미
&&	x && y	AND 연산, x와 y가 모두 참이면 참, 그렇지 않으면 거짓
	x    y	OR 연산, x나 y중에서 하나만 참이면 참, 모두 거짓이면 거짓
!	!x	NOT 연산, x가 참이면 거짓, x가 거짓이면 참

표 10 논리 연산자



## 형변환

- **형변환(type casting)**은 어떤 자료형의 값을 다른 자료형의 값으로 바꾸어 주는 연산

(새로운 자료형) 수식;

y = (double) x;

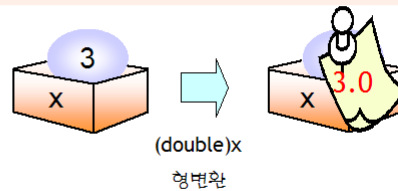


그림 5.21 형변환

## 축소 변환

- `i = (int) 12.5;` // i에는 12만 저장
- (주의) 위의 예에서는 소수점 이하는 사라진다.



## 확대 변환

- 더 큰 크기의 변수로 값을 이동하는 변환
- `double d = (double) 100;` // 정수 100이 변수 d에 100.0으로 형변환되어서 저장

## 예제

TypeConversion.java

```
public class TypeConversion {
    public static void main(String args[]) {
        int i;
        double f;

        f = 5 / 4;                // (a) f는 1.0
        System.out.println(f);
        f = (double) 5 / 4;       // (b) f는 1.25
        System.out.println(f);
        f = 5 / (double) 4;       // (c) f는 1.25
        System.out.println(f);
        f = (double) 5 / (double) 4; // (d) f는 1.25
        System.out.println(f);
        i = (int) 1.3 + (int) 1.8; // (e) i는 2
        System.out.println(i);
    } // end method main
} // end class TypeConversion
```

실행결과

```
1.0
1.25
1.25
1.25
2
-
```

## 3가지의 제어 구조

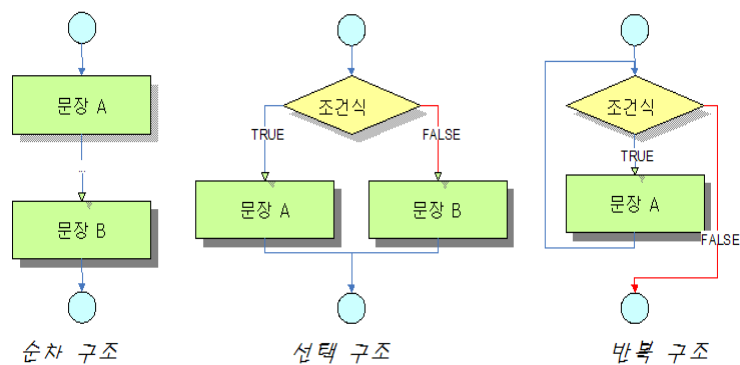
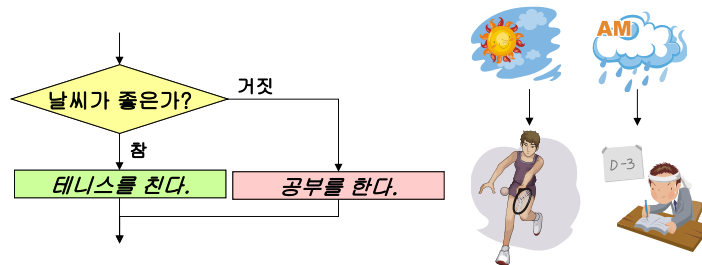


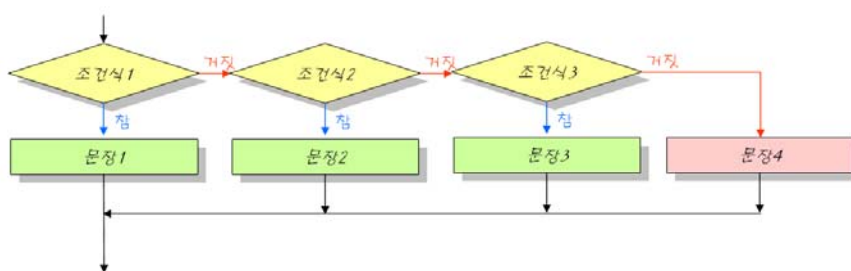
그림 6.1 3가지의 제어 구조

## if-else 문



<pre> if ( 조건식 )     문장 1; else     문장 2;             </pre>	<div style="border: 1px solid black; padding: 2px; width: fit-content; margin-bottom: 10px;">조건식이 참이면 실행된다.</div> <div style="border: 1px solid black; padding: 2px; width: fit-content;">조건식이 거짓이면 실행된다.</div>
--	---

## 연속적인 if

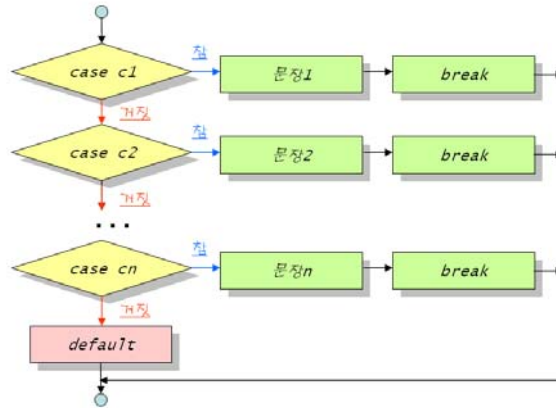


```

if( 조건식 1 )
    문장 1;
else if( 조건식 2 )
    문장 2;
else if( 조건식 3 )
    문장 3;
else
    문장 4;
            
```

## switch 문

- 여러 가지 경우 중에서 하나를 선택하는데 사용



## switch 문의 예

```
import java.util.*;
public class SwitchExample {
    public static void main(String[] args) {
        int number;

        Scanner scan = new Scanner(System.in);
        System.out.print("숫자를 입력하시오: ");
        number = scan.nextInt();
        switch (number) {
            case 0:
                System.out.println("없음");
                break;
            case 1:
                System.out.println("하나");
                break;
            case 2:
                System.out.println("둘");
                break;
            default:
                System.out.println("많음");
                break;
        }
    }
}
```

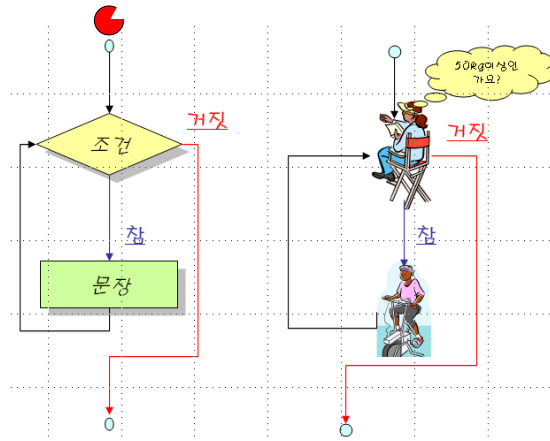
사용자가  
1을  
입력하는  
경우



## while 문

- 주어진 조건이 만족되는 동안 문장들을 반복 실행한다.

while( 조건식 )  
문장;

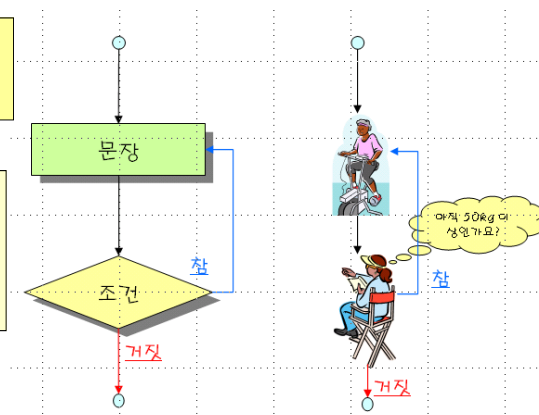


## do...while문

- 반복 조건을 루프의 끝에서 검사

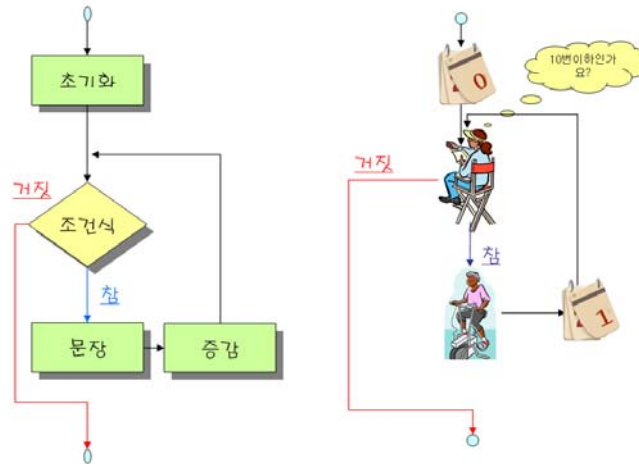
do  
문장  
while(조건)

- 문장들이 실행된다.
- 조건식이 계산된다.
- 결과가 참이면 ①로 돌아간다.
- 결과가 거짓이면 종료된다.



## for 루프

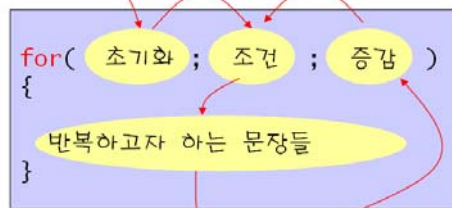
- 정해진 횟수만큼 반복하는 구조



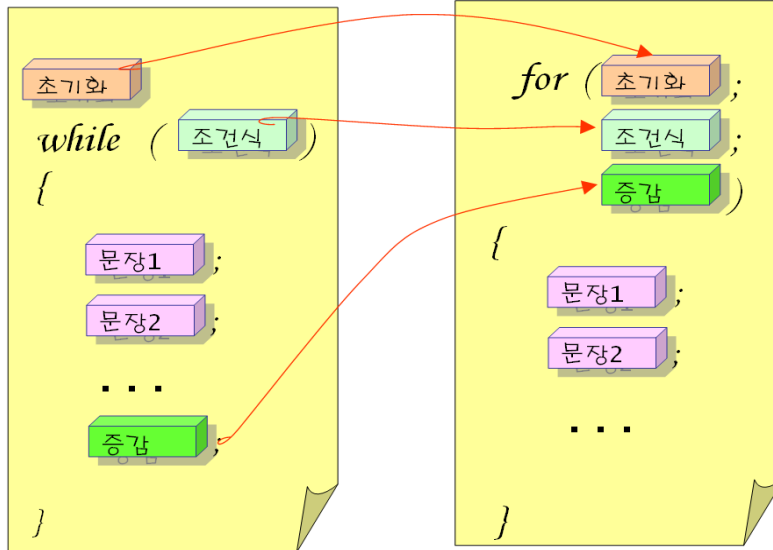
## for 문의 구조

**for** ( 초기화; 조건식; 증감식)  
문장;

- ① 초기화를 실행한다.
- ② 반복 조건을 나타내는 조건식을 계산한다.
- ③ 수식의 값이 거짓이면 for 문의 실행이 종료된다.
- ④ 수식의 값이 참이면 문장이 실행된다.
- ⑤ 증감을 실행하고 ②로 돌아간다.

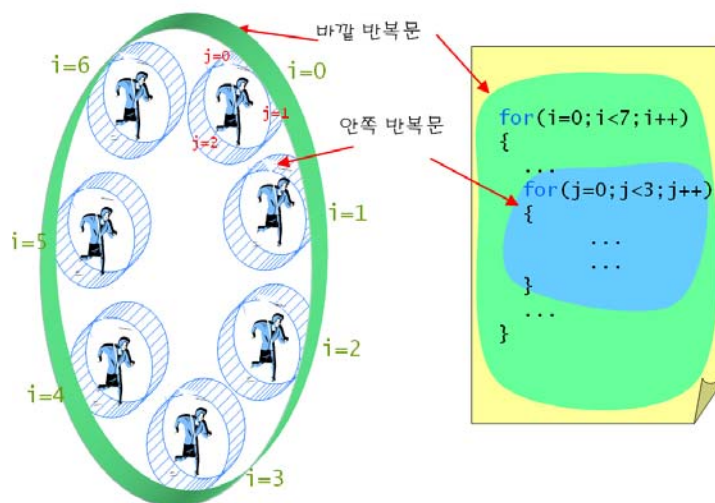


## while 루프와 for 루프와의 관계



## 중첩 반복문

- 중첩 반복문(nested loop): 반복문 안에 다른 반복문이 위치





## 기초 변수와 참조 변수

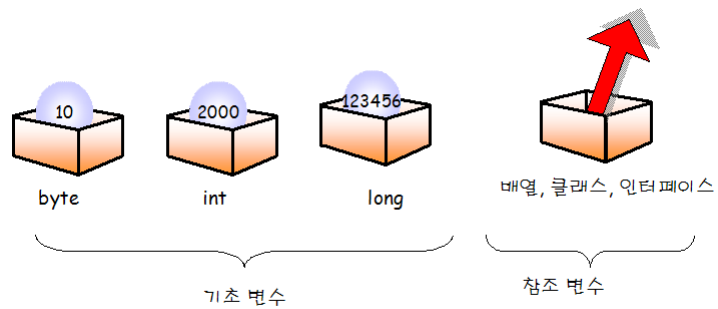


그림 7.12 변수의 종류

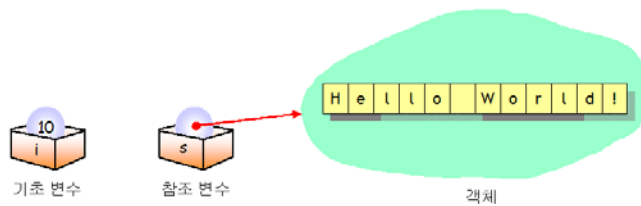
## 기초 변수와 참조 변수의 비교

```
int i; // 기초 변수
String s; // 참조 변수
```

위와 같이 정의하던 두 변수 모두 처음에는 데이터를 담고 있지 않다. 즉 초기화가 되지 않은 상태이다.



```
i = 10; // 기초 변수에 값을 대입
s = new String("Hello World!"); // 객체를 생성하고 참조 변수에 객체의 주소를 대입
```



## 클래스 정의의 예

```
class Car {  
    // 필드 정의  
    public int speed;    // 속도  
    public int mileage;  // 주행거리  
    public String color; // 색상  
  
    // 메소드 정의  
    public void speedUp() { // 속도 증가 메소드  
        speed += 10;  
    }  
  
    public void speedDown() { // 속도 감소 메소드  
        speed -= 10;  
    }  
  
    public String toString() { // 객체의 상태를 문자열로 반환하는 메소드  
        return "속도: " + speed + " 주행거리: " + mileage + " 색상: " + color;  
    }  
}
```

## 객체의 생성

```
Car myCar; // ① 참조 변수를 선언  
myCar = new Car(); // ② 객체를 생성하고 ③ 참조값을 myCar에 저장
```

### ① 참조 변수 선언

Car 타입의 객체를 참조할 수 있는 변수 myCar를 선언한다.



### ② 객체 생성

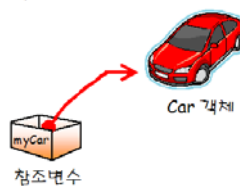
new 연산자를 이용하여 객체를 생성하고 객체 참조값을 반환한다.



Car 객체

### ③ 참조 변수와 객체의 연결

생성된 새로운 객체의 참조값을 myCar라는 참조 변수에 대입한다.

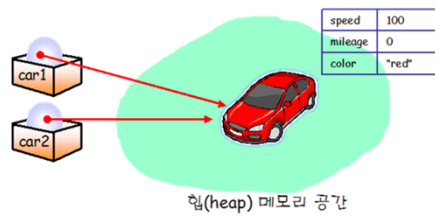


## 참조 변수와 대입 연산

- Car car1 = new Car();

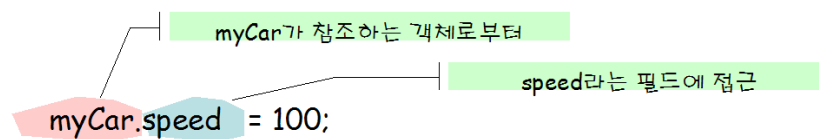


- Car car2 = car1; // 대입 연산의 의미



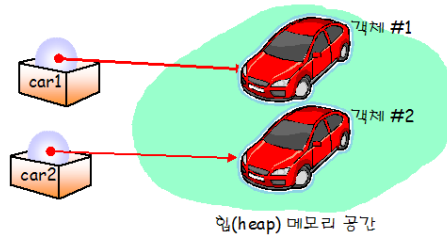
## 객체의 사용

- 객체를 이용하여 필드와 메소드에 접근할 수 있다.

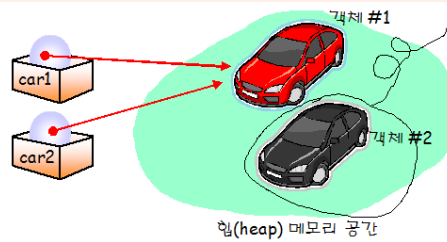


## 객체의 소멸

```
Car car1 = new Car(); // 첫번째 객체
Car car2 = new Car(); // 두번째 객체
```



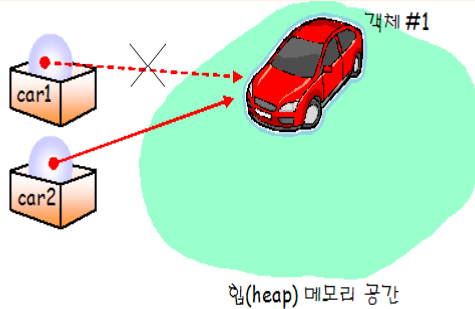
```
car2 = car1; // car1과 car2는 같은 객체를 가리킨다.
// car2가 가리켰던 객체는 쓰레기 수집기에 의하여 수거된다.
```



객체는  
참조가  
없어지면  
소멸!!

## 객체의 소멸

```
car1 = null; // 객체 1은 아직도 활성화된 참조가 있기 때문에 소멸되지 않는다.
```



## 매개 변수

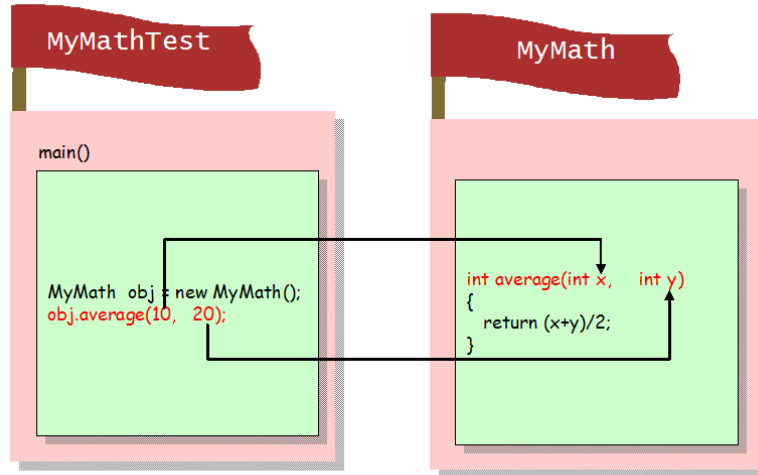
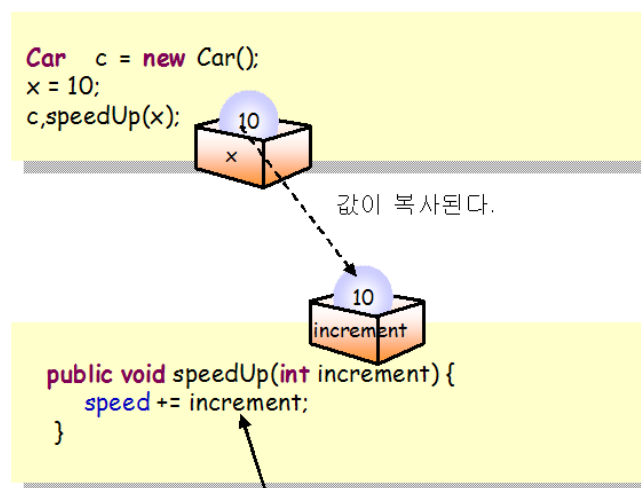


그림 8.7 메소드 호출시 매개 변수의 전달

## 값에 의한 전달



매개 변수는 메소드 안에서 변수로 사용된다.

그림 8.8 매개 변수가 기초 타입의 변수일 경우, 값이 복사된다.

## 매개 변수가 객체인 경우

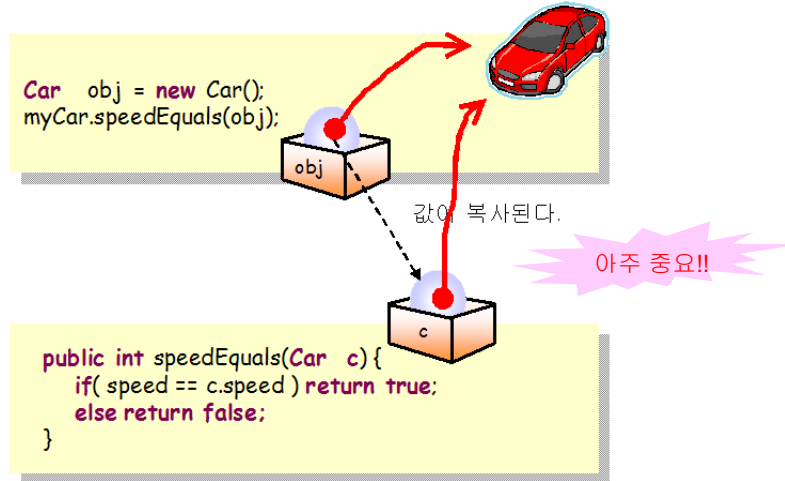
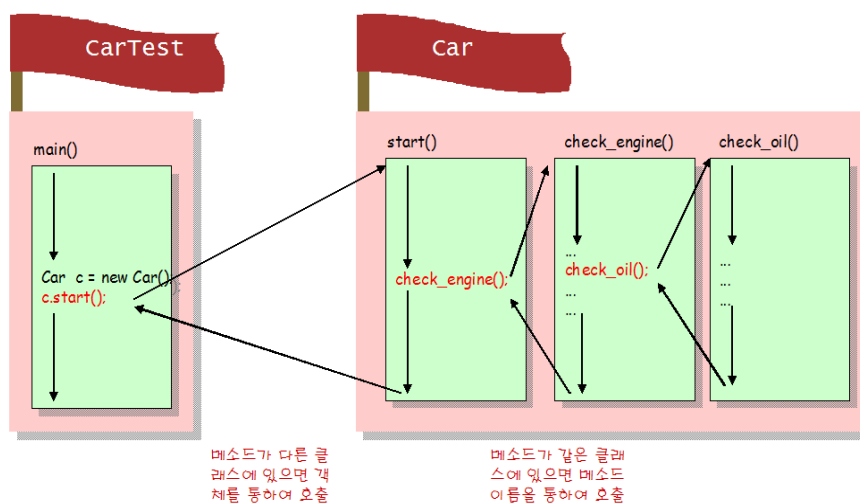


그림 8.9 매개 변수가 참조 타입의 변수일 경우에도 역시 참조값이 복사된다.

## 메소드 호출



## 중복 메소드

- **메소드 오버로딩(method overloading)**
  - 같은 이름의 메소드가 여러 개 존재하는 것
    - 단, 매개변수의 타입이나 개수는 다름

```
// 정수값을 제공하는 메소드
public int square(int i)
{
    return i*i;
}

// 실수값을 제공하는 메소드
public double square(double i)
{
    return i*i;
}
```

- 메소드 호출시 매개 변수를 보고 일치하는 메소드가 호출된다.
- 만약 square(3.14)와 같이 호출되면 컴파일러는 매개 변수의 개수, 타입, 순서 등을 봐서 두 번째 메소드를 호출한다.

## 중복 메소드 예제



```
class Car {

    // 필드 선언
    private int speed; // 속도

    // 중복 메소드: 정수 버전
    public void setSpeed(int s) {
        speed = s;
        System.out.println("정수 버전 호출");
    }

    // 중복 메소드: 실수 버전
    public void setSpeed(double s) {
        speed = (int)s;
        System.out.println("실수 버전 호출");
    }

}
```

## 중복 메소드 예제



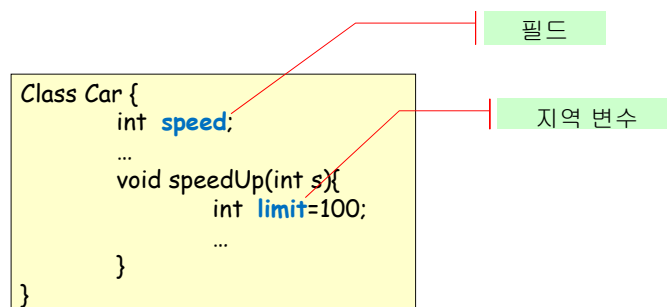
```
public class CarTest1 {  
    public static void main(String[] args) {  
        Car myCar = new Car(); // 첫번째 객체 생성  
  
        myCar.setSpeed(100); // 정수 버전 메소드 호출  
        myCar.setSpeed(79.2); // 실수 버전 메소드 호출  
    }  
}
```



정수 버전 호출  
실수 버전 호출

## 자바에서의 변수의 종류

- **필드(field)** 또는 **인스턴스 변수**: 클래스 안에서 선언되는 멤버 변수
- **지역 변수(local variable)**: 메소드나 블록 안에서 선언되는 변수





## 필드의 사용 범위

Date.java

```
public class Date {
```

```
    public void printDate() {  
        System.out.println(year + "." + month + "." + day);  
    }
```

```
    public int getDay() {  
        return day;  
    }
```

```
    // 필드 선언  
    public int year;  
    public String month;  
    public int day;  
}
```

선언 위치와는 상관없이 어디서나 사용이 가능하다.

## 설정자와 접근자

- **설정자 (mutator) = setter**
  - 필드의 값을 설정하는 메소드
  - setXxx() 형식
- **접근자 (accessor) = getter**
  - 필드의 값을 반환하는 메소드
  - getXxx() 형식

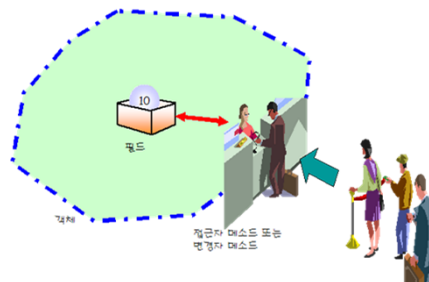


그림 8.12 접근자와 변경자 메소드만을 통하여 필드에 접근한다.

## 설정자와 접근자의 예

```
CarTest2.java
1 class Car {
2
3     // 필드 선언
4     private int speed; // 속도
5     private int mileage; // 주행거리
6     private String color; // 색상
7
8     // 접근자 선언
9     public int getSpeed() {
10         return speed;
11     }
12
13     // 설정자 선언
14     public void setSpeed(int s) {
15         speed = s;
16     }
17
18     // 접근자 선언
19     public int getMileage() {
20         return mileage;
21     }
```

```
23     // 설정자 선언
24     public void setMileage(int m) {
25         mileage = m;
26     }
27
28     // 접근자 선언
29     public String getColor() {
30         return color;
31     }
32
33     // 설정자 선언
34     public void setColor(String c) {
35         color = c;
36     }
37 }
```

## 설정자와 접근자의 사용

```
39 public class CarTest2 {  
40     public static void main(String[] args) {  
41         // 객체 생성  
42         Car myCar = new Car();  
43  
44         // 설정자 메소드 호출  
45         myCar.setSpeed(100);  
46         myCar.setMileage(0);  
47         myCar.setColor("red");  
48  
49         // 접근자 메소드 호출  
50         System.out.println("현재 자동차의 속도는 " + myCar.getSpeed());  
51         System.out.println("현재 자동차의 주행거리는 " + myCar.getMileage());  
52         System.out.println("현재 자동차의 색도는 " + myCar.getColor());  
53     }  
54 }
```

## 필드의 초기화

- 필드값은 선언과 동시에 초기화 될 수 있다.

```
public class Classroom {  
    public static int capacity = 60; //60으로 초기화  
    private boolean use = false; // false로 초기화  
}
```

- 초기값 지정 않으면,
  - int 등 수치 타입은 0
  - 논리 타입은 false
  - 참조 타입은 null
- 생성자를 사용하여 초기화하는 방법도 있다.

## 변수와 변수의 비교

- “변수1 == 변수2”의 의미

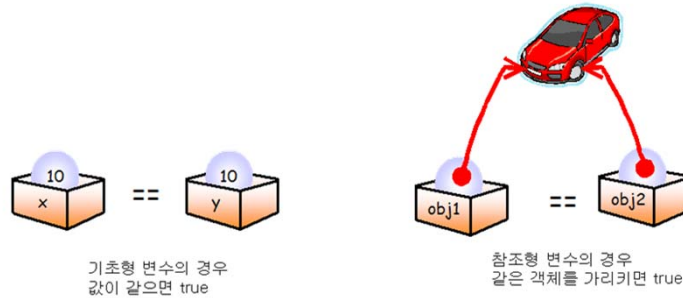


그림 8.14 변수의 비교

- 참조형 변수의 경우, 객체의 내용이 같다는 의미가 아니다.

## 생성자

- 생성자(constructor):
  - 객체가 생성될 때에 필드에게 초기값을 제공하고 필요한 초기화 절차를 실행하는 메소드
  - 객체가 생성되고 나서 실행해야 할 명령문을 써 두는 부분
  - 클래스 안에 선언함
  - 메소드처럼 파라미터를 넘겨줄 수 있음
  - 생성자 이름은 반드시 클래스와 똑같은 이름이어야 함
  - 리턴 타입이 없어야 함



그림 9.1 생성자의 역할

## 생성자의 예제

CarTest.java

```
class Car {  
    public int speed;      // 속도  
    public int mileage;    // 주행 거리  
    public String color;   // 색상  
  
    // 첫 번째 생성자  
    public Car(int s, int m, String c) {  
        speed = s;  
        mileage = m;  
        color = c;  
    }  
    // 두 번째 생성자  
    public Car() {  
        speed = mileage = 0;  
        color = "red";  
    }  
}
```

## 생성자의 예제

```
public class CarTest {  
    public static void main(String args[]) {  
        Car c1 = new Car(100, 0, "blue"); // 첫 번째 생성자 호출  
        Car c2 = new Car(); // 두 번째 생성자 호출  
    }  
}
```

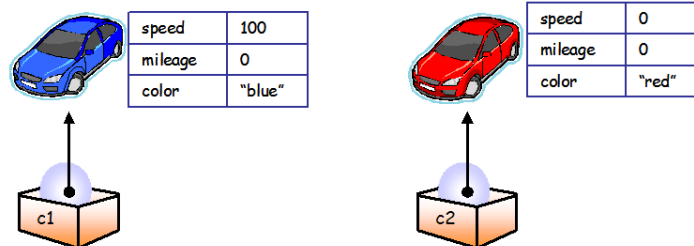


그림 9.2 생성자를 통한 객체의 초기화

## 생성자에서 메소드 호출

- 같은 클래스의 생성자를 호출할 때 **this** 키워드 사용

Car.java

```
public class Car {  
    public int speed; // 속도  
    public int mileage; // 주행 거리  
    public String color; // 색상  
  
    // 첫 번째 생성자  
    public Car(int s, int m, String c) {  
        speed = s;  
        mileage = m;  
        color = c;  
    }  
    // 색상만 주어진 생성자  
    public Car(String c) {  
        this(0, 0, c); // 첫 번째 생성자를 호출한다.  
    }  
}
```

## 정적 변수

- 인스턴스 변수(instance variable): 객체마다 하나씩 있는 변수
- 정적 변수(static variable): 모든 객체를 통틀어서 하나만 있는 변수

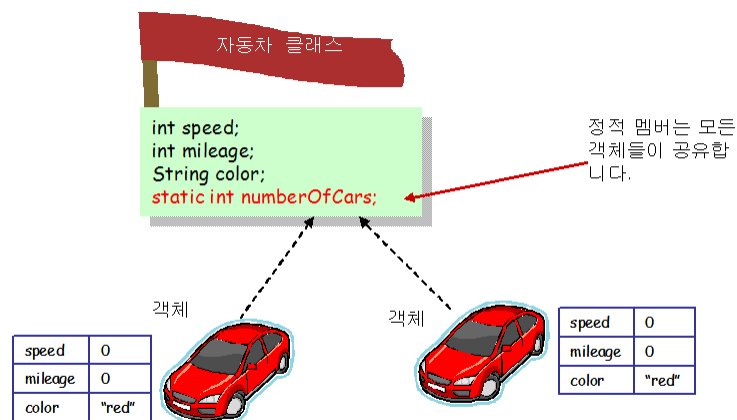


그림 9.4 정적 멤버

## 정적 변수의 예

*Car.java*

```
public class Car {
    private int speed;
    private int mileage;
    private String color;

    // 자동차의 시리얼 번호
    private int id;

    // 실체화된 Car 객체의 개수를 위한 정적 변수
    private static int numberOfCars = 0;

    public Car(int s, int m, String c) {
        speed = s;
        mileage = m;
        color = c;

        // 자동차의 개수를 증가하고 id 번호를 할당한다.
        id = ++numberOfCars;
    }
}
```

## 정적 메소드

- 정적 메소드(static method)
  - 객체를 생성하지 않고 사용할 수 있는 메소드
- (예) Math 클래스에 들어 있는 각종 수학 메소드들

```
double value = Math.sqrt(9.0);
```

## 정적 메소드의 예

CarTest3.java

```
class Car {
    private int speed;
    private int mileage;
    private String color;
    // 자동차의 시리얼 번호
    private int id;
    // 실제화된 Car 객체의 개수를 위한 정적 변수
    private static int numberOfCars = 0;

    public Car(int s, int m, String c) {
        speed = s;
        mileage = m;
        color = c;
        // 자동차의 개수를 증가하고 id 번호를 할당한다.
        id = ++numberOfCars;
    }
    // 정적 메소드
    public static int getNumberOfCars() {
        return numberOfCars; // OK!
    }
}
```

## 정적 메소드의 예

```
public class CarTest3 {
    public static void main(String args[]) {
        Car c1 = new Car(100, 0, "blue"); // 첫 번째 생성자 호출
        Car c2 = new Car(0, 0, "white"); // 첫 번째 생성자 호출
        int n = Car.getNumberOfCars(); // 정적 메소드 호출
        System.out.println("지금까지 생성된 자동차 수 = " + n);
    }
}
```

실행결과

지금까지 생성된 자동차 수 = 2



## 상수

- 공간을 절약하기 위하여 정적 변수로 선언된다.

```
public class Car {  
    ...  
    static final int MAX_SPEED = 350;  
    ...  
}
```

- 정적 메소드 내부에서 인스턴스 메소드를 호출할 수 없다.
- 정적 메소드 내부에서 다른 정적 메소드는 호출가능

```
1 public class Test {  
2  
3     public static void main(String[] args) {  
4         add(10, 20);  
5     }  
6  
7     int add(int x, int y) {  
8         return x + y;  
9     }  
10 }
```

컴파일 에러 발생

```
1 public class Test {  
2  
3     public static void main(String[] args) {  
4         add(10, 20);  
5     }  
6  
7     static int add(int x, int y) {  
8         return x + y;  
9     }  
10 }
```

정상 실행

## 접근 제어

- 접근 제어(access control):
  - 다른 클래스가 특정한 필드나 메소드에 접근하는 것을 제어하는 것

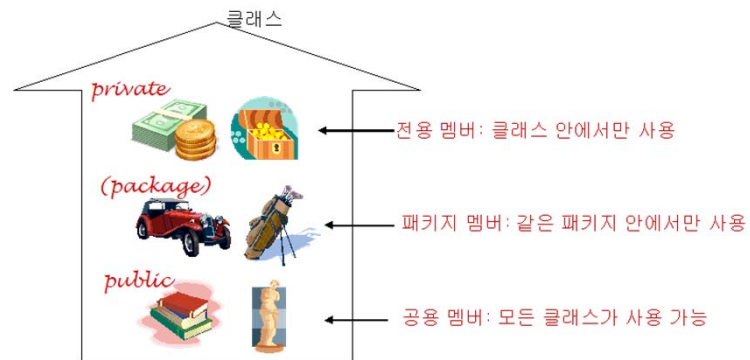


그림 9.5 멤버에 대한 접근 제어

## 클래스 수준에서의 접근 제어

- **public**: 다른 모든 클래스가 사용할 수 있는 공용 클래스
- **package**: 수식자가 없으면 같은 패키지 안에 있는 클래스들만이 사용

패키지(**package**)는  
관련된 클래스를 모  
아둔 것

```
public class myClass {
    ...
}
```

```
class myClass {
    ...
}
```

#### EmployeeTest.java

```
import java.util.*;
class Employee {
    private String name;      // private 로 선언
    private int salary;       // private 로 선언
    int age;                  // package 로 선언

    // 생성자
    public Employee(String n, int a, double s) {
        name = n;
        age = a;
        salary = s;
    }
    // 직원의 이름을 반환
    public String getName() {
        return name;
    }
    // 직원의 월급을 반환
    private int getSalary() { // private 로 선언
        return salary;
    }
    // 직원의 나이를 반환
    int getAge() {           // package로 선언
        return age;
    }
}
```

```
public class EmployeeTest {
    public static void main(String[] args) {
        Employee e;
        e = new Employee("홍길동", 0, 3000);
        e.salary = 300; // 오류! private 변수
        e.age = 26;     // 같은 패키지이므로 OK
        int sa = e.getSalary(); // 오류! private 메소드
        String s = e.getName(); // OK!
        int a = e.getAge();    // 같은 패키지이므로 OK
    }
}
```

#### 실행결과

Exception in thread "main" java.lang.Error: Unresolved compilation problems:  
The field Employee.salary is not visible  
The method getSalary() from the type Employee is not visible  
at EmployeeTest.main(EmployeeTest.java:8)

## this

- 자기 자신을 참조하는 키워드
  - `this.member = 10;`
- 생성자를 호출할 때도 사용된다.
  - `this(10, 20);`

## 배열의 개념

- 배열(array): 같은 타입의 변수들의 모임

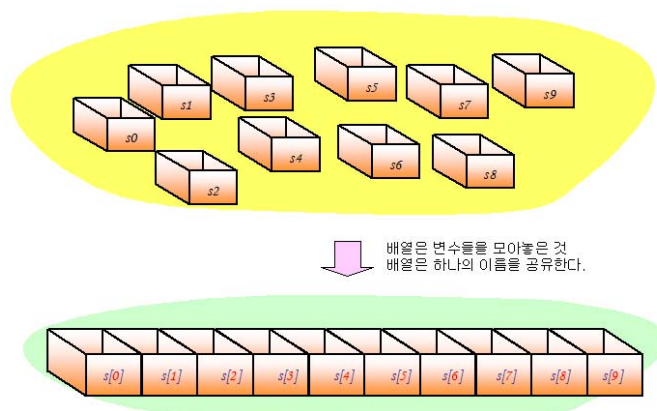
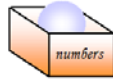


그림 10.1 배열은 변수들의 모임이다.

## 배열을 만드는 절차

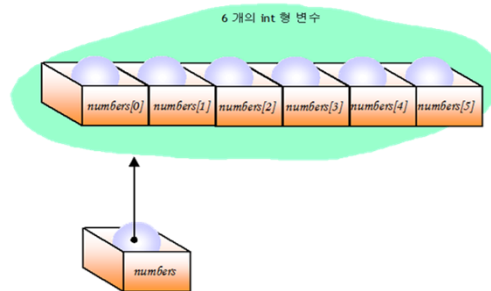
1. 먼저 배열 참조 변수부터 선언

```
int[] numbers;           // 배열 참조 변수 선언
```



2. 배열을 new 연산자를 사용하여 생성

```
numbers = new int[6];     // 배열 객체 생성
```



## 배열 예제

ArrayTest1.java

```
import java.util.Scanner;

public class ArrayTest1 {
    public static void main(String[] args) {
        int[] salary = new int[2];
        Scanner scan = new Scanner(System.in);
        System.out.print("직원1의 월급을 입력하십시오: ");
        salary[0] = scan.nextInt();
        System.out.print("직원2의 월급을 입력하십시오: ");
        salary[1] = scan.nextInt();
        System.out.println("직원1의 월급은 " + salary[0]);
        System.out.println("직원2의 월급은 " + salary[1]);
    }
}
```

## 또 다른 배열 선언 방법

- `int[] values;` // ① 자바 방식 👍
- `int values[];` // ② C언어 유사 방식

## for-each 루프

```
for (자료형 변수 : 배열이름)
{
    //반복 문장들
}
```

*Numbers.java*

```
public class Numbers {
    public static void main(String[] args) {
        int[] numbers = new int[5];
        for (int i = 0; i < numbers.length; i++)
            numbers[i] = (int) (Math.random()*1000);
        for (int value : numbers)
            System.out.println(value);
    }
}
```

실행결과

```
688
773
94
691
349
```

## 예제

Strings2.java

```
public class Strings2 {  
    public static void main(String[] args) {  
        String[] strings = { "Java", "C", "C++" };  
        for (String s : strings)  
            System.out.println(s);  
    }  
}
```

실행결과

```
Java  
C  
C++
```

## 배열을 메소드의 매개 변수로 전달

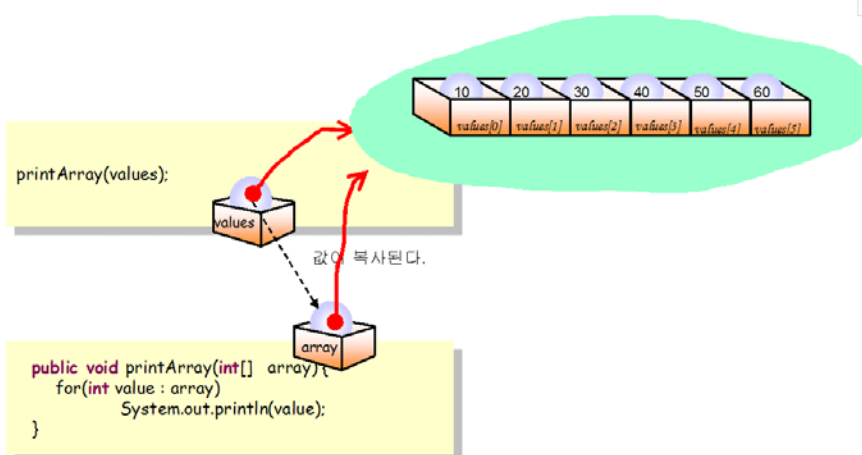


그림 10.2 메소드와 배열

## 객체들의 배열

- 객체들의 배열에서는 객체에 대한 참조값만을 저장  
`Car[] cars = new Car[5];`

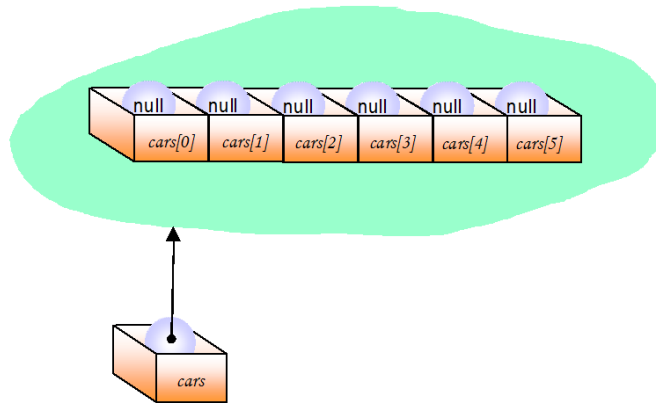


그림 10.3 객체들의 배열

## 객체들의 배열

- 각 원소에 들어가는 객체는 따로 생성하여야 한다.  
`cars[0] = new Cars();`  
`cars[1] = new Cars();`

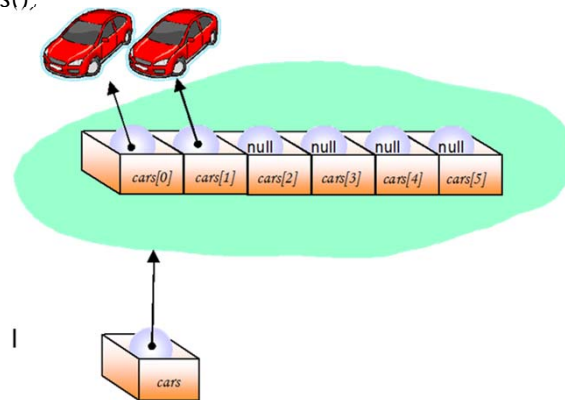
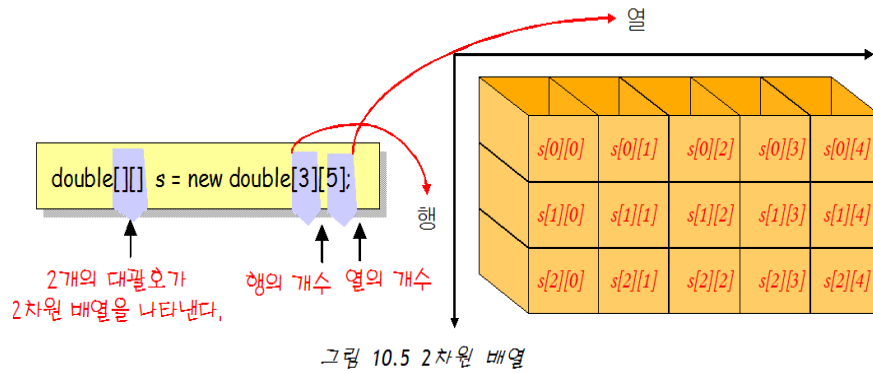


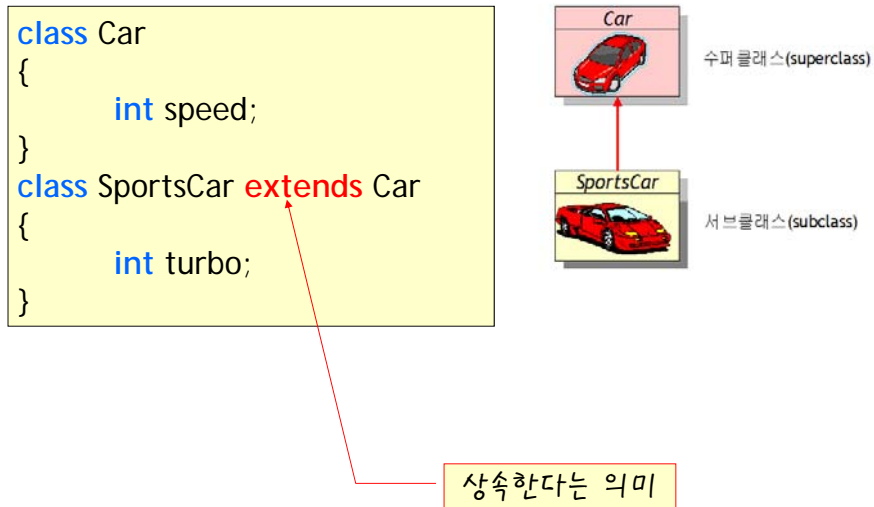
그림 10.4 객체들의 배열은 사실 참조값만을 저장한다.



## 2차원 배열



## 상속



## 수퍼 클래스는 서브 클래스를 포함

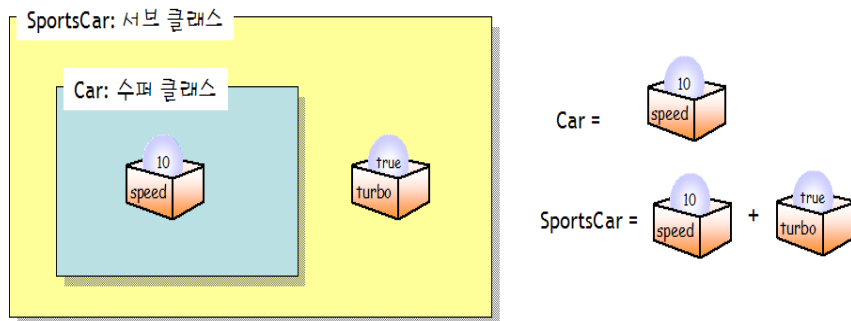


그림 11.3 수퍼 클래스는 서브 클래스를 포함한다.

## 상속의 계층 구조

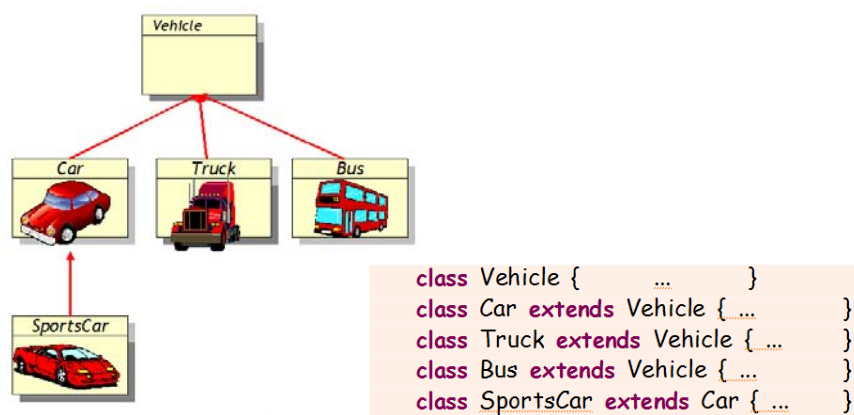


그림 11.4 상속 계층 구조도

## 상속은 중복을 줄인다.

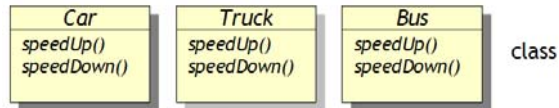


그림 11.6 각 클래스에 코드가 중복된다.

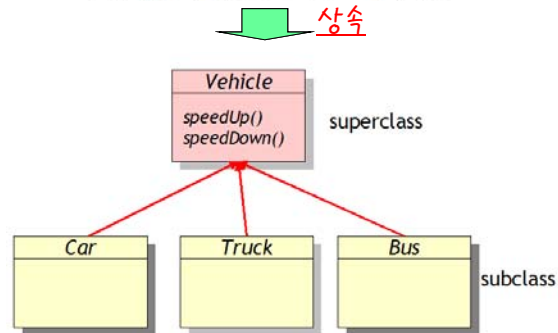


그림 11.7 중복되는 코드는 슈퍼 클래스에 모은다.

## 메소드 재정의

- **메소드 재정의(method overriding):**
  - 서브 클래스가 필요에 따라 상속된 메소드를 다시 정의하는 것

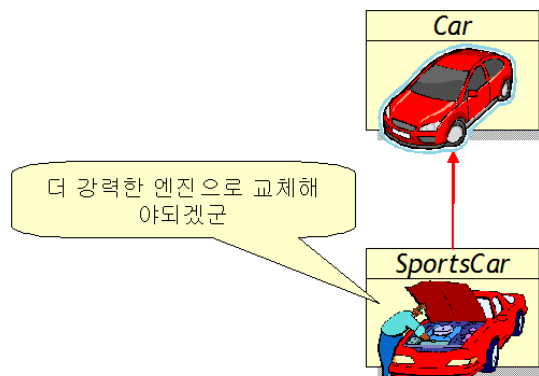
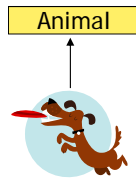


그림 11.9 메소드 재정의

## 메소드 재정의의 예

```
public class Animal {  
    public void makeSound()  
    {  
        // 아직 특정한 동물이 지정되지 않았으므로 몸체는 비어 있다.  
    }  
};
```

```
public class Dog extends Animal {  
    public void makeSound()  
    {  
        System.out.println("멍멍!");  
    }  
};
```



## 메소드를 재정의하려면

- 메소드의 이름, 반환형, 매개 변수의 개수와 데이터 타입이 일치하여야 한다.

```
public class Animal {  
    public void makeSound()  
    {  
    }  
};
```



오버라이드가 아님

```
public class Dog extends Animal {  
    public int makeSound()  
    {  
    }  
};
```

## 중복 정의와 재정의

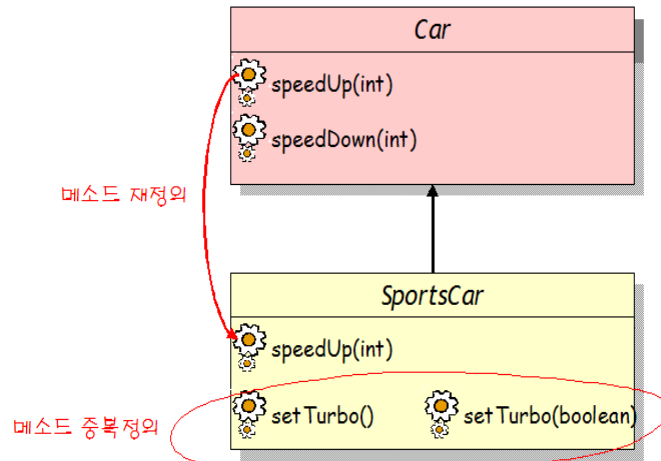


그림 11.10 메소드 재정의와 메소드 중복 정의

## super 키워드

```

class ParentClass {
    int data=100;
    public void print() {
        System.out.println("수퍼 클래스의 print() 메소드");
    }
}

public class ChildClass extends ParentClass {
    int data=200;
    public void print() { //메소드 재정의
        super.print();
        System.out.println("서브 클래스의 print() 메소드 ");
        System.out.println(data);
        System.out.println(this.data);
        System.out.println(super.data);
    }

    public static void main(String[] args) {
        ChildClass obj = new ChildClass();
        obj.print();
    }
}
  
```

수퍼 클래스의 print() 메소드  
서브 클래스의 print() 메소드  
200  
200  
100

수퍼클래스 객체를 가리킨다.

## 접근 지정자

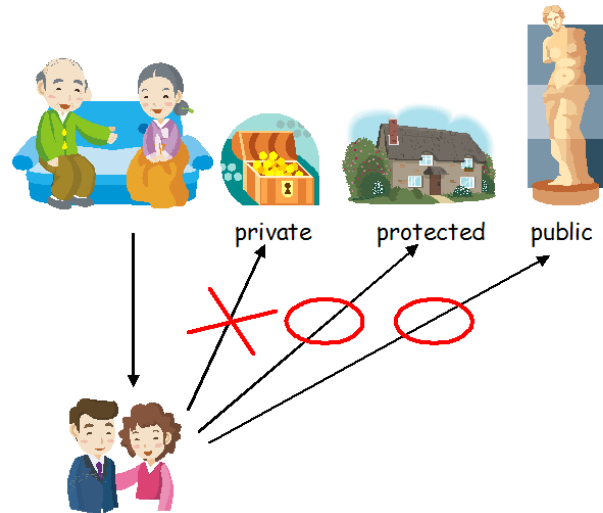


그림 11.12 상속에서의 접근 지정자

- private
  - 외부 객체 접근 불가
- (default)
  - 같은 패키지에 있는 클래스만 접근 가능
- protected
  - 상속 관계에 있거나, 같은 패키지에 있는 클래스만 접근 가능
- public
  - 아무 객체나 접근 가능

## 상속과 생성자

```
class Shape {  
    public Shape(String msg) {  
        System.out.println("Shape 생성자() " + msg);  
    }  
};  
  
public class Rectangle extends Shape {  
    public Rectangle(){  
        super("from Rectangle"); // 명시적인 호출  
        System.out.println("Rectangle 생성자()");  
    }  
  
    public static void main(String[] args) {  
        Rectangle r = new Rectangle();  
    }  
};
```

Shape 생성자 from Rectangle  
Rectangle 생성자

## 묵시적인 호출

```
class Shape {  
    public Shape() {  
        System.out.println("Shape 생성자()");  
    }  
};  
  
public class Rectangle extends Shape {  
    public Rectangle() {  
        System.out.println("Rectangle 생성자()");  
    }  
  
    public static void main(String[] args) {  
        Rectangle r = new Rectangle();  
    }  
};
```

컴파일러가  
Shape();를 자동으로 넣어준다고 생  
각하라.

Shape 생성자  
Rectangle 생성자

## 추상 클래스

- 추상 클래스(abstract class):
  - 몸체가 구현되지 않은 메소드를 가지고 있는 클래스
- 추상 클래스는 추상적인 개념을 표현하는데 적당하다.

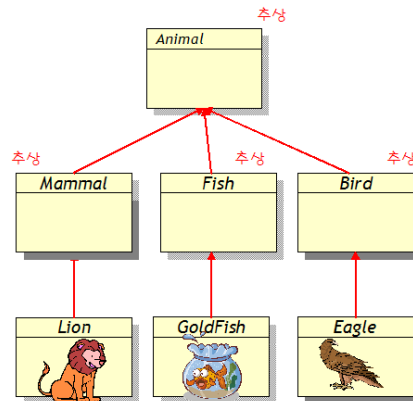


그림 12.1 추상 클래스의 개념

## 추상 클래스의 예

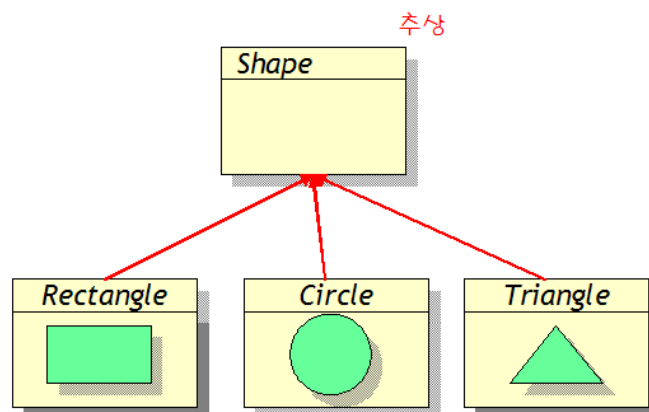


그림 12.2 도형을 나타내는 상속 계층도



## 추상 클래스의 예

```
abstract class Shape { // 추상 클래스 선언
    int x, y;

    public void move(int x, int y) // 일반 메소드 선언
    {
        this.x = x;
        this.y = y;
    }

    public abstract void draw(); // 추상 메소드 선언
}; // ; 으로 종료됨에 유의!

public class Rectangle extends Shape {
    int width, height;

    public void draw() { // 추상 메소드 구현
        System.out.println("사각형 그리기 메소드");
    }
};
```

## 인터페이스

- 인터페이스(interface):
  - 추상 메소드들로만 이루어진다.

```
public interface 인터페이스_이름 {
    반환형 추상메소드1(...); // ; 으로 종료됨에 유의!
    반환형 추상메소드2(...);
    ...
}
```

```
public class 클래스_이름 implements 인터페이스_이름 {
    반환형 추상메소드1(...) {
        .....
    }
    반환형 추상메소드2(...) {
        .....
    }
}
```

## 홈네트워킹 예제

```
public interface RemoteControl {  
    // 추상 메소드 정의  
    public void turnON();    // 가전 제품을 켜다.  
    public void turnOFF();   // 가전 제품을 끄다.  
}
```

인터페이스를 구현

```
public class Television implements RemoteControl {  
    public void turnON()  
    {  
        // 실제로 TV의 전원을 켜기 위한 코드가 들어 간다.  
        ...  
    }  
    public void turnOFF()  
    {  
        // 실제로 TV의 전원을 끄기 위한 코드가 들어 간다.  
        ...  
    }  
}
```

## 홈네트워킹 예제

```
Television t = new Television();  
t.turnOn();  
t.turnOff();
```

Television  
객체를  
생성하여  
메소드들을  
호출



## 인터페이스와 타입

- 인터페이스는 하나의 타입으로 간주된다.

```
public void isEqualSize(Object o1, Object o2) {  
    Comparable co1 = (Comparable)o1;    // 인터페이스 참조 변수  
    Comparable co2 = (Comparable)o2;    // 인터페이스 참조 변수  
    if ( co1.compareTo(co2) == 0)  
        System.out.println("두개의 객체는 같음");  
    else  
        System.out.println("두개의 객체는 같지 않음");  
}
```

인터페이스로 참조 변수를  
만들 수 있다.

## 다중 상속

- 다중 상속이란
  - 여러 개의 슈퍼 클래스로부터 상속하는 것
- 자바에서는 다중 상속을 지원하지 않는다.
- 다중 상속에는 어려운 문제가 발생한다.

```
class SuperA { int x; }  
class SuperB { int x; }  
class Sub extends SuperA, SuperB    // 만약에 다중 상속이 허용된다면  
{  
    ...  
}  
Sub obj = new Sub();  
obj.x = 10;    // obj.x는 어떤 슈퍼 클래스의 x를 참조하는가?
```

## 다중 상속

- 인터페이스를 이용하면 다중 상속의 효과를 낼 수 있다.

```
class Shape {  
    protected int x, y;  
}  
  
interface Drawable {  
    void draw();  
};  
  
public class Rectangle extends Shape implements Drawable {  
    int width, height;  
  
    public void draw() {  
        System.out.println("Rectangle Draw");  
    }  
};
```

## 상수 공유

```
interface Days {  
    public static final int SUNDAY = 1, MONDAY = 2, TUESDAY = 3,  
        WEDNESDAY = 4, THURSDAY = 5, FRIDAY = 6,  
        SATURDAY = 7;  
}
```

```
public class DayTest implements Days  
{  
    public static void main(String[] args)  
    {  
        System.out.println("일요일: " + SUNDAY);  
    }  
}
```

상수를 공유하려면  
인터페이스를 구현하면 된다.

## 다형성이란?

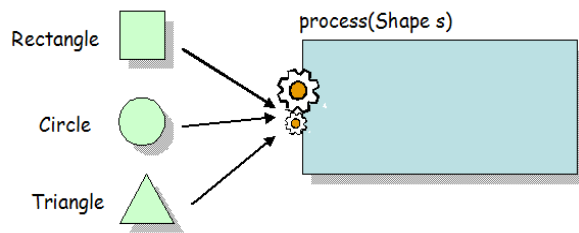


그림 12.5 다형성의 개념

다형성은 다양한 객체들을 하나의 코드로 처리하는 기술입니다.



## 상속과 객체 참조

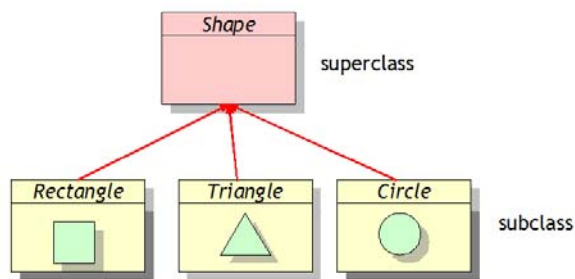


그림 12.6 도형의 상속 구조

Shape s = new Rectangle(); // OK!

Shape 타입 변수로 Rectangle 객체를 참조하니 틀린 거 같지만 올바른 문장!!



## 왜 그럴까?

- 서브 클래스 객체는 수퍼 클래스 객체를 포함하고 있기 때문이다.

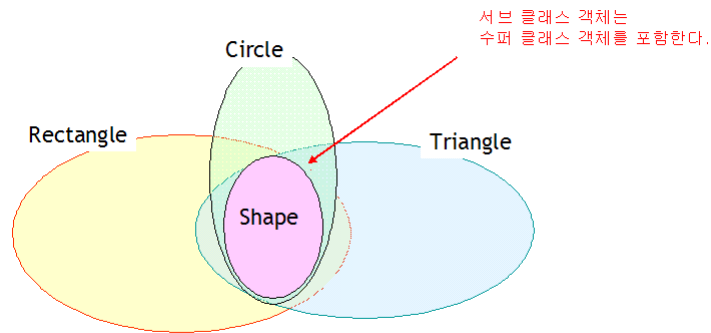


그림 12.7 서브 클래스와 수퍼 클래스의 포함 관계

## 동적 바인딩

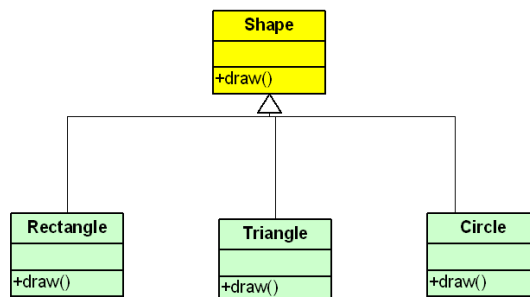


그림 12.8 도형의 UML

```
Shape s = new Rectangle(); // OK!
s.draw();                  // 어떤 draw()가 호출되는가?
```

Shape의 draw()가 호출되는 것이 아니라 Rectangle의 draw()가 호출된다. s의 타입은 Shape이지만 s가 실제로 가리키고 있는 객체의 타입이 Rectangle이기 때문이다.

## 예제

ShapeTest.java

---

```
class Shape {
    protected int x, y;

    public void draw() {
        System.out.println("Shape Draw");
    }
};

class Rectangle extends Shape {
    private int width, height;

    public void setWidth(int w) {
        width = w;
    }

    public void setHeight(int h) {
        height = h;
    }

    public void draw() {
        System.out.println("Rectangle Draw");
    }
};
```

## 예제

```
class Triangle extends Shape {
    private int base, height;

    public void draw() {
        System.out.println("Triangle Draw");
    }
};

class Circle extends Shape {
    private int radius;

    public void draw() {
        System.out.println("Circle Draw");
    }
};
```

## 예제

```
public class ShapeTest {  
    private static Shape arrayOfShapes[];  
  
    public static void main(String arg[]) {  
        init();  
        drawAll();  
    }  
  
    public static void init() {  
        arrayOfShapes = new Shape[3];  
        arrayOfShapes[0] = new Rectangle();  
        arrayOfShapes[1] = new Triangle();  
        arrayOfShapes[2] = new Circle();  
    }  
  
    public static void drawAll() {  
        for (int i = 0; i < arrayOfShapes.length; i++) {  
            arrayOfShapes[i].draw();  
        }  
    }  
};
```

실행결과

Rectangle Draw  
Triangle Draw  
Circle Draw

어떤 draw()가 호출되는가?

## 다형성의 장점

- 만약 새로운 도형 클래스를 작성하여 추가한다고 해보자.

```
class Cylinder extends Shape {  
    public void draw(){  
        System.out.println("Cylinder Draw");  
    }  
};
```

- drawAll() 메소드는 수정할 필요가 없다.

```
public static void drawAll() {  
    for (int i = 0; i < arrayOfShapes.length; i++) {  
        arrayOfShapes[i].draw();  
    }  
}
```



## 객체의 실제 타입을 알아내는 방법

- instanceof 연산자를 사용한다.

```
Shape s = getShape();  
if (s instanceof Rectangle) {  
    System.out.println("Rectangle이 생성되었습니다");  
} else {  
    System.out.println("Rectangle이 아닌 다른 객체가 생성되었습니다");  
}
```

## 메소드의 매개 변수

- 메소드의 매개 변수로 슈퍼 클래스 참조 변수를 이용한다.  
→ 다형성을 이용하는 전형적인 방법



그림 12.9 다형성을 이용하는 메소드의 매개 변수

## 예 제

```
public static double calcArea(Shape s) {  
    double area = 0.0;  
    if (s instanceof Rectangle) {  
        int w = ((Rectangle) s).getWidth();  
        int h = ((Rectangle) s).getHeight();  
        area = (double) (w * h);  
    }  
    ...    // 다른 도형들의 면적을 구한다.  
    return area;  
}
```

## 형 변환

- Shape s = new Rectangle();
- s를 통하여 Rectangle 클래스의 필드와 메소드를 사용하고자 할 때는 어떻게 하여야 하는가?

```
((Rectangle) s).setWidth(100);
```

## 내부 클래스

- 내부 클래스(inner class): 클래스 안에 다른 클래스를 정의

```
public class OuterClass {  
    // 클래스의 필드와 메소드 정의  
    ...  
    private class InnerClass {  
        // 내부 클래스의 필드와 메소드 정의  
        ...  
    }  
}
```

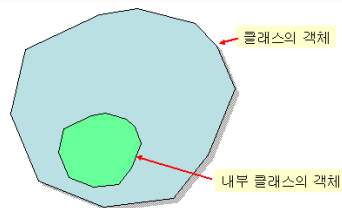


그림 12.10 내부 클래스 객체는 객체 안에 위치한다.

## 내부 클래스의 사용 목적

- 특정 멤버 변수를 **private**로 유지하면서 자유롭게 사용할 수 있다.
- 특정한 곳에서만 사용되는 클래스들을 모을 수 있다.
- 보다 읽기 쉽고 유지 보수가 쉬운 코드가 된다.

## 예제

```
class OuterClass {  
    private String secret = "Time is money";  
  
    public OuterClass() {  
        InnerClass obj = new InnerClass();  
        obj.method();  
    }  
  
    private class InnerClass {  
        public InnerClass() {  
            System.out.println("내부 클래스 생성자입니다.");  
        }  
  
        public void method() {  
            System.out.println(secret);  
        }  
    }  
}  
  
public class OuterClassTest {  
    public static void main(String args[]) {  
        new OuterClass();  
    }  
}
```

내부 클래스

실행결과  
내부 클래스 생성자입니다.  
Time is money

## 무명 클래스

- 무명 클래스(**anonymous class**): 클래스 몸체는 정의되지만 이름이 없는 클래스

```
new ClassOrInterface() { 클래스 몸체 }
```

부모 클래스 이름이나  
인터페이스 이름

## 무명 클래스의 예

```
interface RemoteControl {
    void turnOn();
    void turnOff();
}

public class AnonymousClassTest {
    public static void main(String args[]) {
        RemoteControl ac = new RemoteControl() {           // 무명 클래스 정의
            public void turnOn() {
                System.out.println("TV turnOn()");
            }

            public void turnOff() {
                System.out.println("TV turnOff()");
            }
        };
        ac.turnOn();
        ac.turnOff();
    }
}
```

```
class SatelliteSender extends MessageSender {
    void send(String message) {
        System.out.println("발신: 마이더스");
        System.out.println("수신: 빌 게이츠");
        System.out.println("메시지: " + message);
        System.out.println();
    }
}

SatelliteSender obj = new SatelliteSender();
```

이 부분과  
이 부분을  
삭제합니다

```
MessageSender obj = new MessageSender() {
    void send(String message) {
        System.out.println("발신: 마이더스");
        System.out.println("수신: 빌 게이츠");
        System.out.println("메시지: " + message);
        System.out.println();
    }
};
```

마지막에 세미콜론을 써주어야 합니다

슈퍼클래스의  
생성자를 호출합니다

클래스 본체는  
그대로 둡니다

이런 클래스를 **무명 내부 클래스**  
(**anonymous inner class**)라고 부릅니다.

## 패키지란?

- 패키지(package) : 클래스들을 묶은 것
- 자바 라이브러리도 패키지로 구성
  - (예) java.net 패키지- 네트워크 관련 라이브러리



그림 13.1 패키지의 개념

## 패키지의 장점

- ① 관련된 클래스들을 쉽게 파악
- ② 원하는 클래스들을 쉽게 찾을 수 있다.
- ③ 패키지마다 이름 공간을 따로 갖기 때문에 같은 클래스 이름을 여러 패키지가 사용
- ④ 패키지별로 접근에 제약을 가할 수 있다.

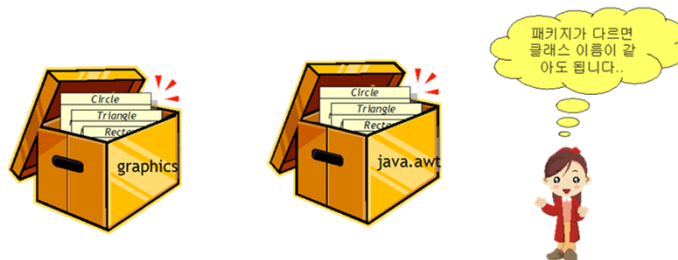


그림 13.2 패키지의 이점

## 패키지 생성하기

```
package business; // 패키지 선언
public class Order {
    ...
}
```

Order 라는 클래스는 business 패키지에 속한다.



Q: 만약 패키지 문을 사용하지 않은 경우에는 어떻게 되는가?

A: 디폴트 패키지(default package)에 속하게 된다.

## 패키지의 이름

- 인터넷 도메인 이름을 역순으로 사용한다.
  - 예를 들면 com.company.test라는 패키지 이름은 도메인 이름 company.com에서의 test라는 프로젝트를 의미한다.

## 소스 파일과 클래스 파일의 위치



Q: 만약 패키지명이 있는 경우에 소스 파일과 클래스 파일의 위치는?

A: 패키지 이름이 디렉토리가 되고 그 아래에 저장된다.

```
package business;  
public class Order {  
    ...  
}
```

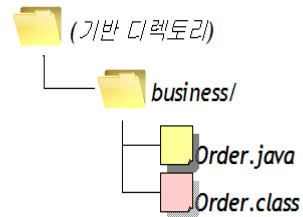


그림 13.3 소스 파일과 클래스 파일의 위치

## 패키지를 사용하는 방법

1. 개별 클래스를 import한다.
2. 전체 패키지를 import한다.
3. 클래스에 패키지 이름을 붙여서 참조한다.

1. import **business.Order**; // business 패키지 안의 Order 클래스 포함
2. Import **business.\***; // 패키지 전체 포함
3. **business.Order** myOrder = new **business.Order**();



## 자바에서 지원하는 패키지

패키지	설명
<a href="#">java.applet</a>	애플릿을 생성하는데 필요한 클래스
<a href="#">java.awt</a>	그래픽과 이미지를 위한 클래스
<a href="#">java.beans</a>	자바빈즈 구조에 기초한 컴포넌트를 개발하는데 필요한 클래스
<a href="#">java.io</a>	입력과 출력 스트림을 위한 클래스
<a href="#">java.lang</a>	자바 프로그래밍 언어에 필수적인 클래스
<a href="#">java.math</a>	수학에 관련된 클래스
<a href="#">java.net</a>	네트워킹 클래스
<a href="#">java.nio</a>	새로운 네트워킹 클래스
<a href="#">java.rmi</a>	원격 메소드 호출(RMI) 관련 클래스
<a href="#">java.security</a>	보안 프레임워크를 위한 클래스와 인터페이스
<a href="#">java.sql</a>	데이터베이스에 저장된 데이터를 접근하기 위한 클래스
<a href="#">java.util</a>	날짜, 난수 생성기 등의 유틸리티 클래스
<a href="#">javax.imageio</a>	자바 이미지 I/O API
<a href="#">javax.net</a>	네트워킹 애플리케이션을 위한 클래스
<a href="#">javax.swing</a>	스윙 컴포넌트를 위한 클래스
<a href="#">javax.xml</a>	XML을 지원하는 패키지

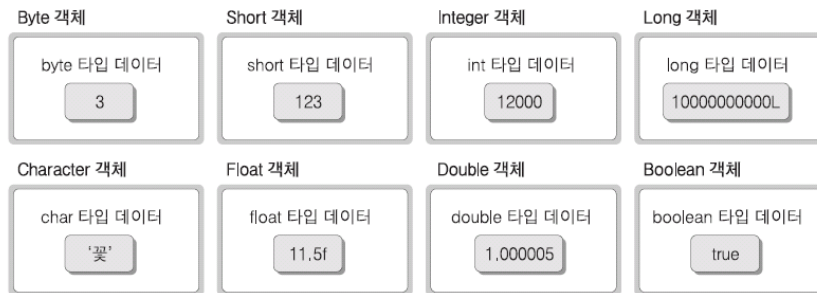
## Java.lang 패키지

- Import 문을 사용할 필요가 없이 기본적으로 포함된다.
  - **Object** 클래스: 기초적인 메소드를 제공하는 모든 클래스의 조상 클래스
  - **Math** 클래스: 각종 수학 함수들을 포함하는 클래스
  - **Wrapper** 클래스: Integer와 같이 기초 자료형을 감싸서 제공하는 래퍼 클래스들
  - **String** 클래스, **StringBuffer** 클래스: 문자열을 다루는 클래스
  - **System** 클래스: 시스템 정보를 제공하거나 입출력을 제공하는 클래스
  - **Thread** 클래스: 스레드 기능을 제공하는 클래스
  - **Class** 클래스: 클래스에 대한 정보를 얻기 위한 클래스

## Wrapper 클래스

- 기초 자료형을 객체로 포장시켜주는 클래스
  - (예) Integer obj = new Integer(10);

기초 자료형	Wrapper 클래스
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean
void	Void



## 문자열 ↔ 기초 자료형

기초 자료형 → 문자열

```
String s1 = Integer.toString(10);
String s2 = Integer.toString(10000);
String s3 = Float.toString(3.14);
String s4 = Double.toString(3.141592);
```

문자열 → 기초 자료형

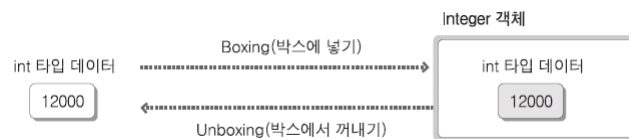
```
int i = Integer.parseInt("10");
long l = Long.parseLong("10000");
float f = Float.parseFloat("3.14");
double d = Double.parseDouble("3.141592");
```

## 오토 박싱(auto-boxing)

- Wrapper 객체와 기초 자료형 사이의 변환을 자동으로 수행한다.

```
Integer box = new Integer(10);
```

```
System.out.println(box + 1); // box는 자동으로 int형으로 변환
```



[예]

```
Integer obj = new Integer(12000); // Boxing  
int num = obj.intValue(); // Unboxing
```

## 제네릭이란?

- 제네릭 프로그래밍(generic programming)
  - 일반적인 코드를 작성하고 이 코드를 다양한 타입의 객체에 대하여 재사용하는 프로그래밍 기법
  - 제네릭은 컬렉션 라이브러리에 많이 사용

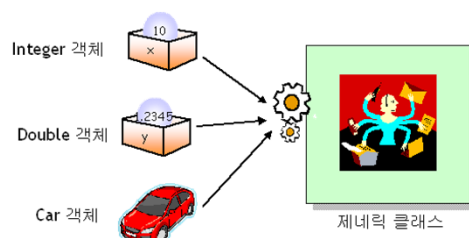
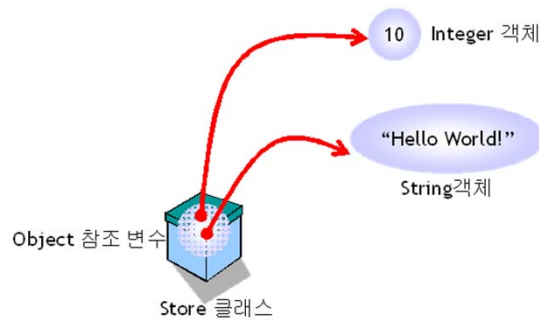


그림 14-1 제네릭 프로그래밍의 개념

## 기존의 방법

- 일반적인 객체를 처리하려면 Object 참조 변수를 사용
- Object 참조 변수는 어떤 객체든지 참조할 수 있다.
- 예제로 하나의 데이터를 저장하는 Store 클래스를 살펴보자.



## 제네릭을 이용한 버전

```
class Store<T> {  
    private T data;           // T 는 타입을 의미한다.  
    public void set(T data) { this.data = data; }  
    public T get() { return data; }  
}
```

- 문자열을 저장하려면 다음과 같이 선언
  - `Store<String> store = new Store<String>();`
- 정수를 저장하려면 다음과 같이 선언
  - `Store<Integer> store = new Store<Integer>();`

```

class Store<T> {
    private T data; // T 는 타입을 의미
    public void set(T data) {
        this.data = data;
    }
    public T get() {
        return data;
    }
}

public class StoreTest {
    public static void main(String args[]) {
        Store<String> s = new Store<String>();
        s.set("Hello world!");
        String str = s.get();
        System.out.println(str);

        Store<Integer> i = new Store<Integer>();
        i.set(new Integer(10));
        Integer a = i.get();
        System.out.println(a);
    }
}

```

## 컬렉션

- 컬렉션(collection)은 자바에서 자료 구조를 구현한 클래스
- 자료 구조로는 리스트(list), 스택(stack), 큐(queue), 집합(set), 해시 테이블(hash table) 등이 있다.

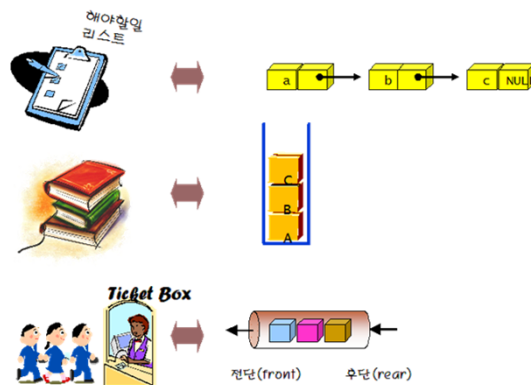
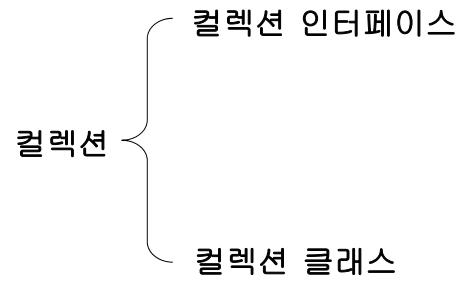


그림 14-4 자료 구조의 예

## 자바에서의 컬렉션



## 컬렉션 인터페이스

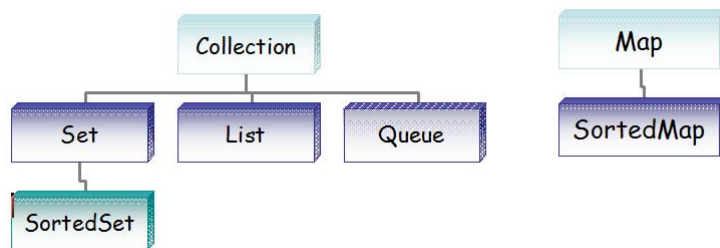
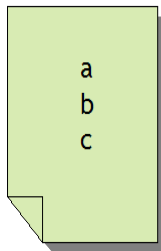


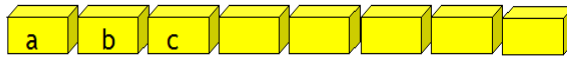
그림 14-5 인터페이스들의 계층 구조

## List 인터페이스

List 인터페이스



ArrayList 클래스

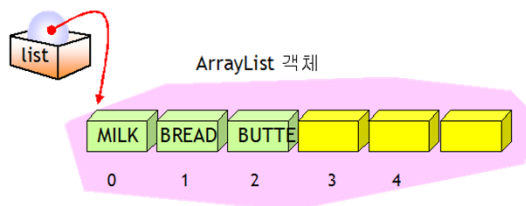


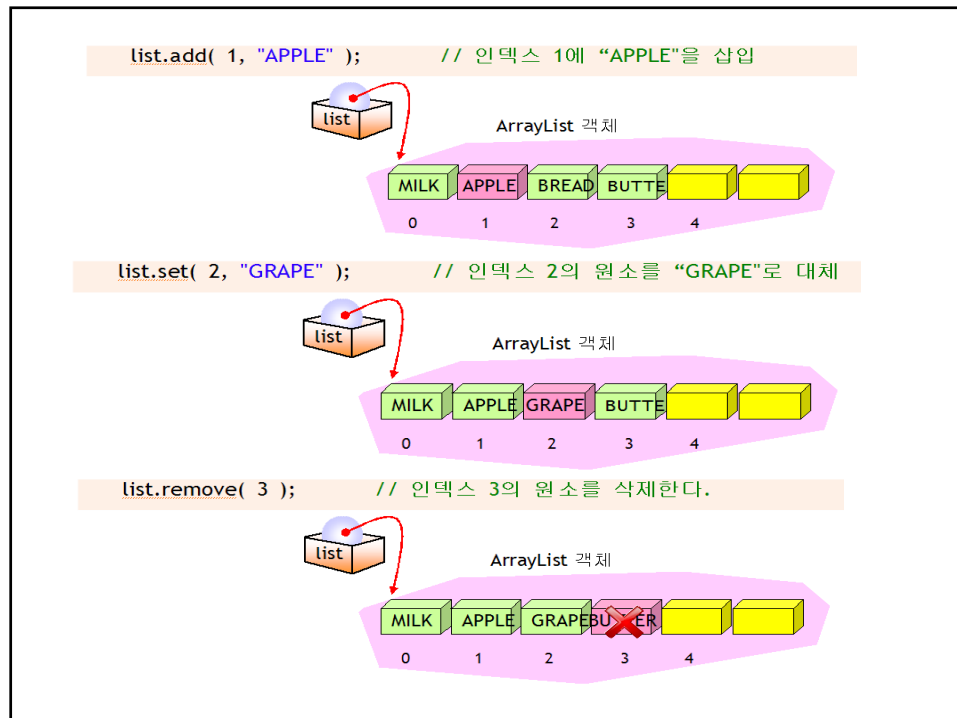
LinkedList 클래스



## ArrayList

- ArrayList
  - 배열(Array)의 향상된 버전 또는 가변 크기의 배열
- ArrayList의 생성
  - `ArrayList<String> list = new ArrayList<String>();`
- 원소 추가
  - `list.add("MILK");`
  - `list.add("BREAD");`
  - `list.add("BUTTER");`





## 예 제

ArrayListTest.java

```
import java.util.*;
```

```
public class ArrayListTest {
```

```
    public static void main(String args[]) {
```

```
        ArrayList<String> list = new ArrayList<String>();
```

```
        list.add("MILK");
```

```
        list.add("BREAD");
```

```
        list.add("BUTTER");
```

```
        list.add(1, "APPLE"); // 인덱스 1에 "APPLE"을 삽입
```

```
        list.set(2, "GRAPE"); // 인덱스 2의 원소를 "GRAPE"로 대체
```

```
        list.remove(3); // 인덱스 3의 원소를 삭제한다.
```

```
        for (int i = 0; i < list.size(); i++)
```

```
            System.out.println(list.get(i));
```

```
    }
```

```
}
```

실행결과

```
MILK
APPLE
GRAPE
```



# LinkedList

- 빈번하게 삽입과 삭제가 일어나는 경우에 사용

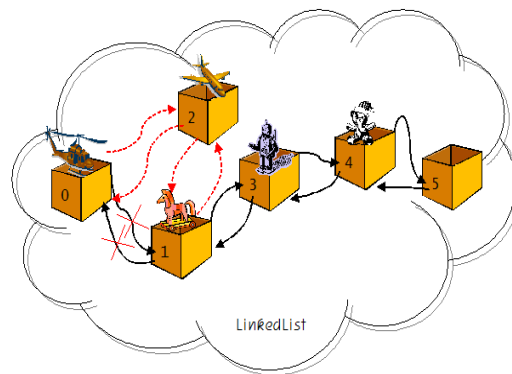


그림 14-8 연결 리스트 중간에 삽입하려면 링크만 수정하면 된다.

## 예 제

LinkedListTest.java

```
import java.util.*;

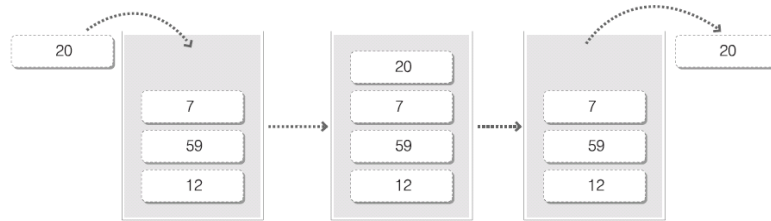
public class LinkedListTest {
    public static void main(String args[]) {
        LinkedList<String> list = new LinkedList<String>();

        list.add("MILK");
        list.add("BREAD");
        list.add("BUTTER");
        list.add(1, "APPLE"); // 인덱스 1에 "APPLE"을 삽입
        list.set(2, "GRAPE"); // 인덱스 2의 원소를 "GRAPE"로 대체
        list.remove(3); // 인덱스 3의 원소를 삭제한다.

        for (int i = 0; i < list.size(); i++)
            System.out.println(list.get(i));
    }
}
```

## ● 스택 (stack)

- 데이터를 넣은 순서의 역순으로만 꺼낼 수 있는 자료구조



- LinkedList로 스택 생성 방법

```
LinkedList<Integer> stack = new LinkedList<Integer>();
```

타입 파라미터

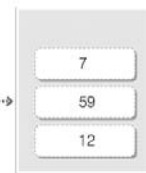
타입 파라미터

- 데이터를 넣는 방법 : push

```
list.addLast(new Integer(12));  
list.addLast(new Integer(59));  
list.addLast(new Integer(7));
```

타입 파라미터에 해당하는  
데이터를 넘겨주어야 합니다.

addLast 메소드는 데이터를  
스택의 아래쪽부터 순서대로 저장합니다.



- 데이터를 꺼내는 방법 : pop

```
Integer obj = removeLast();
```

// 스택의 제일 위에 있는 7을 리턴

```
import java.util.*;
class StackExample1 {
    public static void main(String args[]) {
        LinkedList<Integer> stack = new LinkedList<Integer>();
        stack.addLast(new Integer(12));
        stack.addLast(new Integer(59));
        stack.addLast(new Integer(7));
        while(!stack.isEmpty()) {
            Integer num = stack.removeLast();
            System.out.println(num);
        }
    }
}
```

```
C:\명령 프롬프트
E:\work\chap13\13-2>java StackExample1
7
59
12
E:\work\chap13\13-2>
```

## • 큐 (queue)

- 데이터를 넣은 순서와 같은 순서로만 꺼낼 수 있는 자료구조



- LinkedList를 이용하여 큐 생성하기

```
LinkedList<String> queue = new LinkedList<String>();
```


↑  
타입 파라미터

↑  
타입 파라미터

- 사용 방법 : enqueue

```
queue.offer("토끼");
queue.offer("사슴");
queue.offer("호랑이");
```


파라미터로 넘겨준 데이터를 큐에 저장합니다.



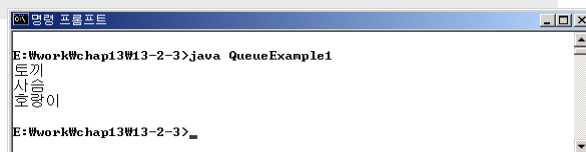
- 사용 방법 : dequeue

```
str = queue.poll();
```

큐의 제일 앞에 있던 "토끼"가 리턴됩니다.



```
import java.util.*;
class QueueExample1 {
    public static void main(String args[]) {
        LinkedList<String> queue = new LinkedList<String>();
        queue.offer("토끼");
        queue.offer("사슴");
        queue.offer("호랑이");
        while(!queue.isEmpty()) {
            String str = queue.poll();
            System.out.println(str);
        }
    }
}
```



```
E:\work\chap13\13-2-3>java QueueExample1
토끼
사슴
호랑이
E:\work\chap13\13-2-3>
```

## 예외란?

- 예외(exception):

- 잘못된 코드, 부정확한 데이터, 예외적인 상황에 의하여 발생하는 오류
- (예) 0으로 나누는 것과 같은 잘못된 연산이나 배열의 인덱스가 한계를 넘을 수도 있고, 디스크에서는 하드웨어 에러가 발생할 수 있다.

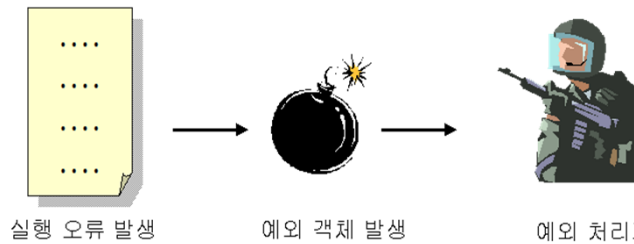


그림 15.1 자바에서는 실행 오류가 발생하면 예외가 생성된다.

## 예외의 예

```
public class DivideByZero {  
    public static void main(String[] args)  
    {  
        int x = 1;  
        int y = 0;  
        int result = x / y; // 예외 발생!  
        System.out.println("이 문장이 출력될까요?");  
    }  
}
```

실행되지  
않음!

실행결과

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at ExceptionTest.main(ExceptionTest.java:5)
```

## 예외 처리기의 개요

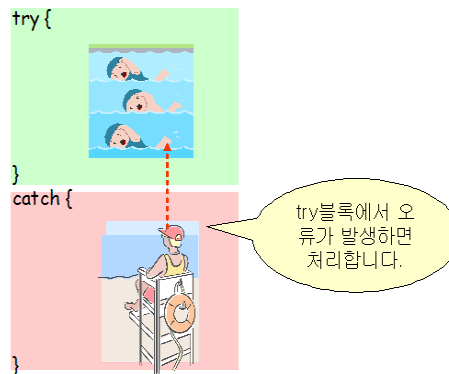


그림 15.2 try블록은 예외가 발생할 수 있는 위험한 코드이다. catch 블록은 예외를 처리하는 코드이다.

## try/catch 블록에서의 실행 흐름

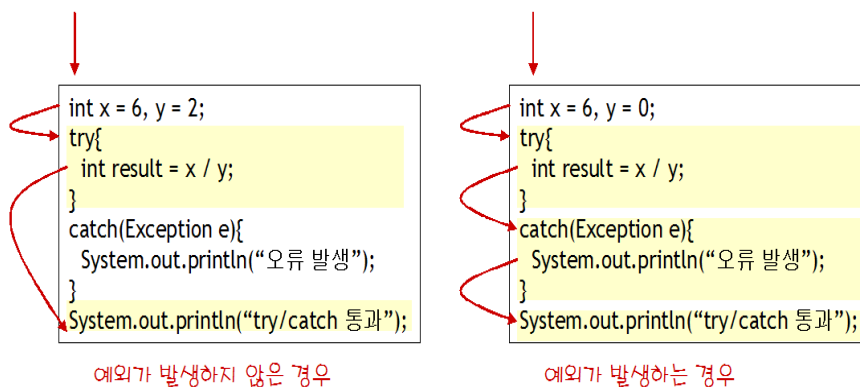


그림 15.3 try/catch 블록에서의 실행 흐름

## finally 블록

- 오류가 발생하였건 발생하지 않았건 항상 실행되어야 하는 코드는 finally 블록에 넣을 수 있다.

```
try {  
    turnTVOn();  
    watchTV();  
} catch (TVException e) {  
    System.out.println("TV를 볼수 없습니다");  
} finally {  
    turnTVOff();  
}
```

## 다형성과 예외

- 다음과 같은 예외 상속 계층도를 가정하자.

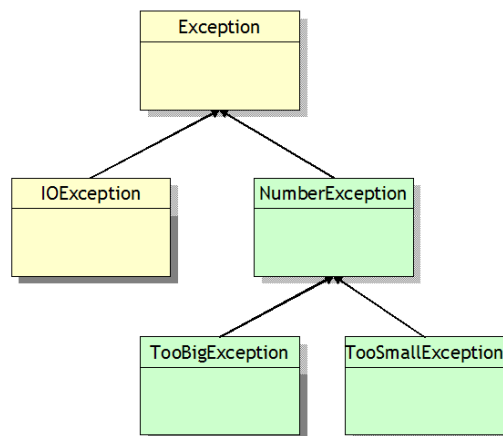


그림 15.5 오류 클래스 상속 계층도

## 다형성과 예외

```
try {
    getInput();
}
catch(NumberException e) {
    // NumberException의 하위 클래스를 모두 잡을 수 있다.
}
```

```
try {
    getInput();
}
catch(Exception e) {
    //Exception의 모든 하위 클래스를 잡을 수 있으니 분간할 수 없다.!!
}
```

## 다형성과 예외

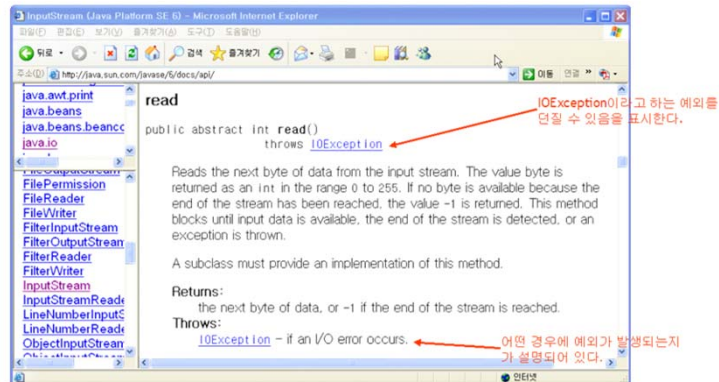
```
try {
    getInput();
}
catch(TooSmallException e) {
    //TooSmallException만 잡힌다.
}
catch(NumberException e) {
    //TooSmallException을 제외한 나머지 예외들이 잡힌다.
}
```

```
try {
    getInput();
}
catch(NumberException e) {
    //모든 NumberException이 잡힌다.
}
catch(TooSmallException e) {
    //아무 것도 잡히지 않는다!
}
```



## 예외가 발생하는 메소드

- 예외를 발생하는 메소드



- 처리 방법
  - 예외를 try/catch로 처리하는 방법
  - 예외를 상위 메소드로 전달하는 방법

## try/catch 블록 이용

```
import java.io.IOException;

public class Test {
    public static void main(String[] args) {
        System.out.println(readString());
    }

    public static String readString() {
        byte[] buf = new byte[100];
        System.out.println("문자열을 입력하십시오:");
        try {
            System.in.read(buf);
        } catch (IOException e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
        return new String(buf);
    }
}
```

예외를 그 자리에서 처리

## 상위 메소드로 전달

```
import java.io.IOException;

public class Test {
    public static void main(String[] args) {
        try {
            System.out.println(readString());
        } catch (IOException e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
    }

    public static String readString() throws IOException {
        byte[] buf = new byte[100];
        System.out.println("문자열을 입력하십시오:");
        System.in.read(buf);
        return new String(buf);
    }
}
```

예외를 상위 메소드로 전달

## 예외 생성하기

- 예외는 **throw** 문장을 이용하여 생성한다.

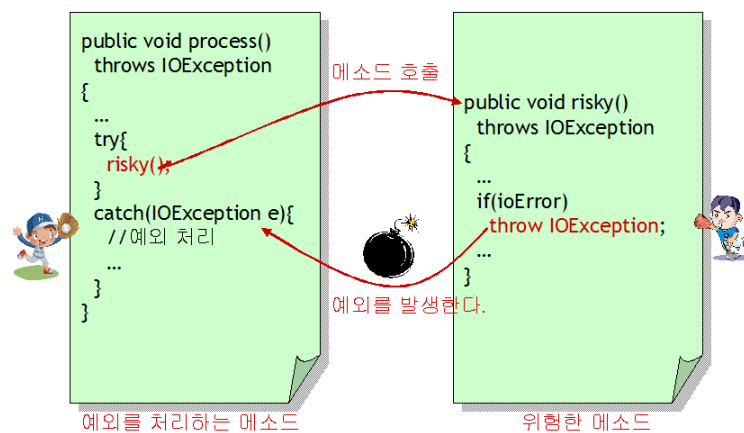


그림 15.7 예외를 던지고 받기

## 예외 생성의 예

```
void allocateMemory()  
{  
    ...  
    if(memory == null)  
        throw new AllocateMemoryFailException();  
    ...  
}
```

예외 객체 생성

## 사용자 정의 예외

- 사용자 정의 예외 클래스
  - Exception 클래스를 상속받아 생성

```
public class MyException extends Exception {  
    ...  
}
```

## 예제

```
class DivideByZeroException extends ArithmeticException {
    public DivideByZeroException()
    {
        super( "0으로 나눌수는 없음." );
    }
}

public class ExceptionTest {
    public static void main(String[] args)
    {
        double result;
        try {
            result = quotient(1,0);
        }
        catch ( DivideByZeroException e ) {
            System.out.println(e.toString());
        }
    }

    public static double quotient( int n, int d )
    throws DivideByZeroException
    {
        if ( d == 0 )
            throw new DivideByZeroException();
        return ( double ) n/ d;
    }
}
```

실행결과

DivideByZeroException: 0으로 나눌수는 없음.