

Constructors and Garbage Collection: Life and Death of an Object

Samkeun Kim <skim@hknu.ac.kr>

<http://cyber.hankyong.ac.kr>

스택과 힙: 객체가 사는 곳

자바에서 모든 객체가 가비지 컬렉션 기능이 있는 힙에서 산다는 것은 이미 알고 있다.

이제 변수가 어디에서 사는지 알아보자.

변수가 사는 곳은 변수의 종류에 따라 달라진다: **인스턴스 변수, 지역 변수**

스택



메소드 호출과

지역변수가 사는 곳

힙



모든 객체가 사는 곳

스택과 힙: 객체가 사는 곳

인스턴스 변수

클래스 안에서 선언된 변수

```
public class Duck {  
    int size;  
}
```

← Every Duck has a "size" instance variable.

지역 변수

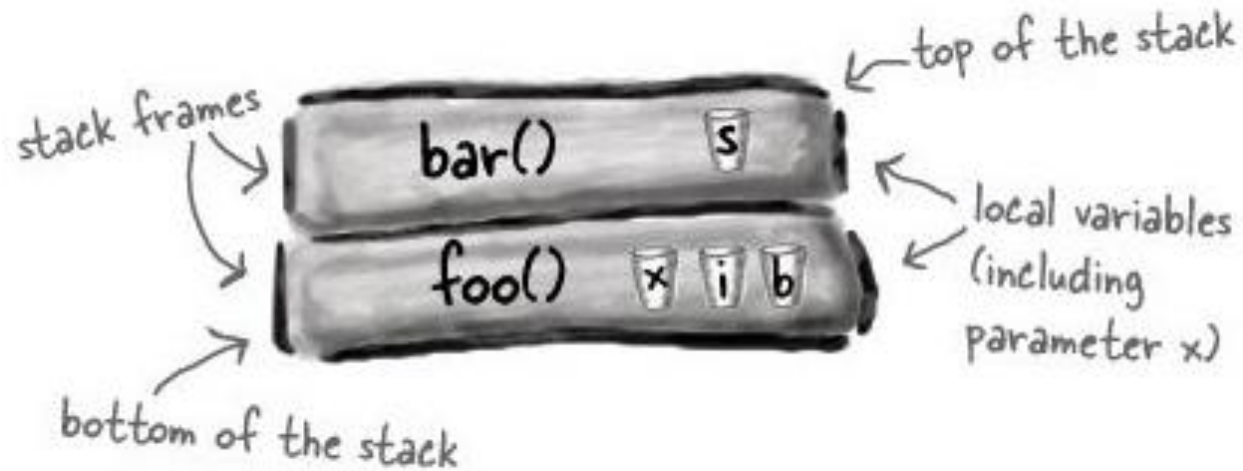
메소드 안에서 선언된 변수

```
public void foo(int x) {  
    int i = x + 3;  
    boolean b = true;  
}
```

The parameter x and the variables i and b are all local variables.

메소드는 스택에 쌓인다

A call stack with two methods:



스택 맨 위에 있는 메소드는 항상 현재 실행 중인 메소드이다.

main() ↗ call

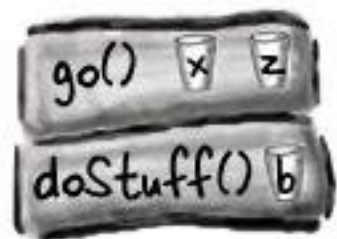
```
public void doStuff() {  
    boolean b = true;  
    go(4);  
}  
public void go(int x) {  
    int z = x + 24;  
    crazy();  
    // imagine more code here  
}  
public void crazy() {  
    char c = 'a';  
}
```

스택 시나리오

1. 다른 클래스의 코드에서 **doStuff()**을 호출하면, **doStuff()**이 스택 맨 위의 스택 프레임으로 들어간다. 부울 변수 'b'도 **doStuff()** 스택 프레임으로 들어간다.



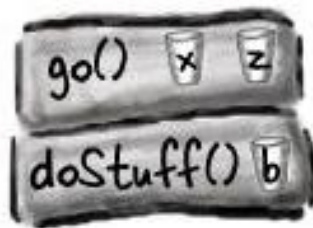
2. **doStuff()**에서 **go()**를 호출하고, **go()**가 스택의 맨 위에 들어간다. 변수 'x'와 'z'가 **go()** 스택 프레임에 들어있다.



3. **go()**에서 **crazy()**를 호출하고, **crazy()**가 이제 스택의 맨 위에 들어간다. 프레임에는 변수 'c'도 들어있다.



4. **crazy()**가 완료되고 그 스택 프레임이 스택에서 제거된다. 실행 제어는 **go()** 메소드로 되돌아가서 **crazy()**를 호출한 바로 다음 행으로 넘어간다.



객체 레퍼런스가 지역 변수인 경우는 어떻게 되나?

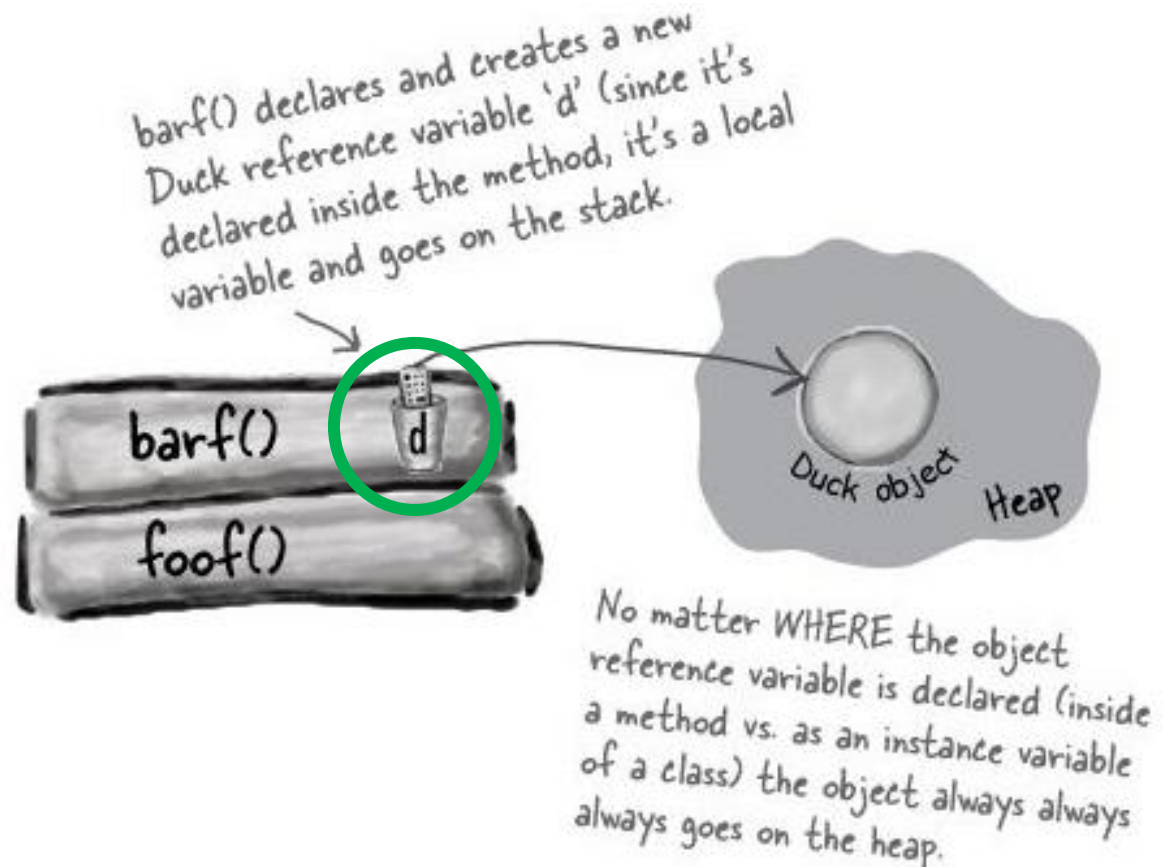
Non-primitive 변수는 객체 자체가 아니라 객체에 대한 레퍼런스를 저장하고 있다.
객체가 어디에서 살고 있는지는 이미 알고 있다. 그들이 선언되고 생성된 장소는 중요하지 않다.

만일 객체에 대한 레퍼런스가 지역변수이면 오로지 변수(레퍼런스/리모콘)만이 스택에 들어간다.

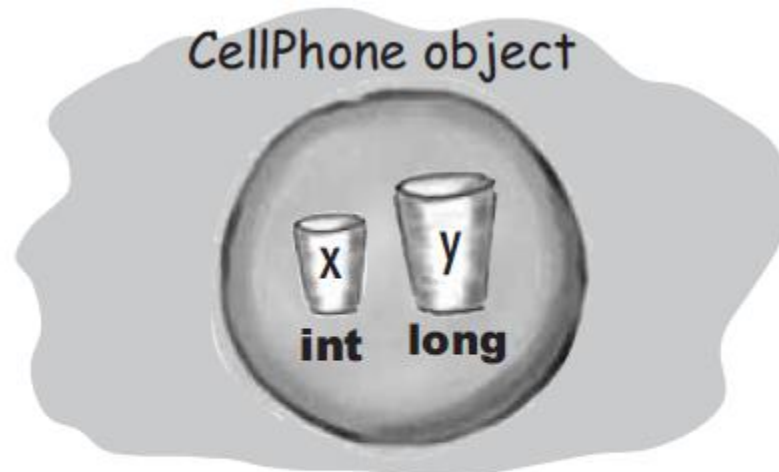
객체 자체는 여전히 힙에서 살고 있다!

```
public class StackRef {  
    public void foof() {  
        barf();  
    }  
  
    public void barf() {  
        Duck d = new Duck(24);  
    }  
}
```

method 안에서 선언!



지역변수는 스택에서 살지만 인스턴스 변수는 어디에서 살까?



두 개의 원시 인스턴스 변수를 가진 객체.

변수들은 객체 안에서 살고 있다



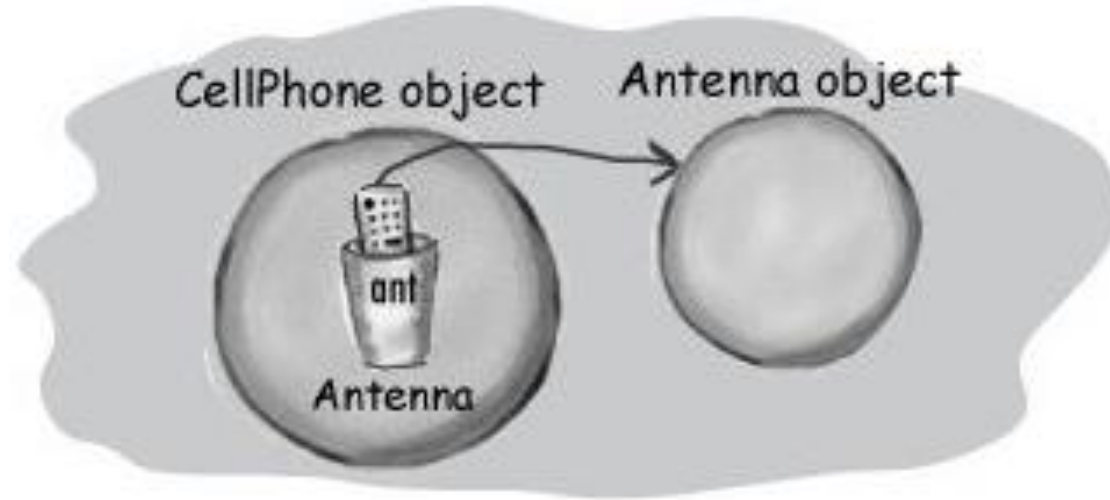
CellPhone 객체 안에 **Antenna**의 레퍼런스 변수에 대해서만 공간을 할당한다.

```
public class CellPhone {  
    private Antenna ant;  
}
```

⇒ 아직 실제 **Antenna** 객체는 없다

⇒ 선언만 하고 초기화하지 않은 경우이다

Antenna 객체의 공간은 언제 할당받는가?

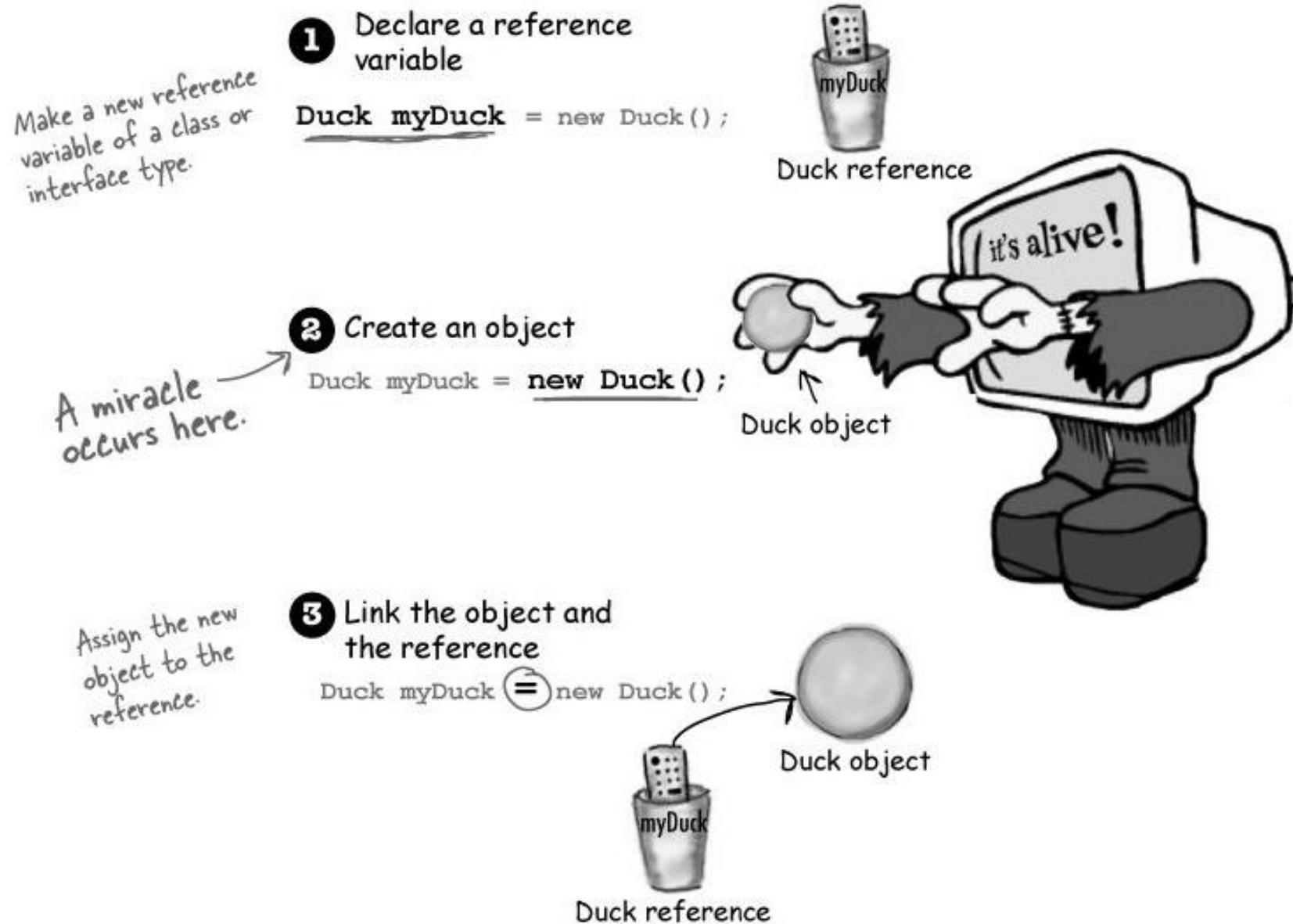


```
public class CellPhone {  
    private Antenna ant = new Antenna();  
}
```

Antenna 변수(**ant**)는 새로운 **Antenna** 객체를 할당받는다.

객체 생성의 기적

객체 선언, 생성, 대입의 3단계를 다시 살펴보자:



`Duck myDuck = new Duck ();` *It looks like we're calling a method named Duck(), because of the parentheses.*

⇒ **Duck()**이라는 이름의 메소드를 호출한 것인가?

No.

Duck 생성자를 호출하고 있는 것이다.

생성자 => **new**라는 키워드로 호출하는 것으로 객체를 인스턴스화할 때 실행하는 코드

생성자는 누가 만드나?

우리가 직접 만들지 않으면 컴파일러가 자동으로 만들어준다.

우리가 직접 만들지 않으면 컴파일러가 자동으로 만들어준다.

```
public Duck() {  
}
```

Where's the return type?
If this were a method,
you'd need a return type
between "public" and
"Duck()".

```
public Duck() {  
    // constructor code goes here  
}
```

Its name is the same as the
class name. That's mandatory.

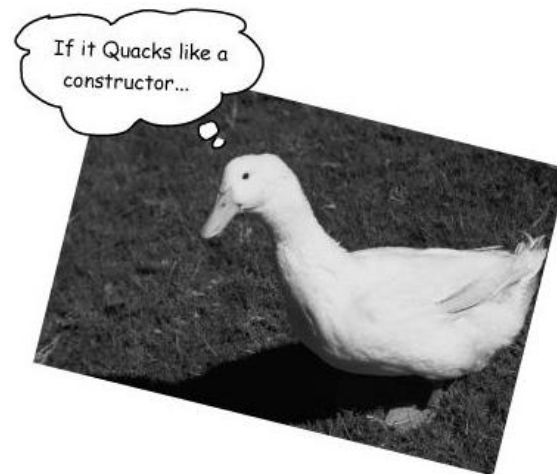
Duck 객체를 만들어보자

생성자의 가장 큰 특징은 객체가 레퍼런스에 할당되기 전에 실행된다는 점이다.

⇒ **new** 중간에 끼어들기 할 수 있는 기회를 준다.

```
public class Duck {  
  
    public Duck() {  
        System.out.println("Quack");  
    }  
}
```

← Constructor code.



```
public class UseADuck {  
  
    public static void main (String[] args) {  
        Duck d = new Duck() ;  
    }  
}
```

← This calls the Duck constructor.



새로운 Duck 객체의 상태를 초기화하는 방법 1

대부분의 경우 생성자를 사용하여 객체의 상태를 초기화한다.

```
public Duck() {  
    size = 34;  
}
```

Duck 클래스 개발자가 Duck 객체 크기를 **미리 알고 있다면** 위와 같이 해도 된다.

그러나, Duck을 **사용**하는 프로그래머가 Duck의 크기를 결정하게 하고 싶다면 어떻게 해야 하나?

즉 Duck 클래스에 **size**라는 **인스턴스 변수**를 두어서, 프로그래머가 Duck 클래스를 사용하여 새로운 Duck의 크기를 설정하게 하고 싶다.

⇒ Duck 클래스에 **setSize()**라는 세터 메소드를 추가하는 방법이 있다.

새로운 Duck 객체의 상태를 초기화하는 방법 2

Duck 사용자에게 다음 두 문장을 작성하도록 하는 방법이다:

(1) Duck을 생성하는 문장

(2) setSize() 메소드를 호출하는 문장

```
public class Duck {  
    int size; ← instance variable  
  
    public Duck() {  
        System.out.println("Quack"); ← constructor  
    }  
  
    public void setSize(int newSize) { ← setter method  
        size = newSize;  
    }  
}
```

```
public class UseADuck {  
  
    public static void main (String[] args) {  
        Duck d = new Duck();  
        d.setSize(42);  
    }  
}
```

이 시점에서 Duck은 살아있다. 그러나 사이즈가 없다! Duck 사용자가 객체 생성이 2 단계로 이루어지는 것을 알고 있다고 가정해야 한다: 생성자 호출 단계/세터 호출 단계

생성자를 이용한 Duck 상태 초기화

초기화 코드를 두기 위한 최적의 장소는 바로 생성자 안이다!!

```
public class Duck {  
    int size;
```

```
    public Duck(int duckSize) {  
        System.out.println("Quack");
```

```
        size = duckSize;
```

```
        System.out.println("size is " + size);
```

```
    }
```

```
}
```

Add an int parameter to the Duck constructor.

Use the argument value to set the size instance variable.

```
public class UseADuck {
```

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

```
        Duck d = new Duck(42);
```

```
    }
```

This time there's only one statement. We make the new Duck and set its size in one statement.

Pass a value to the constructor.

File Edit Window Help Honk

% java UseADuck

Quack

size is 42

Duck을 쉽게 만들 수 있게 해보자

프로그래머가 **Duck**의 크기를 얼마로 해야 할 지를 아는 경우와 모를 경우를 한꺼번에 처리할 수 있는 방법은?

아래 코드처럼 **Duck** 사용자가 **Duck** 객체를 만들 때 두 가지 옵션 중에 하나를 선택할 수 있게 하면 어떨까?

- 1) **Duck**의 크기를 생성자 인자로써 지정하거나 or
- 2) 크기를 지정하지 않고 디폴트 **Duck size**를 얻게 하는 방법

```
public class Duck {  
    int size;  
  
    public Duck(int newSize) {  
        if (newSize == 0) {  
            size = 27;  
        } else {  
            size = newSize;  
        }  
    }  
}
```

인자로 넘어온 값이 0이면 디폴트로 정해진 값(=27)으로 설정하고, 0이 아니면 인자로 넘어온 값을 설정한다.

```

public class Duck {
    int size;

    public Duck(int newSize) {
        if (newSize == 0) {
            size = 27;
        } else {
            size = newSize;
        }
    }
}

```

인자로 넘어온 값이 **0**이면 디폴트로 정해진 값(=27)으로 설정하고, **0**이 아니면 인자로 넘어온 값을 설정한다.

그러나 이 방법은 새로운 **Duck** 객체를 만드는 프로그래머가 "0"을 전달하는 것이 디폴트 **Duck size**를 얻을 수 있는 규칙임을 알아야 한다.

- ❶ 프로그래머가 그 사실을 모른다면 어찌되나?
- ❷ 혹은 정말로 **Zero-size Duck**을 원한다면?

주목해야 할 점은 "나는 실제로 **Zero-size Duck**을 원한다"와 "나는 **Duck** 사이즈가 무엇이던 간에 나에게 디폴트 값을 주도록 **0** 값을 보낸다" 사이의 차이를 구분할 수 없다는 것이다.

새로운 Duck을 만들기 위한 두 가지 방법을 원할 때

```
public class Duck2 {  
    int size;  
  
    public Duck2() {  
        // supply default size  
        size = 27;  
    }  
  
    public Duck2(int duckSize) {  
        // use duckSize parameter  
        size = duckSize;  
    }  
}
```

생성자를 2개 만든다.

크기를 알고 있을 때:

Duck2 d = new Duck2(15);

크기를 모를 때:

Duck2 d2 = new Duck2();

아규먼트가 없는 생성자는 컴파일러에서 항상 자동으로 만들어 주지 않나?

그렇지 않다!

프로그래머가 생성자를 하나라도 만들었다면 컴파일러는 더 이상 자동으로 생성자를 만들어 주지 않는다.

즉 컴파일러는 생성자가 전혀 정의되지 않았을 경우에만 자동으로 하나를 만들어 준다.

따라서 인자가 있는 생성자를 정의했다면 인자가 없는 생성자도 반드시 정의해줘야 한다.


클래스에 생성자가 두 개 이상 있는 경우 각 생성자의 아규먼트 리스트는 모두 달라야 한다.

⇒ 오버로딩된 생성자



생성자 오버로딩을 이용하면 한 클래스에 두 개 이상의 생성자를 만들 수 있다

이때 각 생성자의 아규먼트 리스트가 서로 다르지 않으면 컴파일이 되지 않는다!



```
public class Mushroom {  
    public Mushroom(int size) { }  
    public Mushroom( ) { }  
    public Mushroom(boolean isMagic) { }  
    {  
        public Mushroom(boolean isMagic, int size) { }  
        public Mushroom(int size, boolean isMagic) { }  
    }  
}
```

when you know the size, but you don't know if it's magic

when you don't know anything

when you know if it's magic or not, but don't know the size

these two have the same args, but in different order, so it's OK

when you know whether or not it's magic, AND you know the size as well

실습과제 9-1

new Duck() 선언문과 **Duck**의 인스턴스를 만들 때 실행되는 생성자를 연결해 보시오.

```
public class TestDuck {  
    public static void main(String[] args){  
        int weight = 8;  
        float density = 2.3F;  
        String name = "Donald";  
        long[] feathers = {1,2,3,4,5,6};  
        boolean canFly = true;  
        int airspeed = 22;  
  
        Duck[] d = new Duck[7];  
  
        d[0] = new Duck();  
  
        d[1] = new Duck(density, weight);  
  
        d[2] = new Duck(name, feathers);  
  
        d[3] = new Duck(canFly);  
  
        d[4] = new Duck(3.3F, airspeed);  
  
        d[5] = new Duck(false);  
  
        d[6] = new Duck(airspeed, density);  
    }  
}
```

```
class Duck {  
    int pounds = 6;  
    float floatability = 2.1F;  
    String name = "Generic";  
    long[] feathers = {1,2,3,4,5,6,7};  
    boolean canFly = true;  
    int maxSpeed = 25;  
  
    public Duck() {  
        System.out.println("type 1 duck");  
    }  
  
    public Duck(boolean fly) {  
        canFly = fly;  
        System.out.println("type 2 duck");  
    }  
  
    public Duck(String n, long[] f) {  
        name = n;  
        feathers = f;  
        System.out.println("type 3 duck");  
    }  
  
    public Duck(int w, float f) {  
        pounds = w;  
        floatability = f;  
        System.out.println("type 4 duck");  
    }  
  
    public Duck(float density, int max) {  
        floatability = density;  
        maxSpeed = max;  
        System.out.println("type 5 duck");  
    }  
}
```

리뷰: 생성자에 대해 반드시 알아야 할 네 가지

1. A constructor is the code that runs when somebody says **new** on a class type

```
Duck d = new Duck();
```

2. A constructor must have the **same name** as the class, and no return type

```
public Duck(int size) { }
```

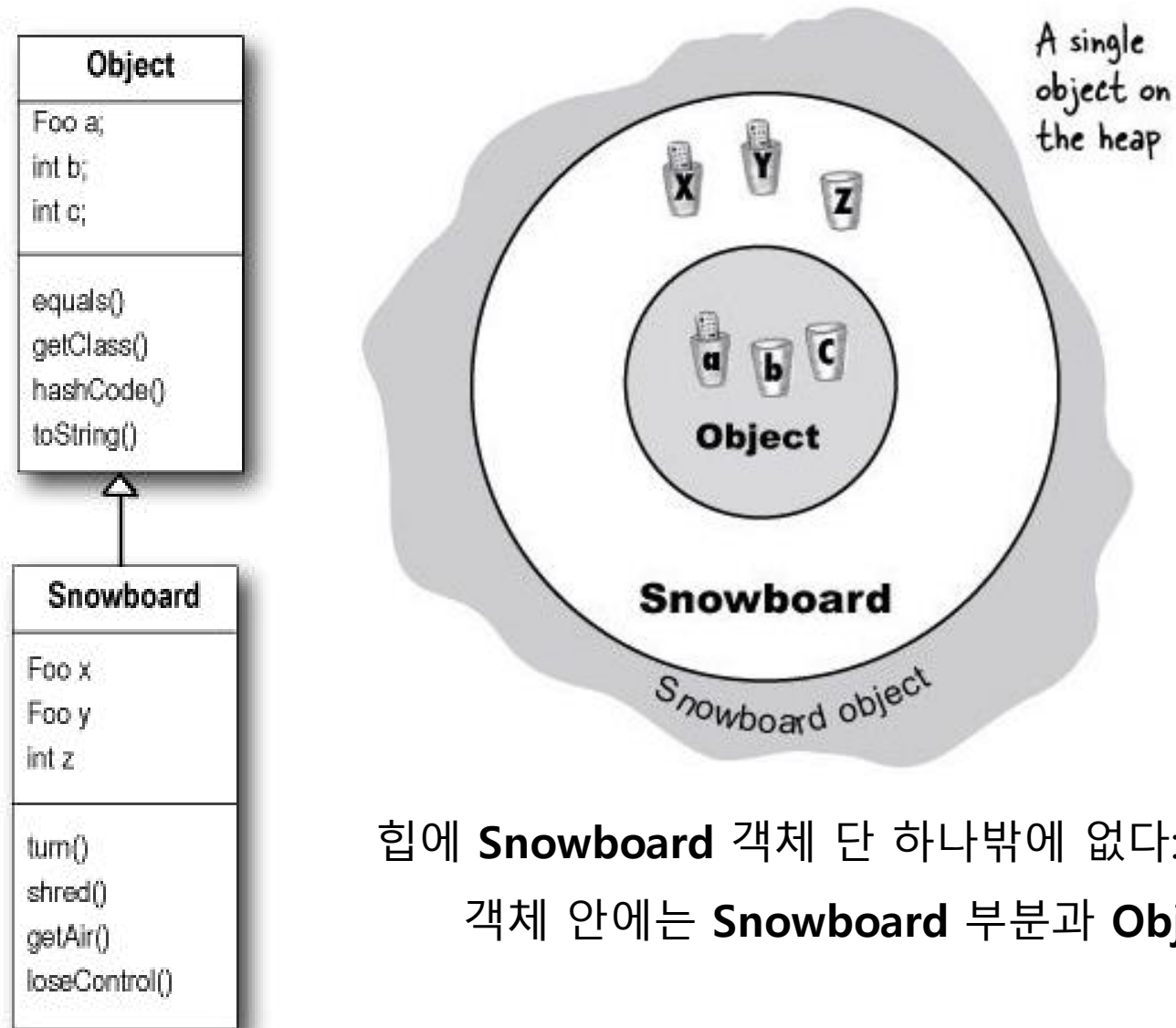
3. If you don't put a constructor in your class, the compiler puts in a default constructor. The default constructor is always a **no-arg** constructor.

```
public Duck() { }
```

4. You can have more than one constructor in your class, as long as the argument lists are **different**. Having more than one constructor in a class means you have overloaded constructors.

```
public Duck() { }  
public Duck(int size) { }  
public Duck(String name) { }  
public Duck(String name, int size) { }
```

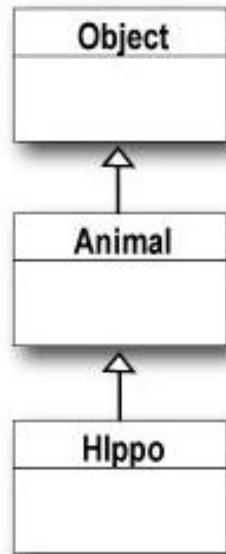
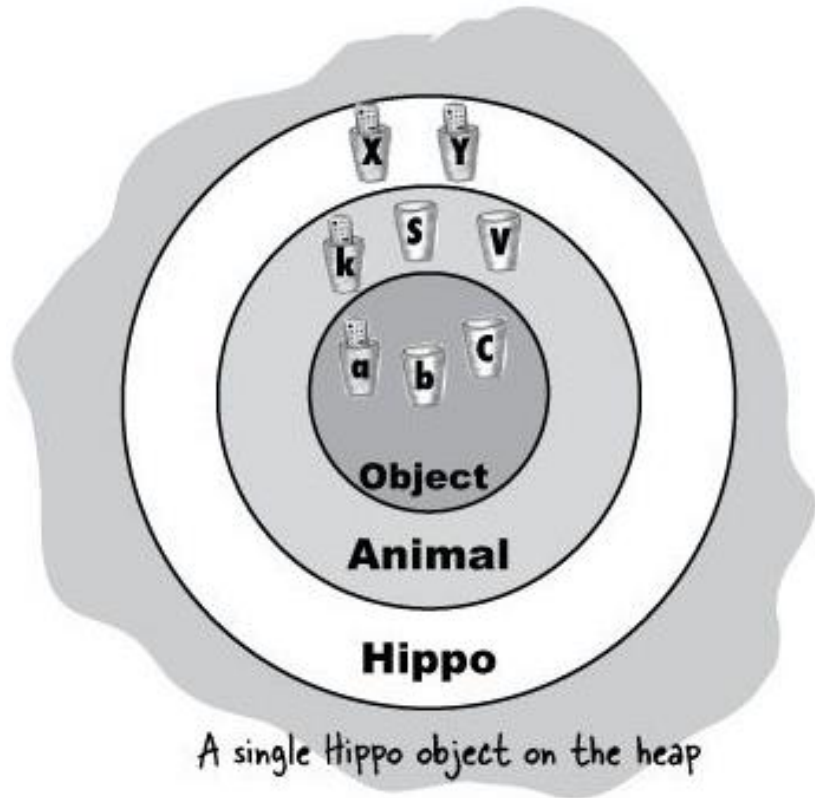
Wait a minute... we never DID talk about superclasses and inheritance and how that all fits in with constructors



힙에 **Snowboard** 객체 단 하나밖에 없다:

객체 안에는 **Snowboard** 부분과 **Object** 부분이 모두 들어있다.

객체의 일생에서 슈퍼클래스 생성자의 역할



새로운 객체를 만들 때 그 객체의 상속 트리에 있는 모든 생성자가 실행되어야 한다!

새로 생성된 **Hippo** 객체는 **Animal** 객체이기도 하고 **Object** 객체이기도 하다.

Hippo를 만들고 싶다면 그 안에 **Animal**과 **Object** 부분도 만들어야 한다.

이러한 모든 과정은 **생성자 연쇄과정**(Constructor Chaining)에서 자동으로 이루어진다.

실습과제 9-2 What's the real output?

```
public class Animal {  
    public Animal() {  
        System.out.println("Making an Animal");  
    }  
}
```

```
public class Hippo extends Animal {  
    public Hippo() {  
        System.out.println("Making a Hippo");  
    }  
}
```

```
public class TestHippo {  
    public static void main (String[] args) {  
        System.out.println("Starting...");  
        Hippo h = new Hippo();  
    }  
}
```

A

```
File Edit Window Help Swear  
% java TestHippo  
Starting...  
Making an Animal  
Making a Hippo
```

B

```
File Edit Window Help Swear  
% java TestHippo  
Starting...  
Making a Hippo  
Making an Animal
```

Hippo()

Animal()
Hippo()

Object()
Animal()
Hippo()

Animal()
Hippo()

슈퍼클래스 생성자를 어떻게 호출할까?

```
public class Duck extends Animal {  
    int size;
```

```
    public Duck(int newSize) {  
        Animal(); ← NO! This is not legal!  
        size = newSize;  
    }  
}
```

BAD! →

```
public class Duck extends Animal {  
    int size;
```

```
    public Duck(int newSize) {  
        super(); ← you just say super()  
        size = newSize;  
    }  
}
```

Can the child exist before the parents?



The superclass parts of an object have to be fully-formed (completely built) before the subclass parts can be constructed.

super()에 대한 호출은 반드시 각 생성자의 첫 번째 문장이어야 한다!

Possible constructors for class Boop

```
✓ public Boop() {  
    super();  
}
```

```
✓ public Boop(int i) {  
    super();  
    size = i;  
}
```

These are OK because the programmer explicitly coded the call to super(), as the first statement.

```
✓ public Boop() {  
}
```

```
✓ public Boop(int i) {  
    size = i;  
}
```

These are OK because the compiler will put a call to super() in as the first statement.

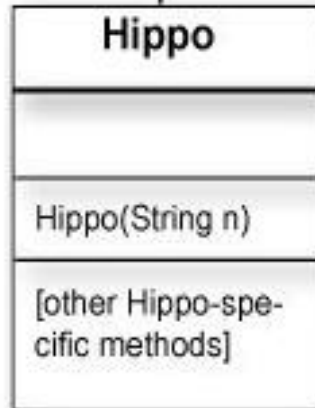
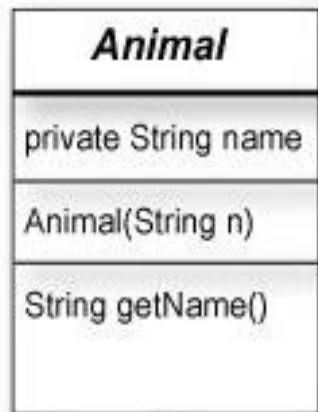
```
⊗ public Boop(int i) {  
    size = i;  
    super();  
}
```

BAD!! This won't compile! You can't explicitly put the call to super() below anything else

인자를 가지는 슈퍼클래스 생성자

슈퍼클래스 생성자가 아규먼트를 가진다면 어떻게 될까?

super()를 호출할 때 뭔가를 전달할 수 있을까? ... 물론 그렇다.



The Animal part of me needs to know my name, so I take a name in my own Hippo constructor, then pass the name to super()



```

public abstract class Animal {
    private String name;
}

public String getName() {
    return name;
}

public Animal(String theName) {
    name = theName;
}

```

← All animals (including subclasses) have a name

← A getter method that Hippo inherits

← The constructor that takes the name and assigns it the name instance variable

```

public class Hippo extends Animal {

    public Hippo(String name) {
        super(name);
    }
}

```

← Hippo constructor takes a name

← it sends the name up the Stack to the Animal constructor

```

public class MakeHippo {
    public static void main(String[] args) {
        Hippo h = new Hippo("Buffy");
        System.out.println(h.getName());
    }
}

```

← Make a Hippo, passing the name "Buffy" to the Hippo constructor. Then call the Hippo's inherited getName()

```

File Edit Window Help Hide
%java MakeHippo
Buffy

```

다른 생성자로부터 오버로딩된 생성자를 호출

```
import java.awt.Color;
class Mini extends Car {
```

```
    Color color;
```

```
    public Mini() {
        this(Color.RED);
    }
```

The no-arg constructor supplies a default Color and calls the overloaded Real Constructor (the one that calls super()).

```
    public Mini(Color c) {
        super("Mini");
        color = c;
        // more initialization
    }
```

This is The Real Constructor that does The Real Work of initializing the object (including the call to super())

```
    public Mini(int size) {
        this(Color.RED);
        super(size);
    }
```

Won't work!! Can't have super() and this() in the same constructor, because they each must be the first statement!

```
}
```

```
File Edit Window Help Drive
javac Mini.java
Mini.java:16: call to super must
be first statement in constructor
    super();
    ^
```

같은 클래스에서 오버로딩된 생성자로부터 또다른 생성자를 호출하려면 **this()**를 이용하면 된다.

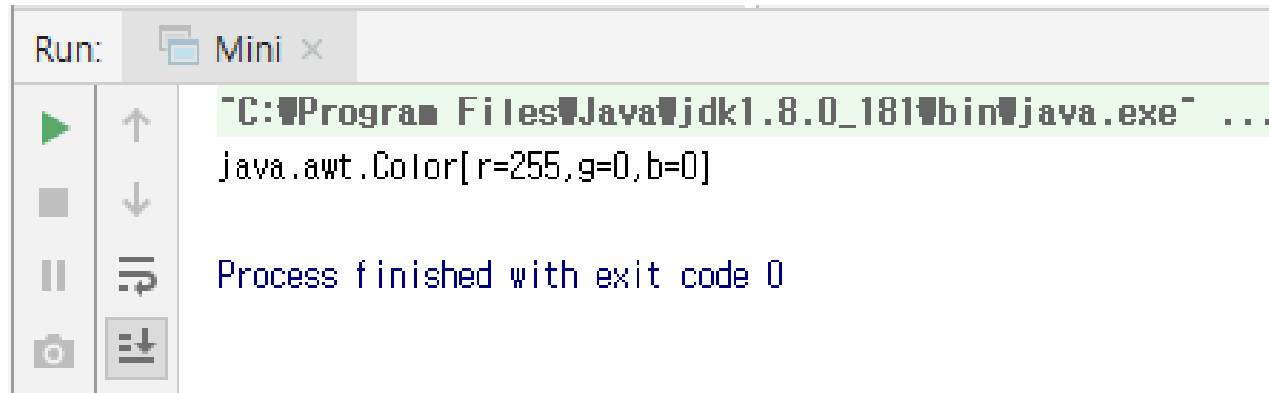
this()는 현재 객체에 대한 레퍼런스라고 간주하면 된다.

this()는 생성자 안에서만 쓸 수 있으며 반드시 첫 번째 문장으로 와야 한다.

모든 생성자는 **super()** 또는 **this()**를 쓸 수 있다.

그러나 동시에 둘을 함께 쓸 수는 없다.

실습과제 9-3



위의 출력이 나오도록 바로 앞 페이지의 Mini 클래스를 완성하시오.

실습과제 9-4

Some of the constructors in the SonOfBoo class will not compile. See if you can recognize which constructors are **not legal**. Match the compiler errors with the SonOfBoo constructors that caused them, by drawing a line from the compiler error to the "bad" constructor.

```
public class Boo {  
    public Boo(int i) { }  
    public Boo(String s) { }  
    public Boo(String s, int i) { }  
}
```

```
class SonOfBoo extends Boo {  
  
    public SonOfBoo() {  
        super("boo");  
    }  
  
    public SonOfBoo(int i) {  
        super("Fred");  
    }  
  
    public SonOfBoo(String s) {  
        super(42);  
    }  
  
    public SonOfBoo(int i, String s) {  
    }  
  
    public SonOfBoo(String a, String b, String c) {  
        super(a,b);  
    }  
  
    public SonOfBoo(int i, int j) {  
        super("man", j);  
    }  
  
    public SonOfBoo(int i, int x, int y) {  
        super(i, "star");  
    }  
}
```

```
File Edit Window Help  
&javac SonOfBoo.java  
cannot resolve symbol  
symbol : constructor Boo  
(java.lang.String,java.la  
ng.String)
```

```
File Edit Window Help Yadayadayada  
&javac SonOfBoo.java  
cannot resolve symbol  
symbol : constructor Boo  
(int,java.lang.String)
```

```
File Edit Window Help ImNotListening  
&javac SonOfBoo.java  
cannot resolve symbol  
symbol:constructor Boo()
```

객체는 얼마나 오래 살아있나?

객체의 일생은 그 객체를 참조하는 레퍼런스의 일생에 의해 좌우된다.

- ✓ 레퍼런스가 살아있으면 그 레퍼런스가 참조하는 객체도 살아있고 그 레퍼런스가 죽으면 객체도 함께 죽는다.

그렇다면 변수는 얼마나 오랫동안 살아 있나?

지역변수는 해당 변수를 선언한
메소드 안에서만 살아 있다!

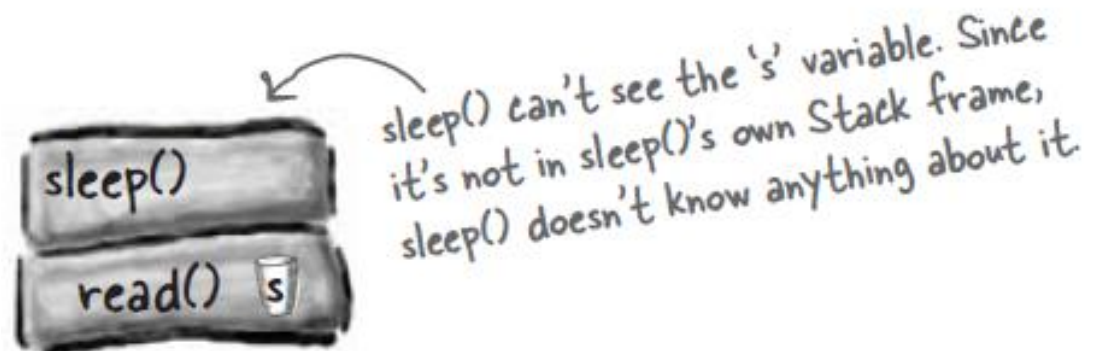
```
public class TestLifeOne {
```

```
    public void read() {  
        int s = 42;  
        sleep();  
    }
```

's' is scoped to the read() method, so it can't be used anywhere else

```
    public void sleep() {  
        s = 7;  
    }  
}
```

BAD!! Not legal to use 's' here!



1. 지역 변수는 그 변수를 선언한 **메소드 안에서만** 살 수 있다.

```
public void read() {  
    int s = 42;  
    // 's' can be used only  
    // within this method.  
    // When this method ends,  
    // 's' disappears completely.  
}
```

2. 인스턴스 변수는 **객체가 살아있는 동안** 계속 살 수 있다. 즉 객체가 아직 살아 있다면 변수도 역시 살아있다.

```
public class Life {  
    int size;  
  
    public void setSize(int s) {  
        size = s;  
        // 's' disappears at the  
        // end of this method,  
        // but 'size' can be used  
        // anywhere in the class  
    }  
}
```

지역 변수의 삶(Life)과 활동범위(Scope)의 차이

```
public void doStuff() {  
    boolean b = true;  
    go(4);  
}  
  
public void go(int x) {  
    int z = x + 24;  
    crazy();  
    // imagine more code here  
}  
  
public void crazy() {  
    char c = 'a';  
}
```

삶 (Life)

지역 변수는 그 스택 프레임이 스택에 들어있는 한 계속 살아있다. 즉, 메소드가 종료될 때까지.

활동범위 (Scope)

지역 변수의 활동범위는 그 변수를 선언한 메소드 내부로 제한된다.

자신의 메소드가 또 다른 메소드를 호출한 경우, 변수는 살아 있지만, 자신의 메소드가 다시 시작될 때까지 활동범위에 속하지 않는다.

“변수는 활동범위에 속해 있을 때만 사용할 수 있다.”



레퍼런스 변수인 경우는?

변수의 삶이 객체의 삶에 어떻게 영향을 미칠까?

객체는 자신을 참조하는 레퍼런스가 하나라도 존재하는 한 살아있다.

아무도 참조하지 않는 객체는 아무런 의미가 없다.

⇒ 아무도 도달할 수 없는 객체는 가비지 콜렉터의 대상이 된다.

Three ways to get rid of an object's reference

객체는 마지막 레퍼런스가 사라지면 GC에 노출된다.

Three ways to get rid of an object's reference:

- ① The reference goes out of scope, permanently

```
void go() {  
    Life z = new Life();  
}
```

레퍼런스 'z'는 메소드가 종료되면 사라진다!

- ② The reference is assigned another object

```
Life z = new Life();  
z = new Life();
```

the first object is abandoned when z is 'reprogrammed' to a new object.

- ③ The reference is explicitly set to null

```
Life z = new Life();  
z = null;
```

the first object is abandoned when z is 'deprogrammed'.

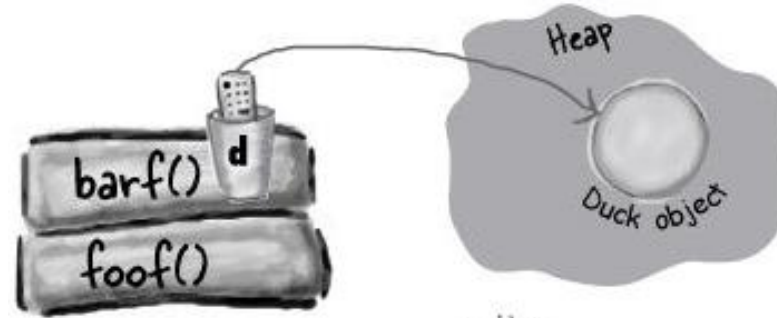
Object-killer #1 Reference goes out of scope, permanently.

```
public class StackRef {  
    public void foof() {  
        barf();  
    }  
  
    public void barf() {  
        Duck d = new Duck();  
    }  
}
```

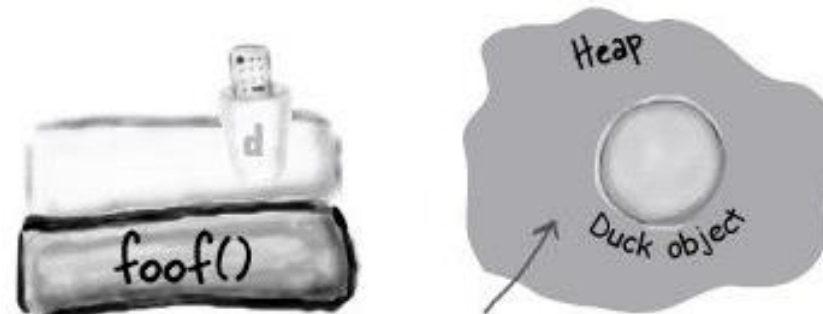
1. **foof()**가 스택에 놓인다.



2. **barf()**가 스택에 놓인다.

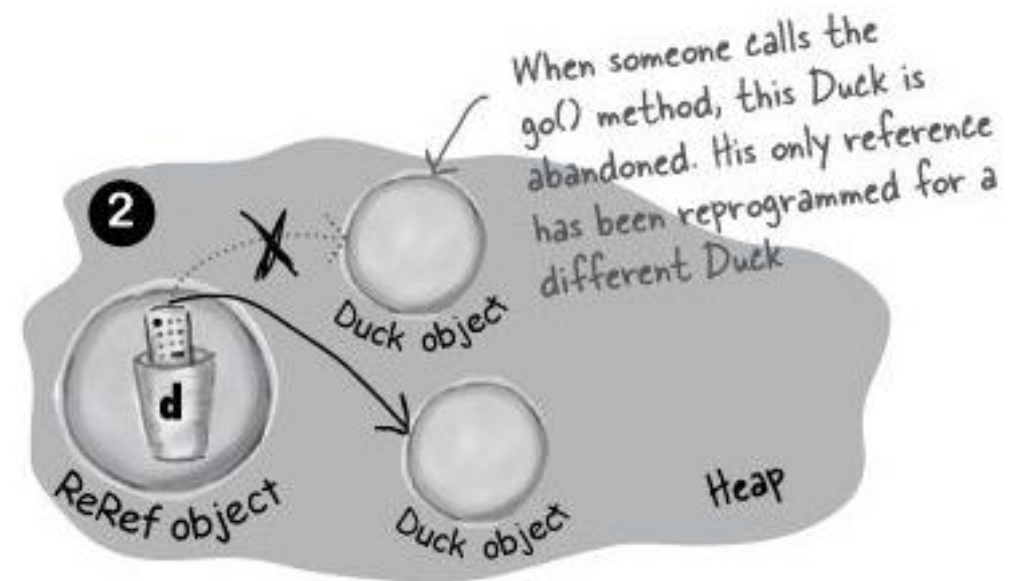
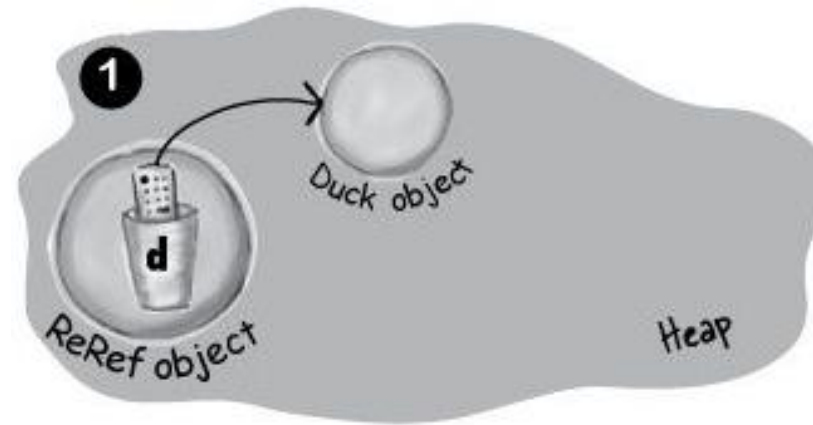


3. **barf()**가 종료되어 스택에서 제거된다.



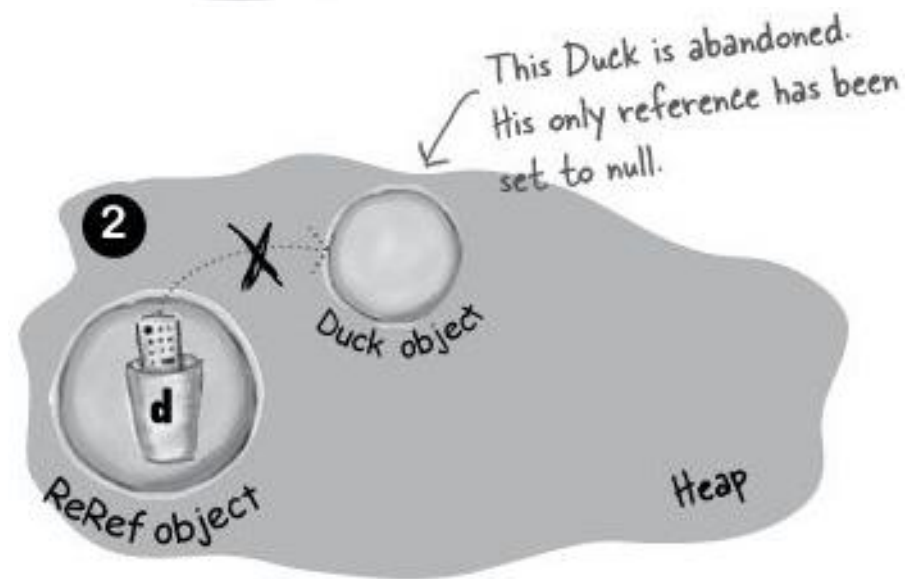
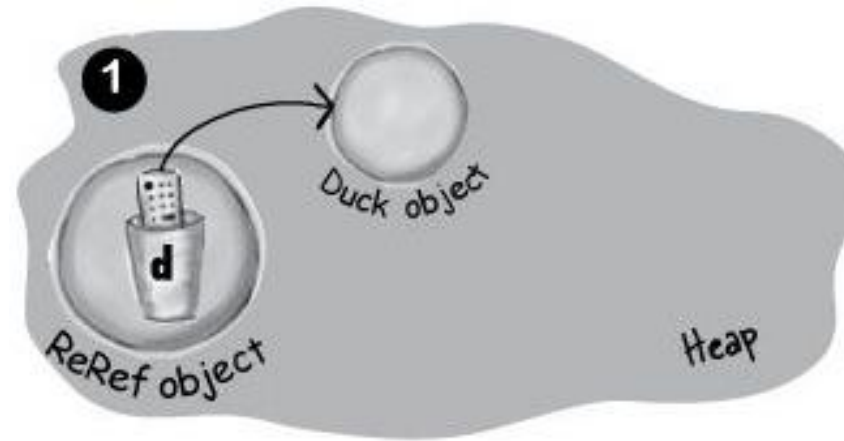
Object-killer #2 Assign the reference to another object

```
public class ReRef {  
  
    Duck d = new Duck();  
  
    public void go() {  
        d = new Duck();  
    }  
}
```



Object-killer #3 Explicitly set the reference to null

```
public class ReRef {  
  
    Duck d = new Duck();  
  
    public void go() {  
        d = null;  
    }  
}
```



실습과제 9-5 BE the Garbage Collector

Which of the lines of code on the right, if added to the class on the left at point A, would cause exactly one additional object to be eligible for the Garbage Collector?

```
public class GC {  
    public static GC doStuff() {  
        GC newGC = new GC();  
        doStuff2(newGC);  
        return newGC;  
    }  
  
    public static void main(String [] args) {  
        GC gc1;  
        GC gc2 = new GC();  
        GC gc3 = new GC();  
        GC gc4 = gc3;  
        gc1 = doStuff();  
  
        A  
  
        // call more methods  
    }  
  
    public static void doStuff2(GC copyGC) {  
        GC localGC = copyGC;  
    }  
}
```

- 1 copyGC = null;
- 2 gc2 = null;
- 3 newGC = gc3;
- 4 gc1 = null;
- 5 newGC = null;
- 6 gc4 = null;
- 7 gc3 = gc2;
- 8 gc1 = gc4;
- 9 gc3 = null;

Project 1: Online Specialty Pizza Shop

ArrayList, Constructor, ...

사이버캠퍼스 '과제' 찜조!!

