JAVA PROGRAMMING

2023/10/20 16:00 PM

CONTENTS

CONDITIONAL STATEMENTS

- If-Else
- Switch Case
- Continue and Break
- While Loop
- For Loop
- For Each
- Nested Loop

If-Else

```
if (condition) {
        // block of code to be executed if the condition is true
}
```

- **`if`**: specify a block of code to be executed, if a specified condition is true.
  - `if` is in lowercase letters. Uppercase letters (If or IF) will generate an error.

```
int x = 20;
int y = 18;
if (x > y) {
  System.out.println("x is greater than y");
}
```

If-Else (cont.)

```
if (condition) {
        // block of code to be executed if the condition is true
} else {
        // block of code to be executed if the condition is false
}
```

- **`else`**: specify a block of code to be executed, if the same condition is false.
  - When using `else`, there is no condition block.

```
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
```

If-Else (cont.)

```
if (condition1) {
        // block of code to be executed if condition1 is true
} else if (condition2) {
        // block of code to be executed if the condition1 is false and condition2 is true
} else {
        // block of code to be executed if the condition1 is false and condition2 is false
}
```

- **`else if`**: specify a new condition to test, if the first condition is false.
  - cf) Python - `elif`

If-Else – Example

```
int time = 22;
if (time < 10) {
  System.out.println("Good morning.");
} else if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
```

- In the example above:
  - `time` (22) is greater than 10, so the first condition is false.
  - The next condition, in the `else if` statement, is also false.
  - So we move on to the `else` condition since *condition1* and *condition2* is both false.

    ⇒ print to the screen "Good evening"

Switch Case

```
switch(expression) {
        case x:
            // code block
            break;
        case y:
            // code block
            break;
        default:
            // code block
}
```

- Instead of writing many `if-else` statements, you can use the `switch` statement.
- The switch statement specifies many alternative blocks of code to be executed
- It selects one of many code blocks to be executed.

Switch Case (cont.)

- The `switch` expression is evaluated once.
- The value of the expression is compared with the values of each `case`.
- If there is a match, the associated block of code is executed.
- The `break` and `default` keywords are *optional*. (will be described later)

Switch Case – Example

- The example uses the weekday number to calculate the weekday name.

- `day` = 4 ⇒ prints “Thursday” to screen.

```
int day = 4;
switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
    break;
  case 3:
    System.out.println("Wednesday");
    break;
  case 4:
    System.out.println("Thursday");
    break;
  case 5:
    System.out.println("Friday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
  case 7:
    System.out.println("Sunday");
    break;
}
```

Switch Case – default

- The `default` keyword specifies some code to run if there is no case match.

- Note that if the `default` statement is used as the last statement in a switch block, it does not need a `break`.

```
int day = 4;
switch (day) {
  case 6:
    System.out.println("Today is Saturday");
    break;
  case 7:
    System.out.println("Today is Sunday");
    break;
  default:
    System.out.println("Looking forward to the Weekend");
}
// Outputs "Looking forward to the Weekend"
```

Continue and Break

- `Break`
  - When Java reaches a `break` keyword, it breaks out of the **switch block**.
  - The break statement can also be used to jump out of a **loop**.
  - This will stop the execution of more code and case testing inside the block.
  - When a match is found, and the job is done, it's time for a break.

    ⇒ There is no need for more testing.
  - A `break` can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

Continue and Break – Break Example 1

```java
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  System.out.println(i);
}
```

```
0
1
2
3
```

- The for loop running when `i == 4`, code **breaks** the loop

  → does not run the print method → to **jump out** of the **loop**.

Continue and Break – Break Example 2

```
int i = 0;
while (i < 10) {
  System.out.println(i);
  i++;
  if (i == 4) {
    break;
  }
}
```

```
0
1
2
3
```

- You can also use `break` in `while` loops.
  - In this example, when the variable `i` becomes 4, the `while` loop breaks.

Continue and Break (cont.) – Continue Example 1

- Continue
  - Breaks *one* iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  System.out.println(i);
}
```

```
0
1
2
3
5
6
7
8
9
```

  - This example the for loop will break one iteration when the value of i reaches 4.
  - Continues with the next iteration, i = 5, and prints number until 9.

Continue and Break – Continue Example 2

```java
int i = 0;
while (i < 10) {
  if (i == 4) {
    i++;
    continue;
  }
  System.out.println(i);
  i++;
}
```

```
0
1
2
3
5
6
7
8
9
```

- You can also use `continue` in `while` loops.
- In this example, when the variable `i` becomes 4, the inner loop *continues* and does not print on the console.

While Loop

```
while (condition) {
    // code block to be executed
}
```

- The `while` loop loops through a block of code as long as a specified condition is `true`.
- Loops can execute a block of code as long as a specified condition is reached.
- Loops are handy because they save time, reduce errors, and they make code more readable.
- Do not forget to increase the variable used in the condition, otherwise the loop will never end.

While Loop – Example

```
int i = 0;
while (i < 5) {
  System.out.println(i);
  i++;
}
```

```
0
1
2
3
4
```

- In the example, the code in the loop will run, over and over again, as long as a variable (`i`) is less than 5.

For Loop

```
for (statement 1; statement 2; statement 3) {
    // code block to be executed
}
```

- When you know exactly how many times you want to loop through a block of code, use the **`for` loop** instead of a `while` loop.

- *Statement 1* : executed (one time) before the execution of the code block.
- *Statement 2* : defines the condition for executing the code block.
- *Statement 3* : executed (every time) after the code block has been executed.

For Loop – Example 1

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

- The example will print the numbers 0 to 4. (0 1 2 3 4)

① Statement 1 sets a variable before the loop starts (`int i = 0`).

② Statement 2 defines the condition for the loop to run (`i` must be less than 5).
If the condition is true, the loop will start over again, if it is false, the loop will end.

③ Statement 3 increases a value (`i++`) each time the code block in the loop has been executed.

For Loop – Example 2

```
for (int i = 0; i <= 10; i = i + 2) {
  System.out.println(i);
}
```

- This example will only print even values between 0 and 10: (0 2 4 6 8 10)

① Statement 1 sets a variable before the loop starts (`int i = 0`).

② Statement 2 defines the condition for the loop to run (`i` must be less or same as 10). If the condition is true, the loop will start over again, if it is false, the loop will end.

③ Statement 3 increases a value (`i+2`) each time the code block in the loop has been executed.

For Each Loop

```
for (type variableName : arrayName) {
  // code block to be executed
}
```

- For-each loop is used exclusively to loop through elements in an array.

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

```
Volvo
BMW
Ford
Mazda
```

  - The following example outputs all elements in the `cars` array.

Nested Loop

```
// Outer Loop
for (statement 1; statement 2; statement 3) {
        // code block to be executed

      // Inner Loop
      for (statement 1; statement 2; statement 3) {
            // code block to be executed
      }
}
```

- It is also possible to place a loop inside another loop.
- The "inner loop" will be executed one time for each iteration of the "outer loop”.

Nested Loop – Example

```java
// Outer loop
for (int i = 1; i <= 2; i++) {
  System.out.println("Outer: " + i); // Executes 2 times

  // Inner loop
  for (int j = 1; j <= 3; j++) {
    System.out.println(" Inner: " + j); // Executes 6 times (2 * 3)
  }
}
```

```
Outer: 1
 Inner: 1
 Inner: 2
 Inner: 3
Outer: 2
 Inner: 1
 Inner: 2
 Inner: 3
```

- In this example:
  - Outer loop runs 2 times.
  - Inner loop runs 3 times per one outer loop execution.

JAVA ARRAYS

- Java Arrays
- Java Arrays Loop
- Java Array Types

Java Arrays

```
type[] arrayName;

type[] arrayName = new type[arraySize];
```

- An object of homogeneous collection of variables.
- immutable
- access an array element by referring to the index number. (starts from [0])
- can handle a large amount of data by iteration. (`for` loop)
- `arrayName.length`: returns the number of size of the array in int type

Java Arrays (cont.)

```
int[] intArr = new int[5];
int[] myNum = {10, 20, 30, 40};
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

- Declaration and initialization can be done together in one line of code.
- Default value of array:

배열 변수형	초기 값
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
boolean	false
char	'₩u0000'
object reference	null

Java Arrays Loop

- Loop Through an Array
  - Loop through the array elements with the `for` loop
  - Use the `length` property to specify how many times the loop should run.
  - Example:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.length; i++) {
  System.out.println(cars[i]);
}
```

```
Volvo
BMW
Ford
Mazda
```

Java Arrays Loop (cont.)

- Loop Through an Array with For-Each

```
for (type variable : arrayname) {
  ...
}
```

  - Use "**for-each**" loop, which is used exclusively to loop through elements in arrays:
  - Example:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

```
Volvo
BMW
Ford
Mazda
```

  - **for each** String element (called **i** - as in **i**ndex) in **cars**, print out the value of **i**.

Java Array Types

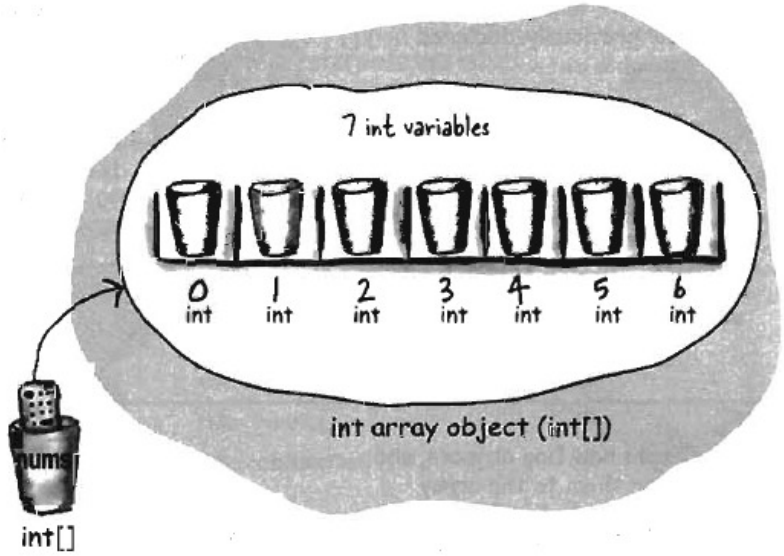
- Array of Primitive Types

  ① Declare an `int array` variable.

  ② Create an `int array` with a `length`, and assign it to the previously declared variable.

  ③ Give each element in the array an `int` value.

  ※ elements in an array are just variables!



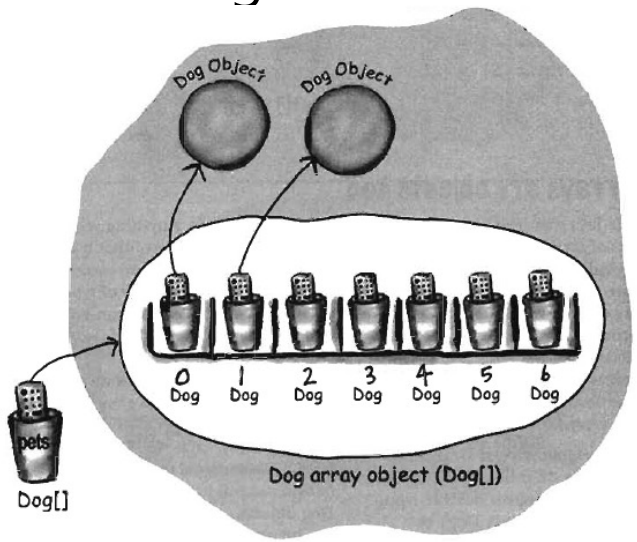
Java Array Types (cont.)

- Array of Objects

  ① Declare an `array` variable.

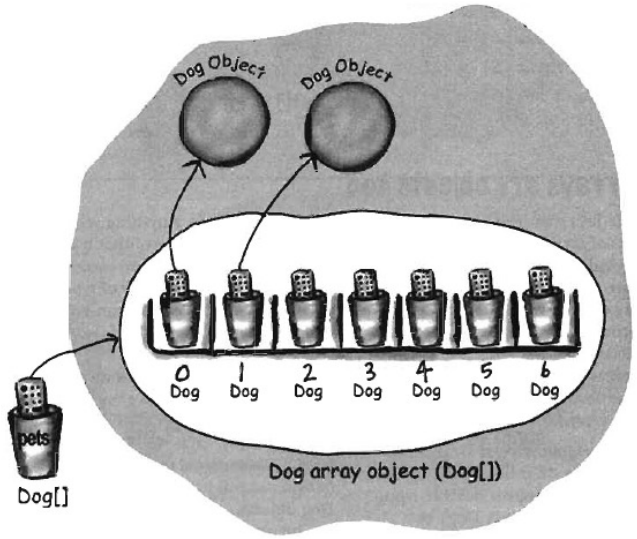
  ② Create an object `array` with a `length`, and assign it to the previously declared variable.

  ③ Create new objects, and assign them to the array elements.



Java Array Types (cont.)

- Accessing an object in an array
  - Typically use dot notation (.) to access object's instance variable and methods.
  - If an object is in an array, there is no actual reference variable.
    ⇒ Need to use **array notation**.




```
Dog[] myDogs = new Dog[3];
myDogs[0] = new Dog();
myDogs[0].name = "Fido";
myDogs[0].bark();
```

Java Array Types – Test

< 애완동물 키우기 프로그램 >

1. 클래스 구조

- 애완동물 이름 (String)
- 애완동물 개월 수 (int)
- 애완동물 종류 (String)
- 애완동물이 배고픈지 (boolean)

2. 메뉴 (애완동물은 최대 5마리까지 등록 가능)

1) Pet 등록하기

- 등록된 애완동물이 5마리를 초과할 경우,
  "더 이상 기를 수 없습니다!"를 출력
- 이름, 개월 수, 종류 입력 받기
- 배고픔은 기본으로 true (배고픈 상태가 기본 상태)

2) Pet 정보 보기

- 모든 동물의 정보를 출력
- 현재 어떤 동물이 배고픈지 이름 출력
- 평균 개월 수 출력

3) Pet 밥 주기

- 누구에게 밥을 주겠습니까? (이름 입력)
- 해당 이름을 가진 애완동물의 배고픔 상태 false로 변경

4) 종료하기

- "**, ㅁㅁ, ㅇㅇ, ㅎㅎ (애완동물들 이름)이가 슬퍼합니다.
  그래도 가실거예요?"

  출력 후 "예"를 입력 받으면 프로그램 종료

JAVA OOP

- Java OOP
- Java Class
- Java Objects
- Java Methods
- Java Constructors
- Java Modifiers
- Garbage Collection

Java OOP

- *Procedural programming* is about writing procedures or methods that perform operations on the data.

- **Object-oriented programming** is about creating objects that contain both data and methods.

- The "Don't Repeat Yourself" (*DRY*) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

Java OOP (cont.)

- Object-oriented programming has several advantages over procedural programming:
  - OOP is faster and easier to execute.
  - OOP provides a clear structure for the programs.
  - OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug.
  - OOP makes it possible to create full reusable applications with less code and shorter development time.

Java OOP (cont.)

- Java is an object-oriented programming language.

- Everything in Java is associated with **classes** and **objects**, along with its **attributes** and **methods**.

- For example: in real life, a car is an **object**.
  - The car has **attributes**, such as weight and color.
  - The car has **methods**, such as drive and brake.

Java Class

- A **class** is like an object constructor, or a "blueprint" for creating objects.
- An **object** is created from a class.

- To create a `class`, use the keyword `class`.
- To create an `object` of a `class`, specify the *class name*, followed by the *object name*, and use the keyword **`new`**.

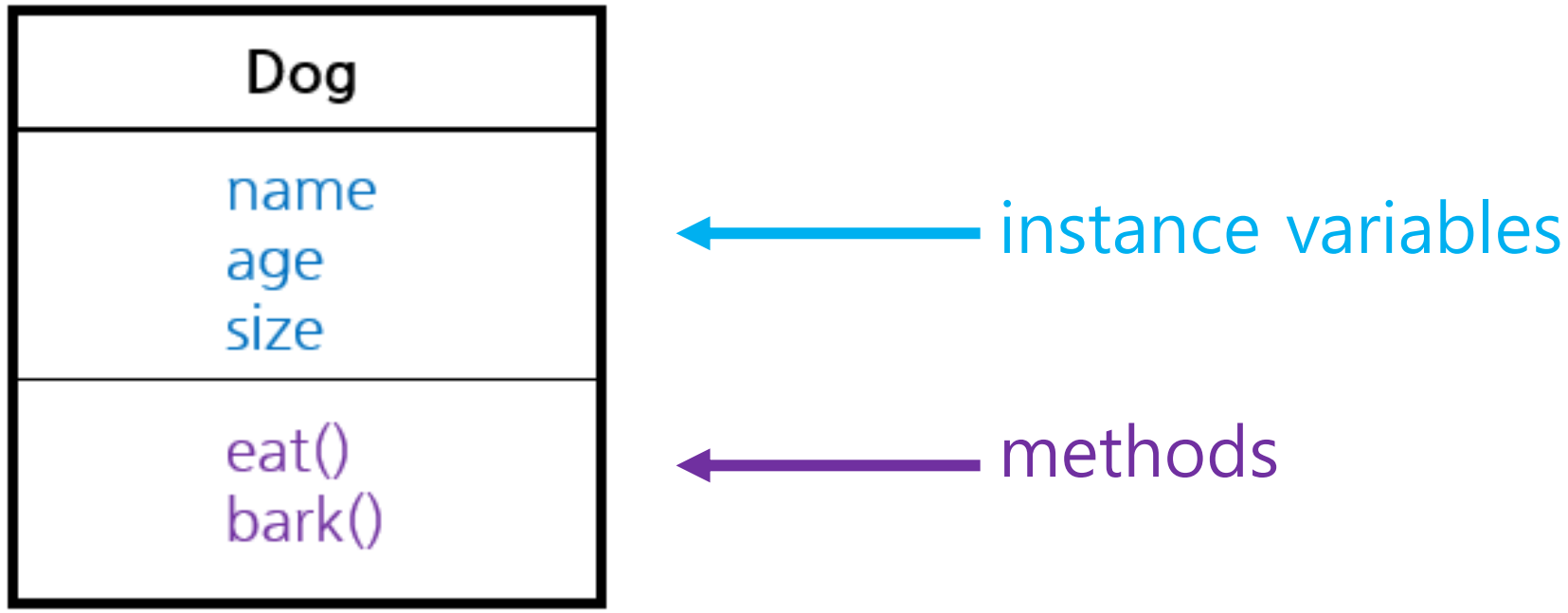
Java Class (cont.)

- Main Class: The class containing the main class of the Java program.

- One Java program can consist of several classes, but should contain **one** main method.

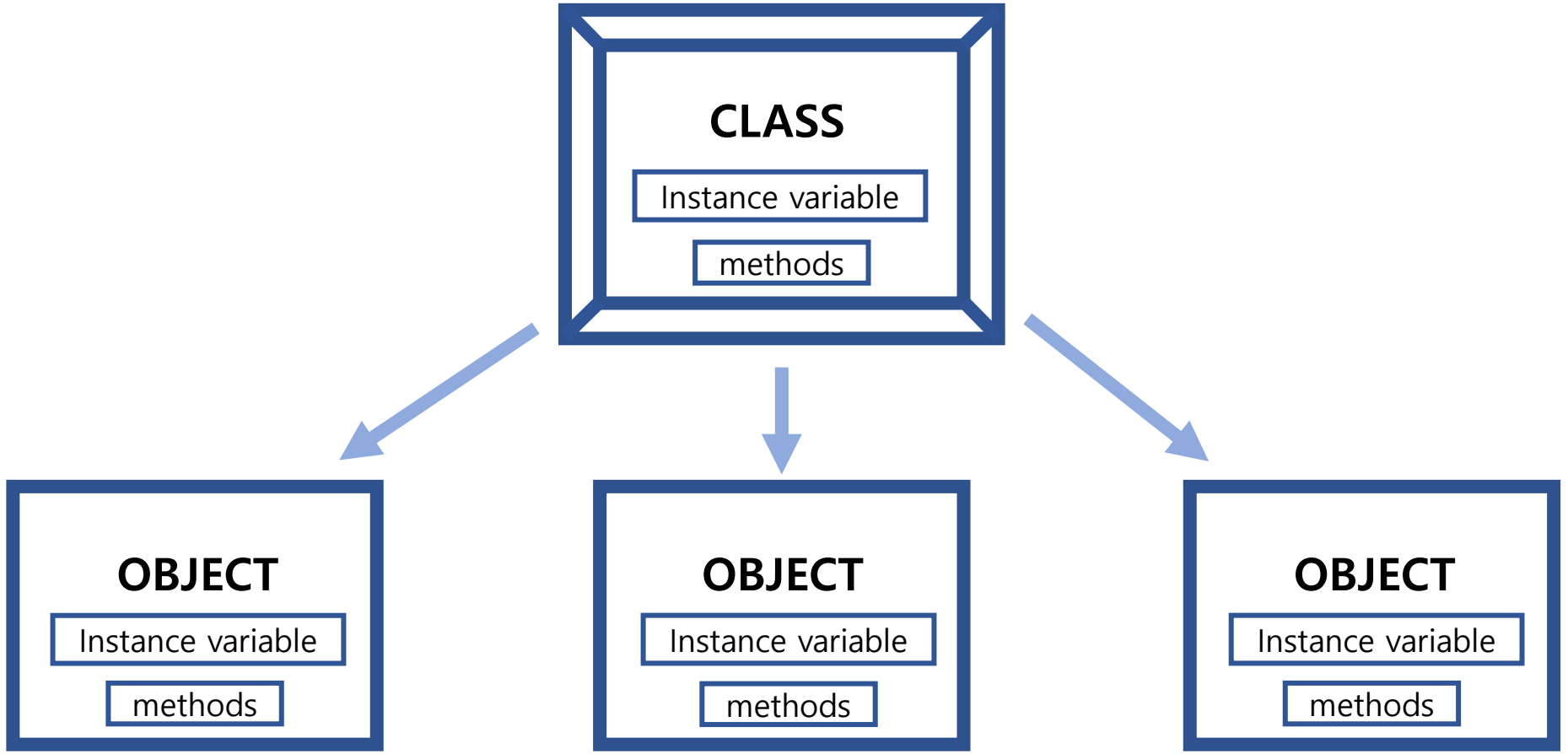
Java Class (cont.)

- A class consists of **variables** and **methods**.



Java Class (cont.)

- A class is like a blueprint. ⇒ Every object is created from a specific class.



Java Class – Example

- Create a class called "`Main`" with a variable `x`:

```
public class Main {
  int x = 5;
}
```

- Create an object called "`myObj`" and print the value of `x`:

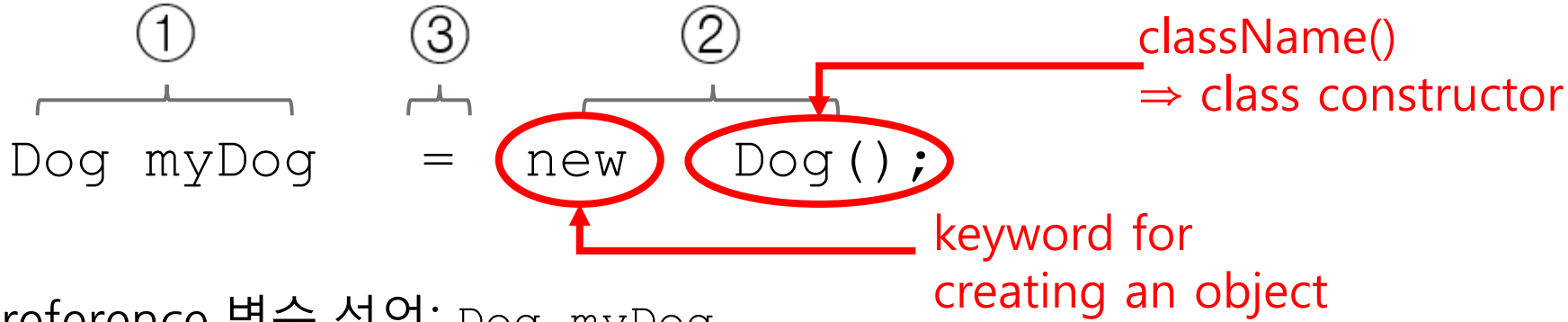
```
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

```
5
```

Java Objects

- An object is created from a class.

- To create an `object` of a `class`, specify the *class name*, followed by the *object name*, and use the keyword **`new`**.



① Object reference 변수 선언: `Dog myDog`
② Object 생성: `new Dog()`
③ Object와 reference 연결: `=`
✓ reference 유형과 object 유형이 같아야 함

Java Objects (cont.)

- Object · Instance vs Reference

```
Dog myDog = new Dog();
```

① Declare a reference variable

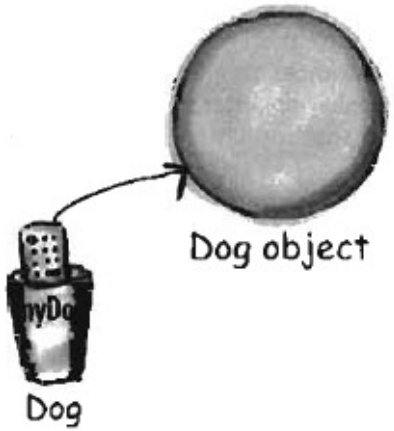
`Dog` **`myDog`**

② Create an object

`new` `Dog()`

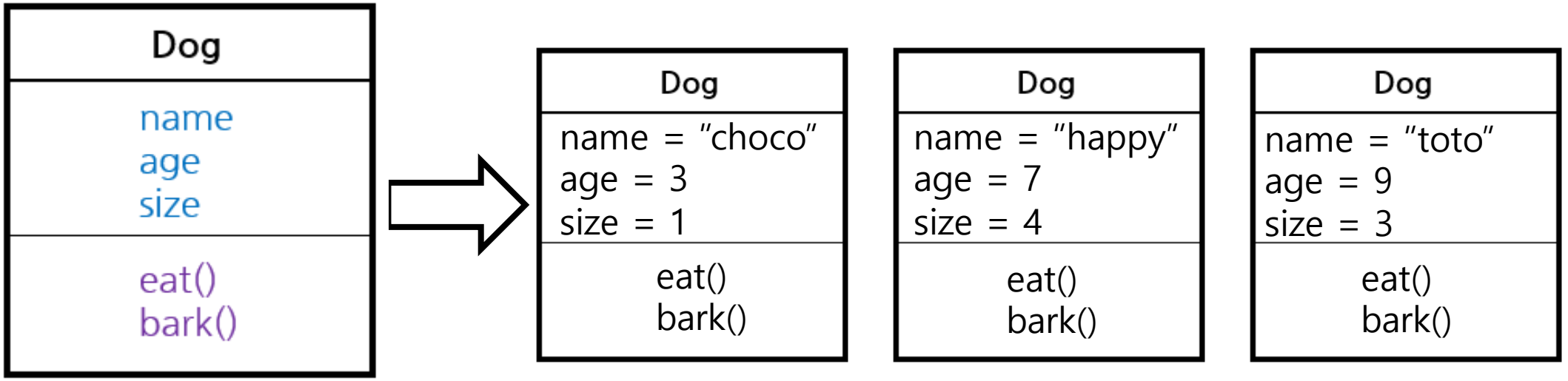
③ Link the object and the reference

=



Java Objects (cont.)

- Object
  - knows: instance variables
  - does: methods



Java Objects (cont.)

- Accessing Attributes and Methods
  - use dot notation (.)
  - `objectName.instanceVariable`
  - `objectName.method( )`

  - Example:

```
myDog.name

myDog.age = 3;

myDog.bark();  ⇒   멍멍!
```

Java Methods

- A **method** is a block of code which only runs when it is called.

- You can pass data, known as **parameters**, into a method.

- Methods are used to perform certain actions.

- Why use methods?
  ⇒ To reuse code: define the code once, and use it many times.

Java Methods (cont.)

- Method Naming Conventions
  - Use verbs that represent the actual function of the method.
  - Use camel-case, start with an upper-case letter without blanks.
  - ex) `my_first_method( )` → `myFirstMethod( )`

- Return Type
  - The data type of the result of running a method.
  - One method returns only one value.

Java Methods (cont.)

- Create a Method
  - A method must be declared within a class.
  - It is defined with the name of the method, followed by parentheses ( ).
  - Java provides some pre-defined methods, such as `System.out.println()`.
  - You can also create your own methods to perform certain actions.

Java Methods (cont.)

- Create a Method

```
public class className {
    static void methodName( ) {
        // code to be executed
    }
}
```

  - `methodName( )`: the name of the method
  - `static` : means the method belongs to the Main class and is not an object.
  - `void` : means that this method does not have a return value.

Java Methods (cont.)

- Call / Invoke a Method
  - To call a method in Java, write the method's name followed by two parentheses () and a semicolon ;
  - A method can also be called multiple times.

```
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

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

```
I just got executed!
I just got executed!
I just got executed!
```

Java Methods (cont.)

- Parameters and Arguments
  - Information can be passed to methods as **parameter**.
  - Parameters act as variables inside the method.
  - Parameters are specified after the method name, inside the parentheses ( ).
  - You can add as many parameters as you want, just separate them with a comma.
  - When a parameter is passed to the method, it is called an **argument**.

Java Methods – Example 1

```
public class Main {
  static void myMethod(String fname) {
    System.out.println(fname + " Refsnes");
  }

  public static void main(String[] args) {
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
  }
}
```

```
Liam Refsnes
Jenny Refsnes
Anja Refsnes
```

- This example has a method that takes a String called `fname` as parameter.
- When the method is called, we pass along a first name, which is used inside the method to print the full name.
- `fname` is a parameter, while Liam, Jenny and Anja are arguments.

Java Methods (cont.)

- Multiple Parameters
  - You can have as many parameters in a method as you like:
  - When working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Java Methods – Example 2

```
public class Main {
  static void myMethod(String fname, int age) {
    System.out.println(fname + " is " + age);
  }

  public static void main(String[] args) {
    myMethod("Liam", 5);
    myMethod("Jenny", 8);
    myMethod("Anja", 31);
  }
}
```

```
Liam is 5
Jenny is 8
Anja is 31
```

Java Methods – Return Values

- The `void` keyword indicates that the method should not return a value.

- If you want the method to return a value ⇒ use a primitive data type (such as `int`, `char`, etc.) instead of `void`. And use the `return` keyword inside the method.

Java Methods – Return Values (Example 1)

```
public class Main {
  static int myMethod(int x) {
    return 5 + x;
  }

  public static void main(String[] args) {
    System.out.println(myMethod(3));
  }
}
```

```
8
```

- This example returns the sum of a method's parameter (x) and 5.

Java Methods – Return Values (Example 2)

```
public class Main {
  static int myMethod(int x, int y) {
    return x + y;
  }

  public static void main(String[] args) {
    System.out.println(myMethod(5, 3));
  }
}
```

```
8
```

- This example returns the sum of a method's two parameters (`x, y`).

Java Methods – Return Values (Example 3)

```
public class Main {
  static int myMethod(int x, int y) {
    return x + y;
  }

  public static void main(String[] args) {
    int z = myMethod(5, 3);
    System.out.println(z);
  }
}
```

```
8
```

- You can also store the result in a variable.
  (recommended, as it is easier to read and maintain)
- This example binds int `z` with the return value of `myMethod(5, 3)`

Java Methods – Test 1

- 별 출력
  - Scanner를 이용하여 숫자 5을 입력받으면 별 5개를 출력하는 method 생성.
  - 한 class file 내에서 method 생성, main method 내에서 작성한 method 실행.
  - static 키워드 사용 필수!

```
public class D_Method_Test_01 {
	static void starMaker(int a) {
		//
	}
	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		int a = sc.nextInt();
		starMaker(a);
		}
}
```

Java Methods – Test 2

<학생 마라톤 기록>

가장 기록이 좋은 학생을 찾아서 그 학생의 이름과 마라톤 기록을 몇 시간 몇 분(예, 4시간 15분)의 형식으로 출력하기

String [] names = {"초코", "해피", "또또", "양파", "계피", "소금", "감자", "쫑쫑", "메리", "냥냥", "야옹", "냐옹", "시츄", "포메", "요키", "말티"}

int [] times = {341, 273, 278, 329, 445, 402, 388, 275, 243, 334, 412, 393, 299, 343, 317, 265}

- 배열을 입력으로 취해, 최단 기록 보유 학생에 대응하는 인덱스를 찾고, 그에 대응하는 인덱스, 시간, 분을 배열로 리턴하는 메소드를 작성

- 그 인덱스에 해당하는 이름을 출력 (main)

- 이 메소드를 각 학생의 기록으로 이루어진 배열에 적용하여 실행

- 반환된 배열을 통해 이름과 시간을 출력

Java Methods – Test 2 (cont.)

```
public class Marathon {

	public int[] findSmallest(int[] arr){
		return [];
	}

	public int[] findBiggest(){
		//
	}

	public static void main(String [] args){
		//names, times 배열 임의 지정

		A a = new A();
		int[] final = a.findSmallest(times);
		String finalName = names[final[0]]
		System.out.println(finalName)
		System.out.println(final[1]+ "시간" + final[2] + "분");
	}
}
```

Java Constructors

- A **constructor** in Java is a special method that is used to initialize objects.
- The constructor is called when an object of a class is created.
- The constructor name must match the class name, and cannot have a return type
- It can be used to set initial values for object attributes.
- The constructor is called when the object is created.
- All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you.

Java Constructors – Example

```
public class Main {
        int x;

        public Main() {
                x = 5;
        }
        public static void main(String[] args) {
                Main myObj = new Main();
                System.out.println(myObj.x);
        }
}
```

- Create a Main Class with a class attribute int `x`.
- Create a class constructor for the Main class `Main( )`
- Create an object of class Main ⇒ call the **`constructor`**

Java Modifiers

- **Access Modifiers**: controls the access level
- **Non-Access Modifiers**: do not control access level, but provides other f unctionality

```
public class Main
```

- The `public` keyword is an *access modifier*, meaning that it is used to s et the access level for classes, attributes, methods and constructors.

Java Modifiers (cont.)

- Access Modifiers
  - For classes, you can use either `public` or *default*:

Modifier	Description
`public`	The class is accessible by any other class.
default	The class is only accessible by classes in the same package. This is used when you don't specify a modifier.

Java Modifiers (cont.)

- Access Modifiers
  - For attributes, methods and constructors, you can use the one of the following:

Modifier	Description
`public`	The code is accessible by any other class.
`private`	The code is only accessible within the declared class.
default	The code is only accessible in the same package. This is used when you don't specify a modifier.
`protected`	The code is accessible in the same package and subclasses.

Java Modifiers (cont.)

- Non-Access Modifiers
  - For classes, you can use either `final` or `abstract`:

Modifier	Description
`final`	The class cannot be inherited by other classes.
`abstract`	The class cannot be used to create objects. (To access an abstract class, it must be inherited from another class.)

Java Modifiers (cont.)

- Non-Access Modifiers
  - For attributes and methods, you can use the one of the following:

Modifier	Description
`final`	Attributes and methods cannot be overridden/modified,
`static`	Attributes and methods belongs to the class, rather than an object.
`abstract`	Can only be used in an abstract class, and can only be used on methods. The method does not have a body. The body is provided by the subclass (inherited from).

Java Modifiers (cont.)

- final

✓ If you don't want the to override existing attribute values, declare attributes as final.

```
public class Main {
        final int x = 10;
        final double PI = 3.14;

        public static void main(String[] args) {
                Main myObj = new Main();
                myObj.x = 50;
                myObj.PI = 25;
                System.out.println(myObj.x);
        }
}
```

✓ myObj.x = 50; will generate an error: cannot assign a value to a final variable

✓ myObj.PI = 25; will generate an error: cannot assign a value to a final variable

Java Modifiers (cont.)

- static
  - ✓ A static method means that it can be accessed without creating an object of the class, unlike public.

```
public class Main {
        static void myStaticMethod() { // Static method
                System.out.println("Static methods can be called without creating objects");
        }
        public void myPublicMethod() { // Public method
                System.out.println("Public methods must be called by creating objects");
        }
        public static void main(String[ ] args) { // Main method
                myStaticMethod();                  // Call the static method
                // myPublicMethod();               // This would output an error
                Main myObj = new Main();           // Create an object of Main
                myObj.myPublicMethod();            // Call the public method
        }
}
```

```
Static methods can be called without creating objects
Public methods must be called by creating objects
```

Java Modifiers (cont.)

- abstract
  - ✓ An abstract method belongs to an abstract class, and it does not have a body.
  - ✓ The body is provided by the subclass:

```java
// Main.java - abstract class
abstract class Main {
        public String fname = "John";
        public int age = 24;
        public abstract void study(); //abstract method
}
// Subclass (inherit from Main)
class Student extends Main {
        public int graduationYear = 2018;
        public void study() {
            // the body of the abstract method
            System.out.println("Studying all day");
        }
 }
```

```java
// Second.java
class Second {
        public static void main(String[] args) {
        // create an object of the Student class
        // inherits attributes·methods from Main
        Student myObj = new Student();
        System.out.println("Name: " + myObj.fname);
        System.out.println("Age: " + myObj.age);
        System.out.println("Graduation Year: " +
                                    myObj.graduationYear);
        myObj.study(); // call abstract method
        }
 }
```

```
Name: John
Age: 24
Graduation Year: 2018
Studying all day
```

Java Final Test – Student.java, Student_Info.java

<학생 정보 프로그램 생성>

* 메인 메소드용 클래스 1개, 설계용 클래스 1개

1. 설계용 클래스
  - 이름, 국어, 영어, 수학점수, 총점, 평균, 등급

2. 메인용 클래스

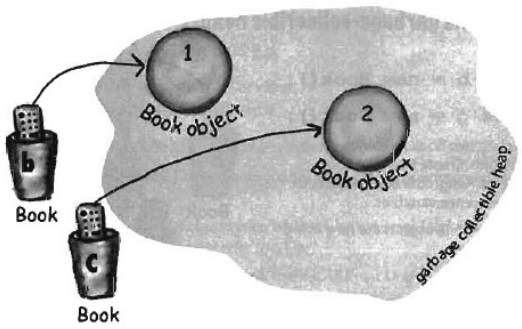
ex) 객체생성 방법 = person s1 = new person();

```
menu: while (true) {
        switch( ) {
          case 1: 학생 정보 입력
           - 이름, 국영수를 입력
           - 총점, 평균, 등급 계산하여 객체에 저장
           - 평균 90 이상 : A
             80 이상 90 이하 : B
             70 이상 80 이하 : C
             60 이상 70 이하 : D
             60 미만 : F

          case 2: 학생 정보 출력
            - 이름, 평균, 등급 출력

          case 3: 결과 보기
            - 평균이 85.5 이상이면 합격/ 미만이면 불합격

          case 4: 종료 하기
              break menu;

          default:
            - "오류" 출력
          }
}
```

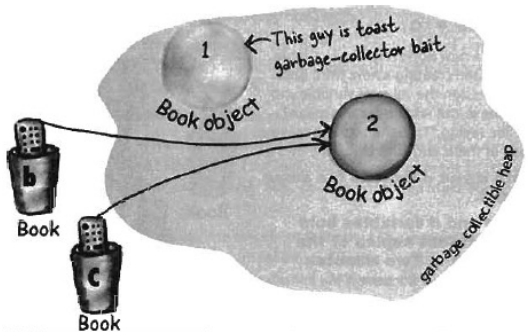
Garbage Collection

- Garbage Collection tracks each and every object available in the JVM heap space, and removes the unused ones.
- Basically, Garbage Collection works in two simple steps, known as Mark and Sweep:
  - **Mark –** garbage collector identifies which pieces of memory are in use and which aren't.
  - **Sweep –** removes objects identified during the "mark" phase.

```
Book b = new Book();
Book c = new Book();
```

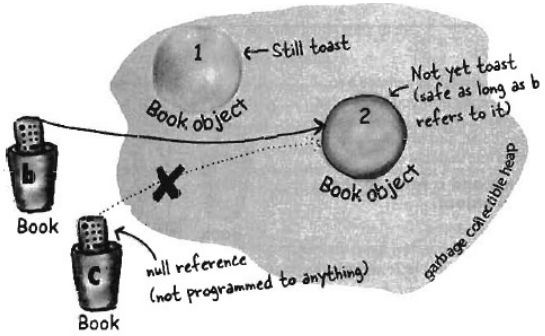



```
b = c;
```



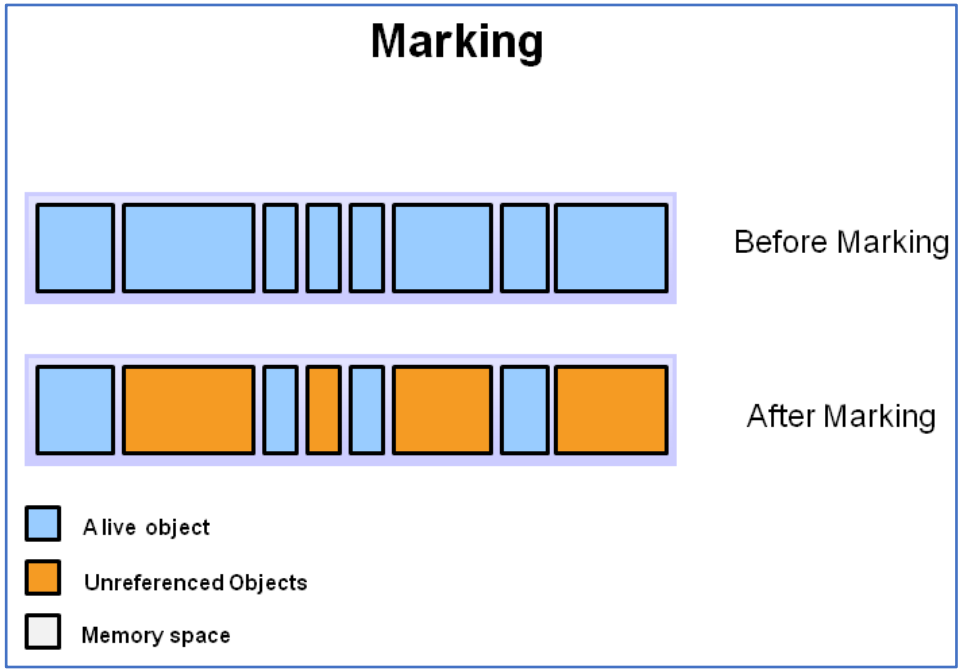
레퍼런스가 없는 객체 (1)는
가비지 컬렉션의 대상

```
c = null;
```



Garbage Collection (cont.)

- Marking
  - This is where the garbage collector identifies which pieces of memory are in use and which are not.
  - Referenced objects are shown in blue.
  - Unreferenced objects are shown in gold.
  - All objects are scanned in the marking phase to make this determination.



Garbage Collection (cont.)

- Normal Deletion
  - Normal deletion removes unreferenced objects leaving referenced objects and pointers to free space.
  - The memory allocator holds references to blocks of free space where new object can be allocated.