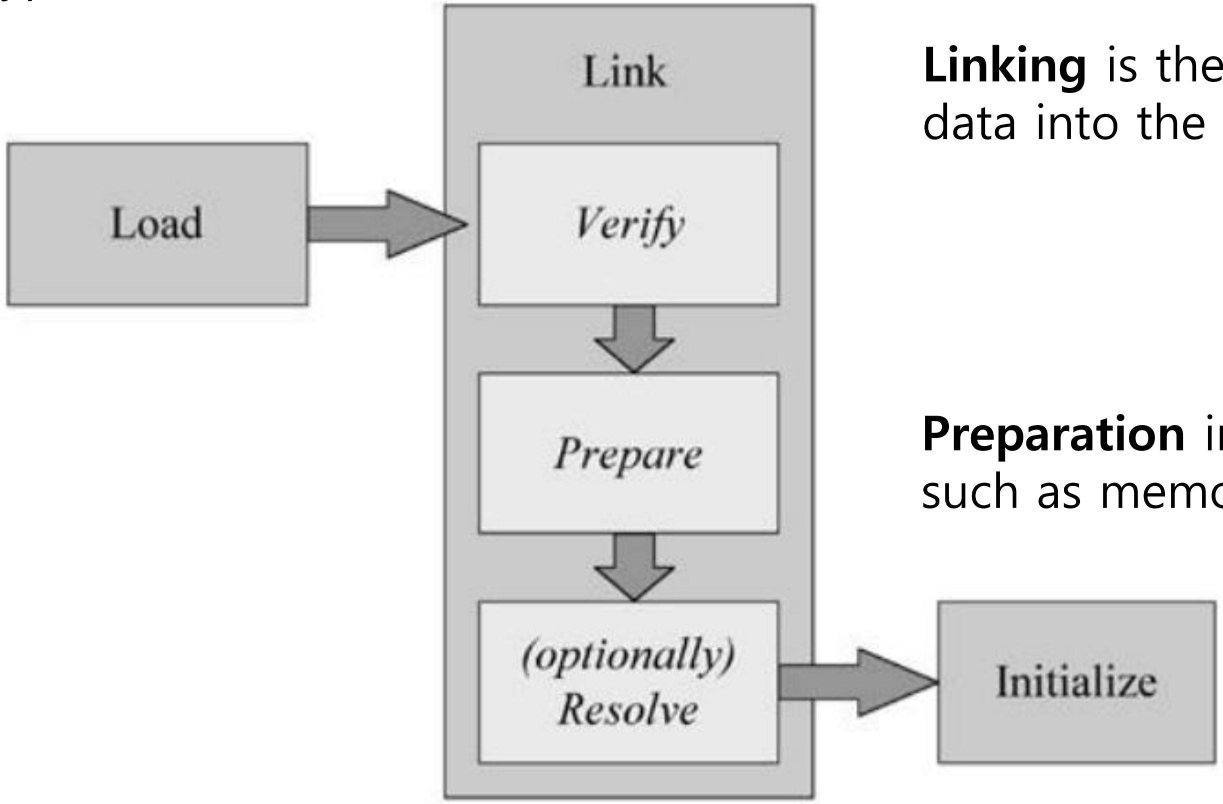
The Lifetime of a Class

The class loader takes care of loading, linking, and initializing types
즉, Class Loader가 다하는 일이다.

Loading is the process of bringing a binary form for a type into the Java Virtual Machine



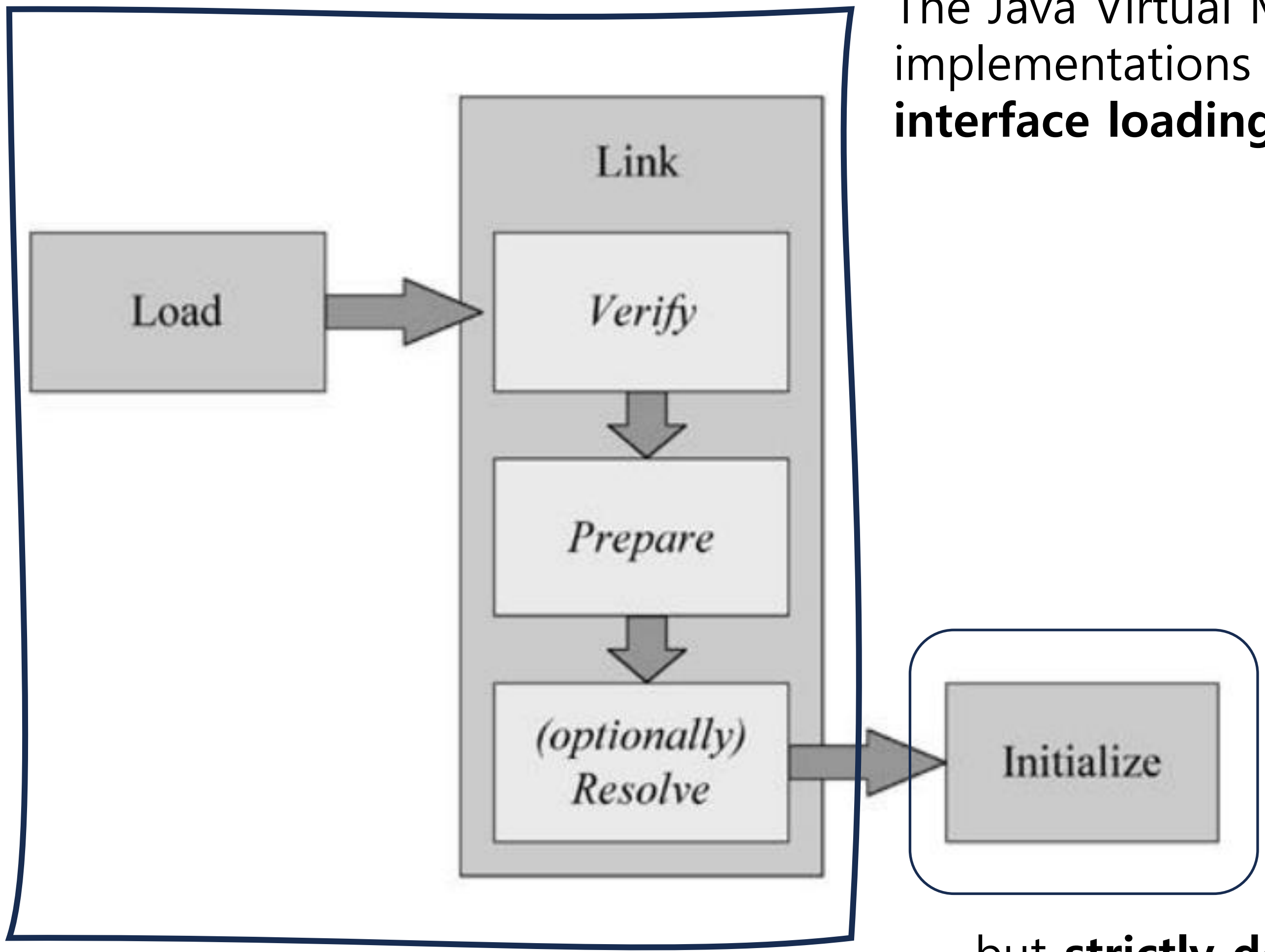
Linking is the process of incorporating the binary type data into the runtime state of the virtual machine.

Preparation involves allocating memory needed by the type, such as memory for any class variables

During **initialization**, the class variables are given their proper initial values

Resolution is the process of transforming symbolic references in the constant pool into direct references

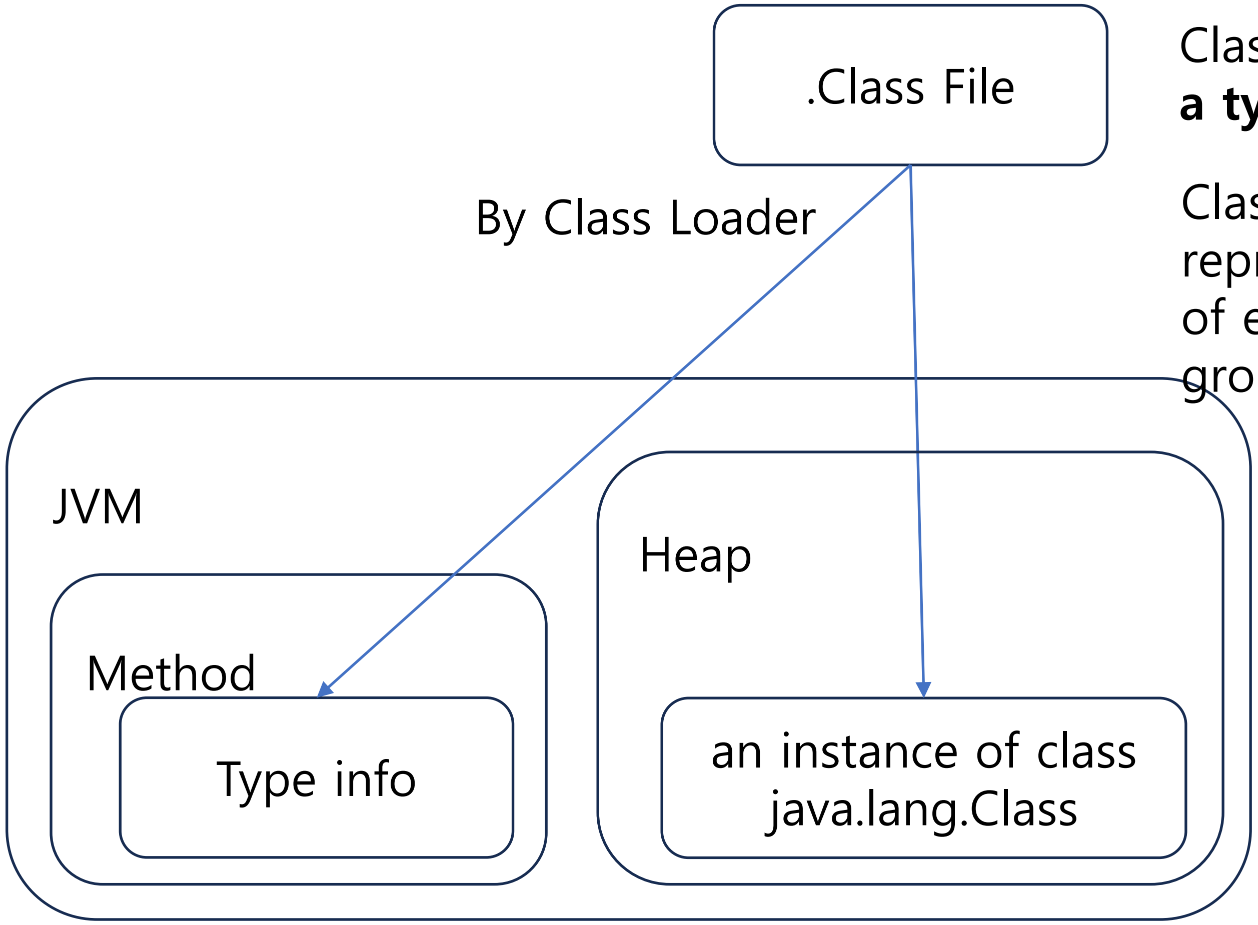
The Lifetime of a Class



The Java Virtual Machine specification gives implementations **flexibility in the timing of class and interface loading and linking**

, but **strictly defines the timing of initialization**. All implementations must initialize each class and interface on its first active use

Loading

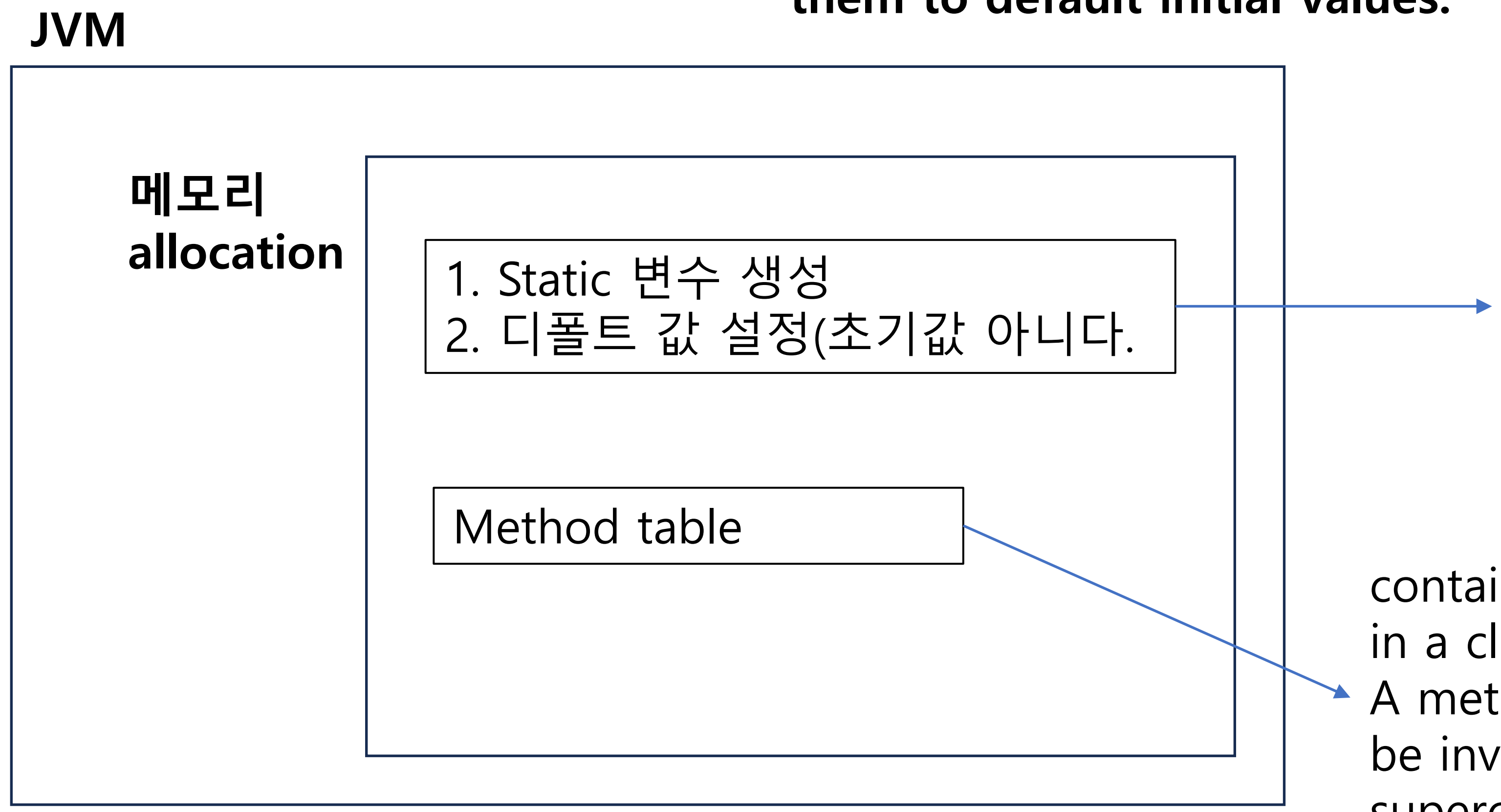


Class loaders (primordial or object) need **not wait until a typeís first active use** before they load the type.

Class loaders are allowed to cache binary representations of types, load types early in anticipation of eventual use, or load types together in related groups

Linking - Preparation

During the preparation phase, the Java Virtual Machine **allocates memory for the class variables** and **sets them to default initial values.**



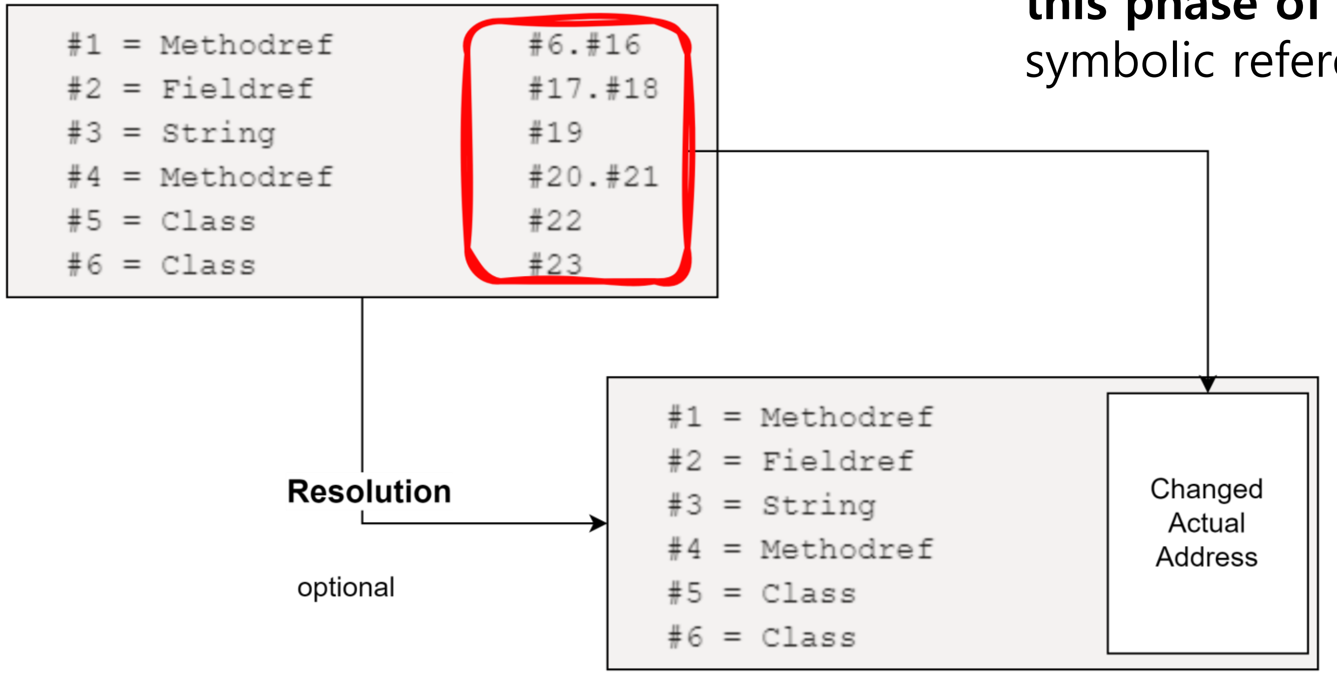
The class variables are not initialized to their proper initial values until the initialization phase. (No Java code is executed during the preparation step.)

contains a pointer to the data for every method in a class
A method table enables an inherited method to be invoked on an object without a search of superclasses at the point of invocation

Linking - Resolution

Resolution is the process of locating classes, interfaces, fields, and methods referenced symbolically from a typeís constant pool, and **replacing those symbolic references with direct references**

this phase of linking is optional until (and unless) each symbolic reference is first used by the program.


```
#1 = Methodref          #6.#16
#2 = Fieldref           #17.#18
#3 = String             #19
#4 = Methodref          #20.#21
#5 = Class              #22
#6 = Class              #23
```

```
#1 = Methodref
#2 = Fieldref
#3 = String
#4 = Methodref
#5 = Class
#6 = Class
```

Linking - Initialization

the process of setting class variables to their proper initial values

A class variable initializer is an equals sign

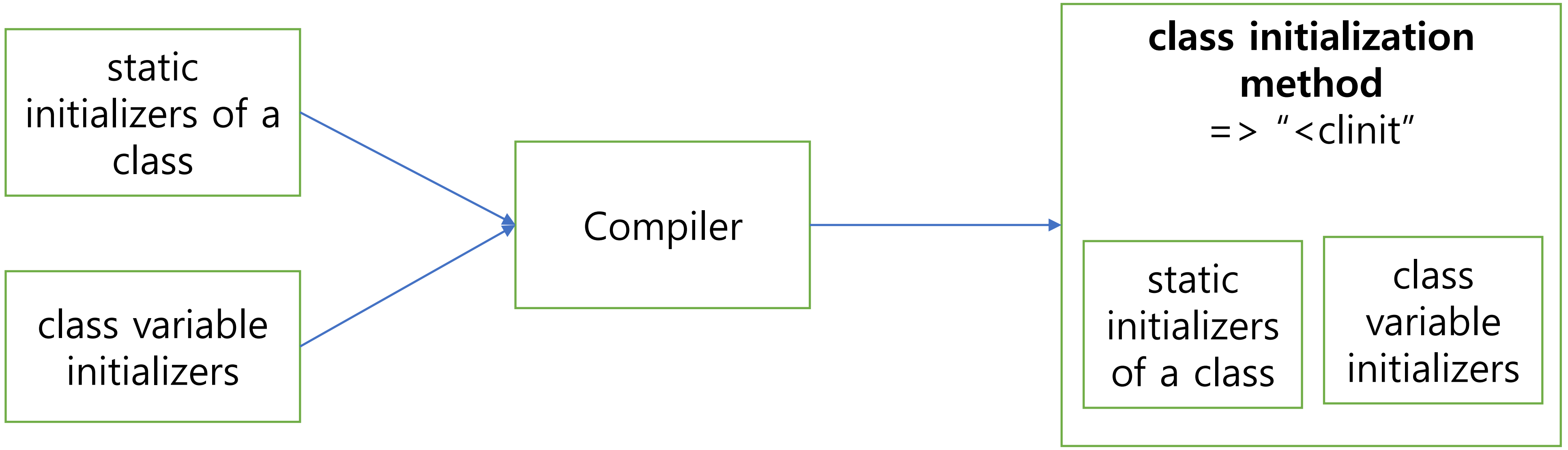
```
class Example1a {

    // "= 3 * (int) (Math.random() * 5.0)" is the class variable
    // initializer
    static int size = 3 * (int) (Math.random() * 5.0);
}
```

A static initializer is a block of code introduced by the static keyword

```
// On CD-ROM in file classlife/ex1/Example1b.java
class Example1b {

    static int size;

    // This is the static initializer
    static {

        size = 3 * (int) (Math.random() * 5.0);
    }
}
```

Linking - Initialization

set a classís static variables to their proper initial values.


static initializers of a class
class variable initializers
Compiler
class initialization method
=> "<clinit"
static initializers of a class
class variable initializers

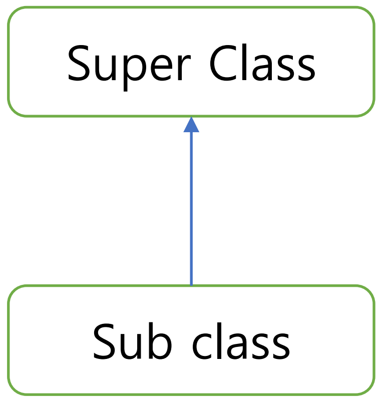
Java Byte Code에서는 이렇게 실행 되는 것 같다.

All the class variable initializers and static initializers of a class are collected by the Java compiler and placed into one special method, **the class initialization method**

the class initialization method is named **"<clinit"**

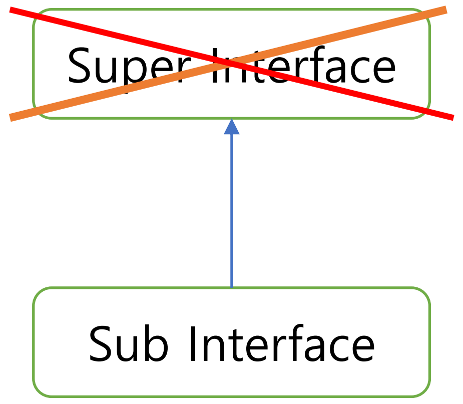
Linking - Initialization

Initialization of a class consists of two steps



1. Initializing the classís direct superclass (if any), if the direct superclass hasnít already been initialized

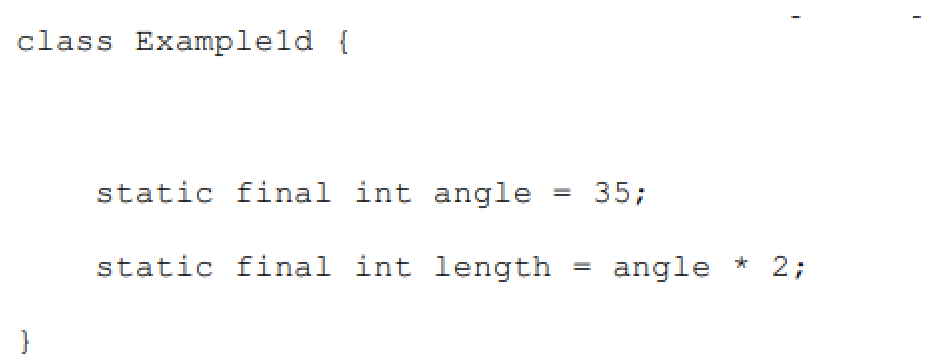
2. Executing the classís class initialization method, if it has one



Initialization of an interface does not require initialization of its superinterfaces

Linking - Initialization

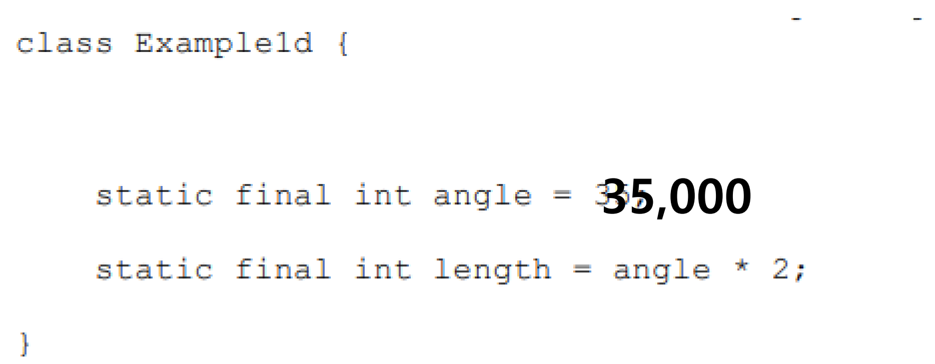
하지만, static final은 "<clinit" method 내부로 컴파일 되지 않는다.



```
class Example1d {

    static final int angle = 35;
    static final int length = angle * 2;
}
```

Case 1.

Constant int value를 constant pool에 넣는 것이 아닌, operan에 35라는 값이 들어가게 된다.



```
class Example1d {

    static final int angle = 35,000
    static final int length = angle * 2;
}
```

Case 2.

If the constant value of angle were outside the range of a short (-32,768 to 32,767), say 35,000, the class would have a **CONSTANT_Integer_info entry in its constant pool** with the value of 35,000.

Linking – Initialization Timing

the Java Virtual Machine initializes types **on their first active use** or,
in the case of classes, **upon the first active use of a subclass**.

Active use 란?

invoking a class initialization method on a new instance of a class

creating an array whose element type is the class,

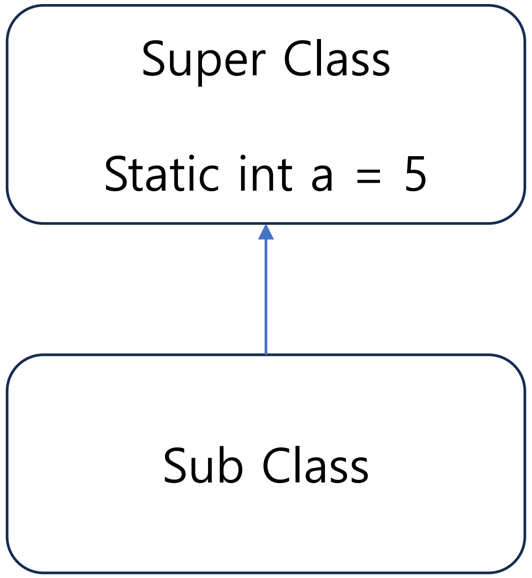
invoking a method declared in a class

accessing a non-constant field declared in a class or interface

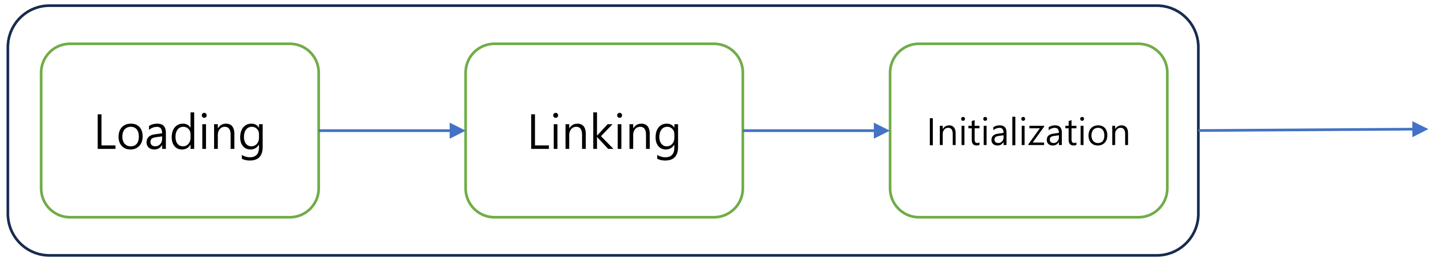
A use of a non-constant field is an active use of only the class or interface that actually declares the field

"SubClasss.a"는Sub Class의 Initialization 발생하지 않는다.
오직, SuperClass 만 Initialized 된다.

또한, static final은 complie-time에 initialized 되므로, static final 변수 사용도 class의 initialization을 발생시키지 않는다.

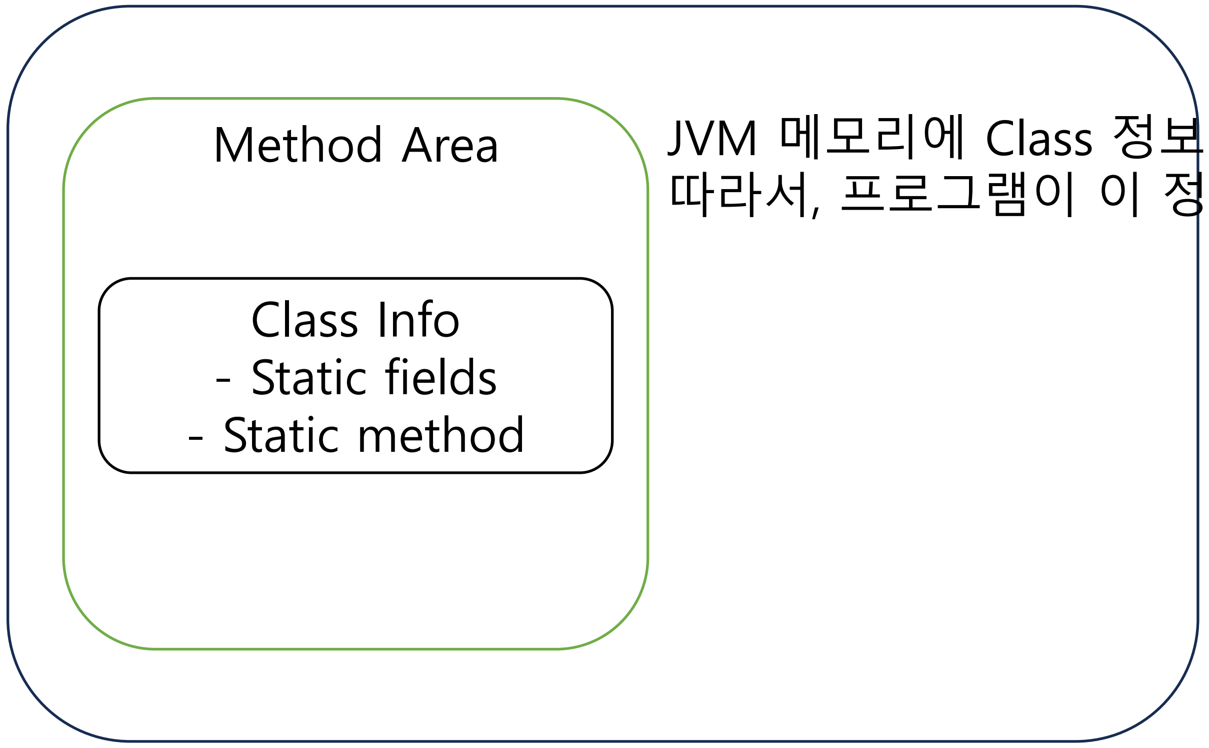


The Lifetime of an Object



The program can
1. access its static fields,
2. invoke its static methods,
3. or create instances of it

JVM



JVM 메모리에 Class 정보가 장착
따라서, 프로그램이 이 정보를 활용 가능

참고)
운영체제에서 CPU가 활용할 수 있는 정보는 기본적으로 Memory에 존재해야 하는 것과 비슷하다.
즉, Class 정보가 메모리에 온전하게 존재해야 활용될 수 있다.

The Lifetime of an Object - Class Instantiation

Class can be instantiated explicitly

The three ways a class can be instantiated

- new operator
- by invoking newInstance() on a Class object
- by invoking clone() on any existing object

Class can be instantiated implicitly

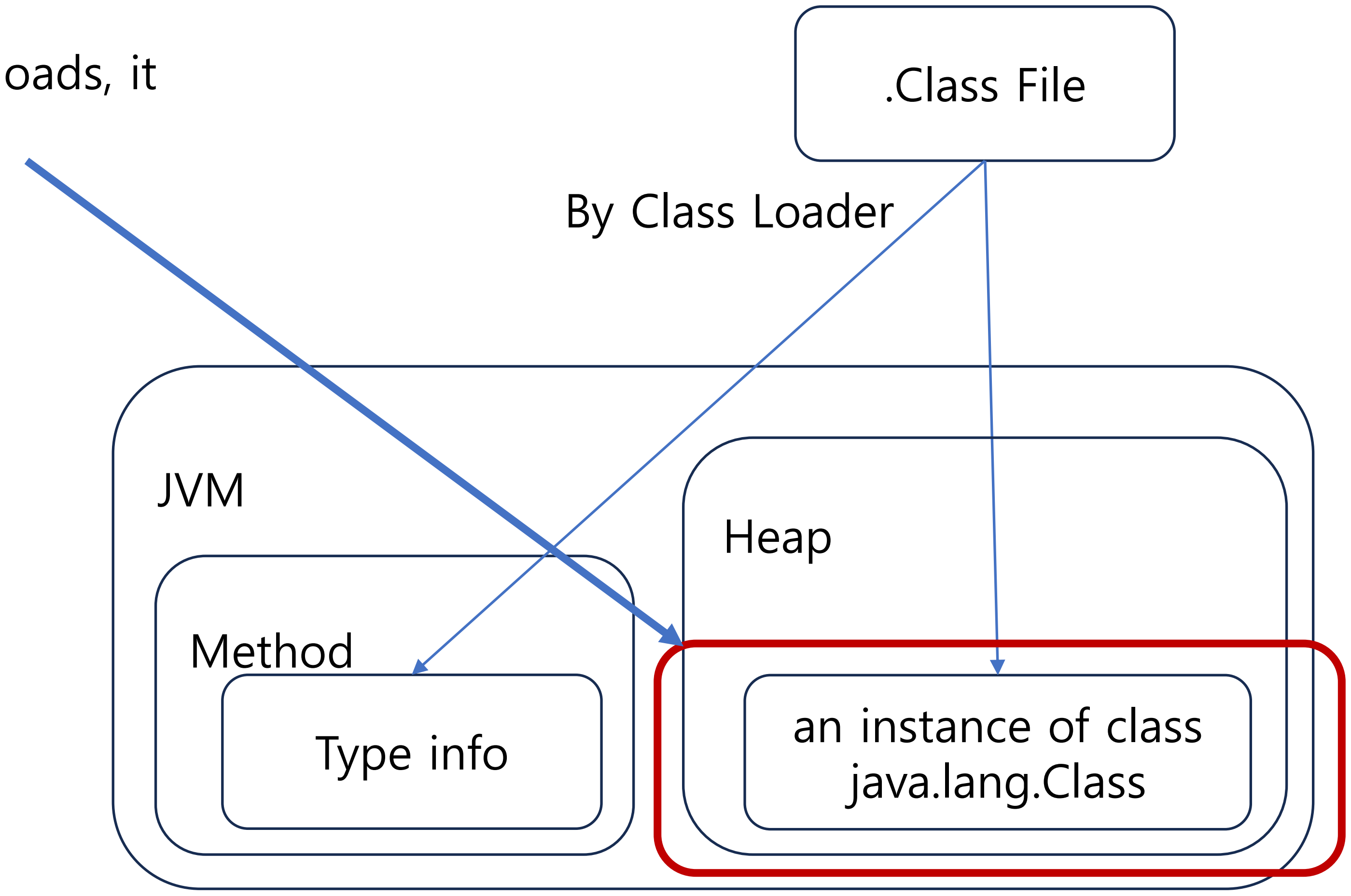
1. **String objects that hold the command line arguments**
2. for every type a Java Virtual Machine loads, it implicitly instantiates a new Class object
3. when the Java Virtual Machine loads a class that contains CONSTANT_String_info entries in its constant pool
4. Another way objects can be created implicitly is through the process of evaluating an expression that involves **the string concatenation operator**

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

The Lifetime of an Object - Class Instantiation

Class can be instantiated implicitly

2. for every type a Java Virtual Machine loads, it implicitly instantiates a new **Class object**



The Lifetime of an Object - Class Instantiation

Class can be instantiated implicitly

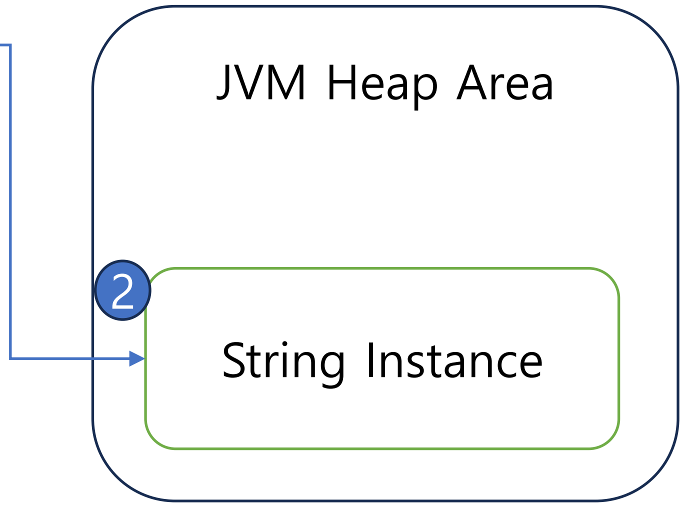
3. when the Java Virtual Machine loads a class that contains CONSTANT_String_info entries in its constant pool, it may instantiate new String objects to represent those constant string literals

The process of transforming a **CONSTANT_String_info entry in the method area to a String instance on the heap** is part of the **process of constant pool resolution.**

3

```
 #2 = Fieldref           #17.#18        // java/lang/System.out:Ljava/io/PrintStream;
 #3 = String             #19            // Hello from Hello.main!
 #4 = Methodref            #20.#21        // java/io/PrintStream.println:(Ljava/lang/String;)V
 #5 = Class              #22            // Hello
 #6 = Class              #23            // java/lang/Object
//생략...
#19 = Utf8 Hello from Hello.main!
//생략...
#24 = Utf8               java/lang/System
#25 = Utf8               out
#26 = Utf8               Ljava/io/PrintStream;
#27 = Utf8               java/io/PrintStream
#28 = Utf8               println
#29 = Utf8               (Ljava/lang/String;)V
```

1 이 문자열을 String으로 만든다.



The Lifetime of an Object - Class Instantiation

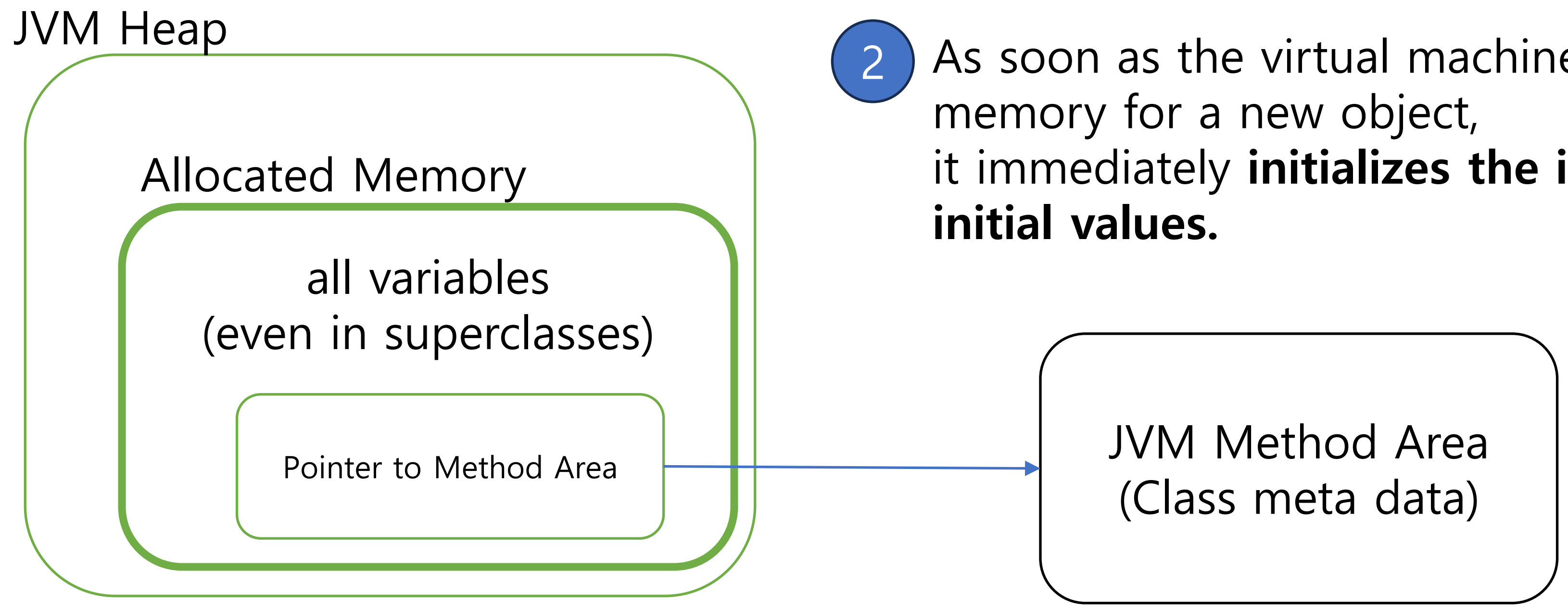
When the Java Virtual Machine creates a new instance of a class

1

it first allocates memory on the heap to hold the objectís instance variables.

a pointer to class data in the method area are also likely allocated at this point.

2 As soon as the virtual machine has set aside the heap memory for a new object,
it immediately **initializes the instance variables to default initial values.**



The Lifetime of an Object - Class Instantiation

3

give the instance variables their proper initial values

The Java Virtual Machine uses **two techniques** to do this

otherwise

object is being created **because of a clone()**

2

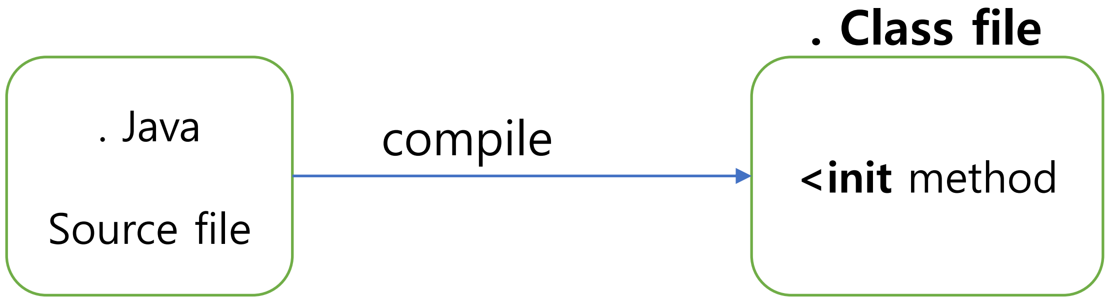
the virtual machine invokes an instance **initialization method** on the object.

1

The virtual machine copies the values of the instance variables of the object being cloned into the new object.

The Lifetime of an Object - Class Instantiation

4 The Java compiler **generates at least one instance initialization method for every class it compiles**. In the Java class file, the instance initialization method is named **"<init."**



<init method

- an invocation of the same-class **<init()** method
- the bytecodes that implement the body of the corresponding constructor

constructor begins with an explicit invocation of another constructor in the same class

```
Class A {
  public A() {
    this(a, b, c)
    //code..
  }
}
```

<init method

constructor does **not begin with a this() invocation** and **the class is not Object,**

- an invocation of a **superclass <init method**
- the bytecodes for any **instance variable initializers**
- the bytecodes that implement the body of the corresponding constructor

```
Class A {
  public A() {
    a = 5;
    b = 6;
  }
}
```

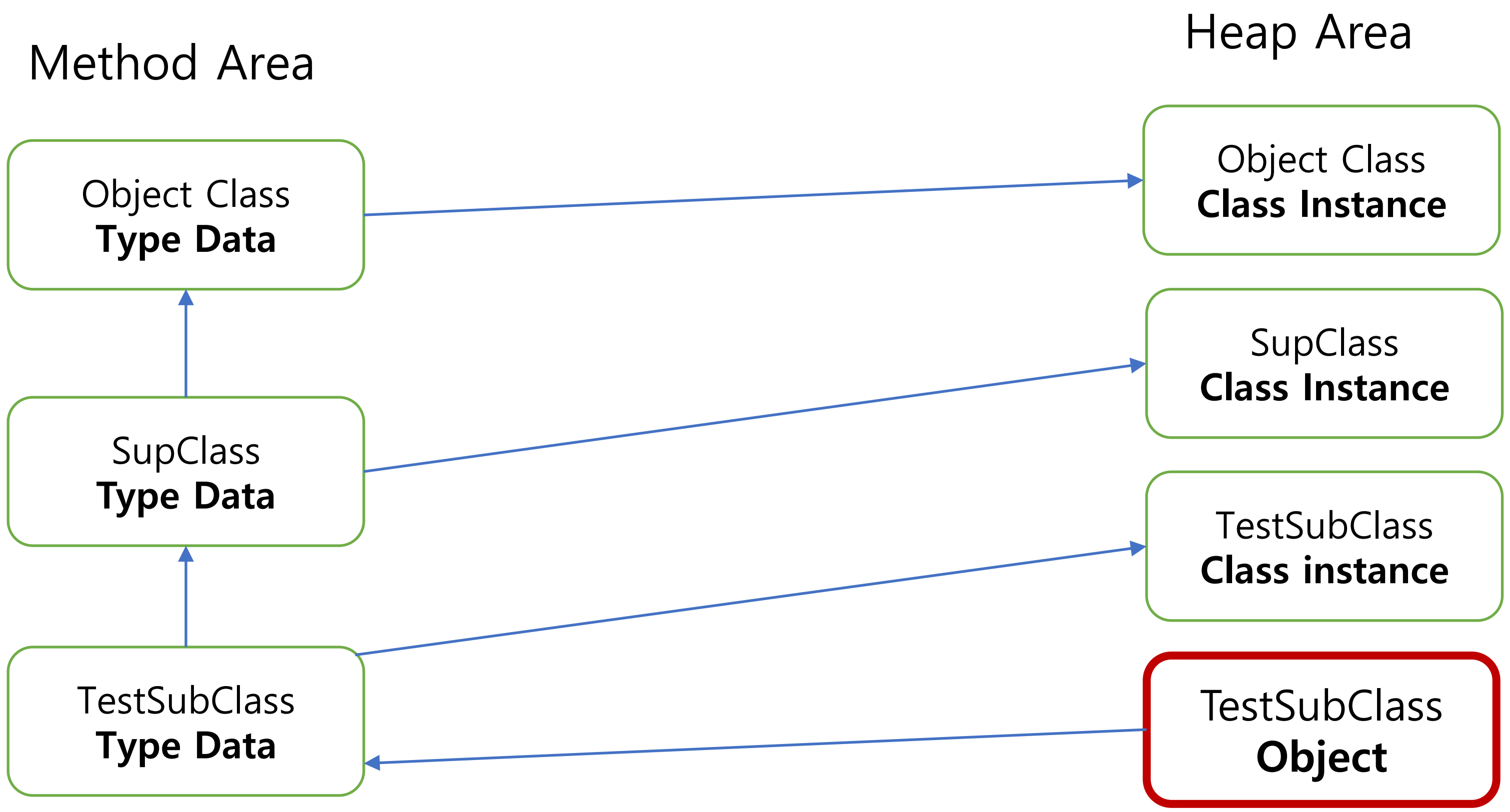
Unloading and Finalization of Classes

- Types loaded through the primordial class loader will always be reachable and **never be unloaded.**
- **Only dynamically loaded types**--those loaded through class loader objects--**can become unreachable and be unloaded** by the virtual machine.

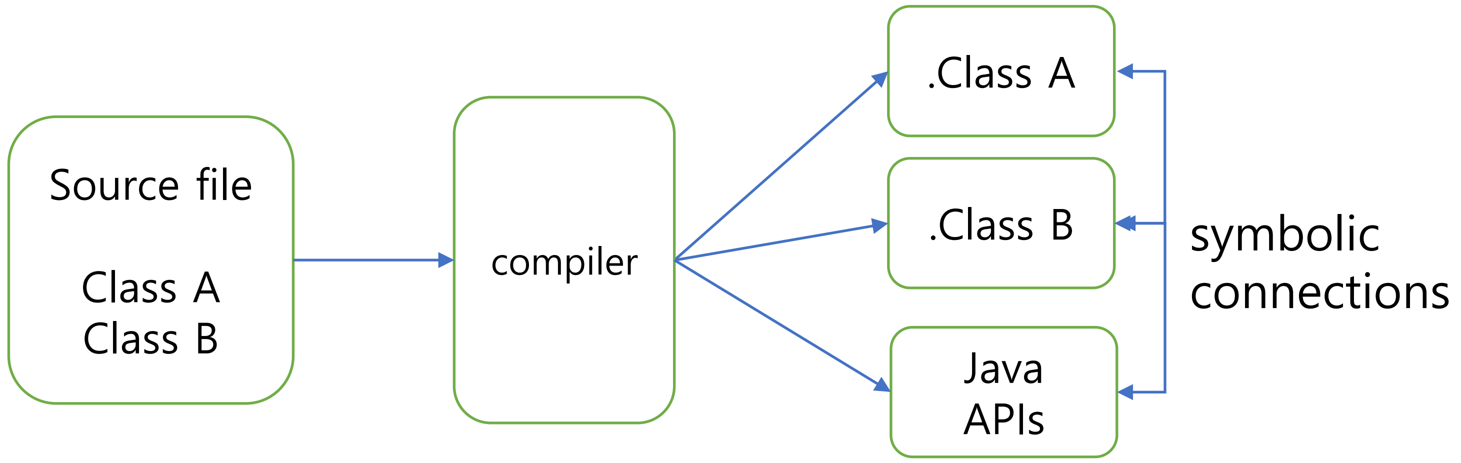
What is reachable?

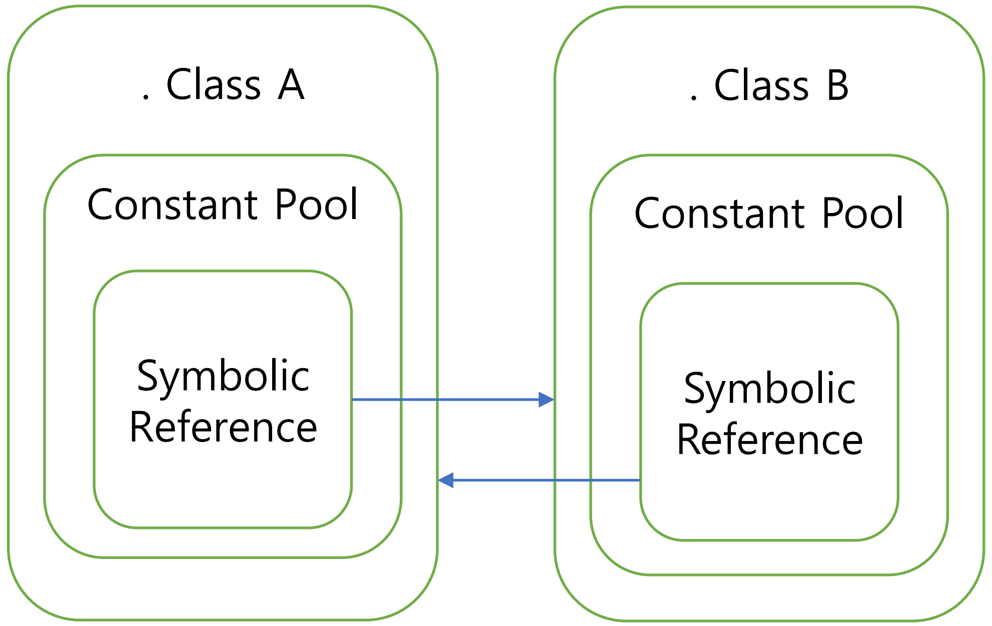
```
class TestSubClass extends SupClass {
}
```

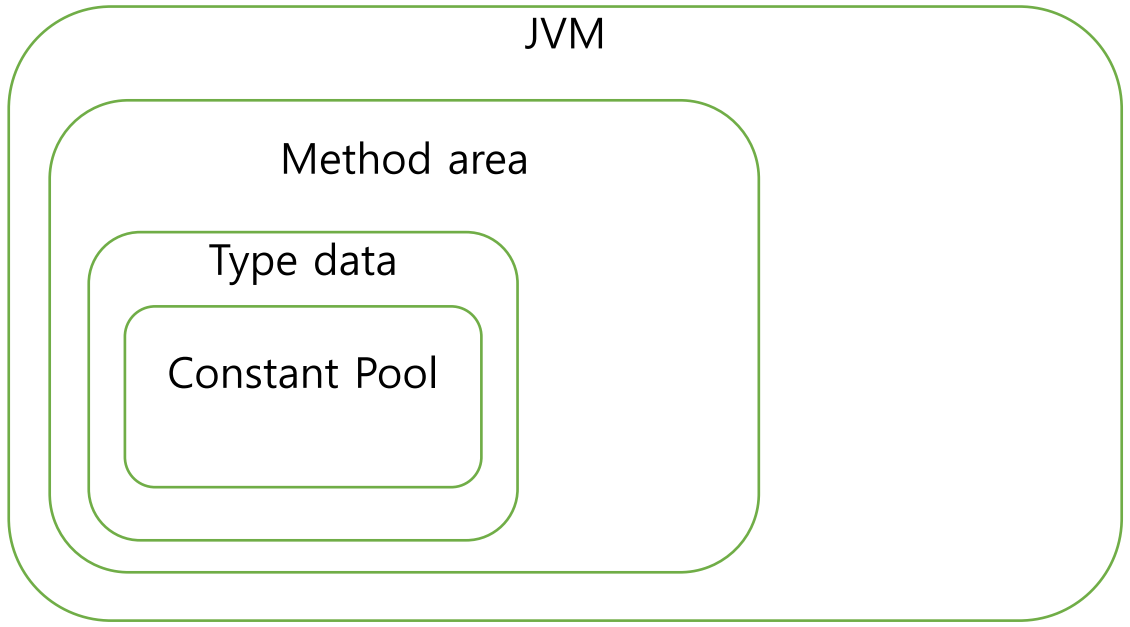
TestSubClass의 객체가 하나 존재하는 경우
JVM Memory의 상황을 살펴보자.



The Linking Model – Dynamic Linking and Resolution



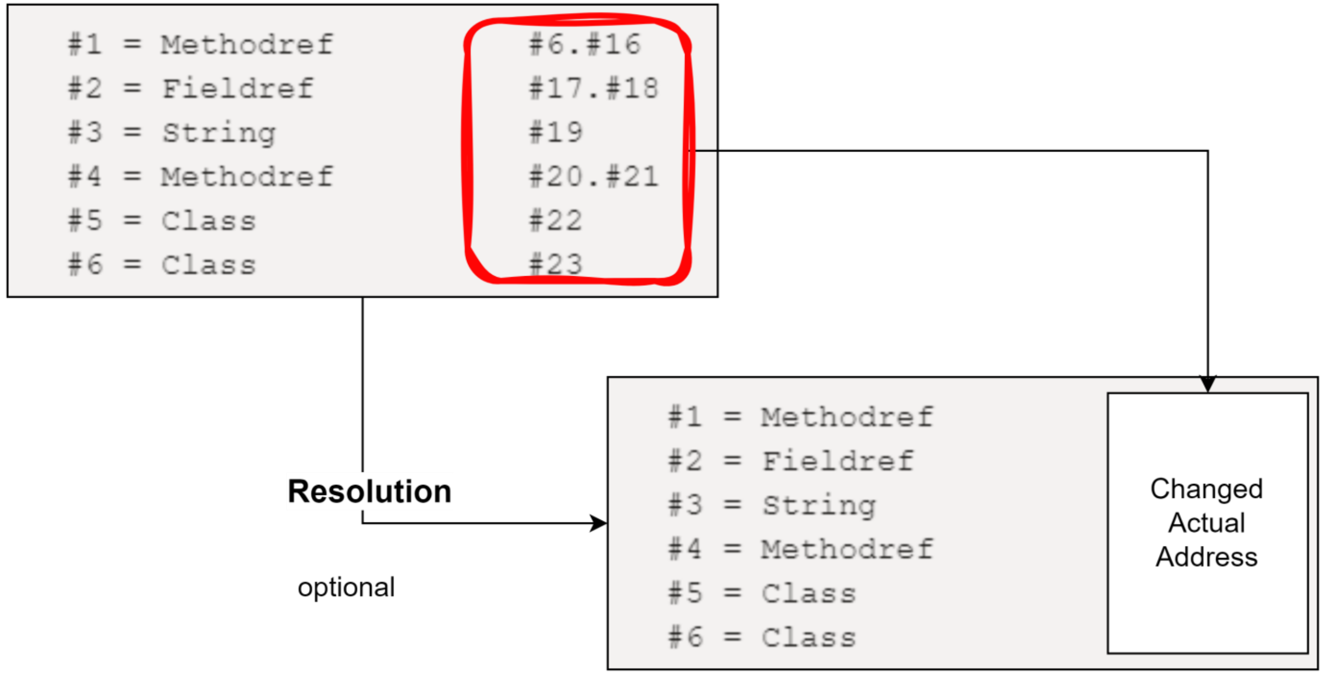




Class file이 JVM 내부로 Load 되면,
Type Data 안에 Contant Pool이 존재하게 된다.

The Linking Model – Dynamic Linking and Resolution

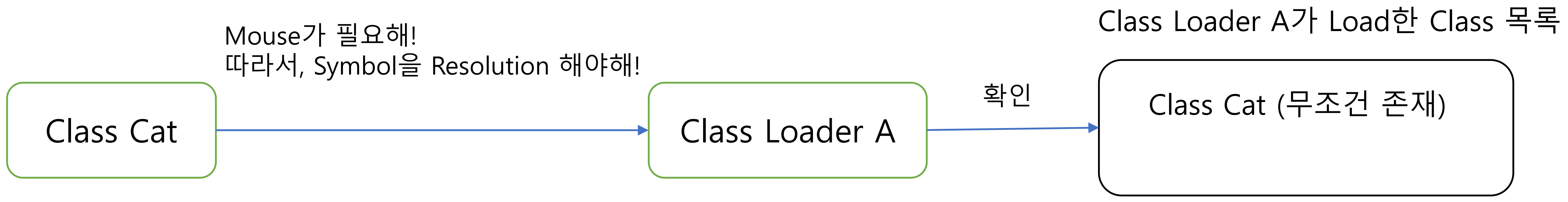
Resolution is the process of finding the entity identified by the symbolic reference and replacing the symbolic reference with a direct reference



```
#1 = Methodref          #6.#16
#2 = Fieldref           #17.#18
#3 = String             #19
#4 = Methodref          #20.#21
#5 = Class              #22
#6 = Class              #23
```

```
#1 = Methodref
#2 = Fieldref
#3 = String
#4 = Methodref
#5 = Class
#6 = Class
```

The Linking Model – Dynamic Linking and Resolution

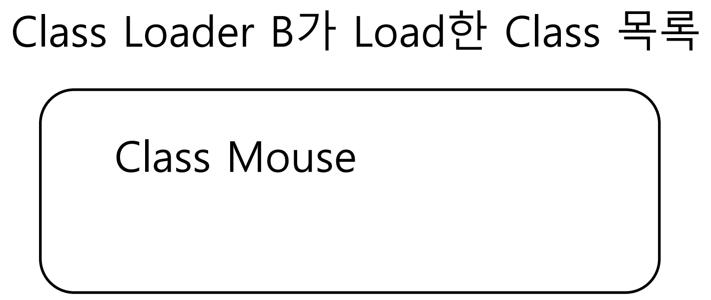

Mouse가 필요해!
따라서, Symbol을 Resolution 해야해!
Class Cat
Class Loader A
확인
Class Loader A가 Load한 Class 목록
Class Cat (무조건 존재)

Constant Pool
#21 = Class ...Mouse

1 Mouse 없는 경우 Class Loader A가 Classs Cat을 Load한다.

2 #21 = Class ...Mouse의 **주소**

※ 다른 Class Loader가 Load 했더라도 다시 Load한다.


Class Loader B가 Load한 Class 목록
Class Mouse