JAVA

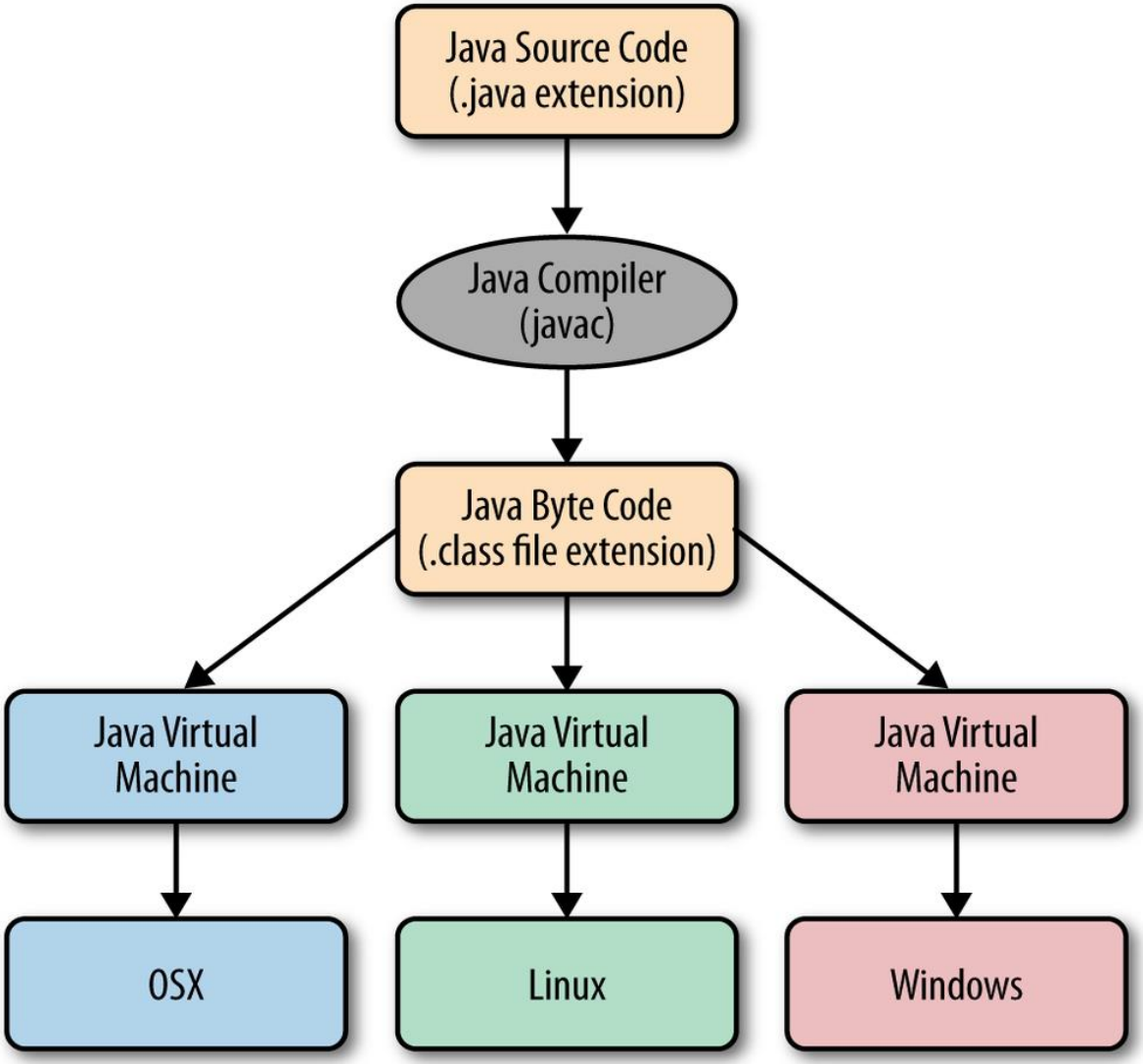
자바기초

김선녕(ksycafe@gmail.com)

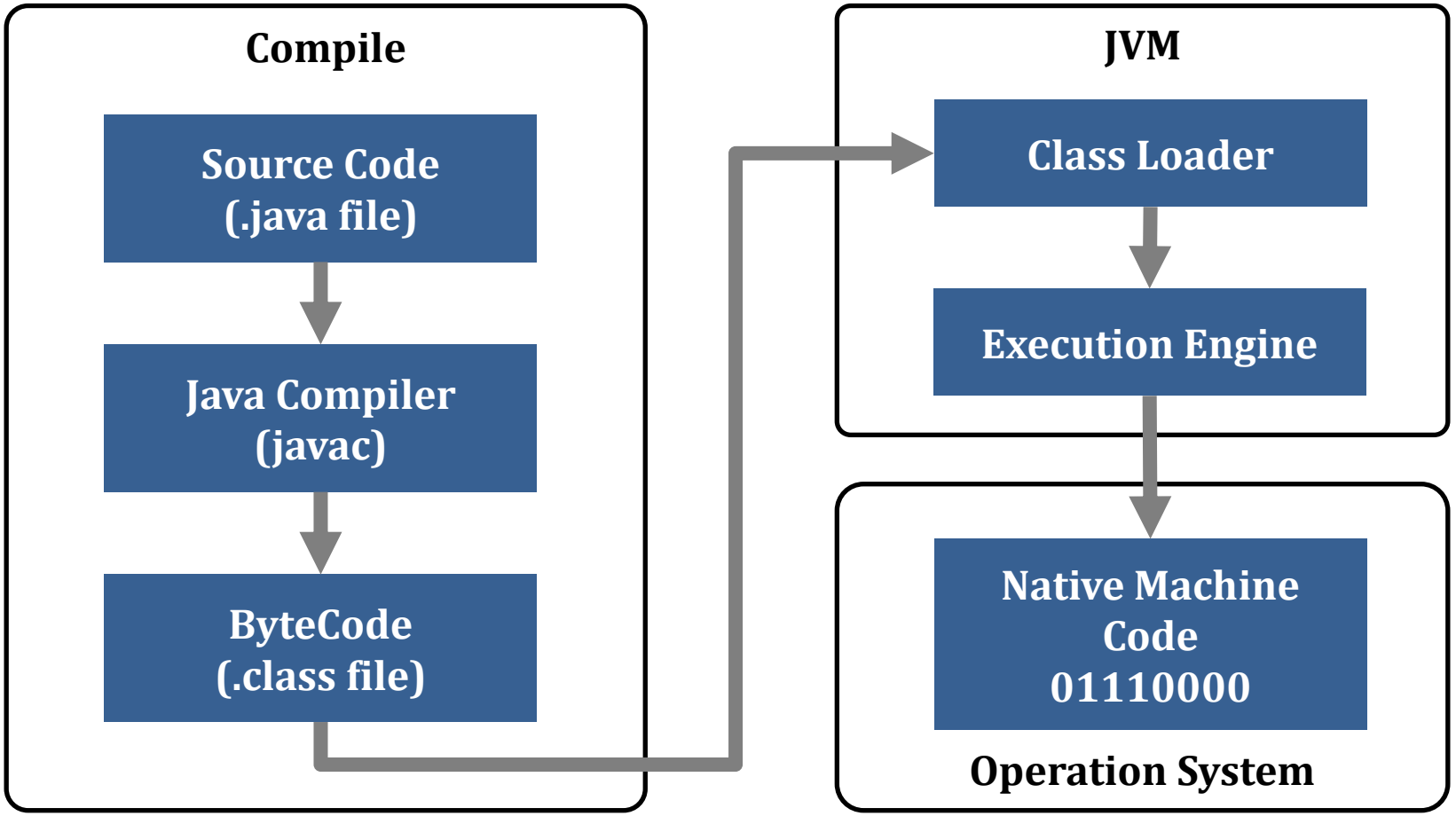
JAVA Memory

JAVA 기초

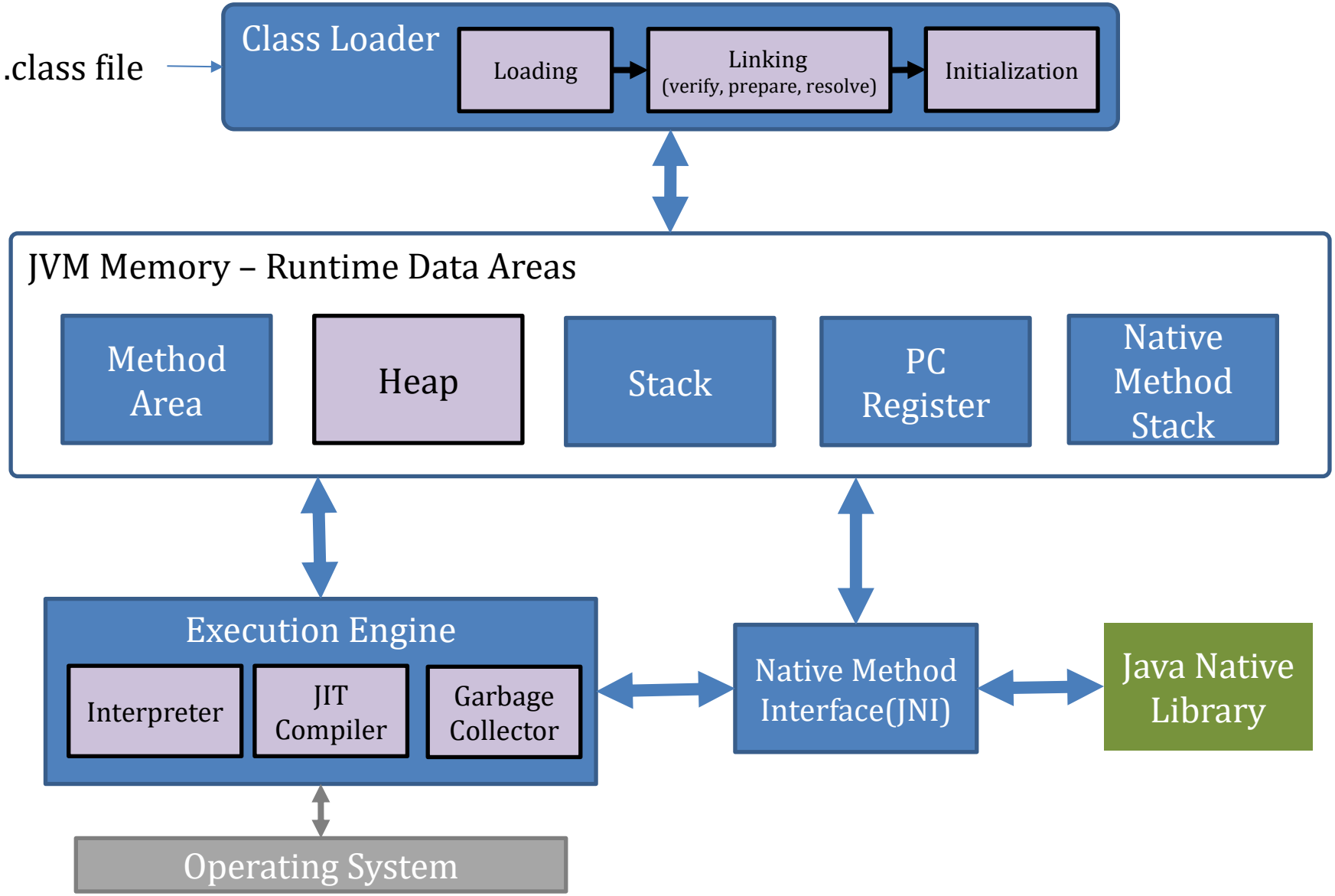
WORA(Write Once. Run Anywhere)



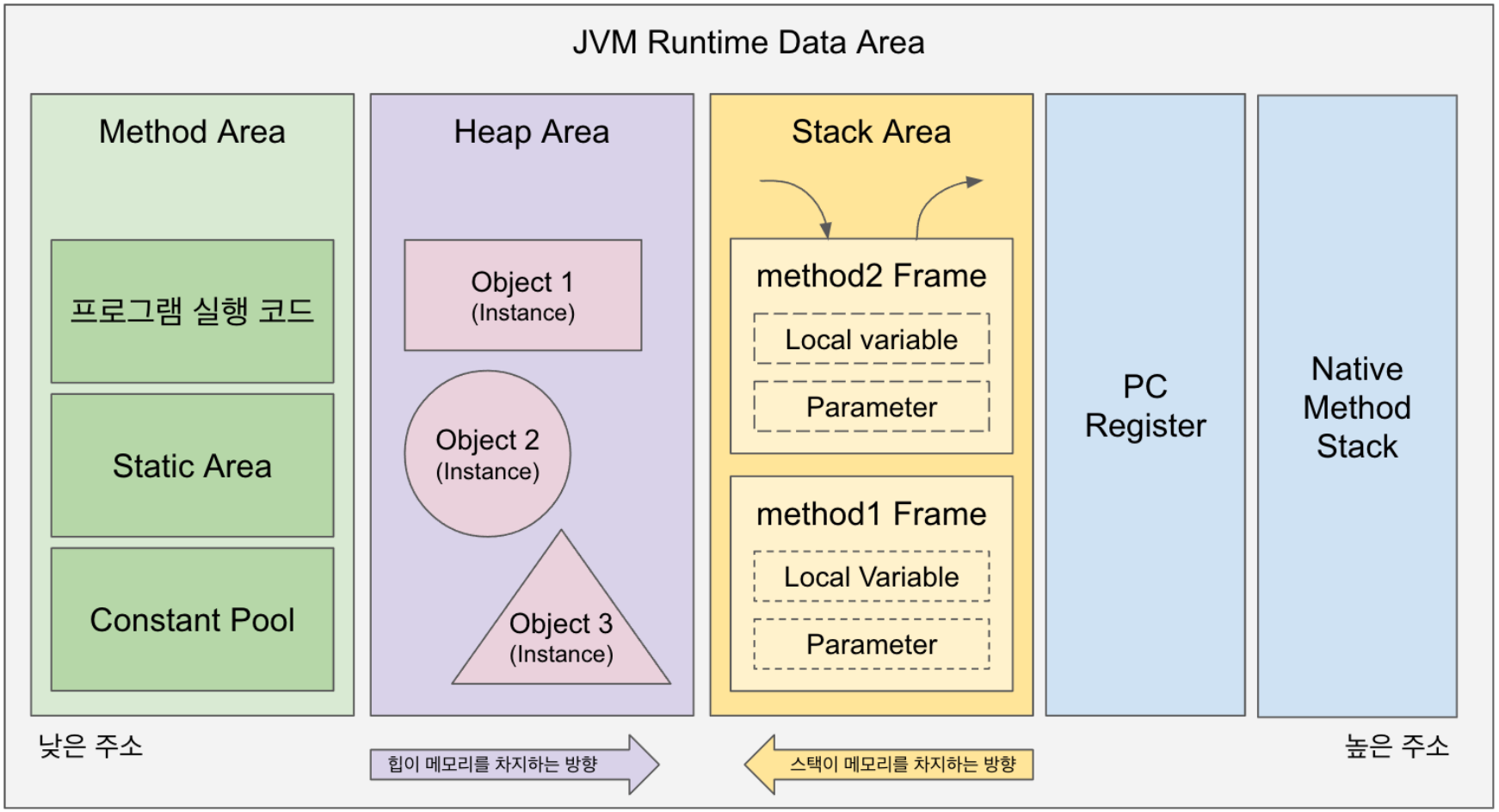
JVM과 ByteCode

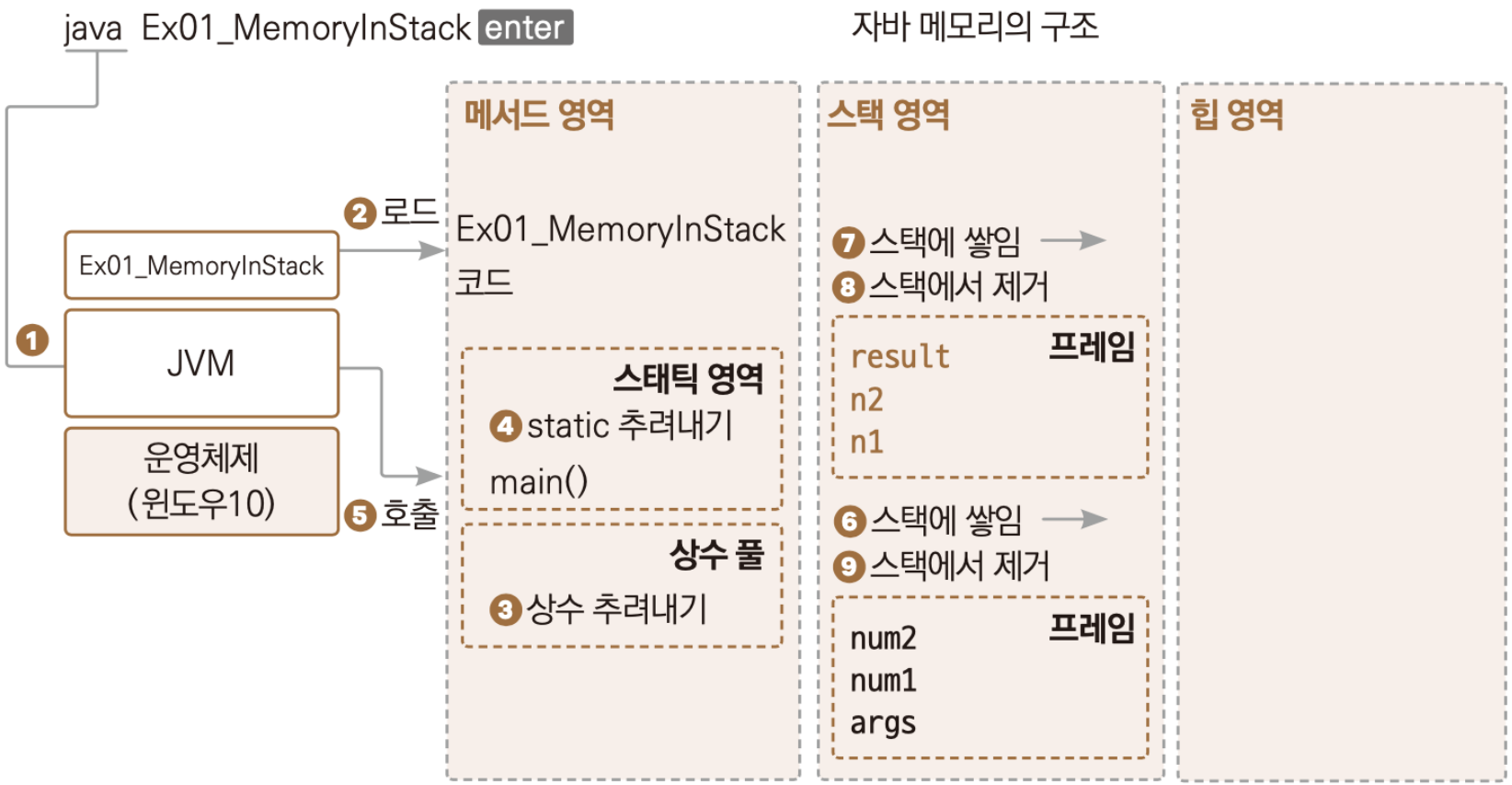


JVM(Java Virtual Machine)

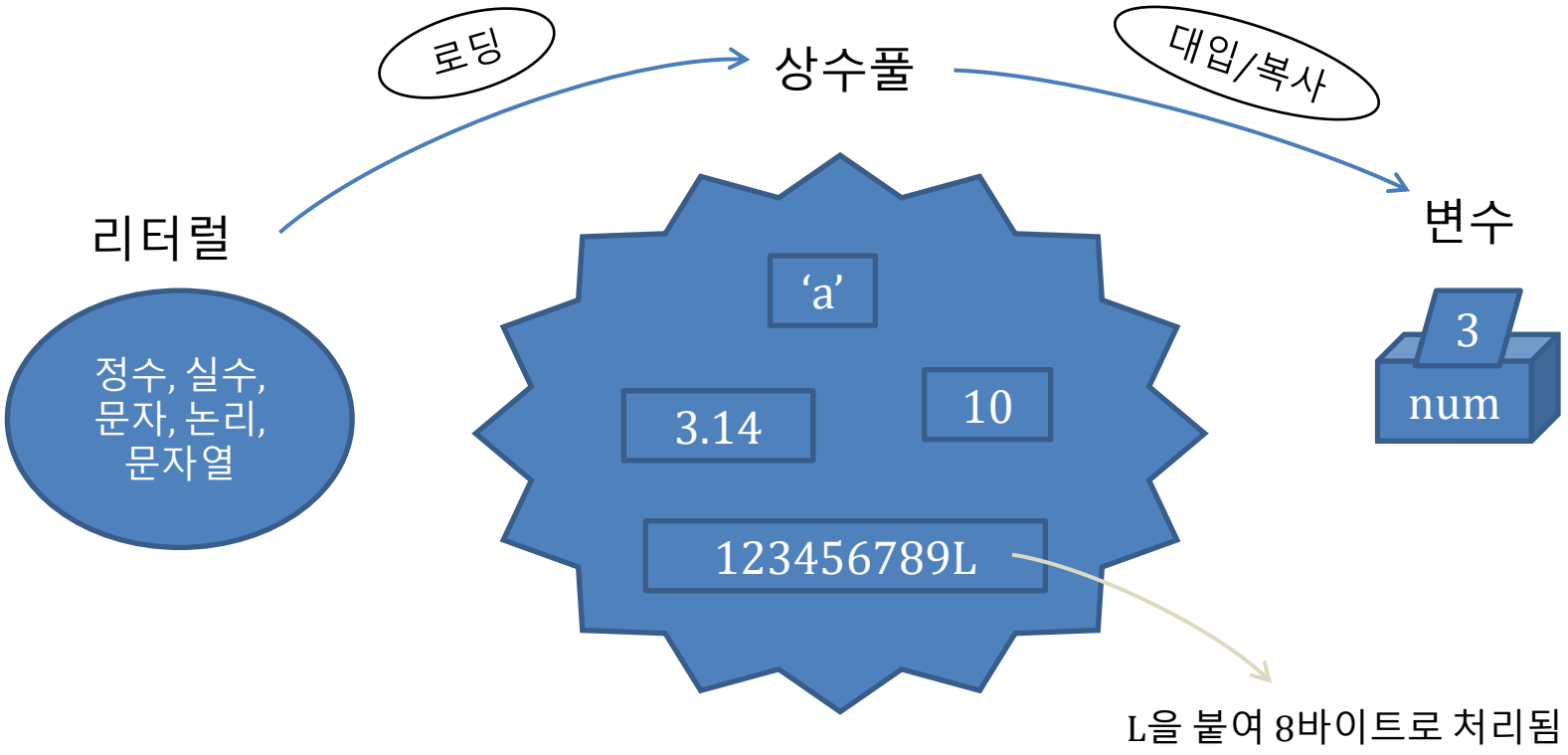


JVM Runtime Data Area




java Ex01_MemoryInStack enter
자바 메모리의 구조
메서드 영역
스택 영역
힙 영역
2 로드
Ex01_MemoryInStack
Ex01_MemoryInStack
코드
1
JVM
운영체제
(윈도우10)
5 호출
스태틱 영역
4 static 추려내기
main()
상수 풀
3 상수 추려내기
7 스택에 쌓임
8 스택에서 제거
프레임
result
n2
n1
6 스택에 쌓임
9 스택에서 제거
프레임
num2
num1
args

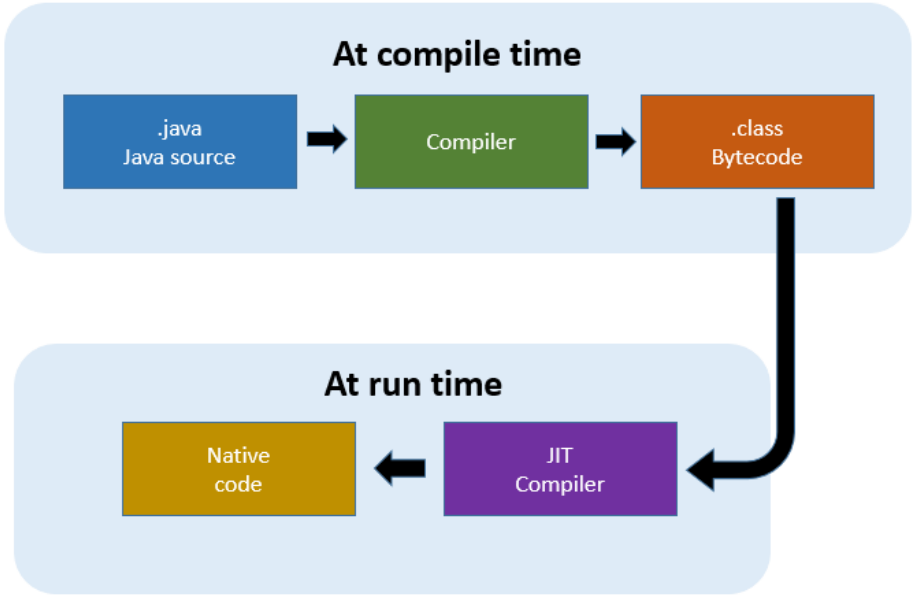
상수풀(Constant Pool)



- Class Loader
  - .class 파일을 찾아 메모리에 로드
  - 링크를 통해 Runtime Data Area에 배치
  - 런타임시 동적으로 클래스 로드
- Execution Engine
  - Runtime Area에 배치된 바이트코드 실행
- Runtime Data Area : OS상의 JVM메모리 영역
  - Method Area
    - 클래스 별 런타임 상수풀, 필드 데이터, 메소드 데이터, 메소드 코드, 생성자 코드
  - Heap Area
    - 객체와 배열이 생성되는 영역
    - 객체와 배열은 JVM스택 영역의 변수나 다른 객체의 필드에서 참조
    - new 연산자를 사용하여 인스턴스가 생성되면 해당 정보를 저장
    - 참조가 해제된 객체(dangling)는 Garbage Collector을 통해 자동으로 제거
  - Stack
    - 메소드 호출(push), 메소드 종료(pop)
    - 후입선출(LIFO, Last-In First-Out)
  - PC Register
    - 연산의 주소 값 저장
    - 현재 명령이 끝난 뒤에 값 증가
  - Native Method Stack
    - 자바의 바이트코드가 아닌 기계어로 작성된 프로그램 호출을 저장
    - JNI(Java Native Interface)를 통해 호출되는 C, C++ 등의 코드를 수행하기 위한 스택
    - JVM 스택에 쌓이다가 해당 메소드 내부에 네이티브 메소드가 있다면 네이티브 스택에 쌓인다

JIT(Just In Time) 컴파일

- 자바는 컴파일러와 인터프리터 모두 사용
- 자바의 느린 실행 요인은 인터프리터 방식으로 바이트 코드 실행
- JIT(Just in Time) 컴파일링 기법으로 개선
  - 실행 도중 바이트 코드를 해당 CPU의 기계어 코드로 컴파일, 해당 CPU가 기계어를 실행
  - 코드가 실행되는 과정에서 실시간으로 일어나며, 전체코드의 필요한 부분만 변환
  - 바이트코드를 기계어(*Native Code*)로 변환(반복되는 코드 컴파일)
  - 기계어로 변환된 코드는 캐시에 저장(재사용 시 컴파일을 다시 할 필요가 없다)



JAVA Memory

JAVA 기초

생성자(*contructor*)

- 생성자 이름은 클래스 이름과 동일
- 생성자는 여러 개 작성 가능(생성자 중복)
- 생성자는 객체 생성시 한 번만 호출 : *new* 연산자
- 생성자의 목적은 객체 생성 시 초기화
- 생성자는 리턴 타입을 지정할 수 없음
- 생성자는 메소드가 아님
- 하나의 클래스에는 반드시 적어도 하나 이상의 생성자가 존재
- 기본생성자(*default constructor*)
  - 프로그래머가 생성자를 기술하지 않으면 기본생성자가 자동으로 생김(컴파일러가 코드에 넣어 줌)
  - 매개변수와 구현부가 없음

생성자 오버로딩


```
package constructor;

public class Person {
    String name;
    float height;
    float weight;

    public Person( ) { }

    public Person(String pname) {
       name = pname;
    }

    public Person(String pname, float pheight, float pweight) {
       name = pname;
       height = pheight;
       weight = pweight;
    }
}
```

디폴트 생성자

이름을 매개변수로 입력받는 생성자

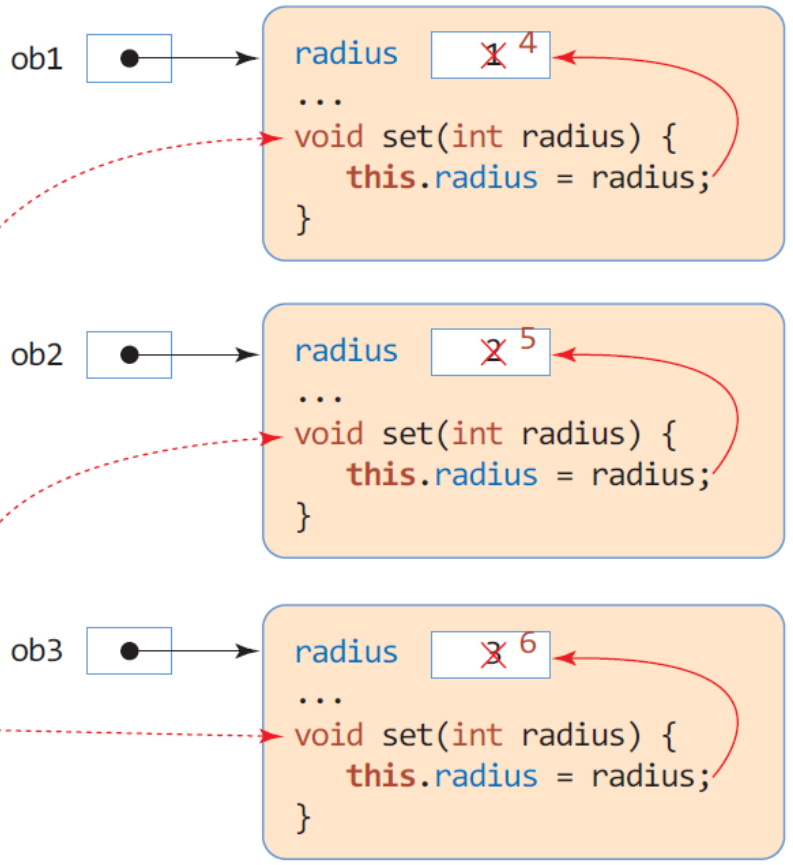
이름, 키, 몸무게를 매개변수로 입력받는 생성자

- 메소드 오버로딩 : 한 클래스 내에서 같은 이름이지만 다르게 작동하는 여러 메소드
- 메소드 오버라이딩 : 슈퍼 클래스의 메소드를 동일한 이름으로 서브 클래스마다 다르게 구현

this 레퍼런스

- 생성된 인스턴스 스스로를 가리키는 예약어
- this.멤버 형태로 멤버를 접근할 때 사용

```
public class Circle {
   int radius;
   public Circle(int radius) {
      this.radius = radius;
   }
   void set(int radius) {
      this.radius = radius;
   }

   public static void main(String[] args) {
      Circle ob1 = new Circle(1);
      Circle ob2 = new Circle(2);
      Circle ob3 = new Circle(3);

      ob1.set(4);
      ob2.set(5);
      ob3.set(6);
   }
}
```

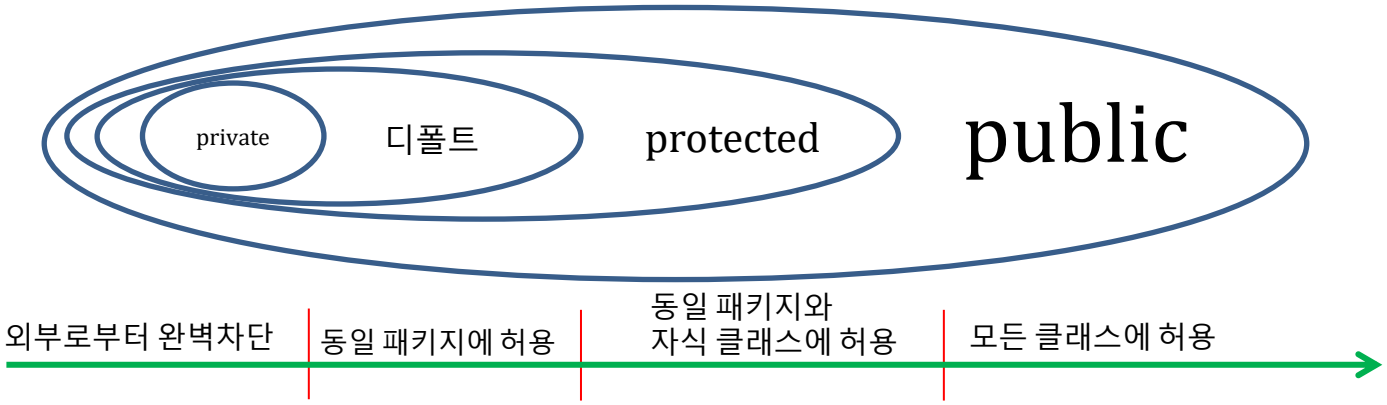


- **같은 클래스의 다른 생성자 호출**
- 생성자 내에서만 사용 가능
- 생성자 코드의 제일 처음에 있어야 함

```
public class Book {
    String title;
    String author;
    void show() { System.out.println(title + " " + author); }

    public Book() {
        this("", "");
    }

    public Book(String title) {
        this(title, "작자미상");
    }

    public Book(String title, String author) {
        this.title = title; this.author = author;
    }
    public static void main(String [] args) {
        Book emptyBook = new Book();
        loveStory.show();
    }
}
```

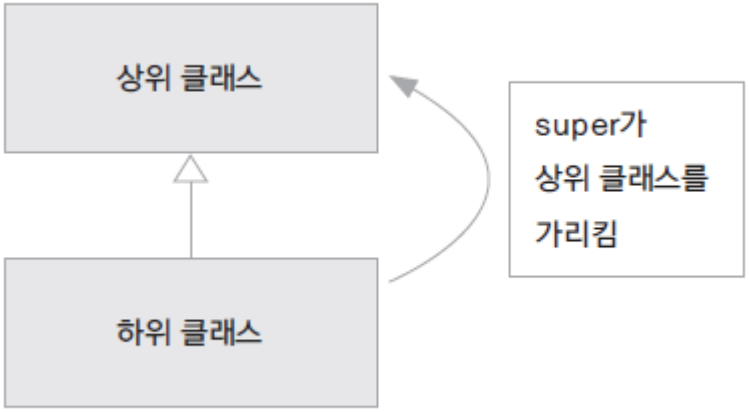
접근 지정자

- private
  - 동일 클래스 내에만 접근 허용
  - 상속 받은 서브 클래스에서 접근 불가
- protected
  - 같은 패키지 내의 다른 모든 클래스에게 접근 허용
  - 상속 받은 서브 클래스는 다른 패키지에 있어도 접근 가능
- public
  - 패키지에 관계 없이 모든 클래스에게 접근 허용
- 디폴트(접근지정자 생략)
  - 같은 패키지 내의 다른 클래스에게 접근 허용



super()

- this 가 자기 자신의 인스턴스의 주소를 가지는 것처럼
- super 는 하위 클래스가 상위 클래스에 대한 주소를 가지게 됨
- 하위 클래스가 상위 클래스에 접근 할 때 사용할 수 있음
- **컴파일러**는 super()를 자동으로 추가



super()


```
class A {
   public A() {
      System.out.println("생성자A");
   }
   public A(int x) {
      System.out.println("매개변수생성자A" + x);
   }
}
```

```
class B extends A {
   public B() {
      System.out.println("생성자B");
   }
   public B(int x) {
      super(x);  // 첫 줄에 와야 함
      System.out.println("매개변수생성자B" + x);
   }
}
```

```
public class ConstructorEx4 {
   public static void main(String[] args) {
      B b;
      b = new B(5);
   }
}
```

final 예약어

- final 변수는 값이 변경될 수 없는 상수임
  - public static final double PI = 3.14;
- final 변수는 오직 한 번만 값을 할 당할 수 있음
- final 메서드는 하위 클래스에서 재정의 (overriding) 할 수 없음
- final 클래스는 더 이상 상속되지 않음
  - java의 String 클래스
- 여러 자바 파일에서 공유하는 상수 값 정의 하기
  - 프로젝트 구현 시 여러 파일에서 공유해야 하는 상수 값은 하나의 파일에 선언하여 사용하면 편리 함

```
public class Define {
   public static final int MIN = 1;
   public static final int MAX = 99999;
   public static final int ENG = 1001;
   public static final int MATH = 2001;
   public static final double PI = 3.14;
   public static final String GOOD_MORNING = "Good Morning!";
}
```

```
public class UsingDefine {
   public static void main(String[] args) {
      System.out.println(Define.GOOD_MORNING);
      System.out.println("최솟값은 " + Define.MIN + "입니다.");
      System.out.println("최댓값은 " + Define.MAX + "입니다.");
      System.out.println("수학 과목 코드 값은 " + Define.MATH + "입니다.");
      System.out.println("영어 과목 코드 값은 " + Define.ENG + "입니다.");
   }
}
```

static으로 선언했으므로 인스턴스를 생성하지않고 클래스 이름으로 창조 가능

인터페이스

- 모든 메서드가 추상 메서드(abstract method)로 이루어진 클래스
- 형식적인 선언만 있고 구현은 없음
- 인터페이스에 선언된 모든 메서드는 public abstract 로 추상 메서드
- 인터페이스에 선언된 모든 변수는 public static final 로 상수
- 인터페이스 선언
  - **interface** 키워드로 선언
  - Ex) public **interface** SerialDriver {...}
- 자바 인터페이스에 대한 변화
  - Java 7까지
    - 인터페이스는 상수와 추상 메소드로만 구성
  - Java 8부터
    - 상수와 추상메소드 포함
    - default 메소드 포함 (Java 8)
    - private 메소드 포함 (Java 9)
    - static 메소드 포함 (Java 9)
  - 여전히 인터페이스에는 **필드(멤버 변수) 선언 불가**

인터페이스 만들기


```
interface PhoneInterface { // 인터페이스 선언
    public static final int TIMEOUT = 10000; // 상수 필드. public static final 생략 가능
    public abstract void sendCall();         // 추상 메소드. public abstract 생략 가능
    public abstract void receiveCall();      // 추상 메소드. public abstract 생략 가능
    public default void printLogo() {        // 디폴트 메소드는 public 생략 가능
        System.out.println("** Phone **");
    }; // 디폴트 메소드
}
```

```
public interface Calc {
    double PI = 3.14;
    int ERROR = -999999999;

    int add(int num1, int num2);
    int substract(int num1, int num2);
    int times(int num1, int num2);
    int divide(int num1, int num2);
}
```

인터페이스에서 선언한 변수는 컴파일 과정에서 상수로 변환됨

인터페이스에서 선언한 메서드는 컴파일 과정에서 추상 메서드로 변환됨

인터페이스의 구성요소들

- 상수
  - 모든 변수는 상수로 변환 됨. public만 허용, public static final 생략
- 추상 메소드
  - public abstract 생략 가능
- default 메소드
  - 인터페이스에 코드가 작성된 메소드
  - 인터페이스를 구현하는 클래스에 자동 상속. 구현 클래스에서 재정의 할 수 있음
  - public 접근 지정만 허용. 생략 가능
- private 메소드
  - 인터페이스 내에 메소드 코드가 작성되어야 함
  - 인터페이스 내에 있는 다른 메소드에 의해서만 호출 가능
- static 메소드
  - public, private 모두 지정 가능. 생략하면 public
- 인터페이스의 객체 생성 불가

오류

```
new PhoneInterface(); // 오류. 인터페이스 PhoneInterface 객체 생성 불가
```

- 인터페이스 타입의 레퍼런스 변수 선언 가능

```
PhoneInterface galaxy; // galaxy는 인터페이스에 대한 레퍼런스 변수
```

Object 클래스

- 모든 클래스의 최상위 클래스
- java.lang.Object 클래스
- 모든 클래스는 Object 클래스에서 상속 받음
- 모든 클래스는 Object 클래스의 메서드를 사용할 수 있음
- 모든 클래스는 Object 클래스의 메서드 중 일부는 재정의 할 수 있음 (final로 선언된 메서드는 재정의 할 수 없음)
- 컴파일러가 extends Object 를 추가 함

```
class Student {
   int studentID;
   String studentName;
}
```

코드를 작성할 때

```
class Student extends Object {
   int studentID;
   String studentName;
}
```

컴파일러가 변환

Object 클래스

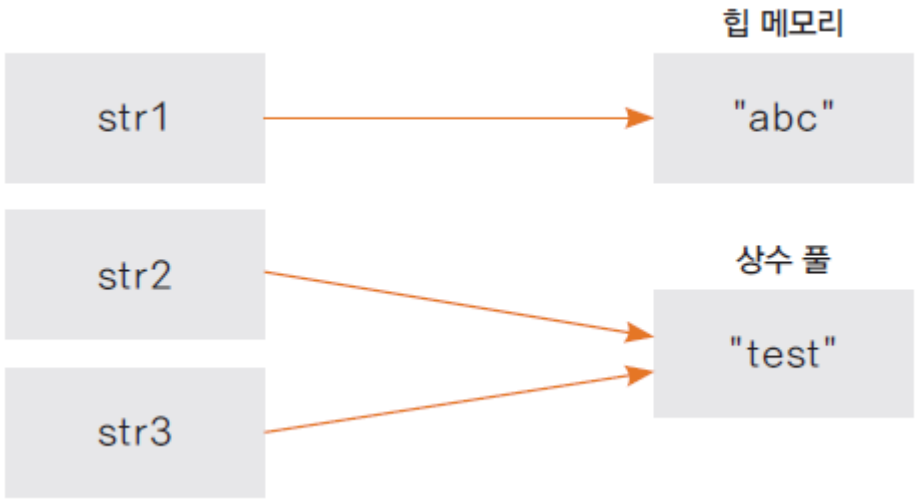
메서드	설명
String toString( )	객체를 문자열로 표현하여 반환합니다. 재정의하여 객체에 대한 설명이나 특정 멤버 변수 값을 반환합니다.
boolean equals(Object obj)	두 인스턴스가 동일한지 여부를 반환합니다. 재정의하여 논리적으로 동일한 인스턴스임을 정의할 수 있습니다.
int hashCode( )	객체의 해시 코드 값을 반환합니다.
Object clone( )	객체를 복제하여 동일한 멤버 변수 값을 가진 새로운 인스턴스를 생성합니다.
Class getClass( )	객체의 Class 클래스를 반환합니다.
void finalize( )	인스턴스가 힙 메모리에서 제거될 때 가비지 컬렉터(GC)에 의해 호출되는 메서드입니다. 네트워크 연결 해제, 열려 있는 파일 스트림 해제 등을 구현합니다.
void wait( )	멀티스레드 프로그램에서 사용하는 메서드입니다. 스레드를 '기다리는 상태'(non runnable)로 만듭니다.
void notify( )	wait( ) 메서드에 의해 기다리고 있는 스레드(nonrunnable 상태)를 실행 가능한 상태(runnable)로 가져옵니다.

String 클래스

- String을 선언하는 두 가지 방법

```
String str1 = new String("abc");  //생성자의 매개변수로 문자열 생성
String str2 = "test";             //문자열 상수를 가리키는 방식
```

- 힙 메모리에 인스턴스로 생성되는 경우와 상수 풀(constant pool)에 있는 주소를 참조하는 방법 두 가지



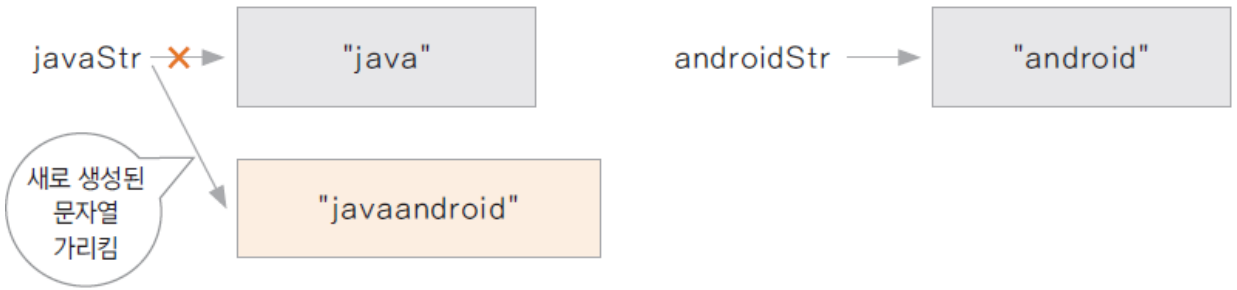
상수 풀의 문자열을 참조하면 모든 문자열이 같은 주소를 가리킴

String 클래스로 문자열 연결

- 한번 생성된 String 값(문자열)은 불변 (immutable)
- 두 개의 문자열을 연결하면 새로운 인스턴스가 생성 됨
- 문자열 연결을 계속하면 메모리에 garbage 가 많이 생길 수 있음

```
String javaStr = new String("java");
String androidStr = new String("android");
System.out.println(javaStr);
System.out.println("처음 문자열 주소 값: "+ System.identityHashCode(javaStr));

javaStr = javaStr.concat(androidStr);
```

문자열 javaStr와 문자열 androidStr를 연결하여 javaStr에 대입



StringBuilder, StringBuffer 사용하기

- 내부적으로 가변적인 char[] 배열을 가지고 있는 클래스
- 문자열을 여러 번 연결하거나 변경할 때 사용하면 유용함
- 매번 새로 생성하지 않고 기존 배열을 변경하므로 garbage가 생기지 않음
- StringBuffer는 멀티 쓰레드 프로그래밍에서 동기화(synchronization)을 보장
- 단일 쓰레드 프로그램에서는 StringBuilder를 사용하기를 권장
- toString() 메서드로 String 반환

```
public class StringBuilderTest {
 public static void main(String[ ] args) {
  String javaStr = new String("Java");
  System.out.println("javaStr 문자열 주소 :" + System.identityHashCode(javaStr));

  StringBuilder buffer = new StringBuilder(javaStr);
  System.out.println("연산 전 buffer 메모리 주소:" + System.identityHashCode(buffer));

  buffer.append(" and");
  buffer.append(" android");
  buffer.append(" programming is fun!!!");
  System.out.println("연산 후 buffer 메모리 주소:" + System.identityHashCode(buffer));

  javaStr = buffer.toString( );
  System.out.println(javaStr);
  System.out.println("연결된 javaStr 문자열 주소 :" + System.identityHashCode(javaStr));
 }
}
```

인스턴스가 처음 생성됐을 때 메모리 주소

String으로부터 StringBuilder생성

문자열 추가

String 클래스로 반환

```
<terminated> StringBuilderTest [Java Application] C:₩Progr
javaStr 문자열 주소 :385242642
연산 전 buffer 메모리 주소:824009085
연산 후 buffer 메모리 주소:824009085
Java and android programming is fun!!!
새로 만들어진 javaStr 문자열 주소 :2085857771
```