Java: ??? ??? vs new String() ?? ?? ??

---

1. ??

Java에서 문자열은 두 가지 방식으로 생성할 수 있다 :

```
String s1 = "hello";             // 리터럴 방식
String s2 = new String("hello"); // new 키워드 방식
```

두 방식은 겉보기에는 동일한 문자열 값을 가지지만 , 내부 메모리 구조 , 생성 방식 , 비교 결과 , 성능 등에서 중요한 차이를 가진다 .

---

2. ??? ?? ??

2.1 ??? ?? (`String s = "hello"`)

- 문자열 리터럴은 **String Constant Pool**이라는 특별한 메모리 영역에 저장된다 .
- 동일한 리터럴이 여러 번 사용되더라도 중복 없이 하나의 객체로 공유 됨 .
- JVM이 자동 최적화 해 메모리 사용이 매우 효율적이다 .

??

```
String a = "hello";
String b = "hello";

System.out.println(a == b);      // true (같은 객체 참조)
System.out.println(a.equals(b)); // true (내용도 동일)
```

→ 같은 상수 풀 객체 를 참조하고 있기 때문에 `a == b`는 `true`.

2.2 new ??? ?? (`String s = new String("hello")`)

- `new String()`은 무조건 Heap 메모리에 새로운 객체를 생성 한다 .
- 내부적으로 리터럴 `"hello"`를 참조하더라도 , 그 값을 복제한 별도 인스턴스 가 생성됨 .

??

```
String a = "hello";
String b = new String("hello");

System.out.println(a == b);      // false (다른 참조)
System.out.println(a.equals(b)); // true  (내용은 같음)
```

→ `==` 결과는 `false`, `.equals()`는 `true`

---

3. ?? ?? ??

3.1 `==` ??? (?? ??)

- 두 객체의 참조 주소값을 비교 함
- 리터럴 방식 끼리는 상수 풀 공유로 `true`일 수 있음
- `new`로 생성한 객체는 항상 `false`

3.2 `.equals()` ??? (? ??)

- 두 문자열의 내용이 같은지 비교함
- 방식과 무관하게 , 내용이 같으면 항상 `true`

```
String s1 = "test";
String s2 = new String("test");

System.out.println(s1 == s2);      // false
System.out.println(s1.equals(s2)); // true
```

---

4. ?? ? ??? ??

항목	리터럴 방식 ("hello")	new 방식 (new String("hello"))
메모리 위치	Constant Pool (Method Area)	Heap 영역
객체 중복 여부	없음 (공유됨)	항상 새 객체 생성
== 비교 결과	true (동일 참조)	false (서로 다른 참조)
.equals() 결과	true	true
메모리 효율	높음	낮음
성능	빠름	느림 (GC 대상 증가)
GC 부담	낮음	높음
사용 권장 여부	□ □□□□ 권장	□ □□□ 경우에만 사용

5. ?? ?? ? ??

5.1 ??? ?? – ????? ??

- 정적 값 , 설정 상수 , 조건 분기 등에 적합
- 성능과 메모리 효율이 우수함

```
if (userRole.equals("ADMIN")) {
    // 권한 체크
}
```

5.2 new ?? – ?? ???

- 동일한 문자열이더라도 서로 다른 객체를 명시적으로 생성 할 때
- 예 : 문자열 캐시 우회 , 디버깅용 비교 , 참조를 강제로 분리해야 할 경우
- 필요하다면 .intern() 메서드를 사용해 수동으로 상수 풀 등록 가능

```
String s = new String("hello").intern();  // Constant Pool 등록
```

6. ??: ??? vs new String()

비교 항목	리터럴 방식 ("hello")	new 방식 (new String("hello"))
저장 위치	String Constant Pool	Heap 메모리
객체 공유 여부	□ □□□ (중복 방지)	□ □□ 새 객체 생성
== 비교 결과	true	false
.equals() 결과	true	true
성능	빠름	느림
메모리 효율	높음	낮음
GC 부담	낮음	높음
사용 추천 여부	□ □□□ 상황에서 적극 권장	⚠ 특별한 목적이 없으면 지양

7. ??

- 문자열은 리터럴 방식으로 생성하는 것이 가장 효율적 이다 .
- new String() 방식은 불필요한 **Heap** 객체 생성 으로 인해 성능과 메모리 측면에서 불리하다 .
- 문자열 비교 시에는 항상 ==이 아닌 **.equals()** 를 사용할 것 .
- 고급 최적화 가 필요한 경우에는 .intern()으로 Constant Pool을 수동 활용할 수 있다 .

이 정리는 Java 메모리 구조 , 성능 최적화 , 참조와 객체 비교에 대해 정확한 이해를 돕기 위한 위키 스타일 자바에서 문자열은 두 가지 방법으로 생성할 수 있다 .

String str = "hello";와 String str = new String("hello"); 두 가지 방식은 같아 보이지만 , 작동 방식 , 메모리 구조 , 성능 , 객체 비교 등 여러 측면에서 차이가 난다 .

```
String s1 = "hello";                    // 리터럴 방식
String s2 = new String("hello");        // new 연산자 사용
```

두 방식은 겉보기에는 같은 값을 가지지만 , 메모리 처리 , 비교 방법 , 성능 등의 측면에서 분명한 차이가 존재한다 .

1. ??? ?? ??

1.1 ??? ??: String s1 = "hello";

문자열 리터럴은 String Constant Pool에 저장된다 .

JVM은 동일한 문자열 리터럴이 여러 번 등장하더라도 중복을 제거하여 하나의 객체만 저장한다 .

메모리 효율이 높고 , 비교 연산 시 빠르다 .

◫ 예시

```
String a = "hello";
String b = "hello";

System.out.println(a == b);  // true (같은 상수 풀 객체)
```

1.2 new ??? ??: String s2 = new String("hello");

new 키워드를 사용하면 Heap에 새로운 객체가 강제로 생성된다 .

비록 내부적으로 "hello"라는 리터럴이 사용되더라도 , Constant Pool의 객체를 복제한 별도 인스턴스가 Heap에 만들어진다 .

따라서 리터럴과는 다른 참조값을 가진다 .

◫ 예시

```
String a = "hello";
String b = new String("hello");

System.out.println(a == b);      // false (주소 다름)
System.out.println(a.equals(b)); // true  (값 같음)
```

2. ?? ?? (== vs equals())

2.1 == (?? ??)

리터럴 방식은 JVM이 Constant Pool을 공유하므로 == 결과가 true일 수 있다 .

new 키워드는 항상 새로운 객체를 생성하므로 == 비교 시 false가 된다 .

2.2 .equals() (?? ??)

두 방식 모두 문자열 내용을 같게 만들면 .equals()는 항상 true를 반환한다 .

문자열 비교에서는 항상 .equals()를 사용하는 것이 안전하다 .

3. ?? ? ??? ??

3.1 리터럴 방식

- 런타임 시 상수 풀을 이용하므로 메모리 사용이 효율적이다 .
- JVM이 자동으로 중복 문자열을 제거해주므로 불필요한 객체 생성을 방지할 수 있다 .
- 가장 선호되는 방식 .

3.2 new 방식

- new String()을 사용할 경우 매번 객체가 새로 만들어지므로 메모리 낭비 가능성이 있다 .
- GC(Garbage Collector) 부담 증가 → 성능 저하로 이어질 수 있음 .
- 특별한 이유 (예 : 강제로 다른 참조를 만들어야 할 필요)가 없다면 지양하는 것이 좋다

4. ?? ?? ? ???

4.1 리터럴 방식의 사용 예

- 상수 정의 , 비교가 필요한 경우
- 조건 분기 , 설정 값 체크 등에서 성능 이점 있음 .

```
if (userRole.equals("ADMIN")) { ... }
```

4.2 new 방식의 사용 예

- 객체 생성을 명시적으로 컨트롤해야 하는 특수한 경우
- 외부 API나 라이브러리에서 intern() 메서드를 사용하여 상수 풀로 다시 보낼 수도 있음

```
String s = new String("hello").intern();  // 풀로 보냄
```

5. ??

	리터럴 `""` ("hello")	**new** 방식 `new String("hello")`
메모리 위치	Constant Pool (Method Area)	Heap 영역
객체 중복 여부	없음 (공유됨)	항상 새 객체 생성
`==` 비교 결과	true (같은 참조)	false (다른 객체)
`.equals()` 결과	true	true
메모리 효율	높음	낮음
성능	빠름	느림
GC 부담	낮음	높음
사용 권장 여부	일반적으로 권장	특별한 경우에만 사용

6. ??

- 문자열을 선언할 때는 특별한 이유가 없다면 "hello"와 같은 리터럴 방식을 사용하는 것이 가장 효율적이다.
- new String()은 불필요한 객체 생성을 유발하며, 메모리와 성능 측면에서 손해를 볼 수 있다.
- 문자열 비교 시에는 항상 ==가 아닌 .equals()를 사용할 것.
- 최적화를 원할 경우 intern()을 활용해 상수 풀을 직접 사용할 수도 있다.

Revision #8
Created 20 May 2025 04:16:07 by Dain
Updated 20 May 2025 06:23:31 by Dain