

BIT BY BIT 4 – JAVA

String

String 클래스가 특별한 이유

1. 문자열 리터럴 사용해서 할당 가능

Ex. `String a = new String("ㅎㅇ")` vs `String a = "ㅎㅇ";`

String 클래스가 특별한 이유

1. 문자열 리터럴 사용해서 할당 가능

Ex. String a = new String("ㅎㅇ") vs String a = "ㅎㅇ";

||

매번 새로운 객체 생성 vs 문자열 풀 사용

String 클래스가 특별한 이유

1. 문자열 리터럴 사용해서 할당 가능

Ex. String a = new String("ㅎㅇ") vs String a = "ㅎㅇ";

||

매번 새로운 객체 생성 vs 문자열 풀 사용

문자열 풀 사용이란, 클래스 로딩 시 즉, 런타임 시점에서 앞으로 종종 사용될 문자열이라고 판단하여 특별한 힙 메모리 구역(문자열 풀)에 String 객체를 만들어놓고 재사용하는 것

String 클래스가 특별한 이유

1. 문자열 리터럴 사용해서 할당 가능

Ex. String a = new String("ㅎㅇ") vs String a = "ㅎㅇ";

||

매번 새로운 객체 생성 vs 문자열 풀 사용

문자열 풀 사용이란, 클래스 로딩 시 즉, 런타임 시점에서 앞으로 종종 사용될 문자열이라고 판단하여 특별한 힙 메모리 구역(문자열 풀)에 String 객체를 만들어놓고 재사용하는 것

⇒ 같은 문자열 리터럴을 사용해서 객체를 선언하는 경우 물리적으로 동일한 객체를 재사용함.

String 클래스가 특별한 이유

1. 문자열 리터럴 사용해서 할당 가능

Ex. String a = new String("ㅎㅇ") vs String a = "ㅎㅇ";

||

매번 새로운 객체 생성 vs 문자열 풀 사용

문자열 풀 사용이란, 클래스 로딩 시 즉, 런타임 시점에서 앞으로 종종 사용될 문자열이라고 판단하여 특별한 힙 메모리 구역(문자열 풀)에 String 객체를 만들어놓고 재사용하는 것

⇒ 같은 문자열 리터럴을 사용해서 객체를 선언하는 경우 물리적으로 동일한 객체를 재사용함.

⇒ 불변객체로 String 클래스를 설계해야 사이드 이펙트 방지 가능

String 클래스가 특별한 이유

1. 문자열 리터럴 사용해서 할당 가능

Ex. String a = new String("ㅎㅇ") vs String a = "ㅎㅇ";

||

매번 새로운 객체 생성 vs 문자열 풀 사용

문자열 풀 사용이란, 클래스 로딩 시 즉, 런타임 시점에서 앞으로 종종 사용될 문자열이라고 판단하여 특별한 힙 메모리 구역(문자열 풀)에 String 객체를 만들어놓고 재사용하는 것

⇒ 같은 문자열 리터럴을 사용해서 객체를 선언하는 경우 물리적으로 동일한 객체를 재사용함.

⇒ 불변객체로 String 클래스를 설계해야 사이드 이펙트 방지 가능

2. + 연산도 사용 가능, 클래스 내부 메서드인 concat을 사용하지 않아도 됨

String 이 불변객체라 계산 과정에서 **새로운 객체**를 매번 반환 => **문자열 많이 생김**

그래서 StringBuilder라는 가변 객체가 만들어지긴 했지만,,

요샌 jvm이 적당히 String 연산 해도 StringBuilder를 활용해서 알아서 최적화 시켜줌.

But, 아래 같은 경우에는 **루프 밖 범위의 최적화**가 필요하므로 우리가 직접 StringBuilder를 적극 사용 해줘야함



```
String result = "";  
for (int i = 0; i < 100000; i++) {  
    result = new StringBuilder().append(result).append("Hello Java  
").toString();  
}
```


반복문의 루프 내부에서는 최적화가 되는 것 처럼 보이지만, 반복 횟수만큼 객체를 생성해야 한다.

String 이 불변객체라 계산 과정에서 **새로운 객체**를 매번 반환 => **문자열 많이 생김**

그래서 StringBuilder라는 가변 객체가 만들어지긴 했지만,,

요샌 jvm이 적당히 String 연산 해도 StringBuilder를 활용해서 알아서 최적화 시켜줌.

But, 아래 같은 경우에는 **루프 밖 범위의 최적화**가 필요하므로 우리가 직접 StringBuilder를 적극 사용 해줘야함



```
String result = "";
```

```
for (int i = 0; i < 100000; i++) {  
    result += i; // This line is highlighted in the original image  
}  
result.toString();
```

반복문의 루프 내부에서는 최적화가 되는 것 처럼 보이지만, 반복 횟수만큼 객체를 생성해야 한다.

바깥 범위에서 StringBuilder 사용하고 포문 돌고 결과만 String 객체로 변환!

10 Java

StringBuffer 찍먹하기

StringBuffer 찹먹하기

구분	불변성	스레드 안전성	성능 (단일 스레드 기준)	사용 용도
String	불변	✔ 안전	✖ 느림	변경이 거의 없는 문자열 상수, 키 등
StringBuffer	가변	✔ 안전	✖ 느림	멀티스레드 환경에서 문자열 수정 시
StringBuilder	가변	✖ 비안전	✔ 빠름	단일 스레드에서 문자열을 자주 수정할 때

StringBuffer 찍먹하기

구분	불변성	스레드 안전성	성능 (단일 스레드 기준)	사용 용도
String	불변	✅ 안전	❌ 느림	변경이 거의 없는 문자열 상수, 키 등
StringBuffer	가변	✅ 안전	❌ 느림	멀티스레드 환경에서 문자열 수정 시
StringBuilder	가변	❌ 비안전	✅ 빠름	단일 스레드에서 문자열을 자주 수정할 때

StringBuffer 찍먹하기

구분	불변성	스레드 안전성	성능 (단일 스레드 기준)	사용 용도
String	불변	✅ 안전	❌ 느림	변경이 거의 없는 문자열 상수, 키 등
StringBuffer	가변	✅ 안전	❌ 느림	멀티스레드 환경에서 문자열 수정 시
StringBuilder	가변	❌ 비안전	✅ 빠름	단일 스레드에서 문자열을 자주 수정할 때

❌ 스레드 안전하지 않은 코드:

java

📋 복사 ✎ 편집

```
StringBuilder sb = new StringBuilder();
sb.append("Hello");
```

- 만약 두 개의 스레드가 동시에 `.append("World")`, `.append("Java")` 를 호출하면
→ 누가 먼저 처리할지 예측 불가, 결과 문자열이 이상하게 섞일 수도 있음
예: HelloWorldJava, HelloJavaWorld, 혹은 깨진 문자열

StringBuffer 는,

Synchronized, Lock 등등을 사용해서
스레드 안전하게 객체를 바꿈