Brincando com o `==` e com o *pool* de Strings

O Java utiliza um mecanismo chamado *String interning*, colocando as Strings num *pool* para tentar armazenar apenas uma cópia de cada sequência de caracteres em memória.
Quando o Java encontra literais String no código, ele retorna sempre uma mesma instância de String, que aponta para uma entrada no *pool* interno da JVM. Sendo assim, é bem possível usar o operador `==` para comparar duas variáveis que recebem literais String:

```
String literal = "str";
String outraLiteral = "str";

System.out.println(literal == outraLiteral); //exibe true
```

Inclusive, como o Java trata literais String como instâncias é possível comparar um literal diretamente, assim:

```
System.out.println(literal == "str"); //também retorna true
```

Por outro lado, não podemos confiar no operador de comparação quando não sabemos como a String foi criada, já que é possível criar outras instâncias de várias formas. Exemplo:

```
String novaInstancia = new String("str");
System.out.println("str" == novaInstancia); //retorna false
```

O código acima cria uma nova instância de String, que não é a mesma retornada pela JVM para o literal `"str"`.
Mas, contudo, todavia, entretanto, isso não quer dizer que temos duas entradas de `"str"` no *pool* do Java. Como podemos verificar isso? Usando o método `String.intern()`, que retorna uma referência para a String que está no *pool*. Exemplo:

```
String novaInstancia = new String("str");
System.out.println("str" == novaInstancia.intern()); //retorna true
```

Aplicando isso para no exemplo da pergunta, teríamos:

```
public class TesteString {
	public static void main(String[] args) {

		String str1 = "teste";
		String str2 = "Oteste".substring(1);

		System.out.println("str1: " + str1 + ", str2: " + str2);
		if(str1 == str2.intern()) {
			System.out.println("str1 igual a str2");
		}
		else {
			System.out.println("str1 diferente de str2");
		}
	}
}
```

E o resultado:

```
str1: teste, str2: teste
str1 igual a str2
```

Tudo muito interessante. Mas, e se criassemos uma String de uma forma mirabolante?

```
StringBuilder sb = new StringBuilder();
sb.append('s');
sb.append('t');
sb.append('r');
System.out.println("str" == sb.toString().intern()); //continua sendo true
```

Os objetos da classe `String` têm uma particularidade interessante. A JVM guarda um *pool de Strings*, onde ele armazena as Strings que passaram no seu código, para evitar ter que ficar carregando Strings repetidas, mas como ele funciona?
Uma `String` irá para o *pool* se você instanciar a `String` de maneira literal, assim:

```
String str1 = "text";
```

O valor `"text"` agora está armazenado no *pool*
Por outro lado se você instanciar com a palavra-chave **new**, o valor utilizado não será o do *pool*, mesmo que seja igual.

```
String str2 = new String("text");
```

O que pode ser visto com um simples teste.

```
System.out.println(str1 == str2); // Imprime false
```

Agora se você quiser que a `String` seja a mesma do *pool* pode usar o método `intern()`, a documentação dele diz o seguinte:

> *Returns a canonical representation for the string object. A pool of strings, initially empty, is maintained privately by the class String.*
> *When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.*

Ou seja, se já existir um objeto com o valor da `String` no *pool*, ele é retornado, se não existir, esse valor é adicionado lá, e a referência que foi adicionada é retornada. E podemos comprovar com outro teste simples.

```
System.out.println(str1 == str2.intern());
```

Mas e o `equals()`?

Se a comparação com `==` é mais rápida do que o método `equals()`, devemos abandonar o `equals()` e usar o `intern()` em todo lugar? A resposta é **não**.
Nem todas as Strings são internalizadas no *pool* imediatamente. Quando chamamos o método `intern()`, se ela não estiver lá, então o Java irá acrescentá-la. O problema é que uma vez no *pool* a String vai para a memória permanente e não será mais coletada pelo *garbage collector*.
Quando se quer velocidade e o conjunto de valores é relativamente pequeno, usar o método `intern()` pode ser vantajoso. Mas se usarmos este recurso, por exemplo, para processamento de arquivos-texto, XML, bancos de dados, logo veremos um `OutOfMemoryError`.

Além disso, adicionar uma Strings no `pool` também pode ser uma operação “cara”. Além de ser necessário verificar se a String já existe, o Java provavelmente terá que tratar acessos concorrentes.
E, finalmente, uma grande desvantagem é o código ficar mais propenso a bugs (*error prone*), já que é preciso que o desenvolvedor sempre coloque o `intern()` quando necessário.

Outras formas de comparação

Indo um pouco além da comparação exata de Strings, temos outras formas interessantes de comparação:

Case insensitive (sem considerar maiúsculas e minúsculas)

```
System.out.println("STR".equalsIgnoreCase("str")); //retorna true
```

Uma string contida em outra

```
System.out.println("###STR###".contains("STR")); //retorna true
```

Qual string é "maior" que a outra?

```
System.out.println("str1".compareTo("str2")); //retorna -1, pois "str1" é menor que "str2"
```

Ou:

```
System.out.println("str1".compareToIgnoreCase("STR2")); //retorna -1, ignorando a capitalização
```

O método `compareTo` retorna:

- `1` se a primeira String for maior que a segunda
- `0` se forem iguais
- `-1` se a primeira String for menor que a segunda

Começa com...

```
System.out.println("str1".startsWith("str")); //returna true, pois "str1" começa com "str"
```

Termina com...

```
System.out.println("str1".endsWith("r1")); //return true, pois "str1" termina com "r1"
```

Expressão regular

```
System.out.println("str2".matches("\\w{3}\\d")); //return true, pois corresponde à expressão regular
```

Está vazia?

```
String str1 = "";
System.out.println(str1.isEmpty());
System.out.println(str1.length() == 0);
System.out.println(str1.equals(""));
```

Particularmente eu prefiro o primeiro método para Java >= 6 e o segundo para as versões anteriores.