Comparing references

1. Comparisons

Programmers need to compare different variables with one another all the time. But, as you have already seen, everything is not so simple.

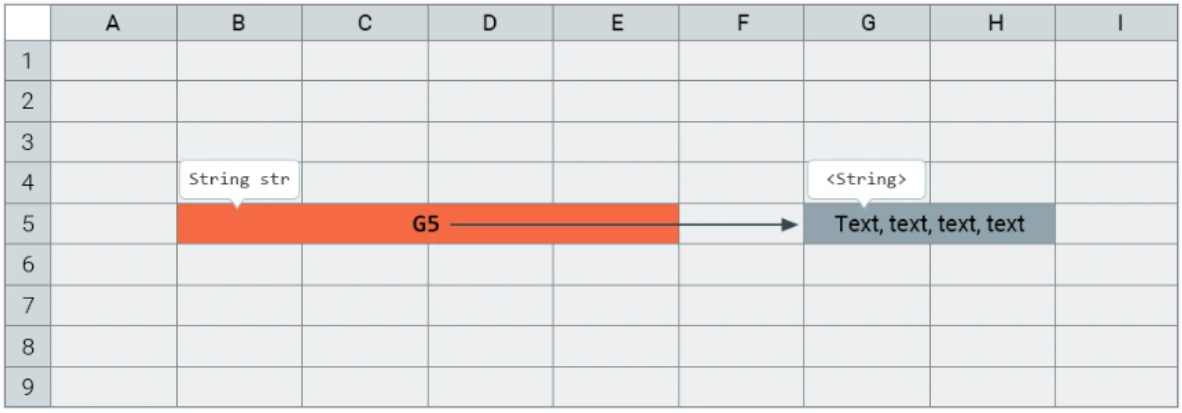
Integers are very easy to compare — you just use == and you're done. To compare *real numbers*, you have to compare their difference (or rather, the absolute value of the difference) with some very small number.

Comparing *strings* is even more difficult. Above all, this is because strings are objects. What's more, programmers often want the string comparison to go a little differently depending on the situation.

---

2. How strings are arranged memory

As you have already seen, strings are stored in memory differently than integers and real numbers:



Two blocks of memory are used to store strings: one block stores the text itself (its size depends on the size of the text) while the second block (4 bytes) stores the address of the first block.

Although an experienced programmer would say something like "the `String str` variable stores a reference to a `String` object.

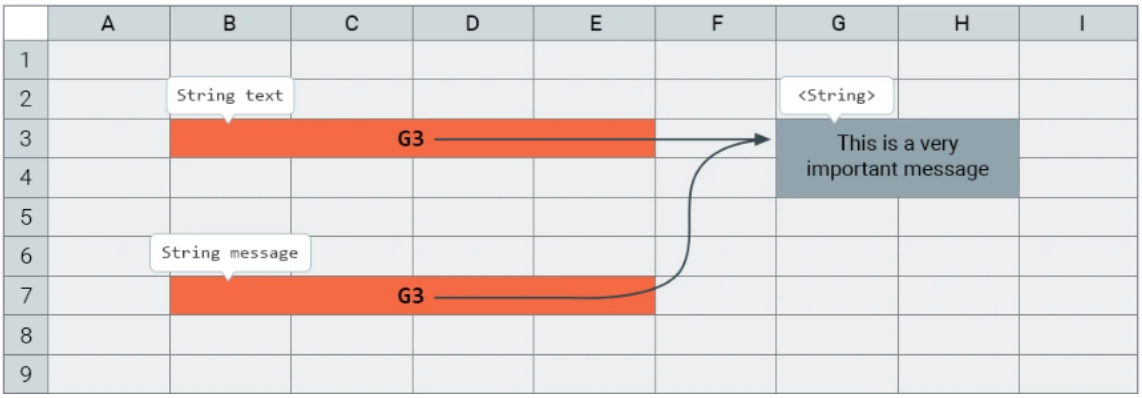
---

3. Assigning references to a string

The benefits of this approach become apparent when you need to assign one string variable to another string variable. Example:

```
String text = "This is a very important message";
String message = text;
```

And here's what memory will contain as a result:



After this type of assignment operation, the `String` object remains where it was, and just its address (a reference to the object) is copied into the `message` variable.

---

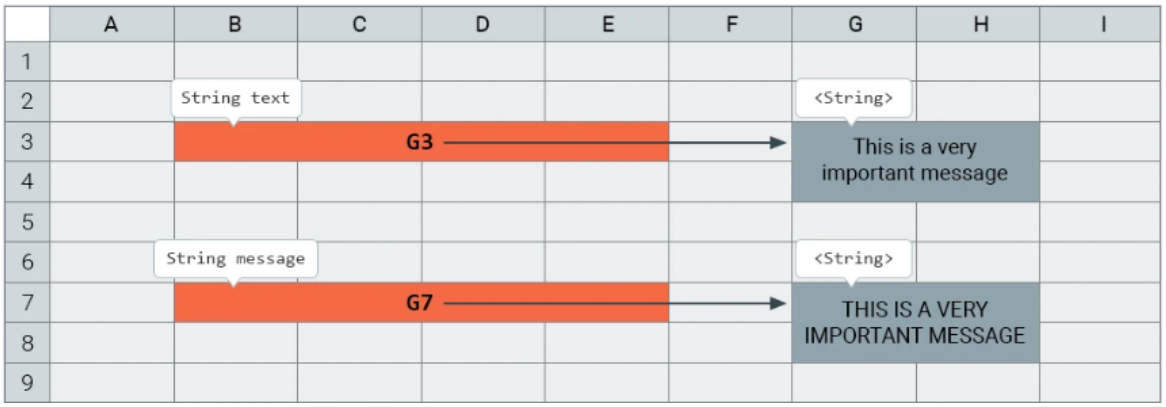
4. Working with references and objects

But if you decide to convert a string to uppercase (capital letters), the Java machine does everything right: you will end up with two `String` objects, and the `text` and `message` variables will store references, each to its own object.

Example:

```
String text = "This is a very important message";
String message = text.toUpperCase();
```

And here's what memory will contain as a result:



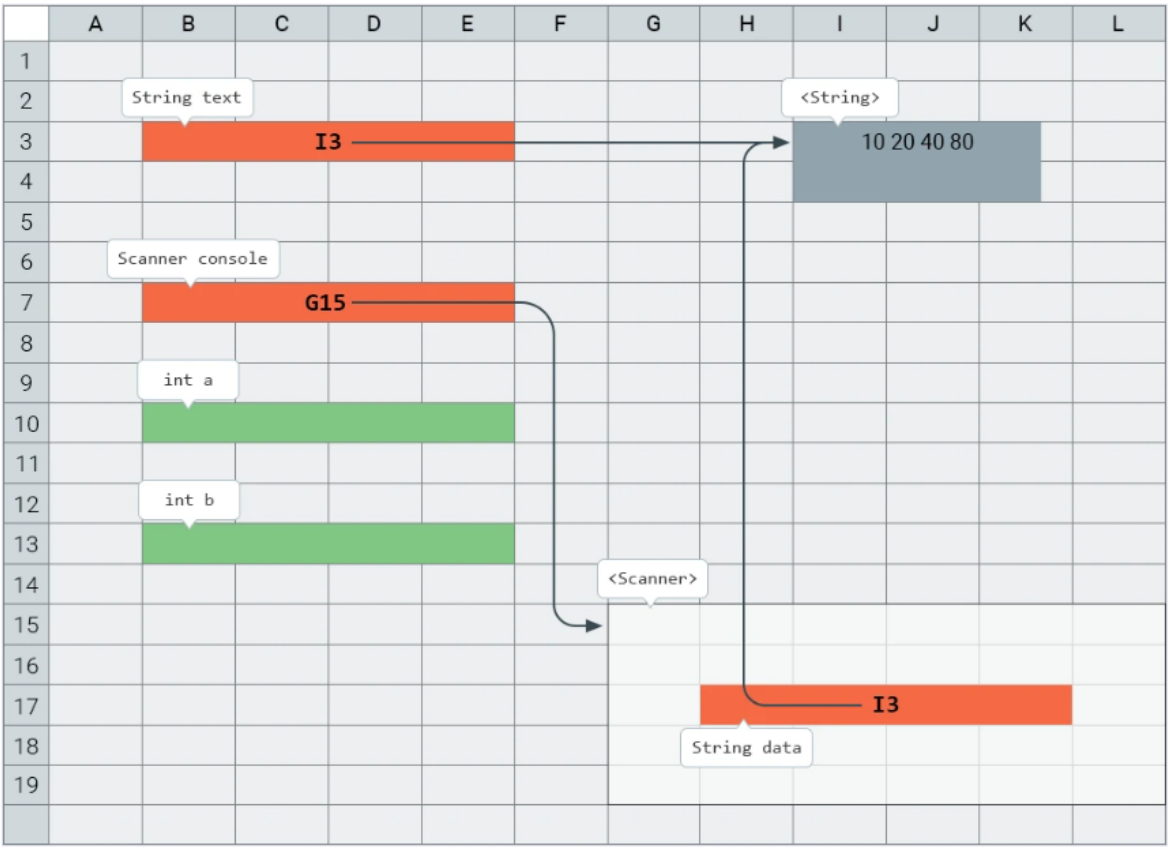
Please note that the `toUpperCase()` method does not change the string it is called on. Instead, it creates a new string (new object) and returns a reference to it.

How about an even more interesting example. Let's say you decide to pass a string to a `Scanner` object (so that it reads values from the string).

Example:

```
String text = "10 20 40 80";
Scanner console = new Scanner(text);
int a = console.nextInt();
int b = console.nextInt();
```

This is how it will all be stored in memory:



In this case, a single `String` object remains in memory just as it was — only references to it are passed around and stored in variables.

---

5. Comparing references to `String` objects

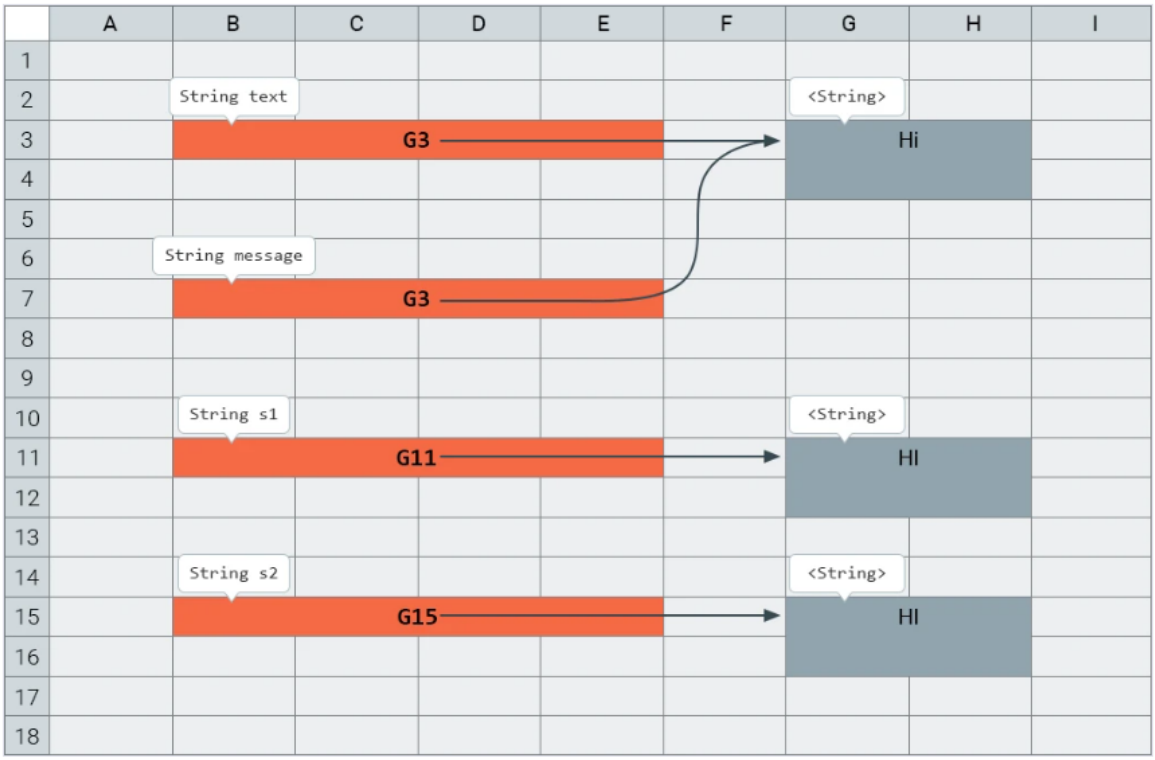
And finally, we've reached the fun part: string comparison.

There are two operators you can use to compare string variables: `==` (equal) and `!=` (not equal). You can't use the "greater than", "less than", or "greater than or equal to" operators — the compiler won't allow it.

But there's an interesting nuance here: what actually gets stored in string variables? That's right: addresses (references) to objects. And it is these addresses that will be compared:

```
String text = "Hi";
String message = text;
String s1 = text.toUpperCase();
String s2 = text.toUpperCase();
```

Here's what will be in memory:



The `message` and `text` variables refer to (store the address of) the same object. But the `s1` and `s2` variables store references to objects that are very similar but distinct.

And if you compare these 4 variables in the code, then you get the following result:

Code	Console output

`String text = "Hi";`
`String message = text;`
`String s1 = text.toUpperCase();`
`String s2 = text.toUpperCase();`
`System.out.println(text == message);`
`System.out.println(text == s1);`
`System.out.println(s1 == s2);`
	
	`true  // The addresses are equal`
	`false // The addresses are different`
	`false // The addresses are different`