Theory: Primitive and reference types

8 minutes 4 / 5 problems solved

Start practicing

In Java, all data types are separated into two groups: **primitive types** and **reference types.**

Java provides only eight primitive types. They are built-in in the language syntax as keywords. The names of all primitive types are lowercase. The most commonly used type is `int` which represents an integer number.

```
int num = 100;
```

The number of reference types is huge and constantly growing. A programmer can even create their own type and use it like standard types. The most frequently used reference types are `String`, `Scanner` and arrays. Remember that Java, like most programming languages, is case sensitive.
In this topic, we will focus on `String`, which is a common example of the reference type.

The new keyword

In most cases, an object of a reference type can be created using the `new` keyword. When we use the `new` keyword, the memory is allocated for the object we create. That is called instantiation of the object because we create an instance of it. Then we initialize the variable by assigning some value to it. Often, as in our example, it is done with one line.

```
String language = new String("java"); //instantiation of String and initialization with "java"
```

You can also use a literal for strings:

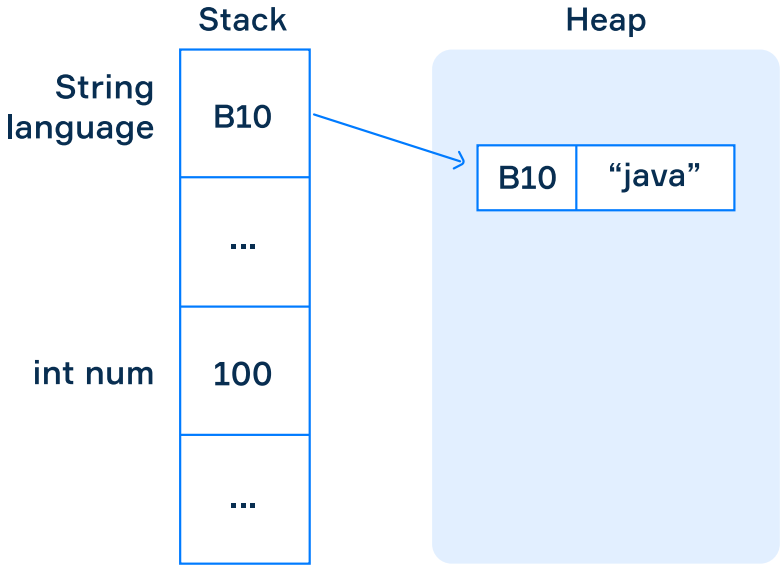
```
String language = "java";
```

The first approach with the keyword new is common for reference types, while the second is only string-specific. Both approaches give us the same result for strings but they have some technical differences which we will not consider here.

The main difference

The basic difference is that a variable of a primitive type stores the actual values, whereas a variable of a reference type stores an address in memory (reference) where the data is located. The data can be presented as a complex structure that includes other data types as their parts.

The following picture simply demonstrates this difference. There are two main memory spaces: **stack** and **heap**. All values of primitive types are stored in stack memory, but variables of reference types store addresses on objects located in heap memory.



We will not consider stack and heap in detail here. Just remember this difference between primitive and reference types.

Assignment

The way to store data also affects the mechanism to assign the value of a variable to another variable:

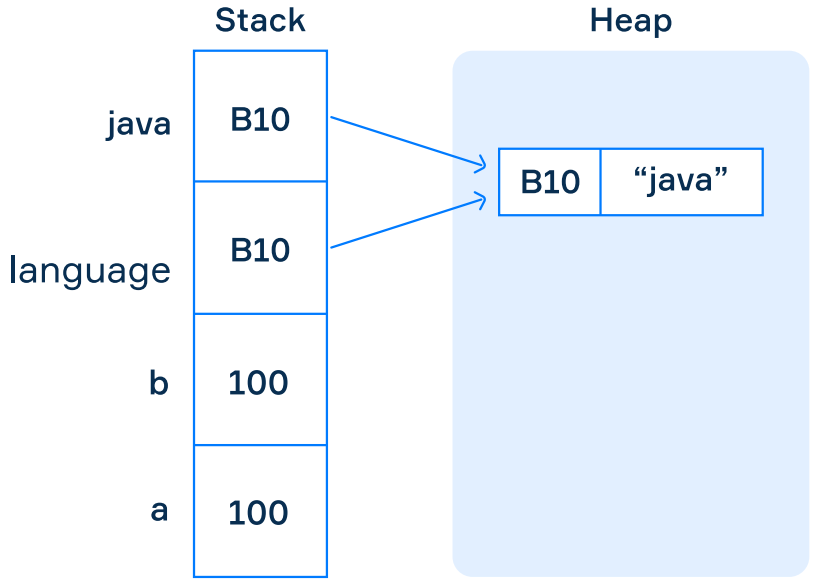
- primitive types: the value is just copied;
- reference types: the address to the value is copied (the data is shared between several variables).

Here is a snippet of code and a picture that demonstrates this.

```
int a = 100;
int b = a; // 100 is copied to b

String language = new String("java");
String java = language;
```

The variable `b` has a copy of value stored in the variable `a`. But variables `language` and `java` reference to the same value, rather than copy it. The picture below clearly demonstrates the difference.



Just remember, when assigning one value of a reference variable to another, we just make a copy of a reference rather than the value itself.

Comparisons

Comparing reference types using `==` and `!=` is not the same as comparing primitive types. Actually, when you are comparing two variables of the `String` type, it compares references (addresses) rather than actual values.

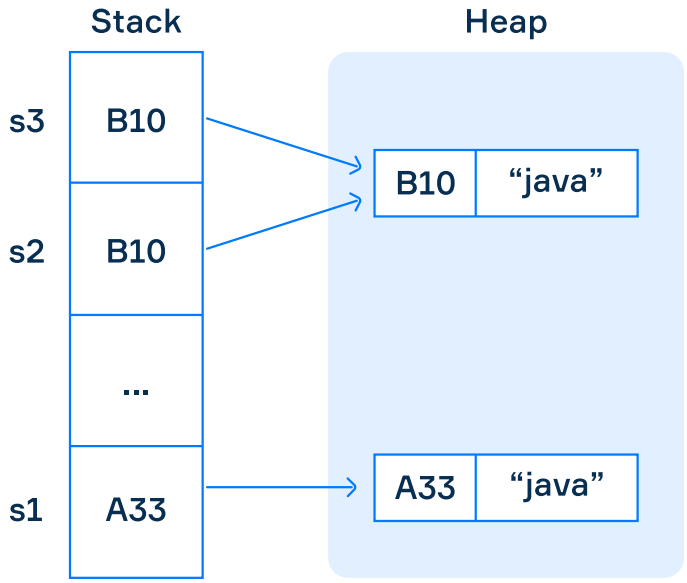
The following code demonstrates it:

```
String s1 = new String("java");

String s2 = new String("java");
String s3 = s2;

System.out.println(s1 == s2); // false
System.out.println(s2 == s3); // true
```

The picture below demonstrates this effect:



So, you do not need to use comparison operators when you want to compare the values. The correct way to compare content is to invoke the special method `equals`.

```
String s1 = new String("java");
String s2 = new String("java");
String s3 = s2;

System.out.println(s1.equals(s2)); // true
System.out.println(s2.equals(s3)); // true
```

The null type

Unlike primitive types, a variable of a reference type can refer to a special `null` value that represents the fact that it is not initialized yet or doesn't have a value.

```
String str = null;
System.out.println(str); // null
str = "hello";
System.out.println(str); // hello
```

The following statement with a primitive type won't compile.

```
int n = null; // it won't compile
```

Unfortunately, the frequent use of the `null` value can easily lead to errors in the program and complicate the code. Try to avoid `null` whenever it is possible, only use it if you really need it.