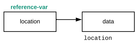
02 - Strings and Reference Types

Reference Types

A **reference** type is a variable whose name contains the **location of the memory allocated for the data rather than the data itself**.

String, *arrays* and *objects* are example of reference types



A **primitive type** is a variable whose name has the data assigned to it.

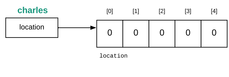
int, **double**, **boolean**, **char**, **float**, **long** (i.e. all standard Java data types) are example of primitive types

Examples:

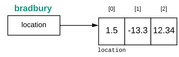
```
int sum = 0;
```

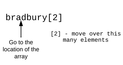

```
boolean shouldLoop = true;
```


```
int[] charles = new int[5];
```




```
double[] bradbury = {1.5, -13.3, 12.34 };
```



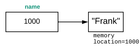


Text

Text

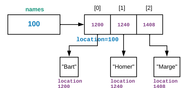
Text

```
String name = "Frank";
```

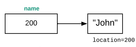



```
String godFather = "Don Vito Corleone";
```

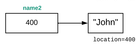

```
String[] names = {"Bart", "Homer", "Marge"};
```




```
String name = "John";
```




```
String name2 = "John";
```

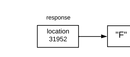


`if (name == name2)` **== compares the locations of the Strings - not the content of the Strings**

`if (name.equals(name2))` **use `stringname.equals(other-String)` to compare contents**

`.equals` is a method provided by String

`if (response == "F")` ---> false
locations are compared so are **unequal**



`if (response.equals("F"))` ---> true
contents are compared so are **equal**

`if (answer == 1)` ---> true

Heap vs Stack

Stack - a memory area assigned to every process/app running in an operating system. the default size of the stack is determined by the language and the operating system the size of the stack can be changed *(but we don't show you how to do that)*

The Stack contains **non-reference variables**, **arguments for functions**, **return values, literals**, the **location part of a reference variable** and any **other temporary storage areas Java might need**.

Since the Stack is fixed in size when a process starts, it could become full and not be able hold anymore data. When that happens you get a Stack Overflow error.

Java has mechanism to clean up data in the Stack that is no longer needed in the process called "**Garbage Collection**".

When a Java program enters a block, primitive variables are created in the Stack. When the programs leaves the block the variables are removed from the Stack. **This is why you cannot reference variables outside the block in which they are defined.**

The concept of **Scope** is used to describe when a variable is available and accessible to a program statement.

A variable is in scope for block in which it is defined and any block that block is in.

The Heap - Any memory in the machine not already allocated to program code, stacks or operating system stuff.

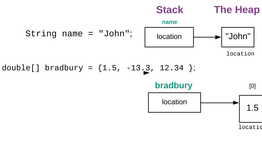
Programmers and programming languages are responsible for managing memory in The Heap.

If memory in The Heap is no longer needed and its not released by the programmer or language, it cannot be reused == **Memory Leak**

Memory Leaks waste memory, slows down processing, generally not good.

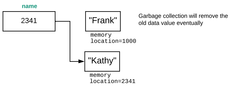
Java **Garbage Collection** handles this for us.

Reference variable data is placed in The Heap.



Strings are **immutable** - Value cannot be changed once it is assigned.
If a **new value** is assigned to a String, a **new String is created**

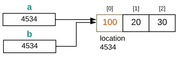
```
String name = "Frank";
```




```
name = "Kathy";
```

Assigning an array to another array does NOT copy the data from the array

```
int[] a = {10, 20, 30};
```



Assigning a reference to a reference copies the location of the data NOT the data itself

```
int[] b = a;

// define an array b
// assign the value in a


// a[0] is 10
// b[0] is 10
// because a and b both hold a reference to same location


b[0] = 100;  // change element 0 in b to 100
             // a[0] and b[0] reference the same element

// copy all the elements in array a to a new array c using for-loop
int[] c = new int[a.length];  // define an array the same size as array a
for(int i=0; i < a.length; i++) {
  c[i] = a[i];
}

// copy all the elements in array a to a new array c using Arrays.copyOf()
int[] c = Arrays.copyOf(a, a.length);
```