String Tokenizer

The String Tokenizer class allows an application to break a string into tokens.

Let us consider a string as, ***"JAVA PYTHON SQL AI"***

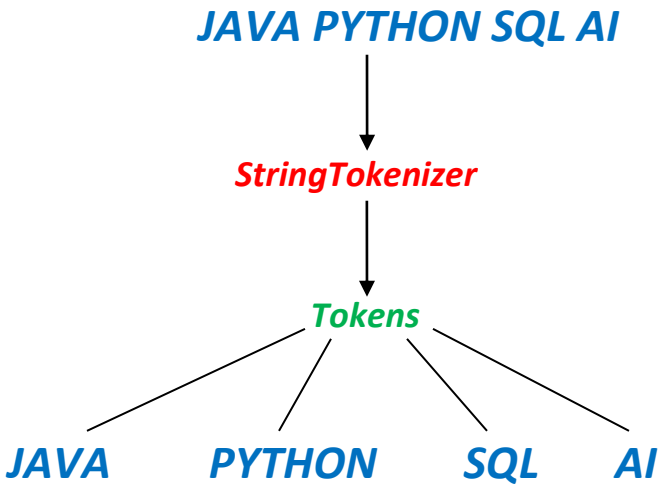
Now from the above string assume we have to parse wherever there are spaces to create smaller strings.

That is, these are the smaller strings that has to be created: ***JAVA***
PYTHON
SQL
AI

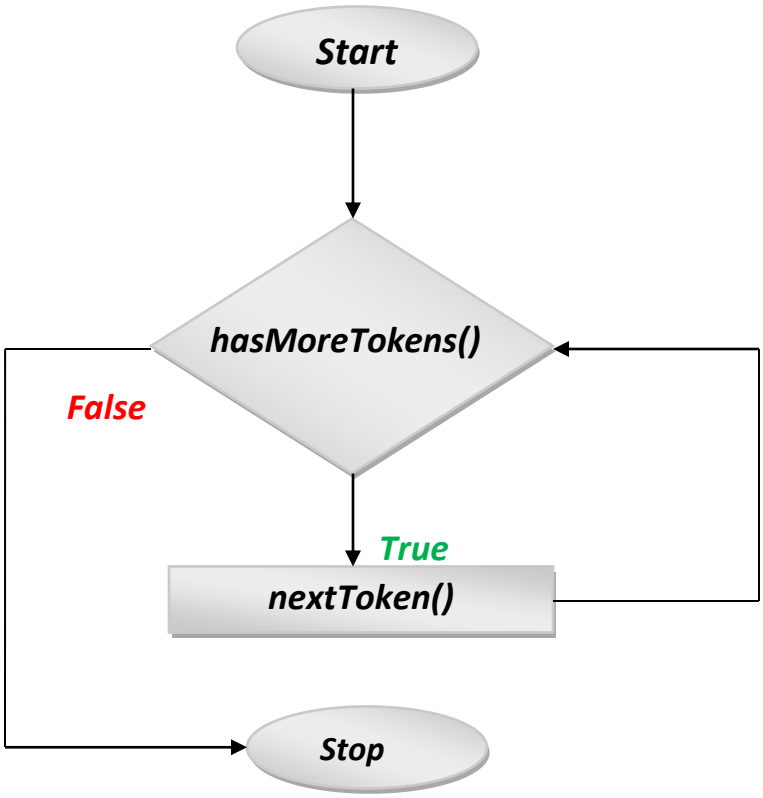
All these smaller strings which are created after parsing a string are only called as ***"Tokens".***

Now you might be wondering how to achieve this.

To achieve this we have to make use of a built in class called ***StringTokenizer****.*
StringTokenizer class provides a means of converting text strings into individual tokens.



Let us look at the flowchart of how it works.



Things to be known about StringTokenizer:

Delimiters *are the characters that separate tokens. The set of delimiters may be specified either at creation time or on a per-token basis.*

Object of StringTokenizer can be created in three different ways:

- ***StringTokenizer st = new StringTokenizer(String str);***

str *is the string to be tokenized. This case, considers the default delimiter as space.*

- ***StringTokenizer st = new StringTokenizer(String str, String delim);***

delim *is the set of delimiters that are used to tokenize the given string.*

- ***StringTokenizer st = new StringTokenizer(String str, String delim, boolean flag);***

The flag serves following purpose:

1) If the flag is false, delimiter characters serve to separate tokens.

2) If the flag is true, delimiter characters are considered to be tokens.

Case-1 Example:

```
import java.util.StringTokenizer;
class Demo
{
    public static void main(String[] args)
    {
        String s = "JAVA PYTHON SQL AI";
        StringTokenizer st = new StringTokenizer(s);
        while(st.hasMoreTokens() == true)
        {
            System.out.println(st.nextToken());
        }
    }
}
```

Case-1: here, the default delimiter is considered to be space.

Output:

JAVA
PYTHON
SQL
AI

Case-2 Example:

```
import java.util.StringTokenizer;
class Demo
{
    public static void main(String[] args)
    {
        String s = "JAVA PYTHON SQL AI";
        StringTokenizer st = new StringTokenizer(s," ");
        while(st.hasMoreTokens() == true)
        {
            System.out.println(st.nextToken());
        }
    }
}
```

Case-2: here, the delimiter is specified as space.

Output:

JAVA
PYTHON
SQL
AI

Case-3 Example 1:

Case-3: when flag is false, delimiter characters serve to separate tokens.

```
import java.util.StringTokenizer;
class Demo
{
    public static void main(String[] args)
    {
        String s = "JAVA PYTHON SQL AI";
        StringTokenizer st = new StringTokenizer(s," ",false);
        while(st.hasMoreTokens())
        {
            System.out.println(st.nextToken());
        }
    }
}
```

Output:

JAVA
PYTHON
SQL
AI

Case-3 Example 2:

Case-3: when flag is true, delimiter characters are considered to be tokens.

```
import java.util.StringTokenizer;
class Demo
{
    public static void main(String[] args)
    {
        String s = "JAVA PYTHON SQL AI";
        StringTokenizer st = new StringTokenizer(s," ",true);
        while(st.hasMoreTokens())
        {
            System.out.println(st.nextToken());
        }
    }
}
```

Output:

JAVA

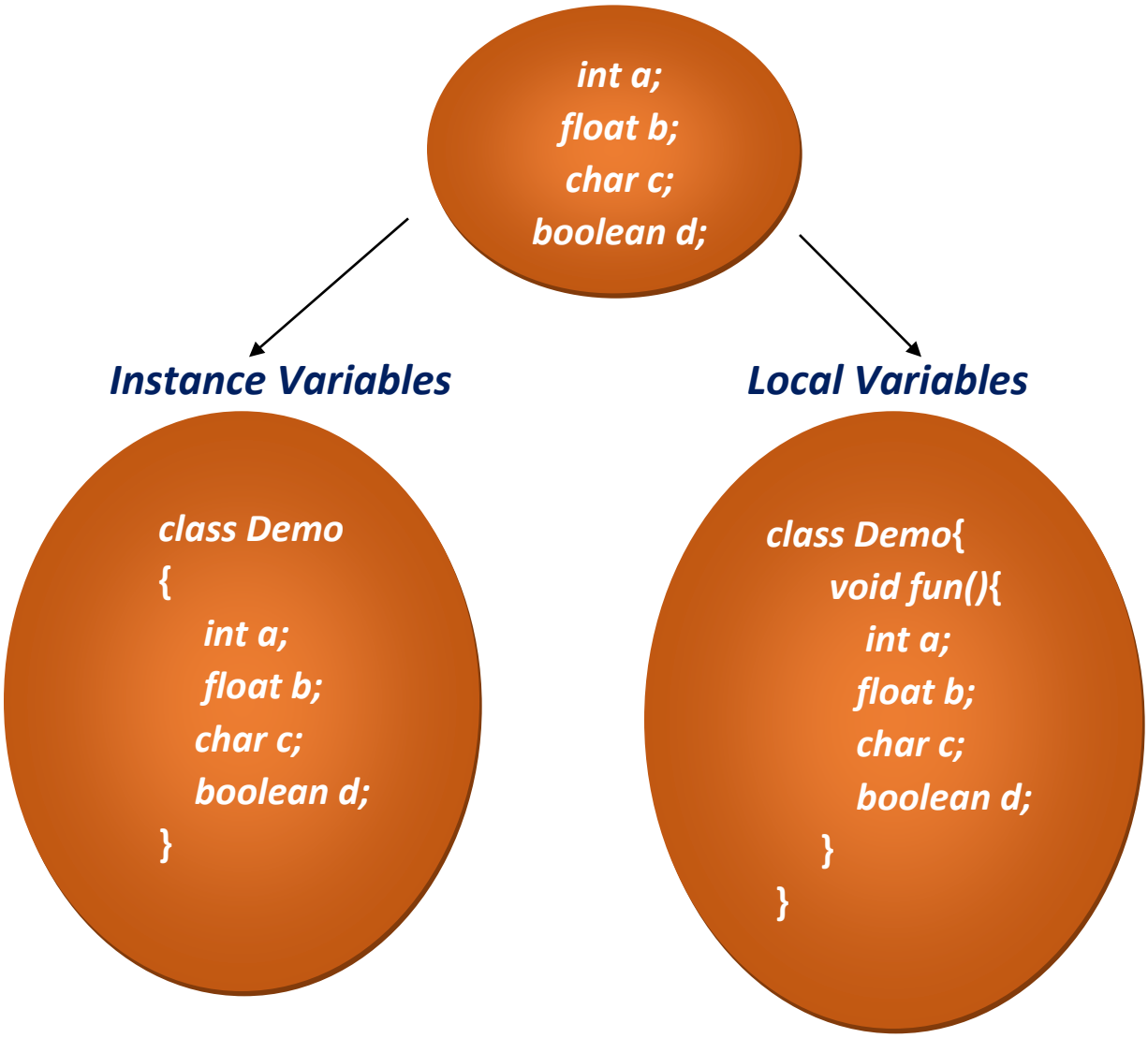
PYTHON

SQL

AI

Variables

A variable is the name given to the memory location. The value stored in a variable can be changed during program execution. Variables are of two types:



Instance variables are non-static variables and are declared in class but outside the body of the method.

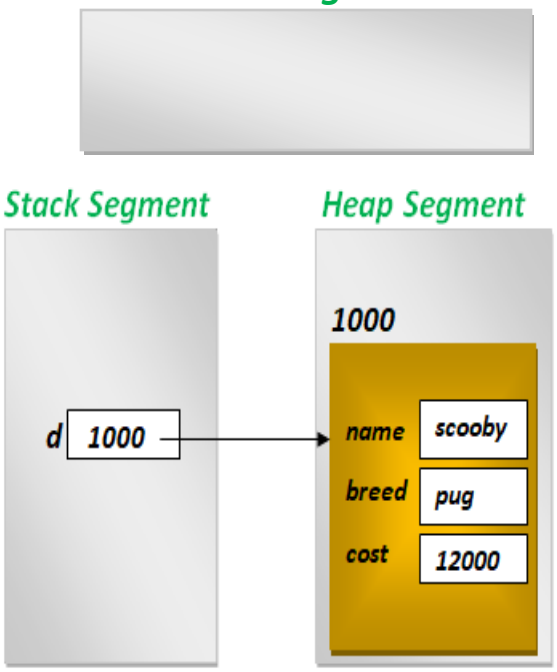
Local variables are declared inside the body of the method. We can use this variable only within that method.

Let us learn more about Instance variable through an example:

Code Segment

```
class Dog
{
    String name;
    String breed;
    int cost;
}
class Demo
{
    public static void main(String[] args)
    {
        Dog d = new Dog();
        d.name = "scooby";
        d.breed = "pug";
        d.cost = 12000;
        System.out.println(d.name);
        System.out.println(d.breed);
        System.out.println(d.cost);
    }
}
```

Static Segment



*Here, we have created three instance variables namely **name, breed and cost** inside the **class Dog.** In the Demo class, by using keyword "**new**" JVM has created an object of Dog class and allocates memory for it in the **heap segment** which has an address assuming to be 1000.*
Now JVM assigns automatically default values for the instance variables.
*Default values for type **String is null** and for **int is 0.**Memory for reference variable d is allocated in the stack segment and it starts refereeing to the object in heap segment. By making use of the reference variables **d.name, d.breed and d.cost** we are assigning values to them.*

Output:

(Before assigning values to instance variables)

null
null
0

Output:

(After assigning values)

scooby
pug
12000

Let us now learn about local variable through an example:

Code Segment

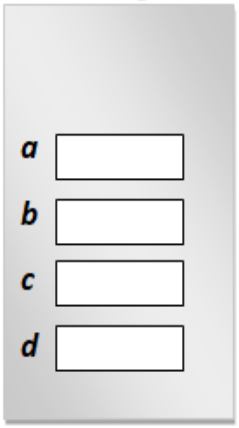
```
class Demo
{
    public static void main(String[] args)
    {
        int a;
        float b;
        boolean c;
        double d;

        //a = 10;
        //b = 99.99f;
        //c = true;
        //d = 100.99;

        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);
    }
}
```

Static Segment

Stack Segment



Heap Segment

*Here, we have created four local variables namely **a, b, c and d** which are directly inside the main method of class Demo.*
*Now these local variables are allocated memory in the **stack segment.***
These variables are not automatically initialized with default values by the JVM, they have to be initialized before use.
In the above example, we have commented the initialized values of local variables to prove that if they are not initialized we get compile time error.

Output:

(Before initializing values to local variables)

Compile time error

Output:

(After initializing values)

100
99.99
true
100.99

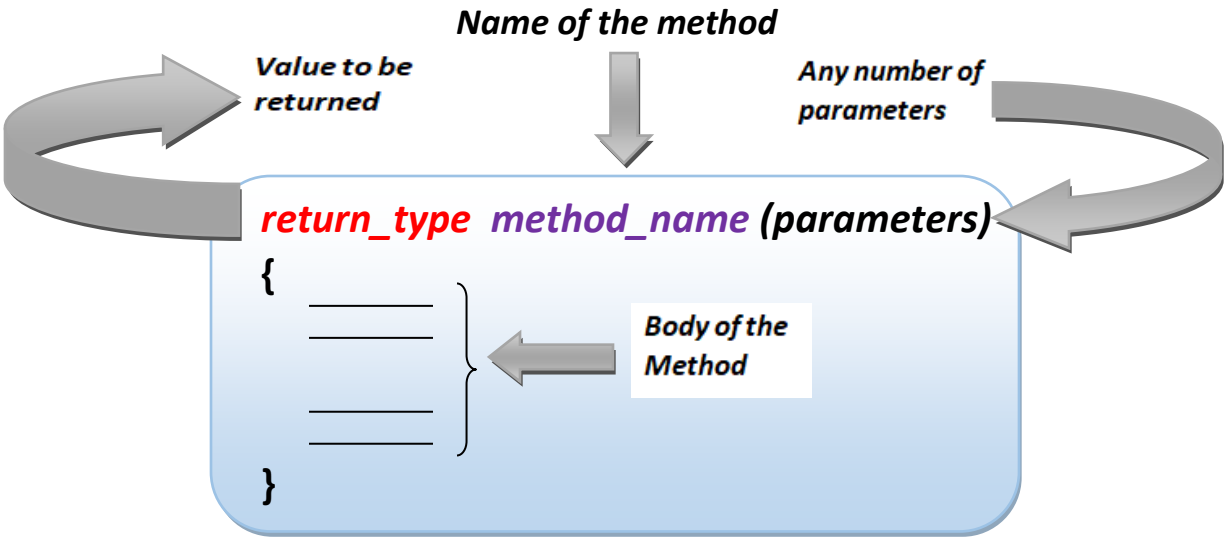
Difference between Local variables and Instance variables

	Local Variables	*Instance Variables*
1)	*These are created within a method.*	*These are created within a class outside the method.*
2)	*JVM does not allocate default values.*	*JVM allocates default values.*
3)	*They cannot be used without initialization.*	*They can be used without initialization.*
4)	*They are allocated memory on the stack frame in the stack segment.*	*They are allocated memory inside the object on the heap segment.*
5)	*They are de-allocated memory when the stack frame is deleted.*	*They are de-allocated memory by the garbage collector.*
6)	*We need not create objects to access local variables.*	*The object has to be created to access instance variables.*

Methods in Java

A method is collection of statements that are grouped together to perform an operation.
*It is **a block of code** which only runs **when it is called**.*
A method can perform a specific task with returning result to the caller or without returning anything.

In general, a method has the following syntax:



return_type: *The returnValueType is the data type of the value the method returns. Some methods perform the desired operations without returning a value. In those cases, the returnValueType is the keyword **void**.*

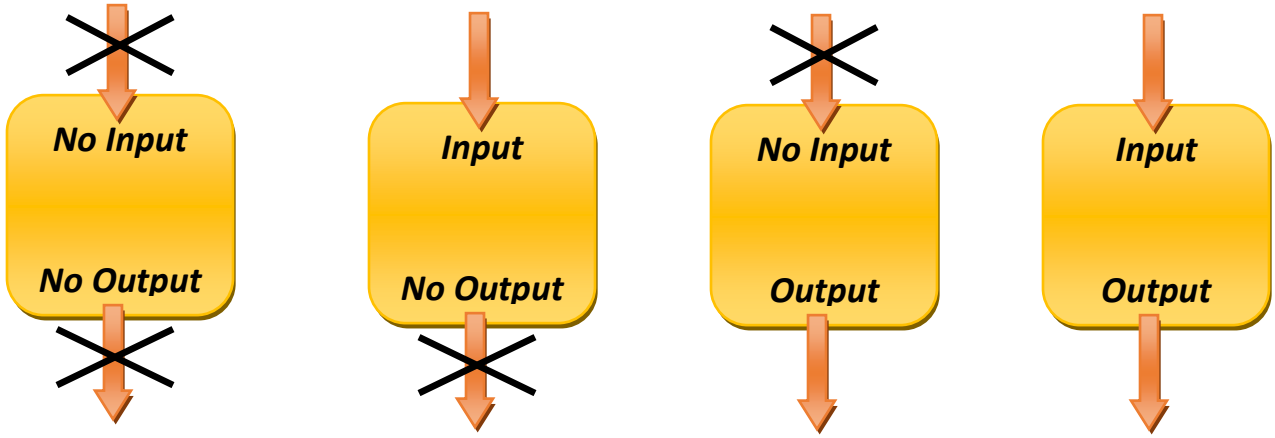
method_name: *This is the actual name of the method. The method name and the parameter list together constitute the method signature.*

parameters: *A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or **argument**. Parameters are optional. That is, a method may contain no parameters also.*

Body of the method: *The method body contains a collection of statements that define what the method does.*

Different types of methods

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control back to the caller of the method. There are four different types of methods in Java.



Let us see an example on each case:

Case-1: Without input and without output

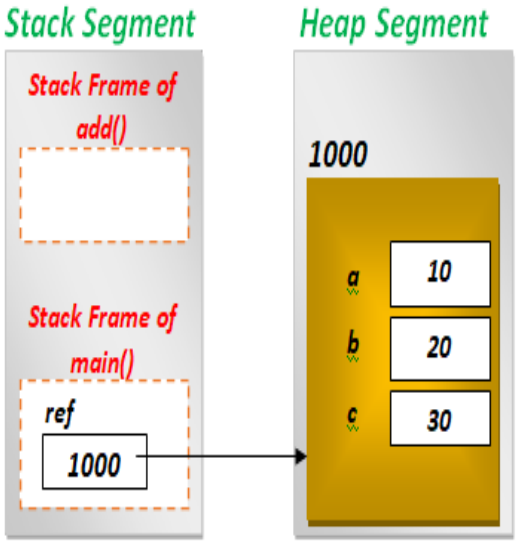
Code Segment

```
class Addition
{
    int a,b,c;

    void add()
    {
        a = 10;
        b = 20;
        c = a+b;
        System.out.println(c);
    }
}

class Demo
{
    public static void main(String[] args)
    {
        Addition ref = new Addition();
        ref.add();
    }
}
```

Static Segment



Output:

30

*Here, the execution starts from main method() which is called by the Operating System. Whenever, a method is called a region is created in the stack segment called **Stack frame**. And therefore, stack frame of main() gets created in the stack segment.*
*By using a "**new**" keyword an object is created and memory for it is allocated in the heap segment. The instance variables **a, b** and **c** are allocated memory in the heap segment and default values are given to them by the JVM. A reference variable is created with name **ref** and created in Stack segment.*
*Now, **add()** is a method which does not have any parameters. This method add() is called and stack frame of add() is created in the Stack segment. Then, the body of add() is executed, but this doesn't return any value and controls goes back to the caller of the method.*

We now know what is a method but why methods??

*The main advantage of method is **code reusability**. We can write a method once, and use it multiple times. We do not have to rewrite the entire code each time.*
Methods make code more readable and easier to debug. For example, getData() method is so readable, that we can know what this method will be doing.

UNDERSTOOD?