Local variable

A *local variable* is a variable that is declared within the body of a method. In this short document, we cover:

1. Basic declaration of a local variable
2. Local variables are not initialized by default
3. Scope of a local variable
4. When space for a local variable is allocated and deallocated
5. Principle: declare a local variable as close to its first use as possible
6. (Since Java 10) Use of a local variable declaration like var k= 6;

The method given to the right uses a local variable `t` of type **`int`**. This is not the best method to return the larger of two values, but it illustrates that the declaration of a local variable can be placed essentially anywhere where a statement can be placed.

```
/** Return the max of x and y. */
public static int max(int x, int y) {
    // ensure that x is the max
    if (x < y) {
        int t= x;
        x= y;
        y= t;
    }
}
```

The declaration of a local variable has the form:

<*type*> <*variable-name*>;

but it can be combined with an assignment statement:

<type> <*variable-name*> = <*expression*> ;

Local variables are not initialized

You know that fields that are not explicitly assigned a value are given a default value, depending on their type. For example, suppose field `f` is declared as follows:

```
public int f;
```

If the constructor does not store a value in `f` when an object is created, `f` will contain 0.

This is not the case for local variables: they are uninitialized. Silly method `zero` to the right won't even compile! It has a syntax error because it can be determined that the reference to `m` in the return statement is illegal because `m` does not have a value. Go ahead —copy method `zero` into a Java class somewhere and see what happens!

```
/** Return 0. */
public int zero() {
    int m;
    return m;
}
```

Scope of a local variable

The scope of a local variable —the places it can be used— consists of its declaration to the end of the block in which it is declared. In the example of method `max` above, the scope of `t` is the whole block

```
{ int t= x; x= y; y= t; }
```

Suppose the if-statement in method `max` was written as in the box to the right. The reference to `t` in the declaration-assignment to `b` is syntactically wrong because that reference to `t` is not within the scope of local variable `t`, so the program would not compile.

```
if (x < y) {
    int b= t;
    int t= x;
    x= y;
    y= t;
}
```

When space for a local variable is allocated and deallocated

As you will learn when you study how a method call is executed, space for all parameters and local variables is created *before* the method body is executed and space is deallocated *after* execution of the method body ends.

The method to the right has parameter `b`, which is (a pointer to) an array, and two local variables. Space for `b`, `k`, and `t` is allocated *before* the method body is executed. In particular, space for local variable `t` is created *once* only, even though it is declared in the body of the loop.

```
/** Return true iff b contains a 0 */
public boolean z(int[] b) {
    for (int k= 0; k < b.length; k++) {
        int t= b[k];
        if (t == 0) return true;
    }
    return false;
}
```

Local variable

Principle: declare a local variable as close to its first use as possible

Beginning programmers have the tendency to place local variable declarations at the top of the method body, as shown to the right. *Fight this tendency*! Logically, it makes no sense to declare `t` there. Why should the reader have to know about it at that place? Logically, `t` should be declared within the then-part of the if-statement, as done at the top of this page.

Follow this principle: *Declare a local variable as close to its first use as possible*!

```
/** Return the max of x and y. */
public static int max(int x, int y) {
    int t;
    // ensure that x is the max
    if (x < y) {
        t= x;
        x= y;
        y= t;
    }
```

Use of var

Since Java version 10, instead of writing a local variable declaration with an initializer like this:

```
int k= 50;
```

one could write:

```
var k= 50;
```

The type of `k` is still **`int`**! The type is inferred from the initializer `50`.

The website http://openjdk.java.net/jeps/286 gives this rationale for the introduction of **var**:

> We seek to improve the developer experience by reducing the ceremony associated with writing Java code, while maintaining Java's commitment to static type safety, by allowing developers to elide the often-unnecessary manifest declaration of local variable types.

Here are other examples:

```
var b= true;                // type of b is boolean
var c= 3 < 4;               // type of c is boolean
var s= "54";                // type of s is String
var ar= new int[] {3, 5};   // type of ar is int[]
var jf= new JFrame();       // type of jf is class-type JFrame
var st= new ArrayList<String>();
```

Keyword **var** can be used only with an initializer; for example, the following is illegal because there is no way to infer the type:

```
var x;
```

Also, if the type cannot be unambiguously inferred from the initializing expression, it won't compile. Here are examples:

```
var n= null;          // Illegal: the type of n cannot be inferred.
var ar= {3, 5};       // Illegal: the type of the initializer is unknown
var lam= () -> {};    // Illegal: the target type of the anonymous function () -> {} is unknown
```