Types

primitives vs. references

Primitive types store the primitive values to which they correspond. The primitive types are:

`byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`

A `byte` variable stores a byte (or 8 bits) of information, an `int` variable stores an integer, a `boolean` variable stores a boolean value (true or false), etc.

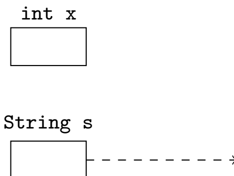
Reference types, on the hand, store reference values (or, addresses). This is similar to a reference in C++ which is signified by the `&` operator. Most data types that are used in Java are reference types and, in fact, *all objects are accessed through reference variables.* Furthermore, all reference variables hold the addresses of memory on the heap as all objects are dynamically allocated.

Declaration

For primitive types, the JVM allocates enough memory to hold that data type. For reference types, the JVM allocates enough memory to hold an address (the size is implementation-dependent).

```
int x;              // enough memory for an integer
String s;           // enough memory for an address
```

The first line of code above allocates memory for an integer, though one is not stored yet. Similarly, the second line allocates memory for a reference to a `String` object, though it does not yet refer to any particular `String` object.

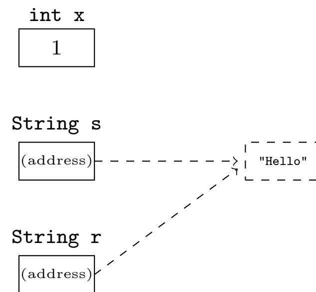


Initialization (and Assignment)

The differences in behavior of primitive and reference types can be illustrated further with the use of initialization. For primitives types, the value assigned to the variable is stored at the memory location which was allocated to that variable. For references types, the address of new or pre-existing object is stored at the memory location allocated to that variable.

```
int x = 1;                        // value 1 is stored in x
String s = new String("Hello");   // String object created, s refers to it
String r = s;                     // r refers to the same object as s
```

The first line of code allocates enough memory for an integer and stores in that memory location the value 1. The second line of code allocates enough memory for an address and a `String` object and assigns the address of the object to the variable. The third line of code allocates enough memory for an address and assigns the address of the object from line 2 to the variable. The same behavior applies to assignment statements, as initialization is simply the combination of a declaration and an assignment statement.



Comparison

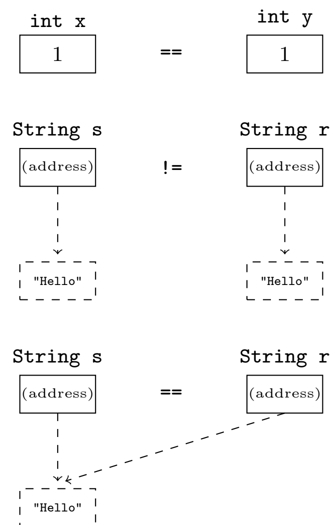
For primitive types, when the contents of two variables are compared the primitive values themselves are compared. For reference types, when the contents of two variables are compared the addresses stored in the variables are compared.

```
int x = 1;
int y = 1;
System.out.println(y == x);           // prints true

String s = new String("Hello");
String r = new String("Hello");
System.out.println(s == r);           // prints false

r = s;
System.out.println(s == r);           // prints true
```

The first output statement prints true, as the contents of `x` and `y` are being compared (and both have a value of 1). The second output statement prints false, as the contents of `s` and `r` are being compared but each holds a different value – i.e. the address of different String objects. Note that although they both refer to `String` objects which have the same value, *it is not the objects which are compared but the addresses*. This is further illustrated in the third output statement which prints true, as both `s` and `r` now refer to the same `String` object.



Arguments and Parameters

When variables are passed to a function as arguments, they are *always passed by value*. For primitives, this means that some primitive value is copied and passed into a parameter. This is behavior is similar for references; however, remember that the value of a reference is an address. And so, an address is copied and passed into a parameter.

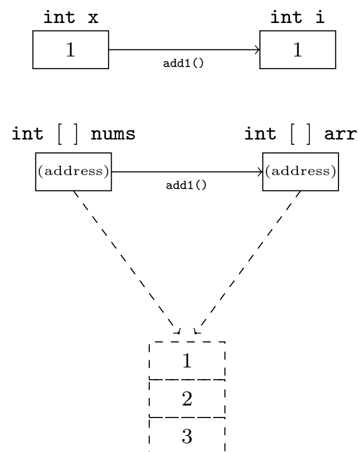
```
int num = 1;
add1(num);                          // value of x unchanged

int [] nums = {1, 2, 3};
add1(nums);                        // value of arr = {2, 3, 4}
```

```
void add1(int i) {
    i = i + 1;
}

void add1(int [] arr) {
    for(int i = 0; i < arr.length; i++)
        arr[i] = arr[i] + 1;
}
```

When `num` is passed to `add1()`, the value remains unchanged after the function call because `num` was copied. In other words, the parameter `i` (in the function) and the argument `num` are two different variables. When `nums` is passed to `add1()`, the values in the array are changed because the address of `nums` was copied. In other words, the parameter `arr` (in the function) and the argument `nums` both contain the same address, and so refer to the same object.



Scope and Lifespan

The lifespan of a variable is dependent on whether it is within the scope of some method. For any variable, the moment the method in which they were declared returns, that variable goes out of scope and “dies”. With regard to objects, the moment all references to an object have “died”, the object then goes out of scope and will eventually be garbage collected.