Variables and Literals

MAY 14, 2015

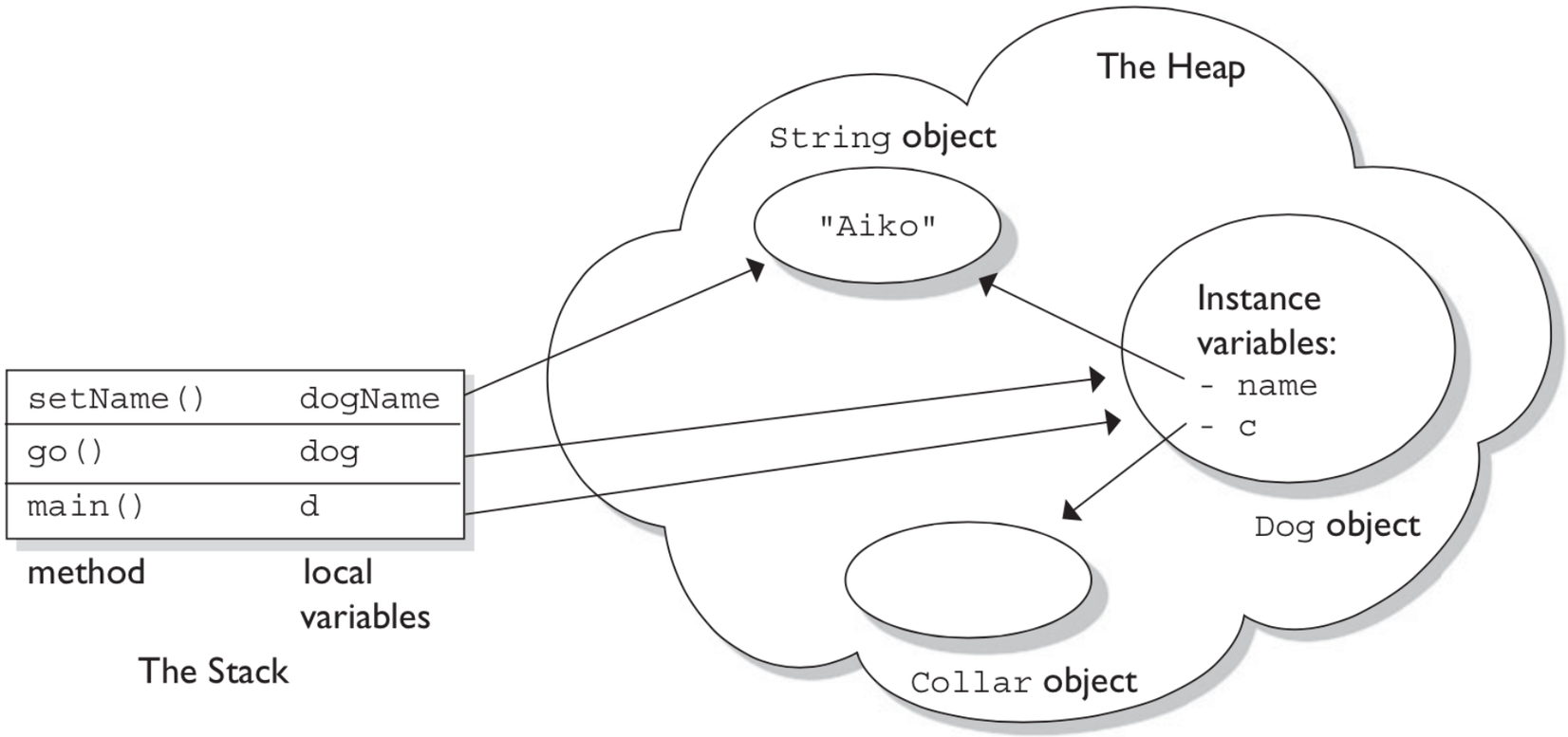
Variables are devices that are used to store data, such as a number, or a string of character data so that we can manipulate them later in our program. Variables can be broadly classified in to 4 types in Java:

1. **Class** variables (static, declared in a class).
2. **Instance** variables (non-static, declared in a class).
3. **Local** variables (declared inside a method).
4. **Block** variables (variables in static blocks, for-loop blocks etc).

Instance variables and objects reside in heap whereas local variables reside in stack. Consider the below program:

```
class Collar {
}

class Dog {
    Collar c; // instance variable
    String name; // instance variable

    public static void main(String[] args) {
        Dog d; // local variable: d
        d = new Dog();
        d.go(d);
    }

    void go(Dog dog) { // local variable: dog
        c = new Collar();
        dog.setName("Aiko");
    }

    void setName(String dogName) { // local var: dogName
        name = dogName;
        // do more stuff
    }
}
```

For the above program, the instance variables, objects and local variables will be stored in memory as shown in the figure below:



Literal Values for All Primitive Types

Literals are nothing but values that a particular data type can hold. A **primitive literal** is merely a source code representation of the primitive data types, in other words, an integer, floating-point number, boolean, or character etc. that you type in while writing code. The following are examples of primitive literals:

```
127          // byte literal
376          // short literal
4290         // int literal
58L          // long literal
2546.343f    // float literal
2546789.343  // double literal
```

```
'b'          // char literal
false        // boolean literal
```

Integer Literals

There are four ways to represent integer numbers in the Java language: decimal (base 10), octal (base 8), hexadecimal (base 16), and from Java 7, binary (base 2).

One more new feature introduced in Java 7 was **numeric literals with underscores (_) characters**. This was introduced to increase readability. See below:

```
int pre7 = 1000000;      // pre Java 7 — we hope it's a million
int with7 = 1_000_000;   // much clearer!
```

But you must keep in mind the below gotchas:

```
int i1 = _1_000_000; // illegal, can't begin with an "_"
int i2 = 10_0000_0;  // legal, but confusing
```

NOTE: You can use the underscore character for any of the numeric types (including doubles and floats), but for doubles and floats, you CANNOT add an underscore character directly next to the decimal point.

Decimal Literals

These are numbers with a radix of 10 which we use most commonly. They do not need prefix of any kind and are initialized as below:

```
int length = 343; // 343 is the literal
```

Binary Literals

From Java 7, you can initialize variables holding binary literals. But they must start with either `0B` or `0b`, as shown below:

```
int b1 = 0B101010;   // set b1 to binary 101010 (decimal 42)
int b2 = 0b00011;    // set b2 to binary 11 (decimal 3)
```

Octal Literals

Octal integers use only the digits 0 to 7. They have a radix of 8. In Java, you represent an integer in octal form by placing a zero in front of the number, as follows:

```
int six = 06;      // equal to decimal 6
int seven = 07;    // equal to decimal 7
int eight = 010;   // equal to decimal 8
int nine = 011;    // equal to decimal 9
```

You can have up to 21 digits in an octal number, not including the leading zero. This is because no mater what number system you use, the range of values that an `int` can hold is always between -2^{31} to $+2^{31} - 1$.

Hexadecimal Literals

Hexadecimal (hex for short) numbers are constructed using 16 distinct symbols. They have a radix of 16. Counting from 0 through 15 in hex looks like this:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
```

Java accepts uppercase or lowercase letters for the extra digits *(one of the few places Java is not case-sensitive)*. You represent an integer in hexadecimal form by placing a `0x` in front of the number, as follows:

```
int x = 0X0001;     // equals to decimal 1
int y = 0x7fffffff; // equals to decimal 2147483647
int z = 0xDeadCafe; // equals to decimal -559035650
```

All four integer literals (binary, octal, decimal, and hexadecimal) are defined as `int` by default, but they may also be specified as `long` by placing a suffix of `L` or `l` after the number:

```
long jo = 110599L;
long so = 0xFFFFl;  // Note the lowercase 'l'
```

Floating-point Literals

Floating-point numbers are defined as a number, a decimal symbol, and more numbers representing the fraction. For example,

```
double d = 11301874.9881024;
```

By default, floating-point literals are defined as `double` (64 bits) so if you want to assign a floating-point literal to a variable of type `float` (32 bits), you must attach the suffix `F` or `f` to the number. So, the below code generates a compiler error:

```
float f = 23.467890;     // Compiler error, possible loss
                         // of precision
```

This happens because we're trying to fit a larger number (64 bits) into a (potentially) less precise "container" (32 bits).

Now as by default floating-point literals are of type `double`, it is optional to attach a suffix of `D` or `d` when you want to assign it to a variable of type `double`. For example,

```
double d = 110599.995011D; // Optional, not required
double  g = 987.897;       // No 'D' suffix, but OK because the
                           // literal is a double by default
```

Boolean Literals

Boolean literals can be either `true` or `false`. In C (and some other languages) it is common to use numbers to represent true or false, but this will not work in Java. For example,

```
boolean t = true;  // Legal
boolean  f = 0;    // Compiler error!
int x = 1;  if (x) { } // Compiler error!
```

Character Literals

A char literal is represented by a single character in **single quotes**:

```
char a = 'a';
char b = '@';
```

You can also assign unicode value to a `char` variable, like:

```
char letterN = '\u004E'; // The letter 'N'
```

Note, characters are nothing but **16-bit unsigned integers**. So, you can assign a number literal, assuming it will fit into the unsigned 16-bit range (0 to 65535) to a `char` variable. For example, the following are all **legal**:

```
char a = 0x892;         // hexadecimal literal
char b = 982;           // int literal
char c = (char)70000;   // The cast is required; 70000 is
                        // out of char range
char d = (char) -98;    // Ridiculous, but legal
```

And the following are **not legal** and produce compiler errors:

```
char e = -29;     // Possible loss of precision; needs a cast
char f = 70000;   // Possible loss of precision; needs a cast
```

Literal values for Strings

You can create a `String` in Java in the following ways:

```
String s = "tutorial";
String str = new String("Rahul roy");
String con = s + str; // concatenate 2 strings
```

Strings are **not primitives** in Java but can be represented as literals, in other words, they can be typed directly into code like:

```
System.out.println("Bill" + " Joy");
```

Literal values for Non-Primitives

Variables are just bit holders, with a designated type. You can have an `int` holder, a `double` holder, a `long` holder, and even a `String[]` holder. This holder is assigned a bunch of bits representing the value. For primitives, the bits represent **a numeric value** but for non-primitives, these bits represent **a way to get to the object**.

For example, a `byte` with a value of 6 means that the bit pattern in the variable (the `byte` holder) is `00000110`, representing the 8 bits. But what happens in case of non-primitives, for example, `Button b = new Button();`, what's inside the `Button` holder `b`? Is it the Button object? No! A variable referring to an object is just a reference variable. A **reference variable** bit holder contains **bits representing a way to get to the object**. We don't know what the format is. The way in which object references are stored is virtual-machine specific (it's a pointer to something, we just don't know what that something really is). All we can say for sure is that the variable's value is not the object, but rather a value representing a specific object on the heap. Or `null`. When it is `null`, i.e, `Button b = null;` you can say that the reference variable `b` is not referring to any object.

There is one important concept to understand here, i.e, a reference variable can refer to any object that is a **subclass** of the declared reference variable type but not a **superclass**. Let's see why.

```
1   class Foo {
2       public void doFooStuff() { }
3   }
4   class Bar extends Foo {
5       public void doBarStuff() { }
6   }
7   class Test {
8       public static void main (String [] args) {
9       Foo reallyABar = new Bar();  // Legal because Bar is a
10                                   // subclass of Foo
11      Bar reallyAFoo = new Foo();  // Compiler error! Foo is not a
12                                   // subclass of Bar
13      }
14  }
```

In line 11, `reallyAFoo` is a `Bar` reference variable (child) so someone would call `reallyAFoo.doBarStuff()` but the reference variable actually holds a `Foo` object (parent) which doesn't have a `doBarStuff()` method. So, the compiler prevents this and gives a `Incompatible types` error.

In other words, a child class is nothing but the parent class with additional properties. So there is no issue in line 9, where a `Foo` reference variable (parent) is holding a `Bar` object (child). Because everything a `Foo` object can do, can also be done by a `Bar` object.

Casting

Casting is a way of converting literal values/objects from one type to another. When the type of variable is **different** from the type of literal/object it's holding/referring, you may require **casting**.

Casting can be done by the compiler *(implicit cast)* or by you *(explicit cast)*. Typically, an implicit cast happens when you're doing a **widening conversion**, in other words, putting a smaller thing (say, a byte) into a bigger container (such as an int). But when you try to put a large value

into a small container (referred to as **narrowing**), you should do an explicit cast, where you tell the compiler that you're aware of the danger and accept full responsibility. Let's for example consider the below program:

```
class Casting {
    public static void main(String [] args) {
        long a = 100;   // literal '100' is implicitly an 'int'
                        //but the compiler does an implicit cast
        int b = (int) 10.23;    // literal '10.23' is implicitly a 'double'
                                // so we require an explicit cast
        int x = 3957.229;   // illegal, can't store a large value in a
                            // small container without explicit cast
    }
}
```

There are some rules which you must be aware of:

- The result of an expression involving anything int-sized or smaller is always an `int`, so we must explicitly cast it. Check this out:

```
byte a = 3;      // No problem, 3 fits in a byte
byte b = 8;      // No problem, 8 fits in a byte
byte c = a + b; // Should be no problem, sum of the two bytes
                // fits in a byte
```

The last line won't compile! You'll get an error like this:

```
TestBytes.java:5: possible loss of precision
found   : int
required: byte
byte c = a + b;
         ^
```

Doing an explicit cast like:

```
byte c = (byte) (a + b);
```

solves the issue.

- In case of compound assignment operators, explicit cast isn't required. The below code compiles just fine:

```
byte b = 3;
b += 7; // No problem - adds 7 to b (result is 10)
```

What happens when you cast a large value to store it in a small container

When you do a explicit cast, the compiler just keeps **the number of bits (from right)** that the variable type can hold and strips off the rest. Consider the below program to understand better:

```
1   class Casting {
2       public static void main(String [] args) {
3           int i = 7;
4           byte b = (byte) i;
5           System.out.println("The 1st byte is " + b); // prints 7
6           // now let's see another example
7           int i = 128;
8           byte b = (byte) i;
9           System.out.println("The 2nd byte is " + b); // prints -128
10      }
11  }
```

So, in line 4, `i` is 7 i.e, `00000000000000000000000000000111` (32 bits) and when we do a explicit cast, the compiler just stores `00000111` (8 bits) in variable `b` (as `byte` can hold only 8 bits) which is also 7. Therefore, it prints 7. But in line 8, `i` is `128` i.e, `00000000000000000000000010000000` and after stripping off the extra bits we are left with `10000000` which is not `128` as the 1st bit is the sign bit. So, after computing the 2's compliment of it we get `–128` as the result.

Scope

Scope refers to the **lifetime and accessibility of a variable**. In simple words, how long will the variable be hanging around so that they can be used by other parts of the program. Different types of variables have different scope.

1. **Class or static** variables have the longest scope. They are created when the class is loaded, and they survive as long as the class stays loaded in the Java Virtual Machine (JVM).
2. **Instance or non-static** variables are the next most long-lived. They are created when a new instance is created, and they live until the instance is removed.
3. **Local** variables are next. They live as long as their method remains on the stack. As we'll soon see, however, local variables can be alive and still be "out of scope".
4. **Block** variables live only as long as the code block is executing.

Below program shows all types of variables and explains their scopes too. Please refer to the comments to understand which are **in scope** and which are **out of scope**.

```java
class Scope {
    static int s = 343; // static variable
    int x; // instance variable

    { // initialization block
        x = 7;
        int x2 = 5; // block variable
    }

    Scope() { // constructor
        x += 8;
        int x3 = 6;
    }

    void doStuff() { // method
        int y = 0; // local variable
        for (int z = 0; z < 4; z++) { // 'for' code block
            y += z + x;
        }
        z++; // compiler error (out of scope)
        x2++; // compiler error (out of scope)
    }

    public static void main(String[] a) {
        x++; // compiler error! 'x' is instance variable, so
             // we need an object to access it
        x2++; x3++; // compiler error! block variables scope
                    // is only inside the block in which they
                    // are declared
    }
}
```

Variable Initialization

Java gives us the option of initializing a declared variable or leaving it uninitialized. When we attempt to use the uninitialized variable, we can get different behavior depending on **what type of variable or array we are dealing with (primitives or objects)**. The behavior also depends

on **the level (scope) at which we are declaring our variable**.

Default values for Instance variables (Primitive and Non-primitive):

Variable Type	Default Value
`byte`, `short`, `int`, `long`	`0`
`float`, `double`	`0.0`
`boolean`	`false`
`char`	`'\u0000'`
Object reference	`null` (not referencing any object)

Therefore, for the below program:

```
public class Book {
    private String title;       // instance reference variable
    private int noOfPages;      // instance primitive variable
    public String getTitle() {
        return title;
    }
    public int getNoOfPages() {
            return noOfPages;
    }
    public static void main(String [] args) {
        Book b = new Book();
        System.out.println("The title is " + b.getTitle());
        System.out.println("No. of pages are " + b.getNoOfPages());
    }
}
```

The output will be: `The title is null No. of pages are 0`

NOTE: `null` is not the same as an empty `String ("")`. A `null` value means the reference variable is not referring to any object on the heap.

Array Instance Variable

An array is an object, thus, an array instance variable that's declared but not explicitly initialized will have a value of **`null`**, just as any other object reference instance variable. But if the array is initialized, all array elements are given their default values, the same default values that elements of that type get when they're instance variables. In short, **Array elements are always, always, always given default values, regardless of where the array itself is declared or instantiated**.

Variable Type	Default Value
Array (uninitialized)	`null`
Array (initialized)	Default values of their respective types as discussed above

Default values for Local (also called Stack or Automatic) variables (Primitive and Non-primitive):

Local variables, including primitives, always, always, always must be initialized before you attempt to use them (though not necessarily on the same line of code). Java does not give local variables a default value, you must explicitly initialize them with a value.

Q&A

Q1. Given the below program:

```java
public class Fishing {
     byte b1 = 4;
     int i1 = 123456;
     long L1 = (long)i1; // line A
     short s2 = (short)i1; // line B
     byte b2 = (byte)i1; // line C
     int i2 = (int)123.456; // line D
     byte b3 = b1 + 7; // line E
}
```

Which lines WILL NOT compile? (Choose all that apply.) A. Line A B. Line B C. Line C D. Line D E. Line E

Q2. Given the below program:

```
class Mixer {
    Mixer() {
    }

    Mixer(Mixer m) {
        m1 = m;
    }

    Mixer m1;

    public static void main(String[] args) {
        Mixer m2 = new Mixer();
        Mixer m3 = new Mixer(m2);
        m3.go();
        Mixer m4 = m3.m1;
        m4.go();
        Mixer m5 = m2.m1;
        m5.go();
    }

    void go() {
        System.out.print("hi ");
    }
}
```

What is the result? A. hi
B. hi hi
C. hi hi hi
D. Compilation fails
E. hi, followed by an exception
F. hi hi, followed by an exception

Q3. Given the below program:

```java
class Fizz {
    int x = 5;

    public static void main(String[] args) {
        final Fizz f1 = new Fizz();
        Fizz f2 = new Fizz();
        Fizz f3 = FizzSwitch(f1, f2);
        System.out.println((f1 == f3) + " " + (f1.x == f3.x));
    }

    static Fizz FizzSwitch(Fizz x, Fizz y) {
        final Fizz z = x;
        z.x = 6;
        return z;
    }
}
```

What is the result? A. true true

B. false true

C. true false

D. false false

E. Compilation fails

F. An exception is thrown at runtime

Next »