INFO1103: Introduction to Programming

School of Information Technologies, University of Sydney

Week 2: Variables, Expressions

We will cover: Variables (numbers, characters and Strings), mathematical and Boolean (true/false) expressions, using the Java Console

You should read: Sections 2.1 – 2.4 of Savitch

Lecture 3: Variables and Expressions

Storing and manipulating information in memory

Binary Numbers

It helps to know about binary numbers in your programming, because everything in a computer is stored in binary digits (bits).

You can think of Boolean[1] values as *binary* numbers (in a sense)

$$TRUE = 1, FALSE = 0$$

[1] Why have I written "Boolean" here and not `boolean`? Because a Boolean variable is named after George Boole, who developed the concept of what we now call Boolean algebra in 1854. The variable type in Java uses the lower-case 'b' because it's a *primitive type*, and the naming convention for primitives is to use a lower case.

Integers are stored as binary

All integers can be thought of as binary numbers: 0, 1, 10, 11, 100, 101, . . . :

base 10	**base 2 / binary**	**expansion**
0	0	0
1	1	1×2^0
2	10	$1 \times 2^1 + 0 \times 2^0$
3	11	$2^1 + 2^0$
5	101	$2^2 + 2^0$

Bytes

```
00000000 = 0
00000001 = 1
00000010 = 2
00000100 = 4
00001000 = 8
00010000 = 16
00100000 = 32
01000000 = 64
10000000 = 128
--------
76543210 bits
```

In fact everything in your computer is stored as bits.

A byte is eight *bits*.

Bit means “binary digit”.

Bits

1s and 0s are easy to store as “on” and “off” so binary has become the universal way to store all electronic information.

If the bits are

$$d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0$$

then the total value is

$$d_7 2^7 + d_6 2^6 + d_5 2^5 + d_4 2^4 + d_3 2^3 + d_2 2^2 + d_1 2^1 + d_0 2^0$$

which is equal to

$$128 d_7 + 64 d_6 + 32 d_5 + 16 d_4 + 8 d_3 + 4 d_2 + 2 d_1 + d_0$$

E.g.,

$$01101010 = 0 + 64 + 32 + 0 + 8 + 0 + 2 + 0 = 106$$

Hexadecimal

Sometimes it's handy to have a more compact way to store numbers in a range that's a power of two.
Noting that 2^4 is 16, we use the first 6 letters of the alphabet to represent 10, 11, . . . , 15.

This is more compact: a byte just takes 2 characters: `00` (0) to `FF` (255).

To indicate that a number is to be interpreted in this way we put `0x` in front: `0x00` and `0xFF`.
Two-byte numbers need four such characters, so look like this: `0x0000` to `0xFFFF` (65535).

binary	**hexadecimal**	**decimal**
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Variables

Programs would be pretty simple and useless if they couldn't handle data that could take different values. What if we wanted to print something other than "Hello World!" or "I'm sorry Dave, I can't let you do that" ?

```java
public class AboutSale {
    public static void main(String[] args) {
        System.out.println("You bought 4 items");
        System.out.println("It costs you $12.50");
        System.out.println("Served by autobot 526 on 24/02/2016");
    }
}
```

To do this we need to have some *variables*.

Variables (cont.)

A variable is a piece of information that's stored in memory so we can access it by giving its name later, like this:

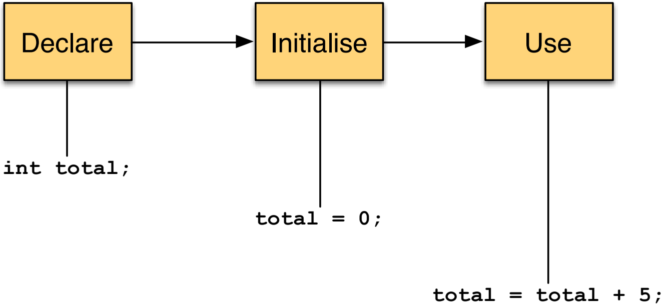
```
    int x = 4;
    int y = 8;
    int z = x + y;
```

- Here any time we need to get the value of x we just call it by name.
- The name doesn't matter: it's just a label.

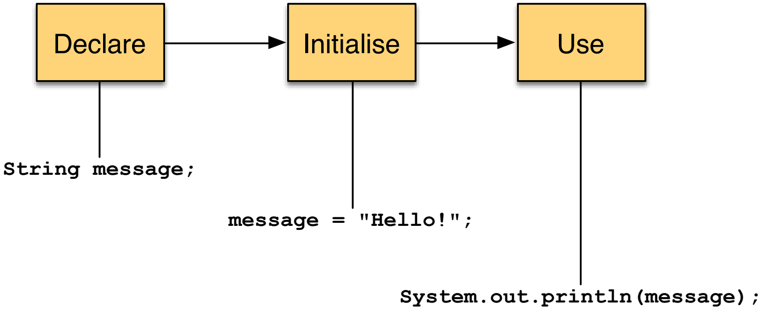
Primitive types are the simplest type of variables e.g. an integer (int). They use a fixed amount of memory. They can be changed with simple math operations e.g. + - * / %

Objects are customised type of variables e.g. String. One object can contain many primitive types as well as many other Objects. The amount of memory used by an Object can change over time. They can have customised operators *defined by the programmer*. Objects will be discussed in later lectures.

Declare → Initialise → Use



Declare → Initialise → Use



```
String message;
```

```
message = "Hello!";
```

```
System.out.println(message);
```

You have to have things in this order, else there will be … *trouble*.

Variables must *declared* before they're used

So you have to *declare* a variable before you use it: you have to decide what type you need first, like an integer, character or string, before you start using it.
Code that misses the declaration out or gets it in the wrong order, like this:

```java
public class DeclareVariables {
    public static void main(String[] args) {
        x = 5;
        int x;
    }
}
```

Simply won't compile:

```
DeclareVariables.java:3: error: cannot find symbol
      x = 5;
      ^
  symbol:   variable x
  location: class DeclareVariables
1 error
```

Initialising Variables

Next, once they're declared, variables have to be *initialised* before they're used.

Initialising is giving the very first value to a variable.

Good programmers *always* initialise their variables.

```
public class Uninitialised {
  public static void main(String [] args) {
                int x;
                System.out.println(x);

                int y;
                y = 0; // initialised before use
                System.out.println(y);
  }
}
```

Scope: Variables have a lifetime

In the code you've seen so far the variables have been short-lived and this hasn't mattered one bit.

Next we'll see this is a very important concept for you to remember (yes, another one).

Blocks of code

I have alluded to blocks already, but to make sure you have the right idea about them, let's be a bit more explicit about what they are, because we'll need this next.

A *block* is a collection of statements that is delimited by a pair of curly braces `{ }`.

```
    int x = 5;
{ // beginning a block here, call it block 1
    x = 10;
    // nested blocks
    { // beginning block 2 inside block 1
        { // beginning block 3
            x = x + 7;
        } // ending block 3
        x = x - 2;
    }
    { // ending block 2 and starting block 4
        x = 13;
    } // ending block 4
} // ending block 1
```

Blocks are logical chunks of code

Blocks of code, delimited by the curly braces { }, keep your code organised.

Within a block, intructions get executed in sequence.

Scope

Once it's been declared, a variable will only persist within its *scope*.

The scope of a local variable is basically the block in which it is declared:

```
int x = 3;
{
   int y = 5;  // y is declared here, but z doesn't exist yet
   int z = y;  // y is still fine, and z is now declared too
   x = y;      // we'll copy the value of y into x
}
// y is no longer defined: it is out of scope
// x now has the value 5
```

In particular the scope is *where the identifier can be used.*

What kinds of variables are there?

Java uses *types* to help organise what variables can do and what can be done with them.
There are two main types to begin with: *primitive* and *reference*. We will first use a lot of primitive types and leave reference types for later, except one very special one, the `String`.

Name	**Kind**	**Memory**	**Range**	**Type**
`boolean`	Boolean	1 byte	true or false	**primitive**
`byte`	integer	1 byte	[-128, 127]	**primitive**
`short`	"	2 bytes	[-32768, 32767]	**primitive**
`int`	"	4 bytes	[-2147483638, 2147483647]	**primitive**
`long`	"	8 bytes	$[\approx -9 \times 10^{18}, \approx 9 \times 10^{18}]$	**primitive**
`float`	floating-point	4 bytes	$\approx \pm 3.4 \times 10^{38}$ to 1.4×10^{-45}	**primitive**
`double`	"	8 bytes	$\approx \pm 1.8 \times 10^{308}$ to $\pm 4.9 \times 10^{-324}$	**primitive**
`char`	character	2 bytes	[0, 65535]	**primitive**
`String`	string	variable	[0, very long] depends on vendor JVM	**object**

⚠ Note: the primitive types above, `boolean` to `char` inclusive, are *reserved* words, but `String` is not reserved. You *can* use it as a variable name or to make your own type but it's very risky!

Basic types of primitive variables

`boolean` Used for true/false; is the type of expressions like "is *A* equal to *B*?"

`char` A single character; delimited by single quotes ' '

`byte`, `int`, `short`, `long` Used to store integer values: 0, ±1, ±2, …

`float`, `double` Used to store floating point numbers, with a *coefficient* and an *exponent*. For example, 6.02214179 and 23 respectively, giving the number $6.02214179 \times 10^{23}$.

`boolean` variables

A Boolean variable, in Java written as the reserved word “`boolean`” is one that can take a value *true* or *false*.

You will come across Boolean variables *very* commonly, and you will mainly use them to test whether certain things are true or not.

The default value of a `boolean` variable is `false`. That means if, for some reason you don’t initialise a `boolean` variable, its value will be false.

boolean code example

```java
public class BoolTest {
    public static void main(String[] args) {
        boolean foo = true;
        boolean bar = false;
        boolean flag = foo;
        flag = foo & bar;
        System.out.println("flag = " + flag);
    }
}
```

The above shows how you can *declare*, *initialise* and *operate* on a `boolean` variable called `flag`.

Notes:

- You use the values `true` and `false` directly;
- You can assign the value of one boolean variable to another;
- You can calculate the result of two `boolean` values with & operator.

char variables

A `char` is just a letter, like ‘a’ or ‘Z’ or $. Usually the characters we use are ASCII characters only:

a, b, c, d, . . . A, B, C, D, . . . , 0, 1, 2, 3, . . .

and other common symbols like

```
+ - * / = , .  > < ?  !
```

and so on.

Represent a `char` with *single quotes*, like this:

```
        char ch = 'A';
```

Why is char 16 bit?

Traditionally, a `char` followed the 7-bit code American Standard Code for Information Interchange (ASCII). Each character only has 128 possible values, this is enough for English.

Java uses 16 bits per character to accommodate characters from other languages as well as other kinds of symbols Unicode 7.0 Code Charts.

The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff'. (hexadecimal)

char code example

```java
public class CharTest {
    public static void main(String[] args) {
        char ch = 'A';
        char letter = 'Z';
        System.out.println("ch = " + ch);
        System.out.println("letter = " + letter);
    }
}
```

Above we see how to declare and initialise `char` variables. You can operate on them using numerical techniques, like *adding* or *subtracting*, but you might get unexpected results: this is a subject covered later.

Running this we get

```
~> java CharTest
ch = A
letter = Z
```

`byte`, `short`, `int` and `long`

These are all *numerical* types: they have a number value.

Note: `char` is *also* a numerical type!

You will most likely be using `int` for your integer variables.

When do I decide against using `int`?

- when range of `int` is not enough
- when memory becomes a serious issue e.g. a mobile device.
  1 million `int` variables consumes less than 4 MB memory

int example

```java
public class IntTest {
    public static void main(String[] args) {
        int a = 45;
        int b = 23;
        int c = a + b;
        System.out.println(a + " + " + b + " = " + c);
    }
}
```

Compiling and running the above we get

```
~> javac IntTest.java
~> java IntTest
45 + 23 = 68
```

Note how I had to separate the parts of the equation that I wanted to print out. If I had put just this:

```java
        System.out.println(a + b + " = " + c);
```

then I would have seen the output

```
      68 = 68
```

`float` and `double` variables

To represent numbers using a decimal point e.g. 3.14, 0.0023, -6.21×10^3

`float` and `double` are referred to as *floating point* types.

`float` type uses 4 bytes of memory, and `double` 8 bytes.

The bits are used to describe the sign, the exponent and the mantissa (fractional part).

The range and the accuracy of floating point types depends on how many bits are used in the *coefficient*, and the *exponent*. `double` has twice as many bits and is termed “double precision".

Always use `double`

double example

```java
public class DoubleTest {
    public static void main(String[] args) {
        double radius = 0.51238761523;
        double area = radius * radius * 3.141592659;
        System.out.println("The area of a circle of radius "
                + radius + " is " + area);
    }
}
```

Compiling and running the above we get this:

```
~> javac DoubleTest.java
~> java DoubleTest
The area of a circle of radius 0.51238761523 is 0.8247970926722155
```

`double` example (cont.)

Note that I used the multiplication sign ‘*’, rather than something like “^2”, which *doesn’t mean squared.* There is *no symbol in Java like that.*

To square a number, it’s simplest to multiply it by itself explicitly.

To raise a number to a power, use this:

```
        double radius = 0.4;
        double volume = Math.pow(radius, 3.0) * 4.0 / 3.0 * 3.1415926;
```

You can also get a more accurate from Java itself than trying to remember π:

```
public class VolumeOfSphere {
    public static void main(String[] args) {
        double radius = 0.4;
        double volume = Math.pow(radius, 3.0) * 4.0 / 3.4 * Math.PI;
        System.out.println("The volume of the sphere is " + volume);
    }
}
```

When to use integer or floating point

First identify the information that is going to be stored and how that changes over time.

Are they whole numbers or can there be a fractional component?

accuracy	if you are only dealing with integer values, integers are exact but floating point numbers are not
range	If you're dealing with very large or very small numbers, you have to use floating point types: their range is much bigger

In this course, you should pick either `int` or `double`.

Is there any danger of using a floating point type in your program if only integers are expected?

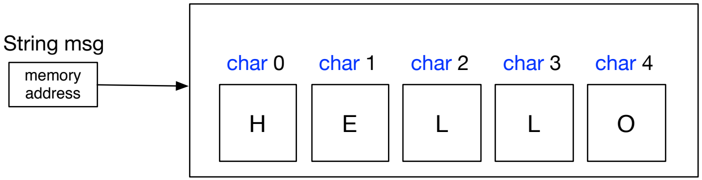
The `String` type

`String` is a sequence of characters *stringed* together to represent text information.

`String` is an object, not a primitive type.

- it can have a few characters or several thousand.
- it has special operators to query or manipulate the text information.

```
        String msg = "HELLO";
```



The `String` operators

+ *operator*. Creates a new String from the concatenation of two existing Strings.

length() method - returns an `int`. The number of characters in the String.

equals() method - returns a `boolean`. `true` if this String matches another, `false` otherwise.

many many more!

Too many. You don't have to know them all, but when you need to do something with a String, or many, refer to the Java documentation to find which operators are appropriate: http://docs.oracle.com/javase/7/docs/api/java/lang/String.html

String example

```java
public class StringTest {
    public static void main(String[] args) {
        String word1 = "Hello";
        String word2 = "World!";
        String twoWords = word1 + word2;
        System.out.println(twoWords);

        System.out.println("string word1 has " + word1.length()
                + " characters.");
        System.out.println("string twoWords has " + twoWords.length()
                + " characters.");

        System.out.println("Does String word1 match String word2 ?"
                + word1.equals(word2));
    }
}
```

The above, when compiled and run, gives

```
HelloWorld!
The string word1 has 5 characters.
The string twoWords has 11 characters.
Does String word1 match text of word2 ? false
```

String notes

`String` is clearly much more complicated than the primitive types. There is internal data and many operators.

The *methods* are *called* and used to query the String object for information.

`String` is an example of a *reference type*, which is a very important part of object-oriented programming that we will learn a great deal more about. Because `Strings` are so useful, we will begin using them now.

The default value of a `String` is a special value, `null`. If a reference type has the value `null` then it means it has not been initialised — and you really shouldn't use it or very bad things will happen!

Default values

There are some circumstances in which variables do not get initialised: in these cases the variables get *default values*.
The default value for the integer types `char`, `int`, `byte`, `long`, `byte` is

0

The default value for a `String` is

`null`

The default value for a `boolean` (Boolean) variable is

`false`