Simple variables and data types

Fundamental concepts

As explained in the first introductory lecture, there are two motivations for writing programs: software engineering for automation, formalization to help people think. Most languages designed for software engineering, including Java, are called "imperative" languages: a program tells the computer what to do, step by step, like a recipe tells the cook what to do.

All imperative languages are based on two fundamental concepts: 1) the computer proceeds through the program step-wise; so there is a conceptual "cursor" that designates "the current point of execution" over time; and 2) the computer is equipped with *memory* where values can be stored and retrieved.

So when you use an imperative language, you have two mechanisms at your disposal to tell the computer what to do: statements that change the *flow of control*, like "`if`" seen in the last lecture, and statements that *modify the state of the memory*, like the assignment.

In hardware, the memory is a circuit that can only store unstructured binary values: very long strings of digits 0 and 1. To ease the task of the programmer, language designers have created the concept of *variable* to add logical structure to the memory circuits.

A variable is a combination of a name for an area of memory; and a type which indicates the computer how to interpret the raw binary digits stored in that area.

Practical introduction

Variables in a program define space in a computer's memory where values can be stored and later retrieved.

In languages like Java, and contrary to other languages like Python, variables must be *declared* before they can be used: the program must "announce" that a variable exists and give it a name before subsequent steps can store values in it or read from it.

Moreover, *each variable can only store a limited set of different values.* For example, one variable could only hold integral numbers between -2147483648 and 2147483647, and another only values between -128 and 127. The programmer selects the set of values that can be potentially held in a variable by selecting a *type* for the variable when it is declared. The language will then automatically determine an area in memory of sufficient size to hold the variable.

Declarations

The general form for a variable declaration is defined thus:

Declaration:

Syntax: <type> <identifier> ;

Semantics: the construct declares a variable named by the identifier on the right, that can potentially store values determined by the type on the left.

Each language has different types for variables already *built-in*, ie provided "out of the box", always available to programs. For example, in Java, we often use the built-in types "`int`" for integral values (like 123) and "`double`" for approximate real values ("numbers with a comma", like 3.5). So we can write:

```
int i;
double x;
int myName;
```

to declare 3 variables, named "`i`", "`x`" and "`myName`", and the respective types `int`, `double` and `int`.

Built-in types in Java

Java provides us with the following built-in types (non-exhaustive):

Type name	**Set of allowed values**
`int`	any integral values between -2147483648 (-2^{31}) and 2147483647 ($2^{31}-1$)
`double`	a finite number of approximations of real values between -1.8×10^{308} and $+1.8 \times 10^{308}$, and representations for positive and negative infinities
`boolean`	either `true` or `false`
`char`	any valid Unicode character (65536 possible values)
`byte`	any integral value between -128 and 127
`long`	any integral value between -2^{63} and $2^{63}-1$

Values for variables, literal forms

Values can be stored in variables using either computations from other values and variables (eg. `a + b`), or using a *literal form*: a value encoded directly in the program.

Like for other language constructs, literal forms have a syntax and a semantics.

Some examples:

Literal integers:

Syntax: [-]? <digit> [<digit>]*

(An optional minus sign, followed by one or more digits)

Semantics: the conventional arithmetic interpretation of the literal digits in base 10.

Examples: `-123`, `0`, `456`.

Literal Booleans:

Syntax: `true`

Semantics: the Boolean "truth" value.

Syntax: `false`

Semantics: the Boolean "falsehood" value.

Literal doubles:

Syntax: [<sign>]? [<integer>]? [. <integer>]? [`e` [<sign>]? <integer>]? [`d`]?

(An optional sign, followed by an optional integer, followed optionally by a dot and an integer, followed optionally by "`e`", an optional sign and an integer, the entire form optionally terminated by the letter "`d`"; the `d` is only optional if there is a dot "`.`" already in the form)

Semantics: the conventional scientific interpretation of the number in base 10; with the number after `e` used as a power multiplier.

Examples: `0d`, `-1d`, `.5d`, `0.123`, `3.14e20` (means 3.14×10^{20}).

Assignments and type compatibility, casts

What happens if the value on the right side of an assignment is not part of the set of possible values for the variable on the left?

For example:

```
int x;
x = 3.1415;
```

(the `double` approximation of 3.1415 is not an integral number, so it cannot directly fit in `x`)

In general, if the value on the right is of a different type than the value on the left, the following rules apply:

- if the set defined by the type on the right is a subset of the set defined by the type on the left, we say that "the type on the right is *compatible* with the assignment" and the assignment works as expected. For example:

  ```
  double y;
  y = 123;
  ```

  here the value of "`123`" has type `int` which is different from `double` on the left, but since the set of values for `int` is contained in the set of values for `double` the assignment can proceed silently.

- if the set on the right is not included in the set on the left, the language processor *rejects* the assignment with a conversion error. This is what happens with "`int x = 3.1415;`" above.

Sometimes, the need arises to *force an assignment*: when the programmer knows better than the language that a specific expression on the right will always evaluate to a value that happens to fit on the left, or when the programmer wants to force an approximation of a value.

The main Java construct to force an assignment is inherited from C and C++, and defined as follows:

Cast expression:

Syntax: `(` <type> `)` <expression>

(An opening parenthesis, followed by a type, followed by a closing parenthesis, followed by an expression)

Semantics: the expression on the right is evaluated. Then its value is forcefully transformed into the type specified on the left. The resulting value is compatible with the new type.

The conversion can incur an approximation, for example converting from `double` to `int` will round the number by removing the digits after the comma.

Example:

```
int x;
x = (int)3.1415;
```

The value of "3.1415" is first forced into type `int` by rounding down to 3, so it becomes compatible with the assignment which can then proceed silently.

Assignments and declarations

The simple assignment statement in Java is defined with a semicolon:

Assignment statement:

Syntax: <identifier> `=` <expression> `;`

(An identifier, followed by one "`=`" sign, followed by an expression, followed by a semicolon)

Semantics: evaluate the expression on the right, then assign the result to the variable designated by the name on the left.

Using the constructs seen so far it becomes easy to read the following examples:

```
int i;
i = 123;

double x;
x = 3.1415;
```

This combination of a declaration followed by an assignment is so common that Java also provides a *combined form*. This is defined as follows:

Combined declaration and assignment:

Syntax: <type> <identifier> `=` <expression> `;`

Semantics: this form is equivalent to: <type> <identifier> `;` <identifier> `=` <expression"

`;`

Examples: `int i = 123;` or `double x = 3.1415;`

There are many combined forms in Java that correspond to the combinations of constructs most often used by Java programmers. We will introduce many of them step by step during the course.

Another combination that is so common it deserves mention here already is when multiple variables are declared together with the same type:

```
int i = 1;
int j;
int k = 2;
```

For this Java proposes the following:
Combined declaration of multiple variables:

Syntax: <type> <identifier> [= <expression>]? [, <identifier> [= <expression>]?]* ;

(A type, followed by an identifier, followed optionally by = and an expression, followed by a repetition of zero or more occurrences of a comma and an identifier followed optionally by = and an expression, the whole eventually terminated by a semicolon)

Semantics: equivalent to <type> <identifier> [= <expression>]? ; repeated once for every identifier-expression pair in the comma-separated list.

Example: `int i = 1, j, k = 2;` is equivalent to the 3 declarations above.

Important concepts

- *variables* and *types*
- *set of allowed values for a type*
- *built-in* types
- `int`, `double`
- type *compatibility* in assignments (general concept)
- *cast expression*
- *combined form* (general concept)
- *combined declaration and assignment*
- *combined declaration of multiple variables*

Further reading

- Absolute Java, section 1.2 (pp 16-23) and sections 3.1-3.2 (pp. 25-27)
- Introduction to Programming, sections 2.2.2 and 2.2.3 (pp. 25-27)

Copyright and licensing