CSE 1710

Lecture 3

The assigned reading was:

- The Assignment Statement (sec 1.3, pp. 25-34)
- IMD 3.2 (p. 117) *try to get the gist of the idea*

Who completed the readings?

What was the take-away?

did it relate to theory?

did it relate to concept?

did it relate to praxis?

[KC 1.8]

A Java **compiler** reads the Java program from the **source file** and produces a **bytecode** program in the **class file**.

1. the compiler is an app – what is its name?
2. what language was used to implement the compiler?
3. how do we (in this course) get our Java programs compiled?

[KC 1.8]

The compiler produces **compile-time errors** when a statement contains **syntax** or **semantic** errors.

1. how are the compile-time errors presented to us?
2. what is the difference between a syntax error and a semantic error?
3. does the presence of syntax or semantic errors mean our program is **incorrect**?

[KC 1.8]

Addendum: syntax vs semantic errors

syntax error: a statement that does not comply with what is acceptable to the compiler

- e.g., semicolon missing, expressions incorrectly constructed (operators missing an operand, non-balanced parenthesis)

semantic errors: the statements are ok, but there is a problem in how they are used

- e.g., use of a non-initialized variable
- assigned a value to a variable that has an incompatible type (`int a = 3.0;`)

[KC 1.8]

The **virtual machine (VM)** is a program that … reads one bytecode instruction at a time, translates it to machine language, and executes it.

1. the VM is an app – what is its name?
2. what language was used to implement the VM?

[KC 1.9]

A byte is the smallest addressable unit of memory. This unit has a content and an address.

The VM makes use of this memory when the app it is running needs to represent values or otherwise use memory.

quick tip! refer back to lecture 02 for more on this

1. Identify the lexical elements below
2. Are these lines valid *expressions*?
3. Are these expressions valid *statements*?

```
4 + 4
4.0 + 4.0
4 + 4.0
8 = 4 + 5
```

A Note about Statements

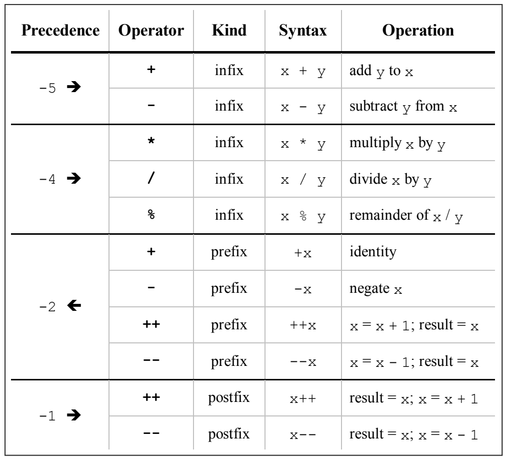
- any statement can be described in terms of the 5 language elements (Fig 1.4)
  - keywords, identifiers, literals, operators, separators
- there are 5 types of statements (JD 1.3)
  - declaration (for variables, for class and methods too)
  - assignment
  - usage of other classes
  - flow control (condition, iteration, branching, exception)

Let's examine these two statements

```
int x = 4 + 4;
int y = 4.0 + 4.0;
```

Let's identify which operator is being used…

there are actually 4 different addition operators
they are all denoted +
one for each of int, long, float, double
what about byte, short, char?



Precedence	Operator	Kind	Syntax	Operation
-5 →	`+`	infix	`x + y`	add `y` to `x`
	`-`	infix	`x - y`	subtract `y` from `x`
-4 →	`*`	infix	`x * y`	multiply `x` by `y`
	`/`	infix	`x / y`	divide `x` by `y`
	`%`	infix	`x % y`	remainder of `x` / `y`
-2 ←	`+`	prefix	`+x`	identity
	`-`	prefix	`-x`	negate `x`
	`++`	prefix	`++x`	`x` = `x + 1`; result = `x`
	`--`	prefix	`--x`	`x` = `x - 1`; result = `x`
-1 →	`++`	postfix	`x++`	result = `x`; `x` = `x + 1`
	`--`	postfix	`x--`	result = `x`; `x` = `x - 1`

[KC 1.10]

A **type** is a specification of allowed data values and their associated operations.

`int` is a specification of a set of 4294967296 representations and a set of associated operators:
`+ - * / % ++ --`

`char` is a specification of a set of 65536 representations and NO set of associated operators

`byte` is a specification of a set of 256 representations and NO set of associated operators

[KC 1.10]

A **type** is a specification of allowed data values and their associated operations.

`int` is a specification of a set of 4294967296 representations and a set of associated operators:
`+ - * / % ++ --`

`char` is a specification of a set of 65536 representations and NO set of associated operators

`short` is a specification of a set of 256 representations and NO set of associated operators

Keep in mind, in addition to the arithmetic operators: `+ - * / % ++ --`

there are many OTHER kinds of operators

boolean operators

relational operators

the ***cast*** operator

[KC 1.13]

Primitive types are types whose names are keywords and whose operations are operators.

So are there types other than primitive types?

Are their operations also operators?

What programming constructs do we know?

- that there are *subroutines* (aka "methods")
  - e.g. `println`
    - refers to a portion of code within a larger program
    - it performs a specific task
    - it is relatively independent
  - to make use of `println`, we need a statement from category #3 ("usage of other classes")

any value must have a *type*

this type determines:

- a set of possible values
- a set of operations on those values
- a set of behaviours for "border" cases

About expressions …

- expressions get **evaluated** to obtain a value
- the goal (PRAXIS):
  - given an expression, can you determine the *type* of the evaluated value?
  1. it can be straightforward and quick
  2. it can be straightforward and not so quick
  3. it can be tricky because you need to remember some anal rules about the compiler

the compiler is **anal**:

- it insists that all variables must have a type (thus, you must declare them)
  - BREAK THE RULE?
    SEMANTIC ERROR! NO BYTECODE!
- for any expression, the operation being attempted **must be defined** for the types of the operands
  - NOT DEFINED? can the compiler find a solution?
    - » YES? eg – it can possibly auto-promote the operands so that the operator becomes defined? OK!
    - » NO? SEMANTIC ERROR! NO BYTECODE

[KC 1.14]

three things give us a *value*: [p. 25]

- literals (each has a type)
  - 3, 3.0, 3.0f, ‘A’, false, 28l, 28L, 5.9F
  - no literals for byte, short [p.17]
  - what’s the point of having 28 ***and*** 28L ?
- variables
  - easy to know the type – it was declared previously!
- expressions
  - the **type** of its value is not so immediately apparent, since the compiler may perform automatic promotion

What is *closure*

- a *type* is: [IMD 1.6]
  - a set of values **and**
  - the operations that can be performed on the values
- an operator has *closure* if the result of that operator belongs to the same set of values as the operands

Recap about *closure*

- **arithmetic** operators have the property of *closure*
  - the result (`numeric`) will be the same type as the operands (`numeric`) [p. 25-26, 29]
- **boolean** operators have the property of *closure*
  - the result (`boolean`) will be the same type as the operands (`boolean`) [p. 180]
- **relational** operators <u>do not</u> have the property of *closure*
  - the result (`boolean`) will not be the same type as the operands (`numeric`) [p. 180]

Expression Evaluation

- What if the expression has:
  - **numeric** operands of the same type, and
  - arithmetic operators of the same precedence and association (left-to-right)
    - it is straightforward and quick to determine the type of the value of the expression
      - the type will be the same
      - for the value, apply operators from left-to-right
  - exceptions: `byte`, `short`, `char` [no operators, p. 29]
- If the expression has arithmetic operators of different precedence levels
  - need to apply operators in order of precedence level

Precedence	Operator	Kind	Syntax	Operation
`-5` →	`+`	infix	`x + y`	add `y` to `x`
	`-`	infix	`x - y`	subtract `y` from `x`
`-4` →	`*`	infix	`x * y`	multiply `x` by `y`
	`/`	infix	`x / y`	divide `x` by `y`
	`%`	infix	`x % y`	remainder of `x` / `y`
`-2` ←	`+`	prefix	`+x`	identity
	`-`	prefix	`-x`	negate `x`
	`++`	prefix	`++x`	`x` = `x + 1`; result = `x`
	`--`	prefix	`--x`	`x` = `x - 1`; result = `x`
`-1` →	`++`	postfix	`x++`	result = `x`; `x` = `x + 1`
	`--`	postfix	`x--`	result = `x`; `x` = `x - 1`

Example (straigtforward)

```
5 + (4 - 3) / 5 - 2 * 3 % 4
```

Example (straigtforward)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
```

Example (straigtforward)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
        ↑       ↑   ↑
```

Example (straigtforward)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
= 5 + 0 - 2 * 3 % 4
```

Example (straigtforward)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
= 5 + 0 - 2 * 3 % 4
               ↑   ↑
```

Example (straigtforward)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
= 5 + 0 - 2 * 3 % 4
= 5 + 0 - 6 % 4
```

Example (straigtforward)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
= 5 + 0 - 2 * 3 % 4
= 5 + 0 - 6 % 4
               ↑
```

Example (straigtforward)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
= 5 + 0 - 2 * 3 % 4
= 5 + 0 - 6 % 4
= 5 + 0 - 2
```

Example (straigtforward)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
= 5 + 0 - 2 * 3 % 4
= 5 + 0 - 6 % 4
= 5 + 0 - 2
```

Example

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
= 5 + 0 - 2 * 3 % 4
= 5 + 0 - 6 % 4
= 5 + 0 - 2
= 5 - 2
```

Example (straigtforward)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
= 5 + 0 - 2 * 3 % 4
= 5 + 0 - 6 % 4
= 5 + 0 - 2
= 5 - 2
= 3
```

Expression Evaluation

- What if the expression has:
  - **numeric** operands of different types and
  - arithmetic operators
    - it is relatively straightforward to determine the type of the value of the expression
    - key thing: remember the promotion rules!

Example (auto promote)

```
   5 / 2 + 2.5
```

Example (auto promote)

```
   5 / 2 + 2.5
 = 2 + 2.5
```

Example (auto promote)

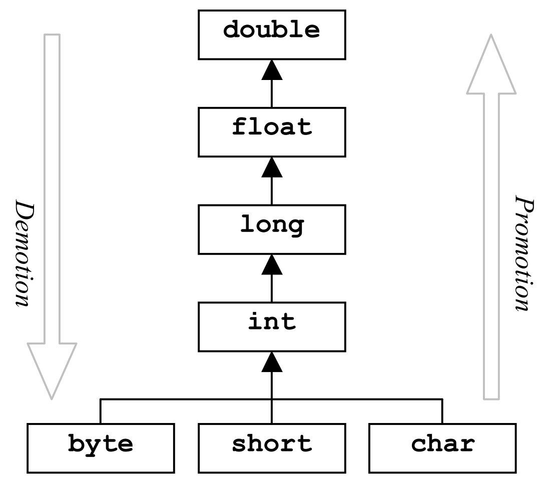
```
   5 / 2 + 2.5
 = 2 + 2.5
```

here we have an `int` operand and a `double` operand

there is a + operator for:
- two `int` operands,
- two `long` operands
- two `double` operands,
- two `float` operands

but there is **no** + operator defined for an an `int` operand and a `double` operand

So what happens?



Example (auto promote)

```
  5 / 2 + 2.5
= 2 + 2.5
= 2.0 + 2.5
```

auto promotion

Example (auto promote)

```
  5 / 2 + 2.5
= 2 + 2.5
= 2.0 + 2.5
= 4.5
```

Expression Evaluation

- The expression has manual promotions and demotions
  - **cast operator** has precedence level (-3) and association right to left

Example (cast operator)

```
  (double) 5 / 2 + (int) 2.5
```

Example (cast operator)

```
  (double) 5 / 2 + (int) 2.5
= (double) 5 / 2 + 2
```

Example (cast operator)

```
  (double) 5 / 2 + (int) 2.5
= (double) 5 / 2 + 2
= 5.0 / 2 + 2
```
manual promotion

Example (cast operator)

```
  (double) 5 / 2 + (int) 2.5
= (double) 5 / 2 + 2
= 5.0 / 2 + 2
= 5.0 / 2.0 + 2
```
auto promotion

Example (cast operator)

```
    (double) 5 / 2 + (int) 2.5
  = (double) 5 / 2 + 2
  = 5.0 / 2 + 2
  = 5.0 / 2.0 + 2
  = 2.5 + 2
```

Example (cast operator)

```
    (double) 5 / 2 + (int) 2.5
  = (double) 5 / 2 + 2
  = 5.0 / 2 + 2
  = 5.0 / 2.0 + 2
  = 2.5 + 2
  = 2.5 + 2.0
```
auto promotion

Example (cast operator)

```
    (double) 5 / 2 + (int) 2.5
  = (double) 5 / 2 + 2
  = 5.0 / 2 + 2
  = 5.0 / 2.0 + 2
  = 2.5 + 2
  = 2.5 + 2.0
  = 4.5
```