CSE 1710

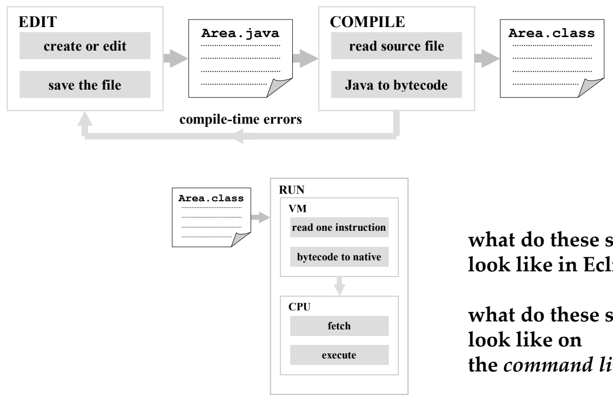
Lecture 3

Variables, Values, Operands, Operators

To end up with a functioning program, we need to:

1. design it:
   - what should it do? (what should it print out?)
2. implement it:
   - need to know programming constructs
   - need to understand correct syntax
   - need to adhere to a style guide
3. get byte code
   - give the text to the compiler
4. run it:
   - give the byte code to a virtual machine

About Steps 3 and 4…



what do these steps look like in Eclipse?

what do these steps look like on the *command line*

Command Line Interpreters (CLIs)

- Windows – CMD.EXE
  - on the machines in CSE1002
    - Start -> Run… -> Type in "cmd.exe"
      - default directory is `C:\Documents and Settings\cseXXXXX`
    - Desktop shortcut "VS 6 CMD Prompt"
      - default directory is `C:\WINDOWS\system32`
    - Desktop shortcut "Fortran Command Prompt"
      - default directory is `Z:\York`

default Eclipse workspace:
`Z:\workspace-win32`

Command Line Interpreters (CLIs)

- Also on the machines in CSE1002,
  - PuTTY (ssh client)
    - Host Name: `red.cse.yorku.ca`
    - Click yes to confirm RSA key, if necessary
    - Log in with the same credentials (login and passwd)
    - you will be given a *unix shell* in a terminal window

  default Eclipse workspace:
  `/??/??/cseXXXXX/workspace-win32`

Command Line Interpreters (CLIs)

- Mac OS X
  - Terminal
    - Invoke the application "ssh"
    - `ssh red.cse.yorku.ca`
    - Click yes to confirm RSA key, if necessary
    - Log in with the same credentials (login and passwd)
    - you will be given a *unix shell* in the terminal window

  default Eclipse workspace:
  `/??/??/cseXXXXX/workspace-win32`

What programming constructs do we know?

- there is something called a *statement*
  - all statements are made up of the 5 language elements (Fig 1.4)
    - keywords, identifiers, literals, operators, separators
  - there are 5 types and they do different things (JD 1.3)
    - declaration, assignment, usage of other classes, flow control, other
    - declaration statement is not for only variables, but also for class and method headers

```
import java.io.PrintStream;
public class Area
{
   public static void main(String[] args)
   {
        PrintStream output;
        output = System.out;
        int width;
        width = 8;
        int height = 3;
        int area = width * height;
        output.println(area);
   }
}
```

Keywords, Identifiers, Literals, Operators, Separators

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”)

What programming constructs do we know?

- that any value must have a *type*
  - a set of possible values
  - a set of operations on those values
  - the compiler is anal:
    - all variables must have a type (thus, you must declare them)
    - the operation being attempted must be defined for the types of the operands
      - compiler will auto-promote if possible

What programming constructs do we know?

- how to *iterate*, how to *select*
  - we have a very basic idea
  - we still need to practise

What programming constructs do we know?

- three things give rise to a *value*: [p. 25]
  - literals
    - know the type for each:
      - `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…

About expressions …

– expressions get **evaluated** to obtain a value
– the goal:
  • given an expression, 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

Recap about *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

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

Example

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

Example

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

Example

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

Example

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

Example

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

Example

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

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
```

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
    ↑   ↑
```

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

```
  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

```
5 / 2 + 2.5
```

Example

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

Example

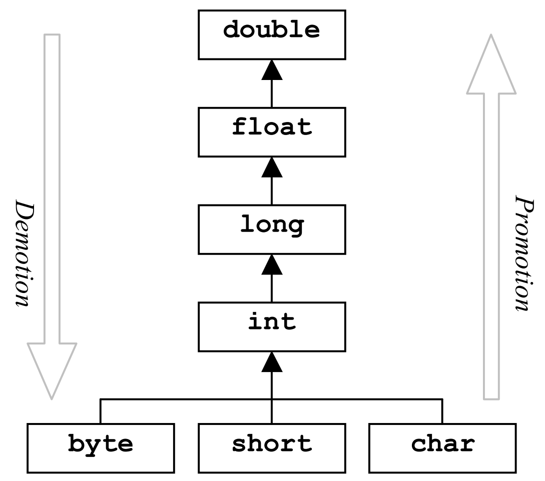
```
  5 / 2 + 2.5
= 2 + 2.5
```

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

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

auto promotion

Example

```
  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

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

Example

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

Example

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

Example

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

Example

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

Example

```
  (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

```
  (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
```