COSC 1020

Yves Lespérance

Lecture Notes

Week 2 — Computer Software

Recommended Readings:

Horstmann: Ch. 1 Sec. 4 to 10, Ch. 3 Sec. 2, 4, 6 & Ch. 10 Sec. 1

Lewis & Loftus: Ch. 1 Sec. 3 & 4, Ch. 10 Sec. 0 & Ch. 2 Sec. 3 to 5

The Software Development Process

Often taken to have 5 phases:

- *Analysis:* decide *what* system should do;
  $\rightarrow$ requirements document.
- *Design:* create "blueprint" for system;
  OO design $\rightarrow$ description of classes, methods, & relationships, API.
- *Implementation:* write code & compile;
  $\rightarrow$ working program.
- *Testing:* run tests to verify that program works;
  $\rightarrow$ report on test results.
- *Deployment:* program is installed & used.

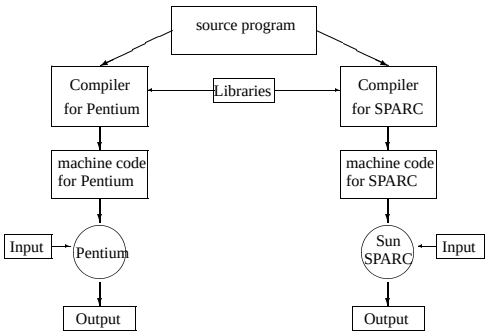
Waterfall model: 5 phases in sequence.

But need to iterate phases. Development Process Models.

Software Life Cycle = Development + Maintenance.

Compiling a Program

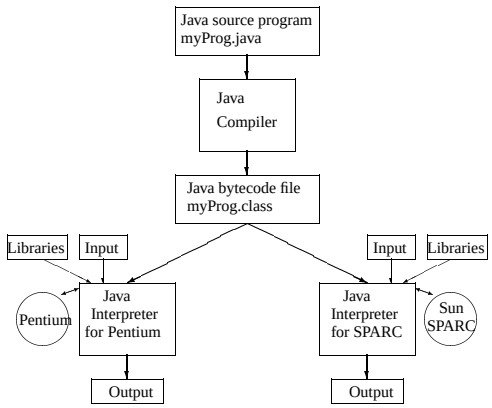
A computer can only execute machine instructions, e.g. "load integer at memory address 40"). Before a program in a high-level language can be executed it must be translated into machine code. This is called *compiling* the program. In languages such as C, C++, Pascal, etc. this works as follows:



Since the machine code for different types of computers (Pentium, Sun SPARC, Mac) is different, you need as many different compiled versions of the program as types of computers you want to handle — not very portable.

Compilation and Execution in Java

Java uses a different approach. A Java program is compiled into machine instructions (bytecode) for the Java virtual machine, an idealized CPU. The bytecodes are then executed by an interpreter that simulates the virtual machine on the specific type of computer used.



This is more portable and more secure.

Steps in Implementing a Program

1. Type the program into a file using an editor such as `nedit` — see Lab #1 for details. Save it into a file whose name ends in `.java`, e.g. `Hello.java`.

2. Run the Java compiler on this source code file, e.g.

```
javac Hello.java
```

If the compiler produces error messages, go back to 1 and fix them. Otherwise, the compiler produces a bytecode file(s), e.g. `Hello.class`.

3. Execute the program by running the Java interpreter on the bytecode file, e.g.

```
java Hello
```

Test the program by providing input data as required. If the program ends with a run-time error message or does not produce the expected results, figure out what the error (bug) is, go back to 1, and fix it.

Compilation and Execution E.g.

```
Script started on Wed Sep 18 18:56:22 2002
blue 301 % more Hello.java
import type.lang.*;
public class Hello
{  public static void main(String[] args)
   {  IO.println("Hello, World!");
   }
}
blue 302 % javac Hello.java
blue 303 %
blue 303 % java Hello
Hello, World!
blue 304 %
blue 304 % exit
exit

script done on Wed Sep 18 18:57:24 2002
```


Compilation/Syntax Error E.g.

```
Script started on Wed Sep 18 18:59:42 2002
blue 301 % more Hello.java
import type.lang.*;
public class Hello
{  public static void main(String[] args)
   {  IO.printl("Hello, World!");
   }
}
blue 302 % javac Hello.java
Hello.java:4: cannot resolve symbol
symbol  : method printl  (java.lang.String)
location: class type.lang.IO
   {  IO.printl("Hello, World!");
        ^
1 error
blue 303 % exit
exit

script done on Wed Sep 18 19:00:27 2002
```


Types of Errors

- *Syntax/Compilation errors:* program violates rules of syntax of programming language; detected during compilation.
- *Run-time errors:* abnormal conditions detected during program execution, e.g. division by 0; can cause execution to be aborted, but can also be caught as exceptions and handled.
- *Logic errors:* program appears to work but does not do what it is supposed to do. Hardest to detect and fix.

Fixing errors/bugs is called *debugging*.

Debugging tools, tracing.

Assigning a Value to a Variable

The most common way of putting a value into a variable is with an *assignment statement*. E.g.

```
int age1, age2;
age1 = 11;
age2 = age1;
age1 = age1 + age2 + 5;
```

The left-hand-side should be a variable, because variables are things that can hold a value.

The right-hand-side may be a literal, a variable, or an expression. Its type must be one that is compatible with that of the variable as specified in the declaration (either the same or one that is automatically promoted). So for example: "`age1 = "XYZ";`" is an error.

Arithmetic Expressions

Arithmetic expressions are written much like you'd expect (`*` is used for multiplication), e.g.

```
2 * x * x + 3 * x - 5
(5.0 / 9.0) * (degF - 32)
```

One peculiarity is with the division operator `/`. It does an integer division (discarding the remainder) when both its arguments have a whole number type, otherwise it does a real division, e.g.

`13 / 5` evaluates to 2
`13.0 / 5` evaluates to `2.6`

The `%` operator is used to find the remainder in an integer division, e.g.

`13 % 5` evaluates to 3

The normal rules of precedence are used in interpreting arithmetic expressions:

operators	precedence
unary -	high
* / %	medium
+ binary -	low

Operators in the same precedence class are evaluated left to right.

You can of course get a different order of evaluation by adding ()s. E.g.

Expression	Value
`14 - 8 / 2 + 1`	
`(14 - 8) / 2 + 1`	
`(14 - 8) / (2 + 1)`	
`10 - 5 - 3`	

Many other useful mathematical operations are provided as static methods of the `Math` class, e.g. `Math.sqrt(x)`, `Math.sin(x)`, etc.

Note that arithmetic operations on floating-point numbers are rarely exact. E.g.

```
1.0 / 3.0 + 2.0 / 3.0
```

will produce a value that is slightly less than than `1.0`.

Even expressions (e.g. `0.5 + 0.35`) that look like they should produce the exact result don't because numbers are represented in binary (`0.35` is `0.01 0110 0110...`).

One must be content with an approximate result. It is often possible to calculate a bound on the error.

Type Promotion and Casting

In general, a value can only be assigned to a variable if its type is the *same* as that of the variable. As well, an operator or method can only be applied to the type of data it is defined on.

However Java will automatically *promote* a value from a smaller numerical type (e.g. `int`) to a larger numerical type (e.g. `double`). For e.g. in evaluating the arithmetic expression

```
2.5 + 3
```

the integer 3 will be promoted to the double `3.0`.

This also applies to the use of methods; if you call a method taking an argument of type `double` on an `int` argument, it will be promoted to `double` automatically.

And this also applies to assignments, e.g. in

```
double x = 3;
```

the 3 will automatically be promoted to `3.0` before being assigned to `x`.

However, a value of a larger type is never automatically converted to a value of a smaller type. If you want to avoid a type mismatch error, you must use a *type cast*, e.g.

```
double x = 3.5;
int n = (int) x;
```

Here, the value `3.5` will be *truncated* to `3` (i.e. the fraction part discarded) before being assigned to `n`.

If the value being converted cannot be represented as a value of the smaller type, an error (exception) will occur.

In many cases, you will want to round up the value being converted to an integer, e.g.

```
int n = (int)(x + 0.5);
```

or

```
int n = (int)Math.round(x);
```

A type cast can also be used perform a real division, e.g. `(double) 5 / 2` returns `2.5`.

An Example Application Calculating Change

```
// in file MkChange.java

import type.lang.*;

public class MkChange
{  public static void main(String[] args)
   {  final int QUARTER_VALUE = 25;
      final int DIME_VALUE = 10;
      final int NICKEL_VALUE = 5;
      IO.print("Enter the amount in cents: ");
      int amount = IO.readInt();
      int nQuarters = amount / QUARTER_VALUE;
      amount = amount % QUARTER_VALUE;
      int nDimes = amount / DIME_VALUE;
      amount = amount % DIME_VALUE;
      int nNickels = amount / NICKEL_VALUE;
      int nPennies = amount % NICKEL_VALUE;
      IO.print("Change is ");
      IO.print(nQuarters + " quarters, ");
      IO.print(nDimes + " dimes, ");
      IO.print(nNickels + " nickels, and ");
      IO.println(nPennies + " pennies.");
   }
}
```

A sample run:

```
Script started on Wed Sep 18 19:06:08 2002
blue 301 % javac MkChange.java
blue 302 % java MkChange
Enter the amount in cents: 97
Change is 3 quarters, 2 dimes, 0 nickels, and 2 pennies.
blue 303 % java MkChange
Enter the amount in cents: 68
Change is 2 quarters, 1 dimes, 1 nickels, and 3 pennies.
blue 304 % exit
exit

script done on Wed Sep 18 19:06:57 2002
```


Applications

`MkChange` is an example of an application (app), a class that is meant to be run by the java interpreter to provide a service to an end user.

An app is a class that contains one and only one `main` method and typically has the following form:

```
public class AppClassName
{  public static void main(String[] args)
   {  // variable and constant declarations
      // input some values
      // perform some calculations
      // output some values
   }//end main method

   //possibly other static methods used by main

}//end class AppClassName
```

The `main` method is the one that runs first when the app starts executing.

An app generally uses methods from classes defined by other programmers, e.g. `MkChange` uses `IO.println`. But an app does not provide anything that can be used by other programmers.