CM1210

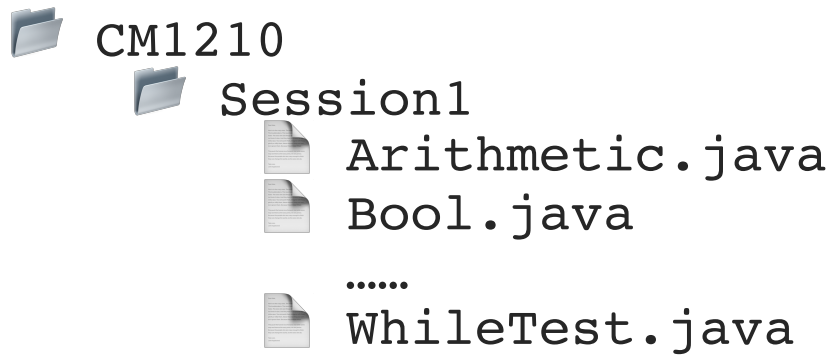
Object Oriented Java Programming

SESSION 1 [TEACHING WEEK 1]

INTRODUCTION TO JAVA
(USING OUR KNOWDLEGE OF PYTHON)

Download Materials

- Download the code examples for this session from the CM1210 module on Learning Central.
- Locate the archive called **Session1.zip** in the "Learning Materials" folder and download it to your H: drive.
- I'd suggest creating a folder on you H: drive called "CM1210", in which you can store all of these files as you get them.
- Unzip the session1.zip file into your newly created folder, so you have the following structure:



Programs and Executing Them

Java

Programs created within source files (**.java** extension). A source file compiles into one or more executable byte code files (**.class** extension).

Programs must first be **compiled** before they can be **executed**. Java programs execute in an interpreter called a Java virtual machine (JVM).

Programs are either **applets** or **applications**. Applets are programs that run in a Web browser. Applications are programs that run on a standalone computer.

Python

Programs created with a source file (**.py** extension) and possibly a byte code file (**.pyc** extension).

An **compiler/interpreter** called a Python virtual machine (PVM) translates Python source files to byte code before execution.

Running a Program

Java

EXAMPLE: HelloWorld.java

A Java application must first be **compiled**, as follows:

> javac HelloWorld.java

The byte code file (**.class** extension) can then be run as follows:

> java HelloWorld

Note: Syntax and type errors are caught at **compile** time. All other errors are caught at **run** time.

Python

A Python program can be run as a script from a command prompt, as follows:

```
> python helloworld.py
```

Note: Syntax and type errors are caught at run time.

First Program: Hello World!

Java

```
public class HelloWorld{
    static public void main(String[] args){
        System.out.println("Hello world!");
    }
}
```

At program startup, the compiled class **HelloWorld** is loaded into the JVM. The JVM always calls the **main** method by default.

A Java application must include at least one class that defines a **main** method. The byte code file for this class is the entry point for program execution.

Python

Can consist of a single statement:

```
print("Hello world!")
```

The PVM simply executes this statement.

Alternatively, within the **main** function and then called that function:

```
def main():
    print("Hello world!")

main()
```

The PVM executes the function definition and then calls the function.

Lexical Structure

Java

Literals include numbers, characters, and strings.

Identifiers include the names of variables, classes, interfaces, and methods.

Reserved words include those of the major control statements (**if**, **while**, **for**, **import**, etc.), operators (**instanceof**, **throw**, etc.), definitions (**public**, **class**, etc.), special values (**true**, **false**, **null**, **this**, **super** etc.), and standard type names (**int**, **double**, **String**, etc.).

Python

Literals include **numbers**, strings, tuples, lists, and dictionaries.

Identifiers include the names of variables, classes, functions, and methods.

Reserved words include those of the major control statements (**if**, **while**, **for**, **import**, etc.), operators (**in**, **is**, etc.), definitions (**def**, **class**, etc.) and special values (**True**, **False**, **None**, etc.).

Syntactical Structure

Java

Indentation is not significant, so all lexical items are separated by zero or more spaces.

Blocks of code in statements and definitions are enclosed in curly braces (**{}**).

Simple statements end with a semicolon (**;**).

Boolean expressions in loops and **if** statements are enclosed in parentheses.

Python

Lexical items on a single line are separated by zero or more spaces.

Indentation is significant and is used to mark syntactical structures, such as statement blocks and code within function, class, and method definitions.

A **phrase** can be broken and continued on the next line after a comma, or by using the **'\'** symbol.

The **headers** of control statements and function, class, and method definitions end with a colon (**:**).

Program Comments

EXAMPLE: Comments.java

Java

An **end of line comment** begins with `//`.

```
// This is an end of line comment
```

A **multi-line comment** begins with `/*` and ends with `*/`.

```
/*
This is a multi-line
comment.
*/
```

Python

An **end of line comment** begins with the # symbol.

```
# This is an end of line comment
```

A **multi-line comment**, also called a **docstring**, begins with """ and ends with """.

```
"""
This is a docstring or
multi-line comment.
"""
```

Data Types

Java

There are two broad categories of data types: **primitive types** and **reference types**.

The **primitive types** include the numeric types (**char**, **byte**, **short**, **int**, **long**, **float** and **double**) and **boolean**.

Reference types are classes. Thus, any object or instance of a class is of a reference type. These include **strings**, **arrays**, **lists**, **maps**, and so forth.

NOTE: Java is a **STRICTLY TYPED** language.

Python

All **data values**, including functions, are **objects**. Thus, all data types are **reference types**.

Primitive Data Types

EXAMPLE: PrimitiveTypes.java

Java

Type	Kind	Example	#bits	Range
boolean	logical	true false	1	
char	integer	'', '0', 'A'	16	u0000,. . . ,uFFFF
byte	integer	0, 1, -1, 117	8	max = 127
short	integer	0, 1, -1, 117	16	max = 32767
int	integer	0, 1, -1, 117	32	max = 2147483647
long	integer	0, 1, -1, 117	64	max = 9223372036854775807
float	floating-point	-1.0f, 0.499f, 3E8f	32	$\pm 10^{-38} \ldots \pm 10^{38}$
double	floating-point	-1.0f, 0.499f, 3E8f	64	$\pm 10^{-308} \ldots \pm 10^{308}$

Python

int represents integers ranging from -2^{31} through $2^{31}-1$.

long represents very large integers to the extent supported by the machine's memory.

float represents floating-point numbers with 16 digits of precision.

Scope

EXAMPLE: Scope.java

The ***scope*** of a variable is the region of the program in which you can refer to the variable by its name. In Java scope is determined by curly braces {}.

```
{
    int x = 12; // Only x available
    {
          int q = 96; // x and q available;
    }
    // AT THIS POINT - Only x available
    // q is out of scope
}
```

Strings and Characters

EXAMPLE: PrimitiveTypes.java

Java

String literals are formed using the **double** quotes as delimiters.

Strings are instances of the **`String`** class.

Character literals are formed using the **single** quotes as delimiters.

Characters are values of the **`char`** data type. This type uses 2 bytes to represent the Unicode set.

An **escape sequence**, either as a character or as a string, is formed in the same was a Python.

Python

String literals are formed using either the **single** quotes **or double** quotes as delimiters.

Strings are instances of the **`str`** class.

Character literals are simply strings that contain a single character.

An **escape sequence** is formed using the **`'\'`** character followed by an appropriate letter such as **`'n'`** or **`'t'`**.

String Concatenation

EXAMPLE: Concatenation.java

Java

The **concatenation operator** **+** joins together two strings to form a third, new string. **Note:** As long as one of the operands is a string, the other operand can be of any type.

```
"matt" + " loves Java."
99 + " red balloons."
```

Any non-numeric object can also be used in a string concatenation, because all Java objects utilise the **`toString()`** method, which returns the name of the object's class by default, but can be overridden to return a more descriptive string. If **`x`** and **`y`** are objects, the code:

`x.toString() + y.toString()` or **`x + y`**

concatenates their string representations.

Python

The **concatenation operator** **+** joins together two strings to form a third, new string:

```
"matt" + " loves Python."
str(99) + " red balloons."
```

If **`x`** and **`y`** are any objects, the code:

```
str(x) + str(y)
```

will concatenate their string representations.

Converting Between Types

EXAMPLE: Casting.java

Java

Numeric types can be **converted** to other numeric types by using the appropriate **cast operators**. A cast operator is formed by enclosing the destination type name in **parentheses**.

```
(int) 6.73
(double) 5
(char) 65
(int) 'm'
```

Note: When casting from an integer to a character or vice versa, the integer is assumed to contain the character's Unicode value.

An easy way to convert any value to a string is to concatenate it with an empty string:

```
"" + 6.73
"" + 65
```

Python

Numeric types can be **converted** to other numeric types by using the appropriate type conversion functions:

```
int(6.73)
float(5)
```

The **ord** and **chr** functions are used to convert between integers and characters:

```
ord('M')
chr(65)
```

str function converts any Python object to its corresponding string representation:

```
str(33)
str(6.73)
```

Arithmetic and Math Functions

EXAMPLE: Arithmetic.java

Java

Arithmetic operators include **+**, **–**, *****, **/** and **%**.

Two integer operands yield an integer result. Given at least one **`float`** operand, a **`float`** will result.

The **`Math`** class includes class methods such as **`round`**, **`max`**, **`min`**, **`abs`**, and **`pow`**, as well as methods for trigonometry, logarithms, square roots, and so forth.

```
Math.round(6.73)
Math.sqrt(5)
```

Python

Arithmetic operators include **+**, **–**, *****, **/**, **%**, **//** and ******.

Two integer operands yield an integer result, usless you use **/**, which yields a **float** result. Given at least one **`float`** operand, a **`float`** will result. **//** yields an integer quotient.

`max`, **`min`**, **`abs`**, and **`round`** are standard functions.

The **`math`** module includes standard functions for trigonometry, logarithms, square roots, etc.:

```
Math.round(6.73)
Math.sqrt(5)
```

Comparisons and Comparables

Java

EXAMPLE: Comparing.java

The **comparison operators** are **==**, **!=**, **<**, **>**, **<=**, and **>=**.

All comparison operators return **`True`** or **`False`**.

All values of **primitive types** are comparable.

Values of **reference types** are comparable if and only if they implement the **`compareTo`** method. **`compareTo`** returns 0 if the two objects are equal (using the **`equals`** method), a negative integer if the receiver object is less than the parameter object, and a positive integer if the receiver object is greater than the parameter object.

```
String m = "MATT";
System.out.println(m.compareTo("MATE") > 0);
```

Python

The **comparison operators** are **==**, **!=**, **<**, **>**, **<=**, and **>=**.

All comparison operators return **`True`** or **`False`**.

Only objects that are comparable, such as numbers and sequences of comparable objects (strings and lists of numbers or strings), can be operands for comparisons.

```
print("MATT" > "MATE")
```

Boolean Values and Logical Operators

EXAMPLE: Bool.java

Java

The type **boolean** includes the constant values **True** and **False**.

The **logical operators** are **!** (not) **&&** (and), and **||** (or).

Compound Boolean expressions consist of one or more Boolean operands and a logical operator. Evaluation stops when enough information is available to return a value. **!** is evaluated first, then **&&**, then **||**.

Python

The type **boolean** includes the constant values **True** and **False**.

The **logical operators** are **not**, **and**, and **or**.

Compound Boolean expressions consist of one or more **Boolean operands** and a **logical operator**. Evaluation stops when enough information is available to return a value. **not** is evaluated first, then **and**, then **or**.

Method Calls

Java

An **instance method call** consists of an **object reference** followed by a **dot**, the **method name** and a parenthesised **list of arguments**:

```
p.translate(10, 20)
```

A **class method call** consists of a **class name**, followed by a **dot**, the **method name** and a parenthesised **list of arguments**.

```
double d = Math.sqrt(5)
```

All methods are defined to **return** a **specific type of value**. When no return value is needed, the method's return type is **void**. Otherwise, the type of value returned must be compatible, at compile time, with the type of value expected (the example shows an assignment of a **double** to a **double**).

Python

An **instance method call** consists of an **object reference** followed by a **dot**, the **method name** and a parenthesised **list of arguments**:

```
myList.sort()
```

A **class method call** consists of a **class name**, followed by a **dot**, the **method name** and a parenthesised **list of arguments**.

A method **returns** the type of value indicated by its return statement, if one exists. If a method does not explicitly return a value, the value **None** is returned by default. Type compatibilities are resolved at run time.

Import Statements

Java

EXAMPLE: Importing.java

A package resource is **imported**: **`import <package name>.<resource name>;`**

The resource is then **referenced** without the package name as a **qualifier**:

```
import javax.swing.JButton;
JButton b = new JButton("Reset");
```

Alternatively, **ALL** of the package's resources can be **imported** using the form:

```
import <package name>.*;
```

Note: two resources could have the same name in different packages. To use both resources, do not import them but just reference them using the package names as qualifiers.

```
java.util.List<String> names = new java.util.ArrayList<String>();
java.awt.List namesView = new java.awt.List();
```

Python

A module is **imported** using the form: **`import <module name>`**

The resources of that module are **referenced** using the form: **`<module name>.<resource name>`**

```
import math
print(math.sqrt(2), math.pi)
```

Alternatively, an individual resource can be **imported** using the form:

```
from <module name> import <resource name>
```

The resource is then **referenced** without the module name as a qualifier.

```
from math import sqrt, pi
print(sqrt(2), pi)
```

Variable Declarations, Typing and Initialisation

Java

```
<type> <variable>,..., <variable>;
<type> <variable> = <expression>;

int x, y;
x = 1;
y = 2;
int z = 3;
```

A **variable** has a **type**, which is specified when the variable is declared. A variable can only be assigned a value that is compatible with its type. Type incompatibilities are caught at compile time.

All **instance** and **class** variables are given default values when declared. However, the compiler requires the programmer to assign temporary variables within methods a value before they can be referenced. This guarantees all variables have a value at run time.

Python

```
<variable> = <expression>

x = 1
x = x + 6.73
```

A **variable** is created and set to an initial value using an **assignment statement**.

A variable picks up the **type** of the object to which it is currently bound.

Type checking and checking for references to **uninitialised** variables is done at run time.

Assignment Statements

Java

<variable> = <expression>;

```
int x;
x = 1;
x += 1;
```

A **variable** has a **type**, which is specified when the variable is declared. A variable **can only** be assigned a value that is compatible with its type.

Values of less inclusive types can be assigned to variables of more inclusive types. Reversing this order requires explicit **type conversion** before assignment:

```
double d;
d = 55;
int i;
i = (int)6.73
```

Python

```
<variable> = <expression>

x = 1
x = x + 6.73
```

Variables themselves are **typeless**. Any variable can name any thing and be reset to any thing. The object to which a variable refers has a **type**.

break, return and Empty Statements

EXAMPLE: BreakAndContinue.java

Java

The **break** statement exits a loop.

```
while (true)
    break;
```

The **return** statement exits a method. If no expression is specified, the value **void** is returned. The value returned must be type compatible with the method's return type.

```
return 0;
```

The empty statement is an extra semicolon and does nothing.

```
while (true)
    ;
```

Python

The **break** statement exits a loop.

```
while True:
    break
```

The **return** statement exits a function or method. If no expression is specified, the value **None** is returned.

```
return 0
```

The **pass** statement does nothing.

```
while True:
    pass
```

Terminal Output

Java

The method **println**, when run with the class variable **System.out**, converts data to text, displays it, and moves the cursor to the next line:

```
System.out.println("Hello world!");
System.out.println(34);
```

To prevent the output of a newline, use the method **print**:

```
System.out.print("Hello world!");
```

Python

The **print** statement automatically converts data to text, displays it, and moves the cursor to the next line:

```
print("Hello world!")
print(34)
```

To prevent the output of a newline, use the optional **end** keyword argument with the empty string:

```
print("Hello world!", end = "")
```

Terminal Input

Java

EXAMPLE: GettingInput.java

The **`Scanner`** class is used for the input of text and numeric data from the keyboard. The programmer instantiates a **`Scanner`** and uses the appropriate methods for each type of data being input. Create a **`Scanner`** object:

```
Scanner keyboard = new Scanner(System.in);
```

Input a line of text as a string:

```
String s = keyboard.nextLine("Enter your name: ");
```

Input an integer or double:

```
int i = keyboard.nextInt("Enter your age: ");
double d = keyboard.nextDouble("Enter your wage: ");
```

Caution: using the same scanner to input strings *after* numbers can result in logic errors. Thus, it is best to use separate scanner objects for numbers and text.

Python

The **`input`** function displays its argument as a prompt and waits for input. When the user presses the Enter or Return key, the function returns a string representing the input text. The programmer then either leaves the string alone or converts it to the type of data that it represents (such as an integer).

Input a line of text as a string:

```
name = input("Enter your name: ")
```

Input an integer:

```
age = int(input("Enter your age: "))
```

while loops

Java

EXAMPLE: WhileTest.java

```
while (<Boolean expression>){
    <statement>
    …
    <statement>
}
```

The statements in the loop body are marked by curly braces {}. **NOTE:** When there is just one statement in the loop body, the braces can be omitted.

The Boolean expression is enclosed in parentheses.

Python

```
while <Boolean expression>:
    <statement>
    …
    <statement>
```

The statements in the loop body are marked by indentation.

for loops

Java

Two types of loop to visit each element in an iterable object. The first type:

```
for (<type> <variable> :
<iterable>){
    <statement>
    …
    <statement>
}
```

Example:

```
for (String s : aListOfStrings){
    System.out.println(s);
}
```

The variable picks up the value of each element in the iterable object. Variable scope is in the loop body.

EXAMPLE: ForExample.java

Second type:

```
for (<initializer>; <continuation>;
<update>){
    <statement>
    …
    <statement>
}
```

Example:

```
for (int i = 1; i <= 10; i++){
    System.out.println(i);
}
```

Note: has the same behaviour as this **while** loop:

```
int i = 1;
while (i <= 10){
    System.out.println(i);
    i++;
}
```

Python

There is just one type of **for** loop, which visits each element in an iterable object, such as a string or list.

```
for <variable> in <iterable>:
    <statement>
    …
    <statement>
```

Example:

```
for s in aListOfStrings:
  print(s)
```

The variable picks up the value of each element in the iterable object and is visible in the loop body.

The statements in the loop body are marked by indentation.

Simple count-controlled loops that iterate through a range of integers have the form:

```
for <variable> in range(<lower
bound>, <upper bound>, <step
value>):
    <statement>
    …
    <statement>
```

Required Parameters
Optional Parameters

if statement

EXAMPLE: IfExample.java

Java

```
if (<Boolean expression>){
    <statement>
    …
    <statement>
}else if (<Boolean expression>){
    <statement>
    …
    <statement>
…
}else{
    <statement>
    …
    <statement>
}
```

The statements are marked by curly braces (**{}**). When there is just one statement, the braces may be omitted. Each Boolean expression is enclosed in parentheses.

Python

```
if <Boolean expression>:
    <statement>
    …
    <statement>
elif <Boolean expression>:
    <statement>
    …
    <statement>
…
else:
    <statement>
    …
    <statement>
```

The statements are marked by indentation.

switch statement

EXAMPLE: SwitchExample.java

Java

Unlike if-then and if-then-else statements, the **switch** statement can have a number of possible execution paths. A switch works with the **byte**, **short**, **char**, and **int** primitive types.

```
switch(condition){
  case 1:
    System.out.println("is 1");
    break;
  case 2:
    System.out.println("is 2");
    break;
}
```

Python

Python does not have a **switch** statement. But the same logic can be accomplished:

```
def f(x):
    return {
        1 : 1,
        2 : 2,
    }[x]
print f(condition);
```

Example

Lets Test What You've Learned ☺

Your task is to construct an application that will determine the average of a list of marks entered by the user.

You've been provided the following initial psuedocode:

```
Loop:
    Input next mark
    Update total
End loop
Calculate average = total / number of marks
Output average
```

** TASK REQUIREMENTS ARE ON THE NEXT SLIDE **

Requirements

You are required to construct three different versions of the given application as follows:

1. calculates an average (using `for`)
2. calculates an average (using `while`)
3. calculates an average (using `do...while`)

NOTE: pseudocode specific to each of these applications/conditional statements is provided on the next three slides.

Calculating an average (using *for*)

Psuedocode:

```
Input number of marks
For i = 0 to number of marks - 1
    Input mark
    Update total
Calculate average = total / number of marks
Output average
```

Calculating an average (using *while*)

Psuedocode:

```
Input mark
While last mark input != -1
    Update total
    Input mark
Calculate average = total / number of marks
Output average
```

Pseudocode (using do...while)

Calculating an average (using ***do...while***)

Psuedocode:

```
Do
    Input mark
    Update total
    Calculate average = total / number of
marks
Output average
While user has more data
Output final average
```

Java `do...while` format:

```
do {
    <STATEMENTS>
} while ( <CONDITION> );
```

Formatted output (using `printf`)

The easiest way to format output (e.g. to limit decimal places) in Java is to use the `printf` method. This method takes an arbitrary number of arguments, with the first specifying the format to be used. For example:

```
System.out.printf("%.3f%n", 1/3f);
```

will output the 2nd argument (i.e. `1/3f`) as a floating point number rounded to 3 decimal places (all shown), followed by a new line.

General Form

```
%[argumentIndex$][flags][width][.precision]conversion
```

- The optional `argumentIndex` is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by `1$`, the second by `2$`, etc.
- The optional `flags` is a set of characters that modify the output format. The set of valid flags depends on the conversion.
- The optional `width` is a non-negative decimal integer indicating the minimum number of characters to be written to the output.
- The optional `precision` is a non-negative decimal integer usually used to restrict the number of characters. The specific behavior depends on the conversion.
- The **`required`** conversion is a character indicating how the argument should be formatted. The set of valid conversions for a given argument depends on the argument's data type. The most common are `%d` – decimal integer, `%f` – decimal floating point, `%e` – decimal floating point in scientific notation, `%o` – octal integer and `%x` – hexadecimal integer. `%n` denotes a newline.

Formatted output

EXAMPLE: PrintfExample.java

Some Examples:

```
1   System.out.printf("%.2f", Math.PI);

2   System.out.printf("%.4f", Math.PI);

3   System.out.printf("%.4e", 1234.5678);

4   System.out.printf("%4d", 12);

5   System.out.printf("%2$d %1$d", 48, 47);

6   System.out.printf("%2$x %1$o", 48, 47);
```

Java 8 API SPecification

https://docs.oracle.com/javase/8/docs/api/

Next week...

We will explore the notion of an

Object

and

Object Oriented Programming

Feedback

Are there things you like?, Things you don't like?, Suggestions of any kind? I really would like to hear from you.

Let me know anonymously at :
https://www.suggestionox.com/r/p2DsyU