CPS122 Lecture: From Python to Java

last revised January 7, 2013

Objectives:

1. To introduce the notion of a compiled language
2. To introduce the notions of data type and a statically typed language
3. To introduce a few of the key syntactic differences between Python and Java
4. To explain the fundamental distinction between primitive types and reference types, and to introduce the Java primitive types

Materials:

1. Projectable showing difference between static and dynamic typing
2. Python interpreter and Dr. Java for demos
3. Projectable showing different syntax for control structures
4. Table of operator precedence from Bruce book

I. Introduction

A. As you may recall, in our course introduction we noted that this class will use Java as a programming language, rather than Python, for three reasons, all related to the size of the problem we are seeing to solve. Do you recall what those reasons where?

ASK

1. Java is designed for an approach to software development known as object orientation. (Python has support for object-orientation, but not as fully.)

2. Java is a compiled language.

3. Java is a statically-typed language.

B. We talked a bit about Object-Orientation in the first lecture, and will say a lot more about it throughout the course! In this lecture, we will explain the latter two, and also say a bit more about Java syntax.

II. **Compilation versus Interpretation**

A. Computer hardware (a CPU chip) is capable of interpreting programs written in binary machine language - 1's and 0's,

1. However, writing programs in machine language has lots of disadvantages:

   a) It is tedious.

   b) It is error-prone.

   c) Correcting such programs when errors are found is extremely difficult.

   d) Modifying such programs is extremely difficult.

   e) Perhaps most important of all, the machine language of each family of CPU chips is unique to that family. Thus, a program written for one family of chips (e.g. the Intel 80x86 used on many current systems) is totally useless for some other family.

2. For this reason, very early in the history of computer science, various *higher-level* programming languages were developed. Though still not as natural as English, they are a whole lot simpler to use than binary machine language. Python is one such language; Java is another; and there are hundreds more.

   In addition to being much easier to use, higher level languages have the advantage of being designed to be cross-platform - i.e. the same program should be capable of being used on multiple families of CPU.

B. With the use of higher-level languages, a new problem arises. The CPU can only interpret its binary machine language - so how does it execute a program written in a higher-level language?

Two approaches to solving this problem have been widely used.

1. Software interpretation. A special program called an i*nterpreter* is written for a particular chip, in the binary machine language of that chip. This program reads the higher-level language program one step at a time and interprets and carries out that step.

2. Compilation. A special program, called a *compiler*, is used to *translate* the higher-level language program into the binary machine language of the target system

3. The key difference between the approaches is how often a given higher-level language statement is "understood". With an interpreter, it is "understood" each time it is executed. With a compiler, it is "understood" once, and then translated, as a separate step before it is executed.

   Example:

   Say to a student who knows German: "Heben sie die hand, bitte" ("Raise your hand, please")

   Ask the same student to translate into English, hand the translation to another student to interpret.

4. Often, one approach or the other is the preferred approach for a given higher-level language. For example, Python is designed to be implemented using software interpretation, and Java - like most higher-level languages - is designed to be implemented using compilation. [There are a few languages that can actually be implemented either way.]

C. A key visible distinction between the two implementation models is that with a compiled implementation there is a distinct compilation step that occurs between writing a program and running it, whereas with an interpreted implementation it is possible to enter a single statement and have it executed immediately.

1. Of course, the interpreted approach is more convenient to use.

2. However, though the compiled approach requires extra effort, it has a number of advantages.

   a) A compiled program can perform the same task much faster than an interpreted program, because the step of "understanding" the source program is done just once, at compile time, and because it is possible to perform some "optimizations" during translation that make the program run more quickly.

      (1) This is not much of an issue in many cases, given the speed of modern computers. But it can be an issue with compute-intensive applications, such as certain kinds of media operations (e.g. movie, sound or picture editing) or scientific "number-crunching".

      (2) Some interpreted implementations mitigate the problem by storing the program internally in a partially-translated form to avoid doing all the work of "understanding" the source each time a statement is executed, and may even save the translated form to facilitate subsequent loading. (For example, Python does this with its .pyc files). But this still doesn't yield the kind of efficiency possible with a compiled implementation.

   b) A more significant issue is software reliability. A compiler can be designed to catch many kinds of errors (but by no means all)

when a program is compiled, rather than having them show up when the program is running.

(1) We will see lots of examples of this as we learn Java.

(2) There will be times when you will be tempted to think of detection of errors by the compiler as a nuisance, but remember this is much less of a nuisance than having something go wrong when the program is running!

c) Compiled implementations also have significant security advantages.

D. Having said all this, for demonstration purposes we will make use from time to time of a program known as Dr. Java. This has an interaction window which allows you to enter a Java statement and have it executed immediately - just like in an interpreted language. But what is really happening is that Dr. Java is compiling a mini-program containing the statement "behind the scenes".

E. The IDE we will use in lab - NetBeans - performs the compilation step automatically when you try to run a program that has been modified since it was last compiled - so you will not necessarily be conscious of compilation being done.

III. **Static versus Dynamic Typing**

A. Before we discuss the issue of static versus dynamic typing, we need to first introduce the notion of a data type. Both Python and Java (and indeed almost all programming languages) have this concept.

1. A data type defines a set of possible values and a set of possible operations. For example, most programming languages (including Python and Java) have the notion of the data type integer (called int in both languages).

a) The possible values of an int are the mathematical integers (actually a finite subset in Java).

b) The possible operations on an int are the standard arithmetic operations such as +, -, *, / etc.

2. Most programming languages (including Python and Java) are strongly-typed languages - which basically means that one cannot apply an operation to a data item for which it is not appropriate.

   Example: DEMO in Python

   4 - 3 is OK
   "four" - "three" is not - the type of the values is string, not integer

3. In fact, sometimes the same symbol may specify a different operation depending on the types of the items it is applied to

   Example: DEMO in Python

   4+3 - here "+" is arithmetic addition
   "four" + "three" - here "+" is string concatenation

B. To illustrate the difference between static and dynamic typing, consider the following task: calculate the two roots of a quadratic equation using the quadratic formula you learned in high school:

   If an equation is of the form $Ax^2 + Bx + C = 0$, then its roots are

   $$x = -B +/- \text{sqrt}(B^2 - 4AC)$$

   PROJECT TypeDeclarations projectable

   (Note: these are just excerpts from complete programs, written so as to use a similar style insofar as possible. For simplicity we assume we will only use them for cases having two real roots.).

   What differences do you see?

   ASK - Be sure they see the explicit declarations of the variables

1. In the Java program, the variables a, b, c, x1, and x2 are explicitly declared to be of a data type known as double (double-precision real numbers) before they are actually used.

2. In the Python program, these variables are first introduced when they are used.

3. The difference arises because Java is statically typed, which means three things:

   a) A variable must be explicitly declared to be of some type before it can be used.

   b) A variable must only be used in a way that is appropriate for its type.

   c) The type of a variable cannot be changed once it is declared to be of a certain type.

      DEMO with Dr. Java:

      int x
      String x

4. In contrast, Python is dynamically typed.

   a) Variables are not explicitly declared. Rather the type of a variable is based on the value it currently holds.

   b) Though a variable can only be used in a way that is appropriate for its current type, the type of a variable can be changed by giving it a different value.

      DEMO

      a = "four"
      a - 1
      a = 4
      a - 1

5. Although declaring a variable and giving it a value are conceptually two different things, Java does allow these two operations to be combined into a single statement.

   Example:
   ```
   int x;
   x = 3;
   ```

   is equivalent to `int x = 3;`

C. At first glance, it may appear that static typing is a nuisance. However, it has several very significant advantages

1. It provides some protection against typographical errors.

   Example: suppose you used a variable named something in a program, and at one point misspelled it as sonething in an assignment statement

   sonething = 4

   The Python interpreter would allow this, by treating it as the definition of a new variable - which could result a hard-to-find error when the variable something turns out to have the wrong value elsewhere in the program.

   In contrast, the Java compiler would flag this as an unknown symbol.

2. It provides protection against inadvertent misuse of a variable. The compiler can (and does) check each use of a variable to be sure that the use is consistent with its declared type.

   Example: if the variable a were declared as a string, the Java compiler would flag a - 1 as an error

   DEMO with Dr. Java

   String a
   a - 1

3. It facilitates production of more efficient code. With a dynamically typed language, each time a variable is used its current type must be checked to determine whether the operation is appropriate and, if so, what it means. With a statically typed language, all this is checked at compile time.

D. Java has 8 builtin types known as primitive types that have a special status in the language. and are represented in an efficient way in memory. (If you are familiar with the C programming language, these are similar to the primitive types of C.)

1. Four different types of integers - distinguished by the number of bits of memory used to store each, and by the resultant range of possible values.

   a) byte - one byte (8 bits) - range of values -128 .. 127

   b) short - two bytes (16 bits) - range of values -32768 .. 32767

   c) int - four bytes (32 bits) - range of values -2,147,483,648 .. 2,147,483,647

   d) long - eight bytes (64 bits) - -9,223,372,036,854,775,808 .. -9,223,372,036,854,775,807

   e) Unless there is a good reason for doing otherwise, int is typically the one that is used

2. Two different types of real numbers - distinguished by the number of bits of memory used to store each, and by the resultant precision and range of possible values

   a) float: 4 bytes (32 bits). Precision equivalent to about 7 decimal digits, with range of values on the order of $\pm 10^{38}$

b) double: 8 bytes (64 bits): Precision equivalent to almost 16 decimal digits, with range of values on the order of $\pm 10^{308}$

c) Unless there is a good reason for doing otherwise, double is typically the one that is used

d) In contrast, in Python there is just one real type - called float, but using the same amount of memory as a Java double.

3. A boolean type whose possible values are true and false. (Note: in Java the boolean constants are true and false - all lower case, whereas in Python they are True and False. Also, in Python the booleans are equivalent to the integers 1 and 0, which is not the case in Java - boolean is not an integer type).

4. A character type (char) that holds the code for a single character. (Not present in Python)

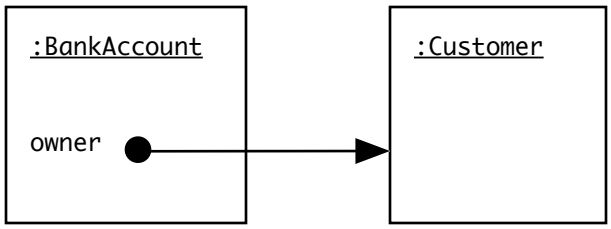
5. For reasons we will see shortly, primitive types are also called <u>value types.</u>

E. Java also makes it possible to define new data types by defining classes (which we will talk about in the next lecture), arrays, interfaces (both of which we will not discuss until much later in the course.) The code for defining such a new type is, itself, written in Java. Collectively, these types are called <u>reference types</u>.

Illustration of the meaning of "reference"

BlueJ Bank Example demo - create a Customer and a BankAccount, then inspect BankAccount and double click owner reference to bring up Customer object.

The situation that holds here looks something like this:



1. A large number of reference types are defined as part of the standard Java libraries. We will make use of a number of these throughout the course.

2. One such type is that used for character strings (String). It is not a primitive type because Strings can vary widely in the amount of storage required. It does, however, have a "privileged" status when compared to other reference types.

3. The process of creating a Java program centers around creating new classes, each of which, in effect defines a new data type.

4. An object of a reference type must always be explicitly created by using the reserved word `new` (We will see examples of this later)

5. Actually, Python supports something similar, but most Python programs do not make use of this facility.

IV. Key Syntactic Differences between Java and Python

A. As you probably noted in the excerpts I projected earlier, there are quite a number of syntactic differences between Java and Python. Most of these will come to the surface as we encounter them, but there are a few worth mentioning now.

B. Delimiting individual statements

1. In Python, a statement runs until the end of a line, though it is possible to extend statements over multiple lines by ending each

line except the last with a '\' . It is not possible to have two statements appear on the same line

2. In Java, statements are terminated by a semicolon. A statement can be spread over multiple lines, and multiple statements can appear on one line.

   DEMO with Dr. Java

   ```
   int
   a
   =
   3;
   int b = 4; int c = a + b; System.out.println(c);
   ```

C. Comments

1. In Python, a comment is started by a "#" and extends to the end of the line

2. In Java, there are two ways of marking comments

   a) If a "//" occurs, everything following it - to the end of the line - is a comment. (So "//" in Java behaves just like "#" in Python)

   b) If a "/*" occurs, everything following it - to the next occurrence of "*/" is considered a comment. This may be a part of a line, or may extend over several lines

      DEMO with Dr. Java

      ```
      /* Short comment */ int d = 3; /* Long
      comment
      extending
      over
      multiple
      lines */
      System.out.println(d);
      ```

D. Strings

1. In Python, character strings can be written using either of the two types of quotation mark ("" and '), as long as the string is terminated the same way it began.

2. In Java, character strings are always written using double quote marks ("").

3. In Java, single quote marks are used for character literals (values of type `char`).

E. Boolean operators

1. In Python, compound boolean expressions can be constructed using the boolean operators `and`, `or`, and `not`.

2. As we've already noted, Java uses the symbols `&&`, `||` and `!` for this purpose, and also has the boolean operator `^` .

F. Comparison operators

1. In Python, the comparison operator (`>, <, >=, <=`) can be applied to objects of any type, and have the expected result when used with numbers and strings.

2. In Java, these operators can only be applied to primitive types.

3. In Python, the equality comparison operators (== and !=) always test their operands for the same value.

4. In Java, when these operators are applied to primitive types, they test for same value; but when applied to reference types, they test for same object.

G. Names

1. In Python, there is no single, well-established convention for how names are assigned to user-defined entities.

2. In Java, there are well-established conventions for naming. Though these are no enforced by the compiler, they do constitute well-accepted good practice.

3. The handout introduces these - but we will speak of them in detail later.

H. Conditional and Looping Statements

1. Both Python have the notion of a conditional statement introduced by the word "if", and looping statements introduced by "while" or "for", but the syntax is quite different.

   PROJECT comparison of syntax for control structures

   a) Python: keyword expression :

   b) Java: keyword (expression)

2. Delimiting the scope of conditional/looping statements

   a) In Python, the scope of a conditional/looping statement is delimited by indentation

      Example:

      ```
      if x == 0:
           print x is 0
      ```

   b) In Java, the scope of a conditional/looping statement is delimited by braces. The Java equivalent would be

```
if (x == 0)
{
     System.out.println("x is 0");
}
```

(1) Actually, if the scope of a conditional/looping statement is just a single statement, the braces are not needed

Example: the above could be written

```
if (x == 0)
        System.out.println("x is 0");
```

(2) It is considered good practice to indent statements in a manner similar to the manner used in Python for readability, but the compiler pays attention to the braces - not indentation - so the following is equivalent, though stylistically poor:

```
if (x == 0)
System.out.println("x is 0");
```

(Indentation conventions for reasons of style preceded Python; Python just adopted them and gave them syntactic significance)

I. Function Definitions

1. Python

a) The first line of a Python function definition begins with `def` and ends with a colon.

b) The body is delimited by indentation - just the way the scope of a control structure is.

c) Formal parameters are listed in parentheses after the function name - or an empty pair of parentheses is used if there are no parameters.

2. Java

   a) Because Java is statically typed, a function definition always begins with a type specifier indicating what kind of value the function - or `void` if the function does not return any value.

   b) The body is delimited by braces - similar to the way the scope of a control structure is delimited - but in this case the braces are always required, even if the body is just one line. Indentation, though not syntactically meaningful, is good style.

   c) As in Python, formal parameters are listed in parentheses after the function name - or an empty pair of parentheses is used if there are no parameters. However, again because Java is statically typed, each parameter name is preceded by a type specifier.

J. Function calls in Python and Java have similar syntax, except that a function call in Java always specifies the object or class to which it is applied. (We'll see examples of this later.)

   (Actually, Python also uses something similar when one is using its object-oriented features.)

V. More About Primitive Types in Java

A. Earlier, we noted that Java data types fall into two categories: primitive (value) types and reference types. There are some important differences between primitive and reference types.

1. The names of primitive types are all lower-case. By convention, the names of reference types begin with an upper-case letter.

2. Primitive types are also known as value types, because an entity of a primitive type stores a value, rather than referring to a value stored elsewhere. Therefore, in Java (though not all OO languages), primitive types are not objects.

a) The operator `new` is <u>not</u> used to create them. (Reference types are always created this way.)

b) Methods cannot be applied to them. You will <u>never</u> see something like:

```
int i;
i.something();
```

3. Java provides standard ways of writing literals of the primitive types. Examples (for the types we will use):

   a) `boolean - false, true`

   b) `int - 1`

   c) `double - 1.0` (the decimal point distinguishes from `int`), or scientific notation can be used - e.g. `6.02E23`

   d) `char - 'A'`

   e) For convenience, and because they are widely used, Java also provides a mechanism for writing literals of one (but only one) of the reference types - String

   ```
   "This is a String"
   ```

4. Java provides various operators that can be applied only to entities of primitive type: Examples:

   a) For `boolean` - boolean operators such as `&&, ||, !, ^`

   (Give truth table for each)

   b) For the numeric types (`int`, `double`) - the familiar arithmetic operators `+, -, *, /, %` [Similar to Python]

   (1) As in Python, the operator `/` has a somewhat different meaning depending on its operand types.

(a) When used with two `ints`, it means integer division. Any remainder is discarded.

DEMONSTRATE with Dr. Java: `2/5, 5/2`

(b) When used with two `doubles`, or a `double` and an `int`, it means real division.

DEMONSTRATE with Dr. Java: `2.0/5.0, 5.0/2.0`

(2) The operator % stands for "remainder upon division by" and is primarily used with integer types, though it can be used with reals as well

DEMONSTRATE with Dr. Java: `5 % 2, 5.0 % 2.0`

c) For the numeric types plus `char` - relational operators such as `>, <, >=, <=`

5. The equality comparison operators (== and `!=`) can be applied to both kinds of types, but they mean something different for reference types than they do for value types.

a) When applied to two entities of value type, they ask “do these two represent the same value?”

b) When applied to two entities of reference type, they ask “do these two refer to the same object?”

Of course, the answer to the first question will always be yes when the answer to the second question is yes. But the reverse is not true. It is possible to have two different objects that - in some sense - represent the same value. [Example: two different bank accounts with the same balance].

DEMONSTRATE with Dr. Java `1 == 1, "hello" == "hello"`

c) Rules of precedence are used to resolved ambiguities in expressions that contain several operators - e.g. `1 + 2 * 3` is interpreted as `1 + (2 * 3)`, rather than as `(1 + 2) * 3`.

(1)Unary operators take precedence over binary operators

(2)*Multiplicative* binary operators (*, /, %) take precedence over other binary operators.

(3)The *additive* binary operators (+, -) are the lowest precedence operators of the ones we have looked at so far.

(4)Ties between operators at the same level are broken left to right for the operators we have considered so far. (There are some that break ties the other way!)

Example: Order of evaluation of operators in

```
- a + b % c / d - e * - f - g
```

(Work out with class)

```
- a + b % c / d - e * - f - g

1   6   3   4   7   5 2   8
```

equivalent to

```
( ( (- a) + ((b % c) / d) ) - (e * (-
f)) ) - g
```

d) As noted in the above example, parentheses can be used to explicitly specify precedence - either because

(1) An order other than the normal one is needed

Example: `(a + b) * (c + d)`

or

(2) There is a desire to make the order clearer to the reader

(3) In the case of boolean expressions, a more complete set of precedence rules takes into account the comparison operators as well.

PROJECT: Table of operator precedence

B. Just as variables have a type, so expressions have a type.

1. The type of an expression is determined by the types of its operands and operators. The following rules deal with the data types we will use - there are some additional rules that apply to the numeric types we will not discuss.

a) For the arithmetic operators, the type of the result of operation is determined by the types of its operands according to a rule that is sometimes called the *rule of numeric contagion*.

(1)For unary operators, the type of the result is the same as the type of the operand.

(2)For binary operators:

(a)If either operand is of type double, the other is converted to double (if necessary) and the result is of type double.

(b)Otherwise, the result is of type int.

b) For the relational and equality comparison operators

(1) If the two expressions being compared are numbers, they are converted according to the rules of numeric contagion and then compared.

(2) A `char` can only be compared to a `char`.

(3) `booleans` and reference types can only be compared for equality

The result of any comparison, of course, is always `boolean`.

c) Note that the rules are applied step by step during the evaluation of the expression.

Example: What is the value of the following expression

(4 / 5) * 2.0

ASK

Answer: 0.0 ! (DEMO with Dr. Java)

The division 4 / 5 is done in type int, because both operands are ints. The quotient is 0 and the remainder of 4 is discarded. The int 0 is then converted to double 0.0 and multiplied to still yield 0!

2. A related requirement is that the type of an expression that is assigned to a variable be *assignment compatible* with the variable.

(1) A variable of type `double` may be assigned the result of any numeric computation. If the result is an integer, it is promoted.

(2) A variable of any other type may only be assigned an expression whose value is of the same type.

3. There are times when it is necessary to perform an operation where the needed type conversion is not automatically performed. In this case, one must use an explicit *type cast*.

Example: The following statement is erroneous, because the expression is of type `double` and the variable is of type `int`:

```
double d;
int i = 3 * d;
```

To make this work correctly, we need to use a type cast, as follows:

```
int i = (int) (3 * d);
```

Note: Java requires an explicit cast, because a computation like this might - in principle - result in the loss of information:

The explicit use of a cast means “I know this could result in the loss of information, but in this case I know this won’t happen, or I’m prepared to accept it.”