COMP6700/2140 Syntax Oddities

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

March 2017

Topics

1. Post- and pre-fix operators
2. Ternary operator
3. Varargs type declaration and varargs methods
4. Formatting string syntax

Cute oddities

Sometimes programming languages have syntax features which may look strange, or even ugly. This may be true, but also may be just a first impression, when later they start grow on you, and eventually you find them appealing.

Compare this with modern art ☺.

Java has elements of syntax which may be considered odd, but they are just a few, and not extreme like in other languages (*eg*, *Perl*, which it seems is built from such oddities from top to toe).

It is good when a language has oddities (personal opinion). They help to keep some degree of emotional engagement with it. Java could have more of them.

Pre- and post-fix value updates

For primitive numerical types, the operators

`+`, `-`, `*`, `/` and `%`

can be combined with the assignment operator `=` when used to increment (decrement, multiply, divide,remainder) an old value :

```
value = value + delta; // is same as
value += delta;  // same for other -, *, /, %
```

In a particular case of incrementing (decrementing) by 1, commonly used in determinate `for`-loops, a more cryptic expression is used :

```
i++; // is identical to "i = i + 1" except that i is evaluated only once!
i--; // similarly for "i = i - 1"
```

The expression `i++` is post-incrementing (first evaluates then increments), the expression `++i` is pre-incrementing (first increments then evaluates).

Quirks like `++i++` are syntax errors.

Repeat the exercise: what is the value the expression `(i++ + i++ + --i)` evaluates to if the value of `i` before was assigned to 3?

String concatenation and operator overloading

The + operator also works for strings — it's called *concatenation*:

```
String s1 = "French Connection "; String s2 = "UK"; s1 += s2; // s1 evaluates to …
```

It is convenient ("easy") to use it when you build a string out of multiple strings (often using a loop). **Avoid this!** since every time you make a concatenation, a new string is created (expensive operation), which lives for brief time only (wasteful). A better option is to use the class *StringBuilder*.

No other arithmetic operators are applicable to string references, or in expressions which involve, for example, a string and an `int`. The multiplication of a string and an integers:

```
String longer = "myPride" * 3; // No can do in Java!
```

[Aside note: string-by-integer multiplication is legal in Python, it is even commutable which make perfect sense!]

Except for string concatenations, the use of standard (arithmetic, logical, bitwise) operators is illegal for any reference types (the wrapper types are OK thanks to automatic auto-boxing, **J5**). The language does not allow you to define such operations when you define a new type. This feature is called *operator overloading* (it is available in C++). Java does not allow operator overloading. This decision was made in the very beginning to keep the language simple(r).

The ternary operator ? :

When a "binary" `if-else` statement is used to assign a value to the variable:

```
if (wonAOFinal)
    prize = 1000000;
else
    prize = 500000;
```

it can be rewritten in a more succinct yet expressive form :

```
prize = (wonAOFinal ? 1000000 : 500000);
```

In a general expression with the ternary `?:` operator:

```
prize = booleanExpr ? value1 : value2;
```

if `value1` and `value2` have different types, the resulting type depends on assignable (`prize`). Automatic in/out-boxing is performed when necessary. In the case of incompatible types for `value1` and `value2`, the first common parent is returned, up to the *Object* type. Error occurs if any type is `void`.

A "reduced" version, called *Elvis Operator*, helps avoid testing for nullity in assignments:

```
Value v = val1 ?: val2; // if val1 is null, val2 will be used to assign v
```

Varags

To write a method with an arbitrary number of parameters requires usage of an array. The new Java 1.5 feature *variable arguments (varargs)*, which allows an alternative way of dealing with such situation. The *last* parameter in a method (or constructor) can be declared as a sequence — finite but otherwise undetermined set of parameters — of a given type. The syntax :

```
public static R foo(T1 t1, T2 t2, T... t);
```

(only one varargs is allowed which must be the last declared parameter) which is treated by the compiler as

```
public static R foo(T1 t1, T2 t2, T[] t);
```

the access and scope modifiers are arbitrary; T1, T2, T and R are parameters and return type. When the method `foo()` is called, the third parameter can be replaced by *any* number of actual parameters of the same type T, including no parameters, *ie*, passing no arguments is a legal (while the "old-fashioned" array form assumes that the last argument is always included, and it's an array). `main` can also be declared with varargs.

```
foo(a1, a2);
foo(a1, a2, v1);
foo(a1, a2, v1, v2,...,vn);
public static void main(String... args) {...};
```

printf(): "March of Progress" (after Cay Hortsmann)

```
printf(String format, Object... args) // the syntax of printf() formatting function:
System.out.printf("%s: %d, %s%n", name, idnum, address); // the usage
```

1980: C

```
printf("%10.2f", x);
```

1988: C++

```
cout << setw(10) << setprecision(2) << showpoint << x;
```

1996: Java

```
java.text.NumberFormat formatter = java.text.NumberFormat.getNumberInstance();
formatter.setMinimumFractionDigits(2);
formatter.setMaximumFractionDigits(2);
String s = formatter.format(x);
for (int i = s.length(); i < 10; i++)
    System.out.print(' ');
System.out.print(s);
```

2004: Java

```
System.out.printf("%10.2f", x);
```

2008 and beyond: Java.next

```
printf("%10.2f", x) // Scala and Groovy
println(f"$x%10.2f") // Scala 2.10
```

String Formatting

Instead of multiple concatenations used to create a long string with various *values of different type* inserted, prefer string formatting. The `static` method

```
String.format(java.lang.String, java.lang.Object...)
```

returns a string given by the first argument, a so called *formatting string*, in which *embedded format specifiers* (those thingies with %) are replaced by appropriately processed values given by the 2nd, 3d,... arguments. Not only primitive values formatting can be achieved in very flexible way:

```
String s1 = String.format("Decimal: %d and hex: %x", 2716);
String s2 = String.format("  is equal %.10f", Math.PI);
```

A complex object like date (an instance of *Calendar* or *Date*), can be formatted to meet the locale specifications:

```
Calendar c = ...;
String s = String.format("Duke's Birthday: %1$tm %1$te,%1$tY", c);
```

Where to look for this topic in the textbook?

- Hortsmann's *Core Java for the Impatient*, Ch. 1.4.3, 1.6, 1.9.3
- Format string syntax