JOURNAL OF OBJECT TECHNOLOGY
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Keyword- and Default-Parameters in JAVA

Joseph (Yossi) Gil[a] Keren Lenz[a]

a. Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel

Abstract Overloading is a highly controversial programming language mechanism by which different methods of the same class are allowed to bear the same name. Despite the criticism, JAVA programmers make extensive use of this mechanism—not just because it is available, but also because the language does not provide an alternative for defining multiple constructors, and because it is useful for expressing similarity of services provided by a class.

In a previous paper we argued that more than 60% of the overloading cases are "justifiable" and that in 35% of the cases overloading is used for emulating a default arguments mechanism. Based on these results, this paper argues that most "justifiable" uses of overloading are better done with a combination of *keyword parameters* and *default parameters* parameter definition mechanisms, and describes our extension of the JAVA compiler which adds these two features to the language.

Keywords ...

1 Introduction

Unlike C++ [Str97], C# [AA10] and ADA [TD97], methods in JAVA [AG96] cannot declare default values for their formal parameters. *Overloading* is available as a substitute: A default argument can be emulated by introducing a new method bearing the same name which invokes the original with an appropriate value for this argument. For example, method `getInteger(String)` in the JAVA's standard `Integer` class, supplies a default argument to the more general `getInteger(String, Integer)` method:

```
public static Integer getInteger(String nm) {
  return getInteger(nm, null);
}
```

For the class's author, this emulation of default parameters means a blown up interface with extra code to document and maintain. In this example, the three-lined function `getInteger(String)` incurs a 28-lines documentation overhead toll. For the class's client, this practice requires familiarity with different versions of essentially the same method, and

understanding of the subtle semantics of overloading, and its not so trivial interaction with overriding [BS07], in order to make sure that the intended method is indeed invoked.

Overloading is a highly controversial language construct [Mey01]. When used correctly it allows to capture similarity between different methods and to emphasize the fact that several different methods represent the same conceptual operation. However, it has the potential of being abused, by assigning the same name to conceptually different methods. As a result, several style guides [1] all but completely forbid the use of overloading.

In this paper we discuss two language features: *keyword arguments* and *default arguments*, and argue that their use in tandem reduces the use of overloading and enhances readability and maintainability of the code. Attaching names to parameters in the invocation command highlights their meaning, and reduces the need to examine lengthy documentation. The combination of keyword parameters and default values allows one to specify the actual parameters in any order while omitting any subset of the parameters that have default values.

This combination is used in several programming languages including PYTHON [Lut96] and LISP [Gra95]. C# designers recognized the advantages of the combination, and a mechanism that supported named and optional parameters was added to version 4.0 of the language[2].

After arguing for the merits of this combination, we describe our JAVA language extension supporting it. The extended compiler is available for download at the following address: `http://ssdl-wiki.cs.technion.ac.il/wiki/index.php?title=Call_by_Name_Java_Extension`. With the extension, every subset of the formal parameters may be assigned default values. A default value is either a constant or an expression involving method calls, data members and other formal arguments, which is evaluated at the scope of the declared method. The dependency between the formal arguments may even be circular, under some constraints.

Our extension allows a programmer to supply actual arguments either **(i)** *positionally*, i.e., in the order of the formal parameters, **(ii)** in a *keyword based* fashion, where each argument is preceded by the name of the formal parameter, or, **(iii)** in a *mixed style* by which a prefix of the actual arguments is specified positionally, while the remainder is specified by name. We identify the difficulties of interaction of this mechanism with inheritance, and explain how to deal with these while preserving the compatibility principle of Meyer [Mey92] and Liskov [LW94].

Outline. The remainder of this paper is organized as follows. Sec. 2 motivates the use of keyword arguments with default values as a substitute for overloading. Sec. 3 describes how the distinction between operands and options is supported using keyword and default arguments. Sec. 4 presents a JAVA language extension that supports keyword and default arguments and describes its implementation. Sec. 5 surveys related work, highlighting the arguments against and for keyword arguments. Sec. 6 concludes.

2 Solving the hardship of overloading in JAVA

In this section we argue that one of the most common uses of function name overloading in JAVA is for the emulation of a default parameters mechanism. We then emphasize the advantages of keyword parameters and default values over the practice of using overloading.

In previous work [GL10] we studied the use of overloading in JAVA programs. In this research we used sampling techniques to estimate the prevalence of various overloading patterns in JAVA code. We found that overloading is extensively used in JAVA programs – 35% of all constructors and 14% of all methods are overloaded.

[1] http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

[2] http://msdn.microsoft.com/en-us/library/dd264739.aspx

In addition, we developed a taxonomy of overloading patterns and applied it to a large corpus of JAVA programs. We found that most uses of overloading are "justifiable", that is, overloading is typically not abused. Fig. 1 presents the results of the study with respect to distribution of overloading sets of methods (a) and constructors (b) by the categories of the taxonomy. The X axis in both graphs of the figure stretches a spectrum of overloading categories, ranging from ad-hoc patterns, in which overloading is coincidental, to systematic patterns, in which overloaded methods are semantically cohesive.

For brevity, we do not discuss the categories in details here, but just outline their description. We refer the reader to our previous paper [GL10] for a full explanation of each category. The INTRINSIC category represents overloading sets in which methods are semantically and sometimes syntactically related. The POTENTIAL category is similar in that it can be brought into the INTRINSIC category with minimal effort. PEER-CALLERS are overloading instances in which a method calls its peer, but it is not clear whether it can be rewritten in the INTRINSIC form. On the dividing line, we find PLACEHOLDERS, in which all methods in the set are not implemented. That is, methods in this category either have an empty body or do nothing but throw an exception or return a trivial value. The overloading kind can fall into any other category, depending on the implementation in the inheriting class or classes, therefore this category is "neutral". The ACCIDENTAL category, refers to cases in which no peer calls occurred, and no other relationship between peers could be identified.

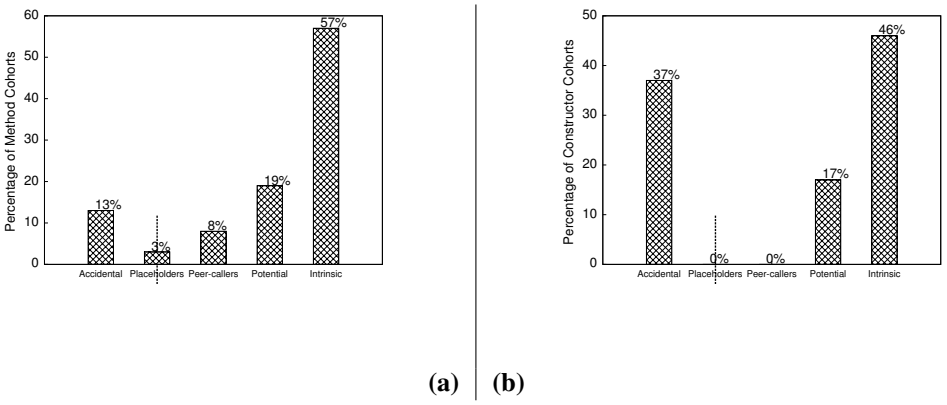

Fig. 1 – Spectrum of systematic overloading in method sets (a) and constructor sets (b)

As can be seen in the figure, there is a clear tendency towards more systematic use of overloading in method sets. This tendency is less strong in constructor sets, but still, 63% of the constructor sets fall into systematic categories.

We further broke down the INTRINSIC category into subcategories, and found that the most common use of overloading in JAVA is for the emulation of keyword and default arguments: 41% of the method sets and 43% of the constructor sets in the INTRINSIC category follow this pattern. We found that the addition of a keyword and default arguments mechanism to the language could eliminate overloading in about 35% of the cases.

Having stressed that a keyword parameters and default values mechanism could replace a substantial portion of the cases of overloading, we now discuss the benefits of such a mechanism. Consider for example JAVA class `Point` depicted in Fig. 2.

In the example, variables `x` and `y` are optional and have `0` as default values. To realize this, the constructor is defined twice and the second definition passes the default values to the first, which does the actual work. In JAVA, overloading is mandatory in the definition of multiple constructors, since programmers are not free to name constructors as they please— all constructors of a given class must bear its name. In a typical class, there are many constructors

```
class Point {
  private int x, y;

  public Point(int x, int y) { this.x = x; this.y = y; }
  public Point() { this(0,0); }
  //...
}
```

Fig. 2 – A JAVA class representing a two-dimensional point.

that forward their work to each other, and to super-classes as well. For example, in order to support the option that the `y` parameter is optional, another constructor has to be added. Such a constructor is likely to be implemented by invoking the first constructor of Fig. 2.

This is a cumbersome solution for the simple problem of realizing an optional default value. For each possible combination of optional arguments, an additional method must be provided. This situation is even more typical with constructors than with ordinary methods. To quote a JAVAWorld article [Smi98]

> *"With* JAVA*, the language design for constructors is quite elegant—so elegant, in fact, that it's tempting to provide a host of overloaded constructors. When the number of configuration parameters for a component is large, there can be a combinatorial explosion in constructors, ultimately leading to a malady known as constructor madness. . .*

Named parameters together with default values, offer a simple substitute. The host of methods (or constructors) doing nothing other than forwarding to other methods of the same name, but still carrying the price tag of extra documentation, interface bloat, and maintenance issues, can be replaced by a single method with appropriate defaults.

This is the case, for example, with those constructors of the JRE's standard implementation of class `String` in charge of creating a `String` object from a **`byte`** array.

```
public String(byte bytes[]){
  this(bytes, 0, bytes.length);
}
public String(byte bytes[], int offset, int length){
  /* ... */
}
public String(byte bytes[], String charsetName){
  this(bytes, 0, bytes.length, charsetName);
}
public String(byte bytes[], int offset, int length, String charsetName){
  /* ... */
}
public String(byte bytes[], Charset charset) {
  this(bytes, 0, bytes.length, charset);
}
public String(byte bytes[], int offset, int length, Charset charset){
  /* ... */
}
```

Fig. 3 – Overloading and forwarding as substitute to default arguments in the constructors of `String`.

Fig. 3 demonstrates the process of forwarding while supplying defaults among the six overloaded constructors: the first constructor in the figure, `String(byte[])`, gives default value to the `offset` and `length` parameters while calling the second constructor whose signature is `String(byte[], int, int)`; the same two arguments are given the same default values in the call of the third constructor, `String(byte[], String)`, to the fourth constructor, `String(byte[], int, int, String)`, and in the call of the penultimate constructor, `String(byte[], int, Charset)`, to `String(byte[], int, int, Charset)`, the last constructor.

A calling mechanism featuring parameter defaults is more than just syntactic sugar for method overloading; it can deal with the situation in which several arguments are of the same type—a situation which baffles JAVA's overloading mechanism. In class `Point` of Fig. 2 for example, we see a constructor in which *both* `x` and `y` default to 0, but it is impossible to declare constructors for the situations in which *either* `x` or `y` are missing, by adding both

```
Point(int x) { this(x,0); }
```

and

```
Point(int y) { this(0,y); }
```

to the set of constructors of class `Point`. These two constructors definitions are contradictory since plain JAVA uses parameter types (and these types only) for resolving overloading ambiguities. The situation is slightly better with methods, whose name can be changed to support a variety of argument combination. With the same `Point` example, an attempt to define a variety of `move` methods by writing e.g.,

```
move(int x) { /* ... */ }
move(int y) { /* ... */ }
move(int x, int y) { /* ... */ }
```

will be rejected by the compiler due to the overloading problem, but with methods (as opposed to constructors), the method name can be changed to circumvent the ambiguity hurdle:

```
moveX(int x) { /* ... */ }
moveY(int y) { /* ... */ }
move(int x, int y) { /* ... */ }
```

Method renaming addresses ambiguity but fails to capture the similarity between the three varieties of `move`—each method must be documented, implemented and maintained separately. The alternative offered by default values and keyword parameters is depicted in Fig. 4.

```
class Point {
  private int x, y;

  public Point(int x = 0, int y = 0) {
    this.x = x; this.y = y;
  }
  public void move(int x = 0, int y = 0) {
    this.x += x; this.y += y;
  }
  // ...
}
```

Fig. 4 – An implementation of class `Point` with keyword parameters and default values.

Note that in this case a default arguments mechanism, as in C++, is not enough. Such a mechanism does not provide any way to omit the `x` value. An elegant way to achieve this is by using a combination of keyword and default arguments.

Another overloading difficulty which finds a more elegant solution with named pa-

rameters is that of passing a **null** value. Class String contains eight versions of the static method named valueOf which accept a single parameter. The parameter types are **boolean**, **char**, **char**[], **double**, **float**, **int**, **long**, and Object. The documentation of the valueOf(Object) method states that this method returns **null** if the passed in object is **null**. However, the call String.valueOf(**null**); surprisingly throws a NullPointerException. The reason for this is that this call invokes the version taking **char**[], as this is the most specific method.

This difficulty in inferring the chosen method can be resolved by using keyword parameters, where the name of the **null** parameter is explicitly specified.

3 Support for operands and options

Recall the famous distinction between *operands*—the (usually few) values on which a subprogram operates and (the usually many) *options*—which set the mode of operation [Mey82]. With a keyword argument calling scheme, the designer places operands first on the formal parameter list, allowing them to be called positionally. Options follow in an arbitrary order, with appropriate defaults (See also an ADA style guide[3] that makes recommendations in this spirit, without making the explicit distinction between operands and options.)

Consider for example a method m taking two operands and n options as depicted in Fig. 5(a). With the suggested language extension, this method can be rewritten (Fig. 5(b)) to

```
class A {
  void m(
    String operand1,
    int operand2,
    O1 opt1, O2 opt2,...,
    On optN) {
    //...
  }
}
```

(a)

```
class A {
  void m(
    String operand1,
    int operand2,
    O1 opt1 = defaultExp1,
      ⋮
    On optN = defaultExpN) {
      //...
  }
}
```

(b)

```
new A().m(
  "Restaurant", 42,
  opt17 := E, opt3 := E′);
```

(c)

Fig. 5 – (a) definition of a method taking two operands and n options using plain JAVA syntax, (b) its rewrite with our language extension, and (c) an example of how this method might be called with this extension.

highlight the distinction between operands and options.

Fig. 5(c) demonstrates a call to method m with "Restaurant" and 42 as operands, while setting opt17 to the expression E and opt3 to the expression E'.

The use of overloading would have required 2^n versions of method m; with each of these versions, there is a need to lift the burden of deciding on, and then remembering the order of parameters. And, in the example, it is not even clear that a call such as

```
m("Restaurant", 42, E, E′);
```

[3] http://archive.adaic.com/docs/style-guide/83style/html/sty-05-02.html#5.2.3

would have carried enough information to pinpoint the correct overloaded version of `m`.

The distinction between options and operands penetrated EIFFEL [ISE97]'s standard library, which uses a variety of means [Mey94, 89–92] to avoid passing options as arguments to methods. These means include the placement of setters and getters for shared or per-object option fields within the class, passing values for options to the class constructor, and, in the case of boolean options, writing two distinct versions of the main operation. Fig. 6(a) shows how class `A`, the class enclosing method `m` can be rewritten to include setters for each of the options that this method takes. Fig. 6(b) demonstrates how these setters are used for setting the options.

```
class A {
  void m(String operand1, int operand2) {
    //...
  }
  O1 opt1 = defaultExp1;
        ⋮
  On optN = defaultExpN;
  A setOption1(O1 value) {
    opt1 = value; return this;
  }
        ⋮
  A setOptionN(On value) {
    optN = value; return this;
  }
}
```

(a)

```
new A()
  .setOption17(E).setOption3(E′).m("Restaurant", 42);
```

(b)

Fig. 6 – (a) defining setters for options for the method of Fig. 5, and (b) using these in a concrete call equivalent to Fig. 5(c).

Clearly, the version using our language extension is shorter and clearer, not only on the supplier side, but also, and more importantly, on the client side.[4] Still, Eiffel's approach is a viable alternative if a number of methods defined in a class share options, in which case, a client would only need to learn once how to use these options. In the case that settings of these options tend to be the same in distinct call sites, the Eiffel approach might be preferred.

An even more viable alternative in this case is of using arguments objects [Nob00] as depicted in Fig. 7(a). Fig. 7(b) shows how an arguments object is used while calling this method. Again, the arguments object alternative is longer and not as direct as using a keyword arguments calling scheme.[5]

4 A JAVA extension for keyword and default arguments

This section describes the design and implementation of a JAVA language extension to support an expressive, yet easy to use, defaults mechanism. Our extension features:

[4] Even the number of tokens in each call is smaller; the overhead in terms of token count of setting m options by the EIFFEL approach is $4m$, compared to $3m$ using the proposed language extension.

[5] As indicated e.g., by the $5+4m$ token-count overhead for setting m options with the arguments object alternative.

```
class Options {
  O1 opt1 = defaultExp1;
        ⋮
  On optN = defaultExpN;
  Options setOption1(O1 value) {
    opt1 = value; return this;
  }
        ⋮
  Options setOptionN(On value) {
    optN = value; return this;
  }
}
```

(a)

```
new A().m("Restaurant", 42,
      new Options()
         .setOption17(E)
         .setOption3(E′)
);
```

(b)

Fig. 7 – (a) defining an arguments object class for the method of Fig. 5, and (b) using it in a concrete call equivalent to Fig. 5(c).

1. *Default arguments*—a denotation that a parameter is optional and for supplying a default value for it. This default value is either a constant or an expression, evaluated at the scope of the declared method. Such an expression may involve other parameters, methods and fields of the class in which the method was defined, and any other entities defined at the scope of the declared method.

2. *Keyword arguments*—a mechanism that allows clients to provide arguments in an invocation as name/value pairs. Because each argument is named, the arguments can be supplied in any order; the binding of actuals to formals is carried out automatically by the compiler.

3. *Extended invocation syntax*, in which the list of actual arguments to a method has two, possibly-empty, parts:

   (a) *Positional Arguments List*, in which arguments are supplied in the order they are defined, followed by

   (b) *Keyword-Based Arguments List*, in which each argument is prefixed with the name of the formal parameter.

   This invocation syntax makes it possible for an invocation style which is either entirely *positional*, entirely *keyword based*, or *mixed*.

In C++ a parameter with no default cannot follow a defaulted parameter, and, the default value's expression can not use other formal parameters, non-static data members and functions. In our extension, just as in C++, the initialization expression is computed in the context of the declared method, however, it may involve any function calls and data members access, as long as these are recognized in this context. This makes default value expressions equivalent

to full-blown methods, with the right scoping and late binding properties. Moreover, thanks to keyword-based invocation, in our defaults mechanism the initialization expression may involve values of other parameters to this method and the dependency may be circular. We may declare a method in an `Interval` class such as:

```
public void setInterval(
  int left = right - width,
  int width = 0,
  int right = left + width) {
    //...
}
```

where the `left` parameter depends on the `right` parameter and vice versa. All we require is that if certain parameters are omitted from an invocation, they could be computed from the supplied arguments.

The above definition supports a full positional specification of all parameters,

```
setInterval( 1, 7, 8);
```

just as a full keywords based call

```
setInterval(width := 7, left := 1, right := 8);
```

Both forms allow omitting arguments with defaults: In the partial positional based call

```
setInterval(1, 7);
```

the defaults mechanism will complete the other parameter, setting `right` to `8`. Similarly, in the partial- keyword-based call

```
setInterval(left := 1);
```

the defaults engine will set `width` to zero and `right` to `1`. The circular defaults dependency, i.e., having both `left` depend on `right` and `right` depend on `left` is never a problem. If one of these arguments is missing, its value is computed based on the other. It is however illegal to invoke `setInterval` while omitting both `left` and `right`.

The dependency relationships between parameters are static, determined at compile time. Thus, in the definition

```
public void f(int a, int b, int c = (b != 0) ? a : -1) {
    //...
}
```

the `c` parameter always depends on the `a` parameter, even though the `a` value may not be used in the computation of the `c` value.

4.1 Methods with Default Arguments

4.1.1 Declaration

For each method with default parameters, our modified JAVA compiler computes the set of *calling patterns*, that is, those subsets of the formal arguments from which the remaining arguments can be computed. Every calling pattern with the exception of the pattern including all parameters, is realized as an auxiliary method computing the remaining arguments and then invoking the original method. This method represents the calling pattern in which all arguments are specified. The names of these auxiliary methods are a mangled encoding of (i) the name of the original method, (ii) the types of arguments to the original method, and (iii) the positions of the arguments taking part in the calling pattern.

In the `setInterval` example above every subset of the parameters which includes either `left` or `right` is a proper calling pattern making a total of six calling patterns. We have therefore five auxiliary methods, whose mangled names and signatures are

1. `setInterval.int.int.int_0(int)`
2. `setInterval.int.int.int_2(int)`
3. `setInterval.int.int.int_0_1(int, int)`
4. `setInterval.int.int.int_1_2(int, int)`
5. `setInterval.int.int.int_0_2(int, int)`

Such names are possible since the JAVA virtual machine does not require method names to be valid JAVA identifiers. In fact, we insist on using invalid JAVA identifiers to avoid clashes with other user defined names.

The generation of the auxiliary methods is carried out in the parsing phase of the compiler so that these methods are being attributed and type checked as if they appeared in the source code.

The current implementation does not warn the user if a calling pattern of one method collides with a calling pattern of another method. (This situation happens only if the two methods have the same name, that is, in case of overloaded methods.) For example, the function call `Y.g(a :=3)` is ambiguous if class `Y` is defined as in Fig. 8.

```
class Y {
  static void g(int a, int b = a) {
    System.out.println("Two arguments");
  }
  static void g(int a) {
    System.out.println("One argument");
  }
}
```

Fig. 8 – Two method definitions leading to an ambiguous calling pattern

The compiler does not warn against such a possibility while compiling class `Y`, although it correctly refuses to make the ambiguous call. Other ambiguous situations may occur due to the combination of positional call and default parameters. In the above example, the call `Y.g(17)` is also ambiguous, and this ambiguity would not have been removed if the first argument of one of the functions was renamed.

The preferred order of evaluation of initialization expressions is left-to-right. That is, whenever two arguments can be computed based on the known values of other arguments, then the method realizing a calling pattern computes first the argument occurring first in the parameters list. In other words, the basic step executed repetitively by a method realizing a calling pattern is computing the leftmost computable argument. For example, consider method `f` defined by

```
void f(int x = y+3, int y = 0; int z = 1){
   //...
}
```

and its parameter-less invocation `f();`. The missing arguments are scanned from left to right. The first argument, `x`, can not be computed until the value of the `y` parameter is obtained. Therefore, the `y` parameter is first computed. Next, the remaining missing arguments (that is, `x` and `z`) are scanned again from left to right. In this iteration parameter `x` can be computed based on the value of `y` as computed in the previous iteration. The order of evaluating the arguments is therefore `y` followed by `x` and finally `z`.

4.1.2 Default parameters and inheritance

An overriding method might change the names of the formal parameters, but still in a method invocation the binding of parameters is done by their name in the static type, as depicted in Fig. 9.

```
class X {
  boolean equals(Object x) { /* ... */ }
  //...
}
class Y extends X{
  boolean equals(Object y) { /* ... */ }
  //...
}
```

Fig. 9 – Overriding a method while changing a parameter name.

In this figure method `equals` declared in class `X` is overridden in class `Y` while the name of the parameter is changed. With this hierarchy, the following invocation of `equals` is legal:

```
X x1 = new Y();
x1.equals(x := new Y());
```

since the compiler looks for matching methods based on the static type of the receiver.

An overriding method may also change the initialization expression of a parameter provided that the set of legal calling patterns is not reduced, that is, the set of auxiliary methods of the overridden method is contained in the set of auxiliary methods of the overriding method.

For example,the following overriding of the method `setInterval` is legal:

```
public void setInterval(
  int left = 0,
  int width = 0,
  int right = 0) {
  /* ... */ }
```

The set of auxiliary methods generated by the compiler for the overriding method contains that of the original method. However, a method declared as

```
public void setInterval(
  int left,
  int width = 0,
  int right = left+width) {
  /* ... */ }
```

is not a legal overriding of the same method `setInterval` since a calling pattern which omits `left` is a proper pattern for the original method but not for the overriding method. In this case the compiler issues an error regarding the incompatible default values.

4.1.3 Invocation

When it encounters a method invocation, a JAVA compiler first computes the set of all applicable methods and then looks for the most specific method among all applicable methods. We had to modify these steps in order to support methods with default arguments.

When checking the applicability of a method to a method invocation with keyword arguments, the actual parameters are reordered to match the order of the currently checked method. The usual applicability check is applied to the reordered parameters. For example consider the invocation `f('a',s:="",x:=1);`

```
f(char t,int x,String s){...}
f(char u,Object s,int x){...}
```

Fig. 10 – Two overloaded method definitions.

When this invocation is tested against the first method of Fig. 10 the actual parameters are reordered so that `'a'`, which is an unnamed parameter, remains the first in the list, then `1`, which corresponds to the second formal argument and finally `""`, which corresponds to the formal argument named `s`. When the same invocation is tested against the second declaration in the figure, the order of the actual parameters is `'a'` followed by `""` and then `1`.

Not only methods whose names are identical to the name of the invoked method are applicable. The applicability test is also applied to mangled methods generated from a method declaration whose name matches that of the invoked method.

Finding the most specific method is done by comparing pairs of methods. Each comparison eliminates the least specific one. Before the comparison our modified compiler sorts all formal parameters of the two methods for which the corresponding actual arguments in the invocation are named, by lexicographic order of their names and then applies the usual selection algorithm. For example, both methods declared in Fig. 10 are applicable for the invocation above. Therefore, their arguments are reordered so that the first argument, which is unnamed in the invocation, remains as is, the second one is `s` and finally `x`. Since these methods differ only in the second argument, and `String` is a subtype of `Object`, the first method is more specific than the second one.

4.2 Constructors with Default Arguments

4.2.1 Declaration

The JAVA virtual machine requires every constructor to have the special internal method name `<init>`. Consequently, the method for realizing the different auxiliary methods, that is, using mangling, can not be applied for constructors. Therefore, we encode the difference between auxiliary constructors in the *types of the formal parameters* of these constructors rather than in their names. Every calling pattern of a constructor accepts, in addition to a subset of the original constructor arguments, a designated first argument whose type encodes the specific subset that this calling pattern represents. The auxiliary types are inner classes, residing in the constructor's enclosing class, generated by the compiler during the parsing of a constructor with default values. For example, a constructor of an `Interval` class declared as:

```
public Interval(int left = right - width,
      int width = 0,
      int right = left + width) {
   /* ... */
}
```

has, just like the `setInterval` method above, six calling patterns and five auxiliary constructors whose signatures are:

1. `Interval(JTC.int.int.int_0,int)`
2. `Interval(JTC.int.int.int_2,int)`
3. `Interval(JTC.int.int.int_0_1,int,int)`
4. `Interval(JTC.int.int.int_1_2,int,int)`

5. `Interval(JTC.int.int.int_0_2,int,int)`

As can be seen, the first parameter of each of the auxiliary constructors encodes both the types of the parameters in the original constructor and the positions of the parameters of the calling pattern. All the auxiliary types implement a common interface in order to distinguish auxiliary constructors from user defined ones, and their names start with a "JTC" prefix which stands for Java Type of Constructor.

The body of an auxiliary constructor, just like the body of an auxiliary method, is composed of missing parameters initialization followed by an invocation of another constructor. This pattern is not valid in the JAVA language, which dictates that a constructor invocation can only be the first statement in a constructor body however, this requirement is not enforced by the JVM, where a constructor invocation can follow other instructions.

4.2.2 Invocation

The process of choosing a constructor to invoke is similar to method invocation, that is, computing all applicable constructors and then selecting the most specific one. When computing applicable constructors, the modified compiler first examines each of the user defined constructors. This step is similar to computing applicable methods and includes reordering of the actual parameters. In order to examine the auxiliary constructors, a dummy **`null`** argument is prepended to the actual parameters to be matched against the distinguishing formal parameter.

Selecting the most specific constructor is similar to selecting the most specific method, however, if one of the compared constructors contains a distinguishing formal parameter this parameter is ignored in this stage.

5 Related Work

The case for- (and against-) keyword and default arguments was previously made in the literature. In this section, we review this historical scholarly discussion, and then briefly discuss the contemporary relevance of these, mostly historical arguments.

As early as 1976 Hardgrave [Har76] urged language designers to make the "*additional effort to include keyword parameter technique in their languages*" on the basis of the enhanced *readability* of this technique (with thoughtful selection of parameter names) and its *flexibility* in supporting changes to the order of parameters. The author, who was the first to argue for this language extension, also suggested a mechanism for supplying compile-time constant default values to parameters whose values are seldom "*different from the standard*", and argued that these defaults should relieve clients from the chore of repetitively supplying values to these special arguments.

Three difficulties with default- and keyword- based invocation mechanisms were identified by Hardgrave: (i) *unnecessary verbosity*, which reaches an extreme in the case of single parameter- methods; (ii) *compile time performance decrease*, since the compiler has to bind each actual argument to a formal parameter; (iii) *unintentional parameter omission may go undetected* since the default values would be used.

In 1977, Francez [Fra77] noticed that that keyword-based calling mechanism can be used to reflect not only the name of the formal parameter an argument is bound to, but also the *kind* of binding, that is, "call by value" vs. "call by reference" vs. "call by name". His work even included a concrete proposal for extending the keyword based notation to make this distinction. A year later, Parkin [Par78] drew the proponents' attention to the fact that keyword based calls may lead to an undesirable confusion between the invoking- and the invoked- scope.

In 1982, Ford and Hansche [FH82] claimed that the keyword based method requires a "*somewhat cumbersome coding required in the actual parameter list*". The authors also disliked the practice of writing e.g., `p(,a,,c,,,c,,,x,,);` to denote missing parameters with defaults in the positional method. Instead, they proposed extending the syntax already present for supplying formatting options to PASCAL [Wir71]'s builtin **`write`** procedure call, to user defined subprograms.

Two years later, Winkler [Win84] included keyword based calls to subprograms among his proposed additions and improvements to ISO-Pascal, while noting an interesting difficulty in passing subprograms as parameters to other subprograms. The mechanisms however did not make it into the language, nor to C [KR88] or OBERON [WR92], languages which may be perceived by some as PASCAL's successors.

Interestingly, ADA designers, observing [IBFW81, Chap. 8.3] that the positional scheme suffers from the disadvantage that

> "*. . . with more than three or four parameters it is hard to follow the text.*"

decided to include in their language a keyword based calling scheme arguing that it "*provides especially high readability*". This calling scheme was used alongside with the "*almost universal*" positional calling mechanism. These four scholars also made the case for allowing both mechanisms in tandem arguing that

> "*Clearly in many contexts the order of parameters is either highly conventional (as for coordinate systems) or immaterial (as in* `MAX(X,Y)`*). Hence* ADA *admits both conventions. The classical positional notation may be used whenever the programmer feels that keyword parameters would add verbosity without any gain in readability.*"

ADA thus allowed mixed positional- and keyword-based- calling mechanism, which was further enriched, as in our JAVA extension, with a default parameter facility which together provide "*a high degree of expressivity and readability*".

The "selectors" of *keyword methods* of SMALLTALK [Gol84] and of OBJECTIVEC [Cox86], e.g., the two arguments `at:` `put:` selector, resemble keyword-based- calling scheme. The similarity is superficial since the invoker may not change the order of parameters, and `at:` `put:` is entirely different from the selector `put:` `at:`. Indeed, earlier versions of the language [KG76] did not even insist on using a keyword in front of every parameter, allowing invocations such as

```
displayFrame put 'hi there' at 150 100
```

In the course of years, several languages in common use, including PERL [Wal94], PYTHON and Transact-SQL stored procedures, have adopted a keyword-based calling scheme. Some of the historical arguments, for- and against- keyword based calling are clearly defunct now: the overhead in compilation time is negligible; Parkin's comment on the confusion of scopes is not as relevant with the growing tendency of using small scopes; The syntax of Ford and Hansche failed to spike enthusiasm; and, the verbosity of keyword-based calls is addressed by the growing understanding that if keyword based parameter passing is allowed, it should be *in addition* to the positional scheme.

The arguments that stayed are that keyword-based calls are, when used appropriately, more flexible, more readable, and more expressive. Indeed, a recent internet page[6] continues the debate, reiterates these arguments, also points out that despite disagreement whether it is easier to remember parameters based on their name and their position, such memorization is totally

[6] http://c2.com/cgi/wiki?KeywordParameterPassing

irrelevant with modern development environments. Similarly, such environments minimize the effort of maintaining the correctness of an invocation in the face of changes to the parameter names or to their order.

The third disadvantage pointed by Hardgrave, that is the risk of unintentionally omitting parameters, is still applicable. However, undetected semantic mistakes may occur in positional invocations as well, for instance, when switching two arguments of the same type in a method invocation.

This combination is used now in several programming languages including PYTHON, LISP and MESA as well as C# 4.0. However, to the best of our knowledge, there was no study of the confusion and abuse that this combination may create.

6 Conclusions

In this paper we argued that there is a clear and present need for an inherent support of keyword and default arguments in JAVA. We based our claim on previously published work, in which it was determined that the most frequent use of overloading is for simulating defaults arguments; this use pattern occurring in about a third of the cases in which overloading is used.

We discussed the advantages of a designated keyword and defaults mechanism in JAVA over the existing solution of method overloading in terms of code length, implications on documentation, maintenance and client's learning curve, and flexibility in handling situations in which several arguments are of the same type.

We bring a proof of concept implementation of keyword and default arguments in JAVA. Our implementation, which does not impinge on the runtime environment, allows supplying defaults to any subset of the formal parameters, and admits method invocations in a *positional* style, *nominal* style (i.e., where the role of parameters is determined by prefixing these with their name rather than their position in the arguments list), or *mixed* style. We argue that this extension may address the famous options-operands dilemma well.

Although such a mechanism can drastically reduce the amount of overloading in JAVA code, it raises its own questions of abuse and confusion. For example, the interaction of default arguments and overloading may lead to confusing semantics and ambiguity. Fig. 8 depicts a situation in which an ambiguous method call may occur. The solution we chose in our extension is to report an error for the ambiguous call. C#'s implementation is different, giving precedence to methods that have no default arguments. In both cases the interaction of default arguments and overloading makes the code confusing and hard to maintain.

There are other choices made in our extension which are different from C#. For example, the evaluation order of the arguments. Our implementation scans the arguments by the order in which they appear in the method *definition*, while arguments are evaluated in their order in the method *invocation* in C#. This difference is significant if evaluating the arguments has side effects.

The implementation of named and optional arguments in C# differs from our extension also by the restrictions imposed on initialization expressions. While in C# only constants may be used as default values, our extension is more flexible, allowing method invocations, data members access and other formal arguments to be used in initialization expressions.

Our design choice of extending the language without modifying the JVM requires the creation of synthetic methods and types. While this implementation approach is sometimes used in JAVA compilers (e.g., **`enum`**s introduce additional types, bridge methods are generated by the compiler to support return type covariants), it may effect runtime tools such as profilers and debuggers, and confuse programmers that use reflection.

The effect of the language extension on performance should be measured. We expect that the extension carries a performance penalty during compilation, mainly due to parameters reordering done for method binding. But, of course a comprehensive benchmark is required to measure the performance effect both on compilation time and on execution time.

References

[AA10] J. Albahari and B. Albahari. *C# 4.0 in a Nutshell*. O'Reilly Media, 2010. Available from: http://books.google.co.il/books?id=VENrFSQFco8C.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[BS07] Antoine Beugnard and Salah Sadou. Method overloading and overriding cause distribution transparency and encapsulation flaws. *Journal of Object Technology*, 6(2):31–46, 2007. doi:10.1145/1141277.1141608.

[Cox86] Brad J. Cox. *Object-Oriented Programming - An Evolutionary Approach*. Addison-Wesley, Reading, Massachusetts, 1986.

[FH82] Gary Ford and Brian Hansche. Optional, repeatable, and varying type parameters. *SIGPLAN Not.*, 17(2):41–48, 1982. doi:10.1145/947902.947906.

[Fra77] Nissim Francez. Another advantage of keyword notation for parameter communication with subprograms. *Communications of the ACM*, 20(8):604–605, 1977.

[GL10] Joseph Gil and Keren Lenz. The use of overloading in java programs. In Theo D'Hondt, editor, *Proc. of the Twenty Fourth European Conference on Object-Oriented Programming (ECOOP'10)*, volume 6183 of *Lecture Notes in Computer Science*, pages 529–551, Maribor, Slovenia, June21–25 2010. Springer-Verlag. doi:10.1007/978-3-642-14107-2_25.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Massachusetts, 1984.

[Gra95] Paul Graham. *ANSI Common LISP*. Prentice Hall, 1995.

[Har76] W. T. Hardgrave. Positional versus keyword parameter communication in programming languages. *ACM SIGPLAN Notices*, 11(5):52–58, 1976. doi:10.1145/956003.956008.

[IBFW81] J. Ichbiah, J. Barnes, R. Firth, and M. Woodger. *Rationale for the design of the Ada programming language*. Cambridge University Press, New York, NY, USA, 1981. doi:10.1145/956653.956654.

[ISE97] ISE. *ISE EIFFEL The Language Reference*. ISE, Santa Barbara, CA, 1997.

[KG76] A. Kay and A. Goldberg. Smalltalk-72 instruction manual. Technical Report SSL-76-6, Xerox Corporation, Palo Alto, California, 1976.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Software Series. Prentice-Hall, Inc., second edition, 1988.

[Lut96] Mark Lutz. *Programming* PYTHON. O'Reilly, first edition, October 1996.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. on Prog. Lang. Syst.*, 16(6):1811–1841, 1994. doi:10.1145/197320.197383.

[Mey82] Bertrand Meyer. Principles of package design. *Communications of the ACM*, 25(7):419–248, July 1982. doi:10.1145/358557.358565.

[Mey92] Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, 1992. doi:10.1109/2.161279.

[Mey94] Bertrand Meyer. *Reusable Software The Base Object-Oriented Component Libraries*. Prentice Hall Object-Oriented Series. Prentice-Hall, Inc., 1994.

[Mey01] Bertrand Meyer. Overloading vs. object technology. *Journal of Object-Oriented Programming*, pages 3–7, 2001.

[Nob00] James Noble. Arguments and results. *The Comp. J.*, 43(6):439–450, 2000. doi:10.1093/comjnl/43.6.439.

[Par78] Rodney Parkin. On the use of keywords for passing procedure parameters. *SIGPLAN Not.*, 13(7):41–42, 1978. doi:10.1145/953863.953870.

[Smi98] Jerry Smith. Avoid 'constructor madness'. *JavaWorld: IDG's magazine for the Java community*, 3(11), 1998. Available from: http://www.javaworld.com/javaworld/javatips/jw-javatip63.htm.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[TD97] S. Tucker Taft and Robert A. Duff, editors. *Ada 95 Reference Manual, Language and Standard Libraries, International Standard ISO/IEC 8652: 1995(E)*, volume 1246 of *LNCS*. Springer-Verlag, 1997.

[Wal94] Larry Wall. *The Perl Programming Language*. Prentice Hall Software Series. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1994.

[Win84] J. F. H. Winkler. Some improvements of ISO-Pascal. *SIGPLAN Not.*, 19(7):65–78, 1984. doi:10.1145/948596.948604.

[Wir71] N. Wirth. The programming language Pascal. *Acta Inf.*, 1:35–63, 1971.

[WR92] N. Wirth and M. Reiser. *Programming in Oberon—Steps Beyond Pascal and Modula*. Addison-Wesley, Reading, Massachusetts, 1992.