# Overloading and Inheritance in Java
## (Extended Abstract)

Davide Ancona[1], Elena Zucca[1], and Sophia Drossopoulou[2]

[1] DISI, University of Genova
[2] Department of Computing, Imperial College
{zucca,davide}@disi.unige.it sd@doc.ic.ac.uk

**Abstract.** The combination of overloading and inheritance in Java introduces questions about function selection, and makes some function calls ambiguous. We believe that the approach taken by Java designers is counterintuitive. We explore an alternative, and argue that it is more intuitive and agrees with the Java rules for the cases where Java considers the function calls unambiguous, but gives meaning to more calls than Java does.

## 1 Overloading and Inheritance

Overloading, already present in the seventies (LIS, Ada, Hope), allows the definition of several, different, functions with the same name and different parameter types. Thus, the programmer is freed from the burden of dreaming different identifiers for functions which perform essentially the same operation, but on different types of parameters. Overloading is usually resolved statically[1], namely, the function that fits the actual parameter types is selected. Thus, overloading resolution corresponds to consistent renaming of the function definitions and the corresponding function calls.

Inheritance, already present in the sixties (Simula), allows classes to be organized in a class hierarchy, and either to inherit functions from their superclass or to redefine these functions. When a function is called for a certain object, the function from the most specific superclass is called. Resolution for inheritance can only take place at run-time, and depends on the dynamic class of the receiver. Inheritance introduces subtyping, namely an object of a subclass may be used where an object of a superclass is expected.

The combination of subtyping and overloading is not straight-forward, since now more than one method may fit the types of the arguments of a function call.

*Java overloading resolution* Assume that Oyster is a subclass of Food, and that aPhl, pascal, aFood and anOyster are variables of type Phil, FrPhil, Food and Oyster respectively. Consider the following example:

[1] Dynamic resolution is also possible, as it actually happens in object-oriented languages for the first argument and for all arguments in languages with *multimethods*; see [1] for a deep analysis of the difference. In this paper, we only consider overloading with static resolution.

```
class Phil extends Object {
   int eat ( Food x ) {  return 1; }
   int eat ( Oyster x ) {  return 2 ;}
}

class FrPhil extends Phil {
   int eat ( Food x ) {  return 3; }
}
```

For the method call aPhl.eat (anOyster ), two methods are applicable, both declared in the class Phil: one with parameter type Food (returning 1) and one with parameter type Oyster (returning 2).

In cases where several methods are applicable, Java (and C++ before) took the approach of selecting the method that "fits" best, called *most specific*.[2] In the call from above, it is clear that the method with parameter Oyster fits better than the method with parameter Phil, therefore this method is selected. In general, if many methods declared in the same class are applicable, then that with most specific argument's type is selected, if any, otherwise[3] the call is ambiguous.

Much less obvious is the case when we have to compare methods declared in different classes, like in the method call pascal.eat (anOyster). Here, both the method returning 2 and that returning 3 are applicable, and in Java's view *none* fits better than the other, hence the method call is ambiguous.

In our experience, many people (even with a deep knowledge of Java) are unaware of the implication of the Java overloading resolution in this case, and expect on the contrary that the method returning 2 is selected. In our opinion this is so, because the latter solution corresponds to an intuitive understanding of inheritance semantics. This is explained in the next section.

*Alternative overloading resolution* In our view the method call pascal.eat (anOyster) should *not* be ambiguous, and should return 2. This view is based both on methodological and language semantics reasons.

On the methodological side, an implication of the Java rule is that programmers who use a class and want to be aware of how method overloading will be resolved need to know not only which methods are inherited, but also the exact class containing the definition of these inherited methods. This conflicts with a modular approach to software development where all the information needed for the correct use of a module (class in this case) should be provided by its specification alone.

For instance, the specification of class FrPhil states that this class has two methods int eat ( Food x ) and int eat ( Oyster x ). However, with this information only, users *cannot* know which will be the effect of the call pascal.eat (anOyster).

[2] Another solution would be to avoid the occurrence of such situations where several methods are applicable, and forbid the definition of overloaded methods with parameter types with non-empty intersections of sets of values. This would make the class Phil from above illegal. However, such a solution would restrict overloading only to non-class, non-interface parameter types, since the value **null** belongs to all classes.

[3] For instance, if there are two applicable methods with two arguments with types Food,Oyster and Oyster,Food, respectively.

The counterpart at the level of language semantics is that inheritance should be explainable as a mechanism for code sharing. In other words, a natural intuitive understanding of inheritance is as a linguistic mechanism allowing to get for free the same effect that one could obtain "by hand" by duplicating parent's code in the heir. Thus, the subclass FrPhil should be equivalent to a copy of Phil, where the overriden methods of Phil are replaced by the corresponding methods from FrPhil. That is, FrPhil should be equivalent to FrPhil_by_Copy, defined as:

```
class FrPhil_by_Copy extends Phil {
    int eat ( Oyster x ) { return 2; }
    int eat ( Phil x ) { return 3; }
}
```

Then, for a variable rousseau of class FrPhil_by_Copy, the call rousseau eat (anOyster ), would be unambiguous, and would return 2. Therefore, by analogy, the call pascal.eat (anOyster) should return 2 as well!

## 2 The alternative approach "subsumes" the Java approach

The alternative approach corresponds to the Java approach for all cases where Java considers the method call unambiguous. As we have seen, in some cases where Java considers the call ambiguous the alternative gives it an unambiguous meaning.

The rest of the paper is devoted to the proof of this claim.

For simplicity, and without restricting the applicability of our result, we assume that all methods have one parameter.

Moreover, we start from considering only non-abstract classes. The generalization to interfaces and abstract classes requires more involved definitions and will be considered in the full paper [2].

Both the Java approach and the alternative approach start from the same set of *applicable* methods, which are all the methods of the receiver's class, either directly declared or inherited (that is, declared in a superclass and not overidden, *cf.* [3] 15.11.1) , which are type compatible with the given method call, *cf.* [3] 15.11.2.1), but they differ in the way they consider methods to "fit better" than others.

Sets of applicable methods are denoted by $\mathcal{A}$, $\mathcal{A}'$ *etc.*, and contain method *types*. Method types consist of the class containing the method declaration, the argument type and the result type.

For example, the applicable methods for the call aPhl.eat (anOyster ) are
$\mathcal{A}_1 = \{< \mathsf{Phil}, \mathsf{Food}, \mathsf{int} >, < \mathsf{Phil}, \mathsf{Oyster}, \mathsf{int} >\}$.
Also, the applicable methods for the method call pascal.eat (anOyster ), are
$\mathcal{A}_2 = \{< \mathsf{Phil}, \mathsf{Oyster}, \mathsf{int} >, < \mathsf{FrPhil}, \mathsf{Food}, \mathsf{int} >\}$.

In Java, a method "fits better" than another one if the former is defined in a subclass of where the latter is defined and the argument types of the first widen[4] to the corresponding argument types of the second, *cf.* [3] 15.11.2.2.

[4] A type t widens to another type t′ if they are identical, or t is a subclass of t′, or t is a subinterface of t′, or t is a subclass of a class that implements a subinterface of t′, *cf.* [3] 5.1.4 .

On the other hand, for the alternative definition, a method "fits better" than another one if the argument types of the former widen to the corresponding argument types of the latter.

So, we define the following two ordering relationships on method types:

- $< t_1, t_2, t_3 > \ll_j < t'_1, t'_2, t'_3 >$ iff $t_1$ subclass of $t'_1$ and $t_2$ widens to $t'_2$
- $< t_1, t_2, t_3 > \ll_a < t'_1, t'_2, t'_3 >$ iff $t_2$ widens to $t'_2$

Notice that, by definition of applicable methods, there can be at most *one* applicable method with a given name and argument type, hence the ordering relationship $\ll_a$ can be equivalently defined as follows:

$< t_1, t_2, t_3 > \ll_a < t'_1, t'_2, t'_3 >$ iff $t_2$ widens to $t'_2$, $t_2 \neq t'_2$, or $t_2 = t'_2$ and $t_1$ subclass of $t'_1$

The equivalence follows from the fact that if $t_2 = t'_2$ then also $t_1$ subclass of $t'_1$ for the reason explained above. This formulation point out that the two ordering relationships correspond to two different ways of combining the ordering relationships existing on the first and second component of method types, that is, componentwise and lexicographical from right to left.

For example, $< \mathsf{Phil}, \mathsf{Oyster}, \mathsf{int} > \ll_j < \mathsf{Phil}, \mathsf{Food}, \mathsf{int} >$, and similarly, in the alternative approach, $< \mathsf{Phil}, \mathsf{Oyster}, \mathsf{int} > \ll_a < \mathsf{Phil}, \mathsf{Food}, \mathsf{int} >$. This makes the call `aPhil.eat (anOyster )` unambiguous in both approaches.

On the other hand, $< \mathsf{Phil}, \mathsf{Oyster}, \mathsf{int} > \ll_a < \mathsf{FrPhil}, \mathsf{Food}, \mathsf{int} >$, but the method types are incomparable in the sense of $\ll_j$. So, `pascal.eat (anOyster )` is unambiguous in the alternative approach and ambiguous in Java.

It is easy to see that both $\ll_j$ and $\ll_a$ are reflexive and transitive. The relation $\ll_j$ is antisymmetric if the program is well-formed (*i.e.,* if the subclass relationship is acyclic), whereas the relation $\ll_a$ is *not* antisymmetric: a counterexample would be a method defined in class $c_1$ and defined with the same parameter type in whichever different class $c_2$. Also, one can immediately see that $\ll_j$ is stronger than $\ll_a$, *i.e.,* that:

$$< t_1, t_2, t_3 > \ll_j < t'_1, t'_2, t'_3 > \ \Rightarrow\ < t_1, t_2, t_3 > \ll_a < t'_1, t'_2, t'_3 >$$

Finally, the type of a method call with applicable methods $\mathcal{A}$ is defined as the return type of the least method type from $\mathcal{A}$ in the sense of the ordering either $\ll_j$ (in Java) or $\ll_a$ (in the alternative). If this minimum does not exist, then the method call is ambiguous.

It only remains to be shown that if a set $\mathcal{A}$ of applicable methods has a least element in the sense of $\ll_j$ then this is also the least element in the sense of $\ll_a$. This is easy, because $\ll_j$ implies $\ll_a$.

This completes the proof that the alternative approach is a conservative extension of the Java approach, in the sense that it gives the same meaning to all the method calls which are unambiguous for Java.

*Another alternative* As stated above, the two ordering relationships correspond to two different ways of combining the ordering relationships existing on the first and second component of method types. It is natural therefore to also consider a third possibility, which corresponds to lexicographical order from left to right, that is:

$< t_1, t_2, t_3 > \ll_{a2} < t'_1, t'_2, t'_3 >$ iff $t_1$ subclass of $t'_1$, or
$t_1 = t'_1$ and $t_2$ widens to $t'_2$

Such a rule resolves overloading by selecting the method that fits *first*. It searches from the most specific subclass following the superclass hierarchy upwards, and only takes those overloaded methods into account which were declared in the first superclass that contains applicable methods. With this rule the method call `pascal.eat (anOyster)` would select the method returning 3. However, this does not influence of course the Java rule for overriding; so, for instance, in the call aPhil.eat(anOyster) the method declared in `Phil` with argument type `Oyster` is selected (and kept at run time) even in the case `aPhl` has dynamic type `FrPhil`.

It is easy to see that this alternative too is conservative (and less restrictive) w.r.t. Java rule; indeed, also $\ll_{a2}$ is implied by $\ll_j$ . We will further investigate the methodological implications of this third possibility.

## 3 Outline of the full paper

We have argued that the Java approach to overloading resolution considers ambiguous some method calls which would have a meaning if we had taken a more intuitive view of inheritance, based on copying. We have given an alternative rule for overloading resolution which gives meaning to more calls than Java does and gives the same meaning as Java when Java considers a method call unambiguous, and have proven this result.

We briefly illustrate now which are the further topics which will be developed in the full paper [2].

*Methodological aspects* We will include a survey of the design space of the semantics of method overloading in the presence of subtyping. In particular we will discuss more extensively advantages and disadvantages of the different possible choices and analyze what happens in other object-oriented languages, notably in C++.

*Extension to abstract classes and interfaces* The main difference w.r.t what has been previously presented is that when considering also abstract classes and interfaces more than one method with the same signature an be inherited, as explicitly stated in [3] 8.4.6.4.

Hence, now in applicable methods there can be two methods with the same name and argument type.

For instance in the following example

```
interface I1 {
    void m ( ) ;
}

interface I2 {
    void m ( ) ;
}
```

```
interface I extends I1, I2 {
}
```

the applicable methods for a call i.m() with i of type I are

$$\{< \mathsf{I1}, \mathsf{void}, \mathsf{void} >, < \mathsf{I2}, \mathsf{void}, \mathsf{void} >\}.$$

Hence this call should be ambiguous following the Java rule in [3] (even though different Java compilers have different, and sometimes obscure, behaviour on this and similar examples), while the alternative approach corresponds to assume that the interface I has just *one* method void m(), regardless of how many copies have been inherited, hence the call is not ambiguous.

We will show that it is possible to generalize the previous formalization of the two rules and that the result that $\ll_{\mathsf{a}}$ is a conservative extension of $\ll_{\mathsf{j}}$ still holds.

An interesting remark is that the above situation shows that there is a contradiction in [3] between the *definition* of overloading given in 8.4.7 and the rule for overloading resolution, and that this contradiction disappears if one considers the alternative rule.

*Copy semantics of inheritance* We have repeatedly stated that the alternative approach corresponds to an intuitive interpretation of inheritance as a mechanism achieving the same effect one would have by copying parent's code in the heir. In some more detail, that means that a simple way for expressing inheritance semantics is to translate a Java program in an intermediate representation (which we call Flat Java) consisting, roughly speaking, of a subtype-hierarchy part and a collection of "flat" classes (that is, without any extends or implements clause). Methods of one of these classes are all the methods (either directly declared or inherited) of the original Java class.

In Flat Java there is no longer notion of inheritance, hence in method calls there is no need for method look-up. Of course, the information about the subtyping relation is still needed, *but only* for type-checking bodies of methods, while no information about that is needed at run time.

This model corresponds to a natural intuitive understanding of the basic inheritance mechanism in object-oriented languages; some more sophisticated features of Java, like the super mechanism and the possibility of hiding fields, do not have a direct copy semantics but can be easily simulated.

In [2] we will provide a formal definition of copy semantics by means of a translation from Java into Flat Java and show that, for what concerns overloading, copy semantics leads to the alternative approach we proposed, while the Java rule is not directly expressible.

## References


1. G. Castagna. *Object-Oriented Programming: A Unified Foundation.* Progress in Theoretical Computer Science. Birkhäuser, 1997.
2. D. Ancona, E. Zucca, and S. Drossopoulou. Overloading and inheritance in Java. Technical Report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 2000. In preparation.
3. J. Gosling, B. Joy, and G. Steele. *The Java™ Language Specification.* Addison-Wesley, 1996.