Subtyping in Java with Generics and Wildcards is a Fractal
(a casual essay)

Moez A. AbdelGawad
moez@cs.rice.edu

December 9, 2014

Abstract
For helping themselves in writing, debugging and maintaining their software, professional OO software developers keep in their minds an image of the subtyping relation between types in their software while they are developing their software. In pre-generics Java [17], the structure of the subtyping mental image was simple: the graph of the subtyping relation between classes and interfaces (i.e., with multiple-inheritance of interfaces) was a directed-acyclic graph (or DAG), and the graph of the subtyping relation between classes alone (i.e., with single-inheritance only, more accurately called the ‘subclassing’ relation) was simply a tree. Trees are well-known data structures, and a DAG, in short, is essentially a tree where a node is further allowed to have more than one parent node (i.e., not just one parent as in a tree) but the node cannot be a parent of itself, even if indirectly; a DAG can thus have no cycles, hence being ‘acyclic.’ This fact about the graph of the subtyping relation applied not only to Java but, more generally, also to the non-generic sublanguage of nominally-typed OO languages similar to Java, such as C# [1], C++ [2], and Scala [21].

The goal of this casual essay is to present and defend, even if incompletely and not quite rigorously, a hunch and intuition the author had years ago about the graph of the subtyping relation in Java after generics were added to it. The author observed that: after the addition of generics—and of wildcards in particular—to Java, the graph of the subtyping relation is still a DAG, but no longer a simple DAG but rather one whose structure can be better understood as a fractal. Today, generics and wildcards (or some other form of ‘variance annotations’) are a standard feature of mainstream nominally-typed OO languages similar to Java, such as C# [1], C++ [2], and Scala [21].

The goal of this casual essay is to present and defend, even if incompletely and not quite rigorously, a hunch and intuition the author had years ago about the graph of the subtyping relation in Java after generics were added to it. The author observed that: after the addition of generics—and of wildcards in particular—to Java, the graph of the subtyping relation is still a DAG, but no longer a simple DAG but rather one whose structure can be better understood as a fractal. Today, generics and wildcards (or some other form of ‘variance annotations’) are a standard feature of mainstream nominally-typed OO languages. Accordingly, the shape of the subtyping relation in nominally-typed OO languages is more complex than a tree or a simple DAG. Given the popularity of fractals, the fractals observation may help OO software developers keep a useful and intuitive mental image of their software’s subtyping relation, even if it is one a little more frightening, and amazing, than before.
1 Main Observation

Fractals (sometimes also called recursive graphs, or self-referential graphs) are drawings or graphs that are characterized by having “minicopies” of themselves inside of them. Given their self-similar nature, when zooming in on a fractal it is not a surprise to find a copy of the original fractal spring up. More generally, the minicopy is not an exact copy, but some transformation of the original: it may be the original rotated, translated, reflected, and so on. As such, when constructing a fractal iteratively it is also not a surprise to add details to the construction by using (transformed) copies of the fractal as constructed so far (i.e., as it exists in the current iteration of the construction) to get a better, more accurate approximation of the final fractal. (See Figure 1.1.)

While the observation may not be immediately obvious to the reader, but “having transformed minicopies of itself” is exactly what the author has noticed also happens in (the graph of) the subtyping relation of Java—and of other similar generic nominally-typed OO languages such as C#, C++, and Scala—after generics and wildcards were added to the Java type system. Figure 1.2 presents a drawing of the first steps in the construction of a subtyping graph. A slightly more precise and more detailed version of the drawing—that uses no “raw types,” and has an additional class D—is presented below.

2 Observation Illustrated

To illustrate the main observation, and to motivate a subsequent related one we make in the next section, let us assume we have the non-generic class Object (which extends no other classes), and have generic classes C and D that extend (a.k.a., subclass) class Object and that take one (unbounded) type parameter. Importantly, we also assume we have a “hidden” (i.e., inexpressible in most OO languages) non-generic class Null at the bottom of the class hierarchy whose only instance is the null object (which in Java is an instance of every class and can be assigned to a variable of any object type. The possibility of having non-nullable classes in some OO languages, e.g., C#, may need some special provision, such as having an additional class Empty that extends class Null and

---

1This observation, first made in 2007, and it seemingly not being made by anyone else since then, is the main motivation behind writing this essay—seven years later!

2The drawing was included in a demonstration poster on Java wildcards, in particular on the possibility of them getting better understood as being intervals over the partially-ordered set of types when ordered by subtyping. The poster was developed while the author was a Ph.D. graduate student at Rice University.

3With wildcards, particularly the wildcard type ?, it is unnecessary to use raw types as the “default” type argument of generic classes (as class C was used in the 2007 graph.)

4Even when the instanceof operator in Java, only for developers’ convenience, returns false as the value of the expression ‘null instanceof C’ for any class C.
Figure 1.1: Fractals: (First Steps in Constructing) The Koch Curve and (a Step in Constructing) a Fractal Tree

Figure 1.2: First Iterations of Constructing a Subtyping Graph in Java (Reproduced, with permission, from a 2007 poster by the author on interpreting Java wildcards as intervals)
has no instances. For domain-theorists, the class type corresponding to class Empty will be the empty object domain, i.e., one whose only “instance,” or member, is \( \perp_O \), “the non-terminating object.”

Figure 2.1 demonstrates the subclassing hierarchy based on assuming these class declarations.

The declared inheritance relation between class (and interface) names in a program (as in Figure 2.1) is the starting point for constructing the graph of the subtyping relation in Java (Note the use of the identification of inheritance and subtyping in nominally-typed OOP [5, 7, 8, 10, 8, 15, 6] to interpret ‘class extension’ as ‘subtyping between corresponding class types.’) Figure 2.2 shows that the “default type argument,” namely \(?\) (the unbounded wildcard type), is used in this step as the type argument for all generic classes to form type names of corresponding class types.

Figure 2.3 demonstrates that the (names of) types in the next level (level \( i+1 \), zooming one step closer) in the subtyping relation are constructed by replacing all the \(?\) in level 0 with three different forms of each type \( T \) in the previous level (level \( i \)), namely \(? \text{ extends } T\) (covariance), \(? \text{ super } T\) (contravariance), and \( T\) (invariance). Covariant, contravariant and invariant subtyping rules are then used to decide the subtyping relation between all the newly constructed types. (Note that, due to the inclusion of Object and Null in level 0 and all subsequent

---

\footnote{Or, “the initial graph,” or “the base step” of the recursion “crank,” or the “skeleton” [12] of the graph, or the “seed” [12] of the fractal.}

\footnote{Which corresponds to equation: \( G = G_0(I(G)) \) (See addendums below). The following is inaccurate? Replacing each of the innermost (or, all?) \(?s\) of a type (“holes” in the type) in level \( i \) with a \# (a hash, as a placeholder), then replacing these \#s with three different forms of each one of the types in the previous level (level \( i \)) or in level 0 to construct names of the types of the new level. (corresponding to equation \( G = G(I(G)) \) or \( G = G(I(G_0)) \))}
levels, all level \( i \) types are also types in level \( i + 1 \). The first level/iteration in which a type appears is called the rank of the type. Thus, types \texttt{Object} and \texttt{Null} are of rank 0.)

Note 1: Types \texttt{Null}, \texttt{C<Null>}, and \texttt{D<Null>}, in dotted graph nodes in Figure 2.3 and Figure 2.4, are currently inexpressible in Java (\textit{i.e.}, as of Java 8, based on the assumption that these types are of little use to developers.) Subtyping relations involving these types are also currently of little use to Java developers, and, accordingly, also are drawn in Figure 2.4 using dotted graph edges.

Note 2: Also as of Java 8, Java does not currently identify \texttt{? super Object} with \texttt{Object}, and as such a variable \( b \) of type \texttt{C<? super Object>} (\textit{i.e.}, for \( a=b \); the Java compiler \texttt{javac} currently emits a type error with an unhelpful semi-cryptic error message that involves so-called “wildcard capturing”) even as Java allows the opposite assignment of \( a \) to \( b \) (\textit{i.e.}, \( b=a; \)), implying that, even though Java currently correctly sees \texttt{C<Object>} as a subtype of \texttt{C<? super Object>}, it currently does \textit{not} consider \texttt{C<? super Object>} as a subtype of \texttt{C<Object>}. Given that there are no supertypes of type \texttt{Object} (the class type corresponding to class \texttt{Object}), and it is not expected there will ever be any, we believe the Java type system should be improved to identify the two type arguments \texttt{? super Object} and \texttt{Object}, and thus correctly allow the mentioned current-
Object -> C<? extends T> -> Null, Object -> D<? extends T> -> Null
(Note: ‘? extends Null’ is the same as ‘Null’. Inexpressible in Java)

(with the level 0 graph copied inside C<?> and D<?>. Includes green arrows in subgraphs.)

(a) Subtyping with Covariance (10 Types: From 2 non-generic classes + 2 generic classes × 4 types in level 0)
Object -> C<? super T> -> Null, Object -> D<? super T> -> Null
(Note: ‘? super Object’ is the same as ‘Object’. See note below on current Java behavior)

(with the level 0 graph turned upside-down inside C<?> and D<?>. Includes red arrows in subgraphs.)

(b) Subtyping with Contravariance (10 Types: As for covariance)
Object -> C<T> -> Null, Object -> D<T> -> Null

(with the level 0 graph flattened inside C<?> and D<?>. No arrows in subgraphs)

(c) Subtyping with Invariance (10 Types: As for covariance)

Figure 2.3: Subtyping: (First) Inductive Step. Nesting Level 1
Object $\rightarrow$ C$<$T$>$ $\rightarrow$ Null, Object $\rightarrow$ D$<$T$>$ $\rightarrow$ Null (For brevity, ‘Object’ $\rightarrow$ ‘O’, ‘Null’ $\rightarrow$ ‘N’)

(with all three transformations applied to level 0 graph and embedded inside C$<$?$>$ and D$<$?$>$. Note that bounds of an interval can degenerately be equal types, corresponding to invariance)

(The Null class is useful in expressing intervals. Can be done without, using extends only or super only while allowing but not requiring a ?; or, for brevity, using a symbol like <:).

Figure 2.4: Subtyping Generalization with Intervals. Nesting Level 1 (20 Types: From 2 non-generic classes + 2 generic classes $\times$ 9 intervals in level 0. If class C or class D have subclasses other than Null, this graph will be richer—i.e., will have more types—than the graph in Figure 2.3)
disallowed assignments.

(Some “graphs” in the figures above are not presented in the most familiar form. Depending on time availability, those graphs will soon get expanded, drawn in a more familiar form, and polished using some graph-drawing software, e.g., yEd or Tulip. The updated graphs will be included in a future version of this essay.)

Afterwards, each next nesting level of generics corresponds to “zooming one level in,” and is done in the same way as above, where wildcards (or, intervals) over the previous subtyping graph substitute all the ? in that graph to produce the next level graph of the subtyping relation. And there is nothing in generics that prevents from arbitrarily-deep, potentially infinite, nesting.

3 Transformations Observation

In the graph of the subtyping relation, when moving from types of a specific level of nesting to types of the next deeper level (i.e., when “zooming in” inside the graph of the relation, or when doing the inductive step of the recursive definition of the graph), we can notice that three kinds of transformations are applied to the level $i$ subtyping graph. We call these the identity transformation, the upside-down reflection, and the flattening transformation. The first transformation (identity) makes an exact copy of the input subtyping relation, the second transformation (upside-down reflection) flips over the relation (a subtype in the input relation becomes a supertype, and vice versa); while the third transformation “attempts to do both (i.e., the identity and flipover transformations),” in effect making types that were related in its input subtyping relation be unrelated in its output subtyping relation (hence the output of the transformation is a “flat” relation, mathematically called an anti-chain.) A striking specific observation in the ‘subtyping fractals observation’ is that the three mentioned transformations correspond to (and in fact result from) the covariant subtyping rule, contravariant subtyping rule, and invariant subtyping rule, respectively. This is demonstrated, in a very abridged manner, in Figures 2.3a, 2.3b, and 2.3c (with the green arrows corresponding to copying the previous level graph, corresponding to covariant subtyping, the red arrows corresponding to flipping over the previous level graph, corresponding to contravariant subtyping.) It should be also noted that the level 1 graph as a whole is the same structure as the level 0 graph when the ‘C Group’ nodes are lumped into one node and the same for the ‘D Group’ node. That means that, in agreement with it being a fractal, when the graph is “viewed from far” it looks the same as level 0 graph. In fact, when looked at from a far enough distance this similarity to the level 0 graph will be the case for all level $i$, $i \geq 1$, graphs.
4 Nominally-typed OOP vs. Structurally-typed OOP

It should be noted that class names information (a.k.a., nominality, and ‘nominal type information’) of nominally-typed OO languages such as Java, C#, C++, and Scala is used in the base step to start defining the subtyping relation between generic types. Structurally-typed OO languages (such as OCaml [19], Moby [16], PolyTOIL [14], and StrongTalk [13]), on the other hand, do not have such a simple base step, since a record type corresponding to a class (with at least one method) in these languages does not have a finite number of supertypes to begin with, given that “superclasses of a class” in the program, when viewed structurally as supertypes of record types, do not form a finite set. Any record type has an infinite set of record subtypes (due to the width-subtyping rule [22]). Accordingly, a record type with a method—i.e., a member having a function type—causes the record type to have an infinite set of supertypes, due to contravariance of the type of the method. Adding-in a depth-subtyping rule makes the subtyping relation between record types with functional member types even more complex.

This motivates suspecting that subtyping in structurally-typed OO language is a dense relation, in which every pair of non-equal types in the relation has a third type, non-equal to either member of the pair, that is “in the middle” between the two elements of the pair, i.e., that is a subtype of the supertype (the upperbound) of the pair and a supertype of the subtype (the lowerbound) of the pair. In fact this may turn out to be simple to prove. Due to a class in generic nominally-typed OO languages having a finite set of superclasses in the subclassing relation, subtyping in generic nominally-typed OO languages is not an (everywhere) dense relation, and the subclassing relation in these languages forms a simple finite basis (the “skeleton”) for constructing the subtyping relation. For structurally-typed OO languages (where record types with functional members are a must, to model structural objects), this basis (the “skeleton”) is infinite and thus the “fractal” structural subtyping graph (if indeed it is a fractal) is not easy to draw or to even imagine.

5 Conclusion

This concludes the main observation connecting subtyping in generic nominally-typed OOP to fractals. More observations and conclusions may be built on top of this observation.

Addendum 1 Benefits and Applications: An obvious benefit of the observation in this essay is to demonstrate one more (unexpected?) place where fractals show up. Yet an additional benefit, and practical application, of the observation may be to apply some of the theory developed for fractals to better the understanding of the subtyping relation in OO languages, possibly leading to providing a better understanding of their generic type
systems and thus developing better OO language compilers.

Addendum 2 Intervals \((I = [S,T], \text{where } S <: T)\): General lower bounds \((S)\) and upperbounds \((T)\) for type arguments (Figure 2.4). Relations on intervals: An interval containing another interval (the ‘contains’ relation: \(S_1 <: S_2 \land T_2 <: T_1\)), and an interval preceding another interval (the ‘precedes’ relation: \(T_1 <: S_2\)).

Addendum 3 Pruning Transformation: Bounds, lower or upper, on a type parameter limit (i.e., decrease) the types of level \(i\) that can substitute the holes (the \(?s\)) when constructing a level \(i + 1\) type so that a substitution respects bounds.

Addendum 4 Demonstration Software: An interactive Mathematica program that demonstrates the subtyping hierarchy for multiple simple class hierarchies up to four nesting levels is available upon request (The program uses the Manipulate function of Mathematica 6, is formatted as a Mathematica 6 demo, and is in the Mathematica .nb format, i.e., the file format Mathematica has used as of 2007.)

Addendum 5 Multi-arity: Generic classes with multiple type parameters simply result in types with multiple “holes” at the same nesting level for the same class.

Addendum 6 Parameterizing classes Object and Null: At least one needs to be non-parameterized, if not both? Otherwise we may have an unbounded infinite ascending chain of supertypes (see Section 4.) (What will then be the meaning of \(?\), and be the default type argument?)

Addendum 7 Equation: As for many fractals, we expect the graph of the subtyping relation to be described by a recursive equation. We anticipate this equation to be (something along the lines of)

\[
G = G_0(copy(G) + flip(G) + flatten(G)),
\]

where \(G_0\) is the initial graph (the “skeleton” of the subtyping fractal, resulting from turning the subclassing relation into a subtyping relation by using \(?\) as the default type argument for generic classes), the application of \(G_0\) means substitution (similar to \(\beta\)-reduction in \(\lambda\)-calculus) of its holes with the argument graph, i.e., \(copy(G) + flip(G) + flatten(G)\) which applies the three above-mentioned transformations, and + means “subtyping-respecting union” of component graphs. (Note: The \(G_0\) in the equation (i.e., the graph of the first iteration of the subtyping relation, which is directly based on the subclassing relation) is what makes (all iterations/approximations of) the graph \(G\) have the same structure “when viewed from far”, i.e., when zooming out of it, as the subclassing relation.)
To construct approximations of $G$ iteratively, the equation can be interpreted to mean

$$G_{i+1} = G_0(copy(G_i) + flip(G_i) + flatten(G_i)),$$

which means when constructing approximations to $G$ we construct elements of the sequence

$$G_0 = G_0,$$
$$G_1 = G_0(copy(G_0) + flip(G_0) + flatten(G_0)),$$
$$G_2 = G_0(copy(G_1) + flip(G_1) + flatten(G_1))$$
$$= G_0(copy(G_0(copy(G_0) + flip(G_0) + flatten(G_0))) +$$
$$flip(G_0(copy(G_0) + flip(G_0) + flatten(G_0))) +$$
$$flatten(G_0(copy(G_0) + flip(G_0) + flatten(G_0))))$$
$$G_3 = G_0(copy(G_2) + flip(G_2) + flatten(G_2)) = ...,\text{ etc.}$$

Another seemingly-equivalent recursive equation for describing the subtyping graph $G$ is

$$G = G_0(copy(G_0) + flip(G_0) + flatten(G_0)),$$

which, even though not in the more familiar $x = f(x)$ format, has the advantage of showing that $G$ (the limit, infinite graph) is equivalent to (isomorphic to) substituting its own holes with transformations of $G_0$, i.e., that the substitution does not affect the final infinite graph $G$ (just as adding 1 to $\omega$, the limit of natural numbers, does not affect its cardinality; $|\omega| = |\omega + 1|$.) It also reflects the zooming-in fact (opposite to the zooming-out fact above) that when zooming-in into $G$ we find (transformed copies of) $G_0$ each time we zoom in, ad infinitum.

**Addendum 8** Equations with Intervals: With intervals, the equation above becomes simpler and more general, where, if $I$ is the function computing all the intervals over a graph, we then have

$$G = G_0(I(G)),$$

or,

$$G = G(I(G_0))$$

or, most accurately,

$$G = G(I(G)).$$

Note that the three equations agree on defining $G_1 = G_0(I(G_0))$. The three equations disagree however on later terms of the construction sequence. They, for example, define

$$G_2 = G_0(I(G_1)), \quad G_2 = G_1(I(G_0)),$$
and $G_2 = G_1(I(G_1))$, respectively. The equivalence of the three equations (i.e., of the resulting graph from each) is unlikely, but a mathematical proof or a convincing intuitive proof of that is needed.)

**Addendum 9** Graph Matrices: Representing successive subtyping graphs as adjacency matrices (0-1 matrices) is useful in computing (paths in graph of) the relation (and in computing containment of intervals)? (Using $(I - A)^{-1}$, with binary addition and multiplication of matrices, to compute the transitive closure of the relation and thus intervals over it.)

**Addendum 10** Subtyping Fractal Drawing Software: A program that takes in a Java program, analyzes its class declarations, and draws its corresponding subtyping fractal, with support for zooming in/out (similar to XaoS) (or, use ‘lefty’ [for Windows]? Use some code from Mathematica demo.

**References**

1. C# language specification, version 3.0. http://msdn.microsoft.com/vsharpc, 2007.

2. ISO/IEC 14882:2011: Programming Languages: C+++. 2011.

3. Nova | hunting the hidden dimension - pbs. http://www.pbs.org/wgbh/nova/physics/hunting-hidden-dimension.html, 2011.

4. Fractal. http://en.wikipedia.org/wiki/Fractal, Dec. 2014.

5. Moez A. AbdelGawad. NOOP: A Mathematical Model of Object-Oriented Programming. PhD thesis, Rice University, 2012.

6. Moez A. AbdelGawad. In nominally-typed object-oriented programming objects are Not merely records and inheritance Is subtyping. Submitted for publication, 2013.

7. Moez A. AbdelGawad. NOOP: A Nominal Mathematical Model Of Object-Oriented Programming. Scholar’s Press, 2013.

8. Moez A. AbdelGawad. An overview of nominal-typing versus structural-typing in object-oriented programming (with code examples). Technical report, arXiv.org:1309.2348 [cs.PL], 2013.

9. Moez A. AbdelGawad. A domain-theoretic model of nominally-typed object-oriented programming. Journal of Electronic Notes in Theoretical Computer Science (ENTCS), DOI: 10.1016/j.entcs.2014.01.002. Also presented at The 6th International Symposium on Domain Theory and Its Applications (ISDT’12), 301:3–19, 2014.
[10] Moez A. AbdelGawad and Robert Cartwright. NOOP: A domain-theoretic model of nominally-typed object-oriented programming. Under consideration for publication in the Cambridge Journal of Mathematical Structures in Computer Science, 2014.

[11] Michael F Barnsley. Fractals Everywhere: New Edition. Courier Dover Publications, 2013.

[12] Richard Bird et al. Introduction to functional programming using Haskell, volume 2. Prentice Hall Europe Hemel Hempstead, UK, 1998.

[13] G. Bracha and D. Griswold. Strongtalk: typechecking Smalltalk in a production environment. In OOPSLA ’93, pages 215–230, 1993.

[14] K. Bruce, A. Schnett, R. van Gent, and A. Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. ACM Transactions on Programming Languages and Systems, 25(2):225–290, 2003.

[15] Robert Cartwright and Moez A. AbdelGawad. Inheritance Is subtyping (extended abstract). In The 25th Nordic Workshop on Programming Theory (NWPT), Tallinn, Estonia, 2013.

[16] K. Fisher and J. Reppy. The design of a class mechanism for Moby. In PLDI, 1999.

[17] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification. Addison-Wesley, 2005.

[18] Douglas R. Hofstadter. Gödel, Escher, Bach: an Eternal Golden Braid. Basic Books, second edition, 1999.

[19] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system. Available at http://caml.inria.fr/.

[20] Benoit B Mandelbrot. Fractals. 1977.

[21] Martin Odersky. The scala language specification, v. 2.7. http://www.scalalang.org, 2009.

[22] Benjamin C. Pierce. Types and Programming Languages. MIT Press, 2002.