Identity Relations in Distributed Computing

Mackenzie Scott

October 2021

1 Abstract

In this paper I examine both the practical and the metaphysical questions of identity for distributed computing in object-oriented languages, including the implications for designing a form of ’seamless’ distributed system. By demonstrating how abstract ideas of identity and access have real implications for object-oriented languages, I show how related problems can be solved at the conceptual level and then applied when designing these ’seamless’ systems. I also consider some of the surface-level impacts of different type paradigms on distribution and how this relates to object identity and equivalence relations, as well as considering the advantages and disadvantages of two basic conceptions. While the main considerations of this paper focus on ideal distribution structures in object-oriented languages, some of the key points will be applicable to cross-language or rudimentary systems.

2 Introduction

In January 2021, I came across the original publication [3] for Java’s Remote Method Interface (*RMI*) library. This was the closest thing I had seen to a truly ’seamless’ distributed system. Inspired by the idea and dismayed by the reality, which had been largely untouched since its 1997 creation, I wanted to create something that would not only simplify the cumbersome boilerplate tasks required to use the *RMI* but also allow even greater control of remote virtual machines by creating a better form of ’seamless’ integration. In this ideal framework, using functions of the seamless integration that interact with a remote machine ought to be indistinguishable from using those on the same machine.[1]

[1]This is not to say that it should be the same as having a local implementation respond to outbound requests to an external API but, crucially, that there should be no need for the programmer

I abandoned my original attempt [1], partly due to my inexperience with the internals of the Java virtual machine, but also because I simply had no idea how to achieve the goals I intended for the project. Although I succeeded in creating a tool that made using the existing *RMI* library easier, my project did not do what I intended it to: to make cross-machine program integration seamless, such that I could interact with a program on another machine as though it was in the same virtual machine. When reviewing my work to see what had gone wrong and, more importantly, why it had gone wrong, I realised that my plan had been flawed from the beginning: I had an idea of what I wanted the end result to look like, but a gap in my understanding of how to design such a system.

As it turned out, the problem was more than a planning oversight. Since I was working outside the regular bounds of the language and the virtual machine it was my responsibility to decide how to handle concepts at a fundamental level: attribution of objects, interactions and equality relations. This paper will deal with the core issues I distilled from my failure at a conceptual level, explaining the practical problems and their conceptual sources. I do not believe these have a trivial solution. Some, such as attribution, I feel are best left to the designer of particular systems to consider, whereas others like identity and typing are artefacts of common language design. The latter may not be problems to be solved but instead dangers to consider.

3 The Identity Problem

While the identity of an object will rarely be worth considering in simple object-oriented programming, since we can often rely on a language's provided equality operator, it poses a more significant problem when designing a seamless distributed system since it is up to the designer to decide how a cross-machine equivalence relation ought to work. The following sections will detail two possible object equivalence relationships and the impact other language design choices could have on them.

Due to the overlap of multiple fields in this topic it will be necessary to borrow terminology and logic symbols, some of which may be unfamiliar. In most cases I have explained the written meaning of any formula or expression used or given an example to make the point clear, but for the sake of clarity some of the more obscure expressions are explained here. The construction $\langle f(j) \rangle_{j \in J}$ is used to express a bijective function detailing a family J in terms of a function $f(j)$, so that J contains all potential values for j.

to interact with any 'request' or transfer layer directly.

$$\forall P \in \mathbb{U} \forall Q \in \mathbb{U}((((\langle j(P)\rangle_{j\in J} = \langle j(Q)\rangle_{j\in J}) \wedge (\forall j \in J[\langle j(P)\rangle_{j\in J}] j(Q) = j(P))) \supset (Q \equiv_{R_1} P))$$

Figure 1: The R_1 equivalence relationship.

$$\forall P \in \mathbb{U} \exists! Q \in \mathbb{U}(Q \equiv_{R_2} P)$$

Figure 2: The R_2 equivalence relationship.

3.1 Equivalence Relations

First, let us consider a very simple object P, which has a single number property n, accessible via $n(P)$, and no additional data. How we define this object's identity will depend upon features of the language it exists in. If P is in a very rudimentary language that is entirely data-oriented with no system of references (so that when it is used, P's entire binary data is there *in situ* rather than accessed via some index reference to a data heap) then it could be argued that since P has nothing other than property data, any object Q with the *same* data is equivalent to P. We will call this data-equivalence relation R_1, detailed in Figure 1 for a universal set of all objects $\mathbb{U}$.

If this is the case, then an object $P_2 \doteq n(P_2) = n(P)$ would be R_1 equivalent to P, and in essence would *be* P. On the other hand, it could be argued that rather than P's identity being defined by its constituent data, that since each use-case of P requires its data be duplicated (on account of our language having no reference system) then in fact no two uses of P are identical to one another. We will call this no-equivalence relation R_2. On the surface, it may seem like no object Q can ever be equivalent to P, but this is not technically correct. If that were the case, $\forall P \in \mathbb{U} \nexists Q \in \mathbb{U}(Q \equiv_{R_2} P)$, which would entail that $P \not\equiv_{R_2} P$ since $P \in \mathbb{U}$. As this conclusion violates the binary equivalence relation, we must instead say $\forall P \in \mathbb{U} \exists! Q \in \mathbb{U}(Q \equiv_{R_2} P)$.[2] In practice, this means there could be any number of objects that are R_1 equivalent to each other but each is R_2 equivalent to only one: itself.

This may seem like an abstract and unimportant consideration, but the distinction is very important when managing copies, duplicates and 'proxy' imitations of an object. Imagine a very simple program distributed across two machines,[3] one of which holds the program's run-time data (we shall call this the *host*) and the other which performs calculations requiring or making alterations to this data (we shall call this the *guest*.)[4] The host's data is aggregated in

[2]This does not violate the binary equivalence relation since there exists no case to break symmetry or transitivity. There are no such objects where an asymmetric or intransitive relation could occur: $\forall x \in \mathbb{U}(\exists! y \in \mathbb{U}(x \equiv_{R_2} y))$.

[3]I am using 'machine' to avoid differentiating between virtual or physical machines as the distinction is unimportant here.

[4]I am deliberately avoiding the common terms 'server' and 'client' to prevent any misunder-

P	$n(P) = 0$	$m(P) = 0$
P_2	$n(P_2) = 10$	$m(P_2) = 0$
P_3	$n(P_3) = 5$	$m(P_3) = 3$

Figure 3: Visualisation of host/guest P-structure.

structured objects, each of which has data-holding properties. If the guest requires use of an object P with number properties n, m, we have a choice: either we give the guest a complete binary copy of P that it can access and manipulate, or alternatively we keep P only on the host machine and respond to the guest's individual requests for access to properties n, m. Given that any request for data made by the guest is likely to incur some overhead from the transfer (even in the event that both host and guest are physically linked) it seems prudent to use the former method: give the guest a data-clone of P, P_2. If we take identity to be based on data-equivalence R_1, then it is the case that $P \equiv_{R_1} P_2$. However, if the guest modifies P_2 during operation then $P \not\equiv_{R_1} P_2$ since their data is no longer identical.

This is not in itself a problem: we could easily create some kind of 'syncing' function to keep the host's P in line with the changes made to P_2, but this has some drawbacks in more complex systems. Imagine a second guest obtains a copy P_3 from the host. If this copy was made before the alterations to P_2 then $P \equiv_{R_1} P_2 \wedge P \equiv_{R_1} P_3$ but after the alterations, P_2 is no longer equivalent to either P or P_3. Suppose the second guest alters P_3 in a different way, so that none of our objects are equivalent. Which copy should the host update P according to?

In Figure 3, the host could update the value of $m(P)$ to $m(P_3)$ since $m(P_2)$ remains unchanged, but the values for the n property have both changed differently, so we have a non-resolvable conflict. The easiest way to prevent this would be to update P as soon as P_2 changes via some data-watching system, and then relay this information to the guest using P_3. However, we are now back in the position we were trying to avoid: each alteration to the data requires us to transfer the data between the host and guests. For this reason, it seems more practical to keep P's data on only the host machine and let guests request data, to avoid 'de-synchronisation' between multiple guests, as each action requires only one transfer of the data between that guest and the host, as opposed to n transfers to synchronise changes across n guests. That said, systems that are designed with only a single guest-host relationship in mind will be able to avoid this conflict problem entirely.

I think it is fair to find the R_1 equivalence relation appealing. The idea that we can pass an object between the host and guest and have it be the same object is a nice idea and, on a surface level, would appear to make seamless

standing caused by preconceived notions. The host is the data-provider and the guest is the data-user.

distribution much simpler. However, the lack of parity between the host's P and the guest's P_2 could be a problem for the R_1 relation. We would like to say that if something is equivalent then it must *always* be equivalent, and R_1 does not allow this for any object that is mutable. This is why, in most cases, the R_2 relation is more practical. We do not need to worry about changes to P_2 since $P_2 \not\equiv_{R_2} P$ even if $\forall j \in J[\langle j(P) \rangle_{j \in J}](j(P) = j(P_2)$. This can be counter-intuitive at first; we would like to think that two trivial objects with identical binary data are themselves identical, since there seems to be no other property they have that distinguishes them. In fact this is not the case, because we are counting the position of their actual binary representation as a property of the object. This means we can say that $P \not\equiv_{R_2} P_2$ since these objects exist on separate machines. In fact, we can say the same even if both are on the same machine, so long as they have separate binary data, allowing there to be a situation where we can change a value of P and not P_2.

At first glance, this might appear to open a new problem for us: how do we deal with the equivalence of primitive numbers? If we are saying that, according to R_2, two objects with matching binary data are not equivalent, then it holds that two numbers (let us say 0 and 0) are not equivalent. This is not a problem for us: the R_2 equivalence relation is not the same as the 'equal to' numerical relation, and as they are separate relations we can define separate rules for them. The reason our two objects are different according to R_2 is that we are not considering the value of the objects but instead their uses. Similarly, these specific 0 and 0 characters in this document are not R_2 equivalent because their character positions are different. The R_2 relation is technically valid on a set of numbers in the abstract: $\forall x \in \mathbb{N} \exists! y \in \mathbb{N}(x \equiv_{R_2} y)$. Both R_1 and R_2 appear have specific uses. R_1 is attractive in that it allows us to compare multiple uses of objects, whereas R_2 does not lead to difficult problems of mutating objects. In fact, it is not uncommon to see a form of both relations in programming languages. Java supports the R_2 relation through the reference equality operator, but allows a value-based R_1 relation through the object *equals* method, though this is left to user implementation.[2]

3.2 Transitive Information

It may seem as though our identity problem is solved. We can pass some sort of reference from the host to a guest and the guest can use this, in the same way as a Java program works with references to an object on the stack rather than cloning its binary data for each use. However, this is not as simple as it may sound. So far we have been considering a trivial conception of 'objects' that are simply collections of indexed binary data properties. In all but a handful of languages, objects will be significantly more than this.

Figure 4 displays a very simple bytecode operation to, provided with an 'Example' object, read the n field and, if greater than 0, set its value to 2. The

```
aload 1
getfield com/example/Example.n : I
ifle L0
aload 1
iconst_2
putfield com/example/Example.n : I
L0
```

Figure 4: A trivial bytecode program.

P		
$a(P)$	Integer	
$b(P)$	Q	
	$n(Q)$	Integer
	$m(Q)$	Integer

Figure 5: An example object layout.

first thing of importance is that the machine requires more than the object's data; it requires a 'schema' of the object. We are not just asking for the n property, we are asking for a specific n property shared by all `com.example.Example` objects. If a guest machine is to access data from an object it must know what to ask for. In most cases, the guest will have some kind of provided template for the object (e.g. its class for Java), but this problem extends further than just the requested object. Suppose we have an object P that has the properties a, b. a might be a simple number, but b could be another type of object (or another of the same type of object.) Let us suppose, for simplicity, that it is the structure in Figure 5.

Not only does the guest machine need to know the structure of P but also of Q. This propagates for every new object type that our object can hold. Fortunately, a structure like this is resolved in the sense that we can tell the guest exactly what P contains and where to access it. In fact, in a compiled language we could even provide the schema ahead of time so that the guest will know what it is looking to access before it has even made a connection to the host machine. So far we have a solution for simple objects, but even in languages with strict static typing we have another issue to deal with. Consider that our example objects are from a language which has some function T to determine

C_1	C_2
a	a
b	b
	c

Figure 6: An example type inheritance.

the type of an object. Our language's types are strict, so all objects of a type will have the same properties. Given this, we know that in Figure 5 it is the case that $T(P) \neq T(Q)$ since P has one integer and one Q-type property, and Q has two integer properties. Our language could have some form of inheritance, as many object-oriented languages do, so that any type can have a number of sub-types. Sub-types must inherit all of the properties of their super-type, but are not restricted to only these properties. As an example, for types C_1 and C_2, for C_2 to be a sub-type of C_1 it must be the case that $C_1 \subseteq C_2$. This may seem counter-intuitive at first but the sub-type relation is in fact an inverse of the subset relation. This is because the sub-type C_2 must encompass *all* of C_1's properties, but is not limited to these. In practice, this would allow for the structure from Figure 6 where the sub-type C_2 has an additional property that its super-type does not.

Given this, an object could have a property of type C_1 but knowing this is not enough to know that its value is only of type C_1. Here we introduce an asymmetry: $\forall P \in \mathbb{U}(T(P) = C_2) \supset (T(P) = C_1)$ but $C_2 \supseteq C_1$. C_1 is a subset of C_2 but the set of all objects of type C_2 is a subset of all objects of type C_1.[5] With this kind of type inheritance, we can imagine a situation where the guest knows to expect an object Q of type C_1, but does not know whether Q is strictly a C_1 or whether it could in fact be a C_2 with additional properties. There is no immediate problem with this situation, though one will arise later. I will demonstrate this through the following example.

Imagine you are staffing a company and hire an accountant. Unbeknownst to you, your new hire is not only an accountant but also a competent chess player. In fact, she has recently won a regional tournament. On the surface, these other properties are irrelevant to her job as an accountant. She does what is expected of an accountant and from your limited interaction with her you would never know about her chess accomplishments. In this example, being an accountant is C_1 and being an accountant who also plays chess competently is C_2. Your hire happens to be C_2 but is also a C_1. Now so far this is fine for our distributed system. In fact, it may seem to help: we do not need to tell a guest machine about type C_2 if all of the functions it needs are covered by C_1. However, this may cause problems for the guest. Imagine our accountant had formatted her documents in an unusual way as a result of her chess-oriented mind. While still performing her job as expected, there would be a procedural discrepancy that we can't explain given our limited knowledge.

In cases like these, where an object has some functional members attached, the guest may need to know some information about the object that cannot be provided ahead of time. This introduces the idea of transitive information: additional details needing to be exported by the host with a complex object. We can imagine a tiered object whose properties contain objects that, in turn,

[5]Note that neither of these are proper subsets; it could be the case that C_2 has no additional properties, or that there are no objects of C_1 that are not also objects of C_2.

contain more nested objects. In such a case the guest will not only need information about the object being provided but also its children, each of which will require, transitively, the information of their own children. In a language like Java where objects have functional members which can accept more objects as parameters, the guest incurs even more transitive dependency as it needs to know about all of these available functions and their argument types. As you can see, this relationship propagates very easily and can become very complex.

3.3 Split Attribution

Now that we have an understanding of the complexity of both object structure and object ownership, we can examine how these two areas overlap to cause a problem I will call split attribution. In simple terms, this occurs when parts of a complex, nested object are owned by different machines, so that the attribution of the object is 'split' between the host and the guest. This seems at first like a situation we ought to avoid at any cost, for the reasons detailed in **3.1**, however there may be cases where this is the best of bad options.

Let us consider the model from Figure 5, where the object P has an object property $b(P) = Q$, and Q has only primitive properties. If our guest asks for P then both P and Q are attributed to the host. No matter whether we are using a model where the host keeps the objects and the guest must make requests to alter or read individual properties or whether we are giving the raw data to the guest, at this moment P and Q have both been created by (and come from) the host. Now let us imagine the guest makes a change: it sets $b(P)$ to a new object value O, replacing Q. Updating the host's copy is trivial: we can marshal the data and transmit it to the host as we would when changing a primitive value. Deciding which side to attribute the newly-created O to is not trivial. Ideally, we would like the host to have the 'master' copy of O and have the guest request it and work with it indirectly, however this may be impossible. If the guest is already using O in some capacity, perhaps using references to it in multiple places, then we cannot simply destroy the guest's copy of O and give it to the host machine. Equally, we do not want to have two separate copies of O, (O_1 and O_2) in play. According to our R_2 equivalence relation $O_1 \not\equiv_{R_2} O_2$, and the R_1 relation has the aforementioned problem that it holds only until one copy is altered.

This brings us to the third option, where we attribute P to the host and the replacement O to the guest. Although this seemed repugnant, it appears to be the best of the given options. The guest acts as a sort of pseudo-host for O, allowing the host to request it. At first this looks like it causes the problem we saw earlier: how will the host use the child O if it is attributed to the guest? The short answer is that the host will not. For all intents and purposes it is a pseudo-guest when accessing this particular object. We are attributing O to the guest on the grounds that the guest had it first, and it is much more difficult to untangle O and marshal it than to have the host make requests. This inverse relationship is not affected in any way by how deeply the object is nested in other objects or by any other means, since it can be treated as completely independent given that there is no function to obtain the parent object from a child. In fact for a language that uses references in structures the object is not actually contained within the parent P in any meaningful way as P simply holds a reference to it.

3.4 Descriptive and Prescriptive Typing

This section of the paper is largely theoretical and will consider language design choices as opposed to system ones, though the conclusion of these choices impacts practical distributed system design. I will explain the distinction between descriptive and prescriptive types, some advantages and disadvantages of each system, and then go into an explanation of how it can affect (or be used by) distributed programs.

Most strongly-typed languages that I am familiar with use prescriptive typing. The Java language specification is a very clear example of this: if the object P is of type C_1 then it *must* implement all of the members defined by C_1. If we can obtain the set of members defined by a type with a function M then for prescriptive typing it is the case that $\forall P \in \mathbb{U}(T(P) = C_1 \supset (M(C_1) \subseteq J[\langle j(P) \rangle_{j \in J}]))$. On the other hand, descriptive typing is such that $\forall P \in \mathbb{U}(T(P) = C_1 \equiv (M(C_1) \subseteq J[\langle j(P) \rangle_{j \in J}]))$. The crucial difference is in the implication for the prescriptive, whereas the *if-and-only-if* material equivalence relationship that a descriptive language has. While only a minor difference in the notation this is a large change in the language's function. For Java, having prescriptive types, any object of class 'Example' necessarily has implementations for all of the methods belonging to 'Example' and the dynamic fields declared by this class (and its super-classes.)[6] For something descriptive, such as Go's interface design, an object is of the type 'Example' as long as it has all of the functions described by 'Example'.

This distinction can be a little awkward at first, since the two have some overlap. If I show you a baked sponge with icing and candles and tell you that it is a cake, you do not know whether I am using the term prescriptively or descriptively. It is not enough to know what the thing is, as we also need to know why it is a cake. I could tell you that it has candles, icing and sponge and is therefore a cake by virtue of having these things, which would mean I was using 'cake' in a descriptive capacity, or I could tell you that it is a cake and, because it is a cake, it is necessary that it has icing, sponge and candles. The difference is that in this prescriptive view the candles, icing and sponge are a

[6]Strictly speaking this is not the case; no virtual machine implementation actually checks for method implementations, though the Java compiler and Java language specification would have it be so.

necessary property of it already being a cake, rather than the things that make it a cake. If we take a function $p(x)$ to be 'x is a cake' and $q(x)$ to be 'x has candles' then prescriptive typing would make it the case that $p(x) \supset q(x)$. This means it is possible to have something y with candles ($q(y)$) but y is not necessarily a cake. It *could* be the case that $p(y)$ but equally y could be a candle-holder with candles. In this way, the type of the thing seems to be some additional property it has. However, if we used descriptive typing then $p(x) \equiv q(x)$. In this case, that x has candles makes it *necessarily* a cake, even if x is also a candle-holder or something else.

Applying this to programming, we can see that both forms of typing have their advantages and drawbacks. Prescriptive typing seems to be the more appealing of the two, indicated by its prevalence in object-oriented language. It allows us to have rigid definitions of things and prevent unintended overlaps, such as when an object P happens to fall into a type we didn't intend by virtue of it having matching properties. Java and similar languages use prescriptive typing. An object P has a defined class, a reference to which is stored before its binary data as a kind of unspoken and immutable property.[7] This allows us to have a case where P is of the class 'Foo', which has a single method, but P is not of the class 'Bar' despite 'Bar' having an identical method. If we return to our prescriptive definition we can see why: P is a 'Foo', which necessarily entails it has the method, but having an identical method does not make it a 'Foo'. This is a separate and intrinsic property. On the other hand, prescriptive typing has its downsides. There may be occasions, particularly in distributed programs where we have overlap of two separate yet interacting parts of the program, where we might want to treat an object as a 'Foo' but cannot, since it is of some other type. If the guest machine obtains an object P but does not have access to P's type, then it cannot create another object of the same type. It could create an object Q with identical properties and functions, but $T(P) \neq T(Q)$.

This might make descriptive typing seem appealing for our distributed system. If Q has matching properties to P then we can say that, descriptively, $T(P) = T(Q)$. We can treat Q as though it were of the same type as P. This distinction may remind you of the R_1 and R_2 equivalence relations discussed in **3.1**; one bases equality purely on content, whereas the other relies on some intrinsic quality that is not immediately visible in the raw data. Like the R_1 relation, descriptive typing encounters some of the same identity problems. We might define a type B as something that has two wheels. Objects of the type M (having two wheels and a motor) would also be of the type B since they have all of the necessary properties: two wheels. This is a minor error but it seems trivial so far. However, objects of type C, having four wheels and a motor, would also be of the type B since having four wheels implies that they have two. Under this relation, $C \subseteq M \subseteq B$. This seems problematic. We do not want to imply that all cars are bicycles, so we have to narrow our definition

[7] It is technically possible to alter this 'klass pointer' although this is incredibly difficult and dangerous.

of B, perhaps to say having *only* two wheels and no motor. This would, however, include two-wheeled carts or wheelbarrows. This leads us to creating increasingly-narrow definitions to exclude unwanted objects from falling into the wrong categories. This problem is solvable in a closed system where the designer is aware of all possible types and objects and so can make sure there are no unwanted overlaps, however when we are working with distributed systems it is possible that we could encounter unintended objects or types provided by a remote machine that we cannot predict and account for.

4 Conclusion

The problems raised in this paper do not have a simple solution. They will be something to consider when designing a distributed system and to tailor to a particular use-case. Considering the equivalence relation that a designer wants objects to have is important since it may not be possible to rely on a language's built-in equality comparison. The purpose of seamless distribution is to make sure a program's operation when spread across multiple machines is the same as when it runs within one machine, so whatever equivalence relation the designer uses must be uniform in both environments. I have offered a demonstration of why a relation like R_2 may be more appropriate for this given that it reduces parity issues, despite the initial appeal of the R_1 relation. Similarly, I have argued that prescriptive typing is simpler to implement despite incurring transitive dependency of type information and the attached problems. Transitive dependency is a difficult problem to avoid in seamless distribution since we must make sure a guest machine can function as though it were part of the host, and that requires us to give the guest all of the necessary information. When dealing with the attribution of objects, I have argued that it is more practical to use a two-way host-guest relationship and keep objects attributed to their creating machines rather than to try and untangle them and give ownership to the host, on the grounds that we cannot, without full knowledge of a program, know whether our transfer of the object to the host in a reference-using (R_2 language could create a separate "shadow" copy on the guest that falls out of sync when altered and causes a malfunction. The common element between all of these conclusions is that the most reliable option seems to be the best. The purpose of seamless distribution is to allow the creation of programs that can be distributed across multiple machines without any functional impact, by hiding all of the distributing work behind the system curtain. If the system is unreliable then so is any program using it.

My conclusions are by no means absolute. There will be individual cases where I might choose to use an R_1 equivalence relation in my design or where I think descriptive typing is simply a more effective model, but when considering the abstract it is generally correct to try and appeal to the greatest number of cases. That said, over-reliance on paradigms is often problematic in edge cases where deviating from them might be the better option. My main intention has been to highlight and explain the main problems to consider when designing seamless systems and how these problems have a significant overlap with areas of metaphysics, particularly concerning identity. My recommendations and solutions have been a secondary focus and conclusion, although the reasoning behind them contributes to my explanation of the problem and, I believe, helps to ground them in a more realistic and easily-understandable context. That said, this paper has discussed only a shallow section of the abstract concepts involved in distributed computing and a much deeper discussion should be had before applying any of this in practice.

References

[1] Mackenzie Scott. *Cobweb*. Version 1.0.0. Jan. 2021. URL: `https://github.com/Moderocky/Cobweb`.

[2] *The Java Language Specification*. Sept. 2021. URL: `https://docs.oracle.com/javase/specs/jls/se17/html/jls-15.html`.

[3] Ann Wollrath, Jim Waldo, and Roger Riggs. "Java-Centric Distributed Computing". In: *IEEE Micro* 17.3 (May 1997), pp. 44–53. ISSN: 0272-1732. DOI: `10.1109/40.591654`. URL: `https://doi.org/10.1109/40.591654`.