Computer Science 15-312
Inheritance and Subtyping in Java

Robert Harper

Spring, 2000

1 Introduction

In this note we discuss the closely-related, but conceptually distinct, notions of *inheritance* and *subtyping* as exemplified in the Java language. Inheritance is a mechanism for supporting *code re-use* through incremental extension and modification. Subtyping is a mechanism for expressing *behavioral relationships* between types that license the provision of a subtype value in contexts where a supertype value is required. While conceptually distinct, these two concepts are frequently conflated, especially in languages such as Java in which inheritance relationships give rise to subtyping relationships. Even in Java not every subtype relationship arises via inheritance, and in some languages (including some extensions to Java), subclasses need not give rise to subtypes.

2 Inheritance Mechanisms in Java

2.1 Classes and Instances

The fundamental unit of inheritance in Java is the *class*. A class consists of a collection of *fields* and a collection of *methods*. Fields are assignable variables; methods are procedures acting on these variables. Fields and methods can be either *static* (per-class) or *dynamic* (per-instance).[1] Static fields are per-class data. Static methods are just ordinary functions acting on static fields.

Classes give rise to *instances*, or *objects*, that consist of the dynamic methods of the class together with fresh copies (or instances) of its dynamic

[1] Fields and methods are assumed dynamic unless explicitly declared to be static.

fields. Instances of classes are created by a *constructor*, whose role is to allocate and initialize fresh copies of the dynamic fields (which are also known as *instance variables*). Constructors have the same name as their class, and are invoked by writing `new` $C(e_1, \ldots, e_n)$, where C is a class and $e_1, \ldots, e_n$ are arguments to the constructor.[2] Static methods have access only to the static fields (and methods) of its class; dynamic methods have access to both the static and dynamic fields and methods of the class.

The components of a class have a designated *visibility* attribute, either `public`, `private`, or *protected*. We'll have more to say about *protected* below. The public components are those that are accessible by all clients of the class. Public static components are accessible to any client with access to the class. Public dynamic components are visible to any client of any instance of the class. The components of a class also have a *finality* attribute. Final fields are not assignable — they are read-only attributes of the class or instance.[3] Final methods are of interest in connection with inheritance, to which we'll return below.

The components of a class have *types*. The type of a field is the type of its binding as a (possibly assignable) variable. The type of a method specifies the types of its arguments (if any) and the type of its results. The type of a constructor specifies the types of its arguments (if any); its "result type" is the instance type of the class itself, and may not be specified explicitly. (We will say more about the type structure of Java below.)

The public static fields and methods of a class C are accessed using "dot notation". If f is a static field of C, a client may refer to it by writing $C.f$. Similarly, if m is a static method of C, a client may invoke it by writing $C.m(e_1, \ldots, e_n)$, where $e_1, \ldots, e_n$ are the argument expressions of the method. The expected type checking rules govern access to fields and invocations of methods.

The public dynamic fields and methods of an instance c of a class C are similarly accessed using "dot notation", *albeit* from the instance, rather than the class. That is, if f is a public dynamic field of C, then $c.f$ refers to the f field of the instance c. Since distinct instances have distinct fields, there is no essential connection between $c.f$ and $c'.f$ when c and c' are distinct instances of class C. If m is a public dynamic method of C, then $c.m(e_1, \ldots, e_n)$ invokes the method m of the instance c with the specified arguments. This is sometimes called *sending a message* m *to instance* c *with*

[2] Classes can have multiple constructors that are distinguished by overloading. We will not discuss overloading in this note.

[3] More precisely, final dynamic fields can be assigned once, by a constructor of the class, to initialize their values.

arguments $e_1, \ldots, e_n$.

Within a dynamic method one may refer to the dynamic fields and methods of the class via the pseudo-variable `this`, which is bound to the instance itself. The methods of an instance may call one another (or themselves) by sending a message to `this`. Although Java defines conventions whereby explicit reference to `this` may be omitted, it is useful to eschew these conveniences and always use `this` to refer to the components of an instance from within code for that instance. We may think of `this` as an implicit argument to all methods that allows the method to access to object itself.

2.2 Subclasses

A class may be defined by *inheriting* the visible fields and methods of another class. The new class is said to be a *subclass* of the old class, the *superclass*. Consequently, inheritance is sometimes known as *subclassing*. Java supports *single inheritance* — every class has at most one superclass. That is, one can only inherit from a single class; one cannot combine two classes by inheritance to form a third. In Java the subclass is said to `extend` the superclass.

There are two forms of inheritance available in Java:

1. *Enrichment.* The subclass enriches the superclass by providing additional fields and methods not present in the superclass.

2. *Overriding.* The subclass may re-define a method in the superclass by giving it a new implementation in the subclass.

Enrichment is a relatively innocuous aspect of inheritance. The true power of inheritance lies in the ability to override methods.

Overriding, which is also known as *method specialization*, is used to "specialize" the implementation of a superclass method to suit the needs of the subclass. This is particularly important when the other methods of the class invoke the overridden method by sending a message to `this`. If a method m is overridden in a subclass D of a class C, then all methods of D that invoke m via `this` will refer to the *new* version of m defined by the override. The "old" version can still be accessed explicitly from the subclass by referring to `super.`m. The keyword `super`[4] is a pseudo-variable that is used to refer to the overridden methods directly.

Inheritance can be controlled using visibility constraints. A sub-class D of a class C automatically inherits the private fields and methods of C

[4]It might have been nice to use "`that`" instead of `super`.

without the possibility of overriding, or otherwise accessing, them. The public fields and methods of the superclass are accessible to the subclass without restriction, and retain their `public` attribute in the subclass, unless overridden. A *protected* component is "semi-private" — accessible to the subclass, but not otherwise publically visible.[5]

Inheritance can also be limited using finality constraints. If a method is declared *final*, it may not be overridden in any subclass — it must be inherited as-is, without further modification. However, if a final method invokes, via `this`, a non-final method, then the behavior of the final method can still be changed by the sub-class by overriding the non-final method. By declaring an entire class to be final, no class can inherit from it. This serves to ensure that any instance of this class invokes the code from this class, and not from any subclass of it.

Instantiation of a subclass of a class proceeds in three phases:

1. The instance variables of the subclass, which include those of the superclass, are allocated.

2. The constructor of the superclass is invoked to initialize the superclass's instance variables.

3. The constructor of the subclass is invoked to initialize the subclass's instance variables.

The superclass constructor can be explicitly invoked by a subclass constructor by writing `super(`$e_1, \ldots, e_n$`)`, but *only* as the very first statement of the subclass's constructor. This ensures proper initialization order, and avoids certain anomalies and insecurities that arise if this restriction is relaxed.

2.3 Abstract Classes and Interfaces

An *abstract class* is a class in which one or more methods are declared, but left unimplemented (ie, left "abstract"). Abstract methods may be invoked by the other methods of an abstract class by sending a message to `this`, but since their implementation is not provided, abstract classes do not themselves have instances. Instead the obligation is imposed on a subclass of the abstract class to provide implementations of the abstract methods to obtain a *concrete* class, which does have instances. Abstract classes are useful for setting up "code templates" that are instantiated by inheritance.

[5] Actually, Java assigns *protected* components "package scope", but since we are not discussing packages here, we will ignore this issue.

The abstract class becomes the locus of code sharing for all concretions of that class, which inherit the shared code and provide the missing non-shared code.

Taking this idea to the extreme, an *interface* is a *fully abstract* class in which

- All fields are `public static final`, *i.e.* are constants.
- All methods are `abstract public`, *i.e.* must be implemented.

Interfaces are a special form of abstract class, and hence have no instances.

The utility of interfaces stems from their role in `implements` declarations. As we mentioned above, a class may be declared to extend a *single* class to inherit from it.[6] A class may also be declared to `implement` *one or more* interfaces, meaning that the class provides the public methods of the interface, with their specified types. Since interfaces are special kinds of classes, Java is sometimes said to provide *multiple inheritance of interfaces*, but only *single inheritance of implementation*. For similar reasons an interface may be declared to extend multiple interfaces, provided that the result types of their common methods coincide.

The purpose of declaring an interface for a class is to support writing generic code that works with *any* instance providing the methods specified in the interface, *without* requiring that instance to arise from any particular position in the inheritance hierarchy. For example, we may have two unrelated classes in the class hierarchy providing a method m. If both classes are declared to implement an interface that mentions m, then code programmed against this interface will work for an instance of *either* class.

The literature on Java emphasizes that interfaces are *descriptive* of behavior (to the extend that types alone allow), whereas classes are *prescriptive* of implementation. While this is surely a noble purpose, it is curious that interfaces are *classes* in Java, rather than *types*. In particular interfaces are unable to specify the public fields of an instance by simply stating their types, which would be natural were interfaces a form of type. Instead fields in interfaces are forced to be constants (public, static, final), precluding their use for describing the public instance variables of an object.

[6] Classes that do not specify a superclass implicitly extend the class `Object` of all objects.

3 Subtyping in Java

As we mentioned in the introduction subtyping is closely related to inheritance, but is conceptually distinct from it. In particular there are languages with subtyping, but no classes, and languages with subclassing, but with no connection between subtyping and inheritance.

What is subtyping? In general terms a subtype relation is a preorder (reflexive, transitive binary relation) σ `<:` τ between types validating the *subsumption principle*:

> If σ `<:` τ, then anywhere a value of type τ is required, a value of type σ may be provided.

In other words the type σ subsumes the type τ in the sense that possessing a value of type σ is at least as good as possessing a value of type τ for all practical purposes.

The intuition underlying the subsumption principle is that values of the subtype should "behave the same" as values of the supertype as far as supertype contexts are concerned. That is, if we have any situation in which are using a value of the supertype by applying to it the operations associated with that type, then it is valid to provide a value of the subtype because *as far as the supertype operations are concerned* the supplied value can "pass" as a supertype value. While this intuition is somewhat vague at this stage, it is a useful guide for understanding the relationship between inheritance and subtyping.

A familiar example of subtyping is to regard the type `int` to be a subtype of the type `float`, based on the mathematical convention of regarding $\mathbb{Z}$ to be a subset of $\mathbb{R}$.[7] The idea is that whenever a value of type `float` is required, it is enough to provide a value of type `int`, because we can *coerce* the integer into a floating point number without changing its "behavior". Informally, this means that the integer, regarded as a floating point number, has the expected algebraic properties. While this may be true in mathematics, it is suspect in programming contexts because the coercion from integers to floating point numbers is, in practice, "lossy" (for large magnitudes).

Subtyping systems can be divided into two general categories:

- *Nominal.* A type σ is a subtype of a type σ *only if* they are explicitly declared to be so. In a nominal (or "by name") subtyping system

[7]NB: Java does *not* treat `int` to be a subtype of `float`, nor is it true in the usual set-theoretic encodings that $\mathbb{Z}$ is a subset of $\mathbb{R}$! Nevertheless it is common to think of integers as certain real numbers, in both mathematical and programming situations.

there are no accidental subtype relations, only those that are explicitly declared to hold.

- *Structural.* A type σ is a subtype of τ only if the behavior expected of a value of type τ is provided by a value of type σ. In a structural subtyping system the question of whether one type is a subtype of another is purely a matter of the properties of the types themselves, and not tied to any declaration form.

The Java subtype system is almost exclusively nominal, with the exception that arrays are treated structurally. The ML signature calculus provides purely structural subtyping through the notion of signature matching — whether one signature matches another is purely a matter of the types and values that they present, and not a matter of declaration.

It seems to me that there is no clear advantage of one form of subtyping over another.

- Nominal subtyping avoids unintended, accidental subtype relationships that arise from coincidental relationships between types. The programmer is forced to explicitly state semantic intentions by declaring intended subtyping relationships. A weakness of nominal subtyping is that it does not support code evolution very well — the *raison d'etre* of object-oriented languages! In a nominal system the programmer can only declare subtype relations for the types already present in the system — if new types are added later, these declarations must be revisited and modified.

- Structural subtyping systems do not suffer from the evolution problem, precisely because the subtype relation is a matter of semantics, rather than declaration. However, it is difficult to avoid unintended, accidental subtype relationships that are not intended by the programmer. This is particularly a problematic in weak type systems (such as those in ML and Java) that cannot specify execution properties of operations such as the requirement that a boolean-valued binary function is a pre-order.

In a sense this distinction between nominal and structural subtyping is a straw man. Although current languages stress one global policy over another, more recent work stresses the need to provide *both* forms of subtyping under program control. This is easily achieved through the notion of *bounded type quantification* within a structural subtype system — the programmer can introduce a "new" abstract type with a specified subtyping relation to other types. Since the abstract type is new, it possesses only those subtyping properties that it is explicitly declared to possess and no other. This is just another example of data abstraction — we make *pro forma* distinctions between types to avoid dependence on accidental coincides.

3.1 Types in Java

The Java type system consists of the following types:

1. *Base types*, including `int`, `float`, `void`, and `boolean`.
2. *Class types* C, which classify the instances of a class C.
3. *Array types* of the form τ`[ ]`, where τ is a type, representing mutable arrays of values of type τ.

The basic types behave essentially as one would expect, based on previous experience with languages such as C and C++. Unlike C or C++ Java has true array types, with operations for creating and initializing an array and for accessing and assigning elements of an array. All array operations are safe in the sense that any attempt to exceed the bounds of the array results in a checked error at run-time.

Every class, whether abstract or concrete, including interfaces, has associated with it the type of its instances, called (oddly enough) the *instance type* of the class. Java blurs the distinction between the class as a program structure and the instance type determined by the class — class names serve not only to identify the class but also the instance type of that class. It may seem odd that abstract classes, and interfaces, all define instance types, even though they don't have instances. However, as will become clear below, even abstract classes have instances, indirectly through their concrete subclasses. Similarly, interfaces may be thought of as possessing instances, namely the instances of concrete classes that implement that interface.

3.2 Subtyping and Subsumption

To define the Java subtype relation we need two auxiliary relations. The *subclass* relation, $C \lhd C'$, is the reflexive and transitive closure of the *extends* relation among classes, which holds precisely when one class is declared to extend another. In other words, $C \lhd C'$ iff C either coincides with C', inherits directly from C', or inherits from a subclass of C'. Since interfaces are classes, the subclass relation also applies to interfaces, but note that multiple inheritance of interfaces means that an interface can be a subinterface (subclass) of more than one interface. The *implementation* relation, $C \blacktriangleleft I$, is defined to hold exactly when a class C is declared to implement an interface that inherits from I.

The Java subtype relation is inductively defined by the following rules. Subtyping is reflexive and transitive:

$$\frac{}{\tau \mathrel{\texttt{<:}} \tau} \qquad \frac{\tau \mathrel{\texttt{<:}} \tau' \quad \tau' \mathrel{\texttt{<:}} \tau''}{\tau \mathrel{\texttt{<:}} \tau''}$$

Arrays are *covariant* type constructors, in the sense of this rule:

$$\frac{\tau \mathrel{\texttt{<:}} \tau'}{\tau\texttt{[ ]} \mathrel{\texttt{<:}} \tau'\texttt{[ ]}}$$

Inheritance implies subtyping:

$$\frac{C \lhd C'}{C \mathrel{\texttt{<:}} C'}$$

Implementation implies subtyping:

$$\frac{C \blacktriangleleft I}{C \mathrel{\texttt{<:}} I}$$

Every class is a subclass of the distinguished "root" class `Object`:

$$\frac{}{\tau \mathrel{\texttt{<:}} \texttt{Object}}$$

The array subtyping rule is *structural* — one need not explicitly declare subtyping relationships between array types for them to hold. On the other hand, the inheritance and implementation rules of subtyping are *nominal* — they hold precisely when they are declared to hold (or are implied by further subtyping relations).

The subsumption principle tells us that if e is an expression of type τ and $\tau \mathrel{\texttt{<:}} \tau'$, then e is also an expression of type τ'. In particular, if a method is declared with a parameter of type τ, then it makes sense to provide an argument of any type τ' such that $\tau' \mathrel{\texttt{<:}} \tau$. Similarly, if a constructor takes a parameter of a type, then it is legitimate to provide an argument of a subtype of that type. Finally, if a method is declared to return a value of type τ, then it is legitimate to return a value of any subtype of τ.

This brings up an awkward issue in the Java type system. What should be the type of a conditional expression $e\,?\,e_1 : e_2$? Clearly e should have type `boolean`, and e_1 and e_2 should have the same type, since we cannot in general predict the outcome of the condition e. In the presence of subtyping, this amounts to the requirement that the types of e_1 and e_2 have an *upper bound* in the subtype ordering. To avoid assigning an excessively weak type, and to ensure that there is a unique choice of type for the conditional, it would make sense to assign the conditional the *least upper bound* of the types of e_1 and e_2. Unfortunately, two types need not have a least upper bound! For example, if an interface I extends incomparable interfaces K and L, and J extends both K and L, then I and J do not have a least upper bound — both K and L are upper bounds of both, but neither is smaller than the other. To deal with this Java imposes the rather *ad hoc* requirement that either the type of e_1 be a subtype of the type of e_2, or *vice versa*, to avoid the difficulty.

A more serious difficulty with the Java type system is that the array subtyping rule, which states that the array type constructor is *covariant* in the type of the array elements, violates the subsumption principle. To understand why, recall that we can do one of two things with an array: retrieve an element, or assign to an element. If τ `<:` τ' and A is an array of type τ`[]`, then retrieving an element of A yields a value of type τ, which is by hypothesis an element of type τ'. So we are OK with respect to retrieval. Now consider array assignment. Suppose once again that τ `<:` τ' and that A is an array of type τ`[]`. Then A is also an array of type τ'`[]`, according to the Java rule for array subtyping. This means we can assign a value x of type τ' to an element of A. But this violates the assumption that A is an array of type τ`[]` — one of its elements is of type τ'.

With no other provisions the language would not be type safe — we could easily construct a counterexample which "goes wrong" by exploiting this hole in the type system. Fortunately this is not possible in Java, because the language is defined so that it must *explicitly check* at run-time for any attempt to exploit this insecurity. More precisely, *every single array assignment* incurs a *run-time type check* to ensure that the "true" type of the assigned value matches the "true" type of the array.[8] This rules out the scenario described above, in the case that the most precise type of x is τ' and the most precise type of A is τ`[]`.

[8]Since `int` and `float` are not class types and are subtypes only of themselves, this check can be suppressed in that case.

3.3 Casting

This brings up the question of how run-time type checks are performed. Java supports a safe form of *casting*, or *change of type*. A cast is written $(\tau)\,e$. The expression e is called the *subject* of the cast, and the type τ is the *target type* of the cast. The type of the cast is τ, provided that the cast makes sense, and its value is that of e. In general we cannot determine whether the cast makes sense until execution time, when the "true" type of the expression becomes apparent and can therefore be compared with the target type. For example, every instance in Java has type `Object`, but its true type will usually be some class further down the type hierarchy. Therefore a cast applied to an expression of type `Object` cannot be validated until the true type of the expression is know.

What is the "true" type of an instance? It is (the instance type of) the class whose constructor created it. When the instance is created its true type is associated with it so that it can be recovered from the instance itself at run time. This is necessary because the static type of an expression is, in general, "attenuated" by the use of subsumption, and hence does not accurately track the true type of the instance. For example, the element type of a general type of lists might be taken to be `Object` so that the list can be used with elements of any class whatsoever. However, when we retrieve an element from such a list, we know statically only that it has type `Object`, and not its true type. We can recover this information at run time by casting the value with its true type as target, necessitating a run-time check. For this reason the true type is sometimes called the *dynamic type* of the instance. Note that it is up to the programmer to maintain a sufficiently strong invariant to ensure that such casts do not fail! The language provides no assistance with maintaining such invariants, which are surprisingly frequent when so-called container classes (such as lists and trees) are used.

Since the static type is an attenuated version of the dynamic type of an object, we can classify casts into three varieties:

1. *Up casts*, in which the static type of the expression is a subtype of the target type of the cast. The type checker accepts the cast, and no run-time check is required.

2. *Down casts*, in which the static type of the expression is a *supertype* of the target type. The true type may or may not be a subtype of the target, so a run-time check is required.

3. *Stupid casts*, in which the static type of the expression rules out the possibility of its dynamic type matching the target of the cast. The cast is rejected.

Similar checks are performed to ensure that array assignments are safe.

4 Methodology

With this in hand we can (briefly) discuss the methodology of inheritance in object-oriented languages. As we just noted, in Java subclassing entails subtyping — the instance type of a subclass is a subtype of the instance type of the superclass. It is important to recognize that this is a methodological commitment to certain uses of inheritance.

Recall that a subtype relationship is intended to express a form of behavioral equivalence. This is expressed by the subsumption principle, which states that subtype values may be provided whenever a supertype value is required. In terms of a class hierarchy this means that a value of the subclass can be provided whenever a value of the superclass is required. For this to make good sense the values of the subclass should "behave properly" in superclass contexts — they should not be distinguishable from them.

But this isn't necessarily so! Since inheritance admits overriding of methods, we can make almost arbitrary[9] changes to the behavior of the superclass when defining the subclass. For example, we can turn a stack-like object into a queue-like object (replacing a LIFO discipline by a FIFO discipline) by inheritance, thereby changing the behavior drastically. If we are to pass off a subclass instance as a superclass instance using subtyping, then we should refrain from making such drastic behavioral changes.

The Java type system provides only weak tools for ensuring a behavioral subtyping relationship between a subclass and its superclass. Fundamentally, the type system is not strong enough to express the desired constraints.[10]. To compensate for this Java provides the finality mechanism to limit inheritance. Final classes cannot be inherited from at all, ensuring that values of its instance type are indeed instances of that class (rather than an arbitrary subclass). Final methods cannot be overridden, ensuring that certain aspects of behavior are "frozen" by the class definition.

Nominal subtyping may also be seen as a tool for enforcing behavioral subtyping relationships. For unless a class extends a given class or is declared

[9] Limited only by finality declarations in the superclass.

[10] Nor is the type system of any other language that I am aware of, including ML

to implement a given interface, no subtyping relationship holds. This helps to ensure that the programmer explicitly considers the behavioral subtyping obligations that are implied by such declarations, and is therefore an aid to controlling inheritance.