Type Systems for the OO Programmer

Marc Etter
OST Eastern Switzerland University of Applied Sciences
MSE Seminar “Programming Languages”
Supervisor: Farhad Mehta
Semester: Autumn 2020

Abstract

Many software developers learn imperative and/or object-oriented programming first in their career. Meanwhile, most of the research done on programming languages is based on the Lambda-Calculus and functional programming languages. These are often not intuitive to grasp for someone who has worked exclusively with object-oriented languages. Some might even be intimidated by the mathematical terminology. This paper aims to show that most of these methods are in common use in Java, but sometimes better known under different names.

Keywords: types, type systems, polymorphism, second-order, lambda-calculus, OOP, object-oriented, Java

1 Introduction

When browsing forums or skimming research papers on type systems, one quickly comes across intimidating-sounding expressions, such as “Intensional Polymorphism”, “Existential Types”, or “Bounded Quantification”. Especially, when getting started with languages more closely designed around the mathematical theories, such as Haskell. However, most of these concepts are used on a daily basis by every object-oriented programmer, they just don’t know it.

In this paper I will go through various concepts of advanced type systems. Every concept will be accompanied by one or more practical examples of how this idea is commonly used in Java. Some typing methods are currently not possible in Java[1]. In these cases, an example in a comparable OO language is given. If not stated otherwise, code listings are original creations for this paper.

In this paper I will purposefully omit most proofs of why a typing works (references to proofs are always given). The goal is not to convince the reader of the integrity of these type systems. Instead, this paper relies on the intuition built through practical experience in programming.

After reading this paper, any object-oriented programmer should feel confident enough around these concepts and their names to dive deeper into the theory in other reading material.

1.1 Assumed Knowledge

This paper assumes the reader is familiar with at least one (statically-typed) object-oriented language, preferably Java. Furthermore, a basic understanding of simple type systems, such as the simply-typed lambda-calculus, is expected. A good introduction to the simply-typed lambda-calculus can be found in [Pie02, p. 89–112].

[1]Current LTS at time of writing is JDK 11.

2 Subtyping and Subclassing

The most fundamental concepts of object-oriented programming languages are subtyping and subclassing. Although these are often used synonymously in informal contexts, they do have a distinct formal definition and function [Bru96, p. 14-15]. The following examples make use of classes and interfaces of JDK 11. A visualization of the inheritance hierarchies can be found in Appendix A.

2.1 Subclassing

When talking about subtyping and subclassing informally, usually programmers mean subclassing. In object-oriented languages, subclassing provides one of the most basic ways to re-use code and avoid unnecessary duplications.

```java
class Dog {
    void bark() {
        System.out.println("Woof!");
    }
}

class GermanShepherd extends Dog {
    // no need to re-implement bark()
}
```

Listing 1. Subclassing for code re-use

Bruce provides the following definition [Bru96, p. 13]:

> *Subclasses support the ability to create incremental differences in behavior by allowing the programmer to define a new class by* inheriting *the code of an existing class, while possibly replacing or adding instance variables and methods.*

From this definition it is clear that the construct of subclassing is intended to be used for small, incremental changes to an already existing class. These changes are not only limited to adding new functionality, but may also modify the superclass’s implementation.

Although subclassing is an easy way to re-use code, care should be taken when to apply this concept. In particular, overuse of subclassing may quickly lead to fragile software [Blo14, p. 81].

2.2 Subtyping

Bruce also provides his definition of subtypes [Bru96, p. 6]:

> *A type S is a* subtype *of a type T (written S <: T) if an expression of type S can be used in any context that expects an element of type T.*

Since an expression of subtype S may be used anywhere where the supertype T is expected, the subtype S should not change the (perceivable) behavior of T.

While interfaces in Java do not implement any behavior per se, they do specify an expected behavior. Since there can be many implementations of the same interface, this can be considered subtyping according to the above definition: Anywhere where the interface is expected, any of its implementations can be used without any change in the behavior that is specified by the interface (assuming the implementations adhere to the interface's specifications). With the addition of default-implementations on interfaces in Java 8, the language blurs the line between interfaces and abstract classes. Formally speaking, if the behavior of the default-implementation is not modified, it is subtyping, otherwise it is subclassing.

An example of subtyping in Java is the JDK's collection API:

```java
List list;

// either is fine
list = new ArrayList();
list = new LinkedList();

list.add("hello");
list.get(0); // retrieves "hello"
```

Listing 2. Subtyping in Java's Collection API

In listing 2, the `List` interface specifies a type that allows adding items and later retrieving them by index. Both the `ArrayList` and `LinkedList` implementations provide this functionality. Although they differ in terms of memory- and time-complexity, the specification of the `List` interface is fulfilled, since it does not make any guarantees in that regard.

2.2.1 Function Subtyping. Function subtyping extends the concept of subtyping from values to functions. Applying the previous definition of subtyping, we can expect a function `subFunction` to be considered a subtype of function `superFunction`, if we can use `subFunction` anywhere where `superFunction` is used.

```java
static List superFunction() {
    return new ArrayList();
}

static LinkedList subFunction() {
    return new LinkedList();
}

static Collection badFunction() {
    return new HashSet();
}

List list;

// either is fine
list = superFunction();
list = subFunction();

// does not typecheck
list = badFunction();
```

Listing 3. Function subtyping without parameters

Listing 3 shows the trivial case of a function without any input parameters. We can easily see that `subFunction` is a subtype of `superFunction`, if the return-type of `subFunction` is a subtype of the return-type of `superFunction`, as visualized in Figure 1.

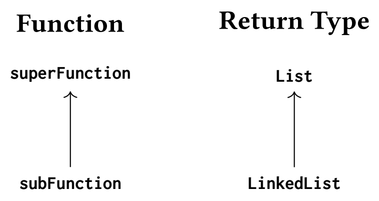

Figure 1. Covariant subtyping relation on return types

On the other hand, looking at functions with an input parameter and without a return-type, we see that the subtyping relation on the parameters is reversed in relation to the functions themselves:

```java
static void superFunction(List list) {
    System.out.println("superFunction");
}

static void subFunction(Collection collection) {
    System.out.println("subFunction");
}

static void badFunction(LinkedList linkedList) {
    System.out.println("badFunction");
}

List arg = new ArrayList();

// either is fine
superFunction(arg);
subFunction(arg);

// does not typecheck
badFunction(arg);
```

Listing 4. Function subtyping without return-type

We have now observed subtyping relations in two different directions. For the return-types the subtyping runs in the same direction, which is called covariant. Whereas for the parameters this relation is inversed (contravariant) [Pie02, p. 185].

This can easily be understood by looking at the counter-example `badFunction`, where the parameter is a subtype of `List`. It is immediately obvious that this will result in an error when attempting to pass the variable `arg` to `badFunction`, as the

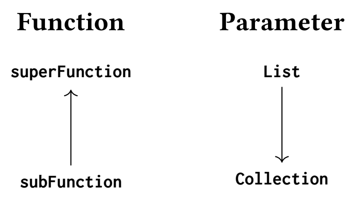

Figure 2. Contravariant subtyping relation on input parameters

type `List` cannot be used in a place where the type `LinkedList` is expected.

Combining both the covariant and contravariant relation, we can subtype more complex functions as follows:

```java
static List superFunction(List list) {
    var result = new ArrayList();
    result.addAll(list);
    return result;
}

static LinkedList subFunction(Collection coll) {
    var result = new LinkedList();
    result.addAll(coll);
    return result;
}

List list;
List arg = new ArrayList();

// either is fine
list = superFunction(arg);
list = subFunction(arg);
```

Listing 5. Function subtyping with parameters

The difference between the covariant and contravariant subtyping relations is visualized in Figure 3.

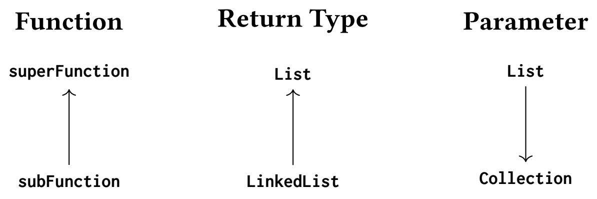

Figure 3. Function subtyping with return-values and input parameters

With the addition of functional interfaces and lambda expressions in Java 8, the concept of assigning types to functions (so-called "arrow-types") got more intuitive. For example the `superFunction` of listing 5 would have type `Function<List,List>` and `subFunction` would be of type `Function<Collection, LinkedList>`. Unfortunately, subtyping of functional interfaces in Java is not as straight-forward, as demonstrated in listing 6.

```java
Function<List,List> superFunction =
    (list) -> {
        var result = new ArrayList();
        result.addAll(list);
        return result;
    }

Function<Collection,LinkedList> subFunction =
    (collection) -> {
        var result = new LinkedList();
        result.addAll(collection);
        return result;
    }

List list;
List arg = new ArrayList();

// this still works as expected
list = superFunction.apply(arg);
list = subFunction.apply(arg);

// assignment is not straight-forward in Java;
// leads to a compile error
superFunction = subFunction
```

Listing 6. Subtyping of functional interfaces

The reason why the assignments in the example above will not compile is a mismatch in the generic types. For assignment to be possible, the type of `superFunction` needs to be slightly modified to `Function<? super List,? extends List>`. This extension to the generics further highlights the covariance of the return-type and contravariance of the parameter-types. The concepts of generics and `extends`/`super` will be revisited in detail in Sections 3.4 and 5.

2.3 Subclassing vs. Subtyping

Although the previous sections have shown the formal definitions of subclassing and subtyping, the exact difference might still not be obvious. Often times subtyping and subclassing are very closely related and therefore, understandably, used synonymously. The difference is most easily summarized as follows: If an expression can be replaced by another expression of a different type, and the typing is still correct, subtyping has been used. If a class extends another class, subclassing has been used.

Since subtyping is defined on a syntactical level and subclassing on a behavioral level, these concepts are not mutually exclusive. In fact, in languages such as Java it is almost always the case that both concepts are used together in a single statement. For example, a subclass in Java is always a (syntactical) subtype of its parent class. This subtyping relation is what allows us to use the subclass anywhere where the parent class is expected. The subclassing relation enables us to re-use behavior of the parent and only change the parts that we choose to.

Most notably, however, subtyping formally does not require that the subtype extends the supertype in terms of class inheritance, although this is always required in Java. An example of subtyping without subclassing is shown in listing 7 using C++ templates. A similar example could be built in Java using reflection, however that would be verified at runtime and no longer by the compiler.

```
template<typename T>
function std::string getName(T const& obj) {
    return obj.name;
}

struct Foo {
    std::string name = "Foo";
};

struct Bar {
    std::string name = "Bar";
};

Foo foo;
Bar bar;

// returns "Foo"
getName(foo);

// returns "Bar"
getName(bar);
```

Listing 7. Subtyping without subclassing

In listing 7 `Bar` can be considered a subtype of `Foo` in the context of `getName`, since `Bar` can be used instead of `Foo` as argument. In Java, an interface specifying `getName` would be necessary, which `Foo` and `Bar` would both have to implement explicitly. Using C++ templates, this interface is implicit, as the template-function will only compile if all occurrences provide an argument with an accessible `name` member. More precisely, `Foo` and `Bar` are both subtypes of the implicit interface that they must possess a `name` member. The example shown also makes use of parametric polymorphism, which is discussed in Section 3.4.

On the other hand, the following listing shows a C++ example of subclassing, where the subtyping relation is broken:

```
struct Parent {
    void print(long l) {
        std::cout << l << std::endl;
    }
};

struct Child : Parent {
    // disallow use of print for ints
    void print(int i) = delete;
};

// Auto-converts 5 to a long
Parent().print(5);

// Compile error: use of deleted function
Child().print(5);
```

Listing 8. Subclassing without subtyping

In contrast to Java, where a subclass is always also a subtype, we have constructed a subclass in listing 8 where we can no longer use the subclass anywhere where the parent is expected. This highlights the importance and necessity of having both the subtyping and subclassing concepts working together.

2.4 Behavioral Subtyping

While the compiler is able to enforce subtyping on a formal, notational level, it is in general impossible to guarantee that all possible subtypes indeed behave as subtypes. The following example illustrates this:

```
interface Executor {
    // Should execute the Runnable r
    void execute(Runnable r);
}

class UndecidableExecutor implements Executor {
    @Override
    public void execute(Runnable r) {
        if (halts(r)) {
            r.run();
        } else {
            throw new UnsupportedOperationException(
                "r does not halt");
        }
    }
}
```

Listing 9. Undecidable behavioral subtyping

Since it is impossible to decide whether `r` halts or not, it is equally impossible to know, whether `execute` will actually run `r` or throw an `UnsupportedOperationException`, breaking the `Executor`'s specification. Although this might be a slightly contrived example, it is easy to imagine real-life cases, where interface contracts are broken by similar exceptions. The concept of (strong) behavioral subtyping and it being undecidable in general was originally described by Barbara Liskov and is commonly known as the *Liskov substitution principle* [LW94].

However, it is noteworthy that Liskov's definition of subtyping is stricter than usually desired for a practical program. In most cases we only want some properties of a type to remain unchanged when substituted with a subtype, while others might change. Using Java interfaces, we can easily specify which properties must be satisfied by every subtype, while leaving any unspecified properties to the implementation (e.g. what memory representation to use).

Lastly, a type might include informal specifications that a subtype should fulfill, which are only expressed in documentation; perhaps because the type system is not powerful enough to describe such properties. For obvious reasons, it is impossible for a compiler to check whether such informal properties are satisfied.

3 Polymorphism

Subtyping allows us to replace expressions with subtype expressions, while still satisfying all compile restraints. However, simply being able to replace an expression in the program's code has very limited benefits. In order to reap the full benefits of subtyping (and subclassing), an additional concept that allows dynamic type substitution both at compile- and at runtime is required.

Precisely this is provided by the concept of polymorphism. In general, polymorphism describes a feature of the type system, where the behavior of the code's execution can change dynamically based on the concrete type provided. In this section I will go over the different forms of polymorphism, as classified by [Str00] and [CW85], summarized by [Pie02, p. 340–341].

3.1 Ad-hoc Polymorphism

Before combining subtyping with polymorphism, we will take a look at ad-hoc polymorphism. It is the simplest form of polymorphism, since it does not rely on any other advanced concepts. Ad-hoc polymorphism allows changing which code to execute solely based on the typing context of an expression. The most common example is method overloading. In method overloading, the typing context is defined by the type(s) of the method argument(s) and the type of the object owning the method, if any.

```java
static String prettyToString(boolean b) {
    return b ? "Yes" : "No";
}

static String prettyToString(int i) {
    // Some int-specific formatting
}

prettyToString(true);
prettyToString(5);
```

Listing 10. Ad-hoc Polymorphism in Java

In Java, method overloading is also possible with a varying number of parameters. In this case, the method to invoke is selected only based on the number of arguments provided, regardless of their types (unless several methods with the same name have the same number of parameters). Therefore, this is no longer a case of polymorphism in the sense of the type-system, but rather a way to provide optional parameters.

Notably, method overloading in Java is always resolved at compile-time. Even if a more specific overload exists, the method will be selected that matches the statically declared type the best.

3.1.1 Method Overriding. Overriding a method in a subclass is a common case of polymorphism, which falls under the classification of ad-hoc polymorphism. At runtime, the JVM checks the concrete type of the object, on which the method is invoked, and the most specific override is selected (child-first).

```java
Animal dog = new Dog(); // Dog extends Animal
Animal bird = new Bird(); // Bird extends Animal

dog.makeSound(); // Woof
bird.makeSound(); // Chirp
```

Listing 11. Method overriding in Java

3.2 Subtype Polymorphism

Subtype polymorphism ties together the previous section on subtyping with polymorphism. In its original definition as provided by Pierce, subtype polymorphism is what enables us to make use of subtyping at runtime, e.g. based on user input or configuration files.

In this type of polymorphism, the concrete type of a variable can be exchanged with any of its subtypes without changing the program's behavior.

```java
List list;

// returns ArrayList or LinkedList
// based on app configuration
list = getList();
```

Listing 12. Subtype polymorphism in Java

3.2.1 Subclass Polymorphism. As previously discussed, (strict) subtyping does not allow changing any program behavior, which severely limits its practical use. Extending subtype polymorphism to include subclasses results in the type of polymorphism that object-oriented programmers are most familiar with. Pierce does not classify *subclass polymorphism* as a separate type of polymorphism. Although intuitively this is the most common form of polymorphism in object-oriented languages, it is actually only a combination of subtype polymorphism and ad-hoc polymorphism. Whenever a subclass is used, that subclass might:

- inherit behavior from its parent
- modify behavior of its parent
- add new behavior to its parent

Any behavior that is inherited unchanged falls directly under the category of subtype polymorphism. Any behavior that is changed is usually done so by method overriding, which is implemented with ad-hoc polymorphism. Finally, any new behavior is irrelevant, as new members are not visible when viewed as the parent type. Therefore, it is not necessary to add *subclass polymorphism* as a fundamental type of polymorphism.

3.3 Intensional Polymorphism

Sometimes it is necessary to check the (concrete) type of a variable at runtime. Traditionally, this is not possible and requires workarounds, such as member variables containing the name of the class. Fortunately, many modern programming languages (including Java) implement so-called intensional polymorphism.

Intensional (not to be confused with *intentional*) means requiring additional information outside of its own context. Intensional polymorphism is therefore a kind of polymorphism that requires extra information to resolve that cannot be deduced exclusively from the type system. This extra information is usually provided by the compiler and added to objects and classes. This metadata is essentially the same as the workaround mentioned above, but is generated automatically, removing any human errors and providing a global naming standard. In some cases, intensional polymorphism even adds additional syntax constructs for ease-of-use.

```java
Animal animal = new Dog();
if (animal.getClass() == Dog.class) {
    Dog dog = (Dog) animal;
    dog.bark();
}
```

Listing 13. Intensional Polymorphism in Java

The `getClass` function and `class` static member in listing 13 show an example of intensional polymorphism. These members are not defined by the programmer, but generated by the compiler and made available during runtime. Thanks to this extra information, we can build a conditional to compare types at runtime and run some specific code based on a type distinction.

This example was chosen to highlight the metadata of the `Class` object at runtime. In practice, such code should use the `instanceof` operator, which works similarly under the hood, but also checks for subclasses:

```java
Animal animal = new GermanShepherd();
if (animal instanceof Dog) {
    Dog dog = (Dog) animal;
    dog.bark();
}
```

Listing 14. Intensional Polymorphism in Java, in practice

3.4 Parametric Polymorphism

Parametric polymorphism is another concept that is well understood in object-oriented programming. It is usually known as *genericity* or *generics*. In parametric polymorphism we tell the compiler that a certain piece of code works "for all possible types". It is a very powerful form of polymorphism and the basis of the popular *System F* [Pie02, p. 341–344], which is the core of Haskell's type system.

Parametric polymorphism, or genericity, allows variables over types, instead of only expressions. This increases the level-of-depth of possible type-checking, for example by specifying that the return-type of a function is always the same as the argument-type.

```java
static <T> T identity(T anything) {
    return anything;
}

identity(true); // returns a Boolean
identity(5); // returns an Integer
```

Listing 15. Parametric polymorphism in Java

Without generics, such code would have to be written using `Object` in Java, where almost all type information is lost. Generics can be found in most modern programming languages with static typing, such as C#, C++, TypeScript, or Haskell.

The concept of some code being applicable to all types is highlighted even further when considering the syntax used in Haskell:

```haskell
length :: forall a. [a] -> Int
-- implementation omitted
```

Listing 16. Parametric polymorphism in Haskell

The `length` function calculates the length of a list, regardless of what type the elements of that list are. Although the `forall a.` is optional in Haskell's syntax, it does state explicitly that the type-parameter `a` is a placeholder for all possible types. This syntax is derived directly from the mathematical notation $\forall x$ which is pronounced "for all x". In mathematics, the concept of some function or statement holding for "everything" is called *universal quantification*.

3.4.1 Higher-Rank Types.

A function with type parameters is considered a rank-1 type [Pie02, p. 359]. If a type-parameter was allowed to represent a rank-1 type, a function with such a parameter would be a rank-2 type. For example, a rank-2 type method would be required, if we want to pass the generic `identity` function as an argument to another function:

```java
// does not compile
static <T> void foo(Function<T,T> func) {
    Boolean b = func.apply(true);
    Integer i = func.apply(5);
}
```

Listing 17. Invalid rank-2 method in Java

Higher rank types (i.e. rank-2 and above), however, are not possible in Java, and are indeed restricted in most programming languages, since they introduce a loss of type inference [Pie02, p. 354–359]. In short, it is no longer possible to automatically infer what type an expression has based on the context alone. This means that explicit type annotations must be given. Although Java requires explicit type annotations almost everywhere anyway, the use-case of higher rank types is fairly rare, so most languages have decided to forgo this feature in lieu of simplicity and robustness.

```java
interface Func {
    <T> T apply(T arg);
}

class Identity implements Func {
    @Override
    public <T> T apply(T arg) {
        return arg;
    }
}

class Null implements Func {
    @Override
    public <T> T apply(T arg) {
        return null;
    }
}

// compiles fine
static void rank2(Func func) {
    Boolean b = func.apply(true);
    Integer i = func.apply(5);
}

rank2(new Identity());
rank2(new Null());
```

Listing 18. Rank-2 workaround in Java

Listing 18 shows a possible workaround in Java, where the `rank2` function is kept at rank-1 by encapsulating the polymorphic function in a non-parameterized object. The type-information that the function should return the same type as its argument is encoded in the new `Func` interface, instead of the type-parameters of `rank2`.

3.4.2 Higher-Kinded Types.

Parametric polymorphism introduced the concept that a value can have any type, which is captured by a type-variable. That type-variable can then be re-used to ensure other values have that same type.

Higher-kinded types extend this concept from values to types. This means a type can have any "type", called *kind*. Just as with higher-rank types, Java does not support higher-kinded types. However, higher-kinded types are very common in functional programming languages, such as Haskell. If Java supported higher-kinded types, an example could look as follows:

```java
class NameStore<T> {
    T<String> names;

    T<String> getNames() {
        return names;
    }
}

// should return List<String>
new NameStore<List>().getNames();

// should return Set<String>
new NameStore<Set>().getNames();
```

Listing 19. Invalid higher-kinded type in Java

To work around this limitation, explicit classes can be created:

```java
class NameList {
    List<String> names;
}

class NameSet {
    Set<String> names;
}
```

Listing 20. Workaround for higher-kinded types in Java using explicit classes

However, we lose the subtyping relation between `NameList` and `NameSet`, unless we use some very verbose generics:

```java
interface NameStore<T> {
    T getNames();
}

class NameList implements NameStore<List<String>> {
    // ...
}

class NameSet implements NameStore<Set<String>> {
    // ...
}

NameStore nameList = new NameList();
NameStore nameSet = new NameSet();
```

Listing 21. Workaround for higher-kinded types in Java with generics

Although we can now use either `NameList` or `NameSet` wherever a `NameStore` is expected, we no longer have the requirement of `names` being `String`s. Using bounded quantification of Section 5 we can restrict our `NameStore` further, at the cost of limiting our representations to subtypes of e.g. `Collection`, which might not be desirable:

```java
interface NameStore<T extends Collection<String>> {
    T getNames();
}
```

Listing 22. Workaround for higher-kinded types in Java with bounded quantification

These workarounds are similar to how the JDK's collection API provides subclasses for different implementations of e.g. `List`, instead of using parameters for what implementation to use.

Just as with higher-rank functions, the use-case for higher-kinded types is fairly rare and there are plenty of workarounds available, which might sometimes even be better understandable. Likely for those reasons most languages have opted to not support higher-kinded types.

4 Existential Types

With parametric polymorphism implementing the universal quantifier ∀ ("for all"), it is only natural to think about implementing the existential quantifier ∃ ("exists"), as they are the two most common mathematical quantifiers [Vau21]. While $\forall x$ specifies that a certain condition or statement must be true for any possible value of x, $\exists x$ demands that there need only be at least one value of x that satisfies the condition or statement.

Applied to polymorphism, this means that instead of having a generic parameter that can take on any type, we get a type construct for which it is satisfactory to have only some types (but at least one) lead to a successful type-check.

An intuitive example of existential types in Java are interfaces. An interface declares a set of functions its implementations must provide, but only some types in the program fulfill this interface. Additionally, the interface is only useful, if there is at least one implementation available at runtime.

```java
interface Flying {
    void takeOff();
    void land();
}

Flying flying;

/* For this program to run, we expect one or more
 * types implementing Flying, or there can never be
 * a value assigned to the variable "flying".
 */

flying = new AirPlane();
flying = new Eagle();

flying.takeOff();
flying.land();
```

Listing 23. Interface as existential type in Java

In contrast to a universal type (or parametric polymorphism), we cannot assign all possible types to `flying`, only some. This gives us sort of a half-way construct between full-on parametric polymorphism, where almost no assumptions can be made about the concrete type, and using concrete types, where we know exactly what capabilities the type has. By expressing with the type system that we only care about a subset of the concrete types' functionalities, we gain the ability to exchange the concrete type in a limited fashion, namely with any other type that guarantees to share the same subset of functionality.

4.1 Package Semantics

In his derivation and use of existential types, Pierce mentions packages or modules as a primary use-case [Pie02, p. 364]. In fact, interfaces are conceptually the same thing as packages or modules, as far as the type system is concerned. All three of these constructs have in common that they expose some functionality in a declarative manner (methods on the interface, or `public` types in a package/module), while hiding the rest.

In the same way that the concrete type of an interface can be exchanged freely, the contents of a package can be exchanged, as well, by changing the `import` statement. An example of this is the Guava[2] library for functional interfaces, which has been partially incorporated into the JDK with version 8. In many instances (unfortunately not in all), it is sufficient to simply exchange the previous Guava-import with the new JDK-import. Therefore, an `import` statement is equivalent to the assignment operator in the sense of choosing which concrete implementation of the interface or package we want to use, hidden behind the abstraction provided by the existential type.

4.2 Object Semantics

Originally, the term "object" was coined as an aggregation of multiple values into a single atomic symbol [MBE+60, p. 88]. This idea was later extended to include modifiers for exposing and hiding certain members (known as *information hiding*). The previous section on package semantics already touched on the concept of using `public` modifiers to declaratively change the visibility of certain functionality. In the case of classes and objects, these modifiers are not only used on the type-level, but more importantly on class members.

[2]https://github.com/google/guava (November, 2020)

We can think of objects as implicitly providing an interface with only their `public` (or in Java possibly `protected` and default) members being declared, depending on the scope where the object is used. As shown above, interfaces are equivalent to creating an existential type. Therefore, the addition of information hiding incorporated the concept of existential types into objects [Par71][Par72].

As with subtype and ad-hoc polymorphism, we can exchange the concrete type of an object with any of its subclasses and the implicit interface (i.e. the existential type) is still satisfied. Notably, it is not possible (in Java) to hide a member in a subclass that was declared to be visible in its parent class, since that would break the interface.

5 Bounded Quantification

Imagine we wanted to write a function that takes a list of integers, increments every element by one, and then returns that list. If we attempt to solve this using polymorphism alone, we get a function that only works in some cases, also known as a "partial function":

```
<T> T incList(T arg) {
    // Use intensional polymorphism
    if (arg instanceof List) {
        // We know it is a list,
        // but not the element type
        List list = (List) arg;
        for (int i = 0; i < list.size(); i++) {
            Object el = list.get(i);
            if (el instanceof Integer) {
                list.add(i, (int) el + 1);
            } else {
                throw new IllegalArgumentException("
                    Not an integer element");
            }
        }
    } else {
        throw new IllegalArgumentException("Not a
            list");
    }
    return intList;
}
```

Listing 24. Increment integer list with polymorphism

On the other hand, if we attempt a solution using subtyping, we run into a different problem:

```
List<Integer> incList(List<Integer> intList) {
    for (int i = 0; i < intList.size(); i++) {
        list.add(i, intList.get(i) + 1);
    }
    return intList;
}

var myList = new ArrayList<Integer>();
var result = incList(myList);
// result is of type List<Integer>,
// not ArrayList<Integer>
```

Listing 25. Increment integer list with subtyping

This solution looks much cleaner and probably preferable to the first one. However, we lose the type information on the concrete type of the argument using subtyping, which might not be desirable. If nothing else, the solution using parametric polymorphism at least managed to preserve the concrete type.

What if we combined subtyping with polymorphism to achieve a solution that satisfies all criteria? Precisely this combination is the concept of *bounded quantification*. In bounded quantification we use universal quantifiers, but impose some additional boundaries on them. This results in an expression that is no longer read as "for all possible types", but rather as "for all possible types within these constraints". Usually, bounds are used to restrict types based on subtyping hierarchies. Java supports those restrictions in both directions: `extends` for upper bounds and `super` for lower bounds.

Applying bounded quantification to the example above, we can write the following solution:

```java
<T extends List<Integer>> T incList(T intList) {
    for (int i = 0; i < intList.size(); i++) {
        intList.add(i, intList.get(i) + 1);
    }
    return intList;
}

var myList = new ArrayList<Integer>();
var result = incList(myList);
// result is now still a concrete ArrayList
```

Listing 26. Bounded quantification in Java

The solution in listing 26 uses parametric polymorphism to preserve the argument type, but imposes the additional constraint that the argument must be a subtype of `List<Integer>`. This simple addition gets rid of all problems we had with either of the standalone solutions. Adding bounded quantification to *System F*, results in the extended type system $F_{<:}$ ("F-sub").

Unfortunately, implementing bounded quantification in a language introduces a whole new class of problems. Namely, given two types S and T, it is not always possible to decide whether S is a subtype of T, or not. Grigore shows that it is possible to construct a type table using bounded quantification in Java, where deciding whether S is a subtype of T is equivalent to the halting problem, which is known to be undecidable [Gri17].

5.1 Bounded Existential Types

By adding constraints to generics, we have applied bounds to universal types. Similarly, we can add bounds to existential types. In short, this means that we expect at least one type to exist that satisfies the contract of our existential type (interface), but also fits into the given bounds.

```java
interface List<T> extends Collection<T> {
    // List-specific methods
}
```

Listing 27. Bounded existentials in Java with inheritance

By having an interface extend another interface (or similarly, an abstract class extend any other class), we not only require that the concrete type implements the methods of `List`, but also those of `Collection`. In this case, `List` is a subtype of `Collection` and is also a bounded existential type.

We can also add bounds to an existential type without inheritance, using Java's generics:

```java
<T extends Flying & Animal> void animalFlight(T
     flyingAnimal) {
    flyingAnimal.land(); // Flying
    flyingAnimal.eat(); // Animal
    flyingAnimal.takeOff(); // Flying
}
```

Listing 28. Bounded existentials in Java with generics

The example above requires that the argument must implement both the interfaces `Flying` and `Animal`. This can be interpreted either as adding the bound of `Animal` to the existential type `Flying` (or vice-versa), or simply as creating a new, anonymous existential type with the combined requirements of both interfaces. If `Animal` was not an interface, but a concrete type (e.g. `Bird`), then the situation would be a bit clearer, where the constraint of having to be a subtype of `Bird` was added onto the existential type `Flying`. Permitted arguments would be any flying bird, but not e.g. `Ostrich` (does not satisfy the existential `Flying`) or `Bat` (does not satisfy the bound of being a subtype of `Bird`).

6 Conclusion

The concepts of advanced type systems are often known with different, sometimes less formal names. This creates the illusion that the theory of (advanced) type systems consists of exotic ideas that can only be understood by studying functional programming languages. As demonstrated in this paper, most concepts of advanced type systems are actually in common use in object-oriented programming languages. Furthermore, all of these concepts can easily be understood using practical experience and intuition with every-day programming languages, such as Java.

To get a deeper understanding of the theory and be able to prove why these type systems are sound (and in which cases they are not!), other material should be consulted, such as the ones referenced in this paper. Keeping in mind the examples and analogies of this paper, the theory-focused work should be way more approachable.

References

[Blo14] Joshua Bloch. *Effective Java*. Addison-Wesley, Boston, MA, USA, second edition, 5 2014.

[Bru96] Kim Bruce. Typing in object-oriented languages: Achieving expressiveness and safety. 11 1996.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985.

[Gri17] Radu Grigore. Java generics are turing complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 73–85, New York, NY, USA, 2017. Association for Computing Machinery.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

[MBE+60] McCarthy, J., Brayton, R., Edwards, D., Fox, P., Hodes, L., Luckham, D., Maling, K., Park, D., and Russel, S. *LISP I Programmer's Manual*, March 1960.

[Par71] David Parnas. Information distribution aspects of design methodology. volume 71, pages 339–344, 01 1971.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[Str00] Christopher Strachey. Fundamental concepts in programming languages. *High. Order Symb. Comput.*, 13(1/2):11–49, 2000.

[Vau21] Math Vault. *Comprehensive List of Logic Symbols*, January 2021. https://mathvault.ca/hub/higher-math/math-symbols/logic-symbols.

A JDK 11 Inheritance Hierarchies

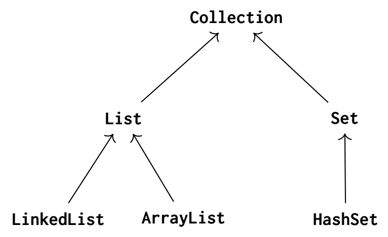

Figure 4. JDK 11 Collection Hierarchy