510: Programming Languages

Subclassing and Subtyping

David Walker

Classes: Code or Types?

Classes in Java play two **distinct** roles:

1. As bodies of code, consisting of constructors, fields, and methods.

2. As “types”, classifying instances of the class.

It is important not to confuse the two roles!

Classes: Code or Types?

Strictly speaking, there is a distinction between

1. The **class** C, thought of as a body of code.

2. The **type** $\mathrm{Inst}(C)$ of objects of the class C.
   In particular, `new` C `(...)` has type $\mathrm{Inst}(C)$.

In Java the distinction is ellided — the class C is **identified** with the type $\mathrm{Inst}(C)$.

Instance Types

Java adopts a **nominal** notion of instance types.

1. **All** object types are of the form Inst(C) for some class C. (But see the discussion of interfaces below.)

2. Two object types are equivalent iff they arise from the **same** class — Inst(C) $\equiv$ Inst(D) iff $C = D$.

In particular, **distinct** classes with **identical** fields and methods determine **distinct** object types!

`Datatype` declarations in ML are treated nominally.

Object Types

Other languages employ a **structural** view of object types.

1. Object types specify the types of the fields and methods of an object; the name of the object type does not matter.

2. Two object types are equivalent iff they specify the same fields and methods with the same types. (This is called **structural equivalence**.)

In a language based on structural equivalence a given object may have many equivalent object types.

Signatures in ML are compared structurally. There is no "`sigof(`S`)`" construct.

Object Types

A (hypothetical) object type:

```
objtype STACK {
  STACK new ();      // constructor
  int size;          // size field
  void push (Object x);
  Object pop ();
  boolean empty ();
}
```

In a structural type discipline, an object has this type iff it has the specified fields and methods with the specified types.

Interfaces

Java's interfaces are **almost** (but not quite) object types.

1. Interfaces may specify the public **methods** of an object.

2. But they may not (unfortunately) specify the public **fields**!

Interfaces

In truth, Java interfaces are a special kind of class:

1. All fields are implicitly `public static final`, and hence are just constants of the interface.

2. All methods are implicitly `public abstract`, so only their types are specified, not their implementation.

Strictly speaking, there is a distinction between an interface I and the object type $\mathrm{Inst}(I)$. In Java this distinction is ellided.

Interfaces

A class in Java can **implement** more than one interface, but it can **extend** only one class.

The restriction on extension is called **single inheritance** — there is always precisely one parent class (with `Object` as root).

Each interface of a class specifies an **aspect** of the behavior of objects of that class by specifying a collection of methods and their types.

This is sometimes called **multiple interface inheritance**.

Interfaces

Java employs a **declarative** view of implementation — a class C implements an interface I iff it is **explicitly** declared to do so.

This is a corollary of two related decisions:

1. The nominal view of object types.

2. The treatment of interfaces as classes.

Interfaces

The alternative is to adopt a **descriptive** view of implementation — a class C implements an interface I iff the objects of the class C satisfy the requirements of I.

The descriptive view is more flexible — you are not required to **anticipate** the possible interfaces of interest for a class C when C is defined. Instead you can ask **post hoc** whether or not C implements I.

Interfaces

The descriptive view relies on

1. The structural view of object types.

2. The treatment of interfaces as types, not classes.

Signature matching in ML is descriptive.

Java's Type System

The type system of Java has these types T:

1. `int`, `boolean`, `float`, ...
2. `void` (which really should be `unit`!).
3. Array types T`[]`.
4. Inst(C), where C is a class.
5. Inst(I), where I is an interface. (This is covered by the previous case.)

Java's Type System

We write $E : T$ to mean that an expression E has type T.

1. 14 : **`int`**, *etc.*.

2. **`new`** T **`[`**n**`]`** : T**`[]`**.

3. **`new`** C **`()`** : Inst(C).

4. E : Inst(D) if E is an instance of C and C extends D.

5. E : Inst(I) if E is an instance of C and C implements I.

Subclassing = Inheritance

A class C is a **subclass** of a class D, written $C \triangleleft D$, iff C is **explicitly declared** to extend D.

1. C has **add** more methods and fields, perhaps by **overloading**.

2. C may **override** methods with new implementations, provided that the argument types in the subclass match the argument types in the superclass exactly.

Subclassing = Inheritance

The subclass relation is **purely** a matter of how the code is written; it does not necessarily carry any semantic force.

1. The subclass may override the superclass's methods with **arbitrary** code.

2. There is no guarantee that the objects of the subclass behave at all like the objects of the superclass.

In practice we use subclassing when the behavior of subclass instances is consistent with the behavior of superclass instances.

Interface Inheritance

Inheritance also applies to interfaces: we may define an interface I by extending another interface J, written $I \lhd J$.

The sub-interface extends the super-interface with new methods (and fields).

Write $C \rightsquigarrow I$ to mean that class C is declared to implement interface I.

Subtyping

Subtyping is a binary relation between types, written T <: U, satisfying these properties:

1. It is reflexive (T <: T) and transitive (if T <: U and U <: V, then T <: V.)

2. It validates the **subsumption** principle: if $E : T$ and T <: U, then $E : U$.

Both conditions must be satisfied to have a subtype relation!

Subtyping

The significance of the subtyping relation is that it supports an important principle of **code re-use**:

> If T `<:` U, then a value of type T may be provided whenever a value of type U is required.

For example, if `int <: float`, then we may call `sin` with an argument of type `int`.

Subtyping in Java

The **fundamental principles of subtyping** in Java:

1.(a) Subclasses determine subtypes: if $C \triangleleft D$, then Inst(C) <: Inst(D).

(b) Subinterfaces determine subtypes: if $I \triangleleft J$, then Inst(I) <: Inst(J).

2. Implementation declarations determine subtypes: if $C \rightsquigarrow I$, then Inst(C) <: Inst(I).

Subtyping in Java

In other words:

1.(a) If C is a subclass of D, then every instance of C may be thought of as an instance of D.

(b) If I is a subinterface of J, then every instance of I is also an instance of J.

2. If C implements I, then every instance of C is also an instance of I.

Subtyping in Java

These are (almost) the **only** principles of subtyping in Java! (More on array subtyping later.)

That is, the Java subtyping relation is **inductively defined** as the **least** relation closed under these principles.

This is called a **nominal** interpretation of subtyping! The **only** way subtyping arises is by **explicitly declared** implementation and extension relations.

Subtyping in Java

The nominal interpretation forces some awkward coding conventions. If you want T `<:` U, you have these choices:

1. (a) Ensure that T and U are object types of classes C and D such that $C \lhd D$.

   (b) Ensure that T and U are interfaces such that $T \lhd U$.

2. Ensure that T is an object type for class C, that U is an interface type, and that $C \leadsto U$.

Structural Subtyping

The alternative is called **structural subtyping**. It relies on having descriptive object types that specify the public fields and methods of an object.

One object type T is a **structural subtype** of another U iff the requirements of U are satisfied by T.

(Compare signature matching in ML.)

Structural Subtyping

More precisely,

1. The fields of T must be subtypes of the fields of U.

2. The methods of T must be such that

   (a) The result type in T is a **subtype** of the result type in U.

   (b) The argument types in T must be **supertypes** of the argument types in U.

   This is called the **co- and contra-variance** principle of subtyping.

Array Subtyping in Java

Oddly, Java supports structural subtyping for arrays . . . and gets it wrong!

Specifically, Java adopts the following principle for array types:

if T `<:` U, then T`[]` `<:` U`[]`.

In words, if T is a subtype of U, then an array of T's is a subtype of an array of U's.

THIS IS AN INVALID PRINCIPLE OF SUBTYPING, BECAUSE IT VIOLATES THE SUBSUMPTION PRINCIPLE!

Array Subtyping in Java

Suppose that `ColorPoint <: Point`, as in the file `subtype.java`.

Consider the following code:

```
Point p = new Point ();
ColorPoint cp = new ColorPoint ();
ColorPoint [] A = new ColorPoint [10];
Point [] B = (Point []) A;  // view A as an array of Points
B[0] = p;      // ok, since B : Point []
A[0].color;    // type correct, but invalid!
```

Array Subtyping in Java

This shows that the Java type system is **unsound**!

We've produced a value of type `ColorPoint` that is not, in fact, an instance of the `ColorPoint` class!

Array Subtyping in Java

Does Java dump core in this case? No, but avoiding this is costly!

1. **Every** array assignment requires a **run-time** type check to ensure that the assigned value is of “true” type of the array. This costs time (to check) and space (for the tag).

2. **Every** object must be **tagged** with its class to make this check possible. This violates the **pay-as-you-go principle** of language design: you pay for the tags, even if you don’t need them!

Casting

Run-time checks are also used to support type-safe **casting**.

If $E : T$, then $(U)E : U$.

Upcasting: If T <: U, then the cast is **safe** and has no run-time cost.

Downcasting: If U <: T, then the cast may or may not be safe, depending on whether in fact $E : U$ (and hence $E : T$). A run-time check is required to ensure safety; an exception is raised if the check fails.

Subclassing $\neq$ Subtyping

In Java subtyping and subclassing are completely identified.

However, the two notions are conceptually distinct!

1. Subclassing = inheritance.

2. Subtyping = validates subsumption.

Is this a distinction without a difference?

Subclassing $\neq$ Subtyping

In a language with nominal subtyping, a subtype relationship **reflects** a subclass relationship.

- Subtyping arises only by sub-classing.

In a language with structural subtyping, a subtype relationship **need not reflect** a subclass relationship.

- The relationship is descriptive, not declarative.

- There are more subtype relationships than subclass relationships.

Subclassing $\neq$ Subtyping

However, **even** in a language with nominal subtyping, subclassing **need not** imply subtyping!

- True for Java, because the language is so simple.

- Not true for many extensions to Java (or other, more expressive, OOPL's).

Example: the extension of Java with "`This`" types violates the principle that subclassing entails subtyping.

The Binary Method Problem

How to define methods that examine another object of the same class?

```
class Point {
  private int x, y;
  public boolean eq (Point p) {
    return p.x == this.x && p.y == this.y;
  }
}
```

The Binary Method Problem

```
class ColorPoint extends Point {
  private String c;
  // called for statically-known ColorPoint's
  public boolean eq (ColorPoint p) {
    return super.eq(p) && p.c == this.c;
  }
  // called for Points, perhaps a ColorPoint
  public boolean eq (Point p) {
    if (p instanceof ColorPoint) {
       return eq ((ColorPoint)p);
    } else {
       return super.eq(p);
    }
}
```

Java with `This` Types

A solution to the binary method problem:

```
class Point {
  private int x, y;
  public boolean eq (This p) {
    return p.x == this.x && p.y == this.y;
  }
}
class ColorPoint extends Point {
  private String c;
  public boolean eq (This p) {
    return super.eq (p) && p.c == this.c;
  }
}
```

Java with `This` Types

The type **`This`** stands for the type of the object itself.

- So **`this`** has type **`This`**.

- The meaning of **`This`** changes in the subclass to be the object type of the subclass.

The method **`eq`** is **overridden** in the subclass to have a more specialized meaning.

Java with `This` Types

Suppose that `ColorPoint <: Point`, as in Java.

Let `p` be a `Point` and `cp` be a `ColorPoint`.

Then `cp.eq(p)` is well-typed since `cp` is a `Point`, and hence `cp.eq` accepts `Point`'s.

But this is unsound! The code for `cp.eq` accesses the instance variable `this.c`, which does not exist for `p`!

Java with `This` Types

Conclusion: in the presence of `This`, subclasses do not yield subtypes!

In the example, the object type of `ColorPoint` is **not** a subtype of the object type of `Point`!

(The counterexample shows that supposing otherwise leads to a run-time error.)

Java with `This` Types

What to do? Two choices:

1. Impose a run-time check whenever `This` is used to ensure that no type error has arisen. Undermines the purpose of `This`, namely to track more type information statically.

2. Separate the notions of subclassing and subtyping. Subclasses do not necessarily determine subtypes; there may be more subtype relationships than subclass relationships.

The Fragile Base Class Problem

Inheritance is a form of **implementation-on-implementation** dependence.

- If you override a method, you change the behavior of the superclass methods that call it.

- How can you tell what are the implications of an override without seeing the superclass's code?

The Fragile Base Class Problem

Indeed, Java offers no means of reducing this to an **implementation-on-interface** dependency.

- Cannot extend an **unspecified** class, even with a **specified** interface. You must extend **one**, **specific** class.

- The notion of interface is too weak in any case, as discussed earlier.

That is, there is no analogue of an ML functor in Java. Even if there were, it would still be very difficult to use it because of the override problem.

The Fragile Base Class Problem

This introduces a fundamental problem for **separate development**, called the **fragile base class problem**:

- The sub-class cannot be compiled separately from the super-class.
- Changes to the super-class require recompilation of the entire hierarchy rooted at that class.

In this sense object-oriented programming is **anti-modular**! You cannot treat the super-class as a "black box".

Summary

Subclassing and subtyping are **separate** notions. Neither necessarily entails the other.

Subtyping is the **essence** of code re-use, because of the subsumption principle.

Subclassing is of limited utility for code re-use since it reflects how the code was **created**, rather than how it **behaves**.

The FBC problem exposes the **anti-modular** nature of inheritance, a serious obstacle to separate development.