Abstract Data Types and Object Orientation

Abstract Data Types

As seen, physical machines “understand” only strings of bits. On the contrary, the more complex programs become, the more complicated it is for developers to reason at that low level (where an array, a table, or a graph have more or less the same “surface”).

Thus, programming languages adopted types also as a kind of “**capsule**” to organise values from undistinguishable strings of bits into distinct entities with their own set of operations.

In type-safe languages, the capsule represented by a type is completely **opaque** and the user cannot interact with its values unless mediated by the capsule. In these cases, we call the type an **abstract data type** (ADT), i.e., one that defines the **possible values** that inhabit the type and the **possible operations** they support and their **behaviour**.

Abstract Data Types

ADTs do not simply enforce the abstractions built into the language (e.g., that `2+true` is a typing error), but also **programmer-defined ones**—i.e., besides “protecting” the machine from the program, they can encapsulate and **protect parts of the program from each other**.

Conventionally an ADT consists of:

- the abstract type name A;
- the concrete representation type T;
- implementations of operations for creating, querying, and manipulating values of type T;
- an **abstraction boundary** that encloses T and makes it accessible only through the operations of A.

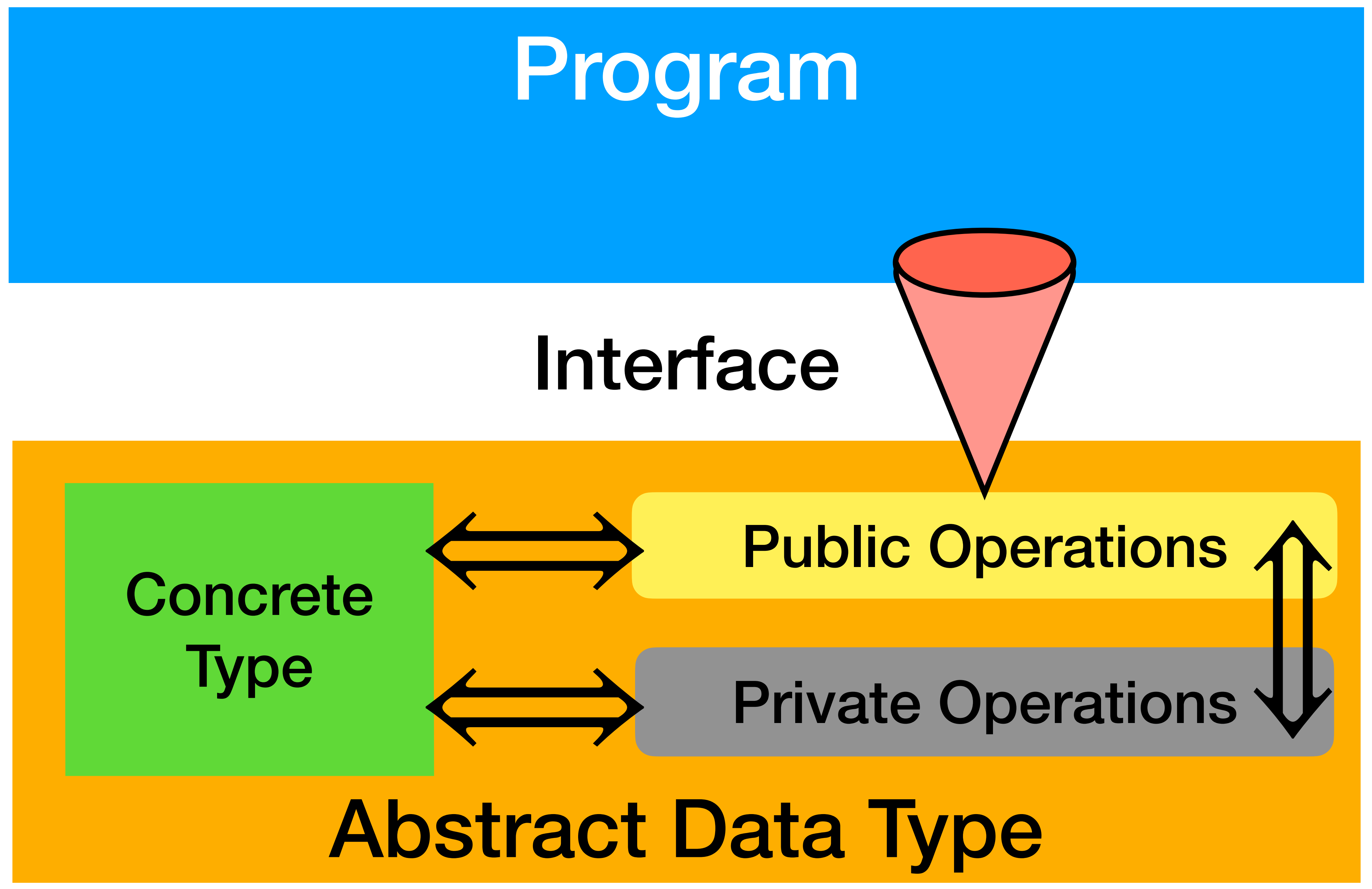
Hence, users can create new values of type A, pass them around their program, store them in data structures, etc., but **they cannot directly inspect and change the concrete representation** of type T.

```
type Counter
 representation int
 signature
  new: Counter
  get: Counter -> int
  inc: Counter -> Counter
 operations
  new = 1
  get = fn( int i ){ i }
  inc = fn( int i ){ i+1 }
```

Using the CLU / Ada notation for Abstract Data Types

Information hiding

Abstracting implementations via **interfaces** is a fundamental principle in programming languages: it allows the producer and consumer of software components to **establish a contract** by agreeing on a "surface"—the interface—the consumer expects to use and the producer promises to support.



Following the metaphor, the "language" of (abstract data) types is the one the two parties use to formalise that contact.

Of course, a synonym of "abstraction", here, is **information hiding**, i.e., we hide behind interfaces the implementation details of a given data type (both its structure and how its operations work).

Representation independence

Interestingly, information hiding **improves code compositionality** in a way similar to what we saw for subtype/parametric polymorphism: by omitting (some of) the details of the implementation (in polymorphism, the actual type of a type parameter) we (consumers) cannot make assumptions on the actual implementation of the type and write code that only uses the given interfaces.

This leads us to the property of type-safe ADTs, called **representation independence**, whereby two correct (well-typed) implementations of the same ADT are **observability non-distinguishable** by consumers of that type.

Since ADTs are representation-independent, it is safe to use any type-compatible implementation of a given ADT and interchange alternative implementations thereof. Of course, the guarantees of types cover as far as types are involved—e.g., they do not track performance differences.

Abstract Data Types — Rust

```rust
trait Counter {
 fn new() -> Self;
 fn get( &self ) -> u32;
 fn inc( &self ) -> Self;
}
```

signature

```rust
struct SC { counter: u32 }
impl Counter for SC {
    fn new() -> Self { SC { counter: 0 } }
    fn get( &self ) -> u32 { self.counter }
    fn inc( &self ) -> Self
     { SC { counter: self.counter + 1 } }
}
```

implementation

```rust
fn use_counter<C>( c: &mut C ) where C: Counter {
  let c = c.inc(); let c = c.inc(); print!( "{}", c.get() );
}
fn main(){ use_counter( &mut SC::new() ) }
```

Rust has a direct way to implement ADTs via **traits**.

In general, traits are a linguistic mechanism to wrap functionalities common to different types.

The trait represents the abstract type whose interface gives access to users to the concrete type, by means of the **impl**ementation of its operations.

Modules

From the point of view of code organisation, ADTs are also a way to “wrap” the code that belongs to the same (abstract data) type within a single interface. However, e.g., when we write libraries but also large programs, we might need a way to provide multiple ADTs within the same “package”.

The linguistic mechanism that achieves this is usually called **module** (or, case in point, **package**).

Different programming languages attribute to modules different properties, but a common one is to let developers partition they programs, so that each module contains data (types, variables, etc.) and operations (functions, code, etc.) available within the module. The module also defines the **visibility** of the data it encloses, i.e., what a user of the module can “see” and use.

Abstract Data Types and Existential Types

From the point of view of type theory, abstract data types and modules introduce the concept of **existential types** — as a shorthand for existentially quantified types — where a definition like

```
trait Counter { fn new…; fn get…; fn inc…; }
```

would correspond to the existential type

$$CounterADT = \{\, \exists X, \{\mathsf{new} : () \to X,\ \mathsf{get} : X \to int,\ \mathsf{inc} : X \to X\}$$

Notice that the type does not “witness” the concrete type, which is tied to the internal representation of a value of type `Counter`. This indicates that we can have inhabitants of $CounterADT$ whose internal details we are able forgo, still obtaining type safety. With the existential, as long as we have an inhabitant of that type, we can safely “access” the operations of the type.

Abstract Data Types and Existential Types

Given the existential type

$$CounterADT = \{\, \exists X, \{\text{new} : () \to X,\ \text{get} : X \to int,\ \text{inc} : X \to X\}$$

A valid implementation of $Counter$ is

$$\{Counter, c\} = \overbrace{\{*int, \{\text{new} = 1,\ \text{get} = \text{fn}(i : int)\{i\},\ \text{inc} = \text{fn}(i : int)\{i + 1\}\}\}}^{SC}\ \text{as}\ CounterADT$$

The concrete type (reference) used by the implementations of the operations, **bound to** X **in** $Counter$

Since the same implementation can inhabit different existential types, we introduce the keyword **as** to ascribe the intended existential type of the implementation.

$$Counter\ c_1 = c\,.\,\text{new}()$$

$$c\,.\,\text{get}(\ c\,.\,\text{inc}(\ c_1\ )\ )\ //\ 2$$

After we "opened" the existential, the type name is bound, i.e., we can only have one implementation of $Counter$ in the continuation of the program. ADTs guarantee that we can replace the assignment of a given implementation with any compliant one without risking to break our program.

N.B. the definition above of SC does not manage some internal state (or work with references) like the more complex example in Rust — which implements a hybrid form of ADTs

Modules

From a principled point of view, ADTs and modules are quite close—e.g., we can see ADTs as a degenerate case of a module carrying one ADT. However, depending on the language, ADTs and modules can provide different degrees of flexibility to the user, e.g., modules could express the visibility of data and operations in a much finer grade (the "permeability" of the capsule) than ADTs.

```
mod counter { pub trait Counter { … } }
mod sc {
 use crate::counter::Counter;
 pub struct SC { … }
 impl Counter for SC { … }
}
use crate::counter::Counter;
use crate::sc:SC;
fn use_counter<C>( c:&mut C ) where C: Counter{ … }
fn main(){ … }
```

Existential Objects

The kind of (existential) types we used for ADTs are not the only interpretation of existential types—for capturing information hiding and representation independence.

Indeed, while in ADTs the existential type hides (omits) the concrete type, when we use an implementation, our program binds the implementation to that type name in the program continuation, making it impossible to have multiple, interoperating implementations of the same abstract data type.

$\{Counter, c\} = \{*int, \{\text{new} = 1,\ \text{get} = \text{fn}(i : int)\{i\},\ \text{inc} = \text{fn}(i : int)\{i + 1\}\}\}$ as $CounterADT$

$\{ACounter, ac\} = \{\, * \,\{c : int\}, \{\text{new} = \{c : 1\},\ \text{get} = \text{fn}(i : \{c : int\})\{i\,.\,c\},\ \text{inc} = \text{fn}(i : \{c : int\})\{\{c : i\,.\,c + 1\}\}\}\}$ as $CounterADT$

$c1 = c\,.\,\text{new}()$

$c2 = ac\,.\,\text{new}()$

$ac\,.\,\text{inc}(c1)$ Type mismatch error

Existential Objects

(Existential) Objects provide an alternative view on ADTs that allows multiple implementations of the same existential type to interact.

To do this, we need existential-type definitions to live in programs like any other type, so that $Counter = \{\, \exists X, \ldots \}$ is a type in our typing system. Then, we can define implementations, called **objects**, as concrete types that keep their state (implementation) internal and carry with them their association with their existential type, for example

$Counter = \{\, \exists X, \{\text{state} : X, \text{methods} : \{\text{get} : X \to int, \text{inc} : X \to X\}\}\}$

$c1 = \{*int, \{\text{state} : 1,\ \text{methods} : \{\text{get}(int\ i)\{i\}, \text{inc}(int\ i)\{i+1\}\}\}\}$ as $Counter$

$c2 = \{\, * \{c : int\}, \{\text{state} : \{c : 1\},\ \text{methods} : \{\text{get}(\{c : int\}\ i)\{i\,.\,c\}, \text{inc}(\{c : int\}\ i)\{i\,.\,c + 1\}\}\}\}$ as $Counter$

$f(\ Counter\ \{*A, a\},\ Counter\ \{*B, b\}\ )\{$

$\{*A, \{state : a\,.\,methods\,.\,inc(b\,.\,methods\,.\,get(b)),\ methods : a\,.\,methods\}\}$ as $Counter$

$\}$

$f(c1,c2)$

Where the two implementations of $Counter$ can coexist, since their state is always “wrapped" within their implementation and the only way to read it is by means of the operations and state “changes” mean the creation of new objects (as seen in function f)

Objects vs ADTs

Summarising, intuitively, ADTs adopt an “open” view on existential types and the implementations that they induce. When we “import” an implementation (or, complementarily, when we “build” one) we immediately open it for usage. With ADTs, the values manipulated by client code (the programs that import the implementation) are the elements of the underlying representation type (e.g., in the counter example, integers). Thus, at run time, all the Counter values generated from the $CounterADT$ are just bare elements of the same internal representation type, and **there is a single implementation of the** counter **operations that works on** this **internal representation**.

The opposite happens with objects, where we keep “closed” an “imported” object as long as possible, up until we use one of its methods to access its internal state. In our example of the counter, each counter is a whole package—including the implementation of its state and of its operations. Hence, **each** counter **object carries its own representation type** together with its own set of operations that work for this representation type.

Objects vs ADTs

Hence, a relevant difference between ADTs and Objects is that, since each object chooses its own representation and carries its own operations, a single program can **freely intermix many different implementations of the same (existential) object type**.

We will see how this becomes particularly convenient in the presence of **subtyping** (and **inheritance**): we can define a single, general class of objects and then produce many different refinements, each with its own slightly (or completely) different representation. Since instances of these refined classes all share the same general type, they can be manipulated by the same generic code, stored together in lists, etc.

Objects

Let us now focus on the object-oriented paradigm and the peculiarities that determine it.

Of course, the main construct of any object-oriented language is that of an **object**: a **capsule** containing both **data** and **operations** to manipulate them, and which provides an **interface** to the outside world through which the object can be accessed.

In object-speak, operations are called **methods** and can access the data contained in the object, reachable via (instance) **variables** (or **fields**).

In the original definition of objects, one would execute an operation by **sending a message** to the object, which contains the name of the method to be executed and its possible parameters. However, (mainly for performance reasons) the most common form of operation invocation works via procedure invocation (hence, the large adoption of the term “invoke” for method execution) and reference passing.

Objects

The syntax of method invocation is not new (it is similar to that of ADTs and of records, in general) where `o.m( p1, p2, … )` reads "we invoke the method m, with parameters `p1, p2, …`, on the object o". Since objects (as ADTs) are essentially records, we can use a similar syntax to access their internal variables, e.g., `o.v` reads "we access variable v of object o".

In both cases, implementations of object-oriented languages equip objects with the capability to express in fine details the opacity of their capsule: we can express that operations and variables can be **visible everywhere**, some may be **visible only for some objects**, and some may be **completely private**, i.e. only available within the object itself.

Objects and Classes

Although it is conceptually permissible to have each object specify its implementation, it can become overwhelming and wasteful to specify many times the same implementation just to have different **instances** of fundamentally the same object (e.g., imagine we need three counters to keep track of some distinct events in our code, which then we want to sum).

This compelled the introduction of a new concept, called **class**, which allows us to specify a canvas or template implementation that contains, once and for all, the **variables and methods common to the same class** (hence, the name) **of objects**. Once we defined a class, to create new objects, we invoke a special function, usually called **new**, able to take the class and create—*instantiate*, in object-speak—a new object from it.

Classes

Hence, a class is a model for a set of objects: it establishes what their data are (how many, of what type, with what visibility) and fixes the name, signature, visibility, and implementation of its methods.

In a language with classes, each object 'belongs' to (at least) one class, in the sense that the structure of the object corresponds to the structure fixed by the class.

```
class Counter {
 private int x=1;
 public int get(){
  return x;
 }
 public void inc( int i ){
  x = x+i;
 }
}

Counter c1 = new Counter();
Counter c2 = new Counter();
Counter c3 = new Counter();
```

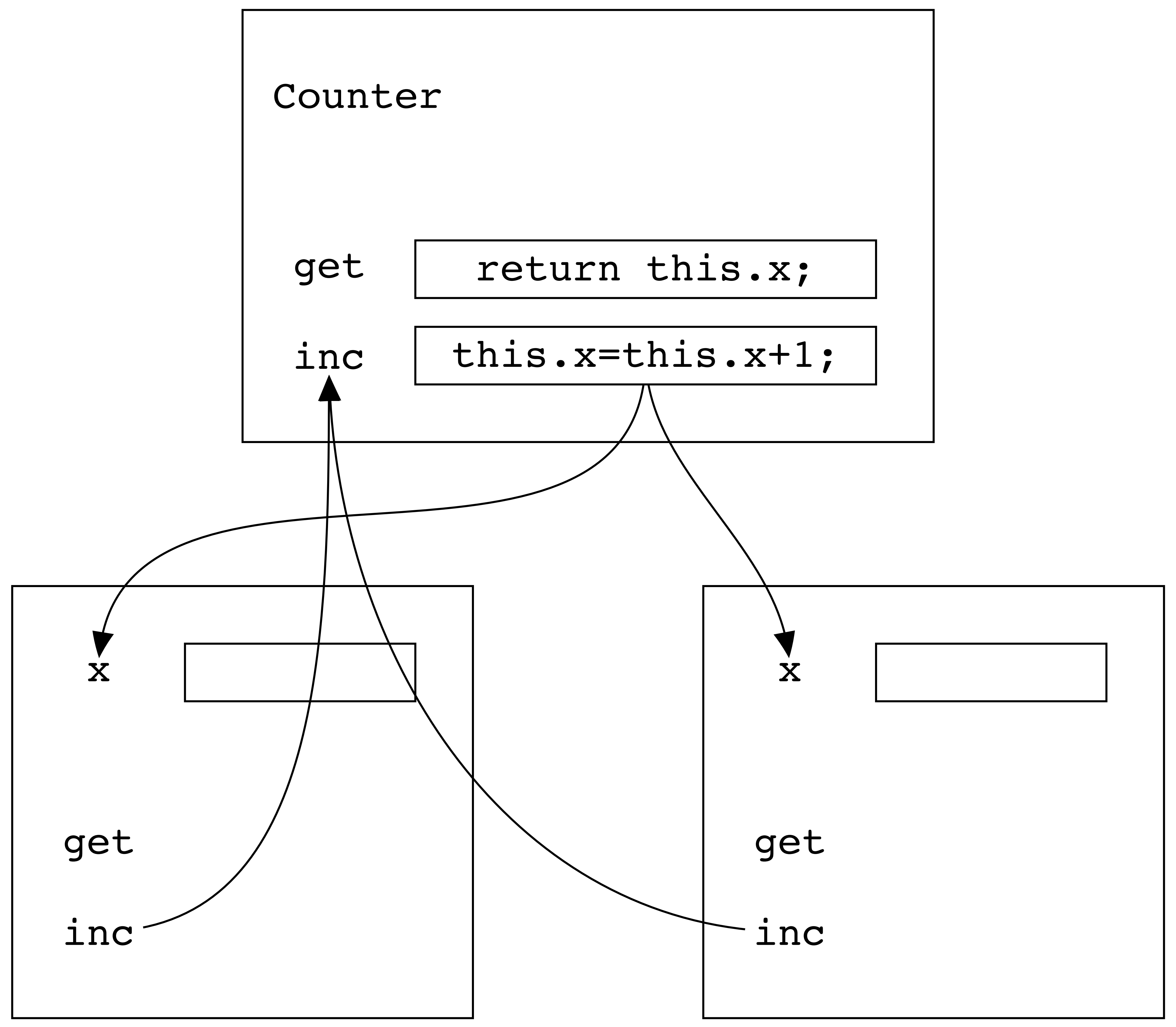
Classes

On the left, we see some **Java** code where (similarly to C++), a **class corresponds to a type** (more on this, later) and all instance objects of a class are values of that type.

However, different languages adopted different notions of classes. E.g., in Simula, a class is a procedure that returns a (pointer to a) stack frame (activation record) containing local variables and function definitions, while in Smalltalk a class is linguistically an object, which serves as a schema for the definition of the implementation of a set of objects.

```
class Counter {
 private int x=1;
 public int get(){
  return x;
 }
 public void inc( int i ){
  x = x+i;
 }
}

Counter c1 = new Counter();
Counter c2 = new Counter();
Counter c3 = new Counter();
```

Classes



In general, classes store the (only) code implementation of all its objects and when we invoke a method of an object, we actually reach out to that class-bound implementation, referenced by the object.

Of course, the method would not execute on the (inexistent) state of the class but rather on the state of the object.

Classes

```
class Counter {
 private int x=1;
 public int get(){
  return this.x;
 }
 public void inc( int i ){
  this.x = this.x+i;
 }
}

Counter c1 = new Counter();
Counter c2 = new Counter();
Counter c3 = new Counter();
```

What actually happens is that (as we saw in Rust, with the **self** parameter) object methods implicitly receive as parameter the object that invoked them, so that when the method body refers to instance variables, there is an implicit reference to the current object that is executing the method.

From a linguistic standpoint, the current object is usually denoted by a particular name, usually **self** or **this**.

Storing Objects in Memory

All object-oriented languages create objects dynamically, which usually reminds us of heap-allocated structures.

While this is the case for, e.g., Java, which allocates objects on the **heap**, access them **via references** (the language has no pointer type and variables of non-primitive objects are references to heap-stored objects), and uses **garbage collection** to manage their deallocation, languages like C++ allows one to specify the allocation and deallocation of objects on the stack.

In C++, when the language processes the declaration of a variable of type class, it creates and initialises an object of that type, bound to that variable and, thus, to that context.

Classes vs Prototypes

An alternative to classes are **prototypes**. This style also takes the name of **delegation**, as it hinges on the possibility for objects to delegate parts of their implementation to other objects.

Javascript (JS) is one of the most famous languages based on prototypes.

In prototypes, objects can delegate the definition of values and methods to another object, via their prototype property. These languages provide two ways to create new objects. One, called **ex-nihilo**, happens through some form of object literal (e.g., {...} in JS) and it assigns no prototype to the created object. The other, called **cloning**, creates (e.g., through the **new** keyword, in JS) an object by making a copy of an existing one, which is its prototype.

```
function Counter() {
 this.x = 1;
}
Counter.prototype.get=function(){
  return this.x;
}
Counter.prototype.inc=function(i){
  this.x = this.x+i;
}
c = new Counter();
c.inc( 1 );
c.get();
```

Prototype

Classes vs Prototypes

Prototypes are similar to classes in the sense that they serve as models for the structure and functioning of other objects. However, contrary to classes, **prototypes are (linguistically) ordinary objects**, possibly used as models.

Similarly to classes, prototypes can define common methods to their “children”, i.e., when we try to access some field or invoke a method of an object, if the object does not have that field or does not define that method, it delegates the action to its parent. If the parent owns that field or implements that method, then it performs the associated action and report back to its child the result. Contrarily, the chain of calls goes “up”, possibly until we reach the empty prototype and report an error.

Classes vs Prototypes

A practical difference between prototypes and classes stands on the flexibility of the former vs the safety guarantees given by the latter.

Indeed, a **prototype-based object can change its parent at runtime** (possibly completely changing its whole interface). One such behaviour is usually prevented in class-based languages. There, a class defines the interface of any object of that type and only relatively constrained associations of that object to other classes are allowed; e.g., we can cast the objects as the inhabitants of a class with fewer fields/methods, marked in the language as the "parent" of their class.

```
function Counter() { this.x = 1; }
Counter.prototype.get=function()
 { return this.x; }
Counter.prototype.inc=function(i)
 { this.x = this.x+i; }
function OtherCounter(){Counter.call(this);}
function Multiplier(){}
Multiplier.prototype.mult=function( i )
 { this.x = this.x*i; }
OtherCounter.prototype = Counter.prototype;
c = new OtherCounter();
c.inc( 1 );
Object.setPrototypeOf(c,Multiplier.prototype);
c.mult( 2 );
Object.setPrototypeOf(c, Counter.prototype);
c.get(); // 4
```

Encapsulation and interfaces

Encapsulation and information hiding are one of the cornerstones of ADTs, which also holds for object orientation: the language allows us to define an object by hiding parts of it (its data and/or methods).

Hence, we distinguish at least two views: the **private** and the **public** one. The private view is the most complete one, where all methods and fields are visible. The public one sees only those parts of the object that have been explicitly exposed in the definition of the object.

The public view of an object is usually called its **interface**, with the meaning mentioned for ADTs: the methods (and fields) that client code can use to interact with the value of a certain type.

Encapsulation 🏛

Subtypes

In OO, classes identify the set of objects that are its instances, i.e., we can consider **classes as** the **types** of those objects. Typed languages makes this relationship explicit: a class definition also introduces a type definition, whose values are the instances of the class.

As discussed, type systems usually come with a compatibility relation. In particular, we can see subtyping in OO as the compatibility relation $S <: T$ where the type associated with the class S is a subtype of the type associated with the class T whenever all client code expecting to work with objects of type T can work with objects of type S.

This concept, generally known as the Liskov substitution principle, is more formally specified as “Let $p(o)$ be a provable property for any object o of type T and let S be a subtype of T then, for any object o' of type S, $p(o')$ is provable.”

Subtypes

Concretely, the properties mentioned in the Liskov principle refer to the possibility to access in S fields available in T as well as being able to invoke in S methods available in T.

While in a **structural** type system this relation would boil down to a correspondence check between the elements of T with respect to those of S (as seen for records), the usual path taken by class-based languages is to adopt a **nominal** style, which helps avoiding possible accidental type equivalences.

Of course, this additional layer of safety comes with its limitations: the user now needs to declare the expected relations of subtyping among types, so that the language can check that the property above holds for subtypes.

Classes, interfaces, types, and subtypes

In principle, the direct relation we established earlier between types and classes is a bit misleading: types talk about structure and operations, while classes also define visibility constraints, hold state, and carry executable code.

This is where **interfaces** come into play, acting as (linguistic) bridge between classes and types.

Then, we take the definition of a class as the implicit definition of a companion interface from the public view of that class. In turn, the class **implements** that interface, establishing with it a relation of subtyping.

```
interface CounterInterface {
 int get();
 void inc( int i );
}

class MyCounter
 implements CounterInterface {
  private int x=1;
  public int get(){ … }
  public void inc( int i ){ … }
  private void doubleInc( int i ){ … }
}
```

inherently public

Classes, interfaces, types, and subtypes

In addition, interfaces allow us to provide clients with a **description of the "contract" our objects promise to fulfil**, without forcing us to provide their actual implementation.

This notion forms another pillar of object-oriented languages, which falls under the name of **abstraction** principle.

```
interface CounterInterface {
 int get();
 void inc( int i );
}

class CounterUser {
 void incCounter( CounterInterface c ){
  c.inc( 1 );
 }
}
```

Abstraction

Frequently coalesced with Encapsulation

Subtyping and Inheritance

The interface-to-class relation is not the only way we can specify subtyping in OO. Indeed, in principle, interface-to-class subtyping, e.g., `Counter` **`implements`** `CounterInterface`, entails two actions:

- we define a class, say `Counter`, which **`implements`** its associated companion interface, let us call it `Counter_Interface`;
- we declare that `Counter_Interface` is/must be a subtype of `CounterInterface`.

Hence, interface-to-class subtyping assumes a relation of interface-to-interface subtyping, which is usually referred as **extension**.

```
interface CounterInterface {
 int get();
 void inc( int i );
}

interface MultCounterInterface
 extends CounterInterface
{
 void mult( int i );
}
```

Subtyping and Inheritance

When we apply subtyping at the level of classes, i.e., class-to-class subtyping, we apply the idea of interface-to-interface **extension** to cover the state, the encapsulation constraints, and the method implementation of classes.

This relation takes the name of **inheritance**, since the subtype of the class “inherits” from the latter its definition of state (more on this later, with constructors), its encapsulation constraints, and its method implementations.

Inheritance

```
class Counter {
 int x=1;
 public int get(){ return this.x; }
 public void inc( int i )
 { this.x = this.x+i; }
}
class MyCounter extends Counter {
 private void doubleInc( int i ){…}
}
class MultCounter extends MyCounter {
 public void mult( int i ){ … }
}

MultCounter mc = new MultCounter();
mc.inc( 1 );
mc.mult( 2 );
mc.doubleInc( 3 );
```

subclass
superclass
subclass
superclass

Subtyping and Inheritance • Variable Shadowing

Variable shadowing indicates that a subclass can “mask” fields of its superclass by defining fields with the same name (but not necessarily the same types).

Variable shadowing comes from the standard block-level scoping rules, with the peculiarity that we consider the superclass as an outer block to the subclass, which can shadow any of the variables “inherited” from the outer one.

To avoid mistakes, Java is explicit on this matter: if a subclass wants to access a variable of its superclass, it must use the prefix **super** (akin to how **this** is a reference to the object itself).

```
class Counter {
 int x=1;
 int get(){ return x; }
 void inc( int i ){ x = x + i; }
}

class MultCounter extends Counter {
 int x=2;
 void multMult( int i ){
  super.x = super.x * this.x * i;
 }
}
MultCounter mc = new MultCounter();
mc.inc( 1 );
mc.multMult( 2 );
mc.get(); // 8
```

Subtyping and Inheritance • Method Overriding

As seen for interfaces, a class that **extends** another class can add, on top of those inherited from its supertype, new parts of state and new methods (along with their encapsulation constraints).

Another peculiarity of inheritance is the possibility to **override** inherited method implementations. Indeed, the principle of abstraction tells us that we can change method implementations as long as they respect interfaces.

```
class Counter {
 int x=1;
 int get(){ return x; }
 void inc( int i ){ x = x + i; }
}
class MultCounter extends Counter {
 void mult( int i ){
  super.x = super.x * i;
}
 @Override
 void inc( int i ){ mult( i ); }
}
Counter c = new MultCounter();
c.inc( 2 );
c.inc( 3 );
c.get(); // 6
```

Override annotation

🤔

Subtyping and Inheritance • Method Overriding and Variable Shadowing

Hence, one difference between **method overriding** and **variable shadowing** is that the **former is dynamically resolved,** while the latter is **statically resolved**.

In method overriding, it is **the actual class of the object** (from what class it was instantiated) **that determines to what method we shall dispatch the invocation.**

Contrarily, we solve **variable shadowing statically**, i.e., by either explicitly indicating what variable we want to access (**this**, **super**) or via type cast/coercion.

```
class Counter {
 int x=1;
 int get(){ return x; }
 void inc( int i ){ x = x + i; }
}
class MultCounter extends Counter {
  int x=1;
  void mult( int i ){
  super.x = super.x * i;
}
 @Override
 void inc( int i ){ mult( i ); }
}
MultCounter mc = new MultCounter();
Counter c = mc;
c.inc( 2 );
c.inc( 3 );
c.get();  // 6
mc.get(); // 6
c.x;      // 6
mc.x;     // 1
```

Override annotation

This is one of the reasons behind the practice of setter and getter methods

Subtyping and Inheritance • Refined Visibility

We have seen two notions of encapsulation visibility (also called visibility **modifiers**): **private** and **public**. Although this binary division covers the base case of transparent-vs-opaque encapsulation, there are cases where we might want to define encapsulation as semi-opaque (or -transparent), e.g., to allow subclasses to “see” methods and fields of their superclasses. This is the case covered by two additional visibility modifiers: **package** and **protected**.

The **package** case (the default one, in Java) extends the visibility of fields/methods of a class to all classes that belong in the same module of that class (remember the relation between ADTs, modules, and existential types).

The **protected** case extends the package one to also allow any subclass (in any module) to (internally) interact with the protected fields/methods of their superclass.

```
class Counter {
 int x=1;
 public int get(){ return this.x; }
 public void inc( int i )
 { this.x = this.x+i; }
}
class MyCounter extends Counter {
 package void doubleInc( int i ){…}
}
class MultCounter extends MyCounter {
 public int mult( int i ){ … }
}

MultCounter mc = new MultCounter();
mc.inc( 1 );
mc.mult( 2 );
mc.doubleInc( 3 ); 😀
```

subclass
superclass
subclass
superclass

Subtyping and Inheritance • Recap

It is important to keep in mind the difference between the inheritance and subtype relations:

- **subtypes** have to do **with the possibility of using an object in another context**: it is a relation between the **interfaces** of two classes;
- **inheritance** has to do with the **possibility of reusing code that manipulates an object**: it is a relation between the **implementations** of two classes.

Then, while these two relations are independent, they frequently intermingle, due to the conventions that languages impose, e.g., that class-to-class extension (inheritance) implies their related interface-to-interface extension (subtyping).

```
class Counter {
 int x=1;
 int get(){ … }
 void inc( int i ){ x = x + i; }
}

class MultCounter extends Counter {
 int x;
 int multMult( int i ){
  super.x = super.x * this.x * i;
 }
}

MultCounter mc = new MultCounter();
mc.inc( 1 );
mc.multMult( 2 );
```

Abstract Classes

In some cases, inheritance is too tight a relation, e.g., we might have a set of classes that implement 1) the same interface and 2) could share part of their implementation *but* they are not in an inheritance relation with each other, i.e., they are “siblings”.

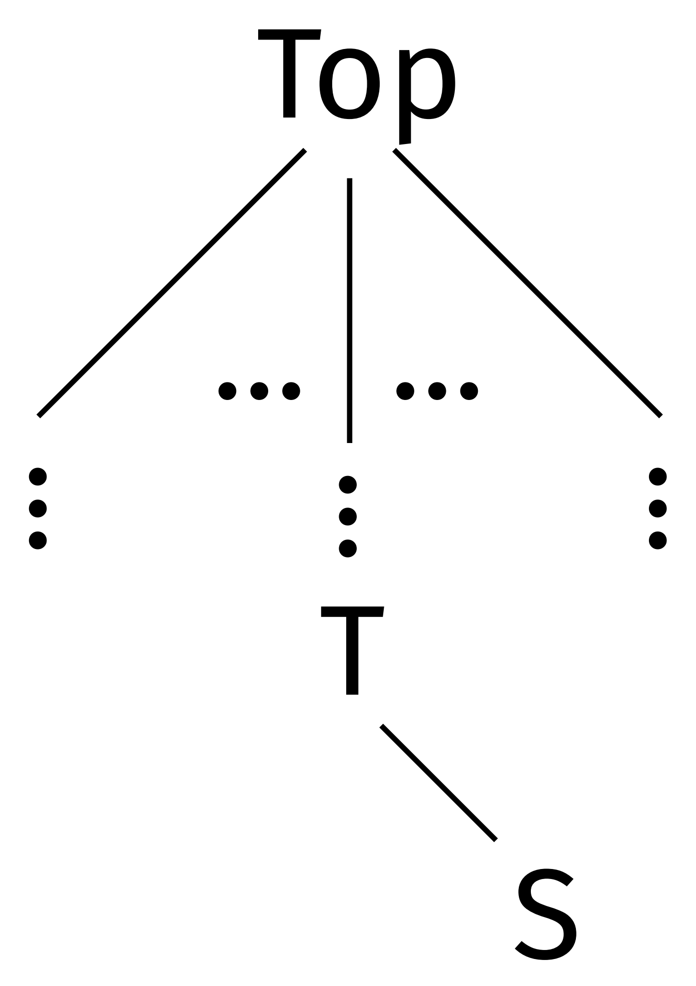
Abstract classes strike a middle ground between interfaces and classes, so that they can define fields and method implementations as well as leaving some methods as abstract, like interfaces, which its subclasses would need to implement/override.

```
abstract class AbstractCounter {
 int x=1;
 int get(){ return x; };
 abstract void inc( int i );
}

class Counter extends AbstractCounter {
 @Override
 void inc( int i ){
  super.x = super.x + i;
 }
}
```

Subtype Relation, Top

When we presented the subtype relation $<:$, we mentioned it being a(n antisymmetric, $S <: T \wedge T <: S \implies T = S$) **partial** (reflexive $T <: T$ and transitive $S <: T \wedge R <: S \implies R <: T$) **preorder**.

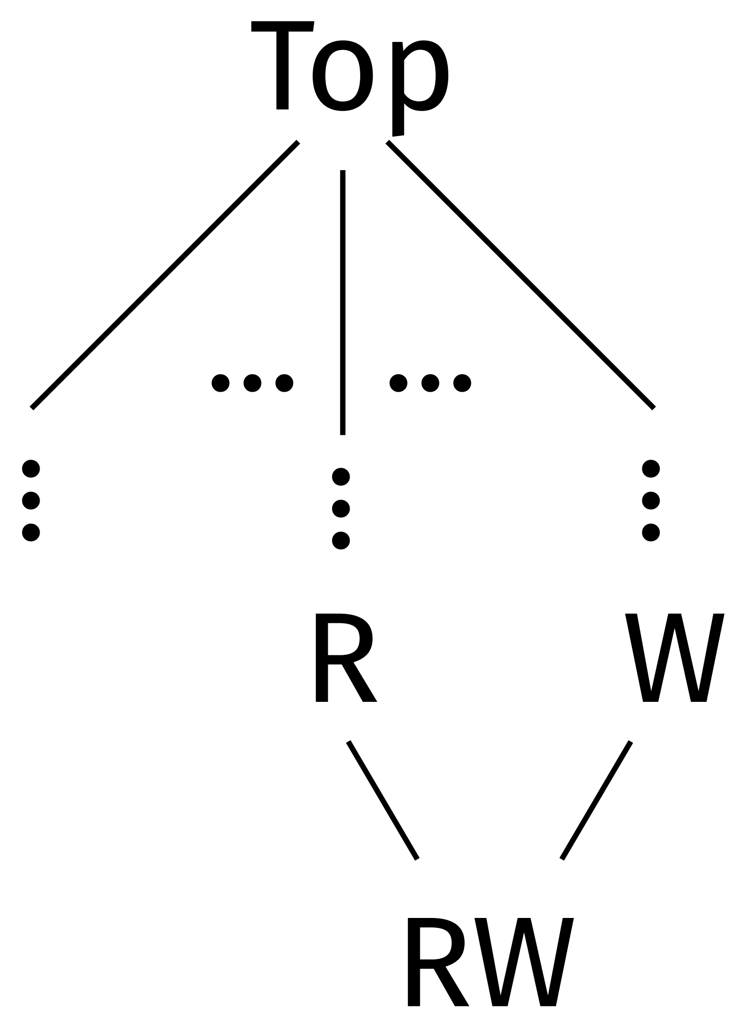
If we define cycles such as $T <: R, S <: T, R <: S$, antisymmetricity would reject them ($S <: T \wedge R <: S \implies R <: T$, but since $T \neq R$, it negates $T <: R \wedge R <: T \implies T = R$). Thus, the subtyping relation takes the form of a **directed acyclic graph** (DAG) among types.



Partial orders (and their related graphs) do not guarantee the presence of a single **maximal element**—here, the type that has no supertype and “fathers” (through transitivity) all other types, usually called **Top**. However, having Top in the type system is generally useful (e.g., to specify operations that accept values of any type) and many languages enforce the existence of Top, e.g., in Java, Top is **`Object`**, from which all other classes inherit (basic object-level methods, such as cloning and equality checking).

Subtype Relation, Intersection types

Since the subtyping relation describes a DAG, in principle, we are allowed to express something like $S <: T \wedge S <: R$, i.e., that S is both a subtype of type T and type R, with T and R unrelated. Visually, we are making the part of the DAG that involves T, R, and S converge on S.



In general, this kind of types take the name **intersection types**, written $S <: T \wedge R$, which comes from the observation that, when the values of T and R overlap, S **indicates the intersection of values that inhabit both T and R.**

On the other hand, if T and R do not overlap, S represents a kind of union (not to be mistaken for Union/Sum types) of the capabilities of the two types. E.g., a type RW which is both a subtype of the Reader (R) and the Writer (W) types. In Java, this is expressed, at the level of interfaces, with the **list** of types **an interface extends**.

```java
interface Reader {Readable read()}
interface Writer {void write(Writable w)}
interface RW extends Reader, Writer {}
```

Constructors

As mentioned, classes offer a way to define templates for the creation of objects and OO languages provide a special keyword, e.g., **new** in Java, to instantiate an object. However, there might be different *methods* for creating an object, e.g., we might also want to create `Counter` by setting x, instead of using the default starting value 1.

Constructors cater this need and we can see them as special methods of a class that can take in some parameters and return an instantiated object of that class.

```
class Counter {
 int x;
 public Counter(){
  this.x = 1;
 }
 public Counter( int i ){
  this.x = i;
 }
 public get(){
  return this.x;
 }
 public inc( int i ){
  this.x = this.x + i;
 }
}
```

Constructors

Both at the level of types and implementation-wise, objects are complex structures and their creation makes no exception: a) we need to **allocate the necessary memory** (on the heap or on the stack) and b) we need to **correctly initiate the data**.

Action b) is what the class and the constructor are there for: they define the code whose execution guarantees the creation of a correct instance of the class. This code becomes more **complex** the more features we use, e.g., with **inheritance** b) not only needs to initialise the internals of the object (fields, the pointers to methods, …) but it must also **link the data declared in superclasses**.

```
class Counter {
 int x;
 public Counter(){
  this.x = 1;
 }
 public Counter( int i ){
  this.x = i;
 }
 public get(){
  return this.x;
 }
 public inc( int i ){
  this.x = this.x + i;
 }
}
```

Constructors • Choosing a constructor (method)

On top of this, classes can provide different constructors among **which** the compiler/runtime has **to choose**.

In some languages (e.g., C++, Java), the name of the constructor coincides with the name of the class, and distinguishing among them follows the same rules for **overloaded methods** (solved statically, based on the number and types of the arguments). Other languages allow the programmer to freely choose the name of constructors, although they remain syntactically distinct from ordinary methods.

```
class Counter {
 int x;
 public Counter(){
  this.x = 1;
 }
 public Counter( int i ){
  this.x = i;
 }
 public get(){
  return this.x;
 }
 public inc( int i ){
  this.x = this.x + i;
 }
}
```

Constructors • Inheritance and Chaining Constructors

Another aspect is how and when to initialise the parts of an object that come from superclasses.

Some languages simply execute the constructor of the class whose instance is being created; if the programmer wishes to call the constructors of superclasses, he must do so explicitly.

Other languages (e.g., C++ and Java) enforce that the initialisation of an object first calls the constructor of the superclass (constructor chaining). Also in this case, the compiler/runtime must decide which of the possibly many constructors available of the superclass(es) to use.

```
class Counter {
 int x;
 public Counter( int i ){
  this.x = i;
 }
 public int get(){…}
 public void inc( int i ){…}
}
class MyCounter extends Counter {
 public MyCounter( int i ){
  super( i );
 }
}
```

Single vs Multiple Inheritance

In some languages a class may inherit from only one immediate superclass: the inheritance hierarchy is then a **tree** and the language is said to have **single inheritance**. This is the case of Java (n.b., single *inheritance*, not subtyping).

Other languages, however, allow a class to inherit methods from multiple superclasses; those languages are said to have **multiple inheritance** and the inheritance hierarchy is a **DAG** (like in the general case of subtyping).

The most part of languages support the simpler case of single inheritance, but some, e.g., C++ and Eiffel, support multiple inheritance. The reason is that multiple inheritance poses conceptual and implementation problems that did not yet find an elegant solution.

Problems (and solutions) to Multiple Inheritance

Conceptually, we have a problem with **name clashes**: a name clash occurs when a class inherits from two or more classes that provide the implementation of methods with the same signature.

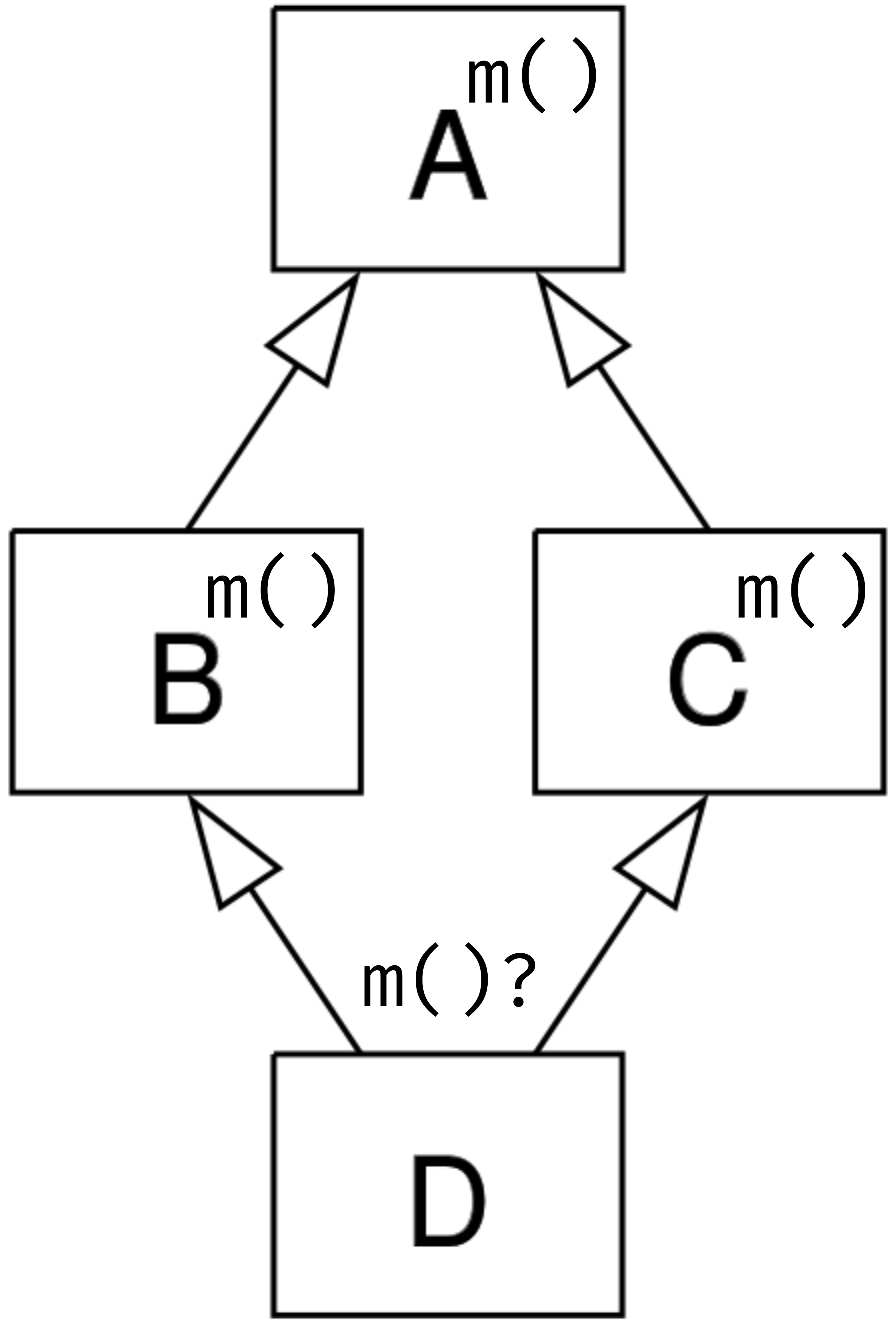
As mentioned, up to now we just found partial solutions: a) **syntactically forbid conflicts**; b) **ask the developer to explicitly fix any clash**, e.g., by appropriately qualifying each reference to the conflicting name (e.g., in C++, if `C` inherits from `A` and `B` where the method `m()` clashes, in the body of `C` the programmer needs to call `B::m()` or `A::m()`); c) **establish a convention** to resolve clashes (as in Scala mix-ins), e.g., by considering as “winning” the method of the left-most class that appears in the extension clause.

Problems (and solutions) to Multiple Inheritance

Practically, we have a **Deadly-Diamond-of-Death** kind of problem, from the diamond-like shape assumed by the schematic representation of the inheritance relation that emerges when a) two classes B and C inherit from A, and class D inherits from both B and C and b) there is a method in A that B and C have overridden, and D does not override it.

Which of the method implementations does D inherit?

We can follow one of the (excluding the first one) conceptual solutions mentioned before. While this solves the problem architecturally, we still have the practical problem of *efficiently* resolving clashes and run the correct implementation.



Dynamic Method Dispatch

Dynamic method dispatch, also called method **overriding**, is at the heart of the object-oriented paradigm: it is where **abstraction** and **inheritance** meet and give rise to one of the paradigmatic traits of object-orientation.
Conceptually, the mechanism is very simple: a subclass can redefine (**override**) a method implementation, so that, at runtime, the code run **depends on the type** of the object receiving the message. The stress on **types** is important: the dispatch is **dynamic** because, in general (e.g., in a method) we know the actual type of the object only at runtime.

Dynamic Dispatch

```
class Counter {
 int x=1;
 int get(){ … }
 void inc( int i ){ x = x + i; }
}

class MultCounter extends Counter {
 void mult( int i ){ … }
 @Override
 void inc( int i ){ mult( i ); }
}

Counter c = new MultCounter();
c.inc( 2 );
c.inc( 3 );
```

Overriding, Overloading, and Early/Late Binding

Method **overriding** is similar to **overloading**; indeed, they both resolve an ambiguous situation in which the same name can have several meanings.

The difference is summed up by the terms **early** and **late binding**. In early binding we use **static information** (of the type of the variables) to resolve the ambiguity and bind the name. Contrarily, in late binding information is available only at runtime (the types of the actual objects), which is when we can perform the binding of the name.

Dynamic Dispatch

```
class Counter {
 int x=1;
 int get(){ … }
 void inc( int i ){ x = x + i; }
}

class MultCounter extends Counter {
 void mult( int i ){ … }
 @Override
 void inc( int i ){ mult( i ); }
}

Counter c = new MultCounter();
c.inc( 2 );
c.inc( 3 );
```

Static methods

Static methods are more language-specific than dynamic dispatch—they do not define the OO paradigm—but some languages provide them as **a way to indicate** (and optimise for) **methods that the compiler can statically resolve**, because they are independent from instance (object) states/variables (they neglect `this`), and do not depend on the actual class of a given object.

An example of these are `static` methods in Java, which one accesses from the class that defines them, rather than from an instance (object) of that class. Since they are statically resolved, **static methods cannot be overridden**.

However, we can **shadow static methods** (similarly to variables) and pair them with subtyping. In this case, the resolution follows the subtyping relation—if we have two methods o, one taking a type and one its subtype, we disambiguate by applying the method specific to the type of the variable (not the object), as shown on the right.

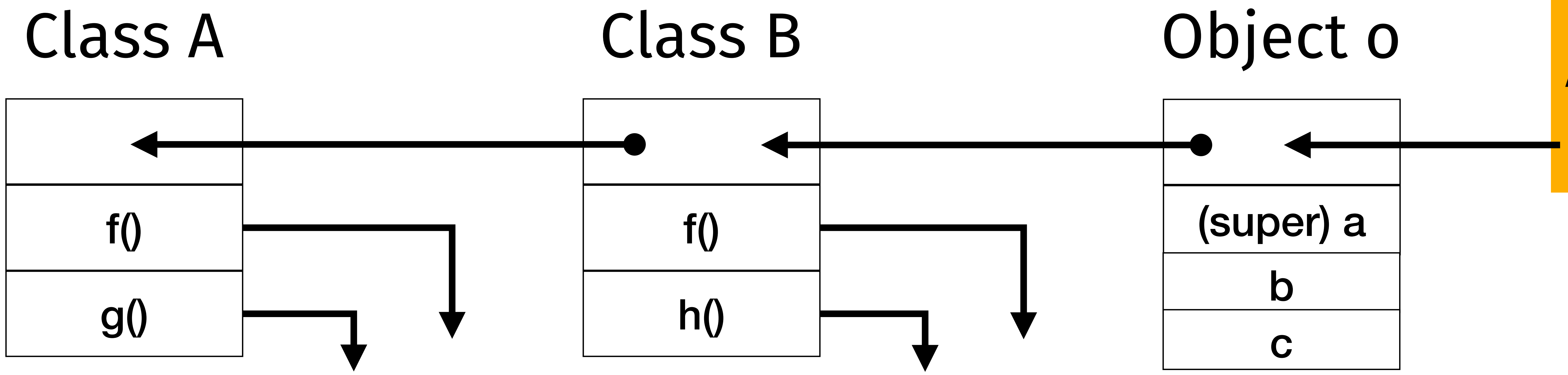
```
class Counter {
 int x=1;
 int get(){ … }
 void inc( int i ){ x = x + i; }
 static void inc( Counter c ){
  c.inc( 1 );
 }
}
class MultCounter extends Counter {
 static void inc( MultCounter c ){
  c.inc( 2 );
 }
}

Counter c = new MultCounter();
MultCounter.inc( c ); // Statically solved
c.inc( 1 );
c.get(); // 3
```

Implementation Aspects • Objects

Conceptually, classes are similar to records that define fields and (the code of) operations. This carries also to implementations, where we can represent an object as if it were a record holding the **fields** of the class of which it is an instance, **plus all those** that appear in its **superclasses**.

In case of **shadowing** (early binding), the object has fields corresponding to a different declaration (often the name used in the superclass is not accessible in the subclass, unless via some qualifier, e.g., `super`).

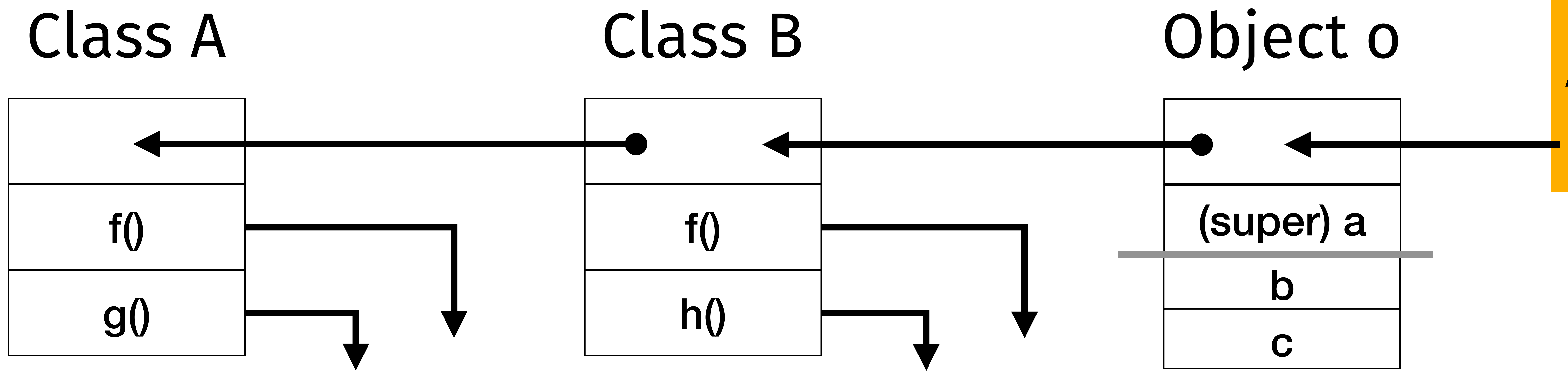
```
class A {
 int a;
 void f(){...}
 void g(){...}
}
class B extends A {
 int b; int c;
 void f(){...}
 void h(){...}
}
A o;
o = new B();
```



Implementation Aspects • Objects

In statically-typed languages this representation allows a simple **implementation of subtype compatibility** (in case of single inheritance): since we statically know the offset (position) of each variable, we can resolve static references by calculating the offset of the starting block belonging to a given superclass and calculating the offset to the referenced field from there—e.g., to find `o.c`, we know we need to start after the offset from the fields of class A and then follow the type/order of fields in B.

```
class A {
 int a;
 void f(){...}
 void g(){...}
}
class B extends A {
 int b; int c;
 void f(){...}
 void h(){...}
}
A o;
o = new B();
```



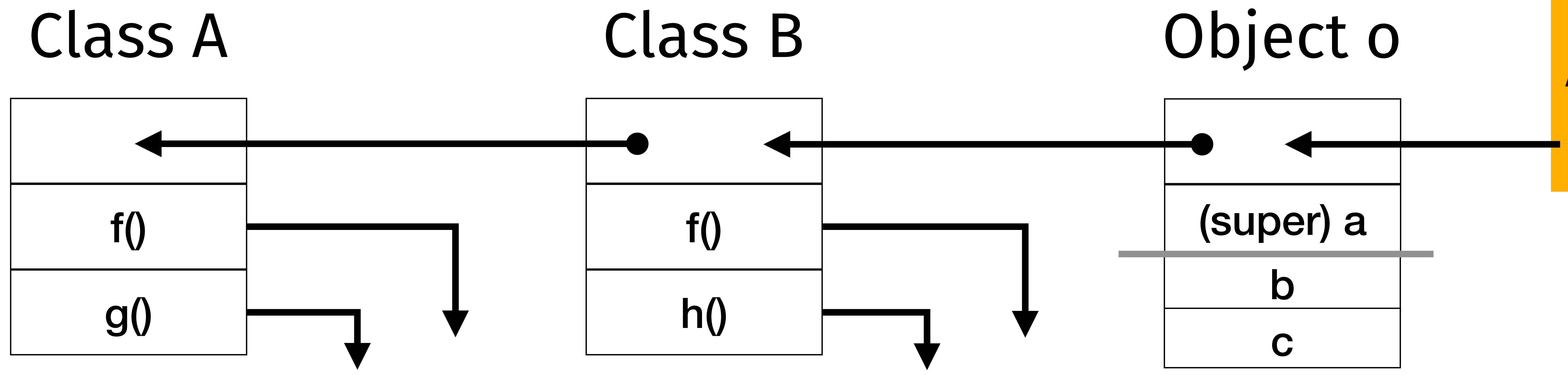
Implementation Aspects • Classes and Inheritance

The simplest and most intuitive implementation of classes and inheritance is by means of a **concatenated list**, where each element: a) represents a class and contains (pointers to) the implementation of all methods explicitly defined or redefined in that class and b) point its immediate superclass.

To implement dynamic method dispatch, we use the pointer from an object to its class to check whether it contains an implementation for that method: if it does, we execute the code pointed there, otherwise we follow the pointer up to the superclass of that class and so forth.

While this implementation is simple, it is also quite inefficient, since late binding implies the linear visit of the hierarchy of classes.

```
class A {
 int a;
 void f(){...}
 void g(){...}
}
class B extends A {
 int b; int c;
 void f(){...}
 void h(){...}
}
A o;
o = new B();
```



Implementation Aspects • Late *self* binding

Executing a method is similar to running a function, where we load on the stack the local variables, the parameters, and the other information for its execution. However, unlike functions, **methods must** also **access** the **instance variables** of the object on which they are invoked, whose address we know only at runtime.

An inefficient solution would be referencing the object (`this`) in the stack frame of the method and then perform a double lookup to find the object in memory and then access the instance fields thereafter.

On the contrary, we can avoid to load in the stack frame the reference to `this` and the double lookup by using the **static knowledge on the structure of the object**, given by its class: we (the compiler) define the **access** to **instance fields** not as an offset from the stack frame (as it happens with local variables/parameters) but **as** the **offset** given by **the address of the current object** (`this`) **plus** the (specific) **offset of each field**, as declared by the class of the object.

Implementation Aspects • Single inheritance

Given a static type system, we can improve the linear-time, chained-list implementation of single-inheritance method selection into a constant-time one.

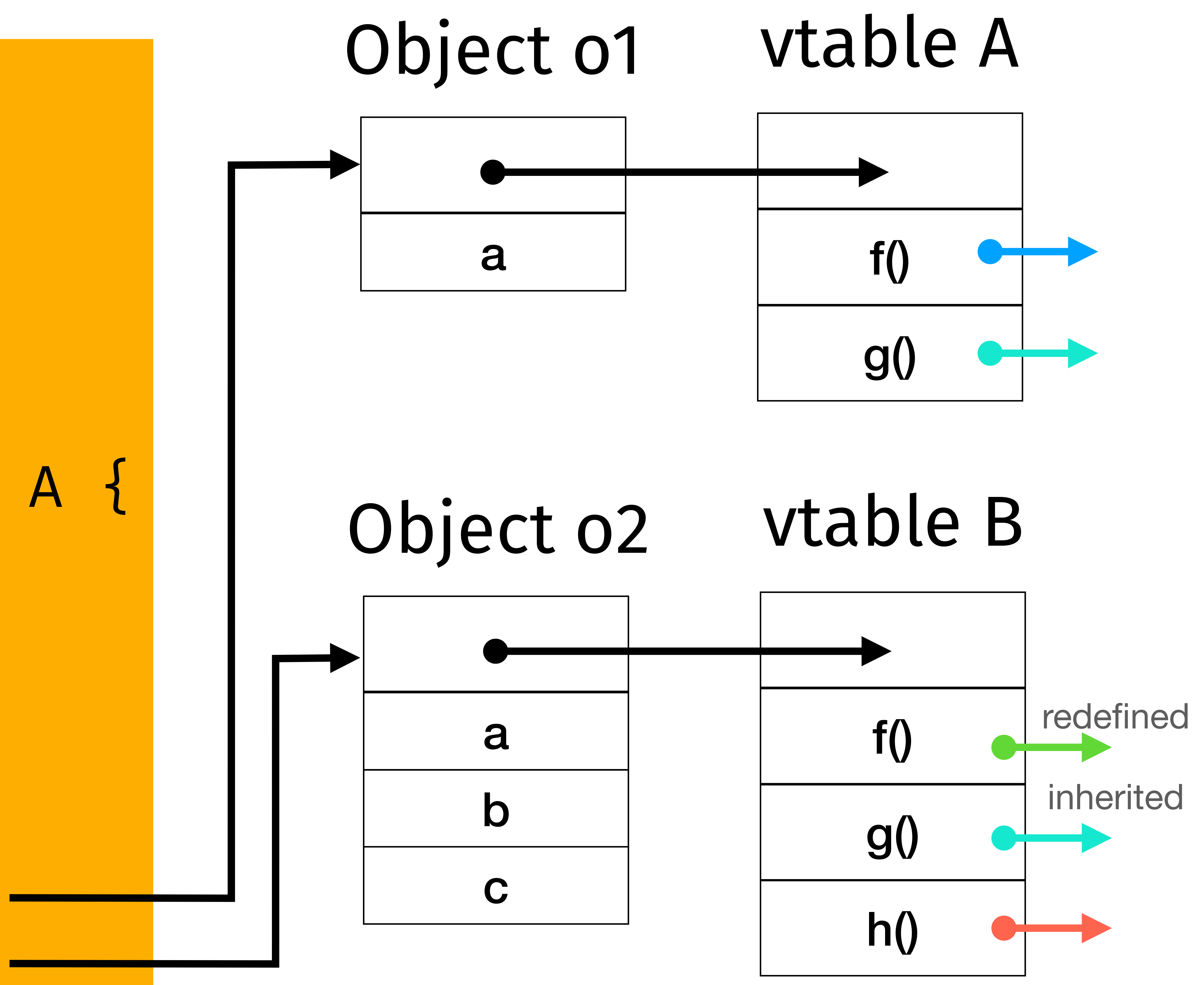
Indeed, if types are static, objects have a finite, static (compile-time) set of methods, which correspond to those found in their class descriptor — which contains both the methods explicitly defined/redefined in the class and all those inherited from the superclasses.

This data structure usually takes the name of **vtable** (from C++, standing for **virtual function table**).

Implementation Aspects • Single inheritance

With vtables, **each class definition corresponds to a vtable**, and all instances of that class share the same vtable. When we define a subclass B of class A, we build the vtable of B by making a copy of the vtable of A, replacing in this copy all the methods redefined in B, and then **adding at the bottom** of the vtable the new methods defined in B.

```
class A {
 int a;
 void f(){...}
 void g(){...}
}
class B extends A {
 int b; int c;
 void f(){...}
 void h(){...}
}
A o1 = new A();
A o2 = new B();
```



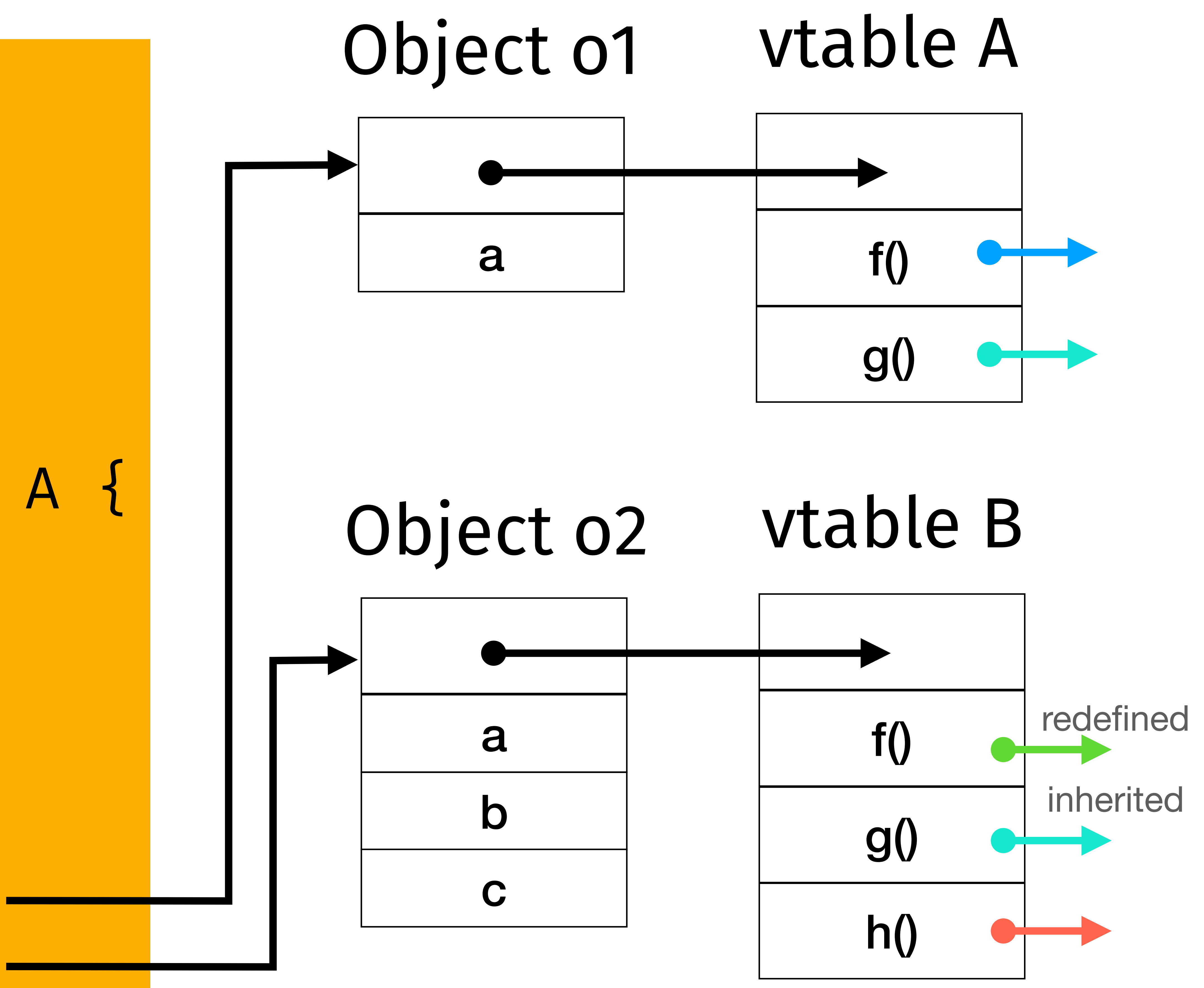
Implementation Aspects • Single inheritance

In this way the invocation of a method:

A) occurs at the (constant) price of two indirect accesses, since it is statically known the offset of each method within the vtable and

B) takes into account that an object can be accessed as one if its superclasses; e.g., when invoking method `f`, the compiler calculates an offset for that method that remains the same whether `f` is invoked on an object of class `A` or `B`, although, in the vtable, the same address corresponds to different implementations.

```
class A {
 int a;
 void f(){...}
 void g(){...}
}
class B extends A {
 int b; int c;
 void f(){...}
 void h(){...}
}
A o1 = new A();
A o2 = new B();
```



Implementation Aspects • Fragile base class

Vtables for simple inheritance are very efficient, since most information is statically determined. However, the late binding of `this` (`self`) is a source of problems in a context known as the **fragile base class** (or superclass) **problem**.

Indeed, the top-down propagation of changes imposed by class hierarchy (from super- to subclasses) breaks compositionality: the only way to detect if some changes in a superclass caused incompatibles in subclasses entails to consider the entire inheritance hierarchy. Conceptually, modularisation makes this check impossible, since the writer of the superclass does not have access to all possible subclasses.

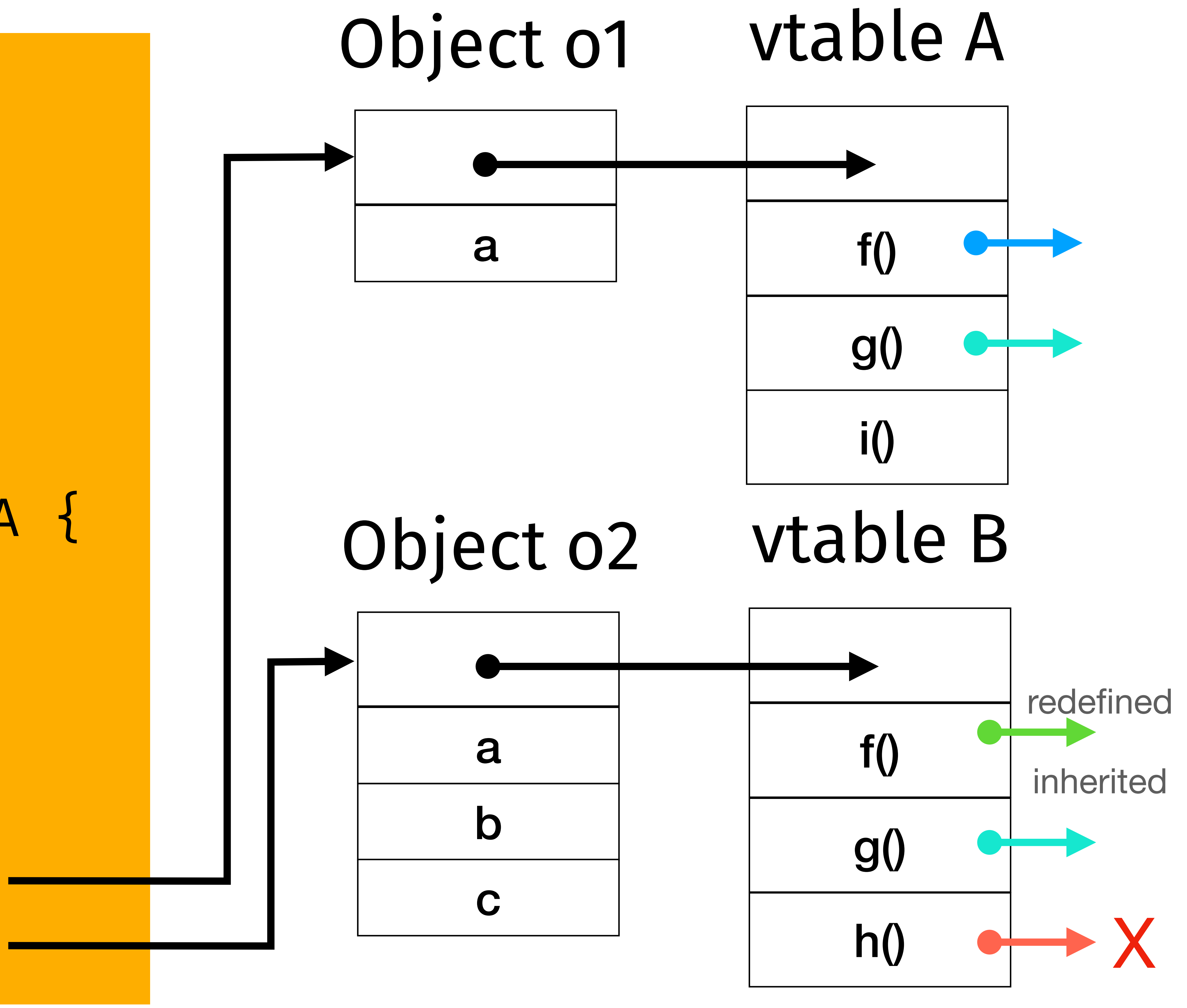
From the architectural point of view, this means that some subclass exploited parts of the superclass that have been changed. This is a software engineering problem which can be solved by limiting inheritance in favour of subtyping.

Implementation Aspects • Fragile base class

From the implementation point of view the failure of the subclass depends only on how the compiler represented the hierarchy in memory. This second case takes the name of **fragile binary interface problem**.

E.g., in the example, we add the method `i` to A. This forces us to recompile also B to add to its vtable the new method, or we risk malfunctions, e.g., executing `h` in place of `i`.

```
class A {
 int a;
 void f(){...}
 void g(){...}
 void i(){...}
}
class B extends A {
 int b; int c;
 void f(){...}
 void h(){...}
}
A o1 = new A();
A o2 = new B();
```



Implementation Aspects • Dynamic Method Dispatch (JVM)

The way in which the JVM implements Dynamic Method Dispatch overcomes this problem, by dynamically (and efficiently) computing the offset of the methods in the vtable (but also of the instance variables in the object representation).

Simplifying, Java compiles classes separately from each other: each class gives rise to a file that the virtual machine **dynamically loads** when the executing program makes a reference to that class.

This file contains a table of symbols (the **constant pool**) used in the class itself: instance variables, public and private methods, methods and fields of other classes used in the method body, names of other classes used in the class body, etc.

Implementation Aspects • Dynamic Method Dispatch (JVM)

In the compiled code, each instance variable and method name has information associated with it, including the type of the names and the class where they are defined.

To save space, whenever the source code uses a name, the JVM intermediate representation uses the **index** of that name in the constant pool (and not the name itself). However, these indexes are not only useful for representation compactness.

When, at runtime, a **name** is **referred for the first time** (through its index), this **is solved**: the virtual machine loads the necessary classes (for example those where the name is introduced) using the information of the constant pool and it checks the constraints on types and visibility (e.g., that the invoked method really exists in the referred class, that it is not private, etc.) and then it **rewrites the running code** to **replace the lookup instructions with instructions for the direct execution of the code loaded in memory**.

Implementation Aspects • Dynamic Method Dispatch (JVM)

We can think of the representation of methods in a class descriptor as similar to a vtable: the table for a subclass starts with a copy of that of the superclass and we replace the overridden methods with the ones redefined by the class. However, we do not compute offsets right away, but rather follow these four invocation modalities (related to their distinct instructions in the bytecode):

- **`invokestatic`**: the method is **static** and it cannot refer to `this`;
- **`invokevirtual`**: the method must be **selected dynamically** (so-called "virtual" methods);
- **`invokespecial`**: the method must be **selected dynamically and is “special”**, i.e., they are constructors or invoked on `this` (e.g., private methods) or `super`;
- **`invokeinterface`**: we call an interface method, which some objects implement.

Implementation Aspects • Dynamic Method Dispatch (JVM)

invokestatic has the simplest resolution: since static methods cannot be overridden and do not reference instance variables, we just statically bind the definition of the related class.

invokespecial follows the same path of invokestatic, with the possible check that `super`/`this` exists (and its binding).

invokevirtual deals with method overriding and resolves it via vtable lookups, optimising visits with index lookups, i.e., calculated on the base that overridden methods have the same signature of superclasses. Starting from the subclass, we lookup the index in the vtable of the class and check we found the method we were looking for. If so, we stop and resolve the index, otherwise, we continue following the pointer to the superclass.

invokeinterface since we ignore which class implements the interface method, we inspect the class of the object to determine a) if that class actually implements the interface, and b) where that interface's methods are recorded within that particular class. Since we do not assume a fixed scheme (which would introduce the *fragile base-class problem*) we need to search through the list of implemented interfaces by the class. Once we find the interface, we can proceed in a more direct way: from the interface, we calculate an **itable** that represents the fixed schema common to all portions of classes that implement the target interface, and we use the offset from the itable to find the method and proceed as seen for dynamic/virtual invocations.

invokedynamic, (for completeness) used since Java 8, the instruction adds more flexibility to the dynamic dispatch mechanisms of the JVM and is mainly used in the implementation of Java lambda expressions.

Parametric Polymorphism and Generics

As seen, Java supports parametric polymorphism with the syntax, e.g., `Set< T >`.

Specifically, Java adopts the nomenclature **generics** [1] to indicate the inclusion of this feature to support generic programming—Java is not alone here, e.g., C#, F#, Python, Go, Rust, Swift, and TypeScript all adopt the same term. The distinction with what ML and Haskell call parametric polymorphism is thin, but one lexical difference is that the latter intend polymorphism as *implicit*, e.g., `OCaml`

```
let max x y = if x > y then x else y;; where max: ‘a -> ‘a -> ‘a
```

while generics assume explicit indication of type parameters, e.g., Java

```
<T> max ( T x, T y ) { return x > y ? x : y; }
```

C++ also support a similar concept as that of generics with **templates**.

[1] Bracha, G., Odersky, M., Stoutamire, D., & Wadler, P. (1998). Making the future safe for the past: Adding genericity to the Java programming language.

Generics and Type Erasure

When writing `Set<T>` we see we can use generics to parameterise entire classes, but how do we generate a parametric version of `Set`, able to “work” with any type parameter?

As with, e.g., dynamic dispatch, generics can be implemented in several ways. E.g., C++ implements them statically, where the complier creates a separate copy of the code for every used instance.

Thanks to **type erasure**, Java makes all instances of a given generic class share the same code. What happens at compilation is that, if the Java type checker validates the use of generics, the compiler proceeds by erasing all type parameter from the code (so that Set<T> becomes the “**raw**” type `Set`) and all objects of the generic class become instances of the Top type Object [1]—since the type checker validated the program, the compiler does not need to add casts. This “trick” allowed Java 5 to introduce generics without breaking *compatibility* with previous versions of the language, VM implementations, and libraries.

[1] Which excludes the use of basic Java types that are not subtypes of Object

Type parameter erasure

While erasing generics in Java has many benefits, it also introduced shortcomings.

The most notable is we cannot invoke `new T( )`, (T type parameter), since the compiler does not know what object to create. Similarly, Java's reflection mechanism (**instanceof**) cannot distinguish between `Set<Integer>` and `Set<String>`, since at runtime they both coalesce to the row type Set — although there are techniques, generally referred as **reification** (as the complementary of type abstraction obtained through erasure), where, e.g., we have the class carry the reified type/class (also called **witness**) of the type parameter.

```
class Box< T > {
 T c;
 Box( T c ){ this.c = c; }
}
class WBox< T > extends Box< T >{
 Class< T > klass;
 WBox( T c, Class< T > klass ) {
  super( c );
  this.klass = klass;
 }
}

Box< String > b1 = new Box<>( "a" );
Box< Integer > b2 = new Box<>( 1 );
WBox< String > wb1 =
  new WBox<>( "a", String.class );
WBox< Integer > wb2 =
  new WBox<>( 1, Integer.class );
```

Java Generics • Wildcards

To be able to express variance annotations on generics, Java introduced the wildcard **`?`** as a special kind of type argument that expresses that T<?> is a supertype of any (type) application of the generic type T.

? can be refined to indicate co(ntra)variance of subtyping, i.e, the covariant case **`T<? extends S>`** allows the use of S and its subtypes while the contravariant case **`T<? super S>`** allows the use of S and all its supertypes (see the examples on covariance and contravariance on bounded parametric polymorphism).