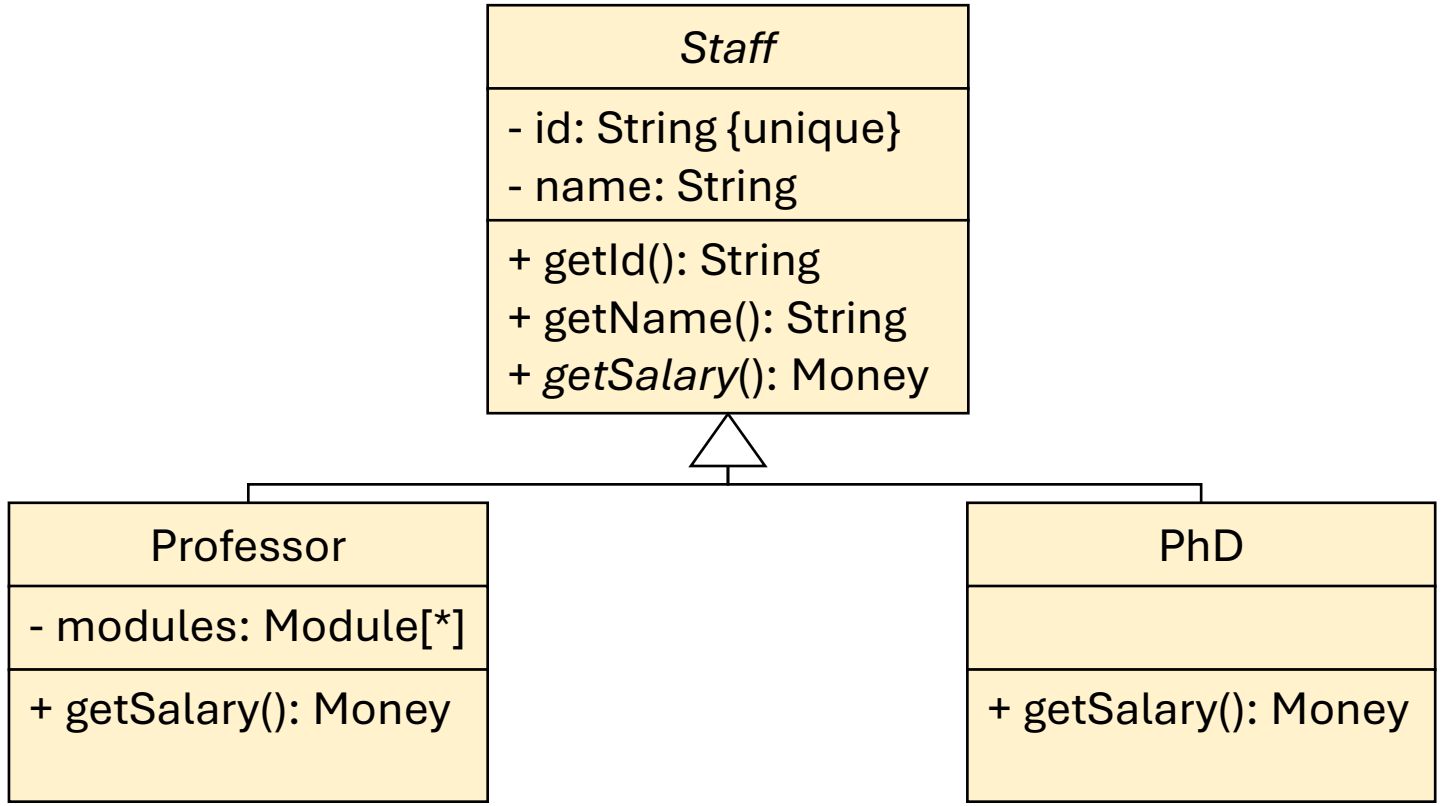
UNIVERSITY OF COLOGNE
Software & System Engineering / Software Technology
Abteilung Informatik, Department Mathematik / Informatik

Object-Oriented Software Engineering

Types and Dynamic Binding

Adrian Bajraktari | bajraktari@cs.uni-koeln.de | SoSe 2024 | 17.04.2024

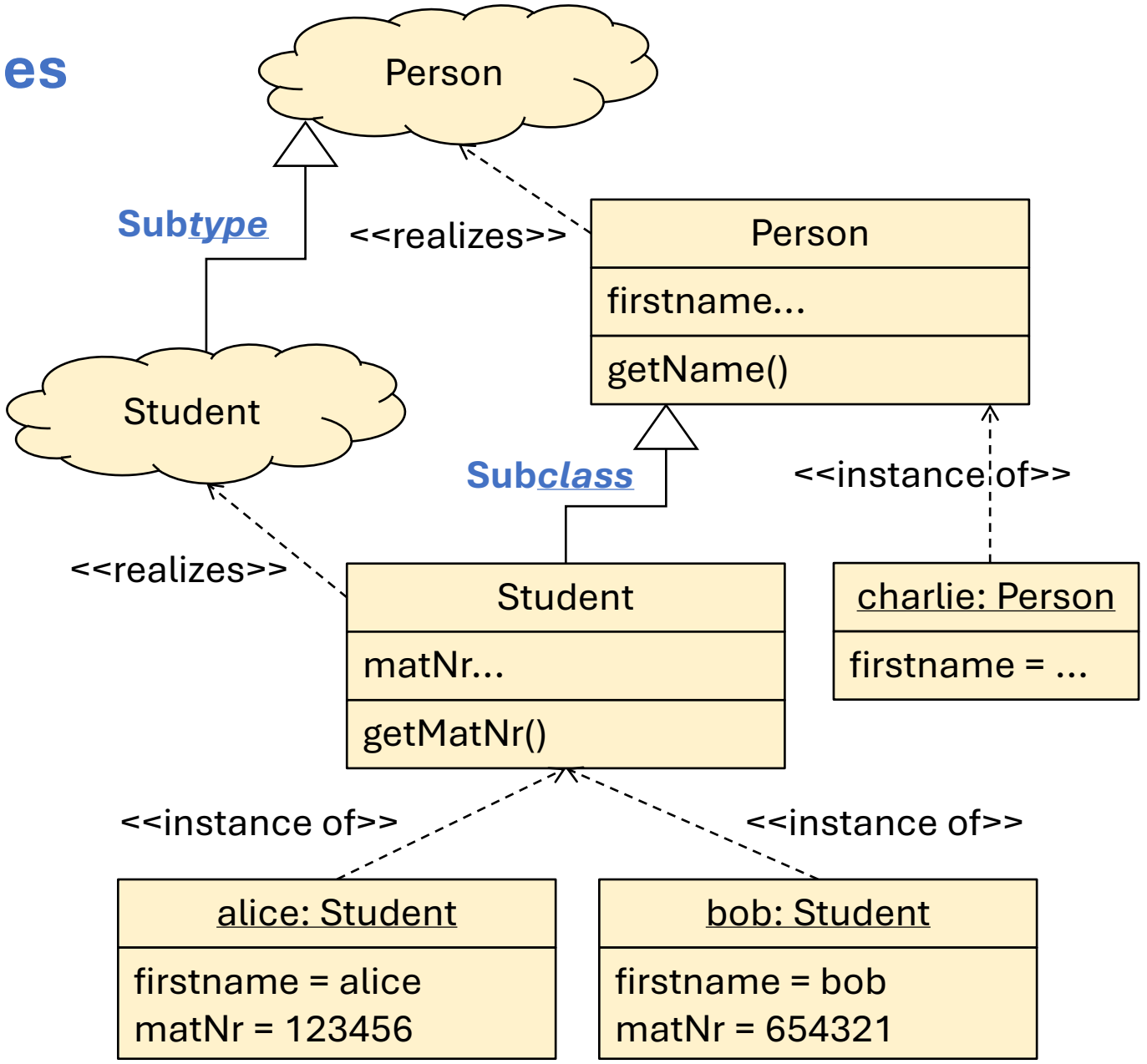


Types and Subtypes

Types, Classes and Instances

Relation between Types and Classes

A class is a concrete, technical realization of a type.



Languages without Static Typing: Python

Static Typing in Python (Type Hints)

In Python, static typing was introduced in 3.5 and is neither checked by the compiler nor mandatory by default.

```
studentAlice = Student(24, alice, 2392381)
studentAlice.setGrade(oose, 1.0)
```

```
class Student:
   def __init__(self, age, name, matNr):
      self.age = age
      self.name = name
      self.matNr = matNr
      self.grades = []

   def setGrade(self, module, grade):
      self.grades.add({module: grade})
```

Static vs. Dynamic Type

Static Type

The static type is the type of an expression used in its declaration, thus also called declared type. The static type lets the compiler know which member accesses are allowed, independent of the dynamic type.

Dynamic Type

The dynamic type is the type of an expression at runtime, thus also called runtime type.

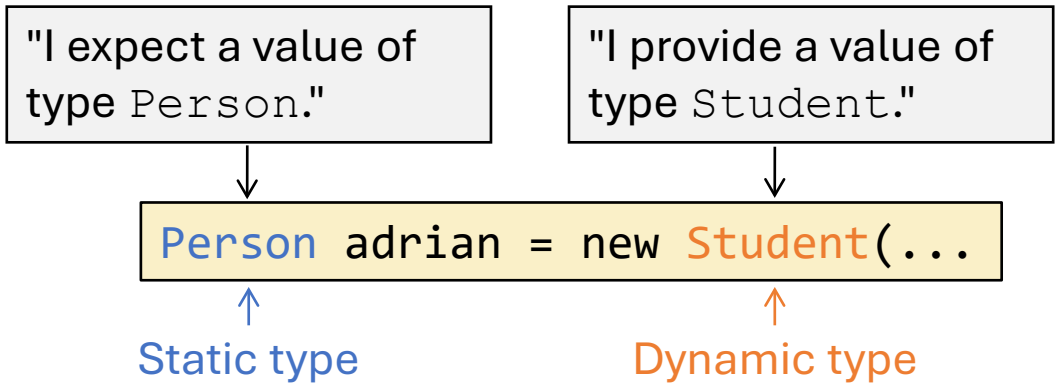
Structural Subtyping

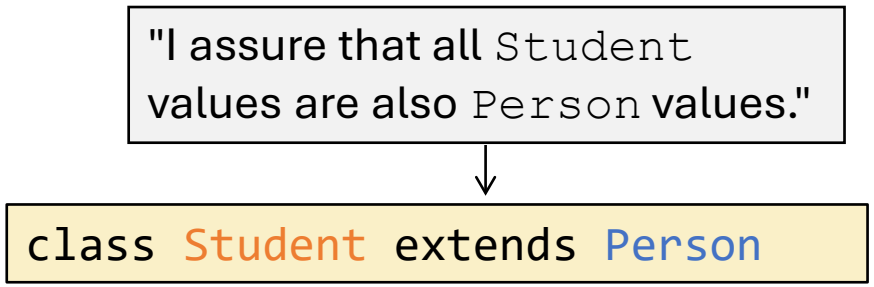
`B` contains at least the same members as `A`.

- Type inference: The compiler derives a type based on object members.
- Duck typing: The runtime system assumes a type based on whether member accesses succeed.

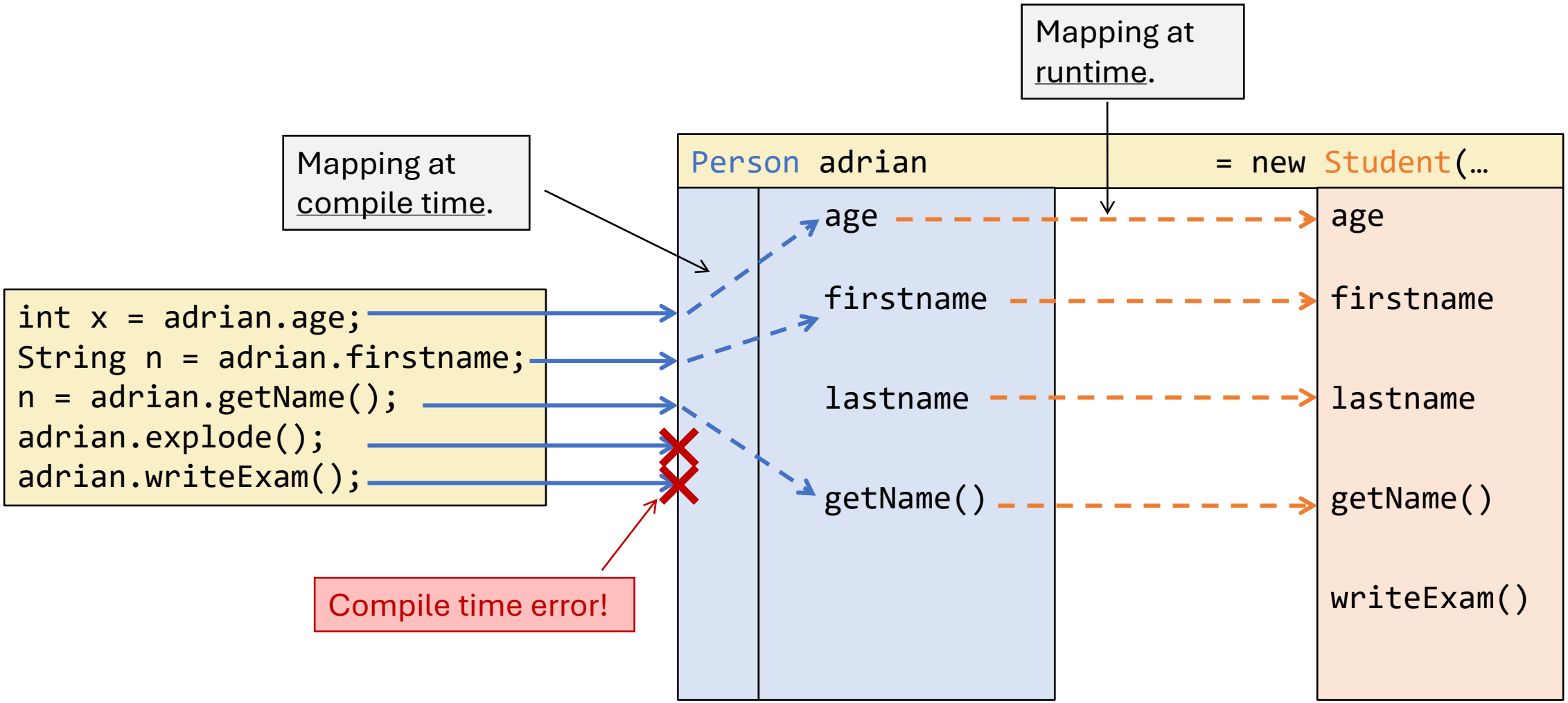
Nominal Subtyping

`B` is explicitly declared as subtype of `A`.
(This is what Java, C++ use and Python, JS support)



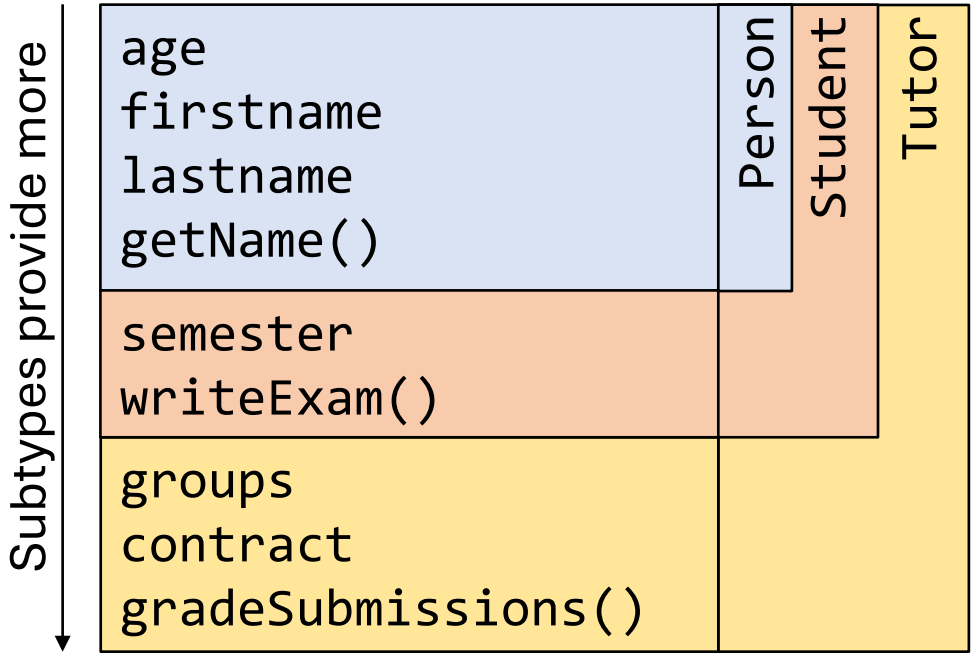


Metaphor: Static Type as Guard with different Dynamic Type

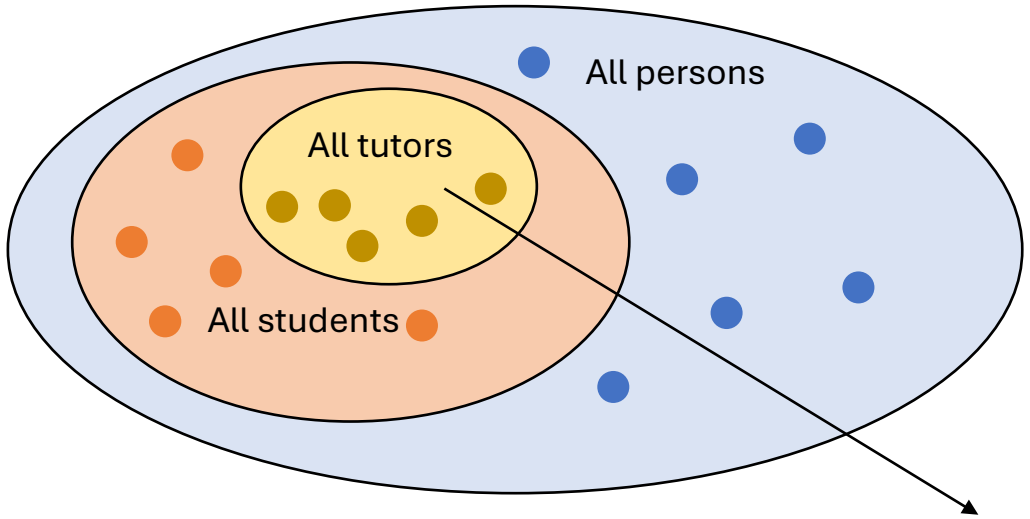


Subtypes: Subset View (Single Inheritance!)

Subtypes provide more details (fields, methods).



Because subtypes define more details, there are fewer instances that provide all of them.



Subtypes have fewer instances

Class Inheritance

Class Inheritance

Class inheritance defines that a class inherits all instance members of another class.

Implementation

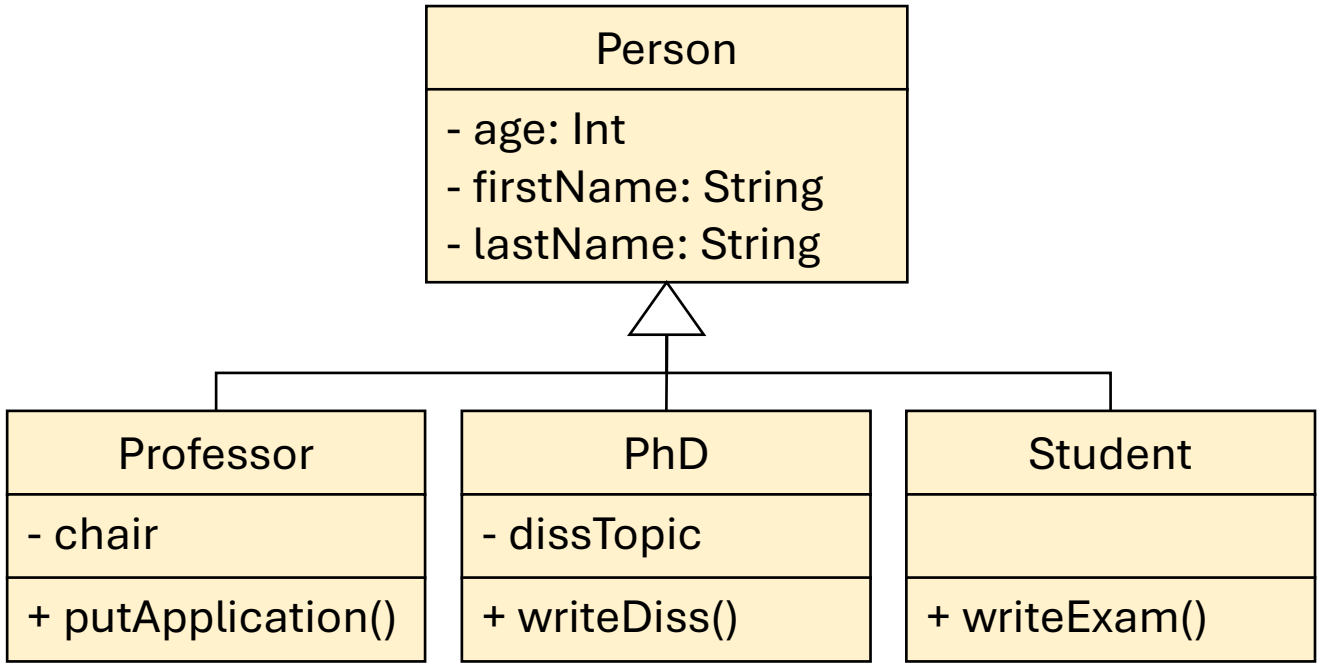
- Java, JavaScript: `class Student extends Person`
- C++: `class Student: public Person`
- Python: `class Student(Person)`

Hints for Modeling

Class inheritance combines subtyping and code reuse.

Good modeling only uses class inheritance if a conceptual specialization exists.

Otherwise: Forwarding or delegation!



Multiple Inheritance

Multiple Inheritance

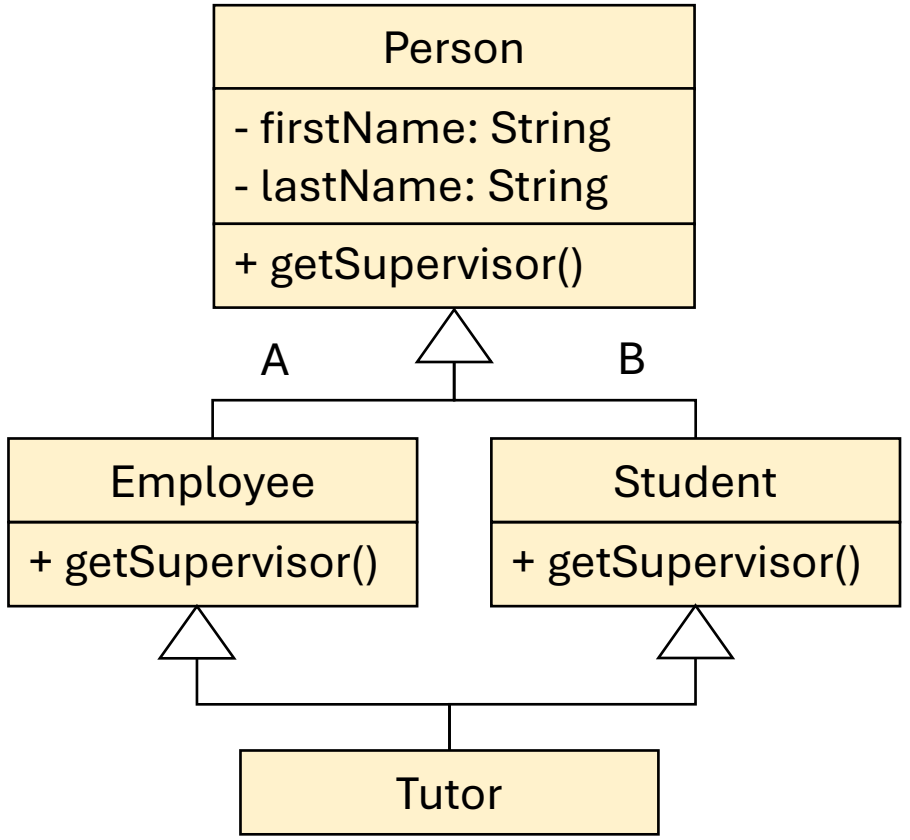
In multiple inheritance a class inherits from two or more classes. (Or in object-based programming: an object inherits from two or more objects)

Diamond of Death Problem

What if a class inherits the same attribute from two different inheritance paths?

Mitigations in different Languages

- Java*, JavaScript: forbidden.
  - In Java interfaces with default methods: forces override with explicit specification of which implementation to use.
- C++: declaring both links A and B to virtual inheritance.
- Eiffel: Renaming.
- Go: Identifies DoD at compile time. Can refer via Tutor.Employee.X / Tutor.Student.X.
- Python: Order of specification determines precedence.



Hints for Modeling

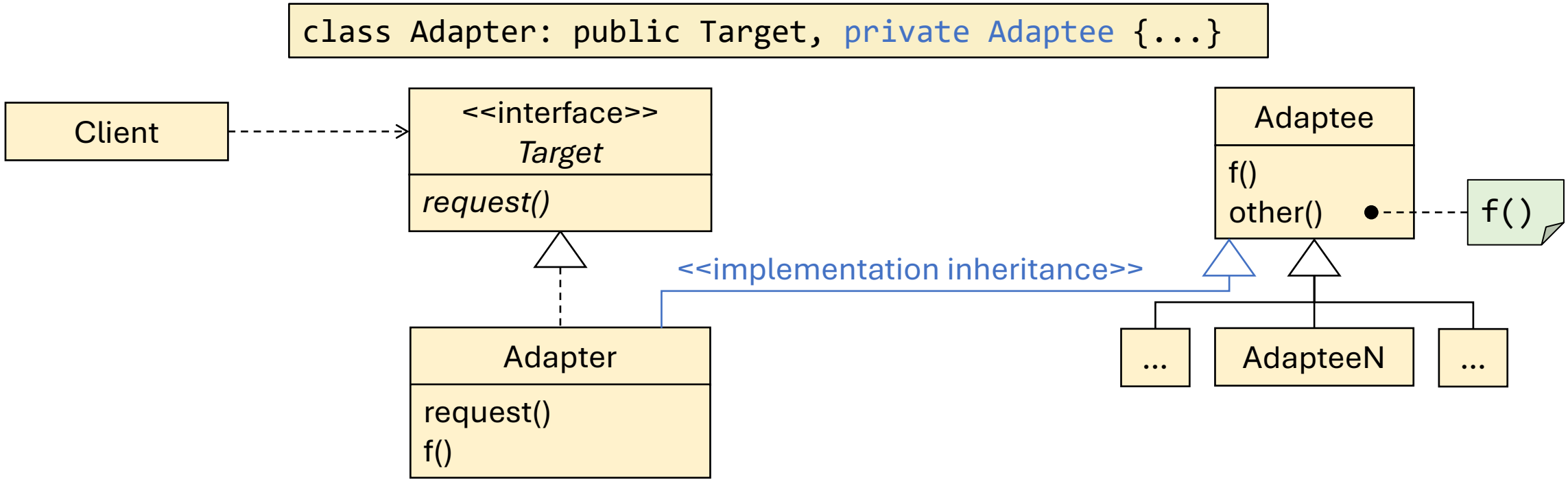
In most cases, multiple inheritance indicates a design flaw.
Use composition and delegation instead.

Protected and Private Inheritance (Pure Code Reuse)

Class Inheritance

C++ offers protected and private inheritance. All inherited members are turned protected / private.

These can be used when we want to reuse base class code while not subtyping, i.e., inheriting the base classes interface.



Runtime Type Checks

`instanceof` Operator

Via `instanceof` we can infer whether an expression is a subtype of specific type at runtime.

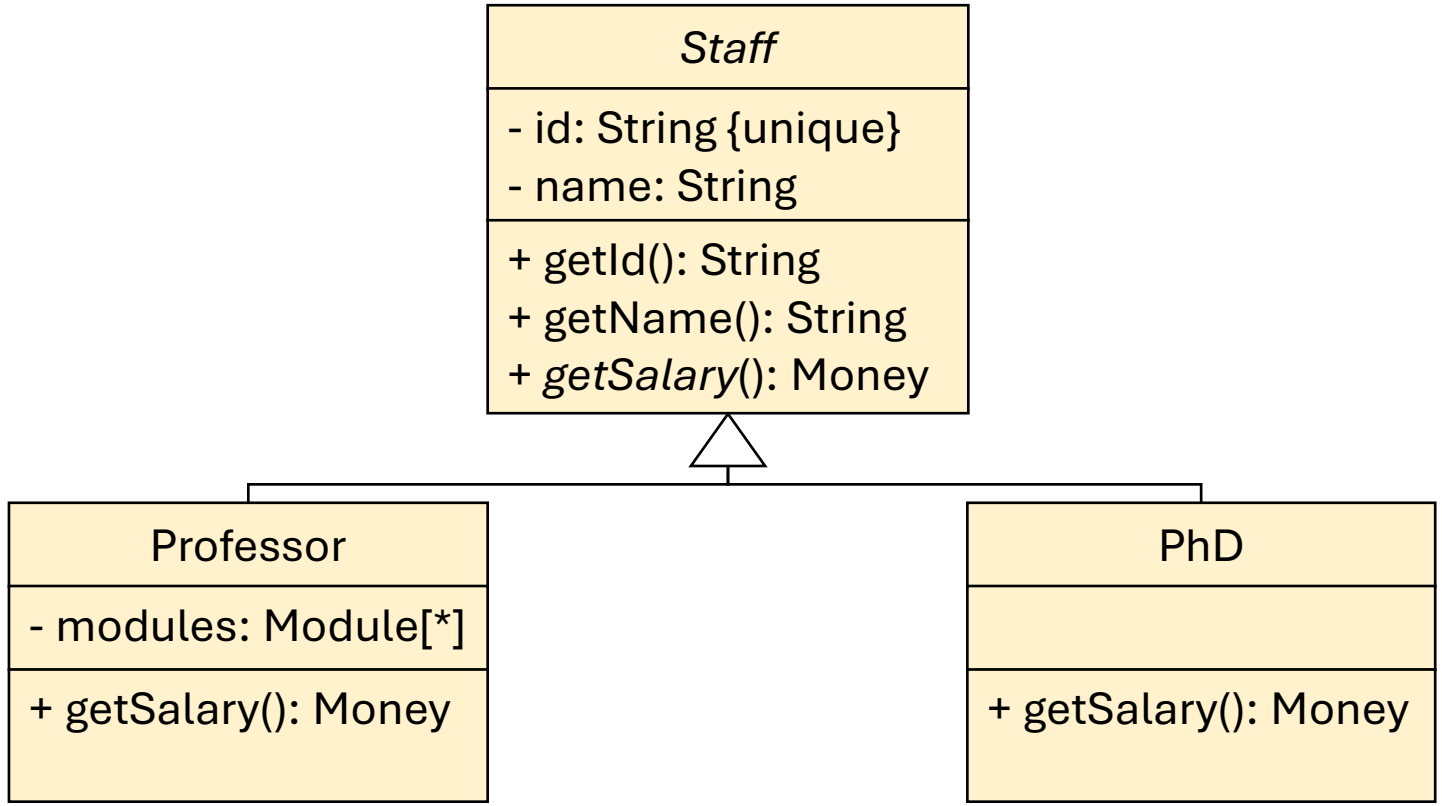
`MyType` must be a reference type.

```
myExpression instanceof MyType
```

`getClass()`

`getClass()` returns the dynamic type of a reference type expression at runtime.

```
myExpression.getClass();
```



Variants of Inheritance Hierarchies

Class Hierarchy Styles

No specific Structure

C++ does not have a common root class. A class that is not defined to inherit from another class indeed does not do so.

Tree-like

C++ does not have a common root class. A class that is not defined to inherit from another class indeed does not do so.

Type Lattice

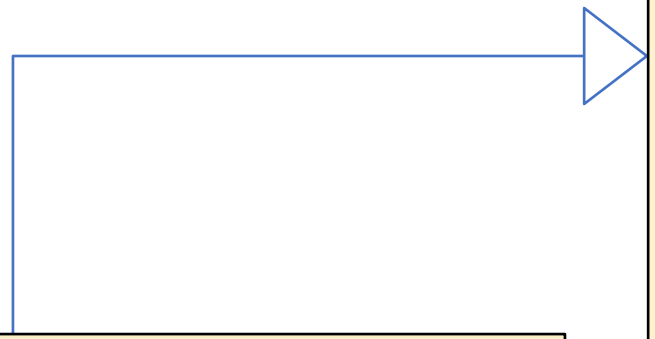
Many modern languages, like Scala and Kotlin, have a type lattice.

This means that they have one common root class (`Any?`) and one common "bottom" class (`Nothing`). (Examples from Kotlin and Scala)

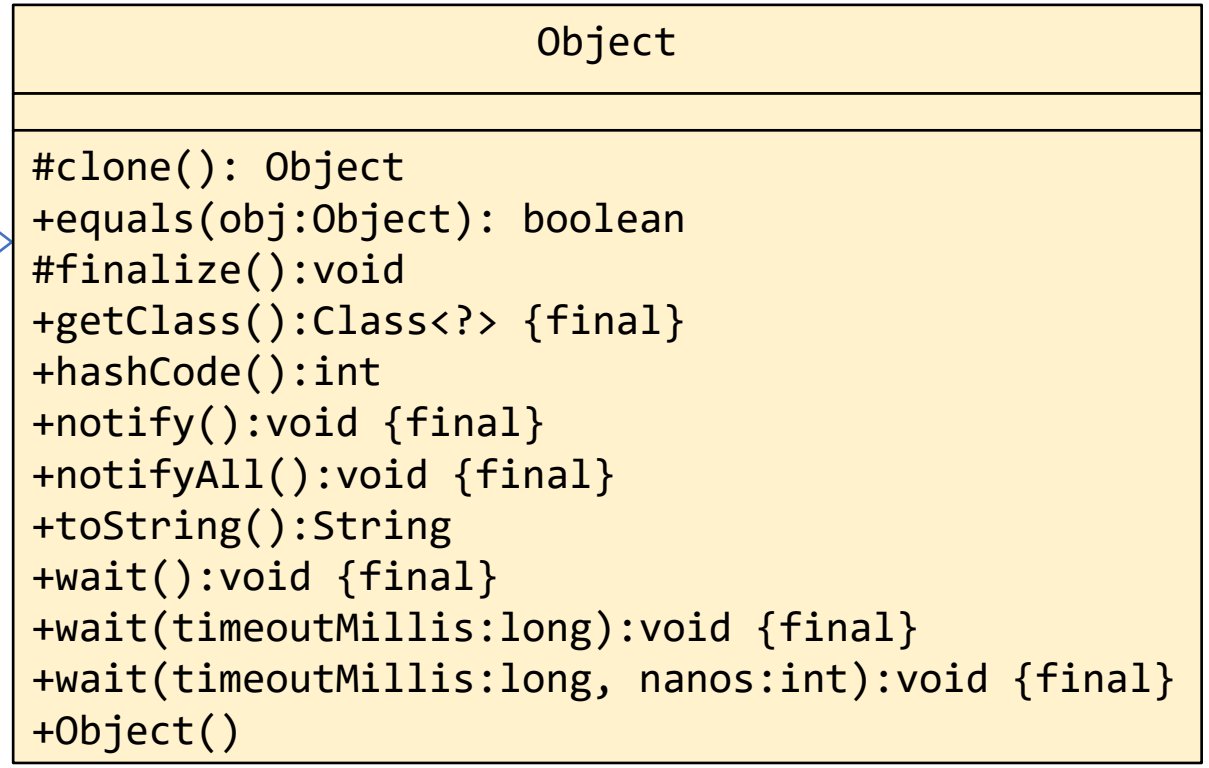
Common Root Class

Root Class

All classes inherit (either implicitly, explicitly or transitively) from a root class.
Java: `Object`
Python: `object`
JavaScript: `Object`, among others.
Kotlin, Scala: `Any?`



```
public class Person extends Object {
   ...
}
```



Abstract Types and Interfaces

Abstract Classes and Methods

Abstract Class

Abstract classes are classes that can not be instantiated directly*.

Abstract Methods

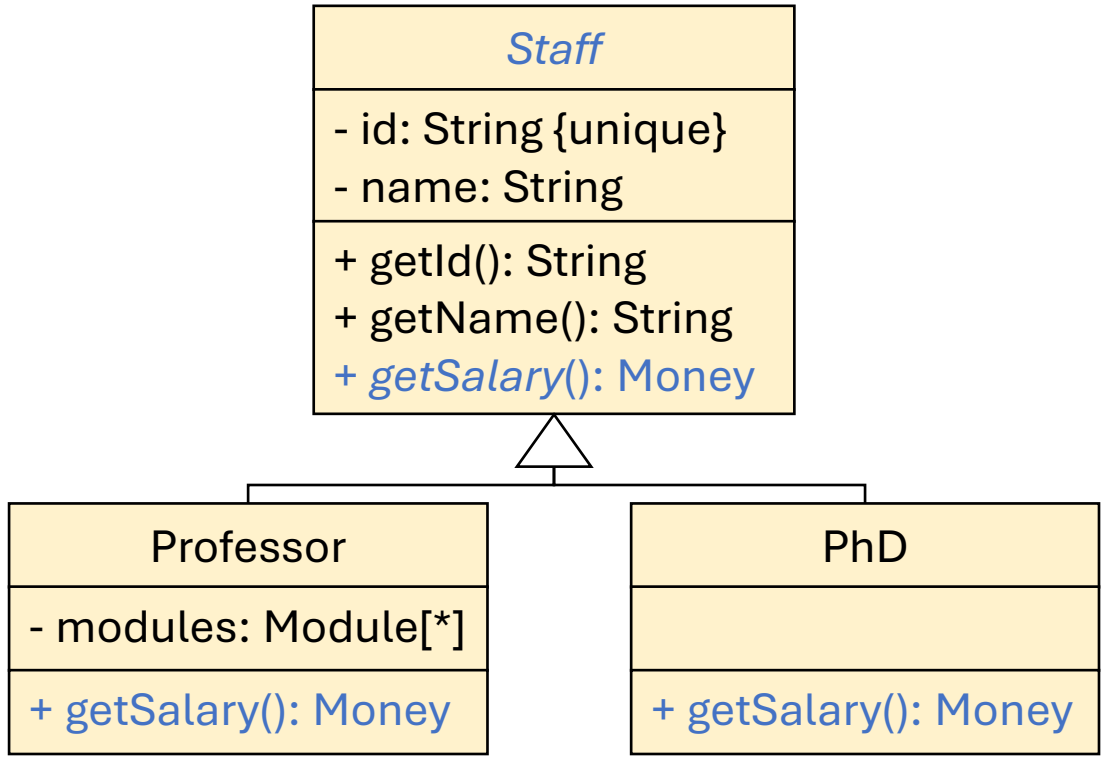
Abstract methods are method headers with no body. They can only be declared in abstract classes and enforce overriding.

```
Staff adrian = new Staff(...);
```

```
Staff adrian = new PhD(...);
```

Implementation

- Java, TypeScript: `abstract` keyword.
- C++: Abstract methods are fully virtual methods (`virtual method() = 0`). Classes: have at least one abstract method.
- Python: Classes: inherit from `ABC`. Methods: `@abstractmethod`.



Interfaces

Interface

Interfaces are maximally abstract types. They only provide public, abstract methods.

Implementation

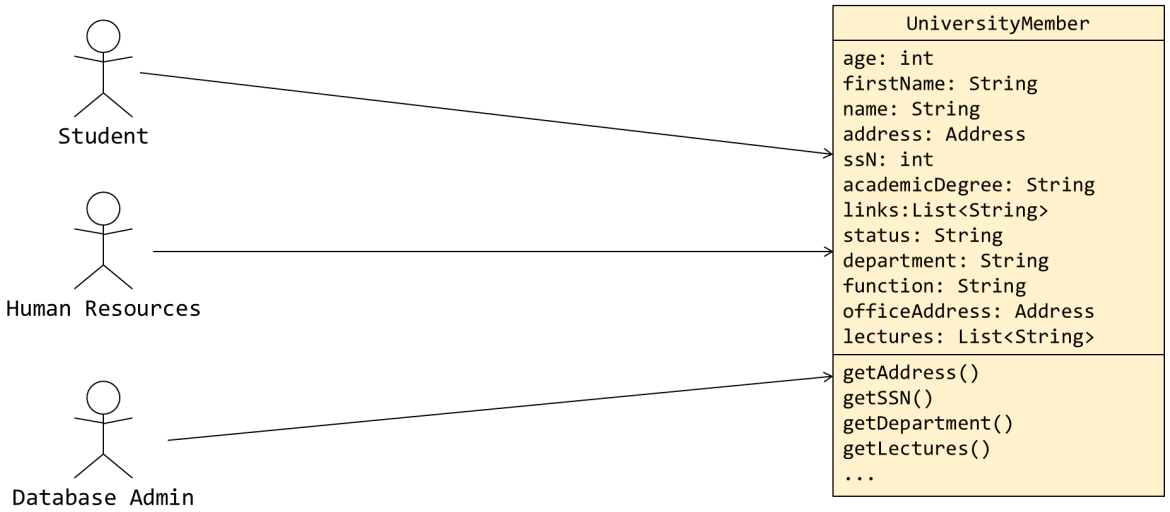
- Java, TypeScript: interface keyword.
- C++, Python: Abstract class with only abstract methods.

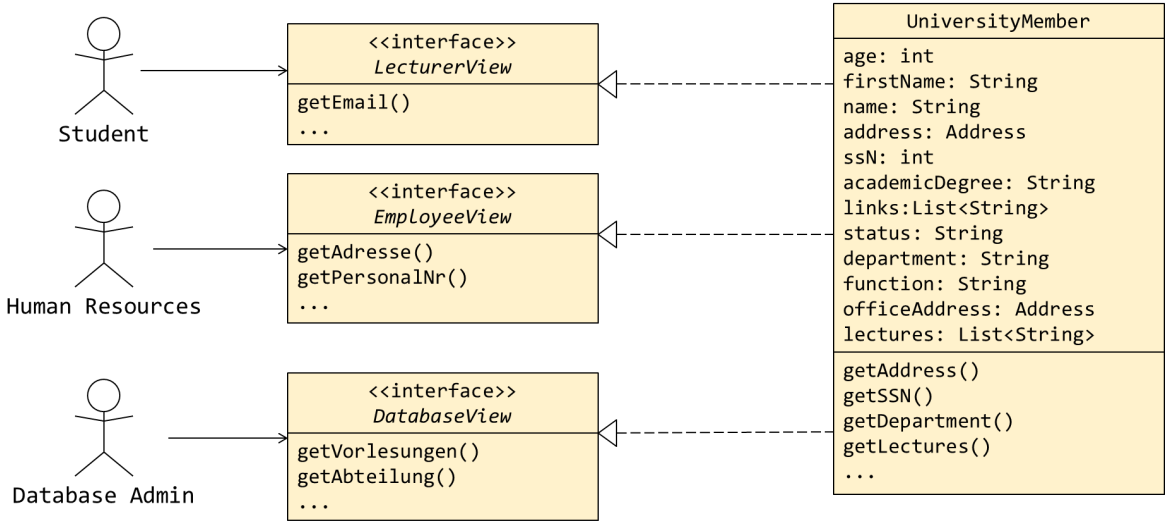
Interface Design

For variables with static type of an interface, all implementing classes can be used as dynamic type. This leads to maximal reusable code.

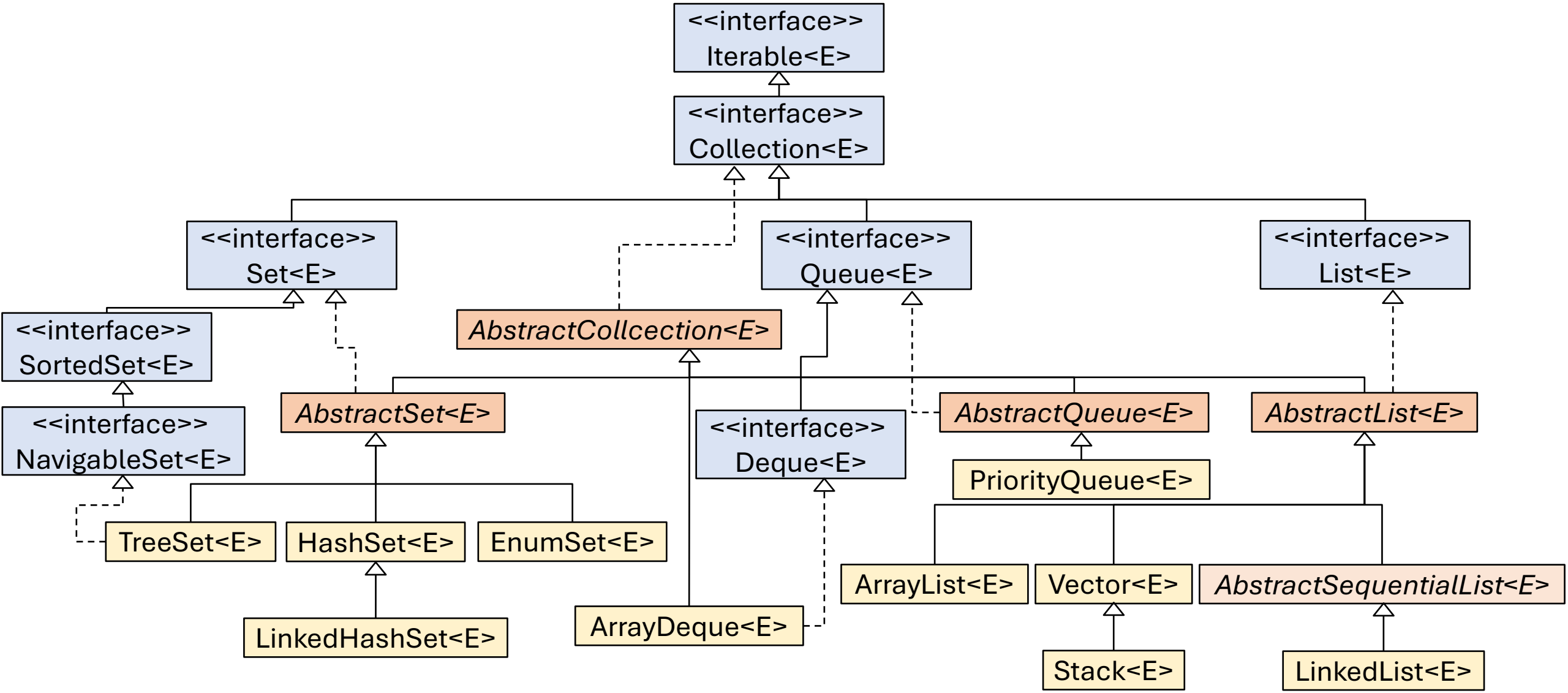
This idea manifests in the Dependency Inversion Principle:

- Use interfaces as static type as often as possible.
- Use concrete classes only for object creation.

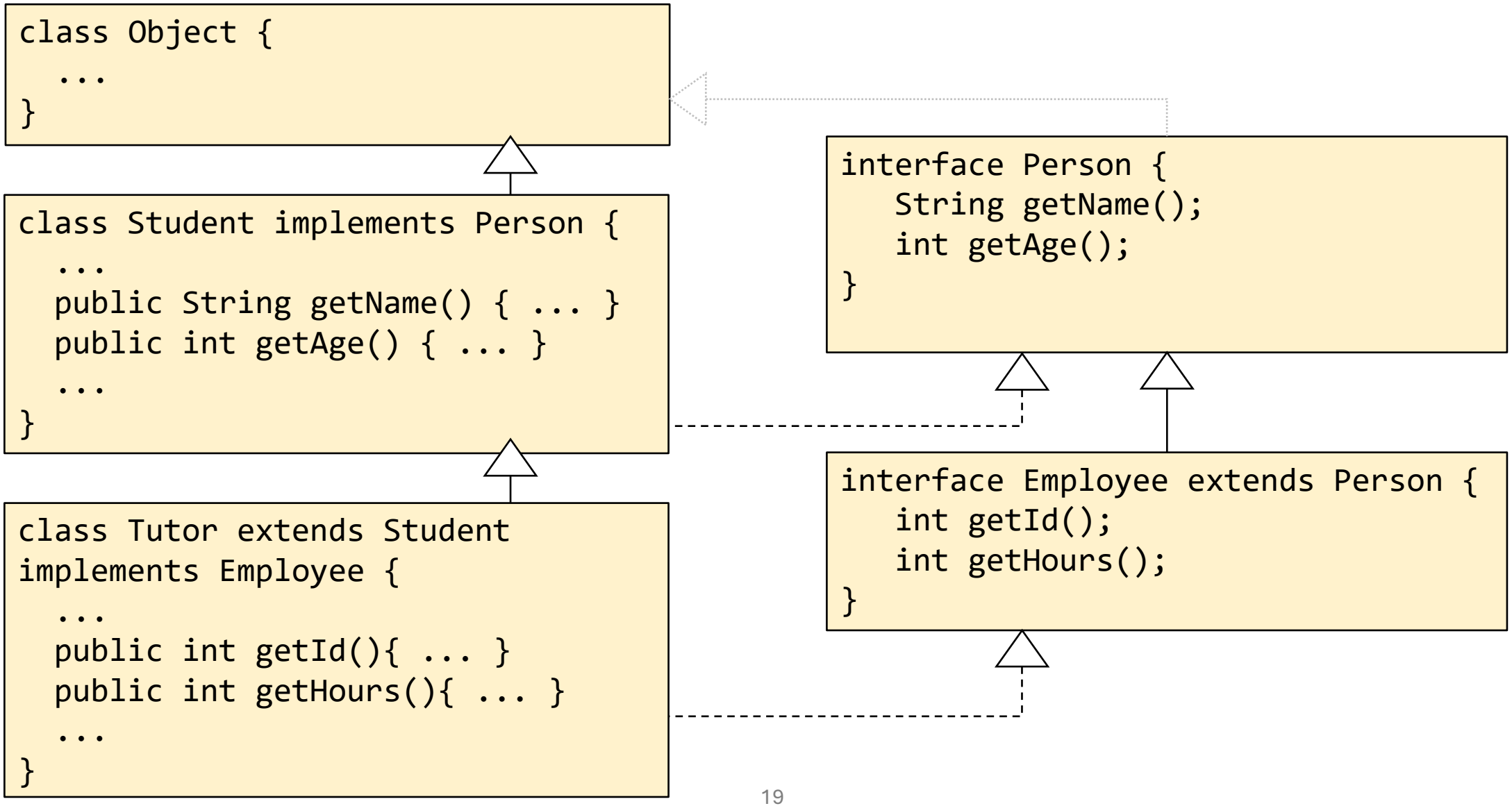


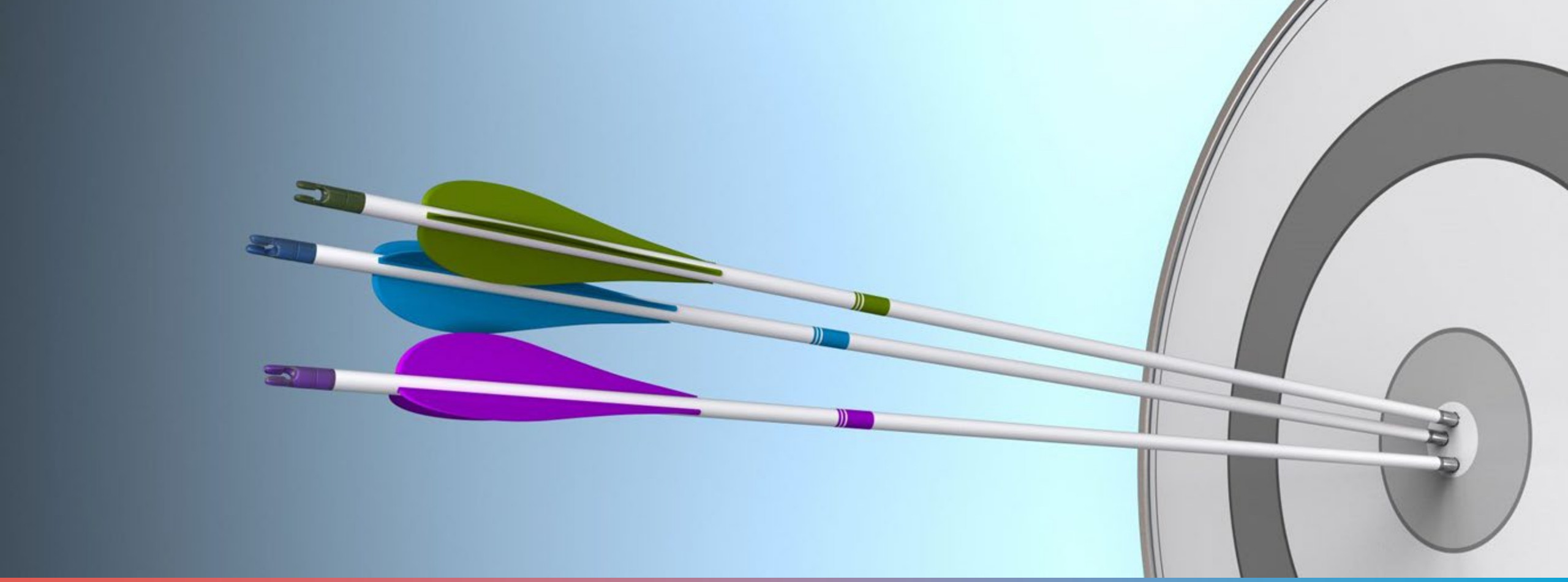


Example Interface Hierarchy: Java Collections



Java's Inheritance and Realization Hierarchies (Example)





Shadowing Hiding, Overloading and Overriding

Shadowing

Shadowing

A variable in a block that has the same name as a variable in the next level scope shadows that outer variable.

In Java, any variable may only shadow member variables.

```
public class Person {
   int age = 15;

   public void setAge(int age) {
      age = age;
      this.age = age;
   }
}
```

Hiding

An instance variable (or class variable or class method) that redefines a member with the same name from a super class hides that member.
Hiding is a backwards-compatibility feature to evade name clashes and should not be used in regular design. It is considered bad design.

```
public abstract class Person {
   String firstname;
      ...
}
```

```
public class Student
extends Person {
   String firstname;
   ...
}
```

Dynamic Binding

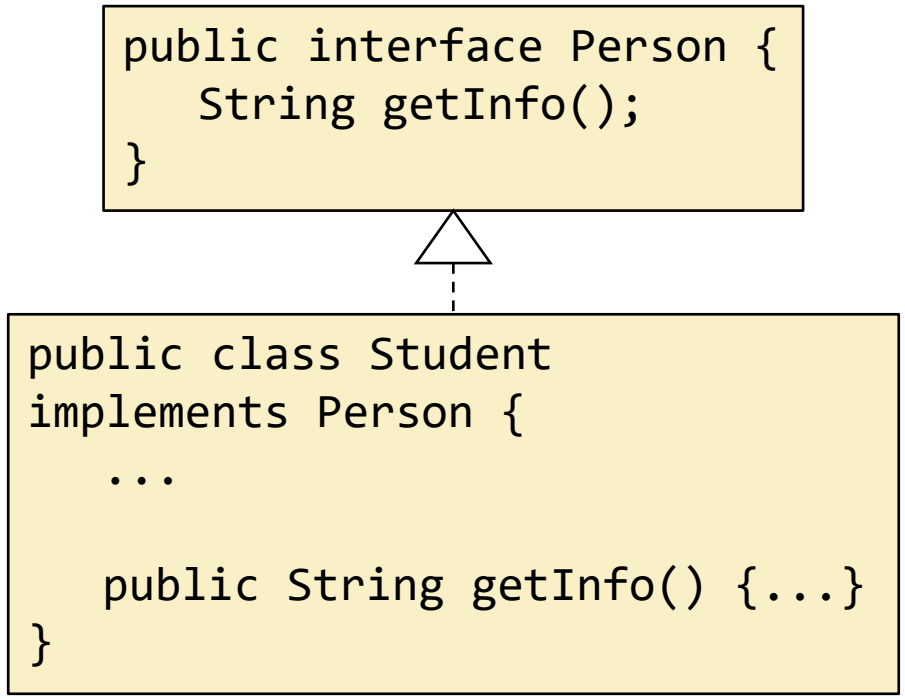
Overriding

An instance method that redefines an instance method with same name and parameter list* overrides that method.

Overridden methods are bound dynamically.

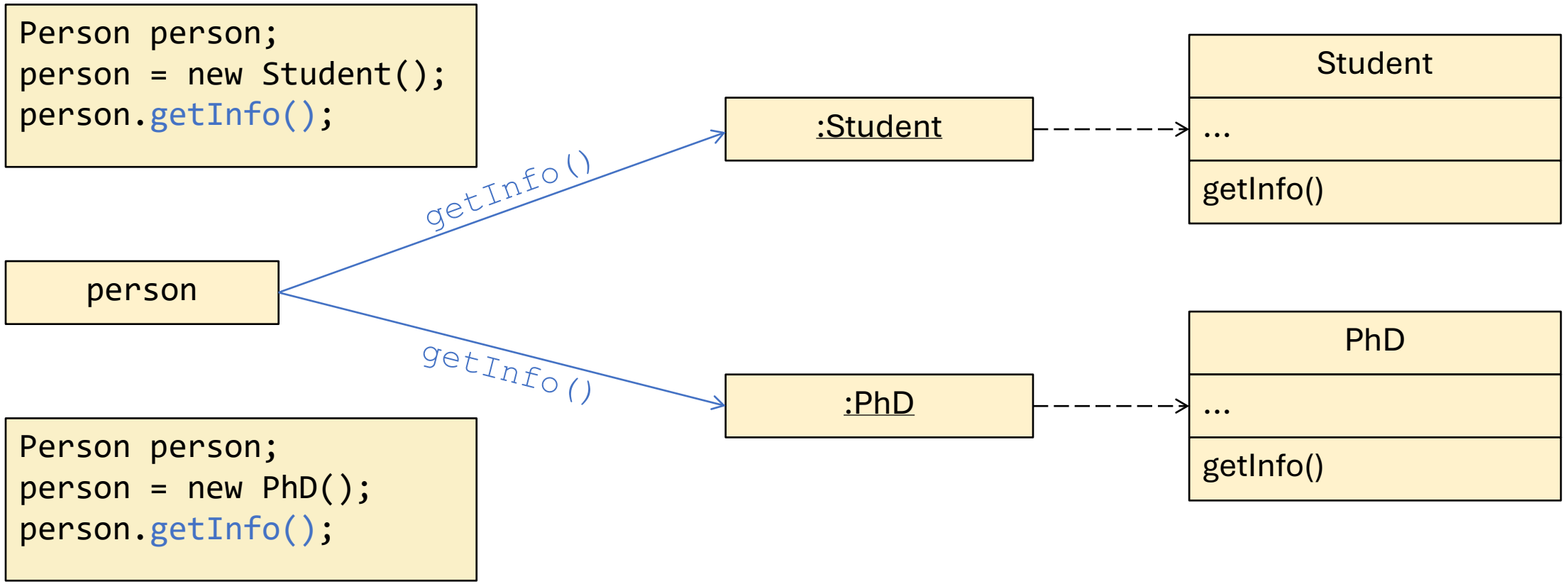
Dynamic Binding

Dynamic binding is the binding (~selection) of a method call at runtime. Based on the current receiver of a message, the implementation of the called operation is executed.




```
Person e;
Student adrian;
...
e = adrian;        //Allowed, Student :> Entity
...
e.getInfo();       //Allowed, getInfo is in e
```

Example of Dynamic Binding



Polymorphism

Polymorphic Expression

Polymorphic expressions are expressions that can take on values of different types.

Ad Hoc Polymorphism (Overloading)

A procedure behaves different depending on the types they operate with.

Subtype Polymorphism

Expression can have values of the static type or of subtypes of the static type.

Parametric Polymorphism

A class or method is defined with some concrete types missing, instead using abstract symbols that can be substituted by concrete types.

Polymorphism & Dynamic Binding

The same code can process objects of different types (polymorphism) and depending on the receiver object, this code has different effects (dynamic binding).

Overloading

Overloading

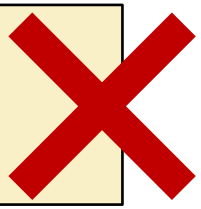
In Java, C++, Python & others, we can define multiple methods with the same identifier in the same class, but different parameter lists.

Overloading based solely on a different return type is not possible.

```
public int plus(int a, int b) {
   return a + b;
}
```

```
public float plus(float a, float b) {
   return a + b;
}
```

```
public float plus(int a, int b) {
   return a + b;
}
```



Determination of the called Method

The compiler must determine the method actually called. There are four possible cases:

- There is no such method: undefined call.
- There is only one method with this name (standard).
- More than one method and signatures are not related: No signature subtype.
- One signature is a subtype of the other.

Signature Subtyping

A signature subSig is a subtype of a signature superSig if

- they have the same number of parameters
- for each position *i*, the parameter type in subSig at *i* is a subtype of the parameter type in superSig at *i*.

$$\text{name}(T_1, \ldots, T_n) \subseteq \text{name}(T'_1, \ldots, T'_n) \leftrightarrow \forall\, i=1..n: T_i \subseteq T'_i$$

Determination of Method

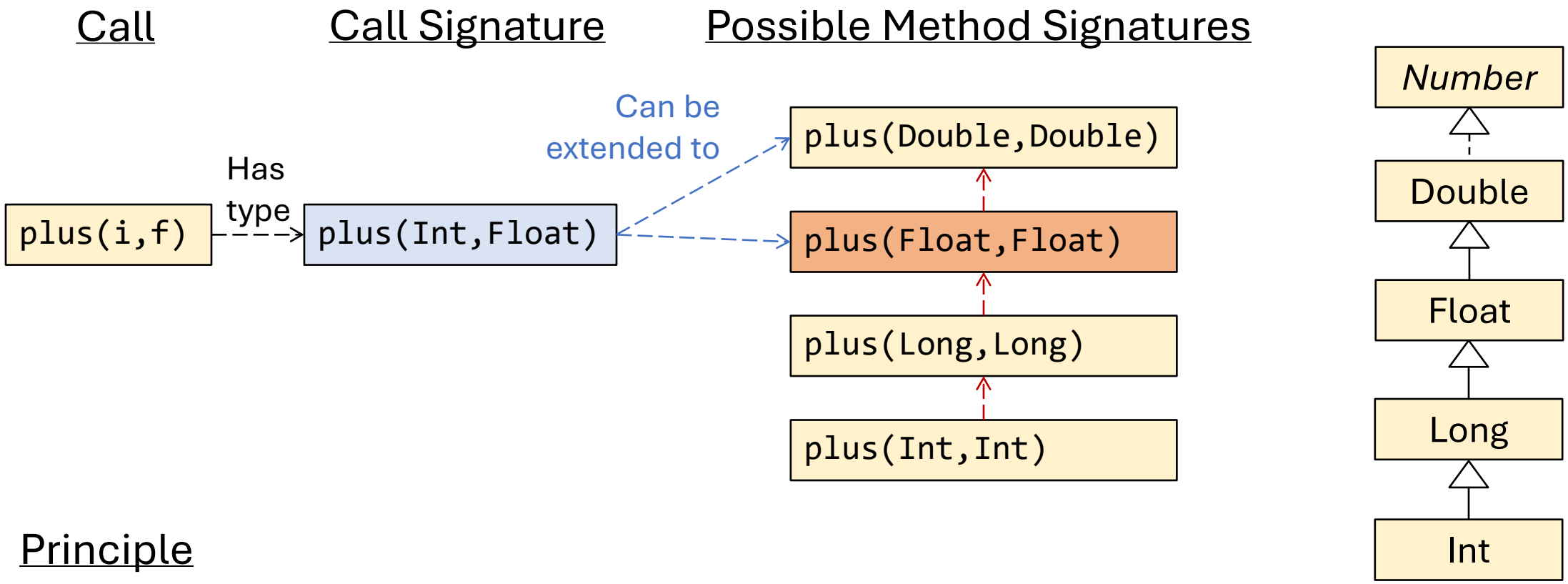
```
Calculator calc = new Calculator();
Int i = new Int(3);
Float f = new Float();

Long n = calc.store(Number.plus(i,f));
```

```
public class Calculator {
  public Float store(Float f) {...}
  public Long store(Long l) {...}
}
```

```
public interface Number {
   Float plus(Float a, Float b);
   Float plus(Integer a, Integer b);
   Float plus(Long a, Long b);
   Float plus(Double a, Double b);
}
```

1. Is this assignment legal?
2. What is the type of this expression?
3. What is the call signature?
4. Which of these methods is called?
5. Which return type has this call?
6. What is the call signature of `plus`?
7. The call signature is `plus(Int,Float)`.
8. Which variant of `plus` is called?

Determination of the Call Signature



Principle

Take the most specific signature that the call signature is a subtype of. → The lowest in the subtype hierarchy.

Determination of Method

```
Calculator calc = new Calculator();
Int i = new Int(3);
Float f = new Float();

Long n = calc.store(Number.plus(i,f));
```

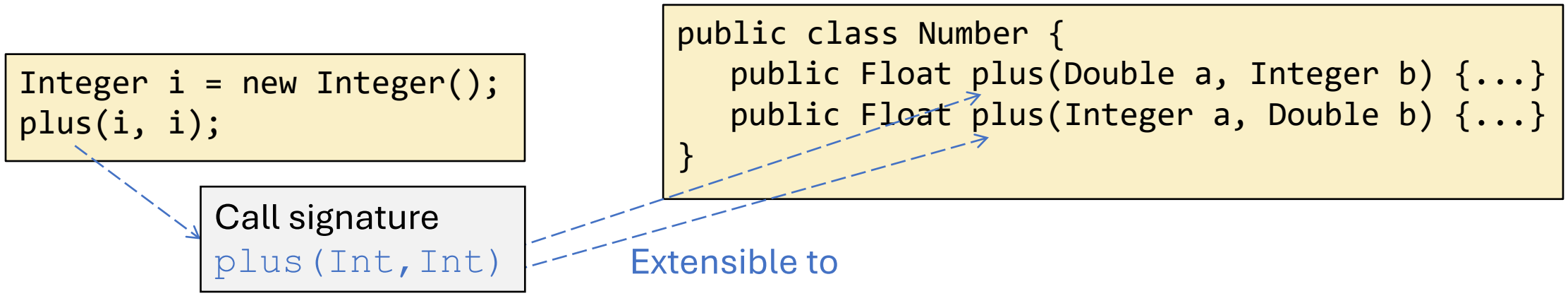
```
public class Calculator {
  public Float store(Float f) {...}
  public Long store(Long l) {...}
}
```

```
public interface Number {
   Float plus(Float a, Float b);
   Float plus(Integer a, Integer b);
   Float plus(Long a, Long b);
   Float plus(Double a, Double b);
}
```

1. Is this assignment legal?
2. What is the type of this expression?
3. What is the call signature?
4. Which of these methods is called?
5. Which return type has this call?
6. What is the call signature of `plus`?
7. The call signature is `plus(Int,Float)`.
8. Which variant of `plus` is called?
9. `plus(Float,Float)` is called.
10. The return type is `Float`.
11. The call signature thus is `store(Float)`.
12. `store(Float)` is called.
13. The type of this expression is `Float`.
14. The assignment `Long = Float` is illegal!

Ambiguous Calls

If there is no most specific call signature, i.e., there are at least two call signatures on the lowest level, the call is ambiguous. This causes a compile time error.



Hiding, Overloading and Overriding

Method Call Resolution

1. Is the message to the receiving object allowed? (At compile time)
   →Everything declared in the static type is allowed.

2. Which variant of a member with same identifier in a class hierarchy is called? (At compile time)
   - Hiding: The static type defines the interface and implementation.
   - Overloading/Overriding: The static type defines the interface; the dynamic type defines the implementation (Dynamic Binding)

3. Overloading: Determine the called operation. (At compile time)
   - Determine the call signature.
   - Determine possible candidates in the receiver's static type.
   - Determine the most specific out of these.

4. Overriding: Determine the called implementation (Dynamic Binding, at runtime).

Class-based OOP Implementation

"Imperative Style"

Imperative Compilation

In imperative, non-OO languages, all memory addresses are statically known. The same is true for class attributes and class methods.

```
public class Student {
   ...
   static int getNumberOfStudents() {
      ...
   }
   ...
}
```

Compiles to →

```
0x4F: ...//code of
      //getNumberOfStudents
```

```
Student.getNumberOfStudents();
```

Compiles to →

```
...//code for passing
   //arguments
call 0x4F;
```

Object-Oriented Style

vTable

A vTable (virtual function table) is a memory space that lists all instance methods of a class.

- The vTable of a class consists of all methods from the class itself as well as its superclass, in order from highest to lowest class in the hierarchy.
- Each method has a dedicated index.
- All inherited methods have the same index as in the super classes.
- Own methods follow after inherited methods.

Objects in Class-Based Systems

Objects in memory only have a reference to their class and their attributes.

- Each object's memory consists of a list of its attributes (from own class and inherited), in the order of the highest to lowest class in the hierarchy.
- Each attribute has a dedicated index.
- Inherited attributes have the same index in the current class as in the super classes.
- Own attributes follow after all superclass attributes.

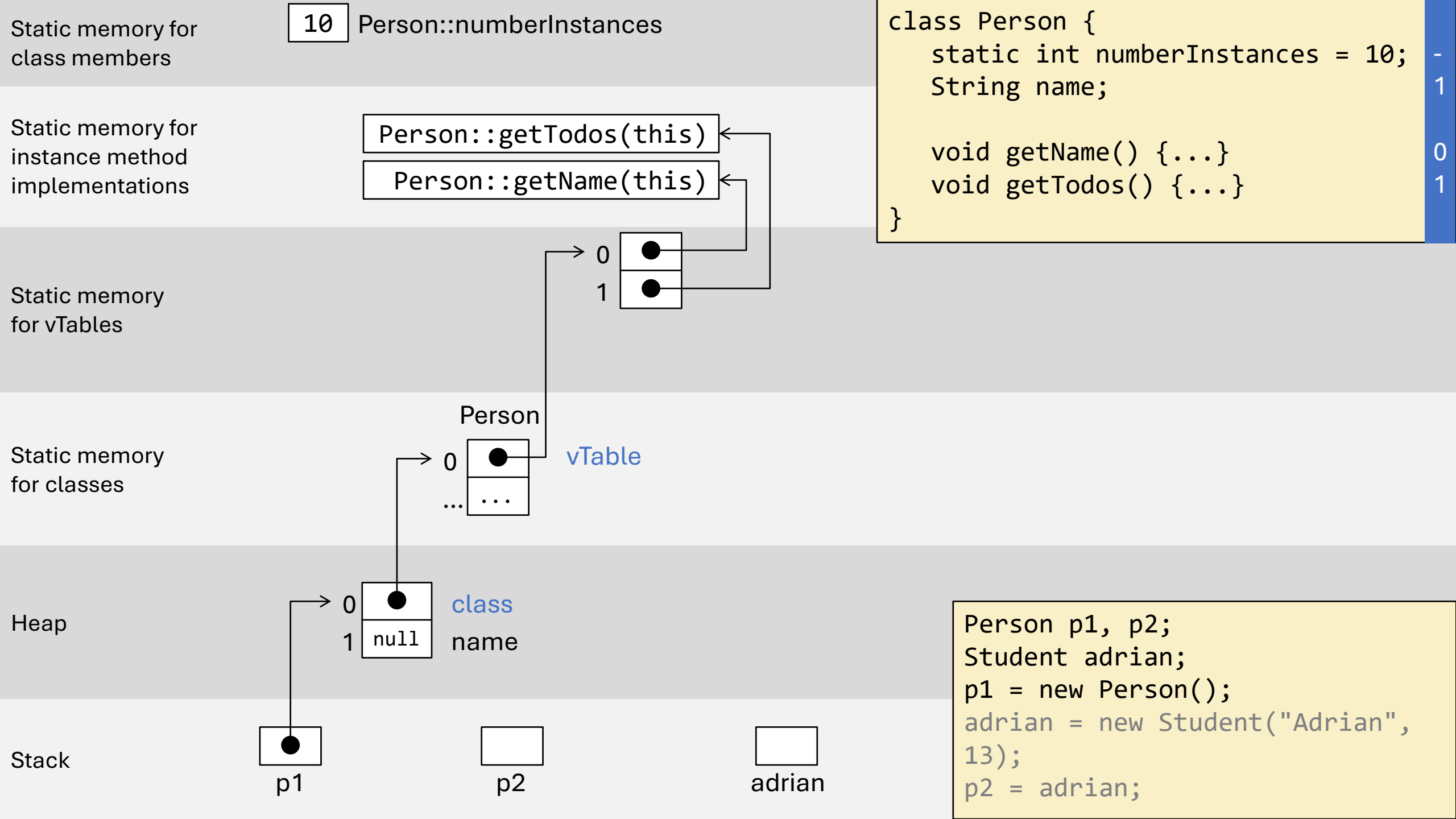
Compiled Code

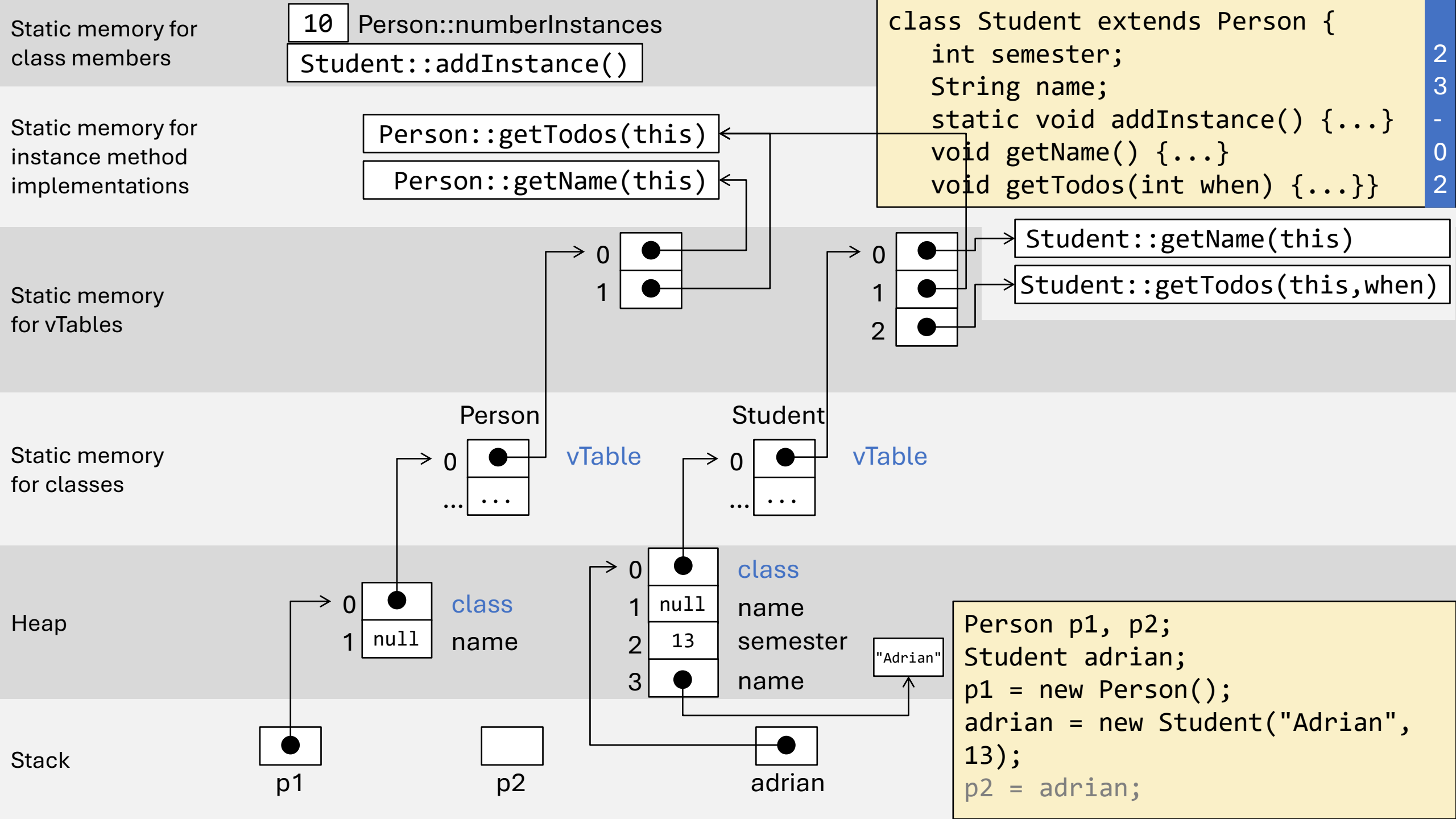
Each attribute access and method call of an object is replaced with an access to their corresponding index in the object memory layout / vTable.

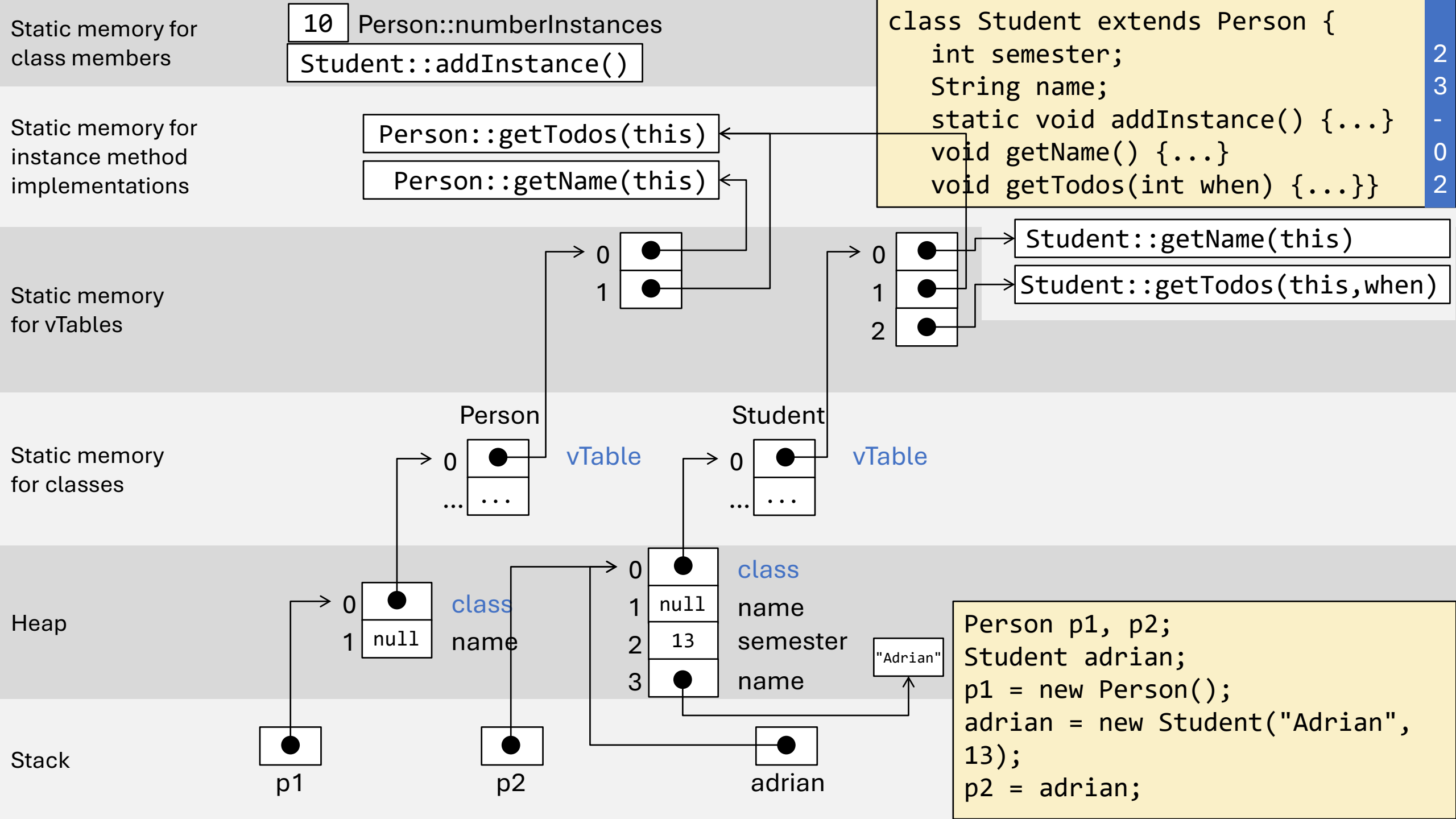
```
Person p1, p2;
Student adrian;
p1 = new Person();                  -
adrian = new Student("Adrian",      -
13);
p2 = adrian;
p1.name;                            1
p1.getName();                       0
p1.getTodos();                      1
p1.getTodos(4);//not in Person      X

p2.getName();                       0
p2.getTodos();                      1
p2.getTodos(4);                     2

adrian.getName();                   0
adrian.getTodos();                  1
adrian.getTodos(4);                 2
adrian.name;                        3
```

```
class Person {
   static int numberInstances = 10;                    -
   String name;                                        1

   void getName() {...}                                0
   void getTodos() {...}                               1
}
```

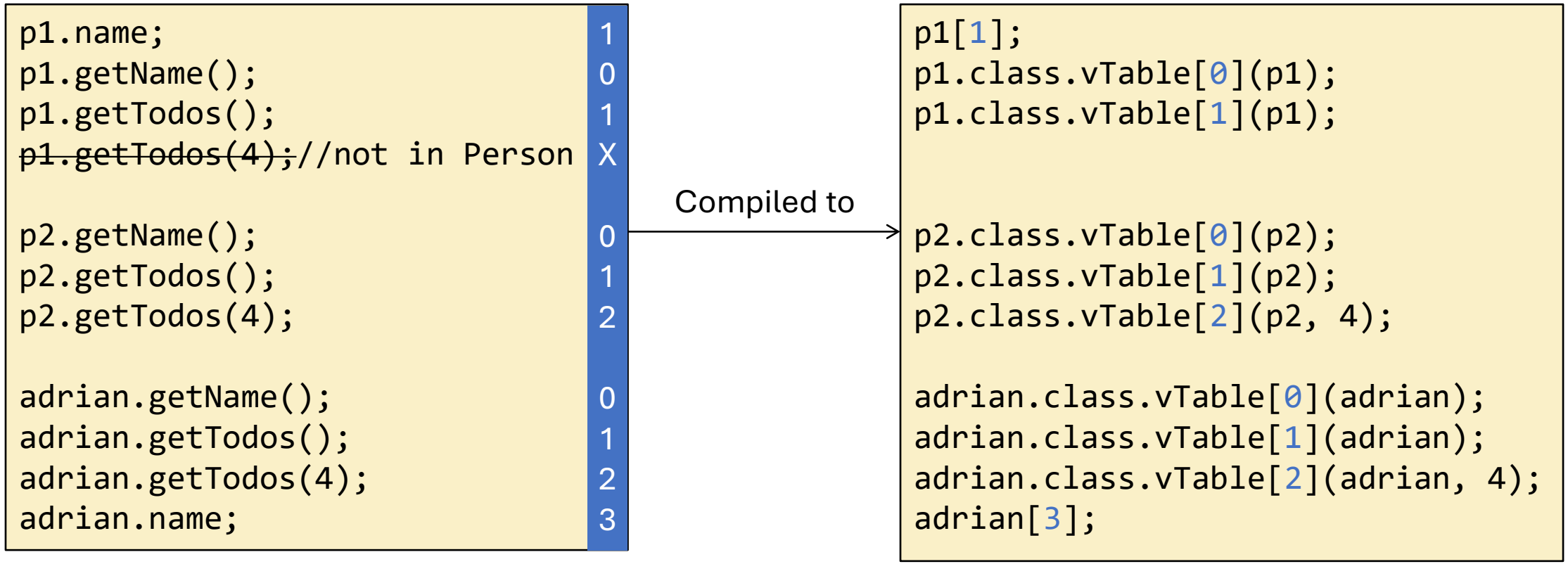
```
class Student extends Person {
   int semester;                    //Addition         2
   String name;                     //Hiding           3

   static void addInstance() {...}                     -
   void getName() {...}             //Overriding       0
   void getTodos(int when) {...}    //Addition + Overloading 2
   Student(String name, int semester) {                -
      this.name = name;
      this.semester = semester;
   }
}
```


Static memory for class members
10
Person::numberInstances
Static memory for instance method implementations
Person::getTodos(this)
Person::getName(this)
Static memory for vTables
0
1
Static memory for classes
Person
0
...
vTable
Heap
0
1
null
class
name
Stack
p1
p2
adrian
class Person {
static int numberInstances = 10;
String name;
void getName() {...}
void getTodos() {...}
}
-1
01
Person p1, p2;
Student adrian;
p1 = new Person();
adrian = new Student("Adrian", 13);
p2 = adrian;

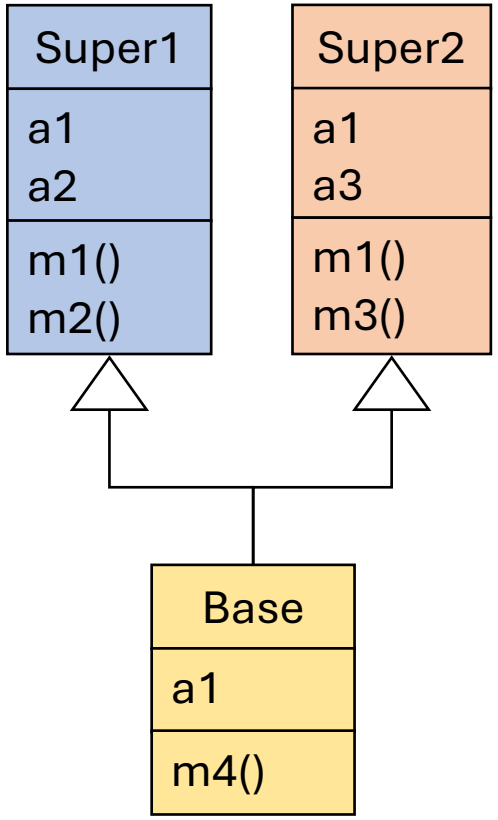

Static memory for class members
10
Person::numberInstances
Student::addInstance()
Static memory for instance method implementations
Person::getTodos(this)
Person::getName(this)
Static memory for vTables
0
1
0
1
2
Student::getName(this)
Student::getTodos(this,when)
Static memory for classes
Person
0
...
vTable
Student
0
...
vTable
Heap
0
1
null
class
name
0
1
null
2
13
3
class
name
semester
name
"Adrian"
Stack
p1
p2
adrian
class Student extends Person {
int semester;
String name;
static void addInstance() {...}
void getName() {...}
void getTodos(int when) {...}}
23-02
Person p1, p2;
Student adrian;
p1 = new Person();
adrian = new Student("Adrian", 13);
p2 = adrian;


Static memory for class members
10
Person::numberInstances
Student::addInstance()
class Student extends Person {
int semester;
String name;
static void addInstance() {...}
void getName() {...}
void getTodos(int when) {...}}
23-02
Static memory for instance method implementations
Person::getTodos(this)
Person::getName(this)
Static memory for vTables
0
1
Student::getName(this)
Student::getTodos(this,when)
Static memory for classes
Person
vTable
Student
vTable
Heap
class
name
null
class
name
semester
name
13
"Adrian"
Stack
p1
p2
adrian
Person p1, p2;
Student adrian;
p1 = new Person();
adrian = new Student("Adrian", 13);
p2 = adrian;

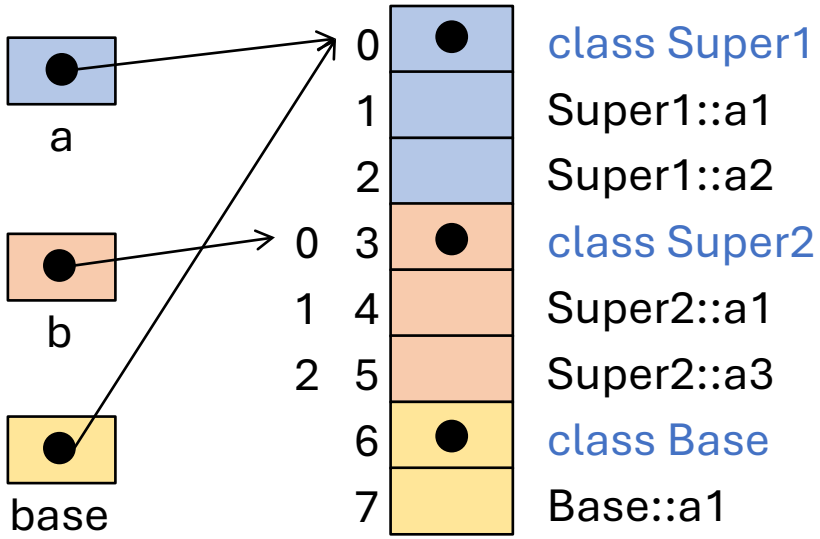
Static Code Generation for Dynamic Access



Multiple Super Classes




```
Super1 a;
Super2 b;
Base base;

a = b = base = new Base();
```



Summary and Outlook

Efficiency of Dynamic Binding

In Java, dynamic binding needs 3 reading accesses and one jump to an apriori unknown position.

- There are further optimizations:
- Inlining
- Polymorphic inline caches
- "Just-in-time" Compilation
- Dynamic "hot-spot" recompilation

Outlook

Things we did not cover (yet):

- Double/dynamic dispatch.
- Manifest and inferred types.
- Dependent, flow-sensitive and refinement types.
- Union and intersection types.
- Strong and weak typing.
- Type and memory safety.

Summary

- Types are realized by classes.
- Types can be subtype relation.
- Expressions have static and dynamic types.
- Only use (public) class inheritance if classes are subtypes.
- Abstract classes and interfaces define common super types / interfaces of classes.
- Polymorphism and Dynamic Binding allow reuse of the same code with different effects.
- Class-based systems are implemented efficiently.