编译原理

14. 面向对象语言

rainoftime.github.io
浙江大学
计算机科学与技术学院

Content

Advanced Languages

- Advanced programming features
  - ML data types, exceptions, modules, objects, concurrency, ...
  - Fun to use, but require special techniques to compile and optimize
  - Today will be looking at how to compile objects and classes similar to those found in Java

Object-Oriented Languages

- **Class-based, object-oriented** (OO) language
  - All (or most) values are objects,
  - Objects belong to a class,
  - Objects encapsulate **state** (fields) and **behavior** (methods).

- Some important features of OO languages
  1. Inheritance
  2. Encapsulation
  3. polymorphism

Outline

1. Classes
2. Single Inheritance
3. Multiple Inheritance
4. Testing Class Membership
5. Private Fields and Methods

1. Classes

Object-Tiger: Adding Declarations

- Extend the Tiger language with new declaration syntax to create classes:

dec → *classdec*
classdec → ***class*** *class-id* ***extends*** *class-id* { {*classfield* } }
classfield → *vardec*
classfield → *method*
method → ***method*** *id*(*tyfields*) = *exp*
method → ***method*** *id*(*tyfields*) : *type-id* = *exp*

Object-Tiger

- class B extends A { ... }
  - declares a new class B that extends class A
  - All the fields and methods of A implicitly belong to B
  - Some of the A methods may be **overridden** (have new declarations) in B. The parameter and result types mut be identical
  - But the fields may not be overridden

- There is a predefined class identifier Object with no fields or methods

What about *self*?

- class B extends A { ... }
- Self is not a keyword in Object-Tiger
  - It is an implicit parameter for each method
  - Automatically bound to the object during runtime

```
Class Car extends Vehicle {
...
  method await(/*self: Car,*/ v: Vehicle) {
    if (v.position < position)
      then v.move(position – v.position)
    else self.move(10)
  }
}
var c := new Car
c.move(60); -> move(c, 60);
```

Object-Tiger: Adding Expressions

- New expression syntax to create objects and invoke methods

exp → ***new*** *class-id*
→ *lvalue . id()*
→ *lvalue . id(exp{, exp})*

- Example
  - new B : makes a new object of type B
  - b.x : the field x of object b
  - b.f(x, y): a call to the f method of object b with explicit actual parameters x and y

Example: Object-Tiger Class

class name

superclass that
Vehicle inherits from

```
let
 class Vehicle extends Object {
  var position := start
  method move (x : int) = (position := position + x)
 }

in ... end
```

field
declaration

method declaration

Example: Object-Tiger Class

```
let
  class Vehicle extends Object {
    var position := start
    method move (x:int) = (position := position + x)
  }

  class Car extends Vehicle {
    var passengers := 0

    method await(v:Vehicle) =
      if (v.position  < position) then
          v.move(position – v.position)
      else
          self.move(10)
  }
in ... end
```

new field declaration

new method declaration

call to inherited method

v’s “position” field

current object’s “position” field

Example: Object-Tiger Class

```
let
  class Vehicle extends Object {
    var position := start
    method move (x:int) = position := position + x
  }

  class Car extends Vehicle { ... }

  class Truck extends Vehicle {
     method move (x:int) =
        if x <= 55 then
           position := position * x

in ... end
```

method override

Example: Object-Tiger Class

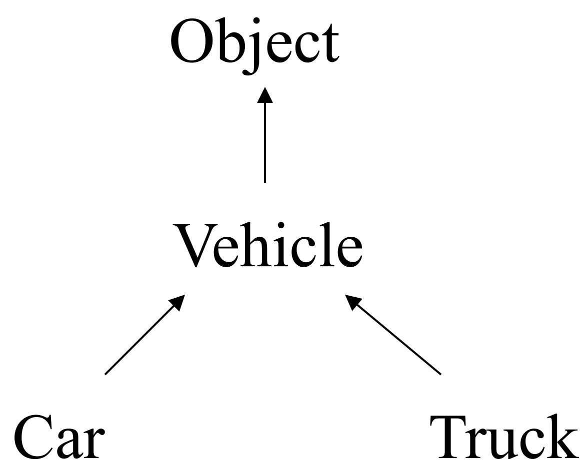
```
let
  class Vehicle extends Object { ...  }
  class Car extends Vehicle { ... }
  class Truck extends Vehicle {...}

  var t := new Truck
  var c := new Car
  var v : Vehicle := c
in
   c.passengers := 2;
   c.move(60);
   v.move(70);
   c.await(t);
end
```

new object created

a car calls an inherited method

Class Hierarchy

- The class hierarchy is the graph of inheritance relationships in a program:



  - In a single-inheritence (SI) language, the graph is a tree
  - In a multiple-inheritence (MI) language, the graph is a DAG
  - Multiple-inheritence languages are much trickier to implement than single-inheritence languages

Challenging in Implementing Object-Tiger

1. Field layout — arranging object fields in memory
   (How do we access object fields?)

2. Method dispatch — finding which concrete implementation of a method to call

3. Membership test — testing whether an object is an instance of some type (e.g, isinstanceof)

2. Single Inheritance

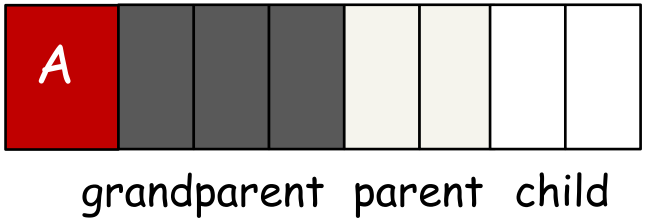
Single Inheritance

- Single-inheritance languages:
  - Each class extends just one parent class
  - E.g., Java, C#, Swift, …
- What are inherited and how to model them?
  - Field inheritance
  - Method inheritance

Field Inheritance

- Fields are inherited from the parent class
  - How to co-locate them with newly-defined fields?

- Simple for single-inheritance: prefixing
  - Recap: object layout in memory (class-descriptor)
  - Inherited fields are **put at the beginning**, in the same order

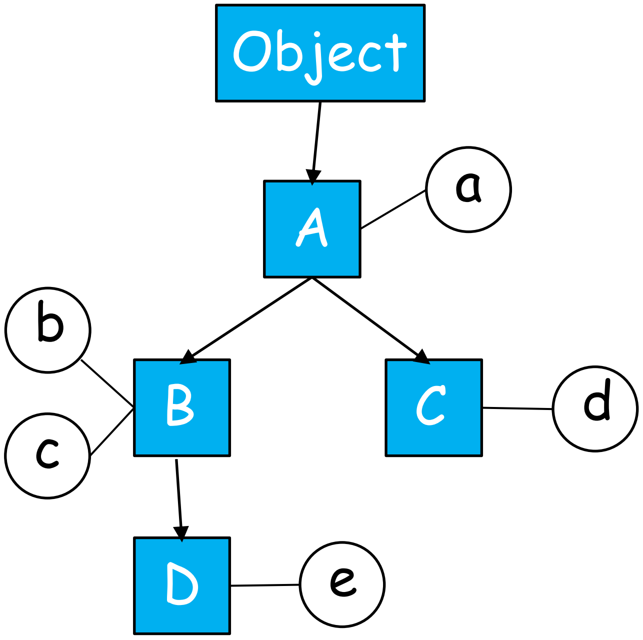


Example: Field Inheritance

- What are the layouts for A, B, C, D?

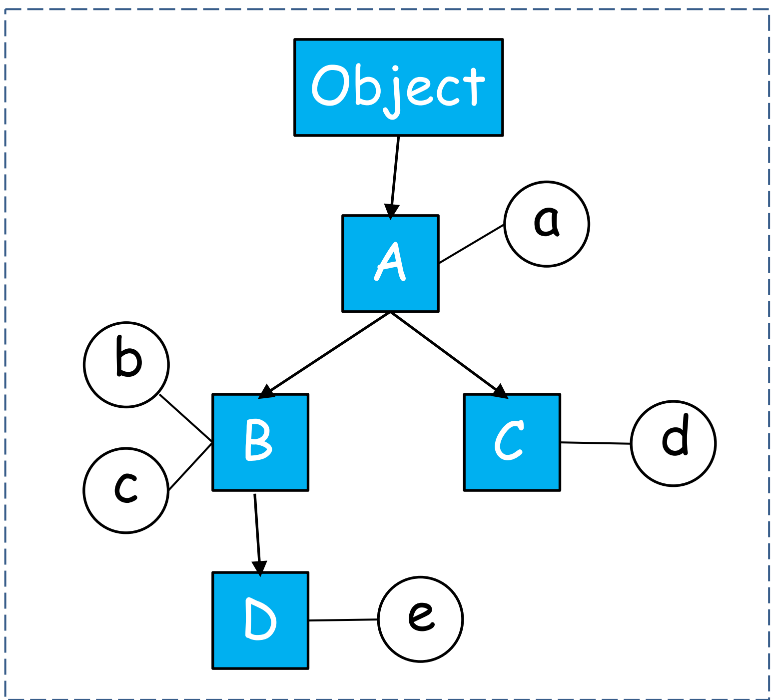
```
class A extends Object
   { var a := 0}

class B extends A
   {  var b := 0
      var c := 0}

class C extends A
   {var d := 0}

class D extends B
  {var e := 0}
```

type hierarchy



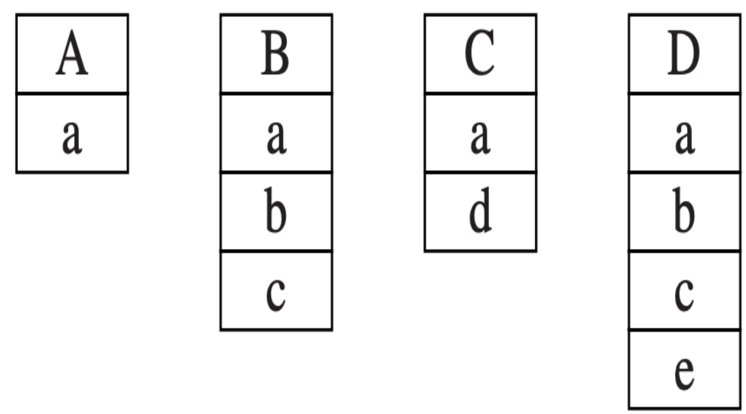
Example: Field Inheritance

- Where B extends A, those fields of B that are inherited from A are laid out in a B record at the beginning, in the same order they appear in A records
- Fields of B not inherited from A are placed afterward



type hierarchy

Fields are ordered by following the path!



object layout

What about Method Inheritance?

Code generation for class methods

1. **Method Code Generation**

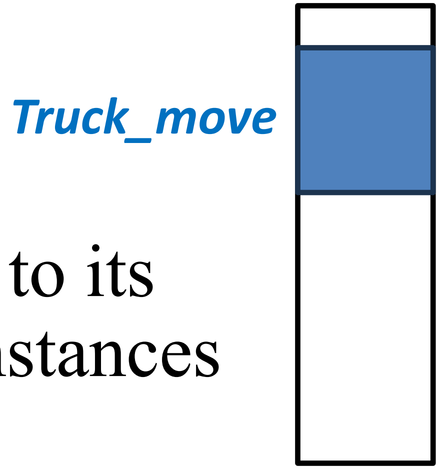
   Compile a method as some machine code located at a particular address

1. **Method Dispatch**

   At a method invocation point, figure out what code location to jump to

Method Code Generation

- A method instance is compiled much like a function
  - It turns into machine code that resides at a particular address in the instruction space
  - For example, the method instance *Truck_move* has an entry point at machine-code label *Truch_move*



- Each class descriptor contains a pointer to its parent class, and also a list of method instances

Instruction Space

Method Dispatch: Static Methods

Method dispatch = generating method calls

- Static method. To compile c.f(), the compiler:
  - Finds the class of c, suppose it is class C
  - Searches in class C for a method f, suppose none is found
  - Searches the parent class of C, suppose it is class B, and so on
  - Suppose in some ancestor class A it finds a static method f ➔ compile a function call to label A_f

c.f() ⟹ **A_f**

Method Dispatch: Dynamic Methods

- What if method m in A is a dynamic method?

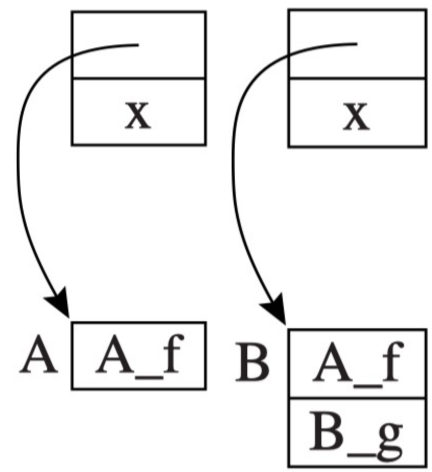
```
class A {
  int x;
  void m() { println("m in A"); }
  void n() { println("n in A"); }
}
class B extends A {
  int y;
  void m() { println("m in B"); }
  void o() { println("o in B"); }
}
void f(A a) { a.m(); }
```

which implementation of m should be invoked?

Method Dispatch: Dynamic Methods

- **Each class** is associated with a **dispatch vector** (aka virtual table, vtable)
  - record of function pointers – one for each method
- **Each object** is associated with a record, with one field for the dispatch vector of its class

```
Class A extends Object {
      var x := 0
      method f() }
Class B extends A {
      method g() }
```

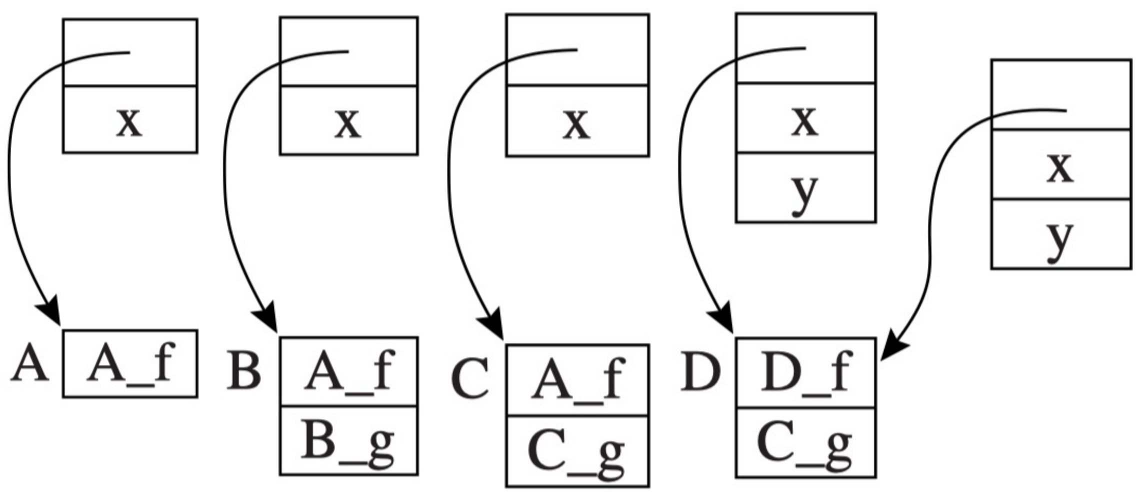


Dispatch vector enables dynamic dispatch

Method Dispatch: Dynamic Methods

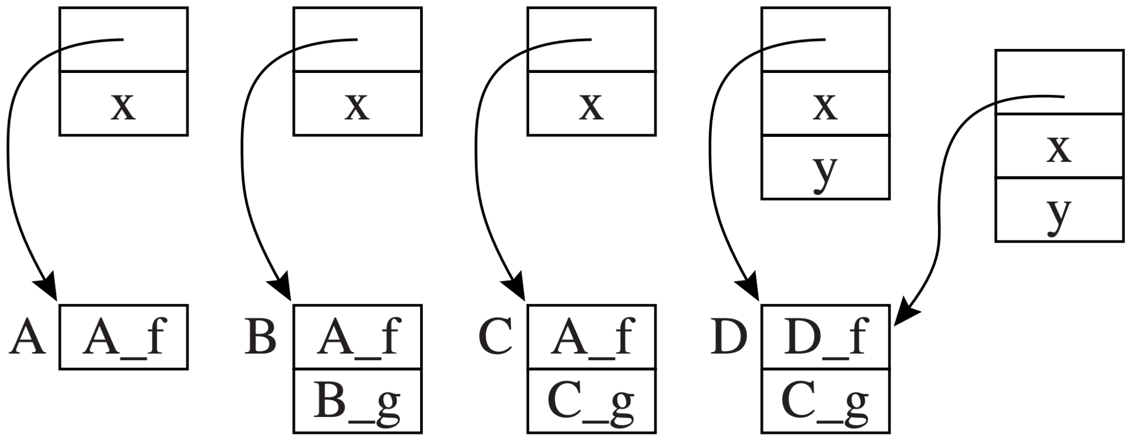
- When class B inherits from A, the method table starts with entries for all method names known to A, and then continues with new methods declared by B
- Just like the arrangement of fields in objects with filed inheritance

```
Class A extends Object {
        var x := 0
        method f() }
Class B extends A {
        method g() }
Class C extends B {
        method g() }
Class D extends C {
        var y := 0
        method f() }
```



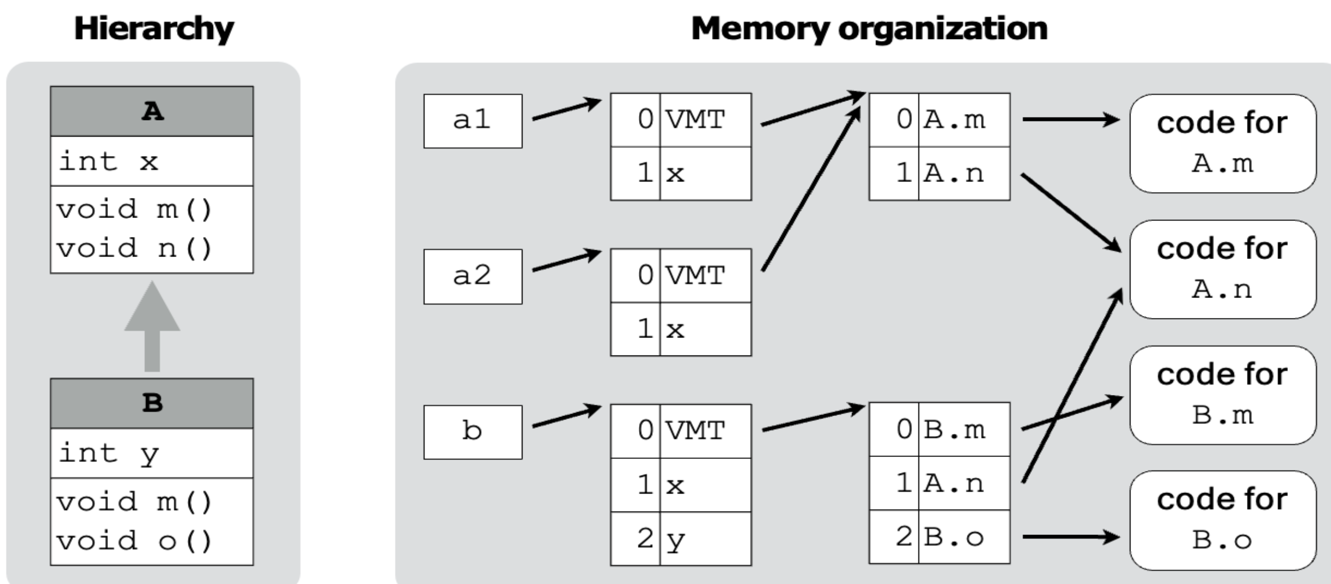
Method Dispatch: Dynamic Methods

- To execute c.f(), where f is a dynamic method, the compiled code must execute these instructions:
  1. Fetch the class descriptor *d* at offset 0 from c
  2. Fetch the method-instance pointer p from the (constant) f offset of d
  3. Call p (Jumps to address p, saving return address)



Example: Method Dispatch

- Method pointers are stored sequentially, starting with those of the superclass, in a **virtual methods table (VMT)** shared by all instances of the class



Program

```
A a1 = new A();
A a2 = new A();
B b = new B();
```

3. Multiple Inheritance

- Graph Coloring
- Hashing

Multiple Inheritance

- Many languages allow classes to inherit from multiple parents
  - E.g., C++, Perl, Python

- In languages that permit a class D to extend several parent classes A, B and C, finding field offsets and method instances is more difficult
  - E.g., it is impossible to put all the A fields at the beginning of D *and* to put all the B fields at the beginning of D

Problems of Multiple Inheritance

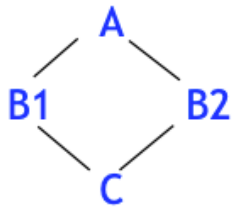
- **Problem 1: Ambiguity**

```
class A { int m(); }
class B { int m(); }
class C extends A, B {} // which m?
```

All methods, files must be uniquely defined

- **Problem 2: field replication**

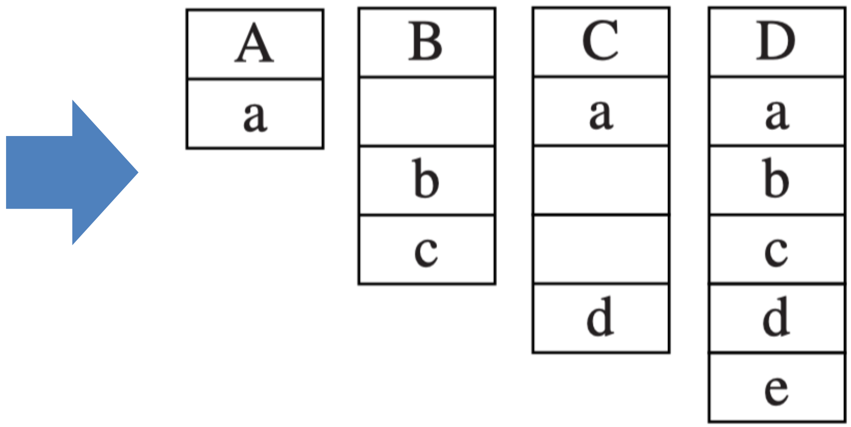
```
class A { int x; }
class B1 extends A { … }
class B2 extends A { … }
class C extends B1, B2 { … }
```



Global Graph Coloring - Fields

- **Goal**: Statically analyze all classes at once, finding some offset for each field name that can be used in every record containing that field
- **Idea**: formulate as a graph-coloring problem

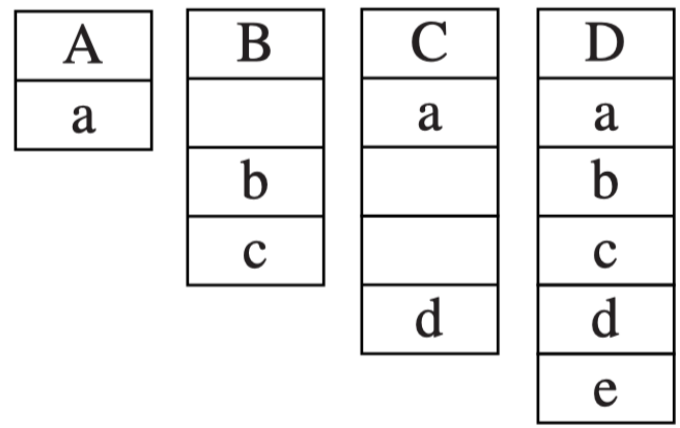
```
class A extends Object {var a := 0 }
class B extends Object {var b:=0
                           var c:= 0 }
class C extends A {var d:=0 }
class D extends A,B,C { var e:=0 }
```



Global Graph Coloring - Fields

- **Goal**: Statically analyze all classes at once, finding some offset for each field name that can be used in every record containing that field
- **Idea**: formulate as a graph-coloring problem

```
class A extends Object {var a := 0 }
class B extends Object {var b:=0
                           var c:= 0 }
class C extends A {var d:=0 }
class D extends A,B,C { var e:=0 }
```



Graph-coloring

- node: a distinct field name
- edge: two fields coexist in a class
- colors: offsets (0, 1, 2 ...)!

如果不同field在同一个类型里出现了，那么它们就不能染成同一个颜色(也就是不能放在同一个位置)！

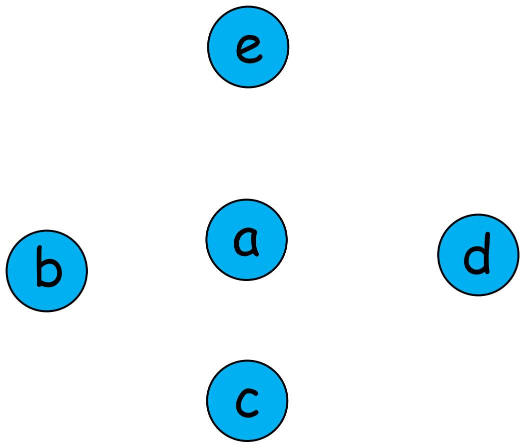
Global Graph Coloring - Fields

- **Step I**: interference graph construction
  - For each pair of instance variable (x, y), draw an edge between x and y, if x and y can not be in same position

code

```
class A extends Object {var a := 0 }
class B extends Object {var b:=0
                        var c:= 0 }
class C extends A {var d:=0 }
class D extends A,B,C { var e:=0 }
```

graph



如果不同field在同一个类型里出现了，那么它们就不能染成同一个颜色(也就是不能放在同一个位置)！

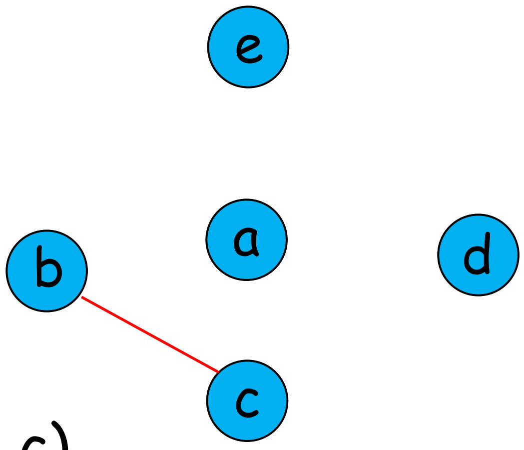
Global Graph Coloring - Fields

- **Step I**: interference graph construction
  - For each pair of instance variable (x, y), draw an edge between x and y, if x and y can not be in same position

code

```
class A extends Object {var a := 0 }
class B extends Object {var b:=0
                        var c:= 0 }
class C extends A {var d:=0 }
class D extends A,B,C { var e:=0 }
```

graph



Consider class B (b, c)

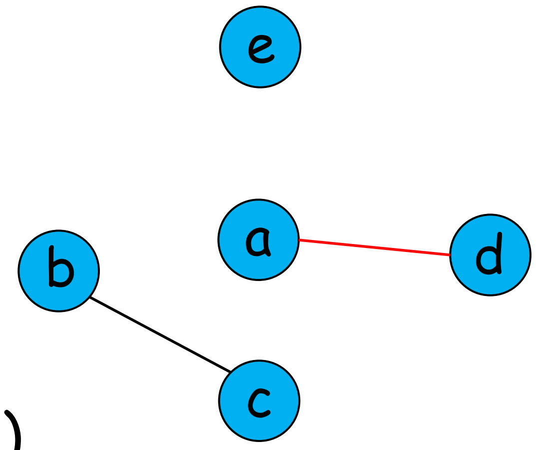
Global Graph Coloring - Fields

- **Step I**: interference graph construction
  - For each pair of instance variable (x, y), draw an edge between x and y, if x and y can not be in same position

code

```
class A extends Object {var a := 0 }
class B extends Object {var b:=0
                        var c:= 0 }
class C extends A {var d:=0 }
class D extends A,B,C { var e:=0 }
```

graph



Consider class C (a, d)

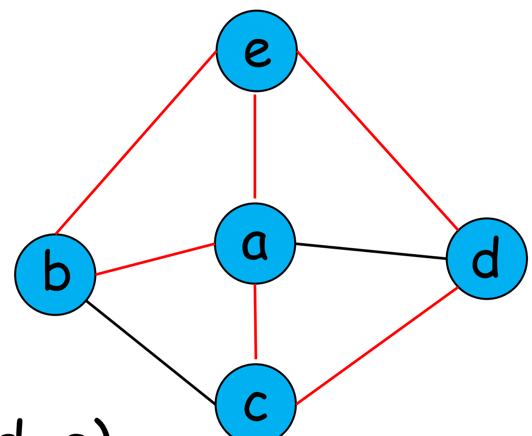
Global Graph Coloring - Fields

- **Step I**: interference graph construction
  - For each pair of instance variable (x, y), draw an edge between x and y, if x and y can not be in same position

code

```
class A extends Object {var a := 0 }
class B extends Object {var b:=0
                        var c:= 0 }
class C extends A {var d:=0 }
class D extends A,B,C { var e:=0 }
```

graph



Consider class D (a, b, c, d, e)

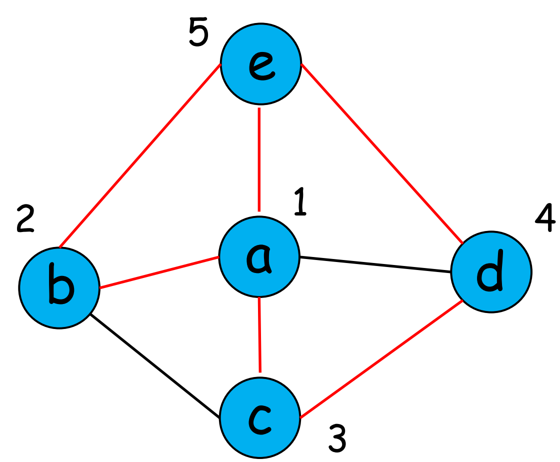
Global Graph Coloring - Fields

- **Step II**: coloring
  - Assign a color (offset) for each node, adjacent nodes are of different colors

code

```
class A extends Object {var a := 0 }
class B extends Object {var b:=0
                        var c:= 0 }
class C extends A {var d:=0 }
class D extends A,B,C { var e:=0 }
```

graph

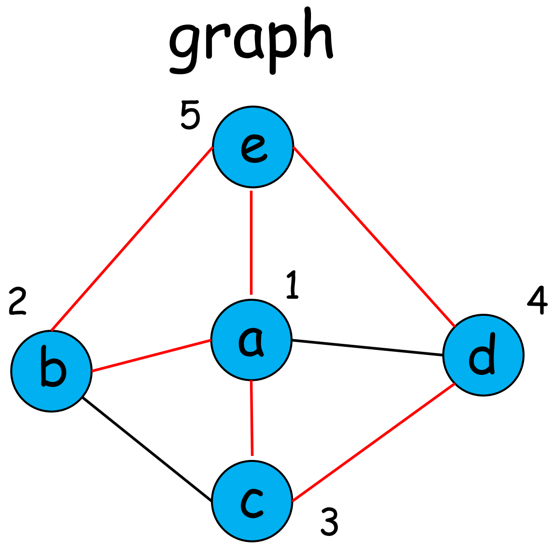


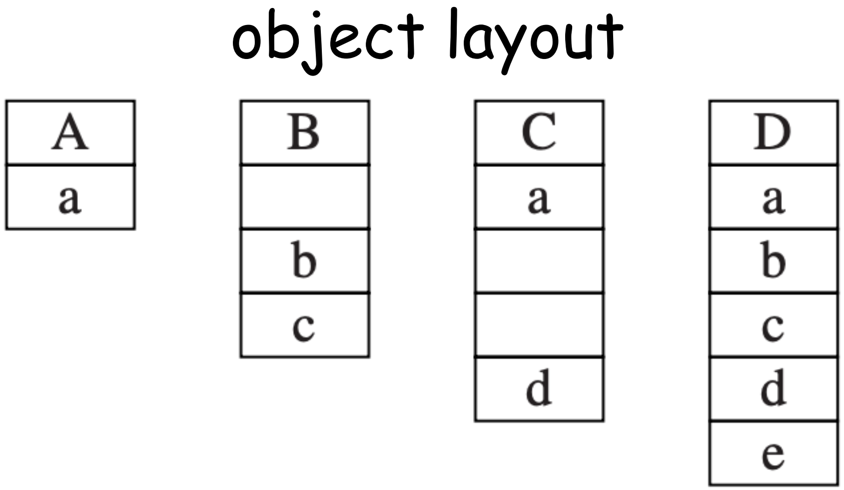
coloring (with offset)

Global Graph Coloring - Fields

- **Step III**: determining layout

```
class A extends Object {var a := 0 }
class B extends Object {var b:=0
                        var c:= 0 }
class C extends A {var d:=0 }
class D extends A,B,C { var e:=0 }
```



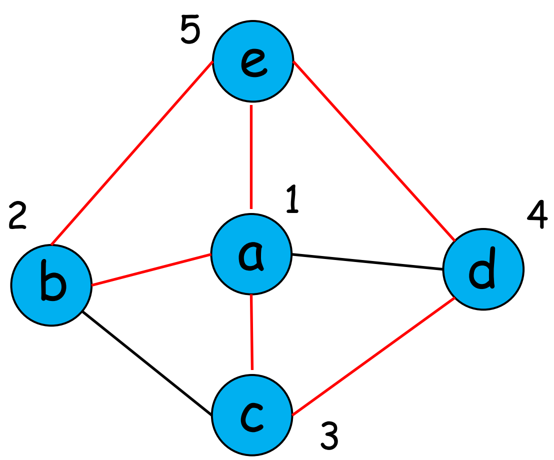


Global Graph Coloring - Fields

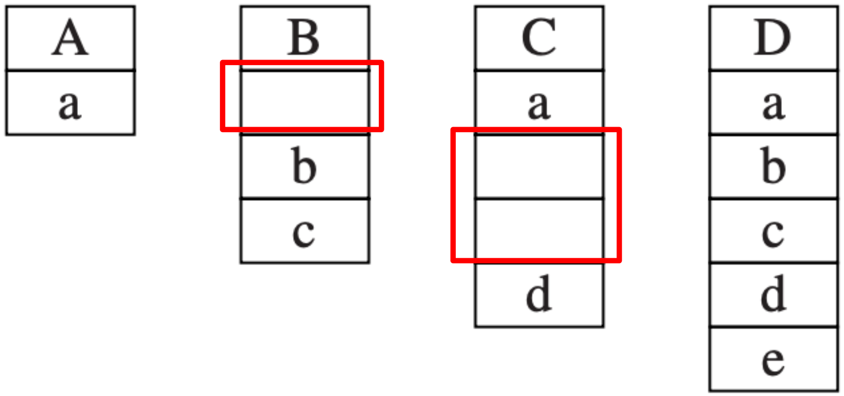
- **Step III**: determining layout

```
class A extends Object {var a := 0 }
class B extends Object {var b:=0
                        var c:= 0 }
class C extends A {var d:=0 }
class D extends A,B,C { var e:=0 }
```

graph



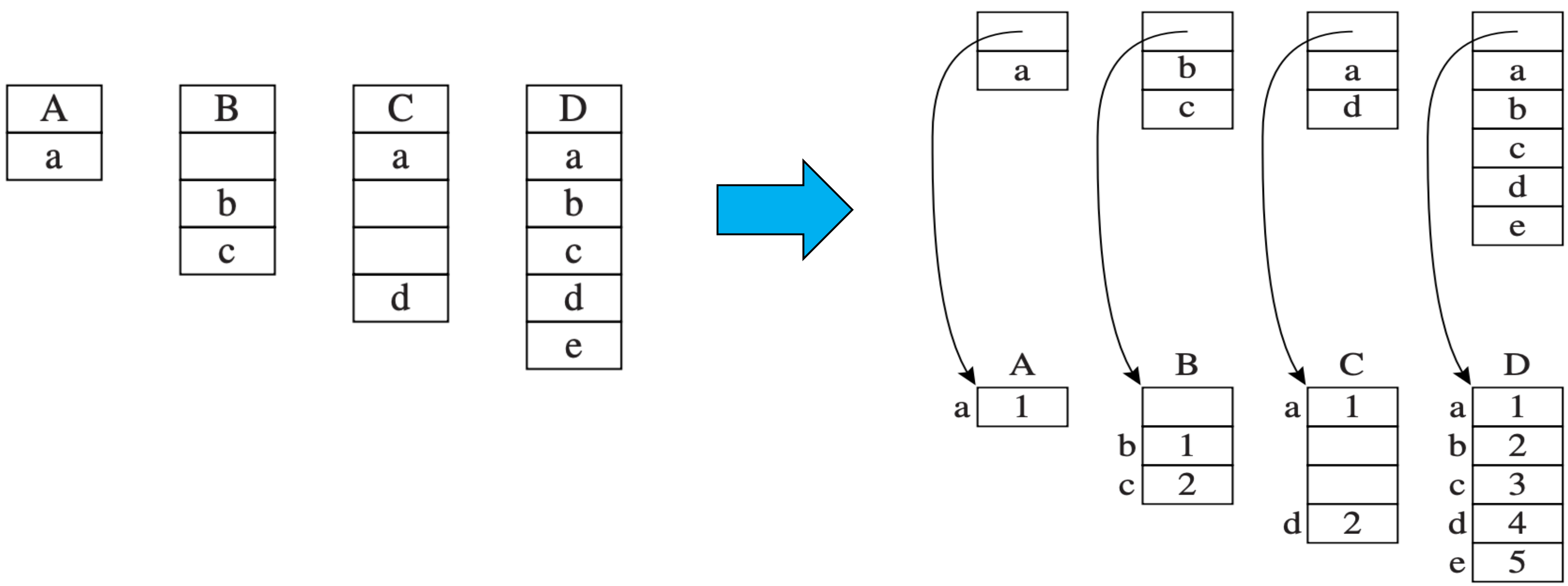
object layout



Problem: wasted empty slots!

Advanced Solution: Coloring on Classes

- Pack the fields of each object and have the class descriptor tell where each filed is
  1. Real colors (offsets) are presented in class descriptors
  2. Fields are compacted in objects



Advanced Solution: Coloring on Classes

- Why preferring class descriptors for coloring?
  - The number of types << the number of objects

- **Problem**: the offset for each field is not fixed
  - E.g., the offset for b is 0 in B, 1 in D
  - Dynamic lookup is required for field access
    1. Fetch the descriptor-pointer from the object
    2. Fetch the field-offset value from the descriptor
    3. Fetch (or store) the data at the appropriate offset in the object

What About Methods?

- Still using the global coloring strategy
  - Method names can be mixed with field names in the same graph
    - Field entries -> offset
    - Method entries -> code address for method
- The cost for dynamic lookup is similar
  - Single inheritance also requires a lookup

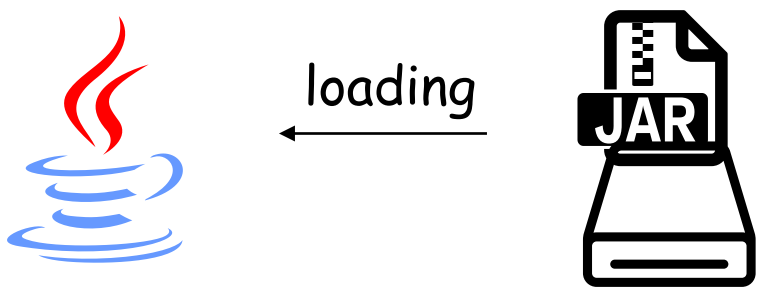
3. Multiple Inheritance

- Graph Coloring
- Hashing

New Problem with Dynamic Linking

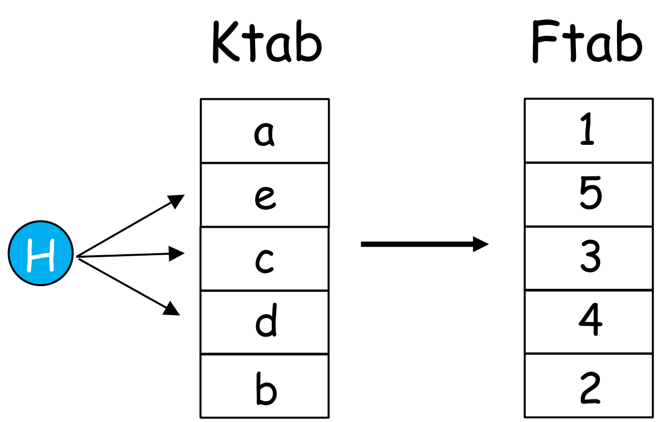
- **Global graph coloring** assumes classes are statically linked together
  - A special-purpose linker can help achieve that

- However: many OO systems can load classes during runtime
  - E.g., Java‘s dynamic class loader



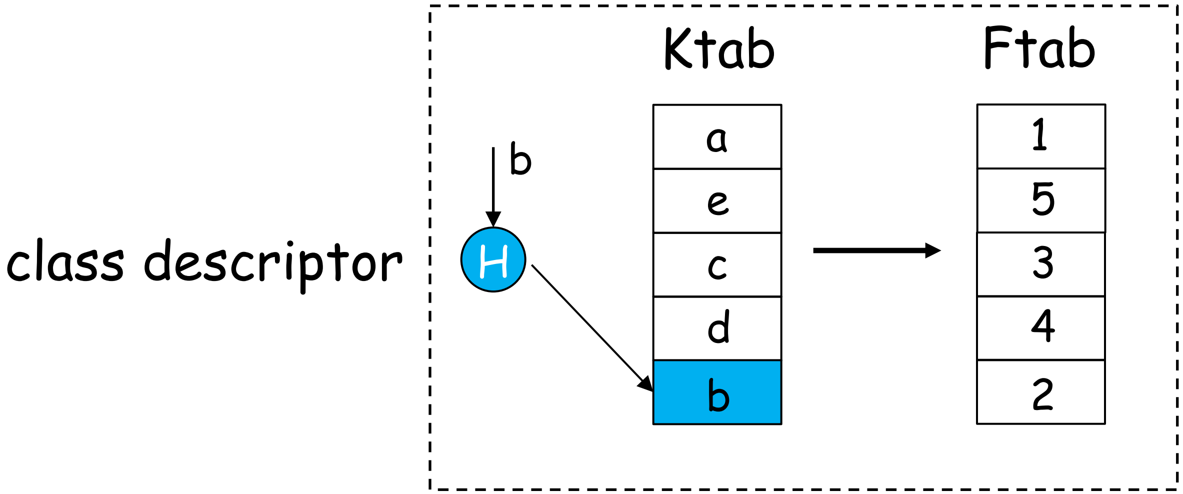
Solution: Hashing

- Building a hash table for each class descriptor
  - Ftab (field table): containing field offsets and method code address
  - Ktab (key-table): containing field/method names
- The table is agnostic to multiple inheritance
  - Fields do not need to have fixed offsets



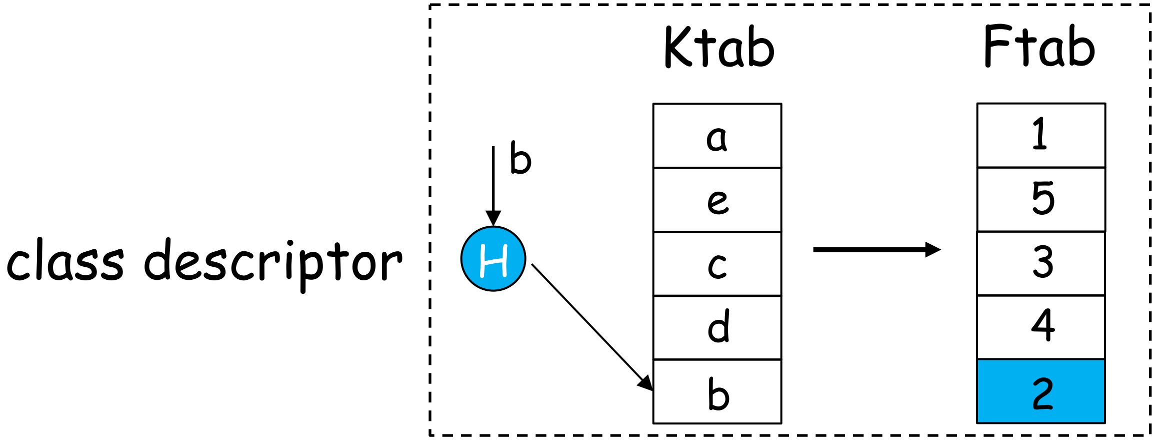
Solution: Hashing

- Steps to fetch a field (say field b)
  - Fetch the class descriptor at offset 0
  - Fetch the field name from offset Ktab + hash_b
  - Compare field name with the input name (EQUAL!)



Solution: Hashing

- Steps to fetch a field (say field b)
  - Fetch the class descriptor at offset 0
  - Fetch the field name from offset Ktab + $hash_b$
  - Compare field name with the input name (EQUAL!)
  - Fetch the field offset from Ftab + $hash_b$ (2)
  - Fetch the field from object + 2



4. Testing Class Membership

The Membership Test Problem

- How to check efficiently at run time that an object has a given type?

- This problem must be solved often, e.g. in Java:
  - when the instanceof operator is used,
  - when a type cast is performed,
  - when an exception is thrown (to find the matching handler)
  - Etc.

Example: Which Type Cast Is Safe?

- **Casting to a super type is always safe (upcast)**
  - Fields/methods in the super class can be accessed by the sub-class
- **Casting to a sub-type is not (downcast)**
  - Child class may define new methods/fields not present in the super class
- How to allow upcast while avoid incorrect downcast?

Type Testing and Casting

- A normal type testing and casting would be:

```
if (a.isClass(A)) {
  A b = (A)a;
  b.somethod();
}
```

- OO languages have supported this feature

	Modula-3	Java
Test whether object x belongs class `C`, or to any subclass of `C`.	`ISTYPE(x,C)`	`x instanceof C`
Given a variable x of class `C`, where x actually points to an object of class `D` that extends `C`, yield an expression whose compile-time type is class `D`.	`NARROW(x,D)`	`(D)x`

TABLE 14.6 Facilities for type testing and safe casting

Testing Class Membership: Instanceof?

- A simple way is to perform the following loop:
  - Recursively compare types with the input type (C)

```
        t1 ← x.descriptor
L1:     if t1 = C goto true
        t1 ←t1.super
        if t1 = nil goto false
        goto L1
```

Can be slow

where $t1$.super is the superclass (parent class) of class $t1$.

- The recursive comparison takes time
  - Can we have a faster approach?

Solution: Display

- Each class descriptor stores a display
- Assume that the class nesting depth is limited to some constant, such as 20. Reserve a 20-word block in each class descriptor

Solution: Display

- Each class descriptor stores a display
- Assume that the class nesting depth is limited to some constant, such as 20. Reserve a 20-word block in each class descriptor
- E.g., class D extends A extends B extends Object
  - The display should look like below:

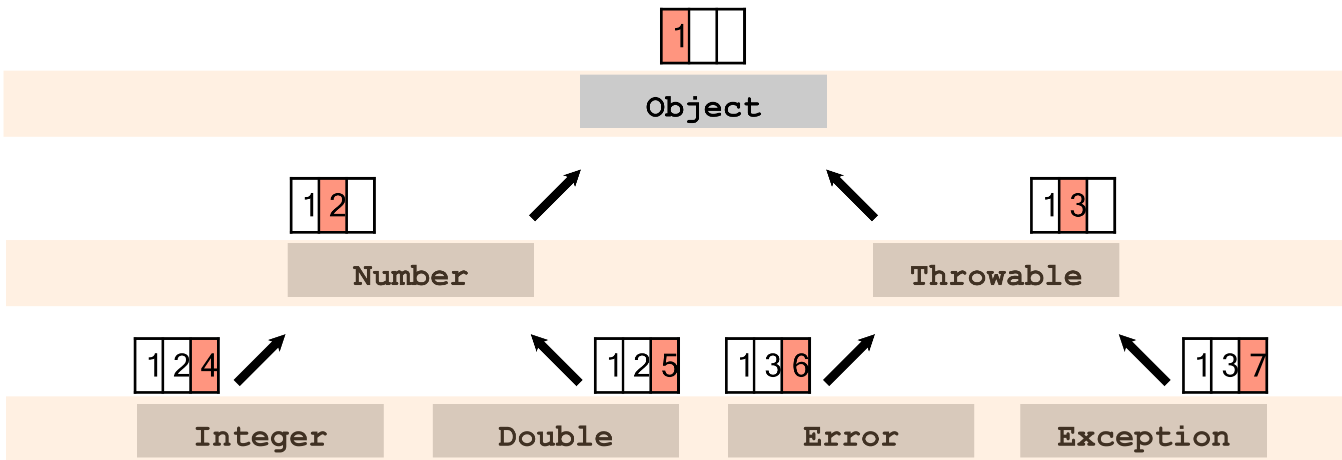
0	Object
1	B
2	A
3	D
	⋮

- We may give each class a numerical identifier for ease of comparison (e.g., A, B, D correspond to different numbers)

Norman H. Cohen. Type-extension type tests can be performed in constant time. ACM Transactions on Programming Languages and Systems (TOPLAS), 1991

Example: Display

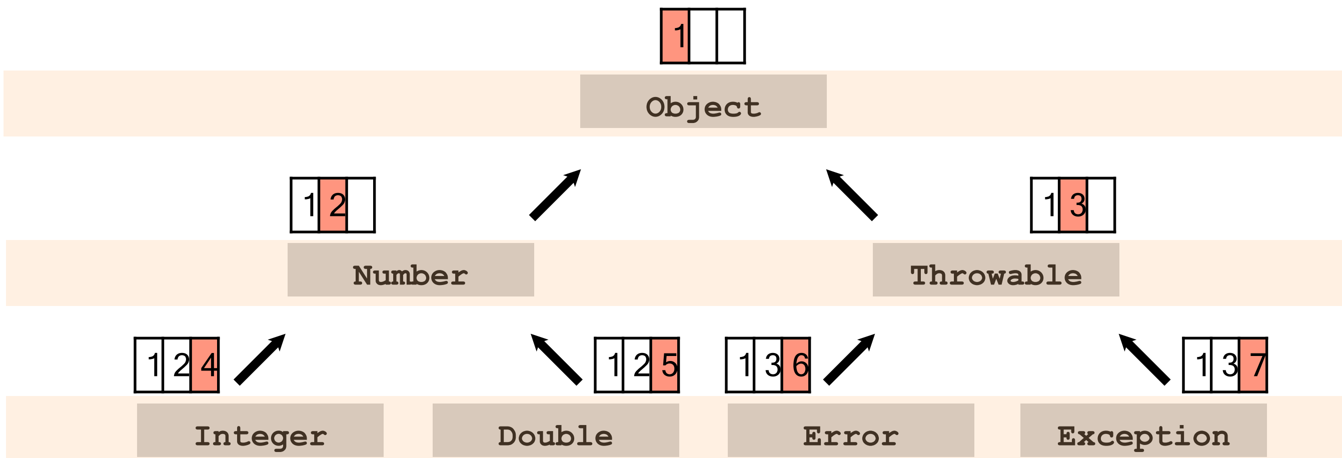
- Assume that we have given each class a numerical identifier



- Suppose the nesting depth of class D is j (which can be known at compile time). In D's descriptor,
  - display[j] = D
  - display[j-1] = D.super
  - display[j-2] = D.super.super,
  - …,
  - display[0] = Object
  - display[k] = nil, where k > j

Example: Display

- Assume that we have given each class a numerical identifier



- Suppose the nesting depth of class D is j (which can be known at compile time). In D's descriptor,
  - display[j] = D
  - display[j-1] = D.super
  - display[j-2] = D.super.super,
  - …,
  - display[0] = Object
  - display[k] = nil, where k > j

Solution: Display

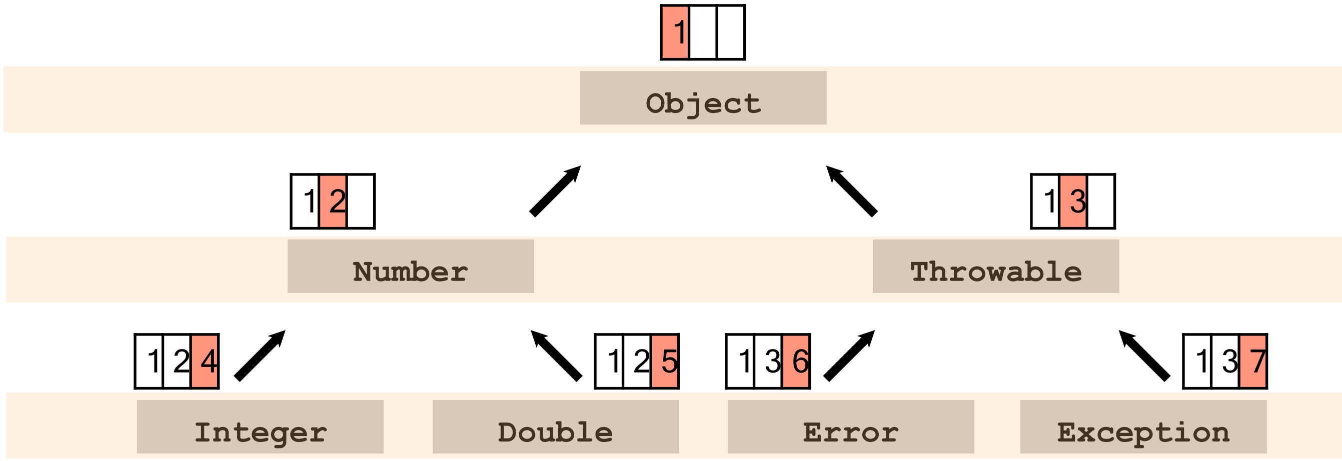
- Each class descriptor stores such a display
  - E.g., class D extends A extends B extends Object

- To implement x instanceof D:
  1. Fetch the class descriptor at offset 0
  2. Fetch the i-th class-pointer slot
  3. Compare with D (which could be a numerical identifier)

0	Object
1	B
2	A
3	D
	⋮

Example: Display (Cont.)

- Assume that we have given each class a numerical identifier



`x` **`instanceof`** `Throwable` ⇔

`x.display[1] == 3`

5. Private Fields and Methods

Private Fields and Methods

- The private keyword can be used for information hiding
  - Private fields/methods cannot be accessed outside the object
- Privacy is enforced by type-checking
  - Encountering c.x/c.f() -> check if x/f is private

Private Fields and Methods

- Different languages have different protection rules for private fields/methods
  - Accessible only to the class that declares them
  - Accessible to the declaring class and any subclasses (*protected* in C++)
  - Accessible only within the same module as the declaring class (package, namespace)
  - Read-only outside the declaring class, but writable by methods of the class

Summary

- Classes
- Single Inheritance
- Multiple Inheritance
- Testing Class Membership
- Private Fields and Methods

Thank you all for your attention