Object-oriented languages

Advanced Compiler Construction
Michel Schinz – 2016–05–19
(parts based on Yoav Zibin's PhD thesis)

Object-oriented languages

In a class-based **object-oriented** (**OO**) language, all values are objects, belonging to a class.

Prototype-based OO languages do not have a concept of class, but we will not cover them here.

Objects encapsulate both **state**, stored in fields, and **behavior**, defined by methods.

Two of the most important features of OO languages are inheritance and polymorphism.

Inheritance

Inheritance is a code reuse mechanism that enables one class to inherit all fields and methods of another class, called its **superclass**.

Inheritance is nothing but code copying, although it is usually implemented in a smarter way to prevent code explosion.

Subtyping and polymorphism

In typed OO languages, classes usually define types. These types are related to each other through a **subtyping** (or **conformance**) relation.

Intuitively, a type T_1 is a subtype of a type T_2 – written $T_1 \sqsubseteq T_2$ – if T_1 has at least the capabilities of T_2.

When $T_1 \sqsubseteq T_2$, a value of type T_1 can be used everywhere a value of type T_2 is expected.

This ability to use a value of a subtype of T where a value of type T is expected is called **inclusion polymorphism**.

Inclusion polymorphism poses several interesting implementation challenges, by preventing the exact type of a value to be known at compilation time.

Subtyping ≠ inheritance

Inheritance and subtyping are not the same thing, but many OO languages tie them together by stating that:

1. every class defines a type, and
2. the type of a class is a subtype of the type(s) of its superclass(es).

This is a design choice, not an obligation!

Several languages also have a way to separate inheritance and subtyping in some cases. For example, Java interfaces make it possible to define subtypes that are not subclasses. C++ has private inheritance that allows the definition of subclasses that are not subtypes.

"Duck typing"

The distinction between inheritance and subtyping is especially apparent in “dynamically typed” OO languages like Smalltalk, Ruby, etc.

In those languages, inheritance is used only to reuse code – no notion of (static) type even exists!

Whether an object can be used in a given context only depends on the set of methods it implements, and not on the position of its class in the inheritance hierarchy.

Polymorphism challenges

The following problems are difficult to solve efficiently because of inclusion polymorphism:

1. **object layout** – arranging object fields in memory,
2. **method dispatch** – finding which concrete implementation of a method to call,
3. **membership tests** – testing whether an object is an instance of some type.

We will now examine each one in turn.

OO problem #1:
Object layout

Object layout

The **object layout problem** consists in finding how to arrange the fields of an object in memory so that they can be accessed efficiently.

Inclusion polymorphism makes the problem hard because it forces the layout of different object types to be compatible in some way.

Ideally, a field defined in a type T should appear at the same offset in all subtypes of T.

Object layout example

```
class A {
  int x;
}
class B extends A {
  int y;
}
void m(A a) { System.out.println(a.x); }
```

at which position in a does x appear?

Case 1:
single inheritance

Single inheritance

In single-inheritance languages where subtyping and inheritance are tied (e.g. Java), the object layout problem can be solved easily as follows:

> The fields of a class are laid out sequentially, starting with those of the superclass – if any.

This ensures that all fields belonging to a type T_1 appear at the same location in all values of type $T_2 \sqsubseteq T_1$.

Example

```
class A {
  int x;
}
```

layout for A

offset	field
0	x

```
class B extends A {
  int y;
}
```

layout for B

offset	field
0	x
1	y

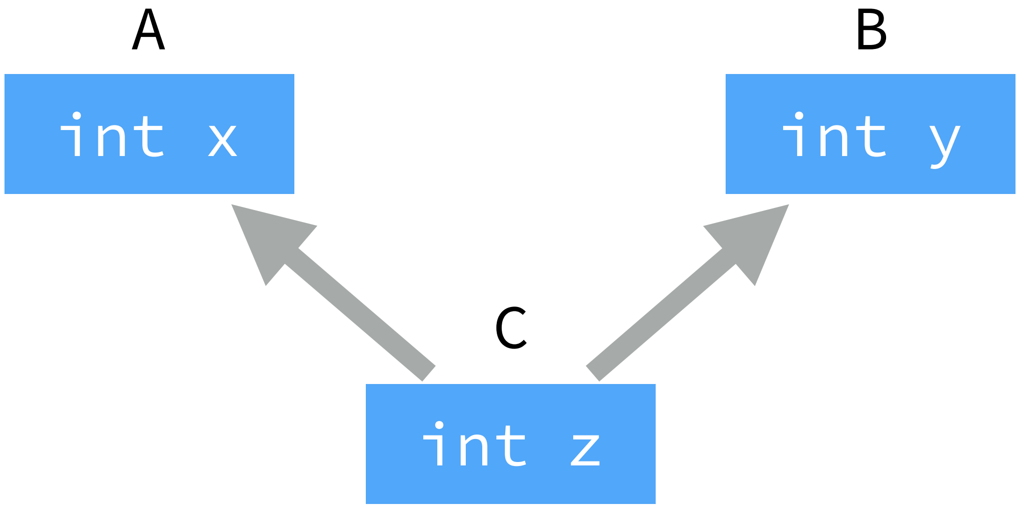
```
void m(A a) { System.out.println(a.x); }
```

access
position `0` of a

Case 2:
multiple inheritance

Multiple inheritance

In a multiple inheritance setting, the object layout problem becomes much more difficult.
For example, in the following hierarchy, how should fields be laid out?



Unidirectional layout

If a standard, unidirectional layout is used, then some space is wasted! Example:

layout for A

offset	field
`0`	`x`

layout for C

offset	field
`0`	`x`
`1`	`y`
`2`	`z`

layout for B

offset	field
`0`	`–`
`1`	`y`

wasted

Bidirectional layout

For this particular hierarchy, it is however possible to use a **bidirectional layout** to avoid wasting space.

layout for A

offset	field
`0`	`x`

layout for B

offset	field
`-1`	`y`

layout for C

offset	field
`-1`	`y`
`0`	`x`
`1`	`z`

Bidirectional layouts

There does not always exist a bidirectional layout that wastes no space.

Moreover, finding an optimal bidirectional layout – one minimizing the wasted space – has been shown to be NP-complete.

Finally, computing a good bidirectional layout requires the whole hierarchy to be known! It must be done at link time, and is not really compatible with Java-style run time linking.

Accessor methods

Another way of solving the object layout problem in a multiple inheritance setting is to always use accessor methods to read and write fields.

The fields of a class can then be laid out freely. Whenever the offset of a field is not the same as in the superclass from which it is inherited, the corresponding accessor method(s) are redefined.

This reduces the object layout problem to the method dispatch problem, which we will examine later.

Other techniques

Bidirectional layout often wastes space, but field access is extremely fast. Accessor methods never waste space, but slow down field access.

Two-dimensional bidirectional layout slows down field access slightly – compared to bidirectional layout – but never wastes space. However, it also requires the full hierarchy to be known.

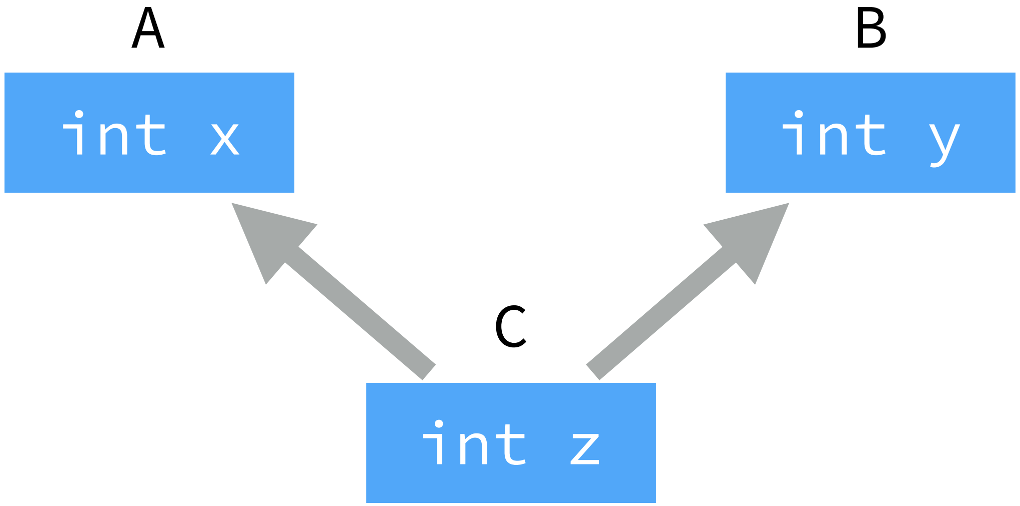
Object layout summary

The object layout problem can be solved trivially in a single-inheritance setting, by laying out the fields sequentially, starting with those of the superclass.

In a multiple-inheritance setting, solutions to that problem are more complicated, and must generally trade space for speed, or speed for space. They also typically require the whole hierarchy to be known in advance.

Exercise

In C++ implementations, the position of a field is not the same in all subtypes of the type that introduced it. For our example, instances of `C` would typically be laid out as follows (gray fields contain information for method dispatch):



A

`int x`

B

`int y`

C

`int z`

offset	field
0	▨
1	x
2	▨
3	y
4	z

With such a layout, what should happen when a method inherited from `B` is invoked on an instance of `C`?

OO problem #2:
method dispatch

Method dispatch

The **method dispatch problem** consists in finding, given an object and a method identity, the exact piece of code to execute.

Inclusion polymorphism makes the problem hard since it prevents the problem to be solved statically – i.e. at compilation time. Efficient dynamic dispatching methods therefore have to be devised.

Method dispatch example

```
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?

Case 1:
single subtyping

Single subtyping

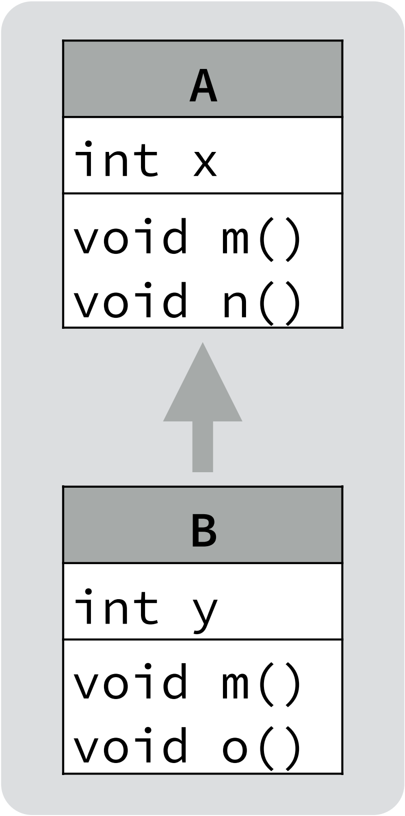
In single-inheritance languages where subtyping and inheritance are tied, the method dispatch problem can be solved easily as follows:

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

This ensures that the implementation for a given method is always at the same position in the VMT, and can be extracted quickly.

Virtual methods table

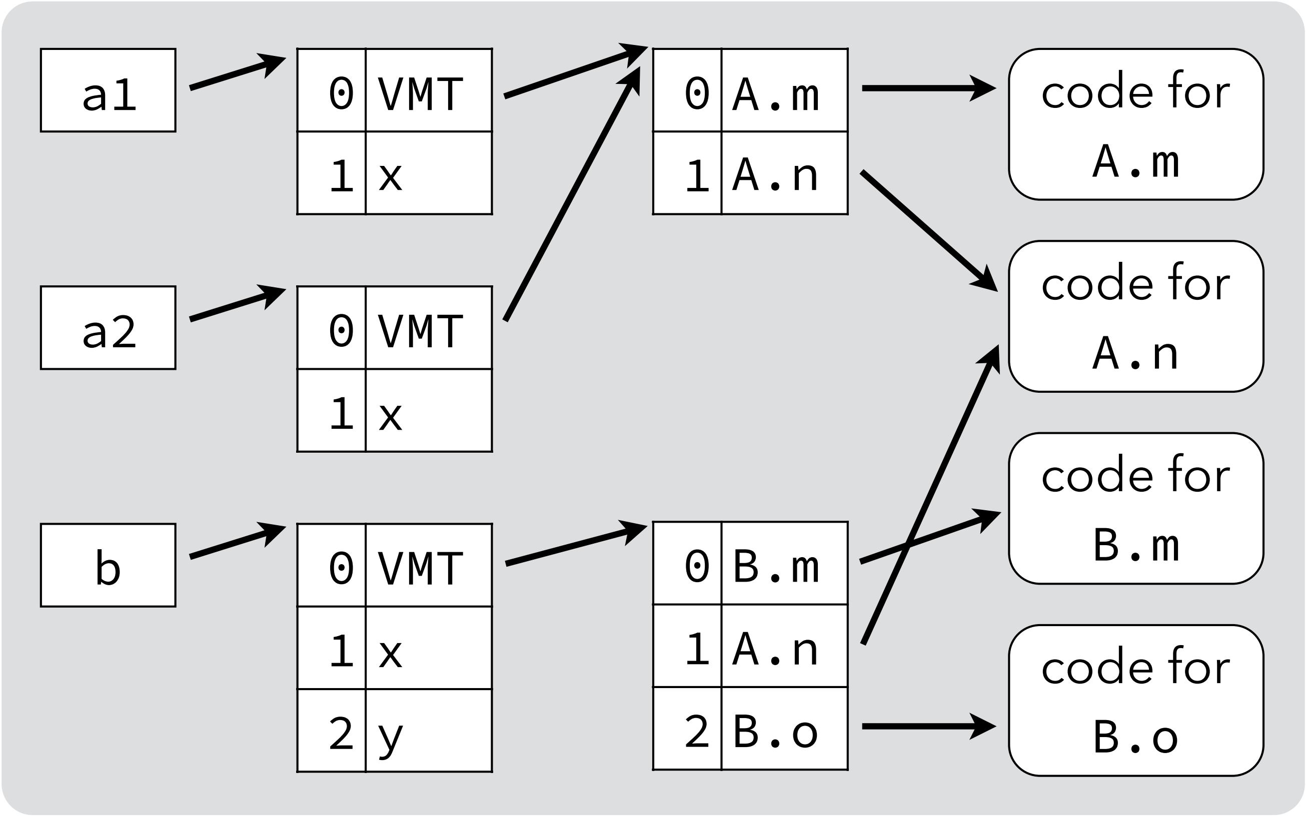
Hierarchy



Program

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

Memory organization



Dispatching with VMTs

Using a VMT, dispatching is accomplished in three steps:

1. the VMT of the selector is extracted,
2. the code pointer for the invoked method is extracted from the VMT,
3. the method implementation is invoked.

Each of these steps typically requires a single – but expensive – instruction on modern CPUs.

VMTs pros and cons

VMTs provide very efficient dispatching, and do not use much memory. They work even in languages like Java where new classes can be added to the bottom of the hierarchy at run time.

Unfortunately, they do not work for dynamic languages or in the presence of any kind of “multiple subtyping” – e.g. multiple interface inheritance in Java.

Case 2:
multiple subtyping

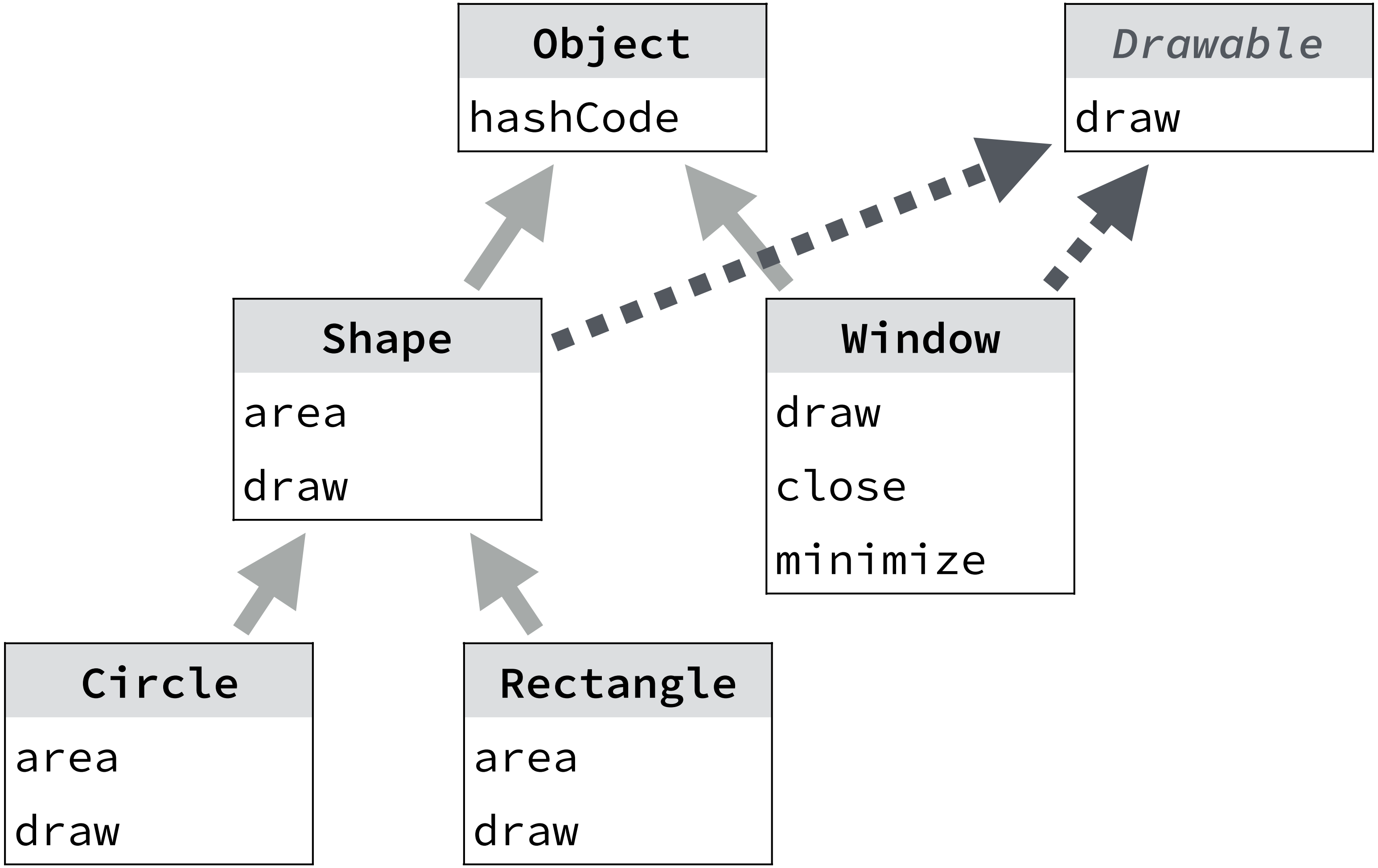
Java interfaces

To understand why VMTs cannot be used with multiple subtyping, consider Java interfaces:

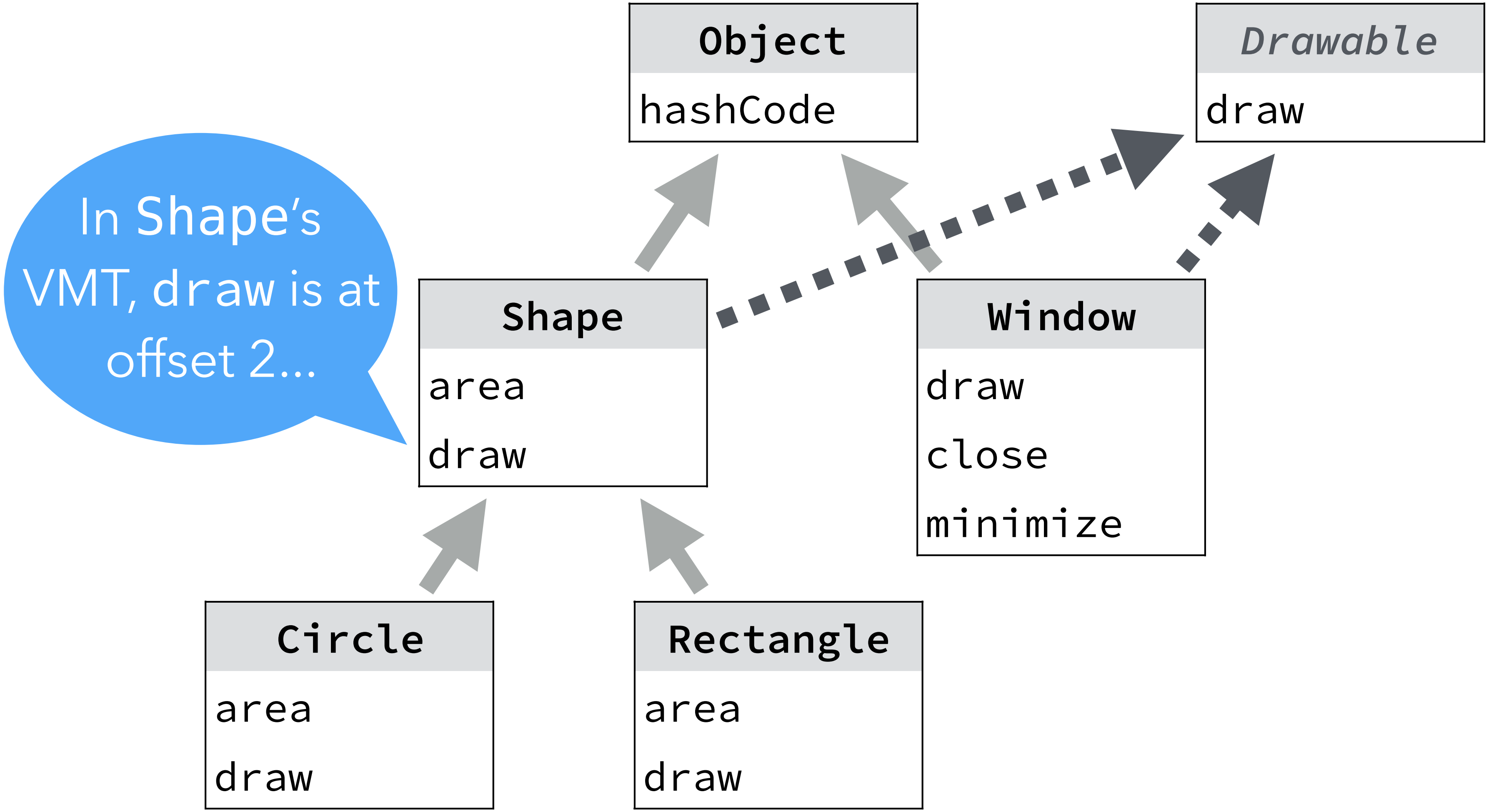
```
interface Drawable { void draw(); }
void draw(List<Drawable> ds) {
  for (Drawable d: ds)
    d.draw();
}
```

When the `draw` method is invoked, the only thing known about d is that it has a `draw` method – but nothing is known about its class, which be anywhere in the hierarchy!

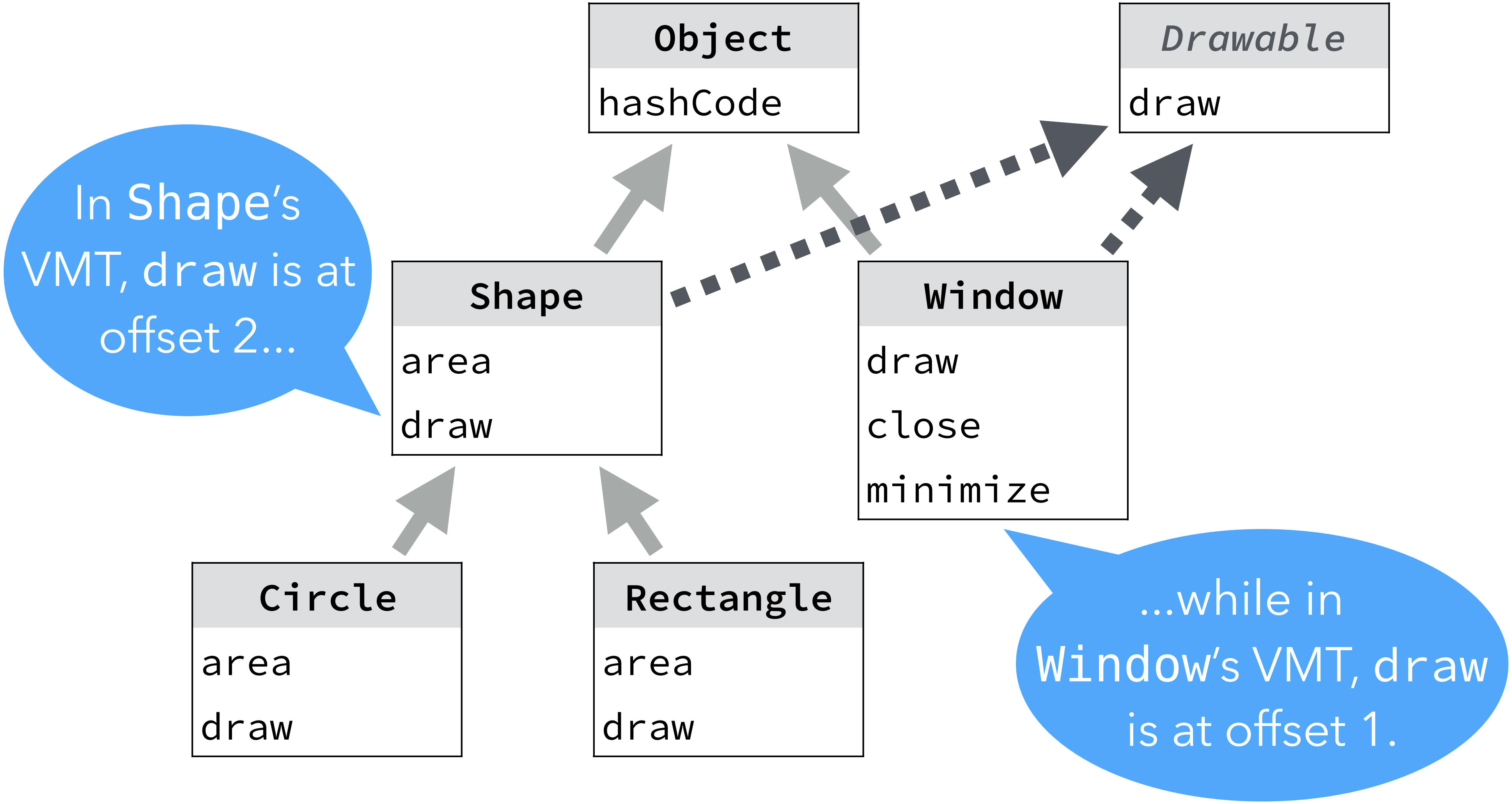
Java interfaces



Java interfaces



Java interfaces



In Shape's VMT, draw is at offset 2…

…while in Window's VMT, draw is at offset 1.

Dispatching matrix

A trivial way to solve the problem is to use a global dispatching matrix, containing code pointers and indexed by classes and methods.

	hashCode	draw	close	minimize	area
Object	hashCode$_O$				
Shape	hashCode$_O$				
Circle	hashCode$_O$	draw$_C$			area$_C$
Rectangle	hashCode$_O$	draw$_R$			area$_R$
Window	hashCode$_O$	draw$_W$	close$_W$	minimize$_W$	

Dispatching matrix

The dispatching matrix makes dispatching very fast.

However, for any non-trivial hierarchy, it occupies so much memory that it is never used as-is in practice.

Various compression techniques have been devised. These techniques usually trade some dispatching efficiency for reduced memory usage. They take advantage of two characteristics of the dispatching matrix:

1. it is sparse, since in a large program any given class implements only a limited subset of all methods,
2. it contains a lot of redundancy, since many methods are inherited.

Null elimination

The dispatching matrix is very sparse in practice. Even in our trivial example, about 50% of the slots are empty.

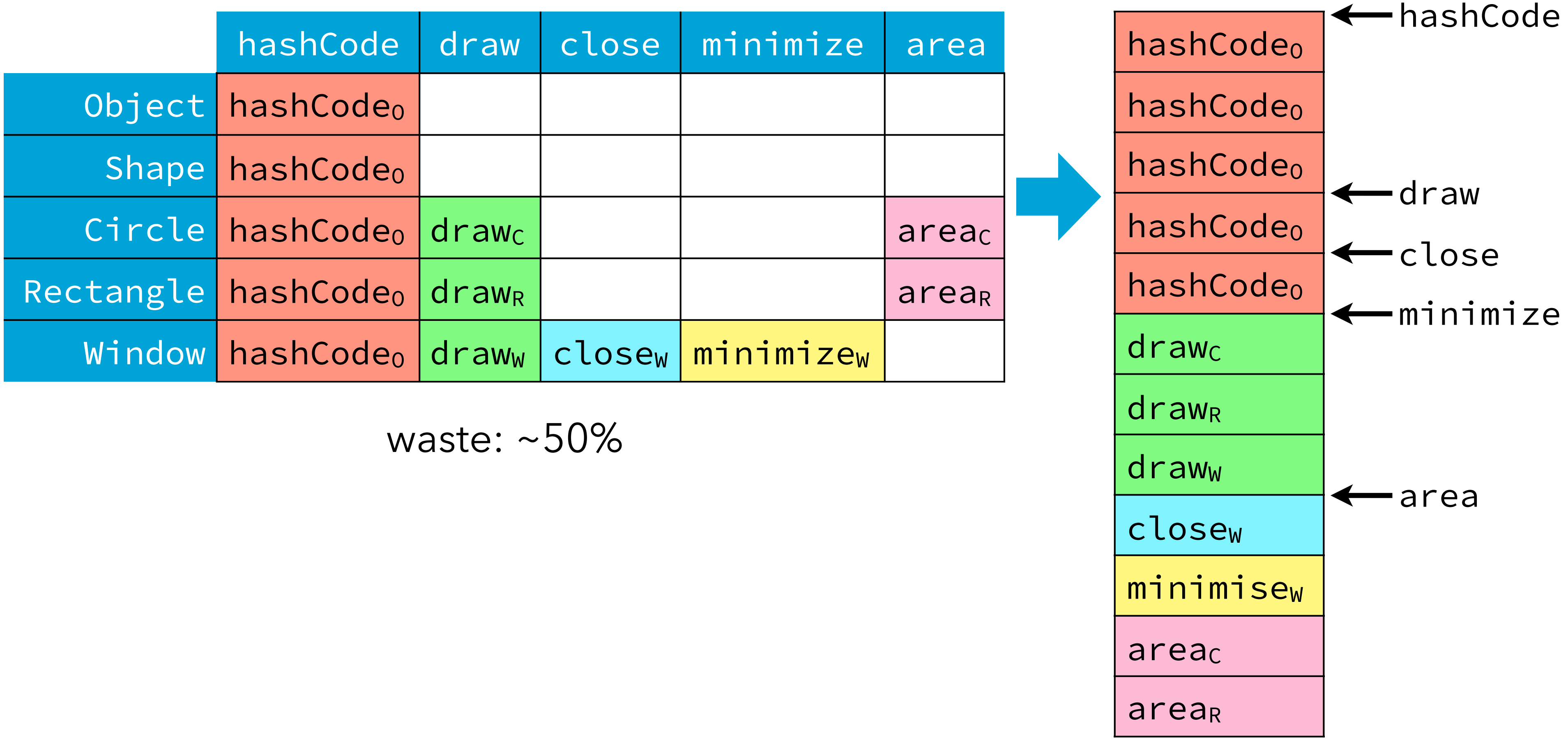
A first technique to compress the matrix is therefore to take advantage of this sparsity. This is called **null elimination**, since the empty slots of the matrix usually contain null.

Several null elimination techniques exist, but we will examine only one: column displacement.

Column displacement

One way of eliminating `nulls` is to transform the matrix into a linear array by shifting either its columns or its rows. Many holes of the matrix can be filled in the process, by carefully choosing the amount by which columns (or, alternatively, rows) are shifted.
This technique is know as **column** (or **row**) **displacement**. In practice, column displacement gives better results than row displacement.

Column displacement



	hashCode	draw	close	minimize	area
Object	hashCode$_O$				
Shape	hashCode$_O$				
Circle	hashCode$_O$	draw$_C$			area$_C$
Rectangle	hashCode$_O$	draw$_R$			area$_R$
Window	hashCode$_O$	draw$_W$	close$_W$	minimize$_W$	

waste: ~50%

hashCode$_O$	← hashCode
hashCode$_O$	
hashCode$_O$	
hashCode$_O$	← draw
hashCode$_O$	← close
draw$_C$	← minimize
draw$_R$	
draw$_W$	
close$_W$	← area
minimise$_W$	
area$_C$	
area$_R$	

waste: none

Dispatching with CD

Dispatching with column/row displacement consists in:

1. adding the offset of the method being invoked – known at compilation time – to the offset of the class of the receptor – known only at run time – to extract the code pointer,
2. invoking the method referenced by that pointer.

It is therefore as fast as dispatching with an uncompressed matrix.

Duplicates elimination

Apart from being sparse, the dispatching matrix also contains a lot of duplicated information. Null elimination does not take advantage of this duplication, and even though it achieves good compression, it is possible to do better.

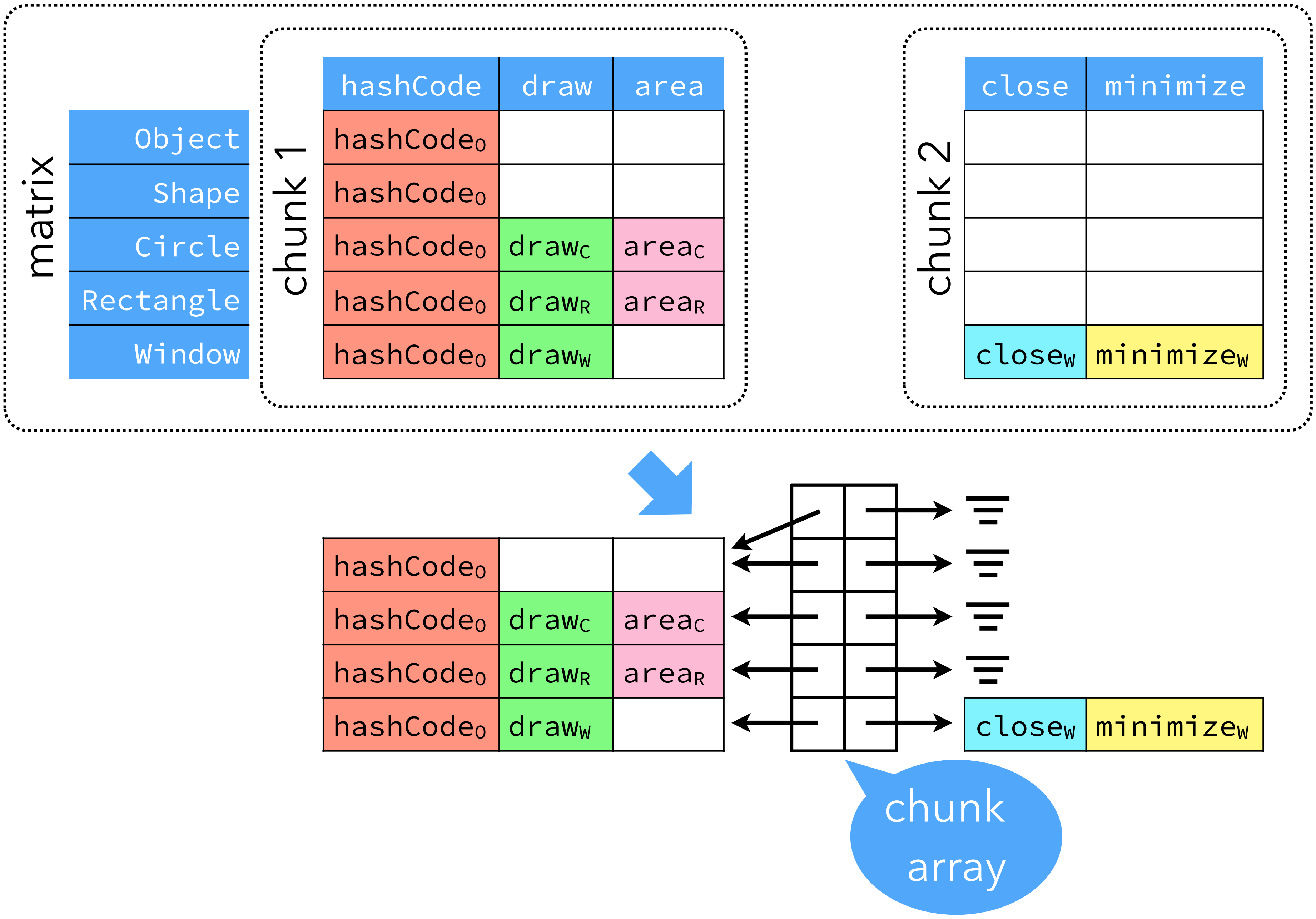
The idea of **duplicates elimination** techniques is to try to share as much information as possible instead of duplicating it.

Compact dispatch table is such a technique.

Compact dispatch tables

The idea of **compact dispatch tables** is to split the dispatch matrix into small sub-matrices called **chunks**.
Each individual chunk will tend to have duplicate rows, which can be shared by representing each chunk as an array of pointers to rows.

Compact dispatch tables



matrix

chunk 1

	hashCode	draw	area
Object	hashCode$_O$		
Shape	hashCode$_O$		
Circle	hashCode$_O$	draw$_C$	area$_C$
Rectangle	hashCode$_O$	draw$_R$	area$_R$
Window	hashCode$_O$	draw$_W$	

chunk 2

close	minimize
close$_W$	minimize$_W$

hashCode$_O$		
hashCode$_O$	draw$_C$	area$_C$
hashCode$_O$	draw$_R$	area$_R$
hashCode$_O$	draw$_W$	

close$_W$	minimize$_W$

chunk array

Dispatching with CDTs

Dispatching with a compact dispatch table consists in:

1. using the offset of the class of the receiver – known only at run time – and the offset and chunk of the method being invoked – known at compilation time – to extract the code pointer,
2. invoking the method referenced by that pointer.

Because of the additional indirection due to the chunk array, this technique is slightly slower than column/row displacement. However, it tends to compress the dispatching matrix better in practice.

Hybrid techniques

VMTs and the more sophisticated techniques handling multiple subtyping are not exclusive.
For example, all Java implementations use VMTs to dispatch when the type of the selector is a class type, and more sophisticated – and slower – techniques when it is an interface type.
The JVM even has different instructions for the two kinds of dispatch: `invokevirtual` and `invokeinterface`.

Method dispatch optimization

Inline caching

Even when efficient dispatching structures are used, the cost of performing a dispatch on every method call can become important.

In practice, it turns out that many calls that are potentially polymorphic are in fact monomorphic.

The idea of **inline caching** is to take advantage of this fact by recording – at every call site – the target of the latest dispatch, and assuming that the next one will be the same.

Implementing inline caching

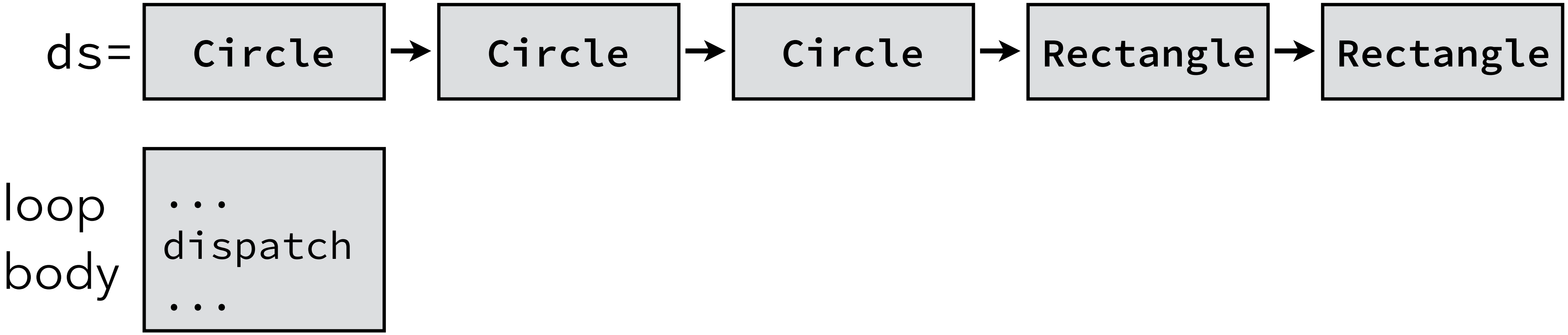
Inline caching works by patching code.

At first, all method calls are compiled to call a standard dispatching function. Whenever this function is invoked, it computes the target of the call, and then patches the original call to refer to the computed target.

All methods have to handle the potential mispredictions of this technique, and invoke the dispatching function when they happen.

Inline caching example

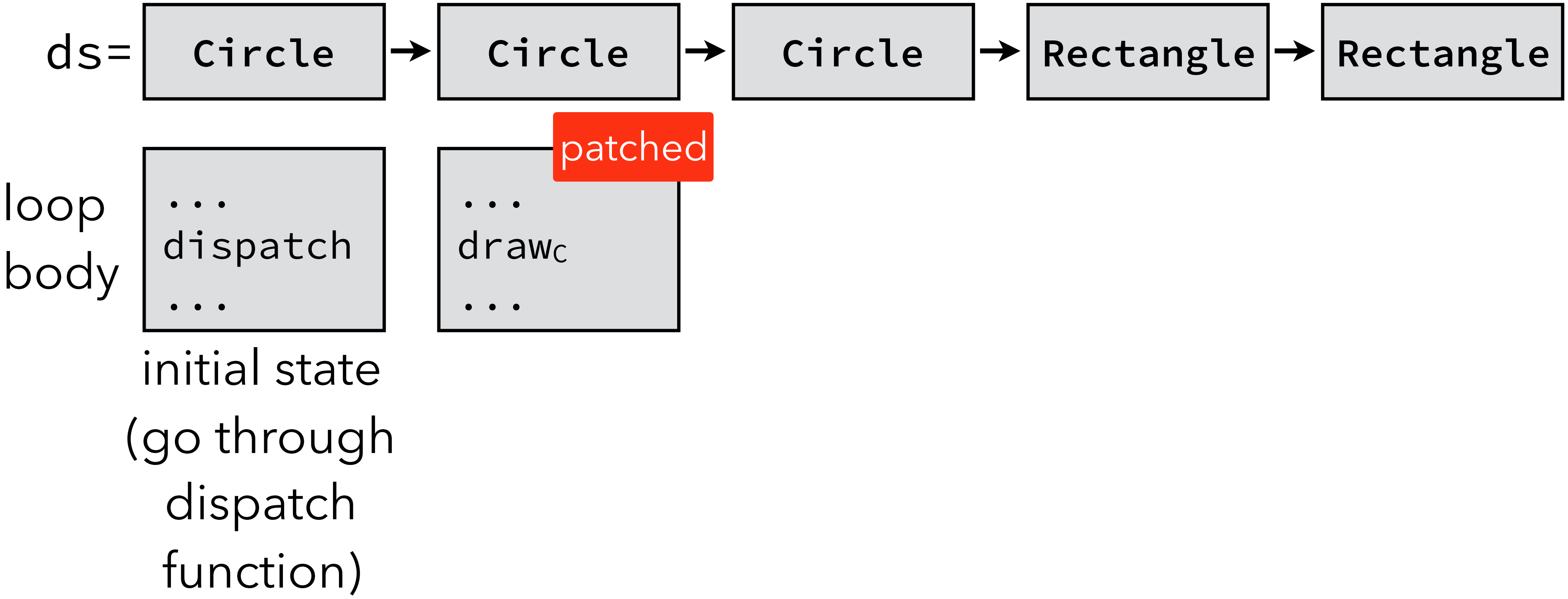
```
for (Drawable d: ds) d.draw();
```



initial state
(go through
dispatch
function)

Inline caching example

```
for (Drawable d: ds) d.draw();
```



ds= Circle → Circle → Circle → Rectangle → Rectangle

loop body

```
...
dispatch
...
```

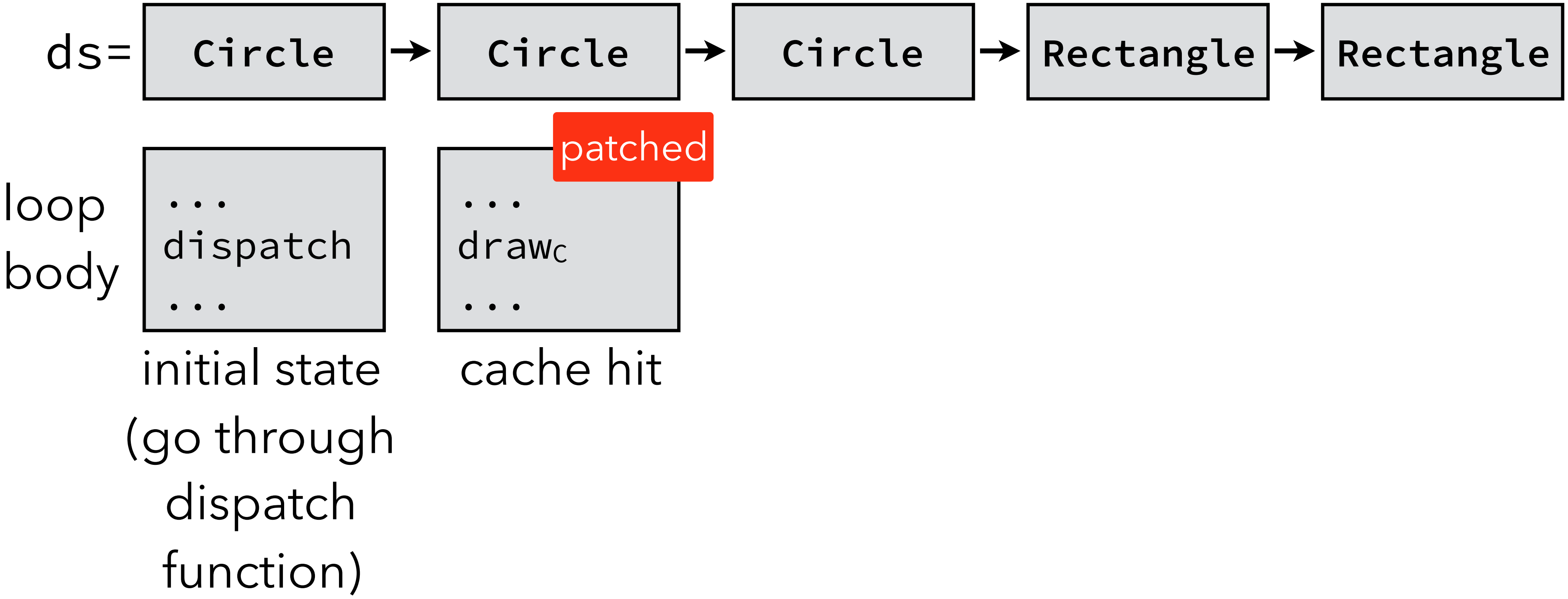
initial state (go through dispatch function)

patched

...
draw$_C$
...

Inline caching example

```
for (Drawable d: ds) d.draw();
```



ds= `Circle` → `Circle` → `Circle` → `Rectangle` → `Rectangle`

loop body

```
...
dispatch
...
```

initial state (go through dispatch function)

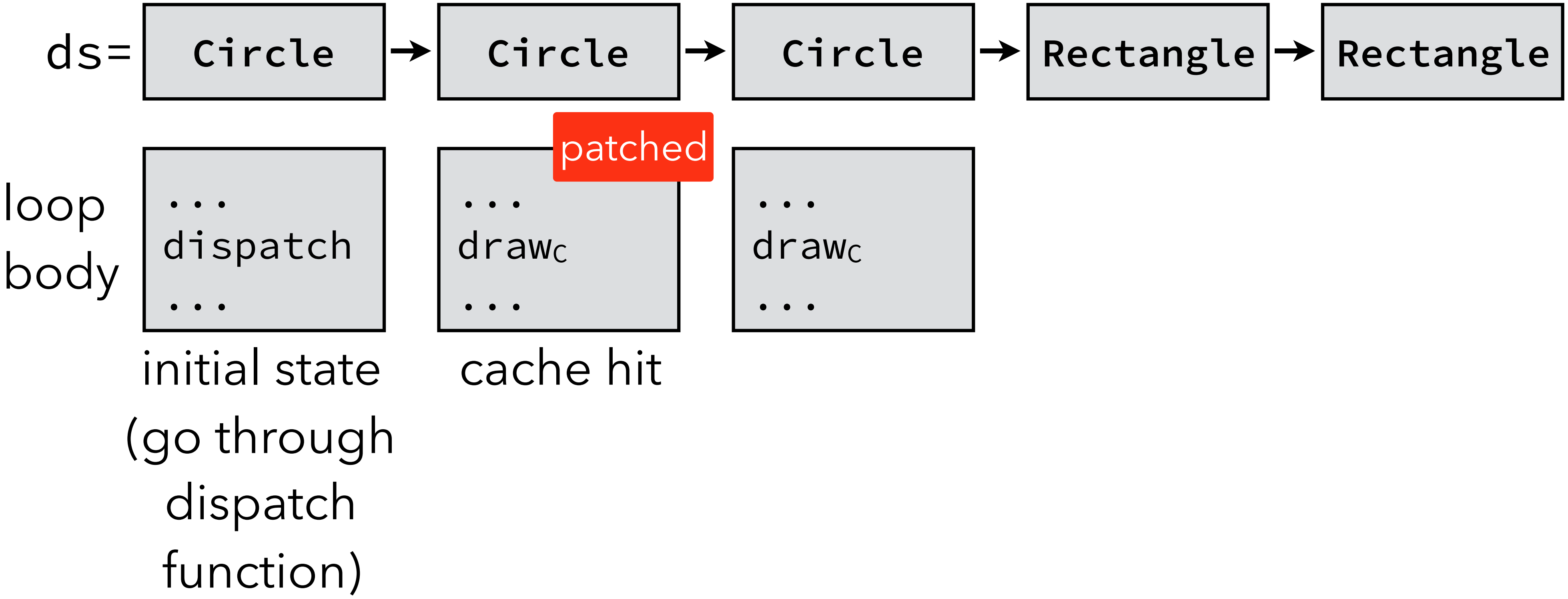
patched

...
draw$_C$
...

cache hit

Inline caching example

`for (Drawable d: ds) d.draw();`



ds= `Circle` → `Circle` → `Circle` → `Rectangle` → `Rectangle`

loop body

```
...
dispatch
...
```

initial state (go through dispatch function)

patched

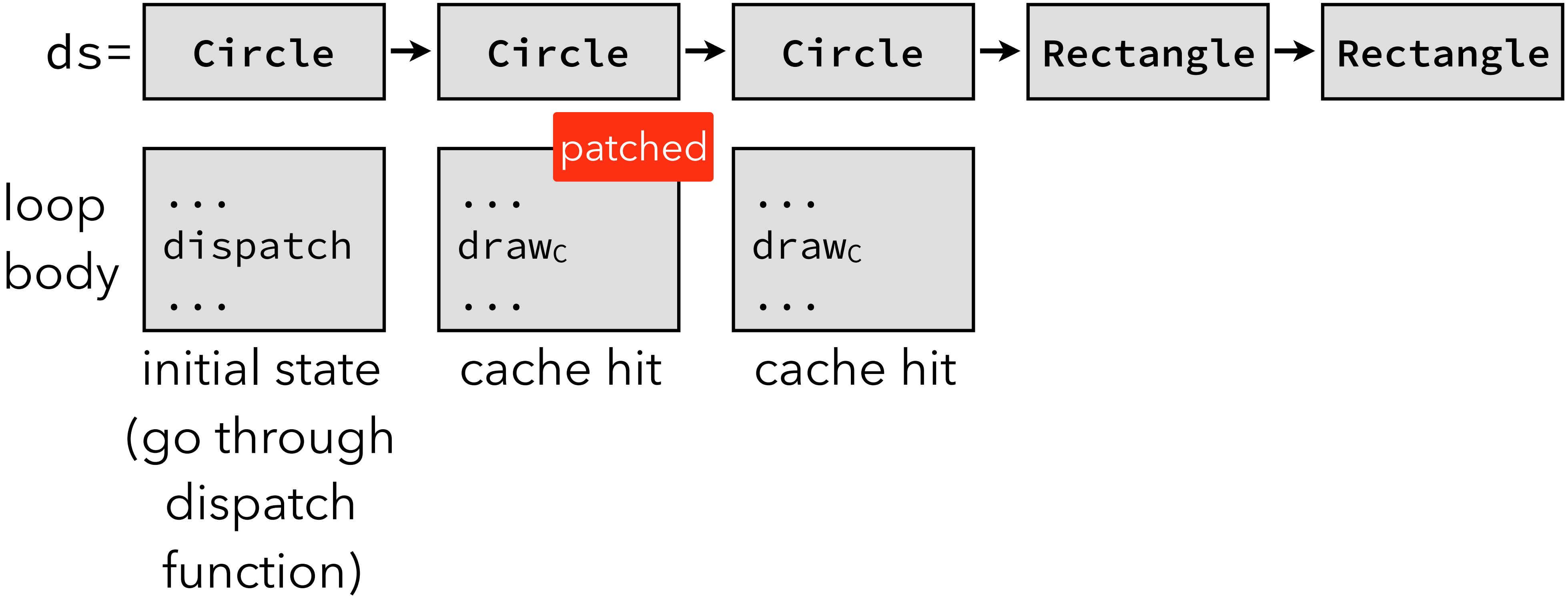
```
...
```
draw$_C$
```
...
```

cache hit

```
...
```
draw$_C$
```
...
```

Inline caching example

`for (Drawable d: ds) d.draw();`



ds= `Circle` → `Circle` → `Circle` → `Rectangle` → `Rectangle`

loop body

```
...
dispatch
...
```

initial state (go through dispatch function)

patched

```
...
```
draw$_C$
```
...
```

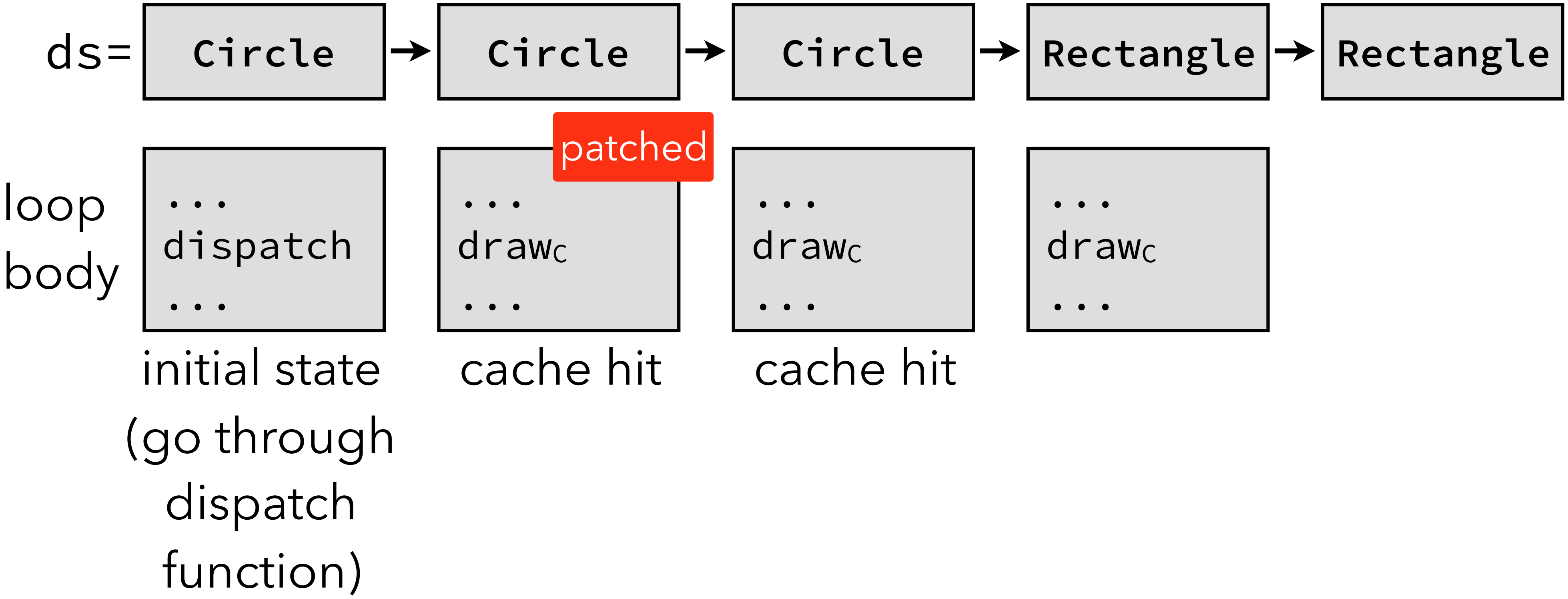
cache hit

```
...
```
draw$_C$
```
...
```

cache hit

Inline caching example

`for (Drawable d: ds) d.draw();`



ds=

`Circle` → `Circle` → `Circle` → `Rectangle` → `Rectangle`

loop body

`...` `dispatch` `...`

initial state (go through dispatch function)

patched

`...` draw$_C$ `...`

cache hit

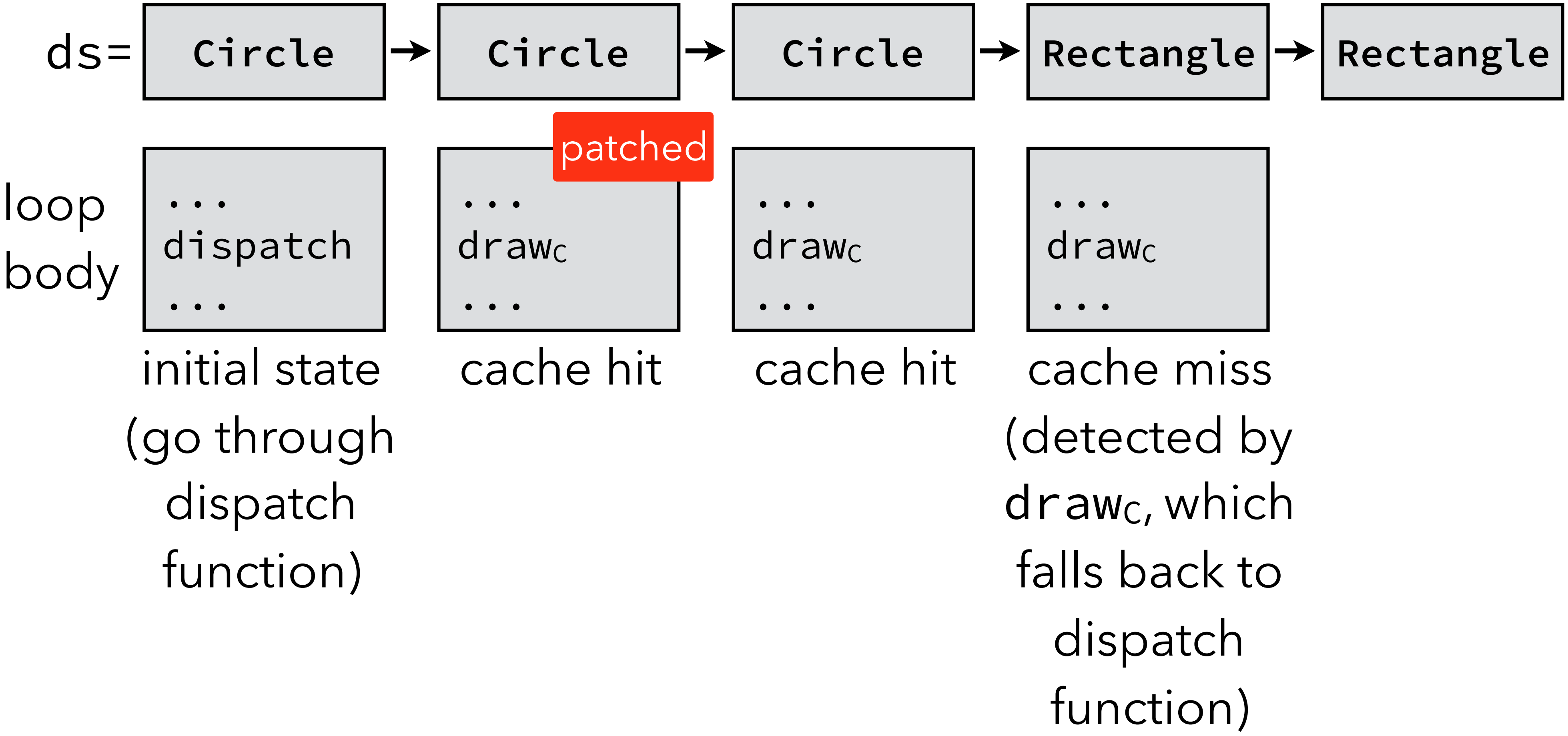
`...` draw$_C$ `...`

cache hit

`...` draw$_C$ `...`

Inline caching example

```
for (Drawable d: ds) d.draw();
```



ds= `Circle` → `Circle` → `Circle` → `Rectangle` → `Rectangle`

loop body

`...` `dispatch` `...`

initial state (go through dispatch function)

patched

`...` $\mathtt{draw}_C$ `...`

cache hit

`...` $\mathtt{draw}_C$ `...`

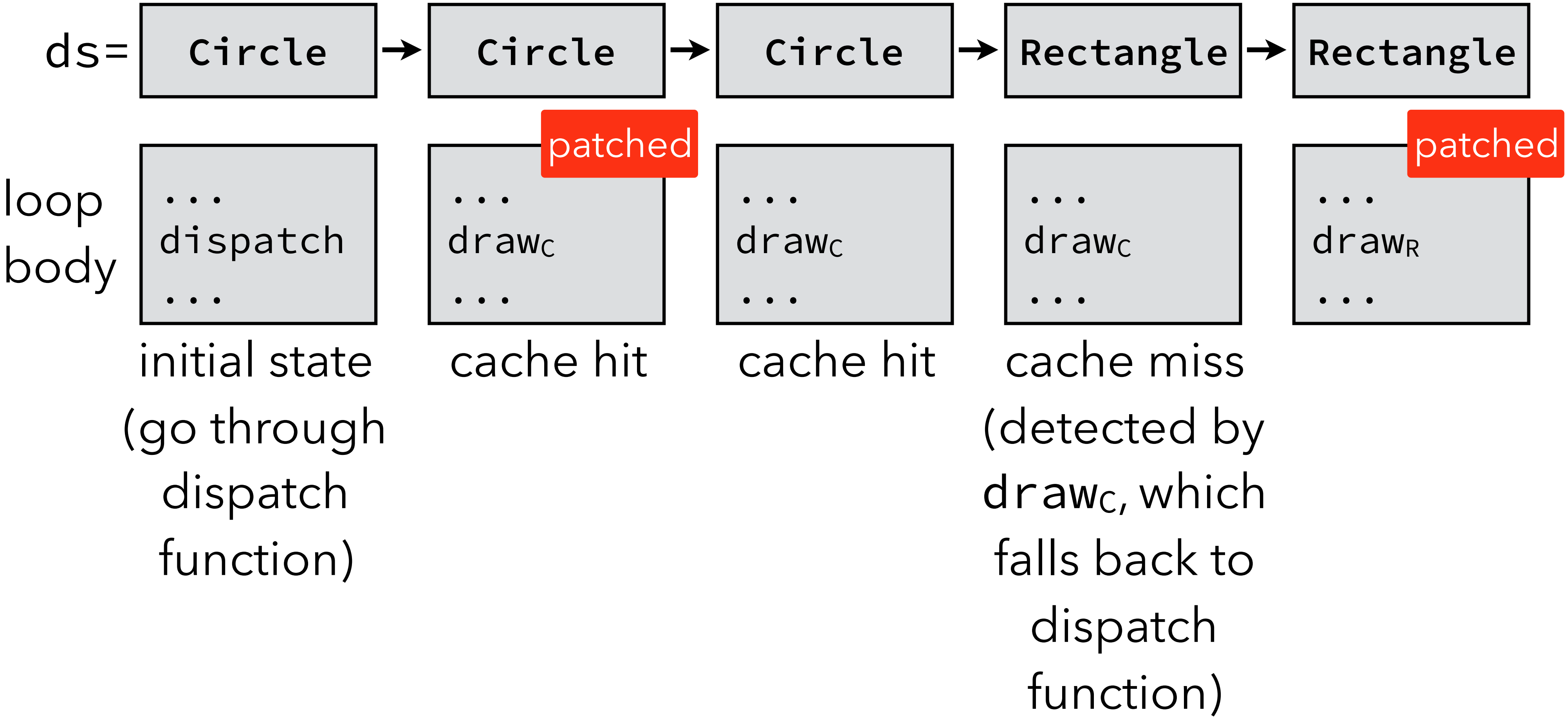
cache hit

`...` $\mathtt{draw}_C$ `...`

cache miss (detected by $\mathtt{draw}_C$, which falls back to dispatch function)

Inline caching example

`for (Drawable d: ds) d.draw();`



ds= `Circle` → `Circle` → `Circle` → `Rectangle` → `Rectangle`

loop body

`...` `dispatch` `...`
initial state (go through dispatch function)

patched
`...` `draw`$_C$ `...`
cache hit

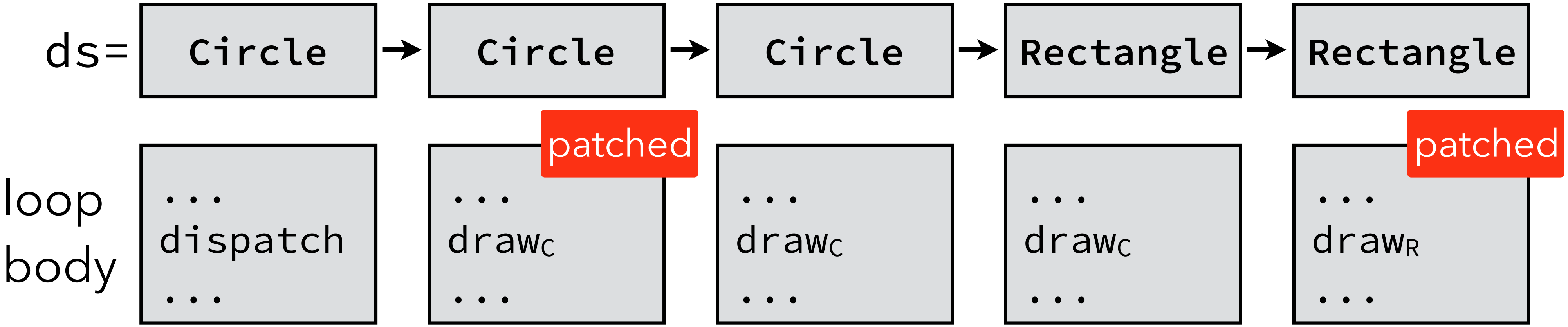
`...` `draw`$_C$ `...`
cache hit

`...` `draw`$_C$ `...`
cache miss (detected by `draw`$_C$, which falls back to dispatch function)

patched
`...` `draw`$_R$ `...`

Inline caching example

```
for (Drawable d: ds) d.draw();
```



ds= `Circle` → `Circle` → `Circle` → `Rectangle` → `Rectangle`

loop body

	patched			patched

`...`
`dispatch`
`...`
	`...`
	$\mathtt{draw}_C$
	`...`
		`...`
		$\mathtt{draw}_C$
		`...`
			`...`
			$\mathtt{draw}_C$
			`...`
				`...`
				$\mathtt{draw}_R$
				`...`

initial state (go through dispatch function)	cache hit	cache hit	cache miss (detected by $\mathtt{draw}_C$, which falls back to dispatch function)	cache hit

Inline caching pros & cons

Inline caching greatly speeds up method calls by avoiding expensive dispatches in most cases.

However, it can also slow down method calls that are really polymorphic! For example, if the list `ds` in our previous example contained an alternating sequence of circles and rectangles.

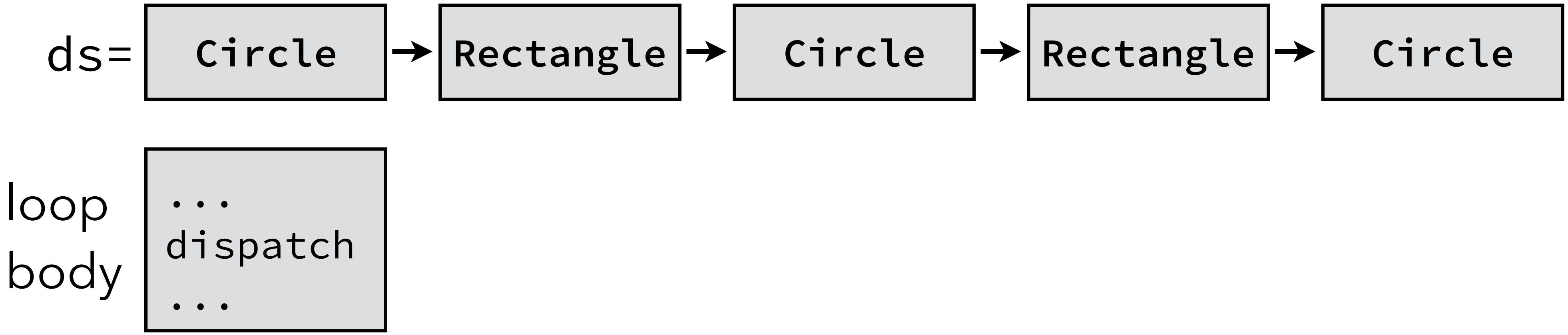
Polymorphic inline caching addresses this issue.

Polymorphic inline caching

Inline caching replaces the call to the dispatch function by a call to the latest method that was dispatched to. **Polymorphic inline caching** (**PIC**) replaces it instead by a call to a specialized dispatch routine, generated on the fly. That routine handles only a subset of the possible receiver types – namely those that were encountered previously at that call site.

PIC example

```
for (Drawable d: ds) d.draw();
```



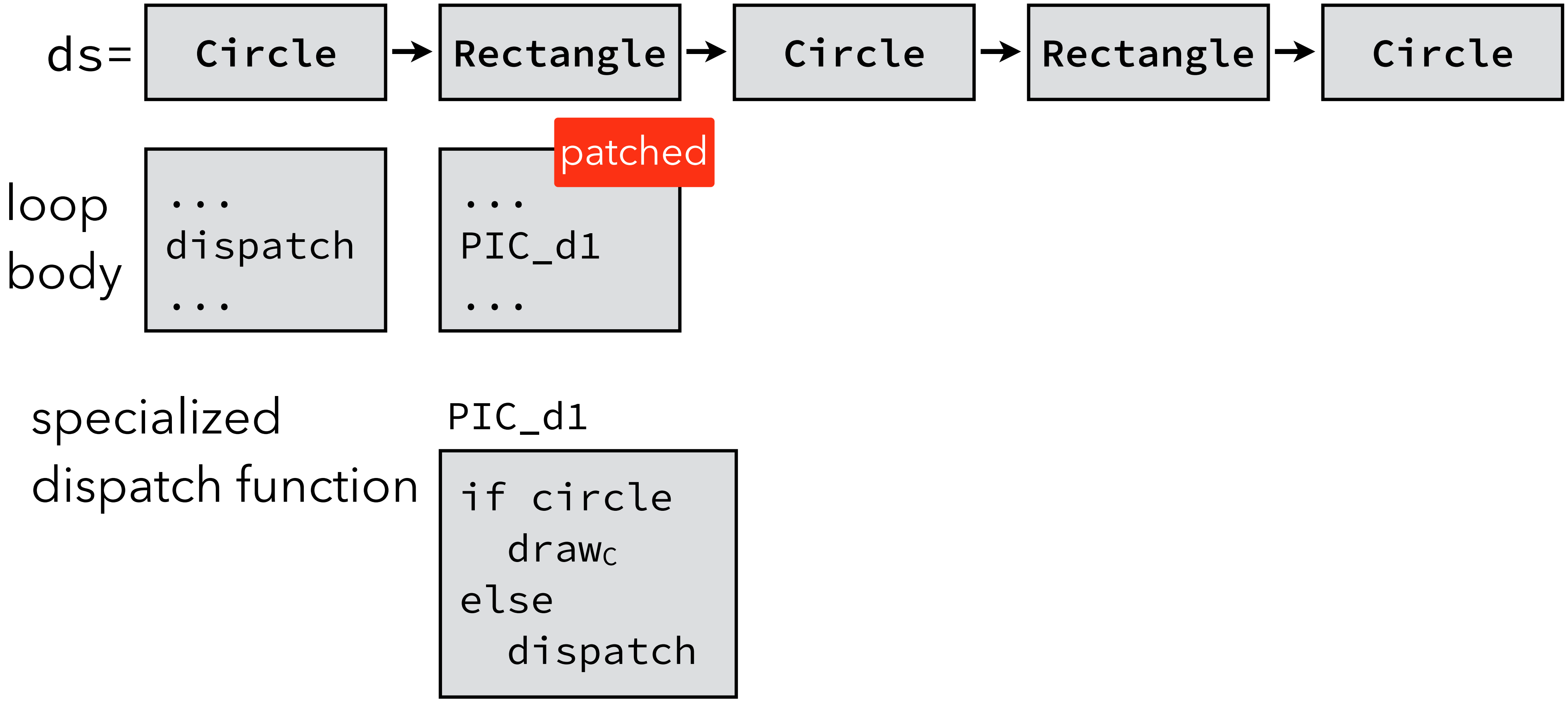
ds= `Circle` → `Rectangle` → `Circle` → `Rectangle` → `Circle`

loop body

```
...
dispatch
...
```

PIC example

```
for (Drawable d: ds) d.draw();
```



ds= `Circle` → `Rectangle` → `Circle` → `Rectangle` → `Circle`

loop body

```
...
dispatch
...
```

patched

```
...
PIC_d1
...
```

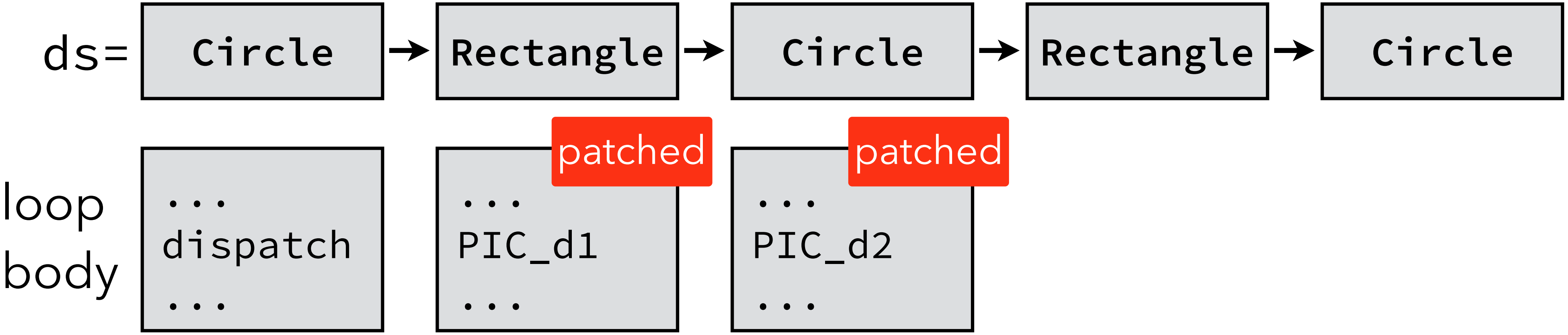
specialized dispatch function

`PIC_d1`

```
if circle
   draw_C
else
   dispatch
```

PIC example

```
for (Drawable d: ds) d.draw();
```



ds= **Circle** → **Rectangle** → **Circle** → **Rectangle** → **Circle**

loop body

```
...
dispatch
...
```

patched

```
...
PIC_d1
...
```

patched

```
...
PIC_d2
...
```

specialized dispatch function

PIC_d1

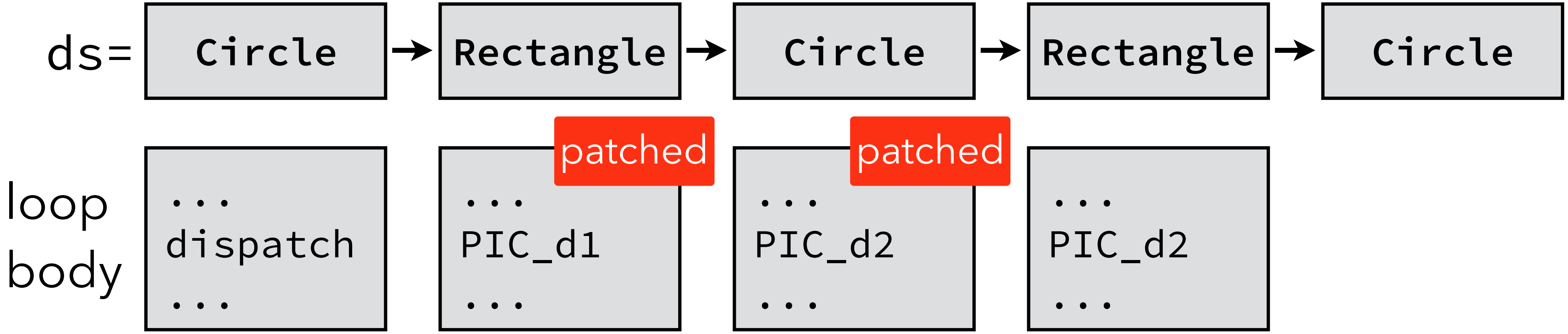
```
if circle
  $\text{draw}_C$
else
  dispatch
```

PIC_d2

```
if rectangle
  $\text{draw}_R$
else if circle
  $\text{draw}_C$
else
  dispatch
```

PIC example

```
for (Drawable d: ds) d.draw();
```



ds= Circle → Rectangle → Circle → Rectangle → Circle

loop body

```
...
dispatch
...
```

patched

```
...
PIC_d1
...
```

patched

```
...
PIC_d2
...
```

```
...
PIC_d2
...
```

specialized dispatch function

PIC_d1

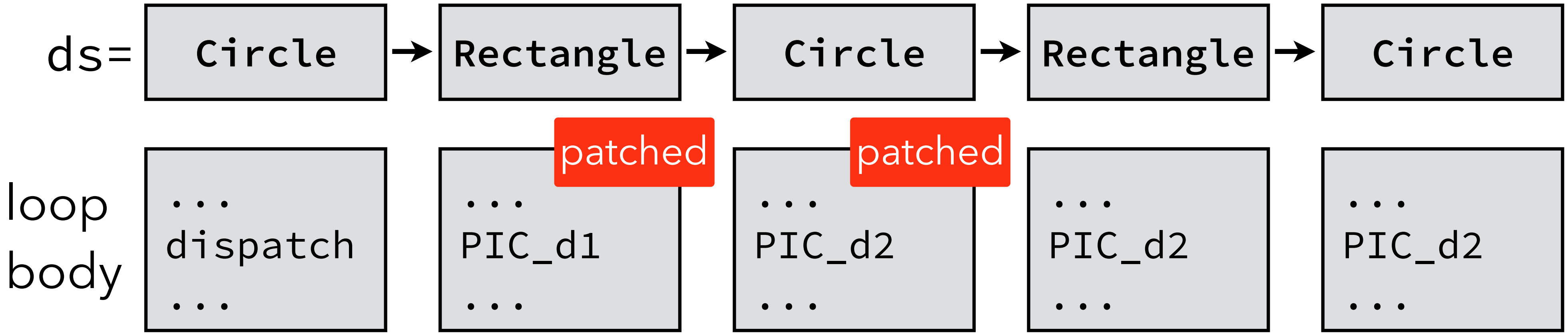
```
if circle
  $draw_C$
else
  dispatch
```

PIC_d2

```
if rectangle
  $draw_R$
else if circle
  $draw_C$
else
  dispatch
```

PIC example

`for (Drawable d: ds) d.draw();`



ds= **Circle** → **Rectangle** → **Circle** → **Rectangle** → **Circle**

loop body

```
...
dispatch
...
```

```
...
PIC_d1
...
```
patched

```
...
PIC_d2
...
```
patched

```
...
PIC_d2
...
```

```
...
PIC_d2
...
```

specialized dispatch function

PIC_d1

```
if circle
    $\text{draw}_C$
else
    dispatch
```

PIC_d2

```
if rectangle
    $\text{draw}_R$
else if circle
    $\text{draw}_C$
else
    dispatch
```

PIC receiver type test

The specialized dispatch function must be able to test very quickly whether an object is of a given type. This can be accomplished for example by storing a tag in every object, representing its class.

It must be noted that this test does not check whether the type of the receiver is a *subtype* of a given type, but whether it is *equal* to a given type.

This implies that several entries in the specialized dispatch function can actually call the same method, if that method is inherited.

PIC optimizations

An interesting feature of PIC is that the methods called from the specialized dispatch function can be inlined into it, provided they are small enough. For example, `PIC_d2` could become:

```
if rectangle
  // inlined code of $draw_R$
else if circle
  // inlined code of $draw_C$
else
  dispatch
```

Moreover, the tests can be rearranged so that the ones corresponding to the most frequent receiver types appear first.

Method dispatch summary

The method dispatch problem is solved by virtual method tables in a "single subtyping" context.

In presence of multiple subtyping, some compressed form of a global dispatching matrix is used. Compression techniques take advantage of the sparsity and redundancy of that matrix.

Inline caching and its polymorphic variant can dramatically reduce the cost of dispatching.

Exercise

As we have seen, inline caching is useful to optimize method dispatch in an object-oriented (OO) language. Could it also be useful in a functional (and not OO) language? Explain.

OO problem #3: membership test

Membership test

The **membership test problem** consists in checking whether an object belongs to a given type.
This problem must be solved very often. In Java, for example, this is required on every use of the `instanceof` operator, type cast or array store operation, and every time an exception is thrown – to identify the matching handler.

Membership test example

```
class A { }
class B extends A { }
boolean f(Object o) {
  return (o instanceof A);
}
```

is o an
instance of A or
any of its
subtypes?

Case 1:
single subtyping

Membership test

Like the other two problems we examined, the membership test is relatively easy to solve in a single subtyping setting. We will examine two techniques that work in that context:

1. relative numbering, and
2. Cohen’s encoding.

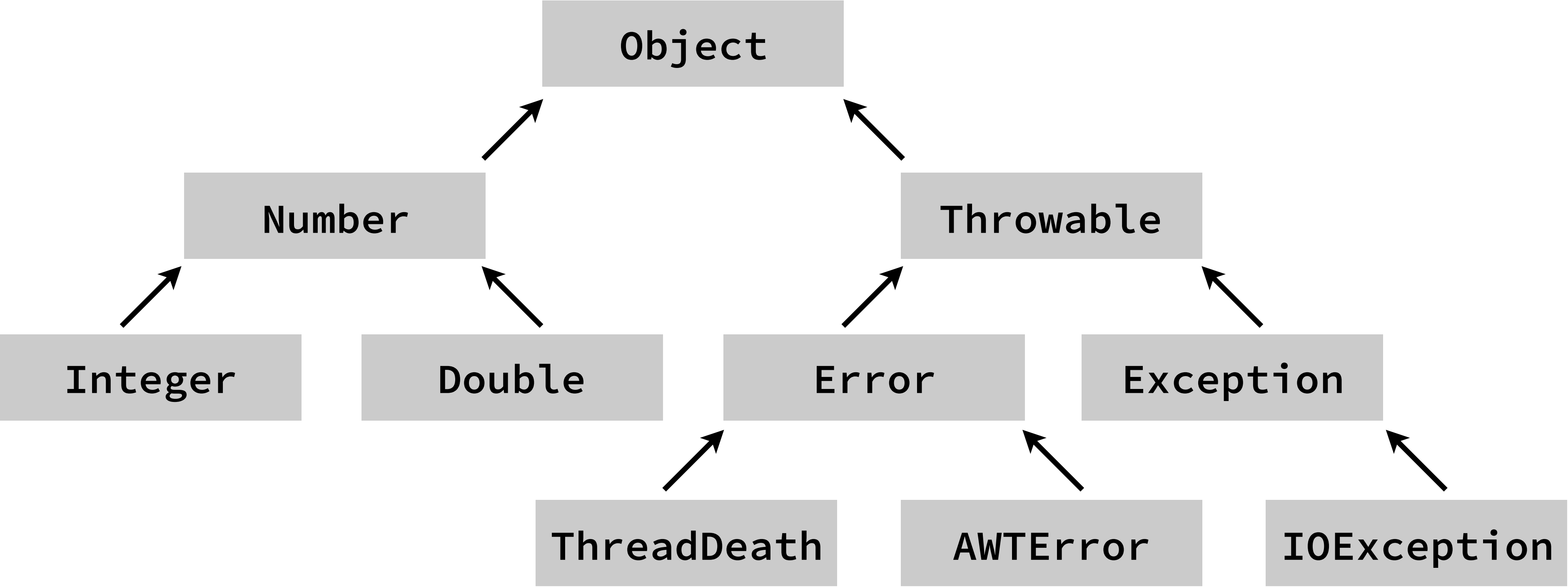
Relative numbering

The idea of **relative numbering** is to number the types in the hierarchy during a preorder traversal.

This numbering has the interesting characteristic that the numbers attributed to all descendants of a given type form a continuous interval.

Membership tests can therefore be performed very efficiently, by checking whether the number attributed to the type of the object being tested belongs to a given interval.

Relative numbering example



Object

Number

Throwable

Integer

Double

Error

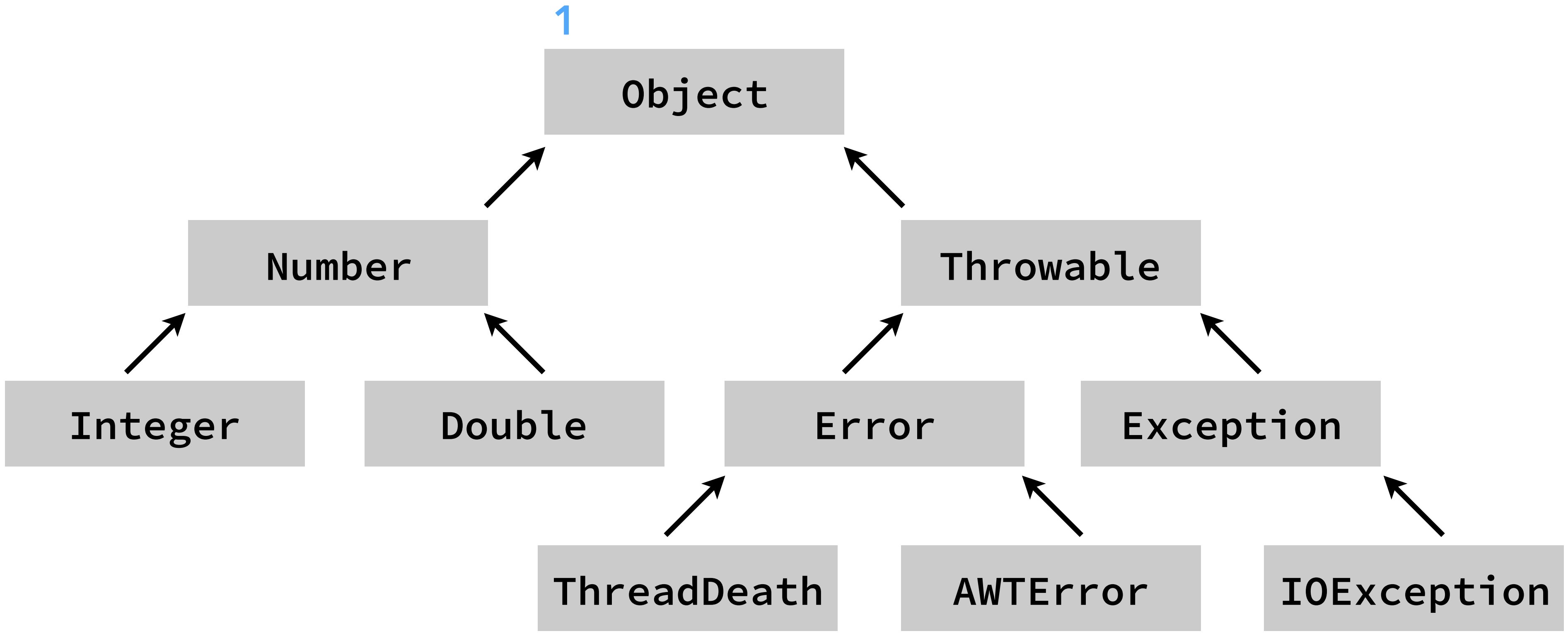
Exception

ThreadDeath

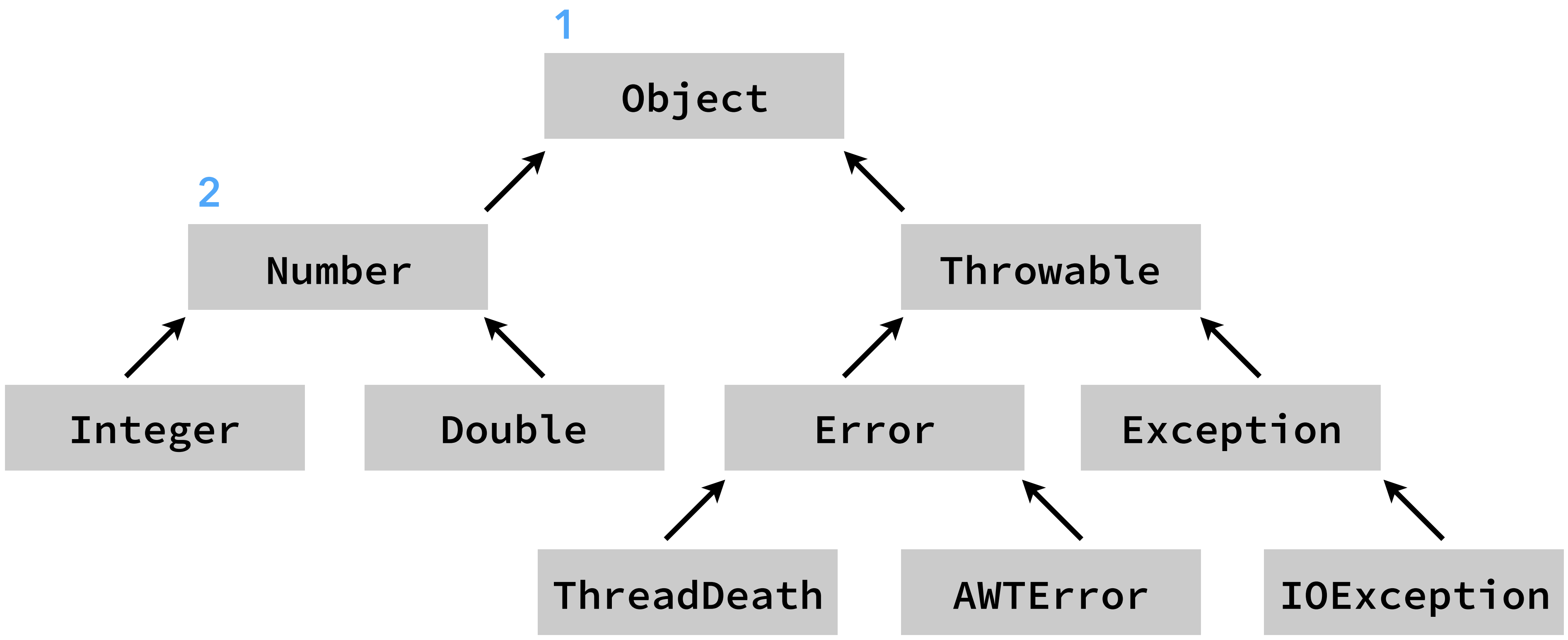
AWTError

IOException

Relative numbering example



Relative numbering example



1

Object

2

Number

Throwable

Integer

Double

Error

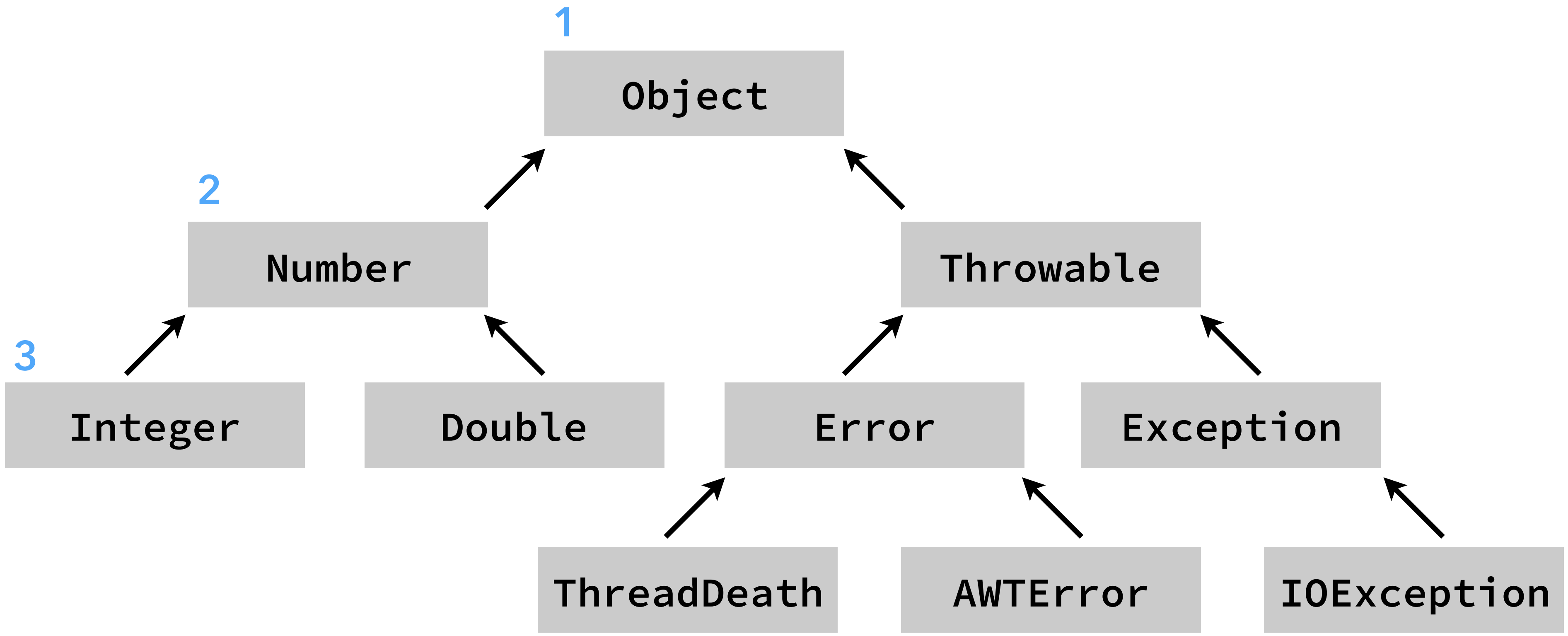
Exception

ThreadDeath

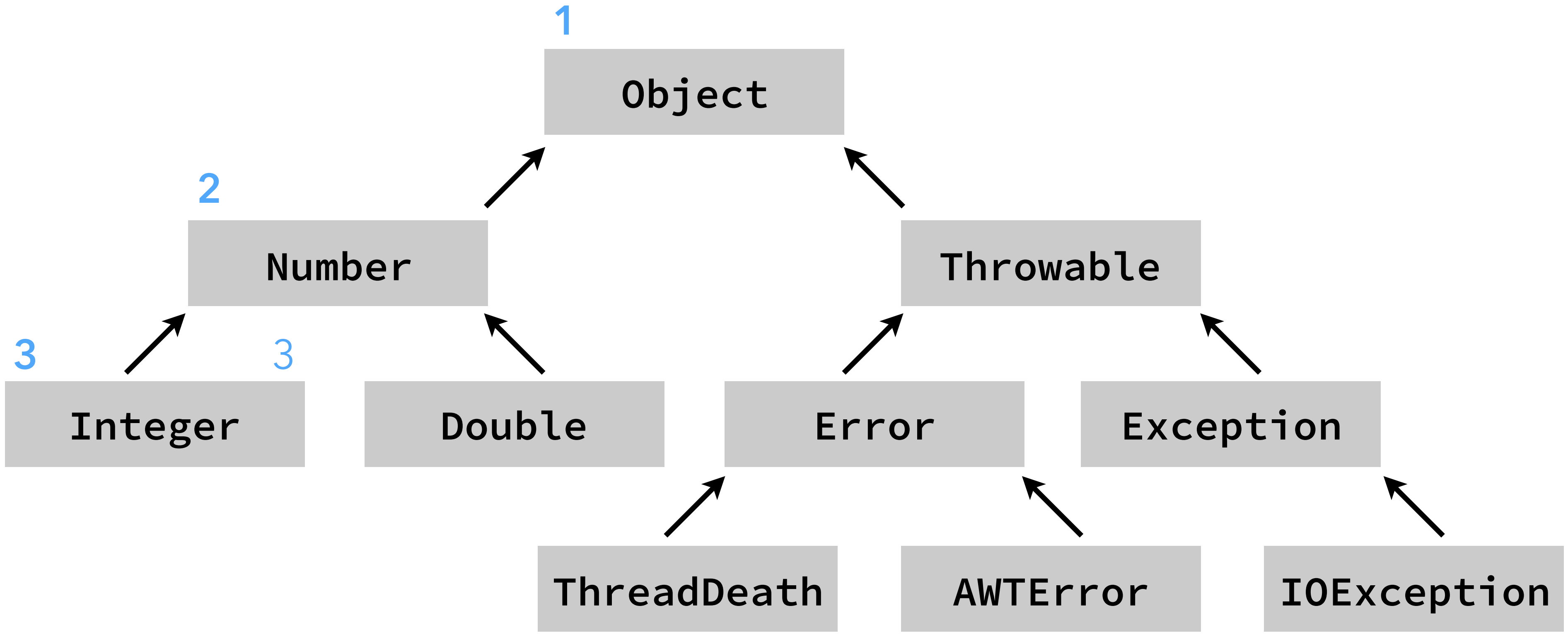
AWTError

IOException

Relative numbering example



Relative numbering example



1

Object

2

Number

Throwable

3

3

Integer

Double

Error

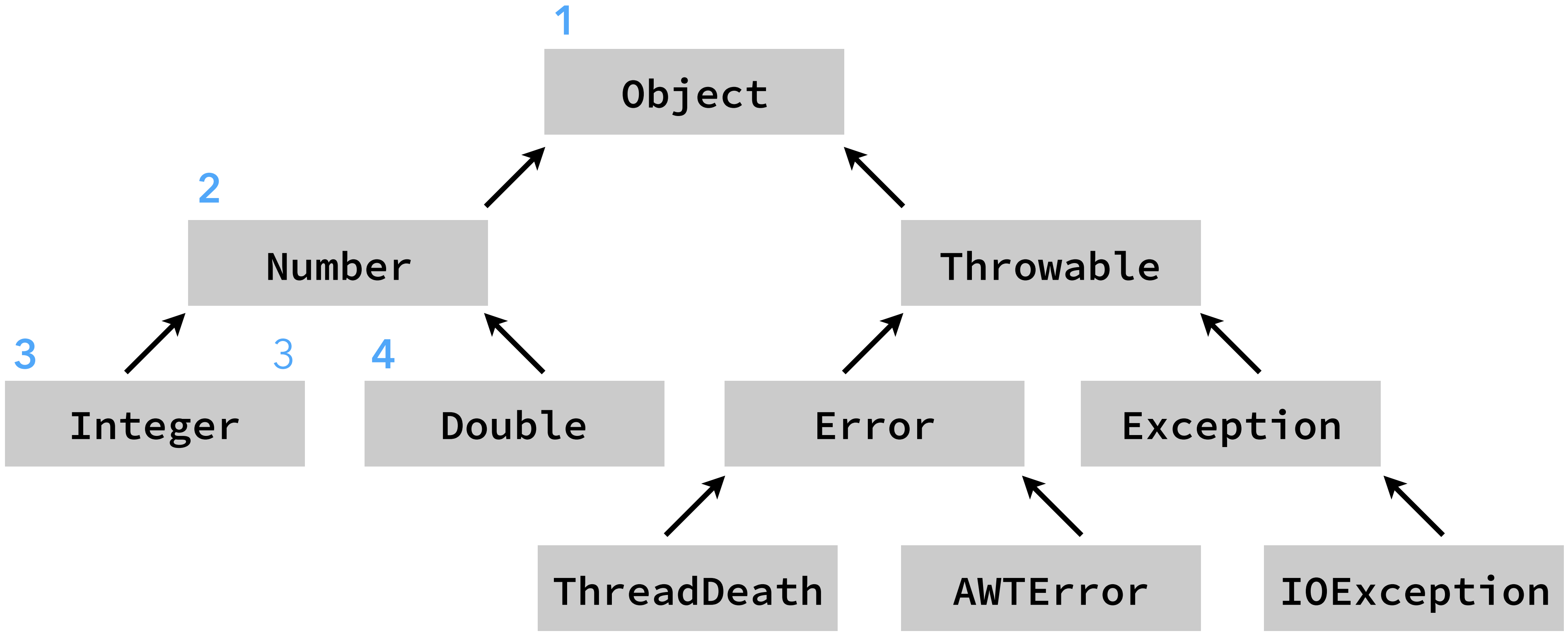
Exception

ThreadDeath

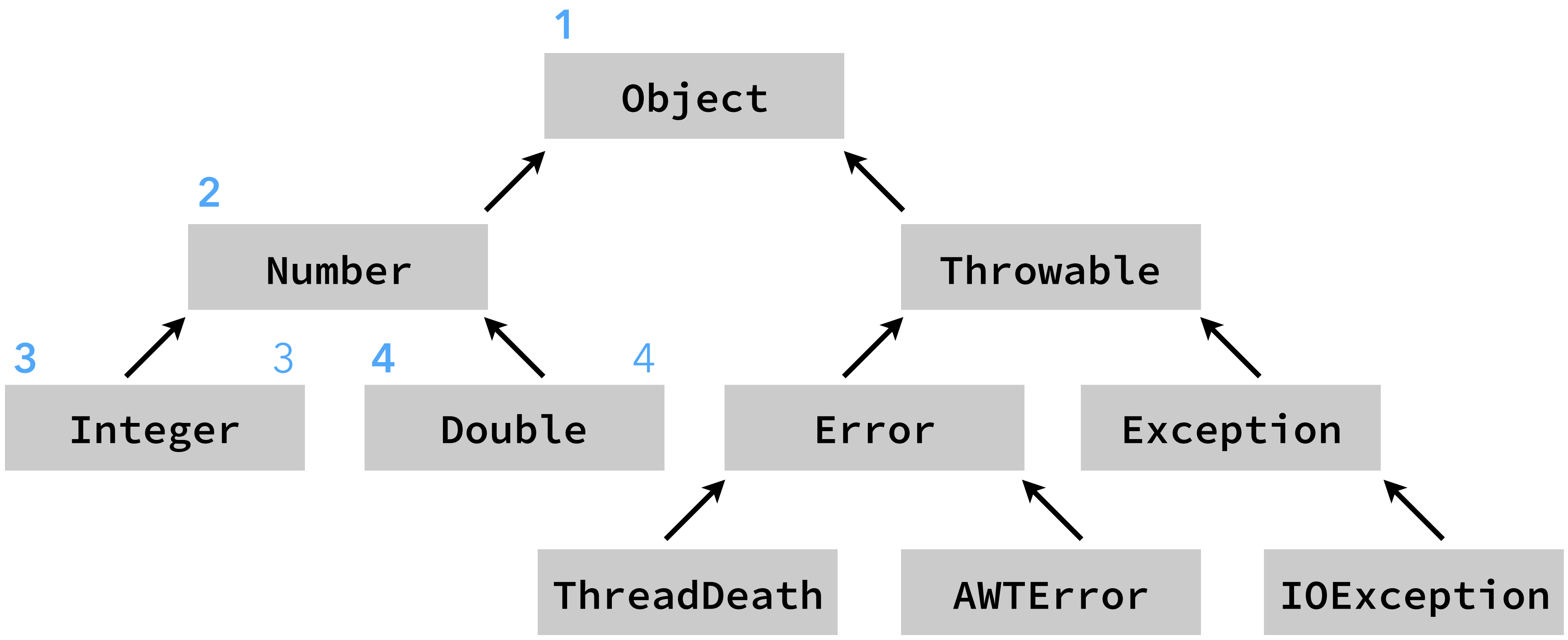
AWTError

IOException

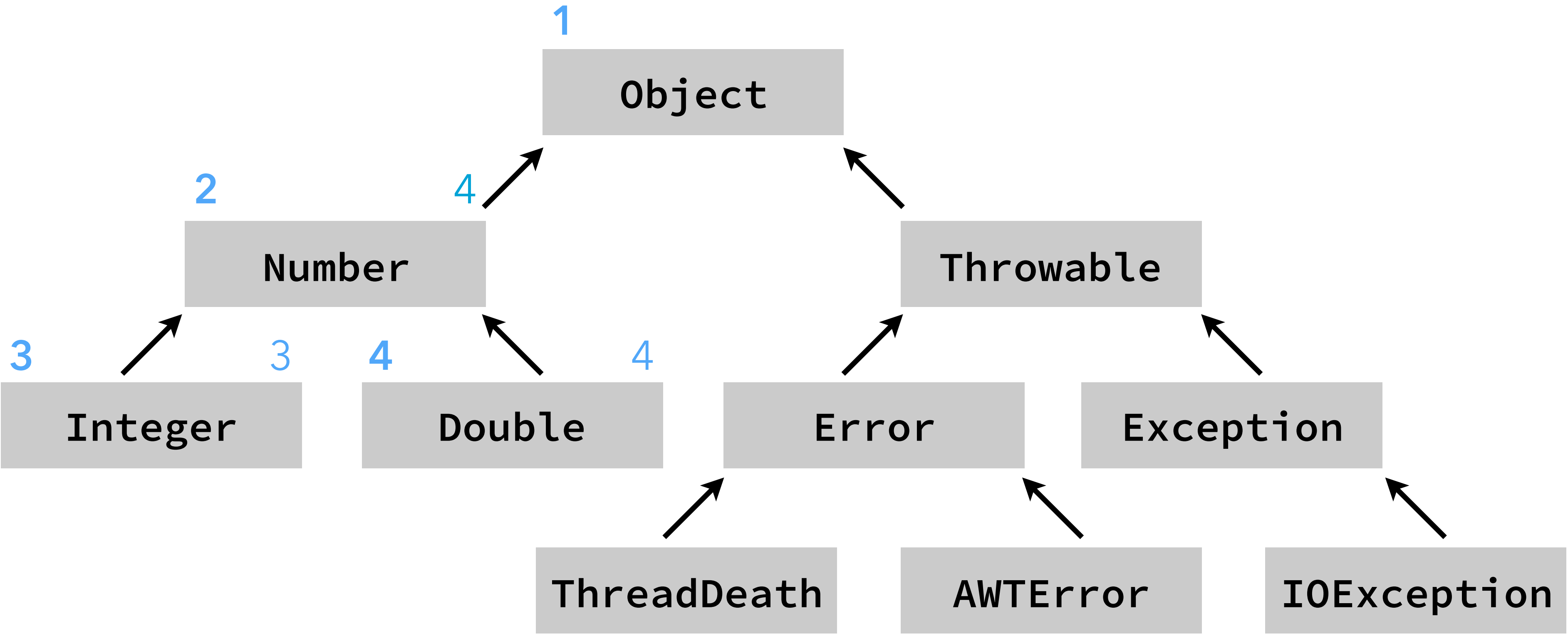
Relative numbering example



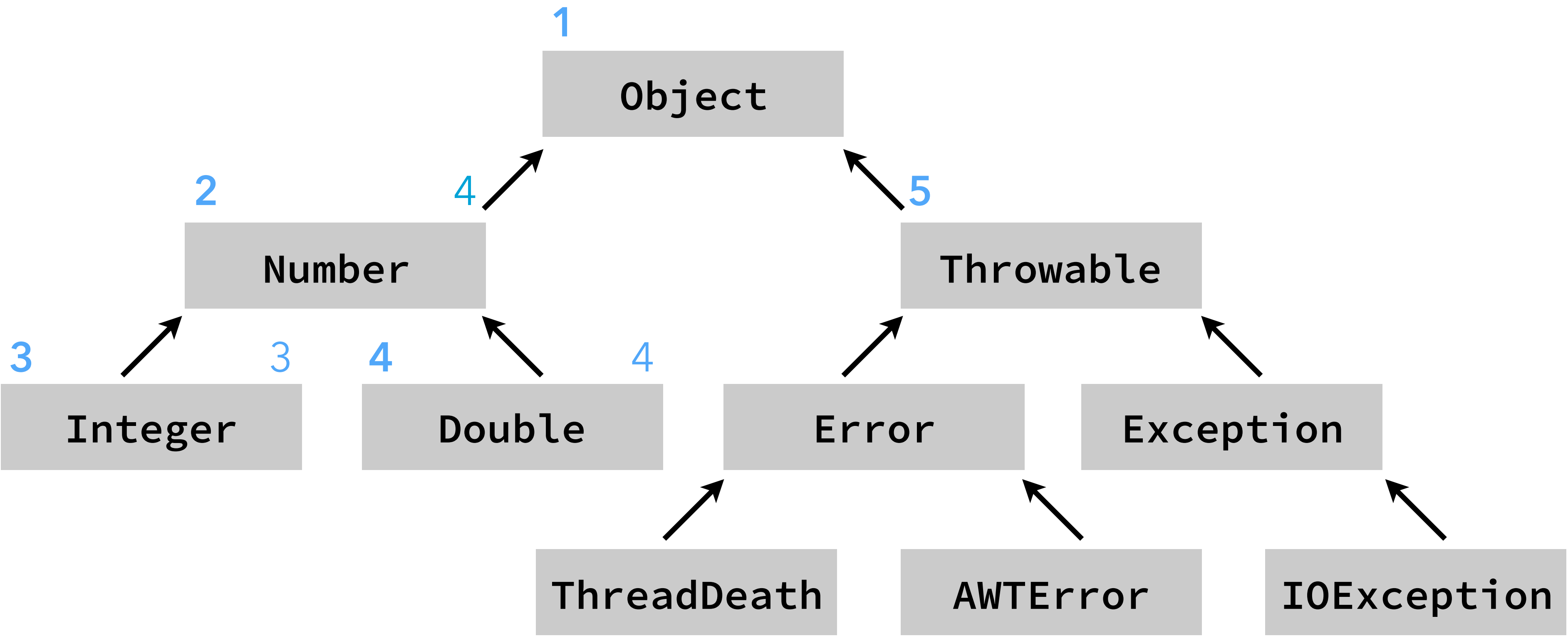
Relative numbering example



Relative numbering example



Relative numbering example



1

Object

2 4 5

Number

Throwable

3 3 4 4

Integer

Double

Error

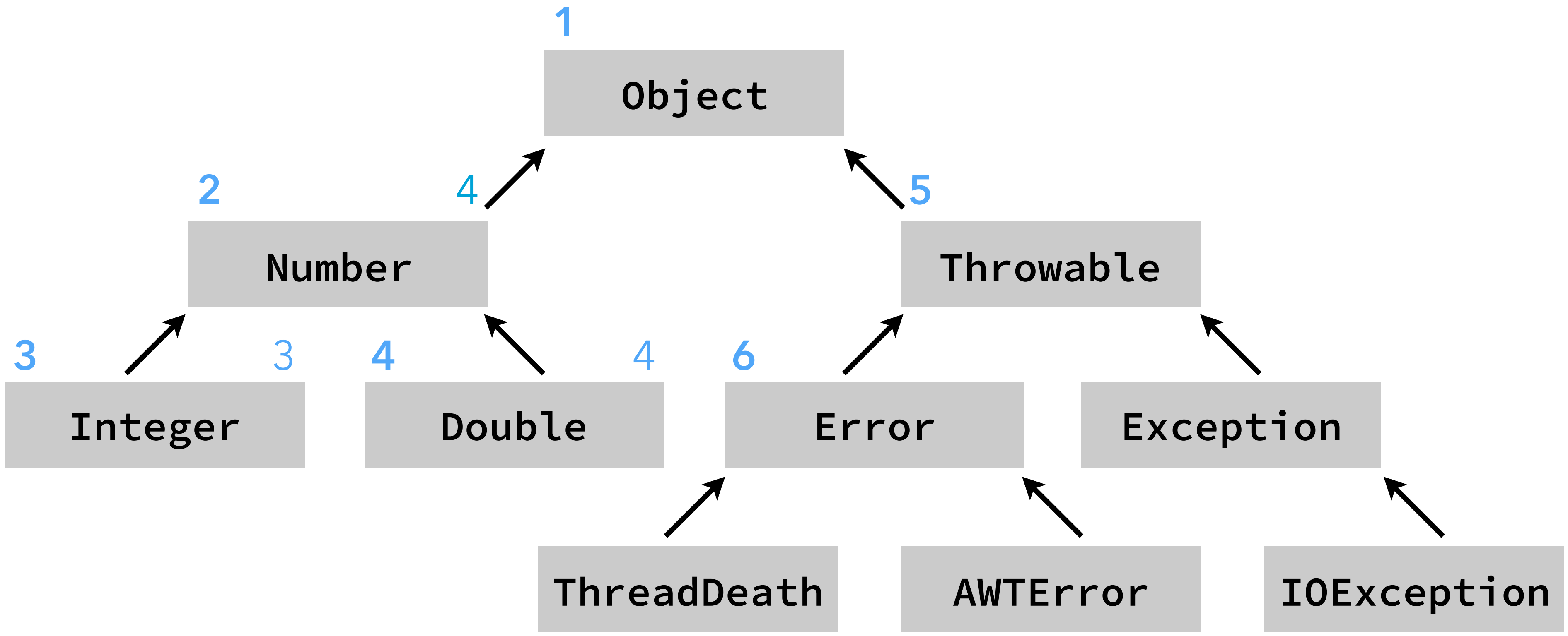
Exception

ThreadDeath

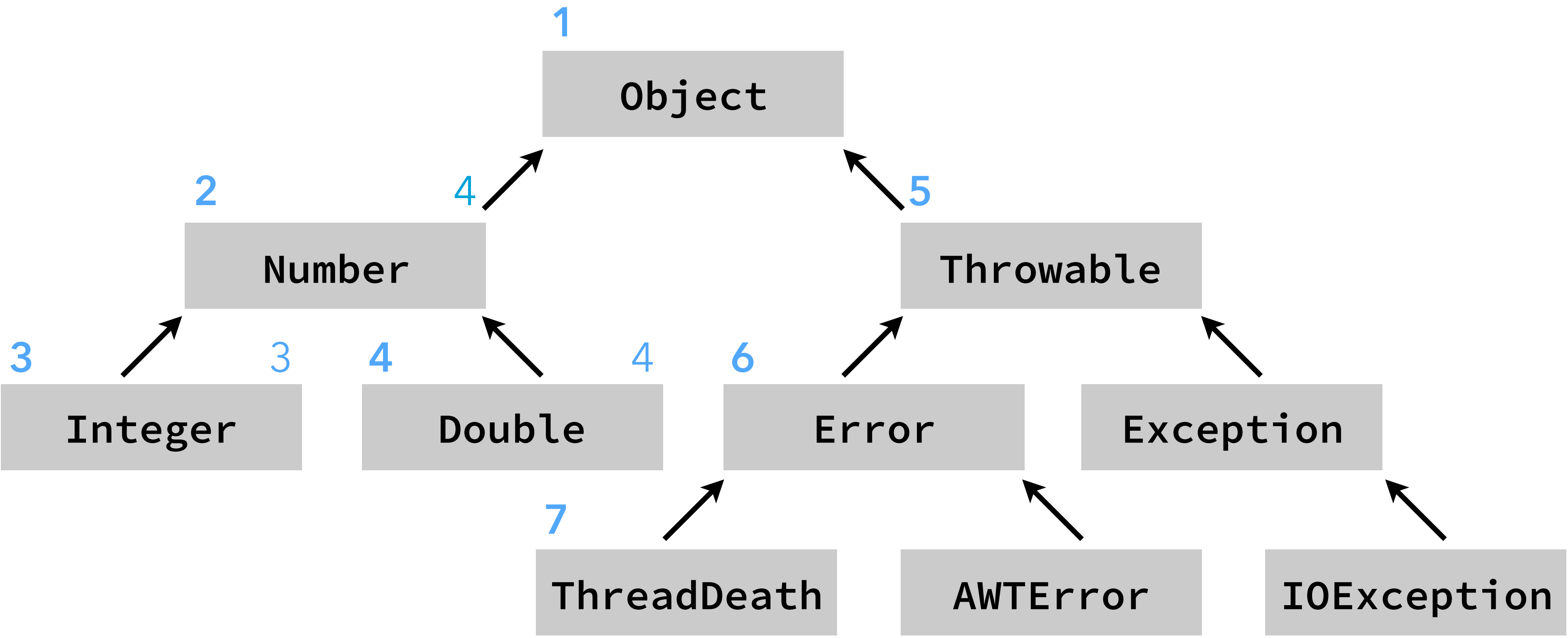
AWTError

IOException

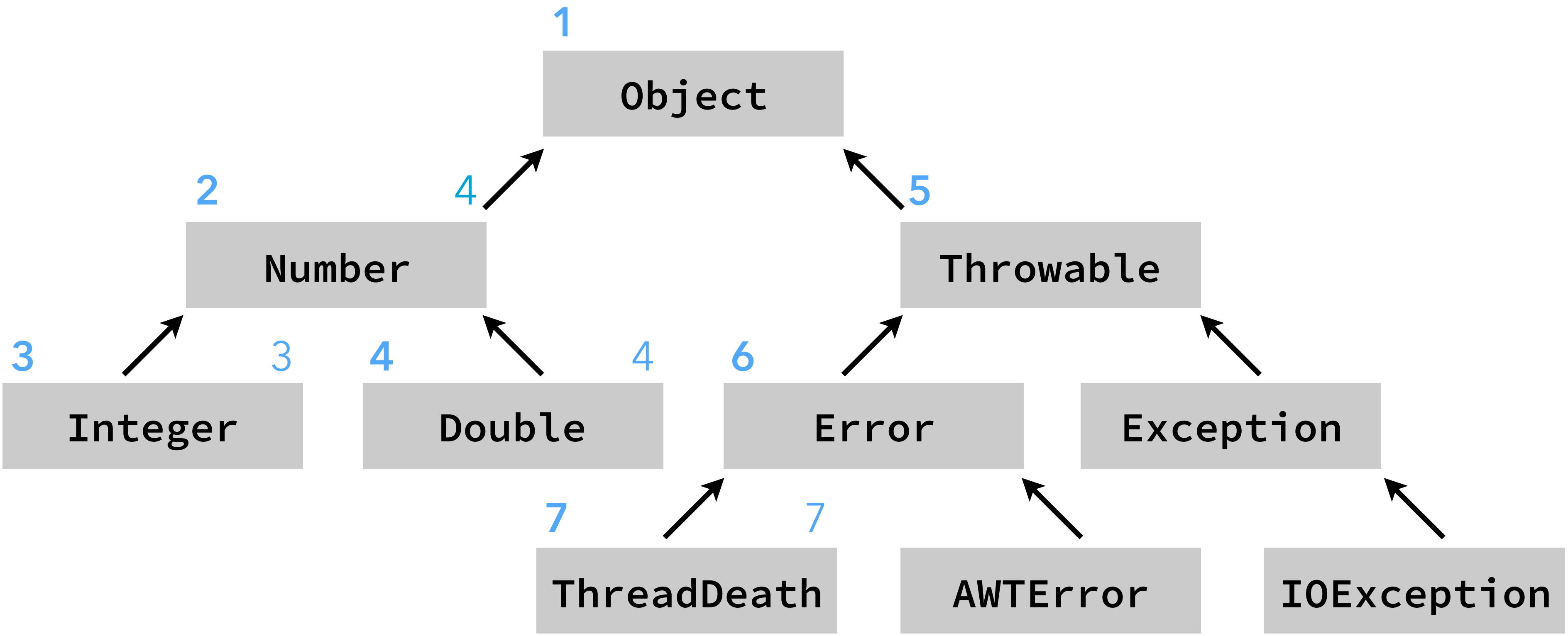
Relative numbering example



Relative numbering example



Relative numbering example



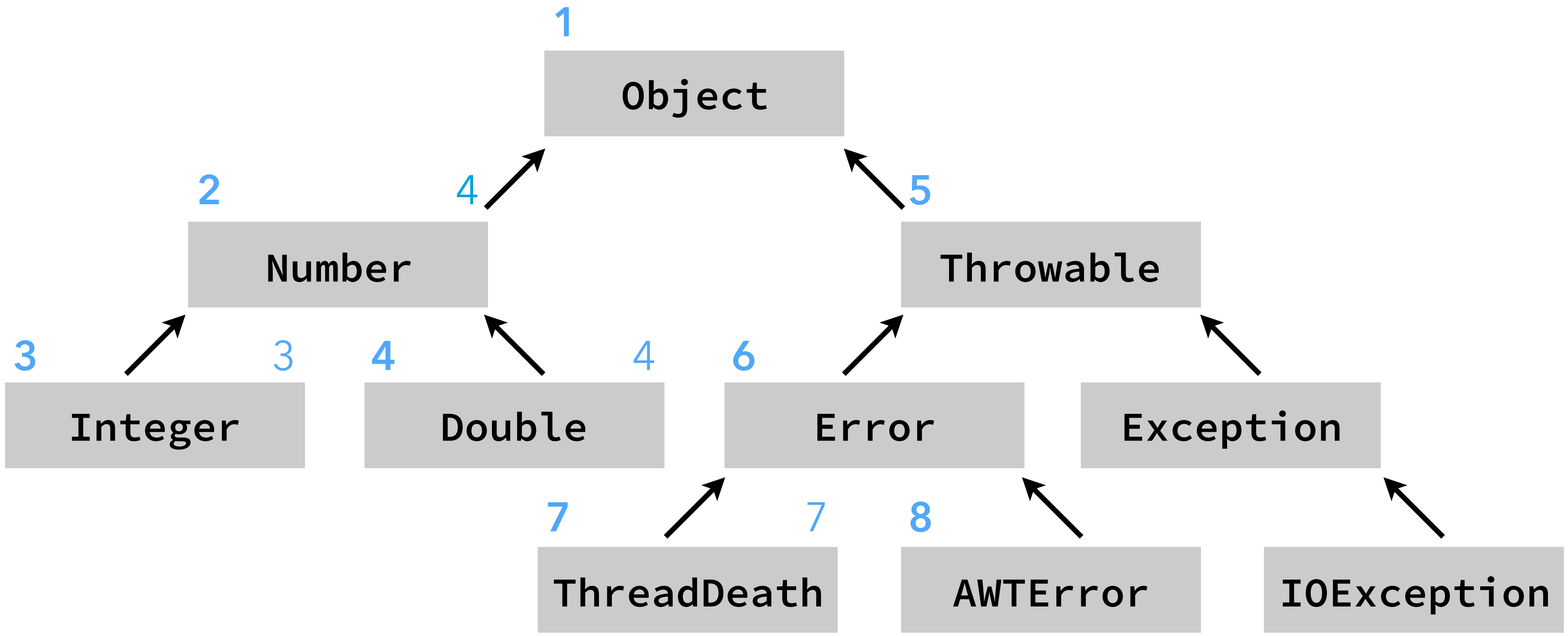
1
Object

2
4
5
Number
Throwable

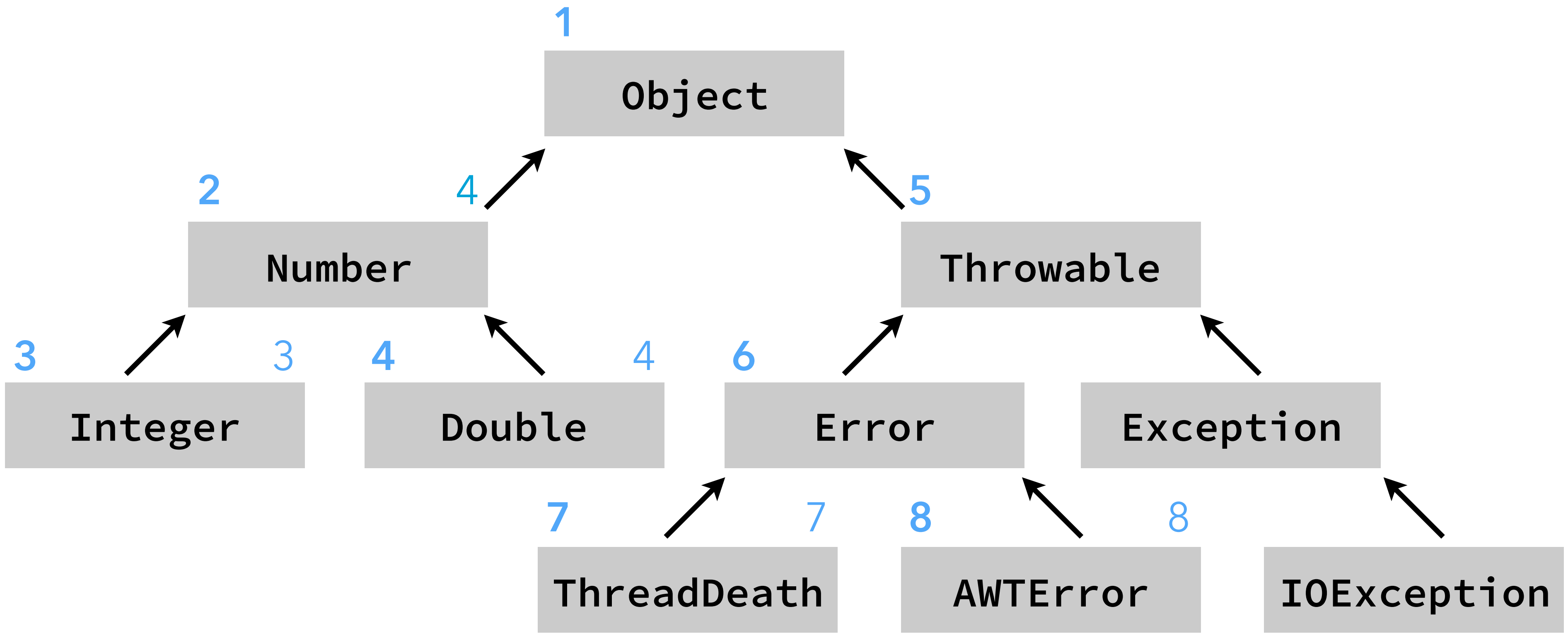
3
3
4
4
6
Integer
Double
Error
Exception

7
7
ThreadDeath
AWTError
IOException

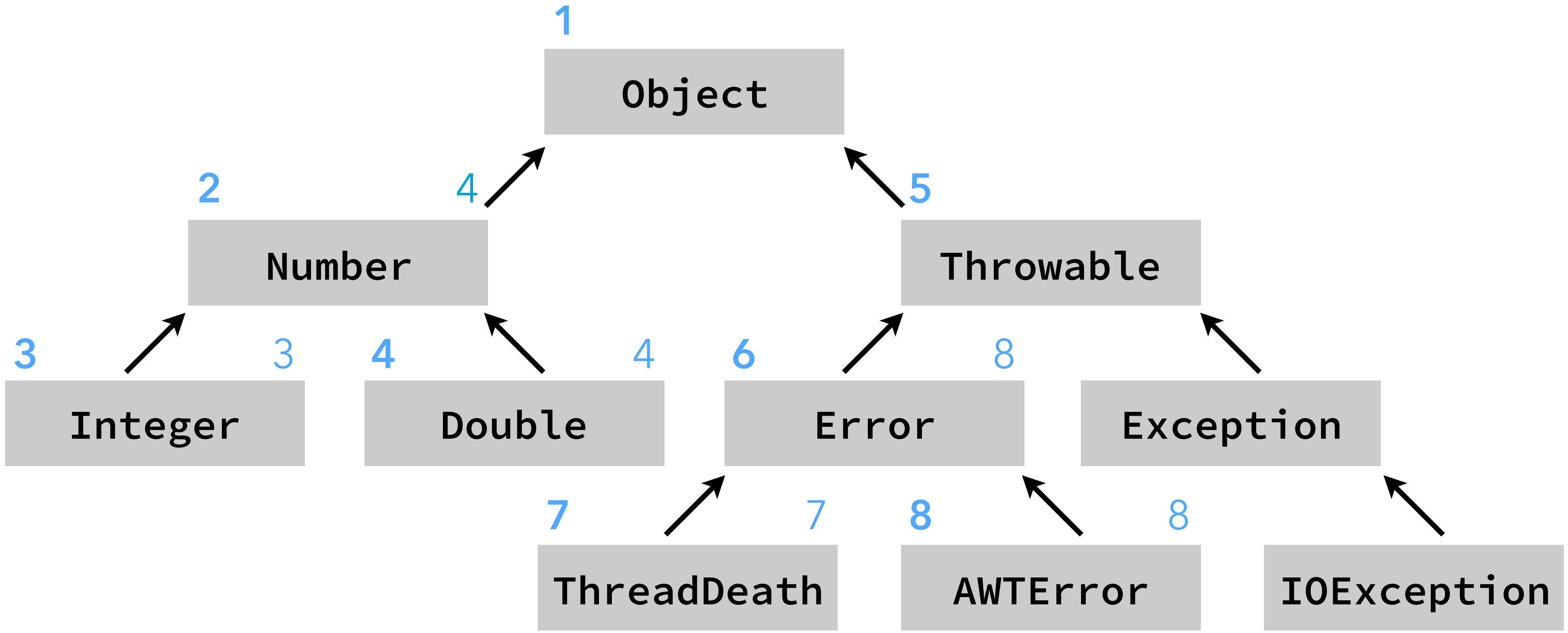
Relative numbering example



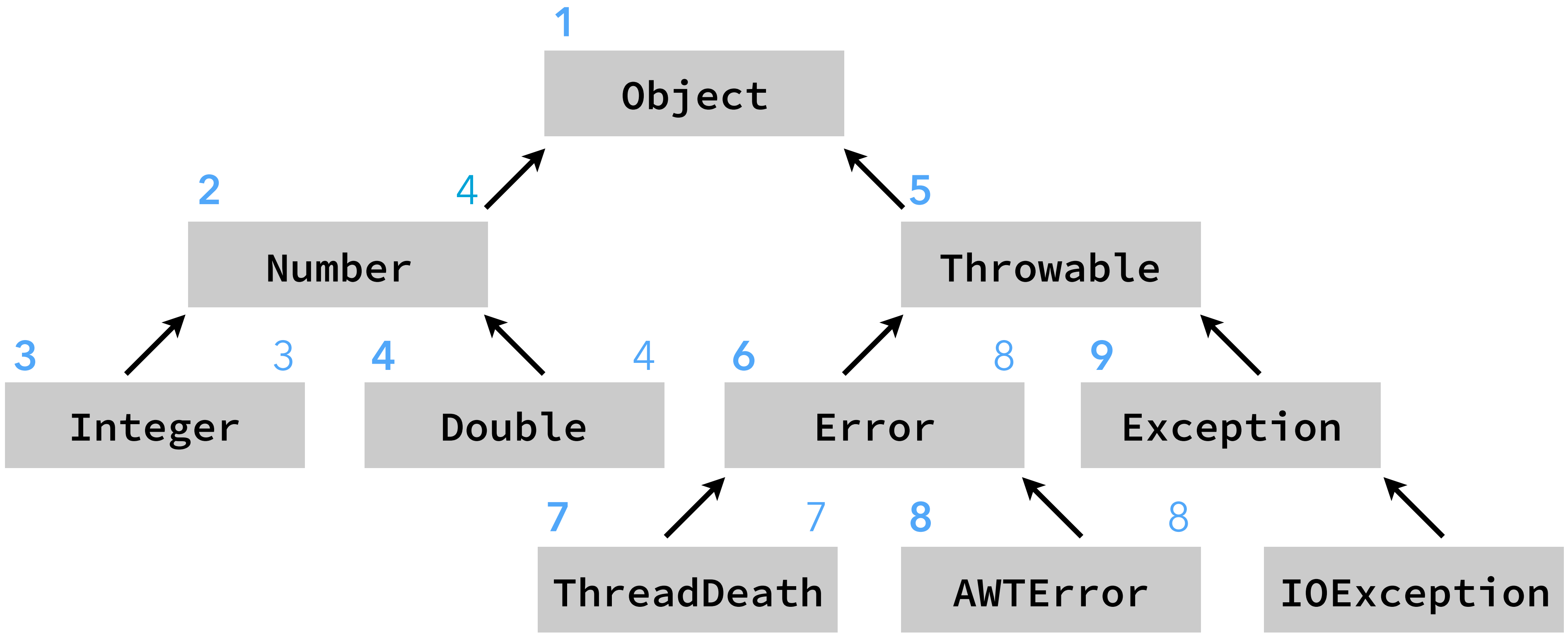
Relative numbering example



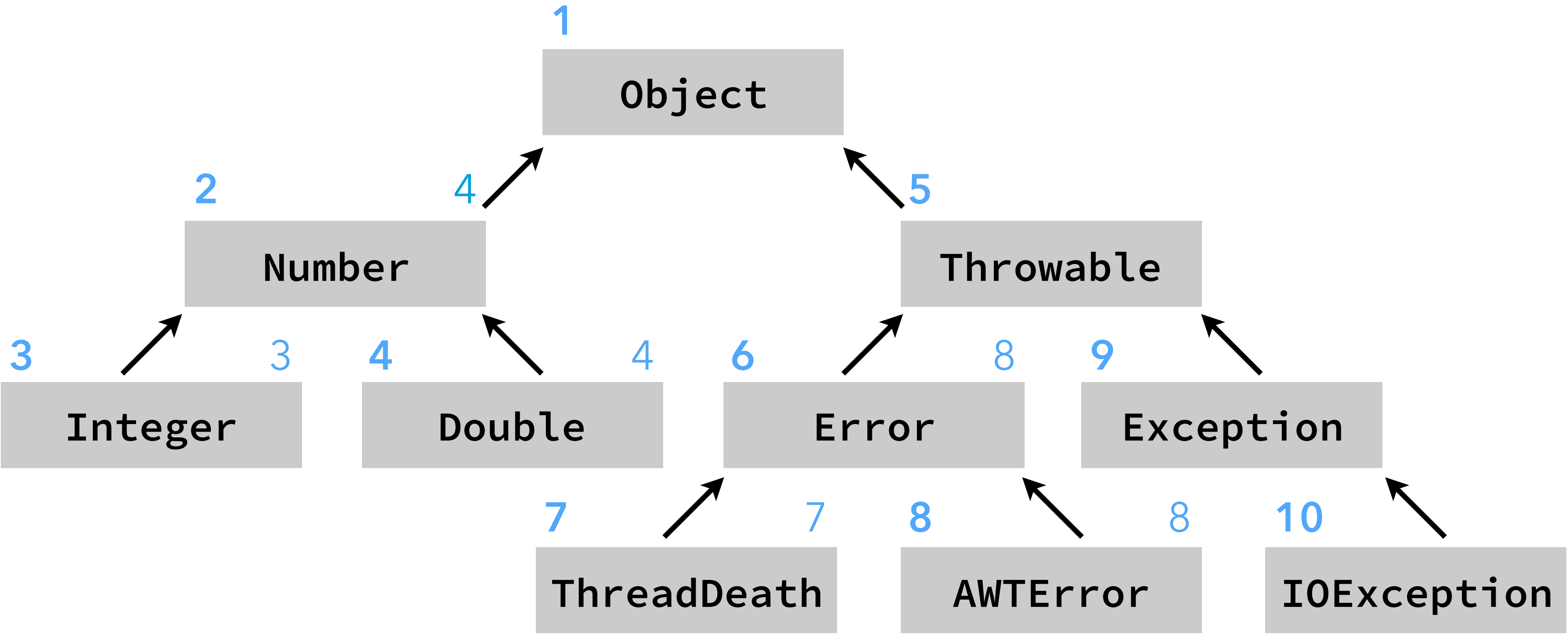
Relative numbering example



Relative numbering example



Relative numbering example



1

`Object`

2 4 5

`Number` `Throwable`

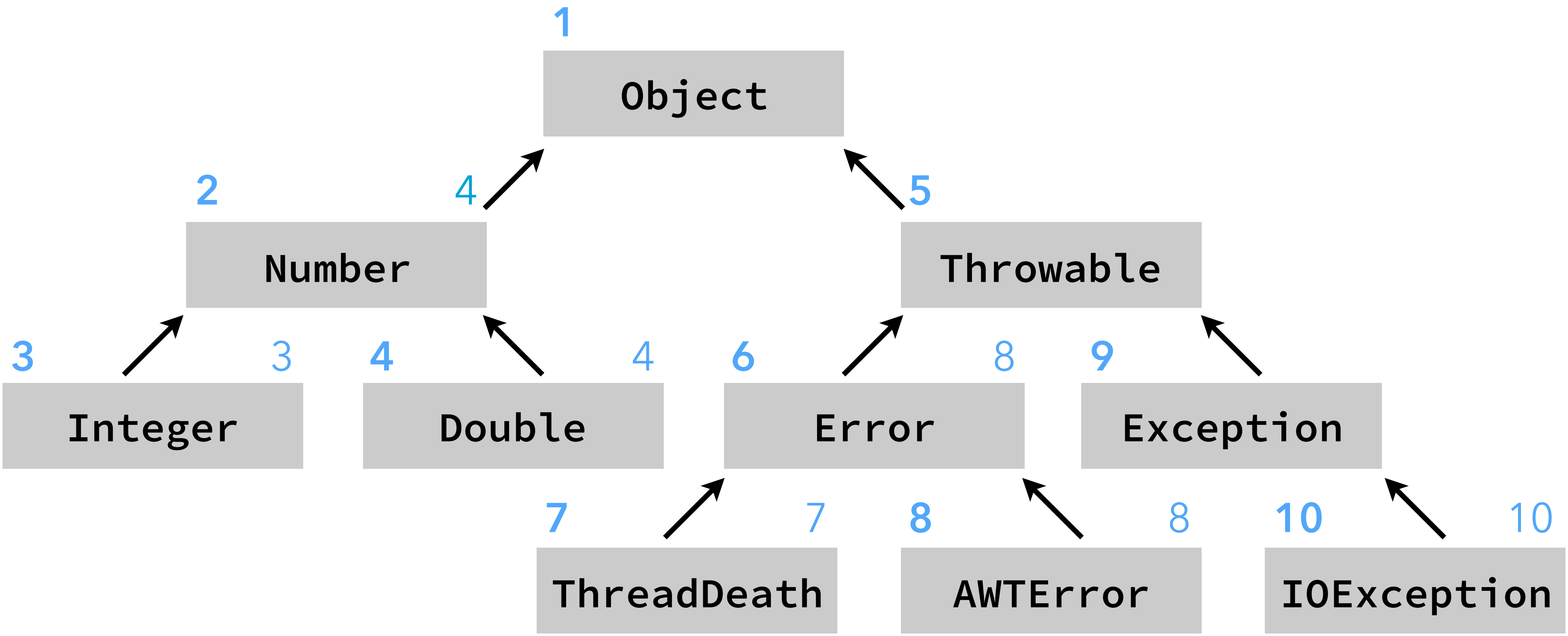
3 3 4 4 6 8 9

`Integer` `Double` `Error` `Exception`

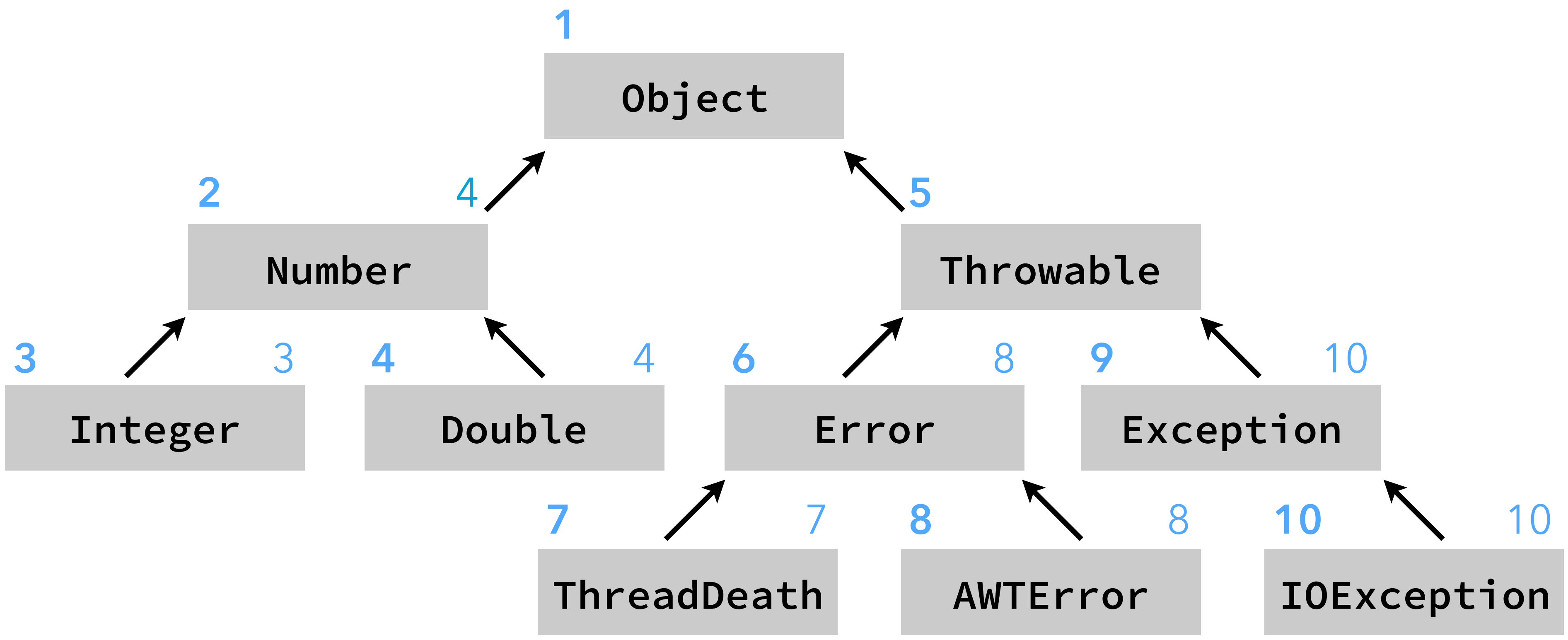
7 7 8 8 10

`ThreadDeath` `AWTError` `IOException`

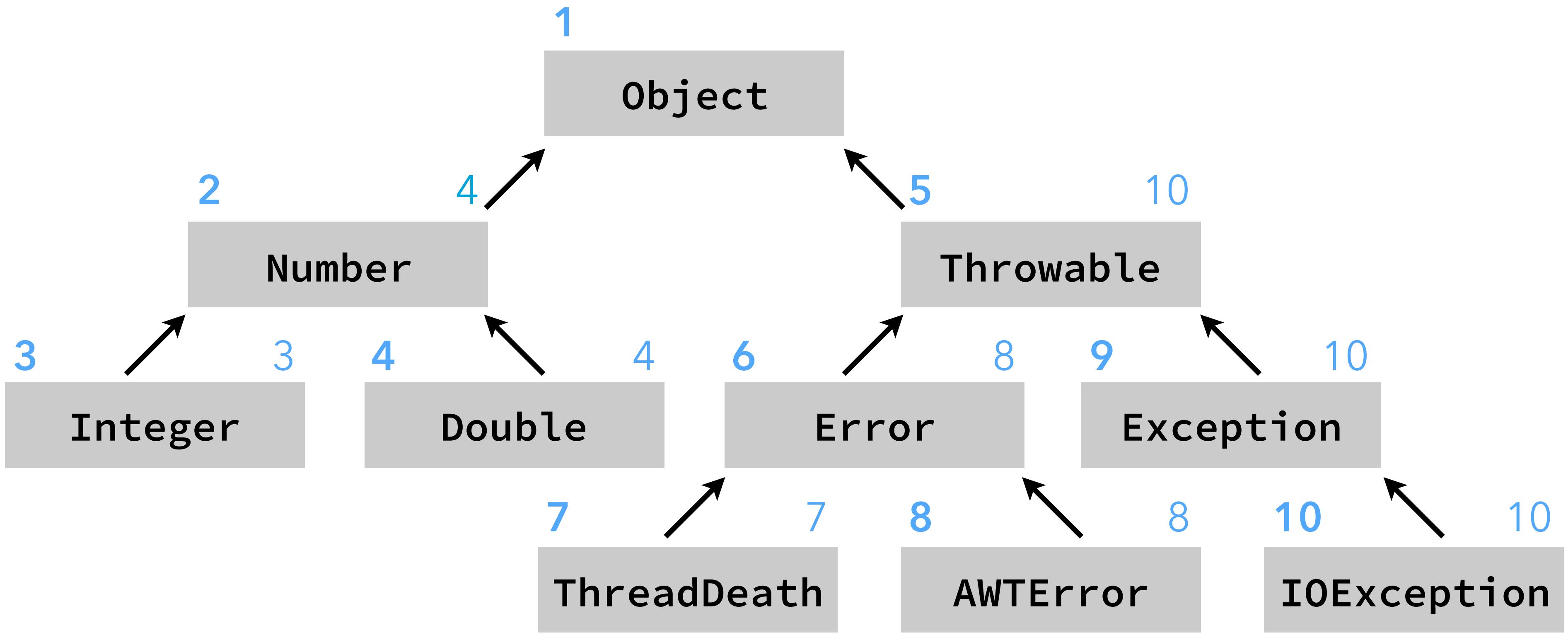
Relative numbering example



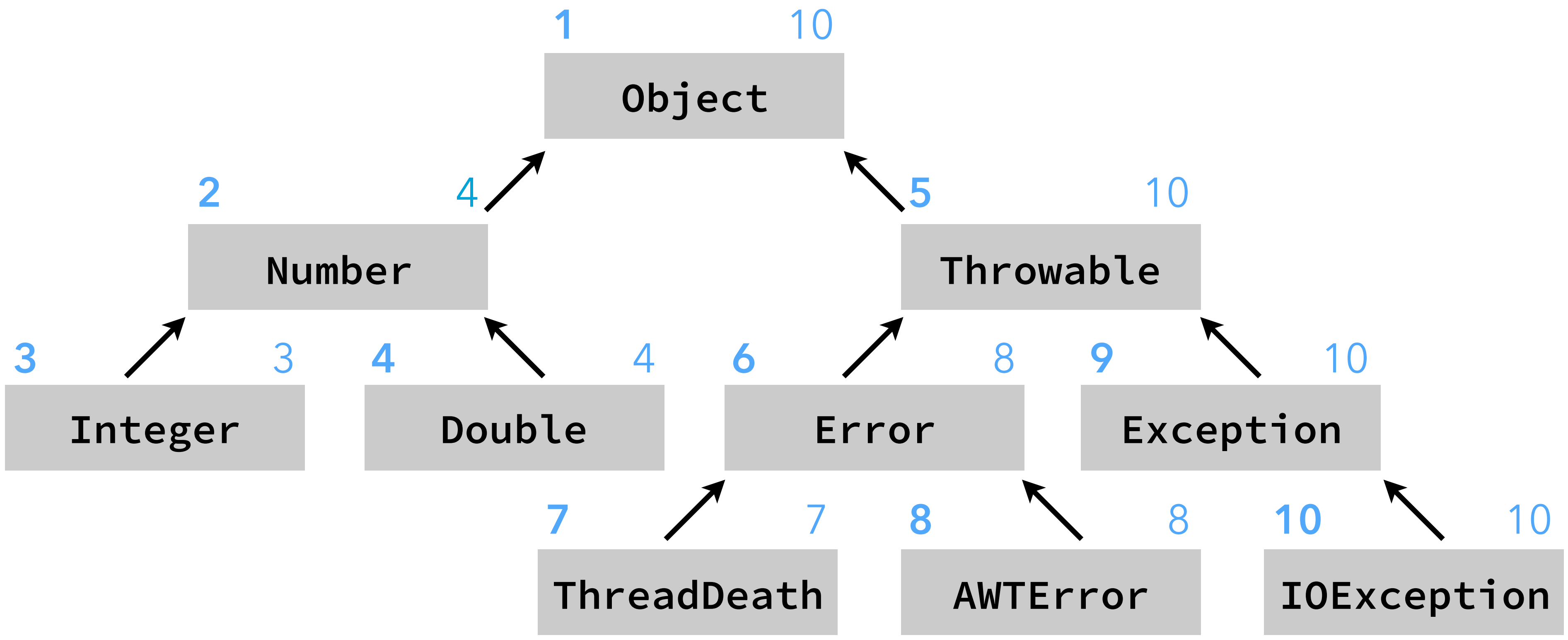
Relative numbering example



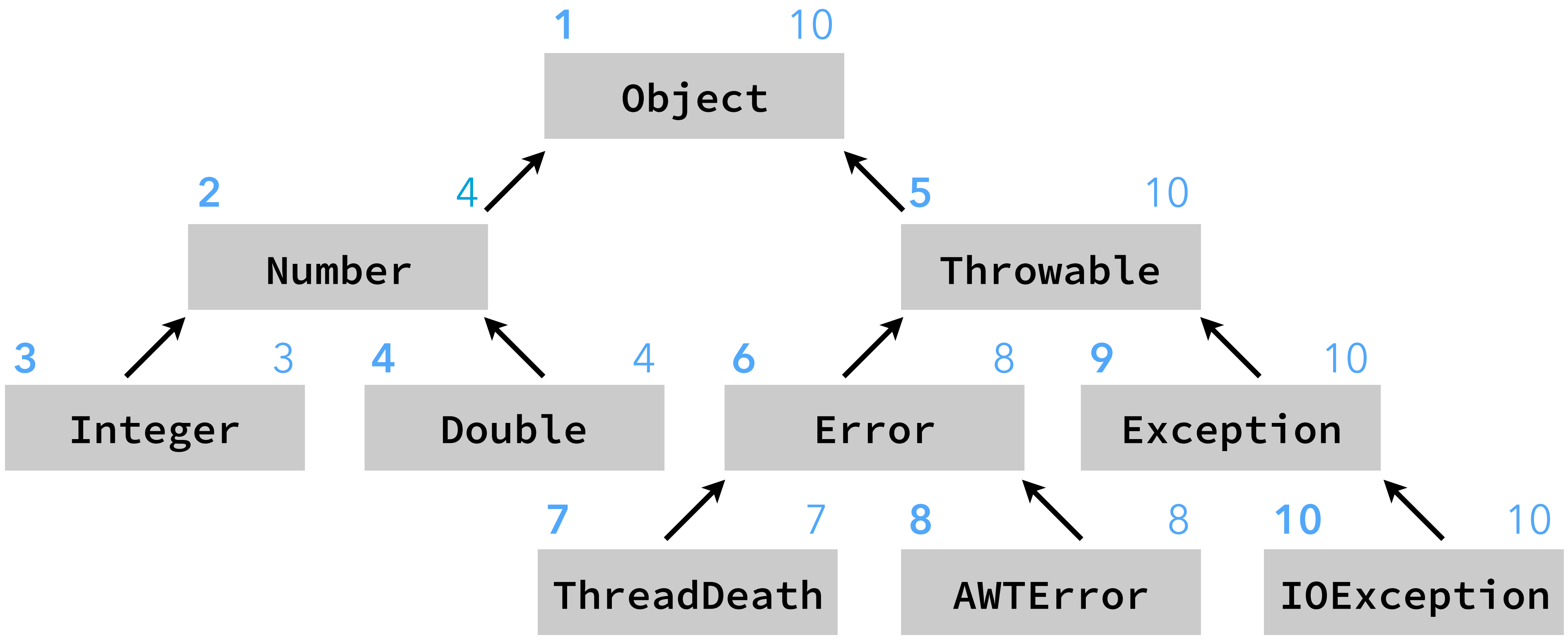
Relative numbering example



Relative numbering example



Relative numbering example

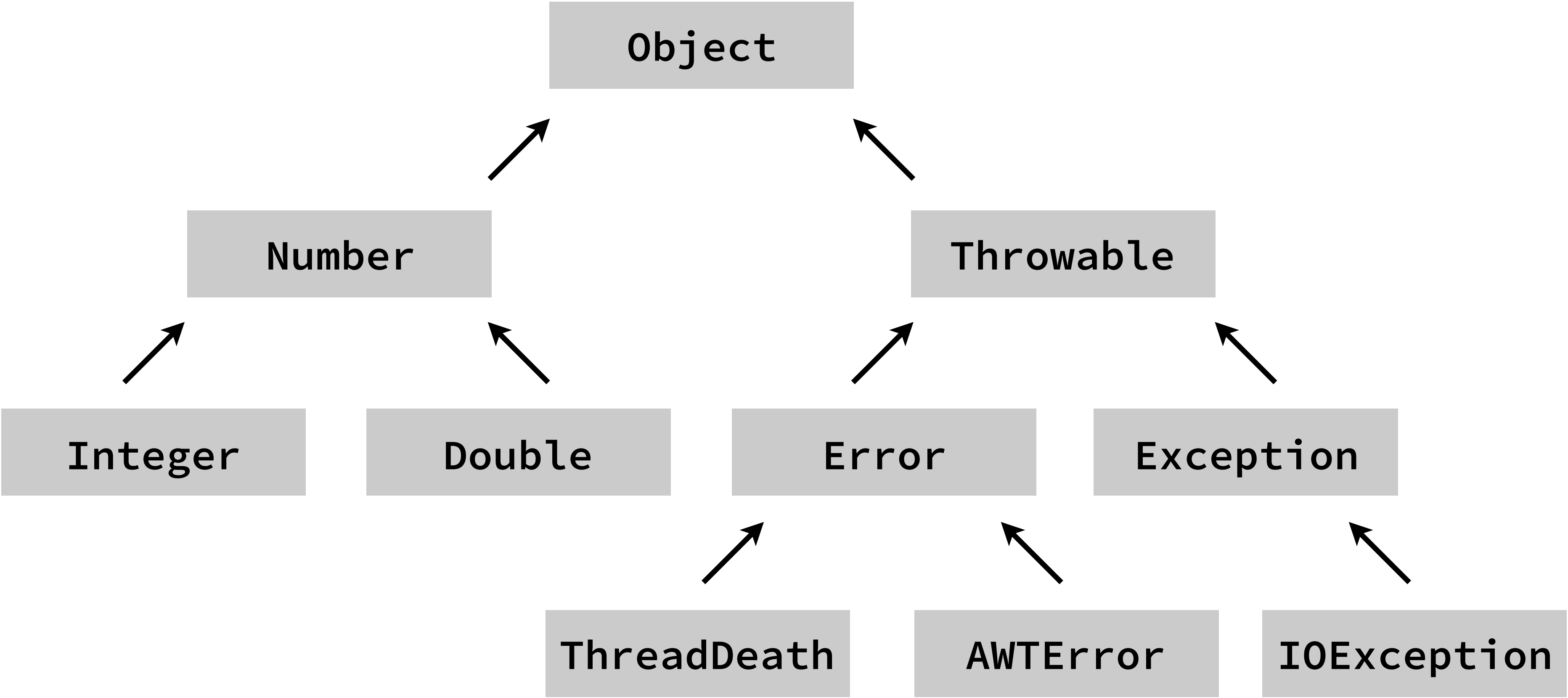


`x` **`instanceof`** `Throwable ⇔ 5 ≤ x.tid ≤ 10`

Cohen's encoding

The idea of **Cohen's encoding** is to first partition the types according to their level in the hierarchy. The **level** of a type T is defined as the length of the path from the root to T.
Then, all types are numbered so that no two types at a given level have the same number.
Finally, a **display** is attached to all types T, mapping all levels L smaller or equal to that of T to the number of the ancestor of T at level L.

Cohen's encoding example



Object

Number

Throwable

Integer

Double

Error

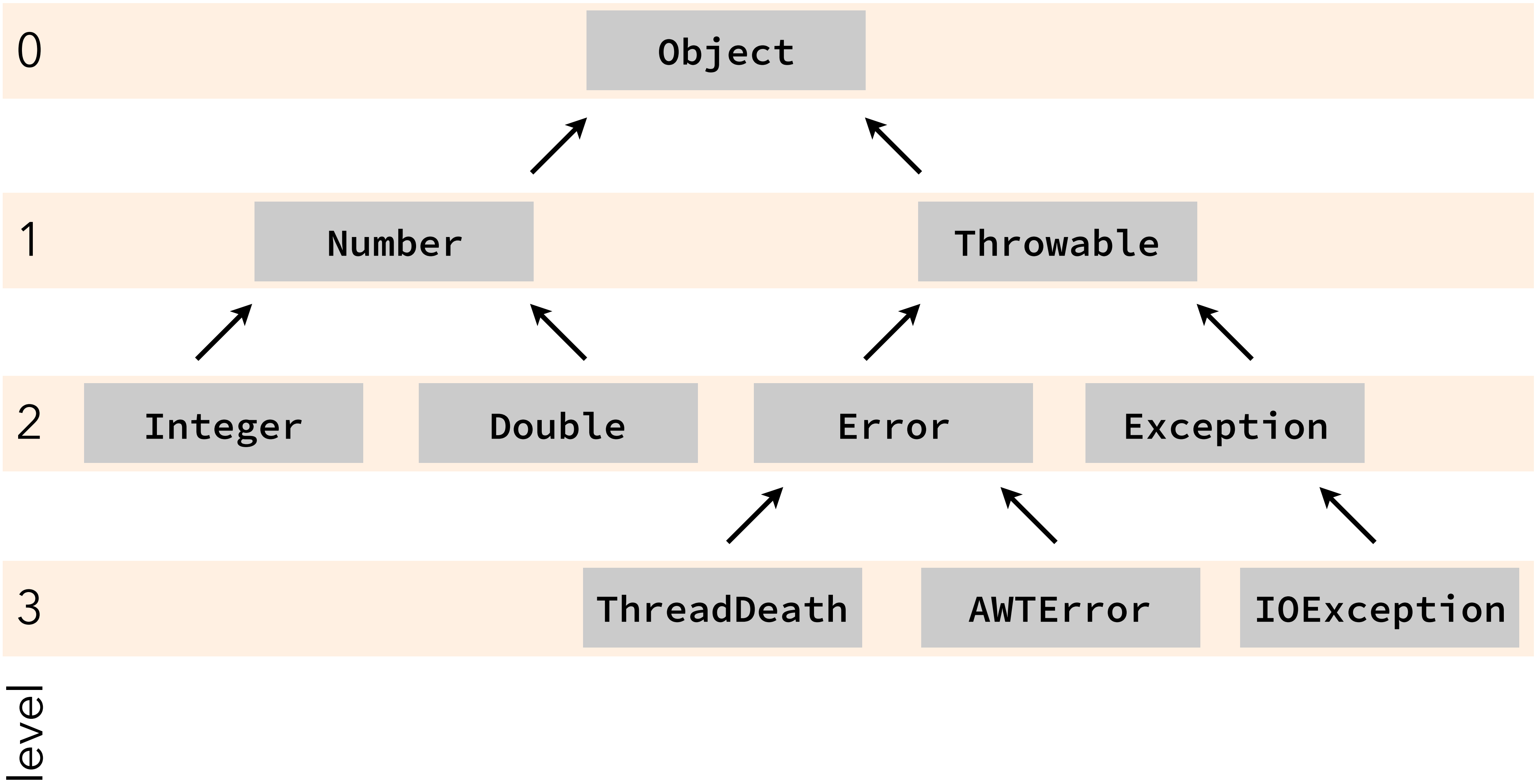
Exception

ThreadDeath

AWTError

IOException

Cohen's encoding example



0 `Object`

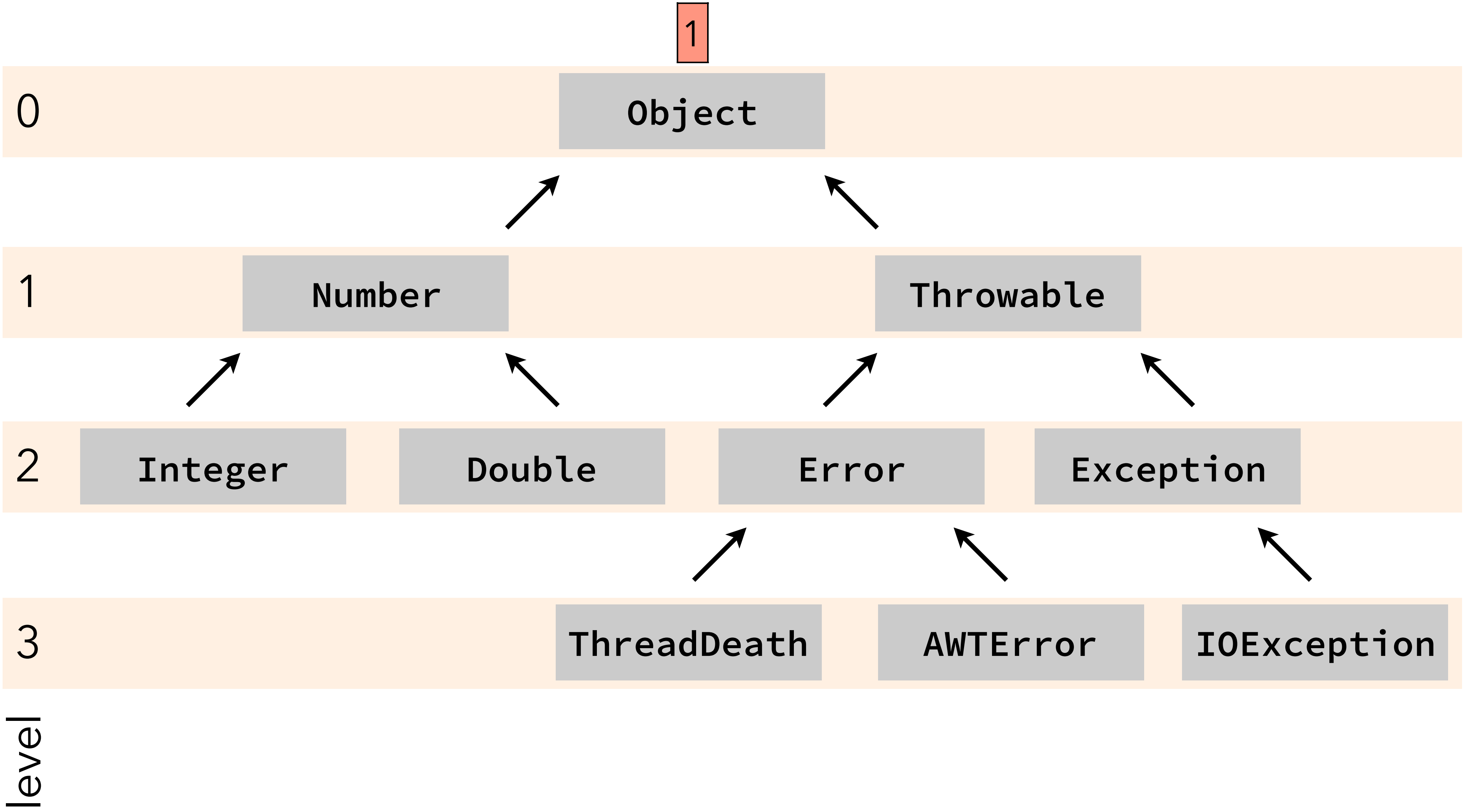
1 `Number` `Throwable`

2 `Integer` `Double` `Error` `Exception`

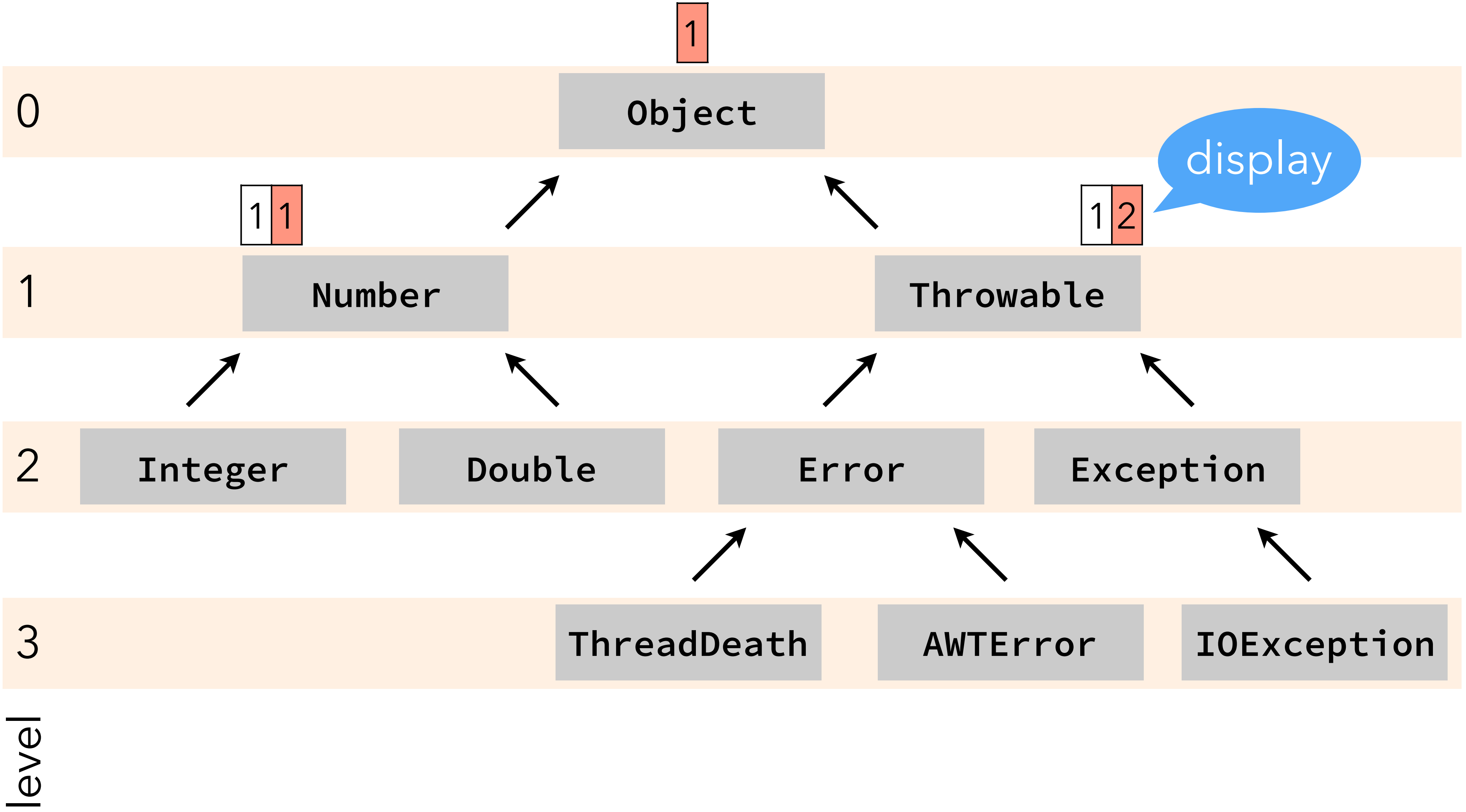
3 `ThreadDeath` `AWTError` `IOException`

level

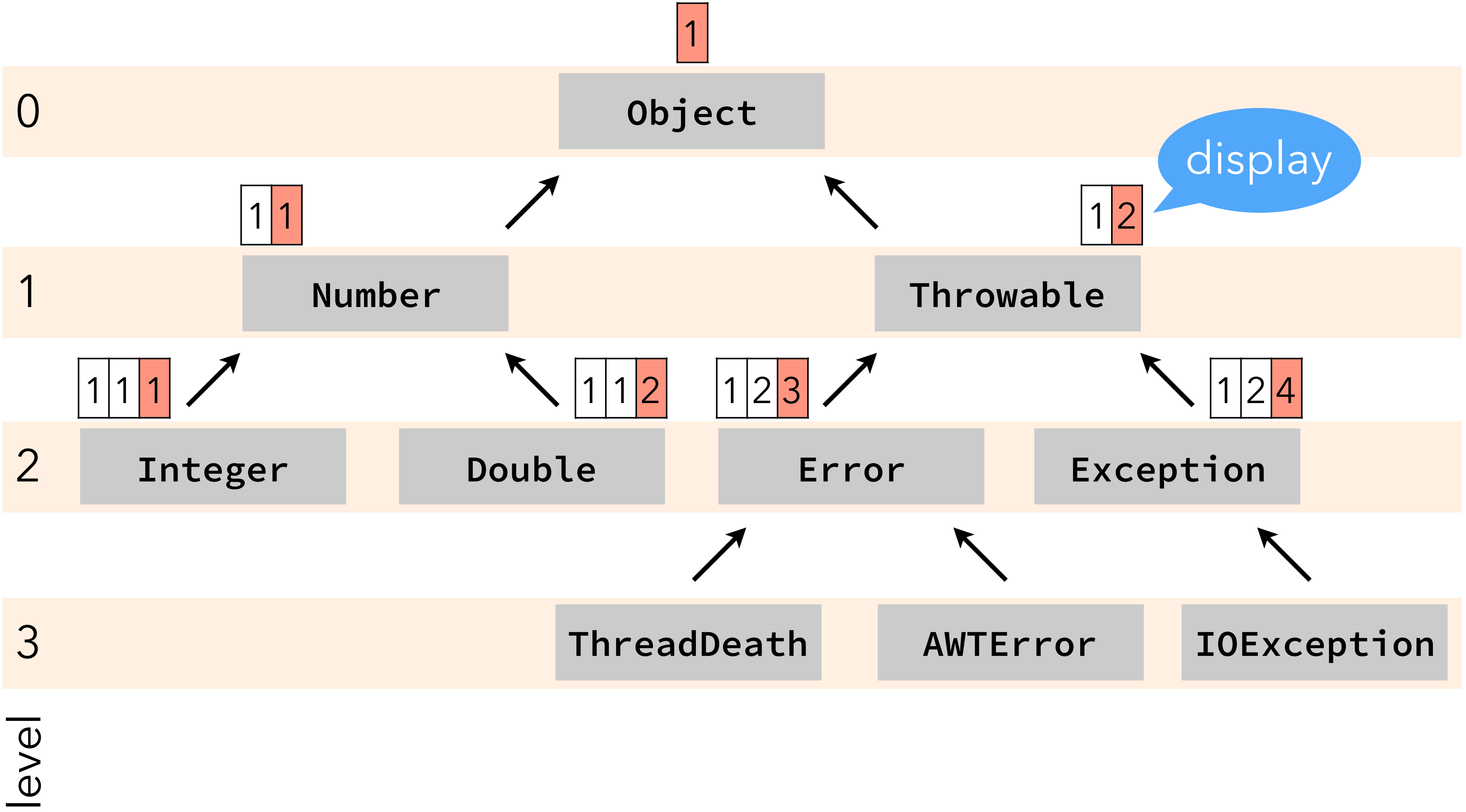
Cohen's encoding example



Cohen's encoding example



Cohen's encoding example



1

0 **Object**

display

1 1

1 2

1 **Number** **Throwable**

1 1 1

1 1 2

1 2 3

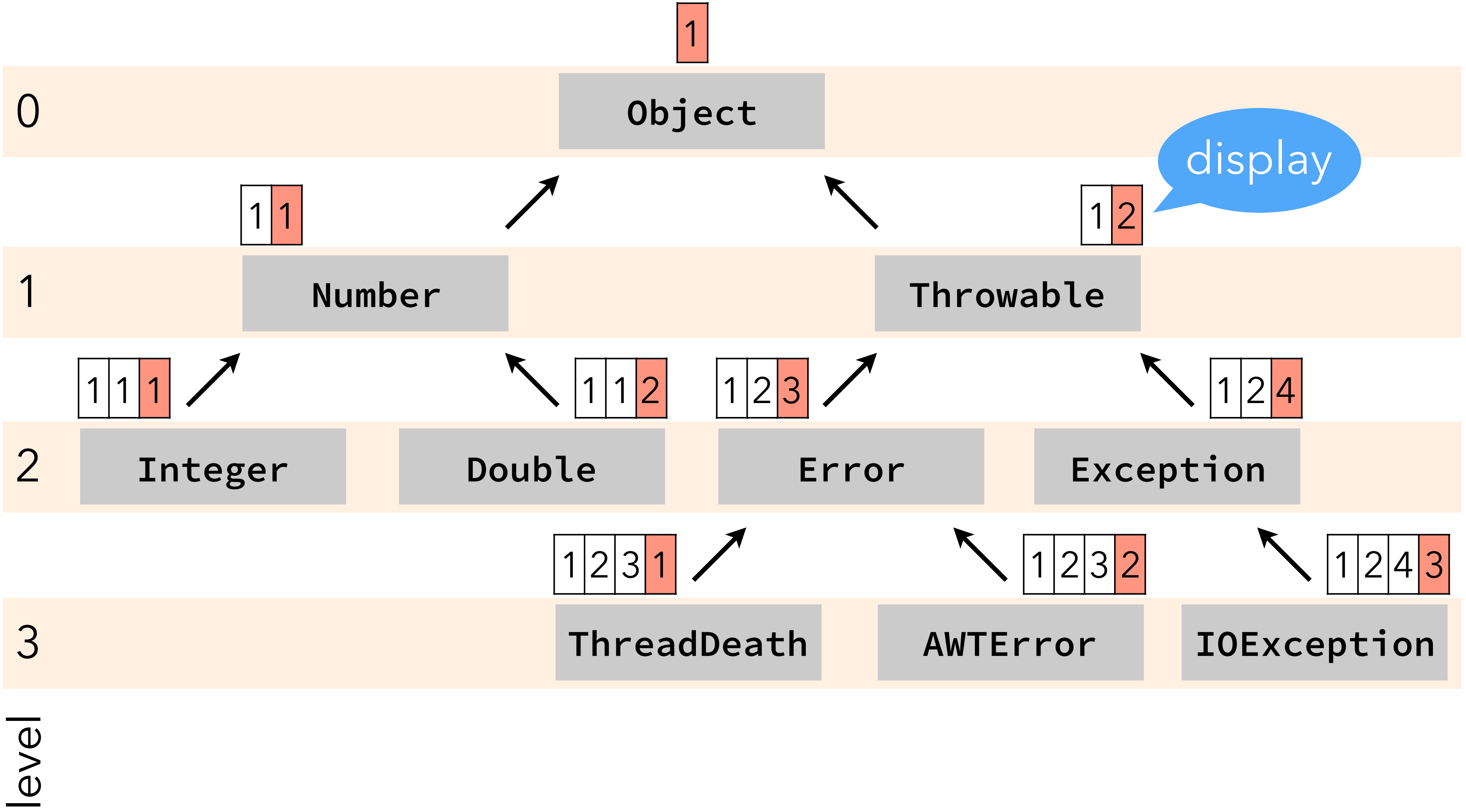
1 2 4

2 **Integer** **Double** **Error** **Exception**

3 **ThreadDeath** **AWTError** **IOException**

level

Cohen's encoding example



1

0 Object

display

1 1 | 1 2

1 Number | Throwable

1 1 1 | 1 1 2 | 1 2 3 | 1 2 4

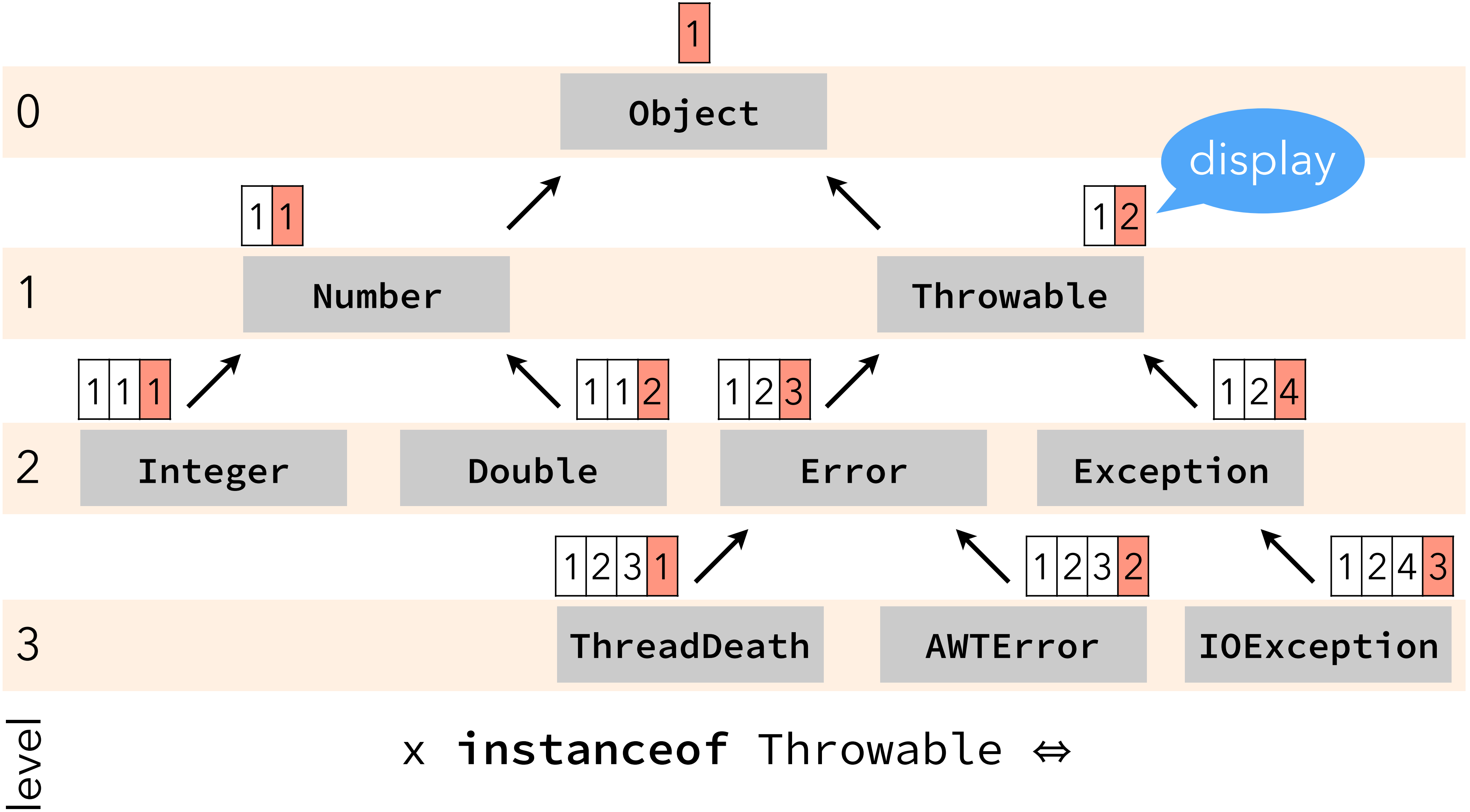
2 Integer | Double | Error | Exception

1 2 3 1 | 1 2 3 2 | 1 2 4 3

3 ThreadDeath | AWTError | IOException

level

Cohen's encoding example



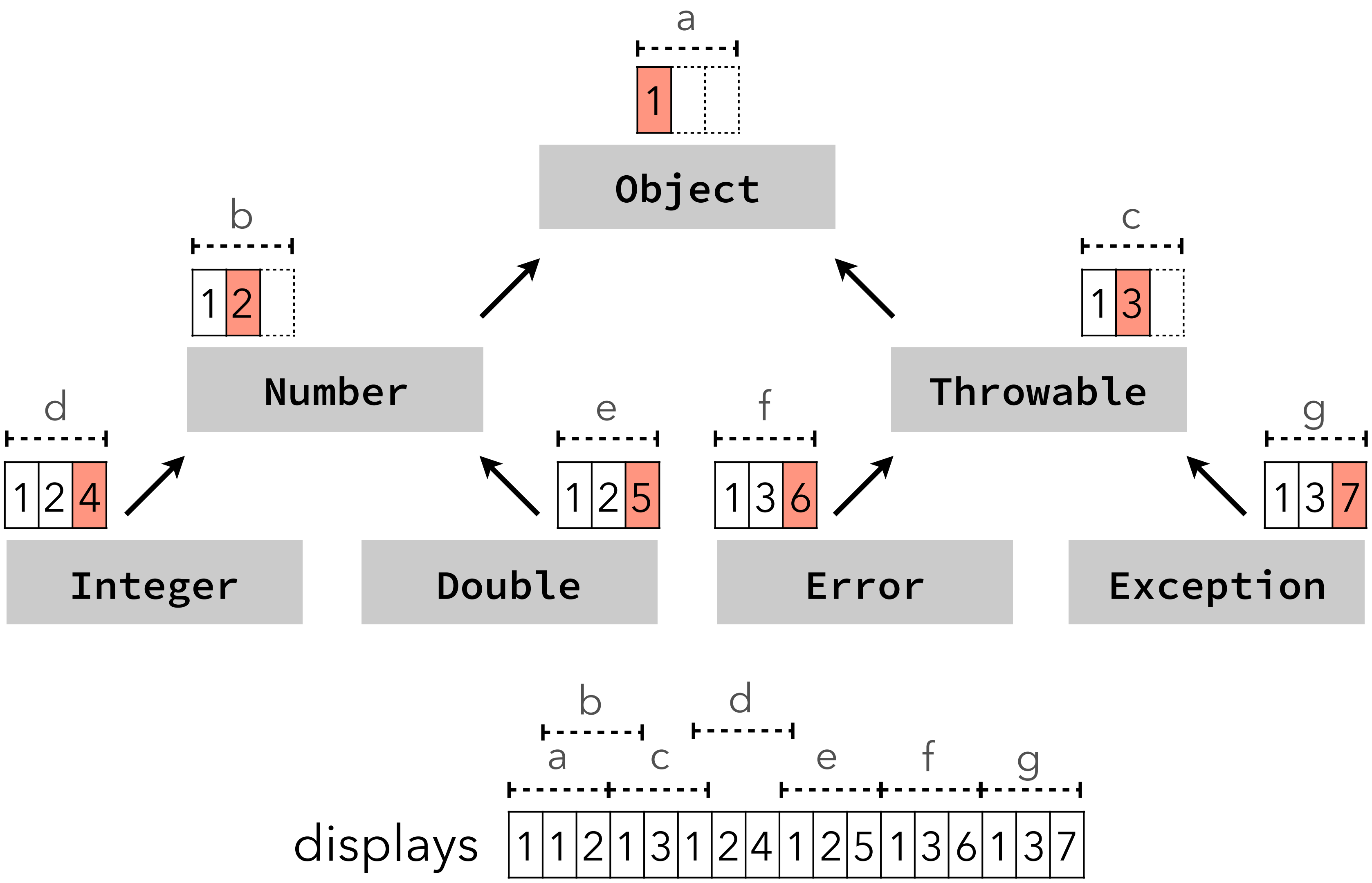
x **instanceof** Throwable ⇔

x.level ≥ 1 ∧ x.display[1] == 2

Global identifiers

If globally-unique identifiers are used for the types – instead of level-unique ones – then the display bounds check can be removed if the displays are stored consecutively in memory, with the longest ones at the end.

Cohen's encoding (global)



`x instanceof Throwable ⇔ x.display[1] == 3`

Comparison

While Cohen’s encoding is more complicated and requires more memory than relative numbering, it has the advantage of being **incremental**.

That is, it is possible to add new types to the hierarchy without having to recompute all the information attached to types.

This characteristic is important for systems where new types can be added at run time, e.g. Java.

Case 2:
multiple subtyping

Membership test

In a multiple subtyping setting, neither relative numbering nor Cohen’s encoding can be used directly.
Techniques that work with multiple subtyping can however be derived from them. We will examine three of them:

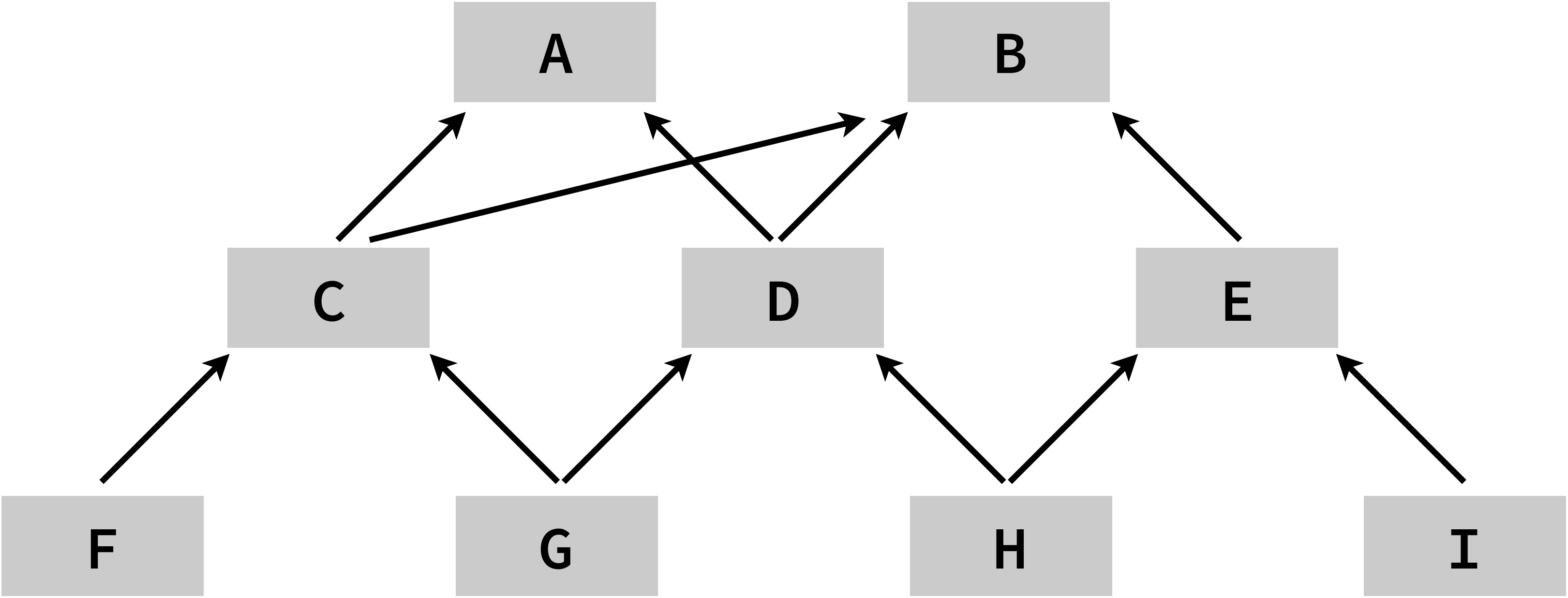
1. range compression,
2. packed encoding, and
3. PQ encoding.

Range compression

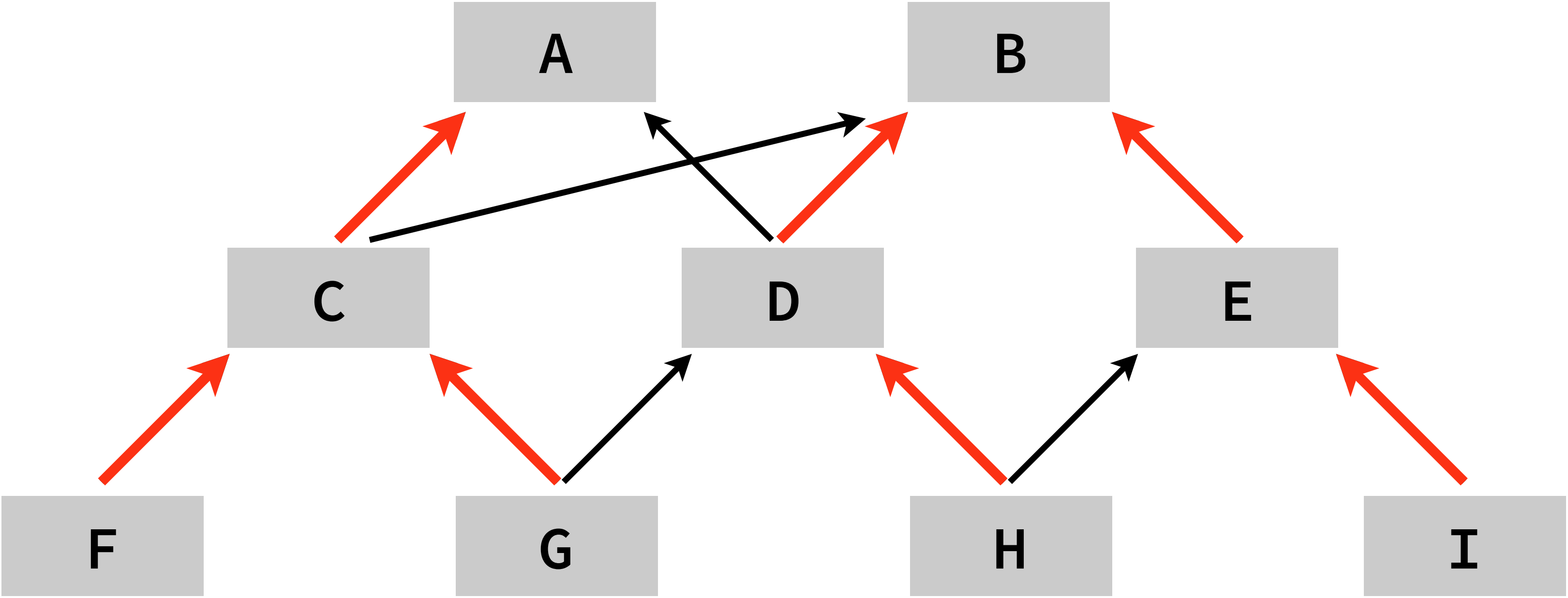
Range compression is a generalization of relative numbering to a multiple subtyping setting.

The idea of this technique is to uniquely number all types of the hierarchy by traversing one of its spanning forests – chosen judiciously. Then, each type carries the numbers of all its descendants, represented as a list of disjoint intervals.

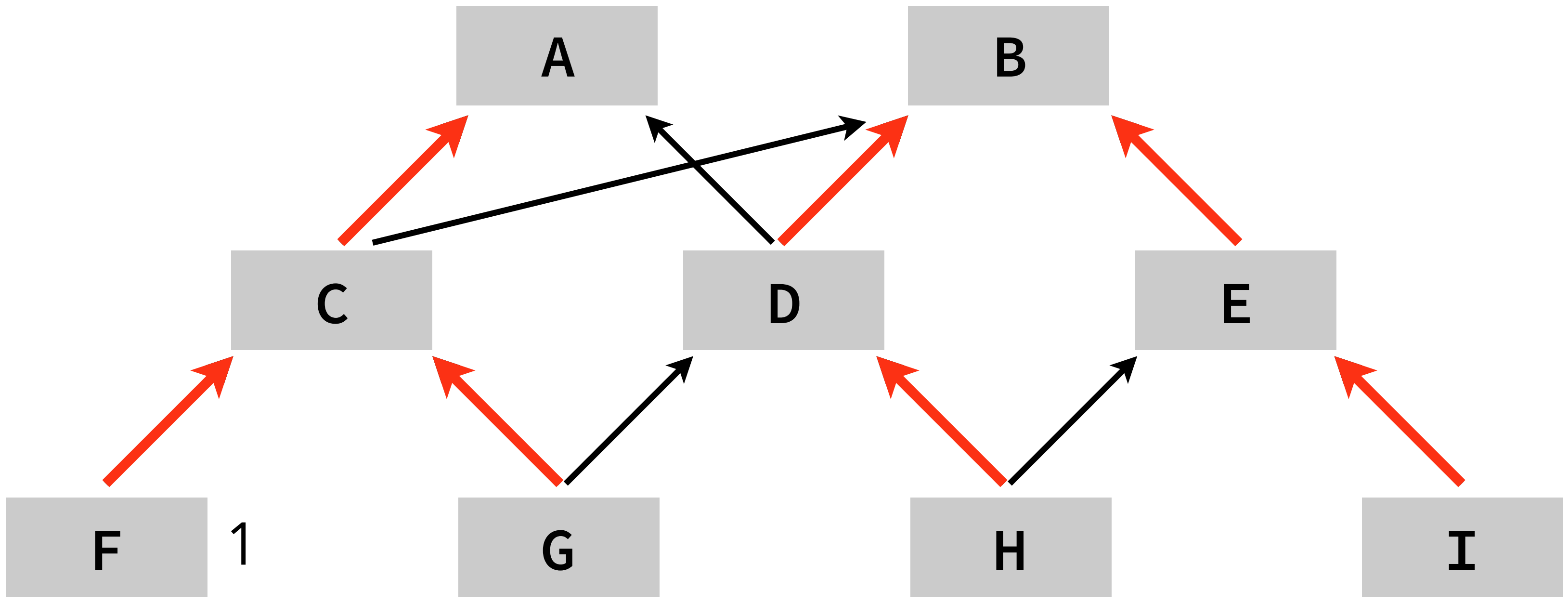
Range compression example



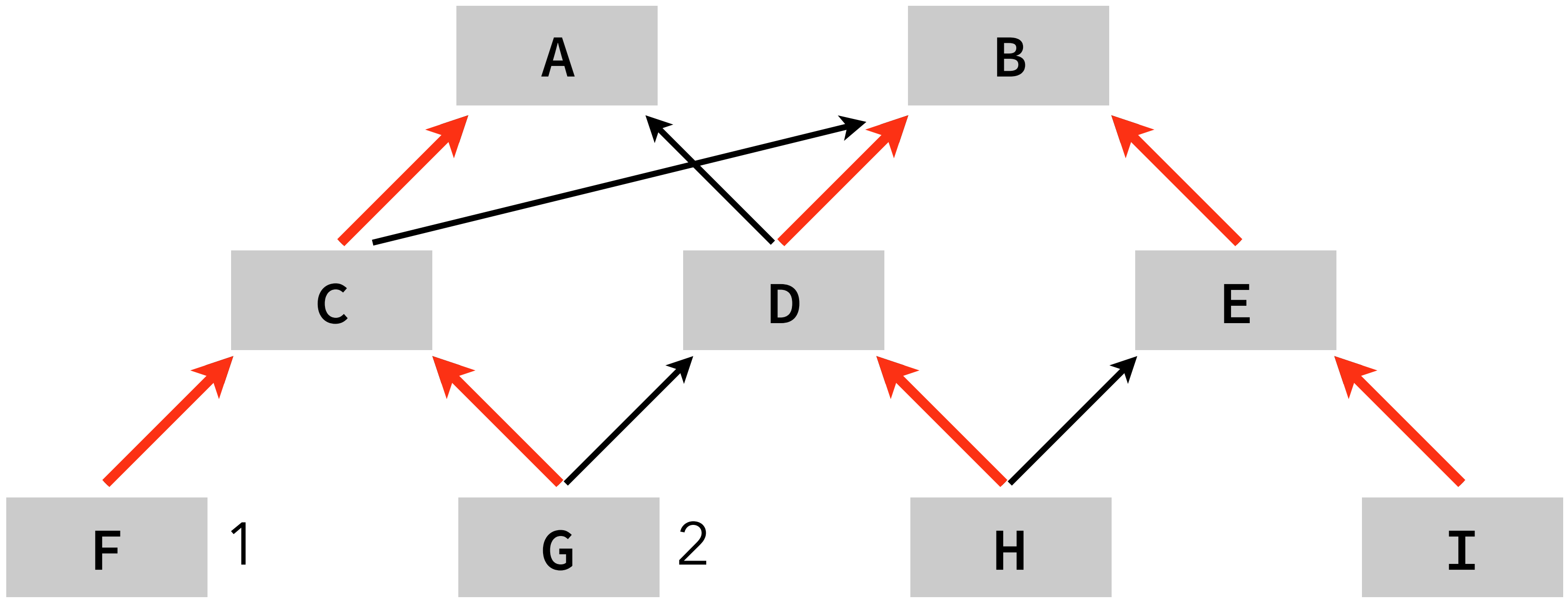
Range compression example



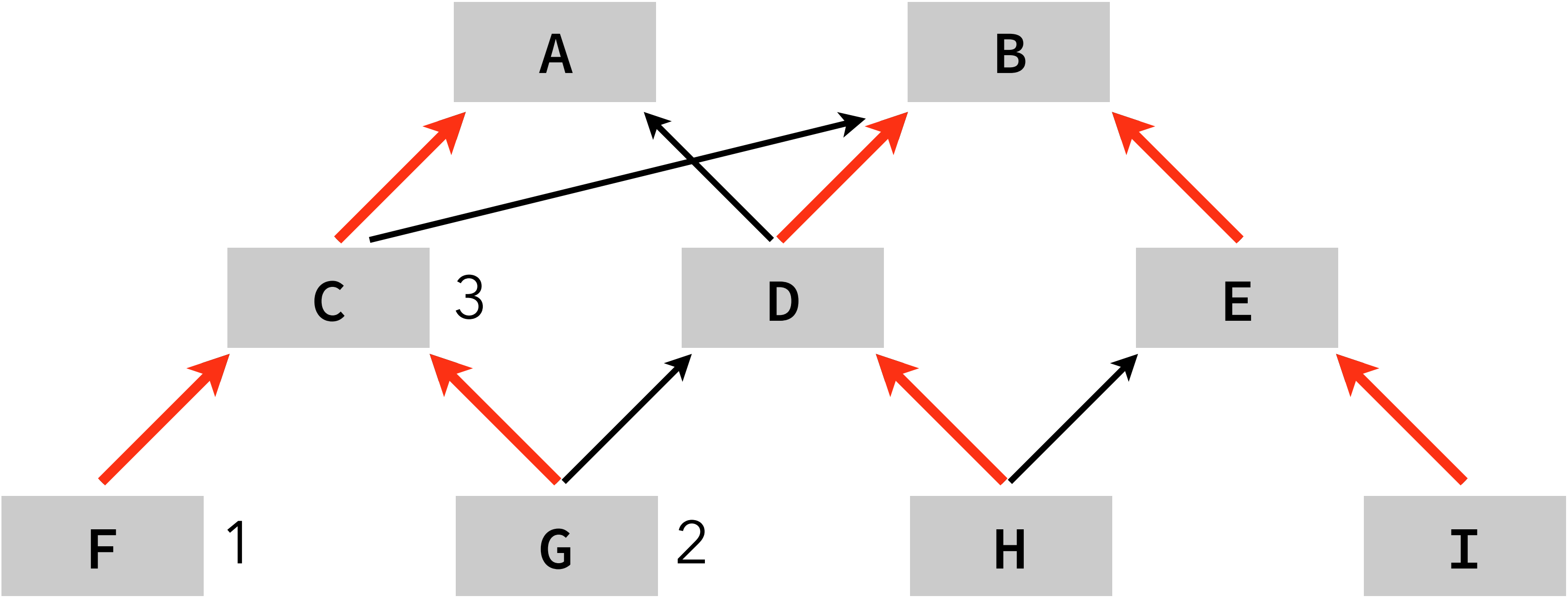
Range compression example



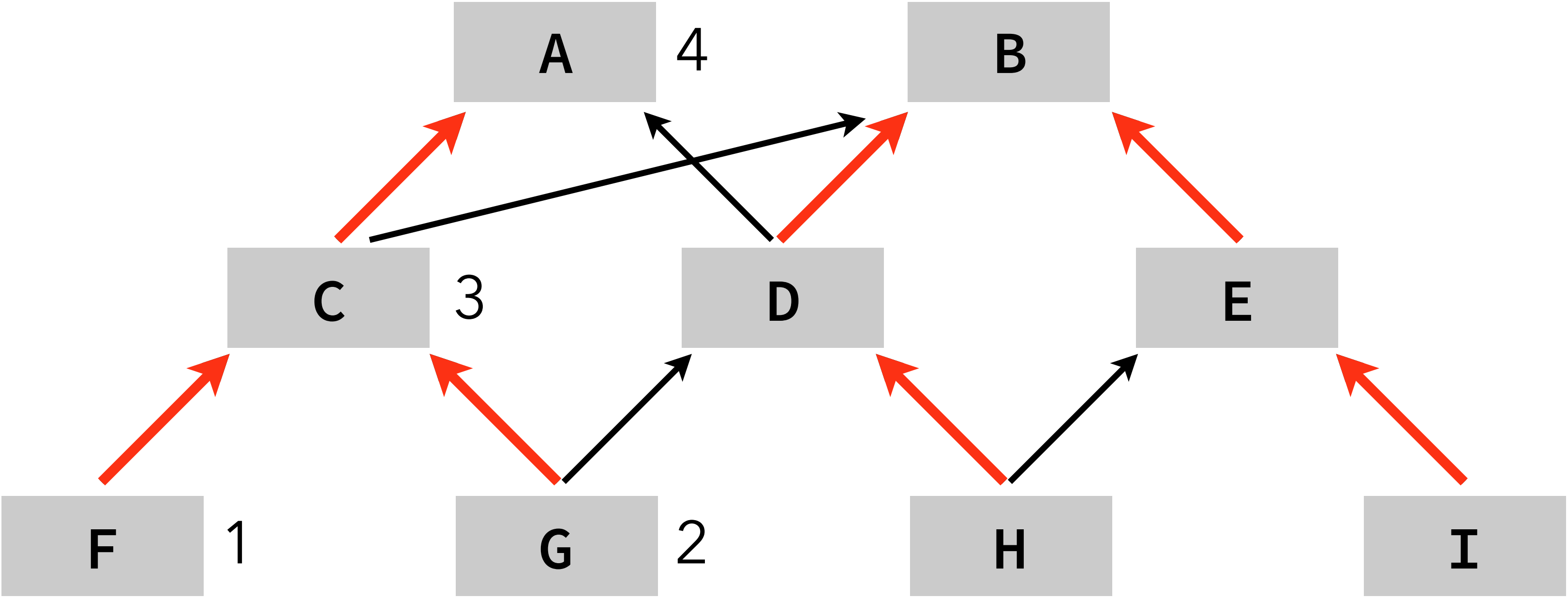
Range compression example



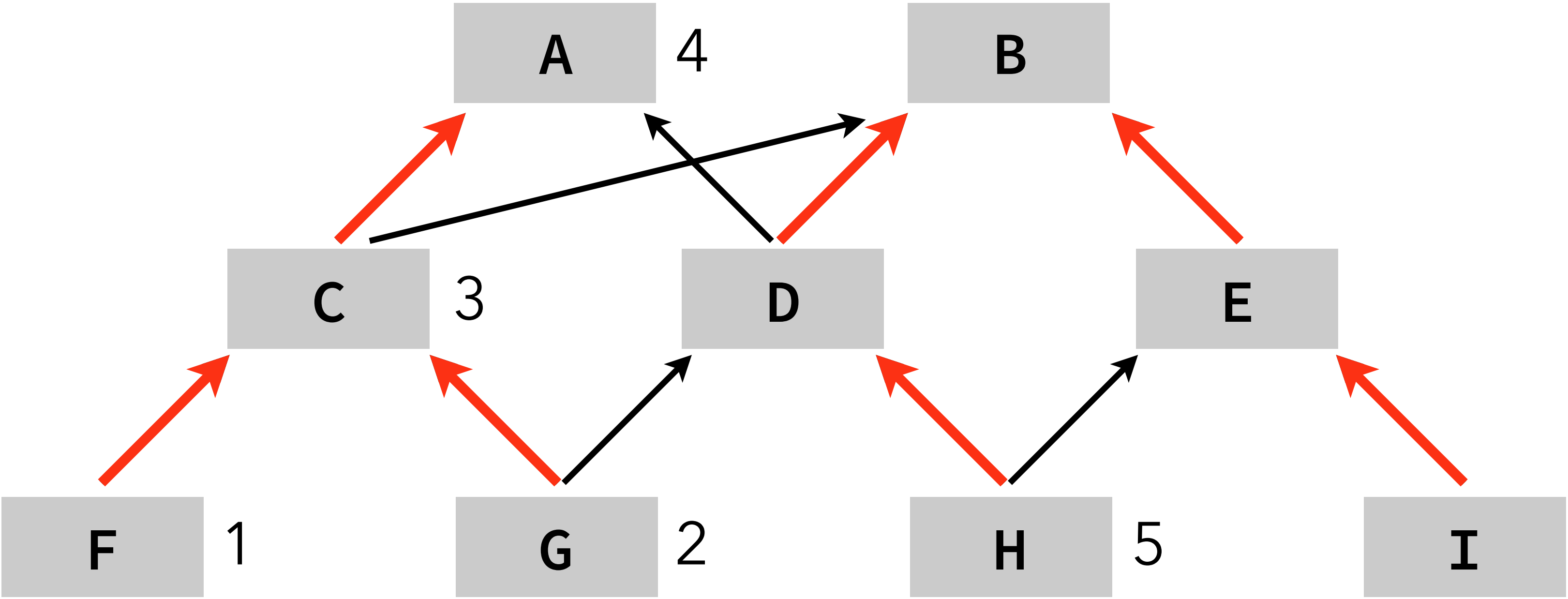
Range compression example



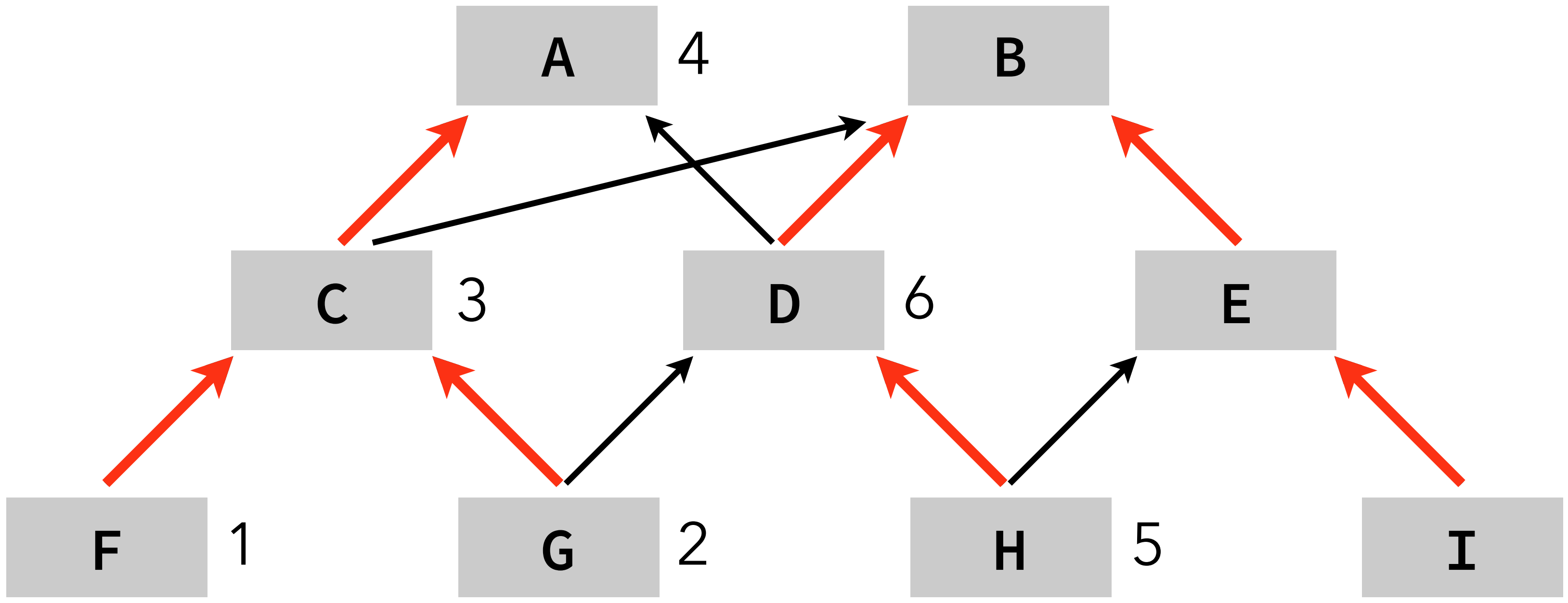
Range compression example



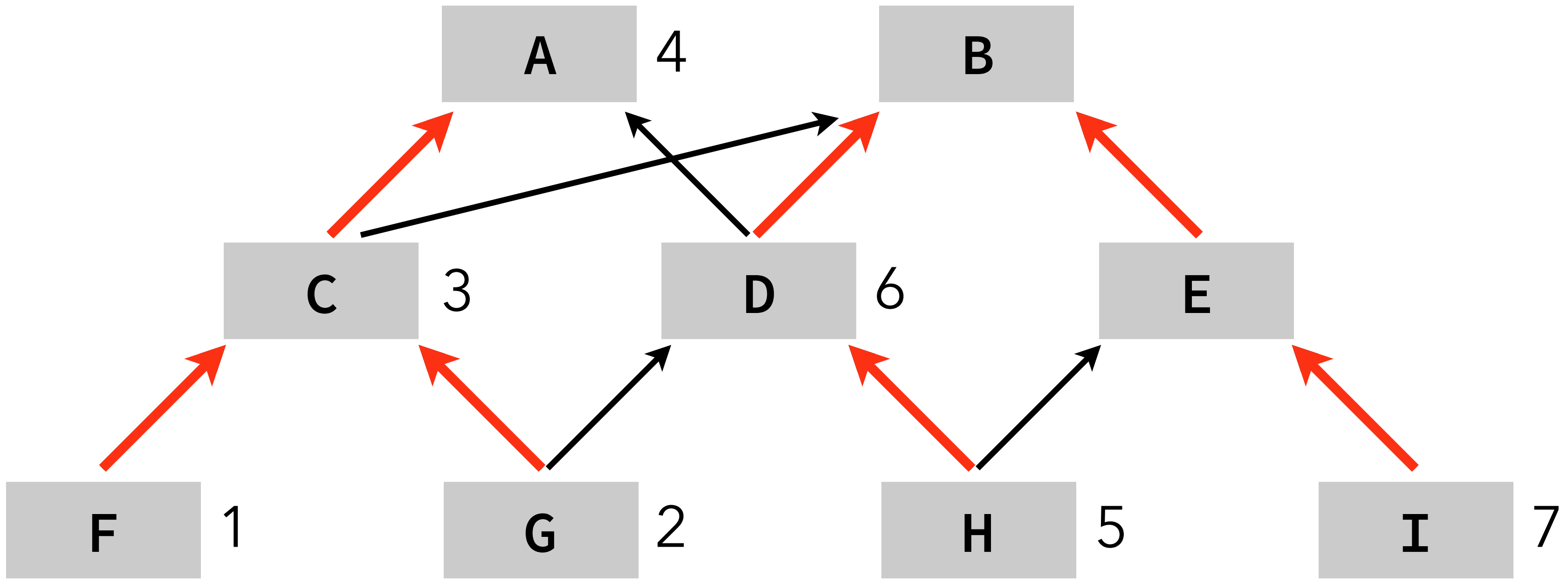
Range compression example



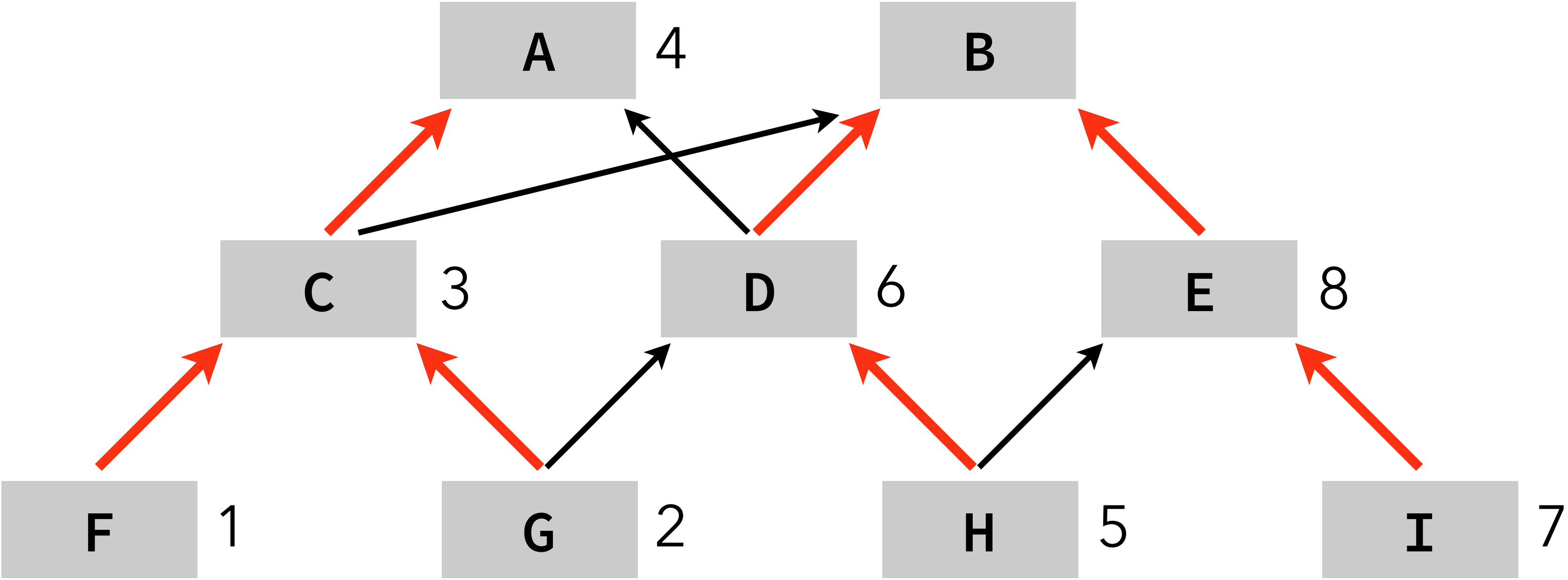
Range compression example



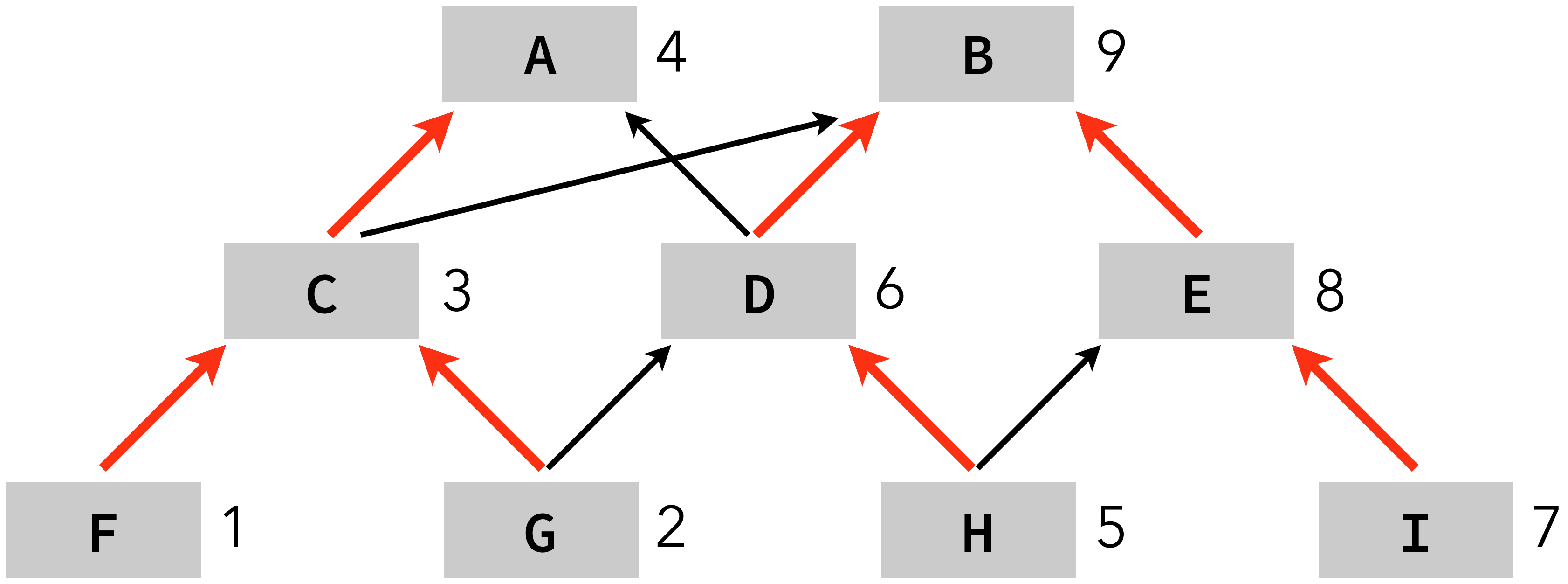
Range compression example



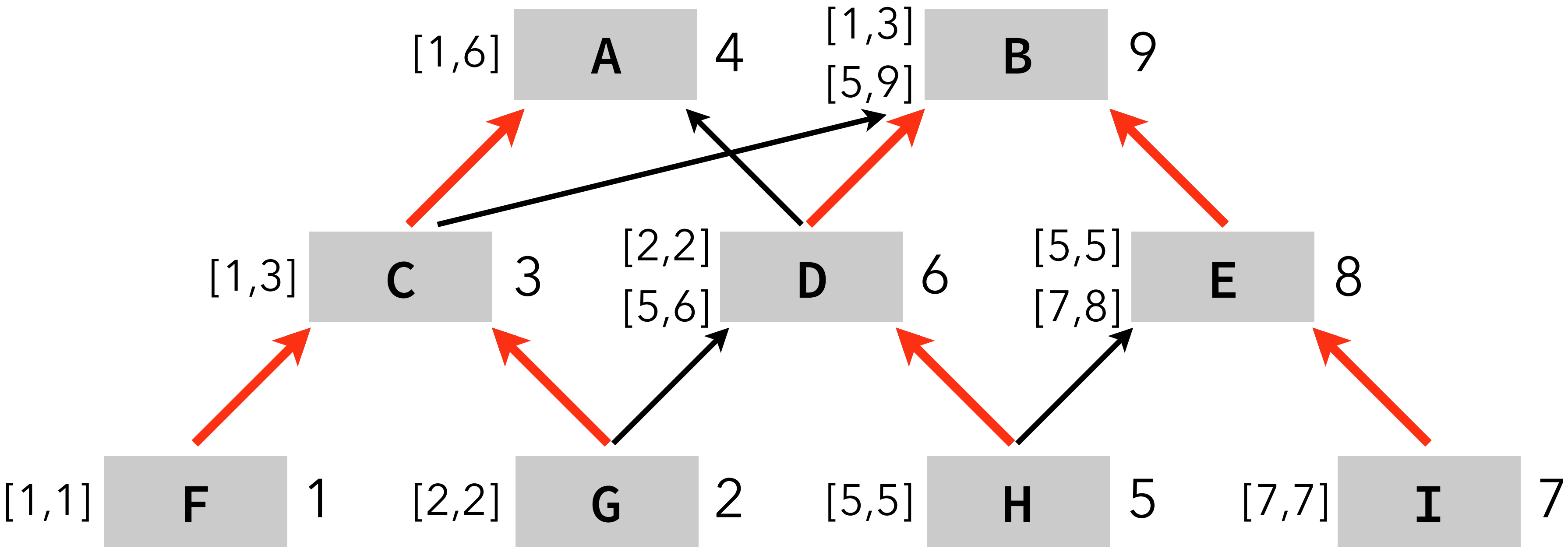
Range compression example



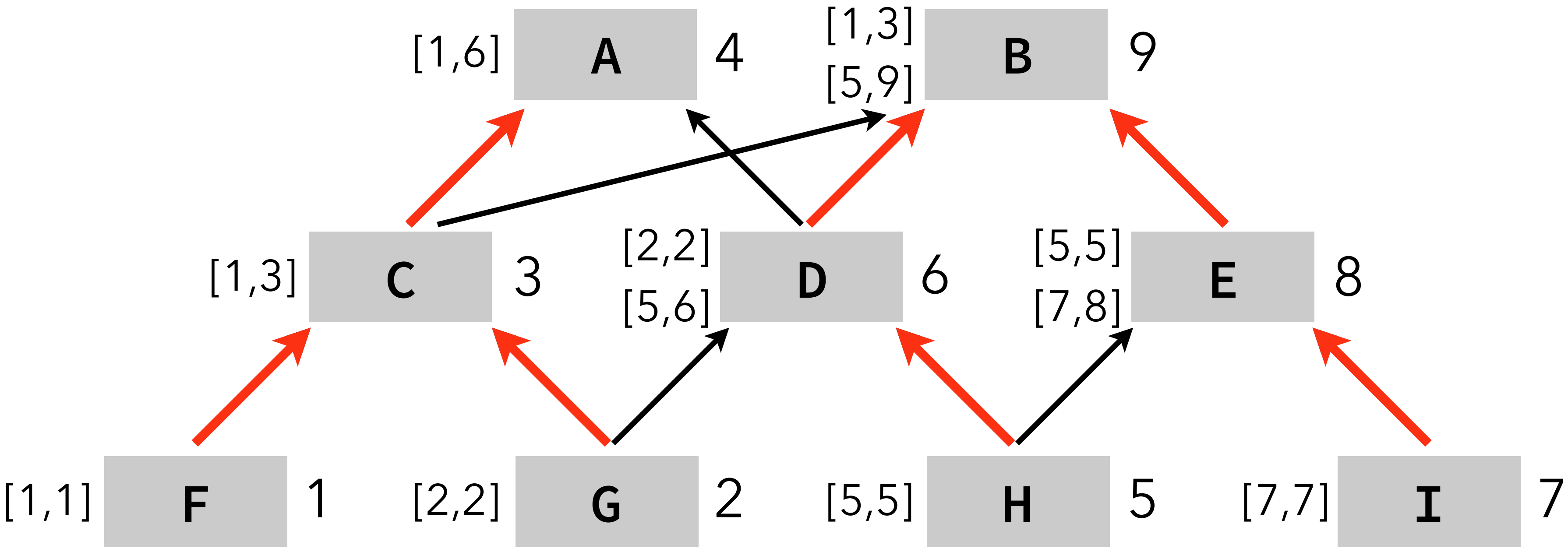
Range compression example



Range compression example



Range compression example

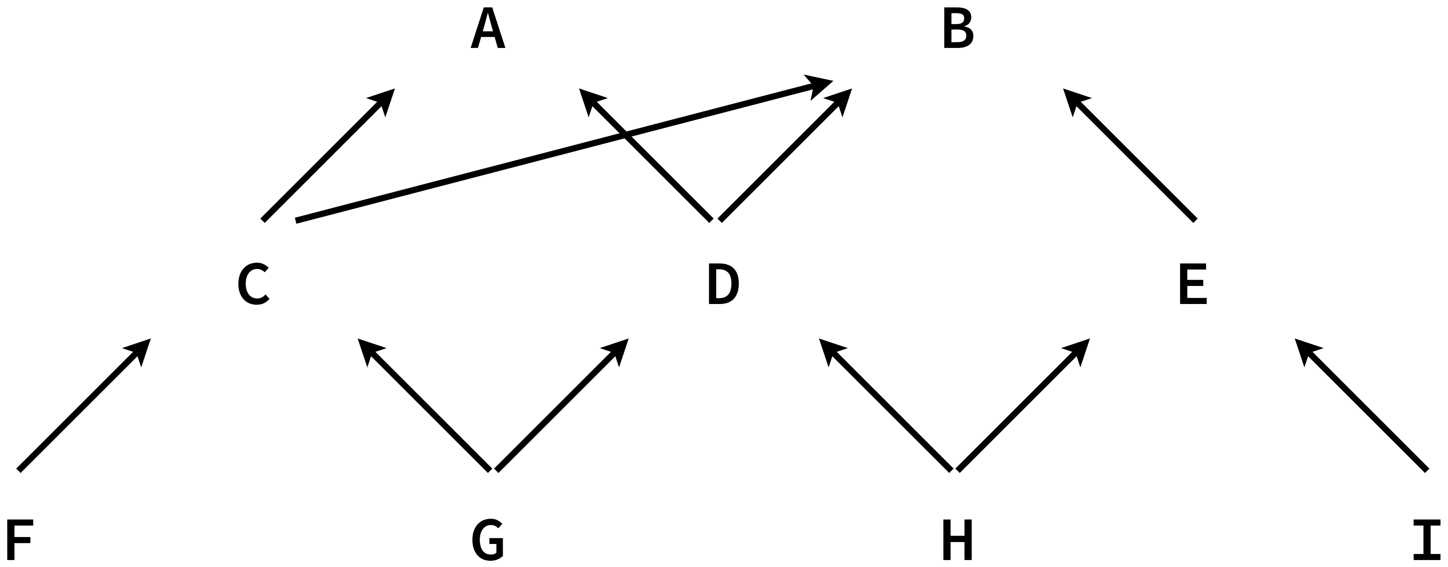


`x` **`instanceof`** `B ⇔ x.tid ∈ [1,3] ∨ x.tid ∈ [5,9]`

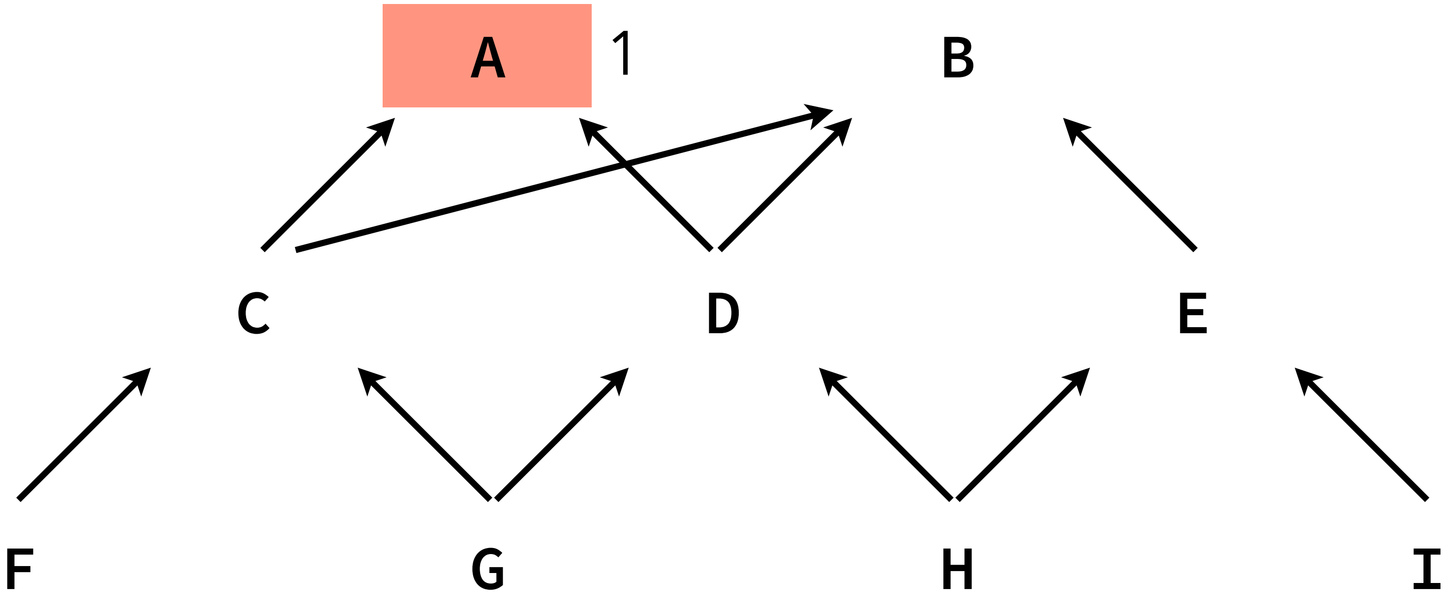
Packed encoding

Packed encoding is a generalization of Cohen's encoding to a multiple inheritance setting.
The idea of this technique is to partition types into **slices** – as few as possible – so that all ancestors of all types are in different slices. Types are then numbered uniquely in all slices. Finally, a display is attached to every type T, mapping slices to the ancestor of T in that slice.

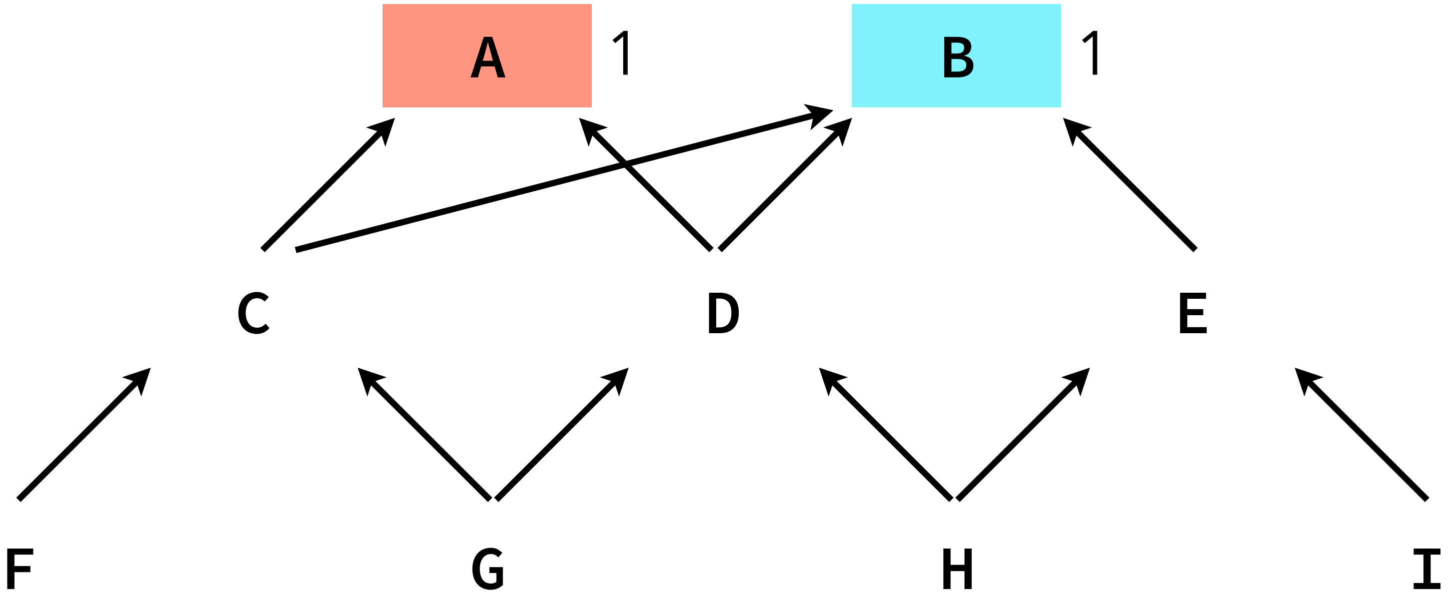
Packed encoding example



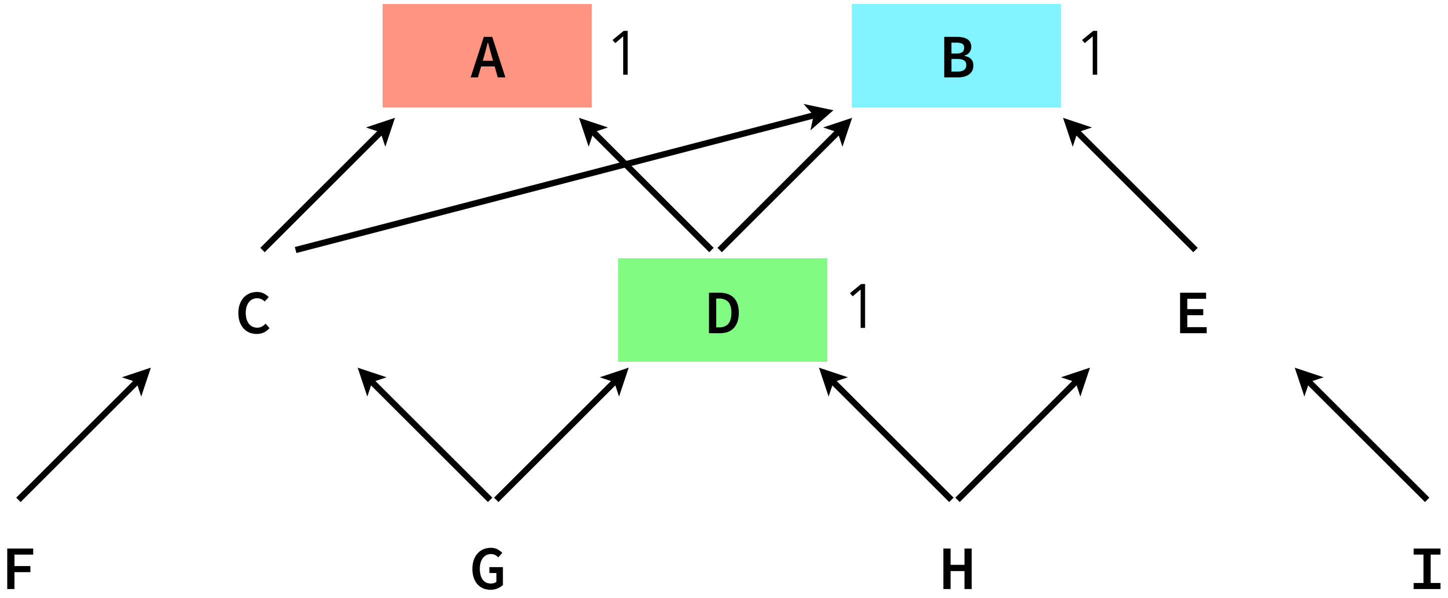
Packed encoding example



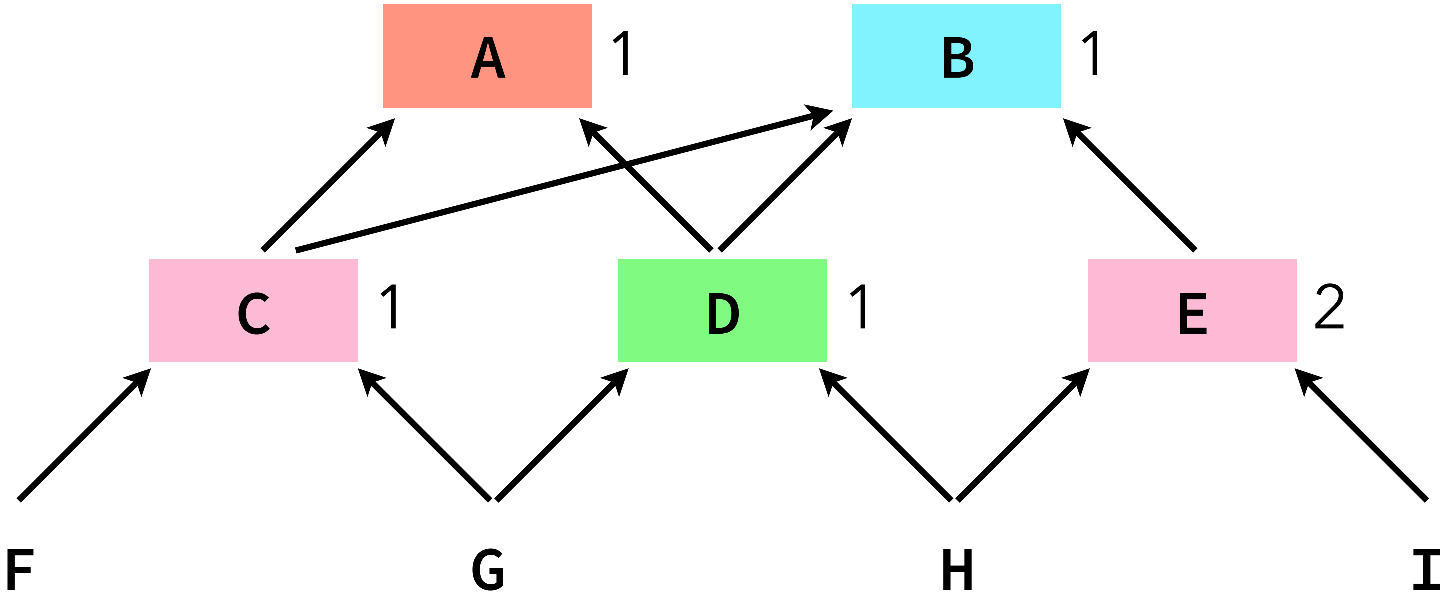
Packed encoding example



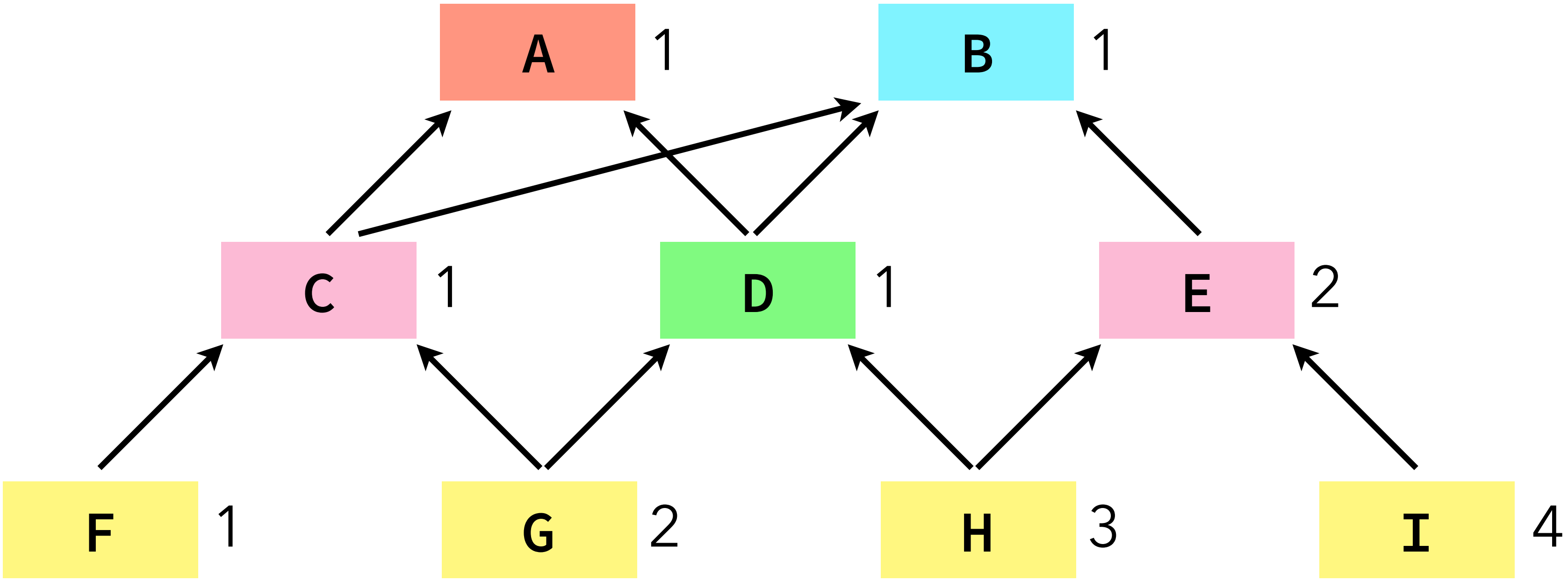
Packed encoding example



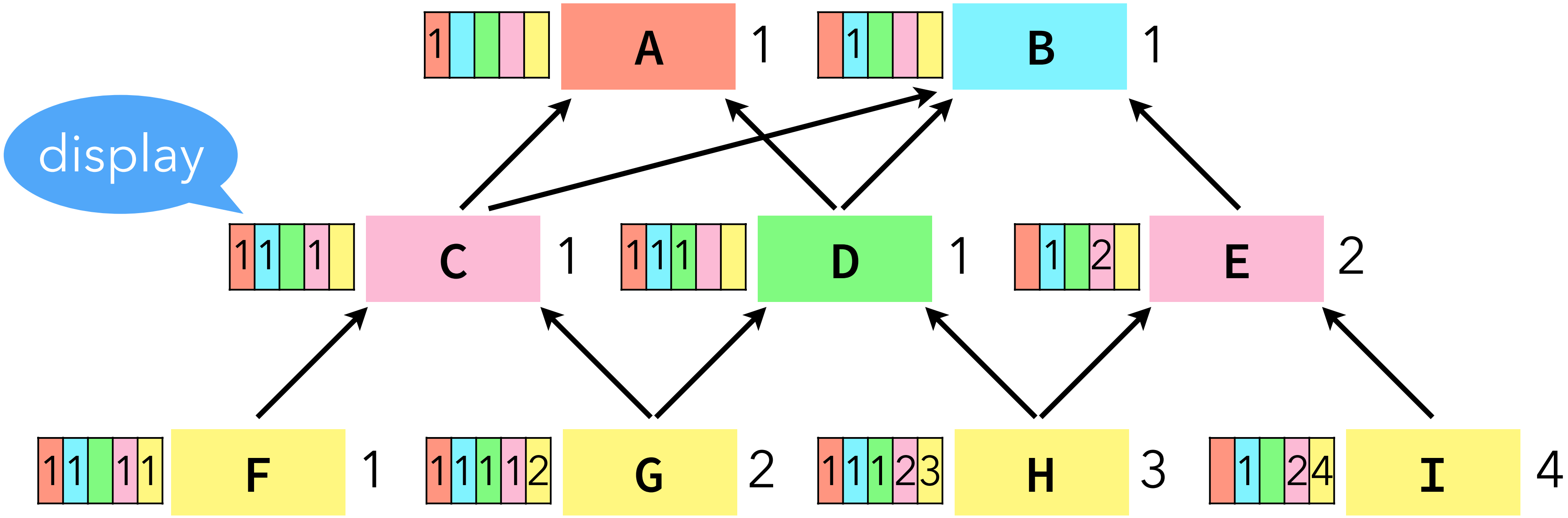
Packed encoding example



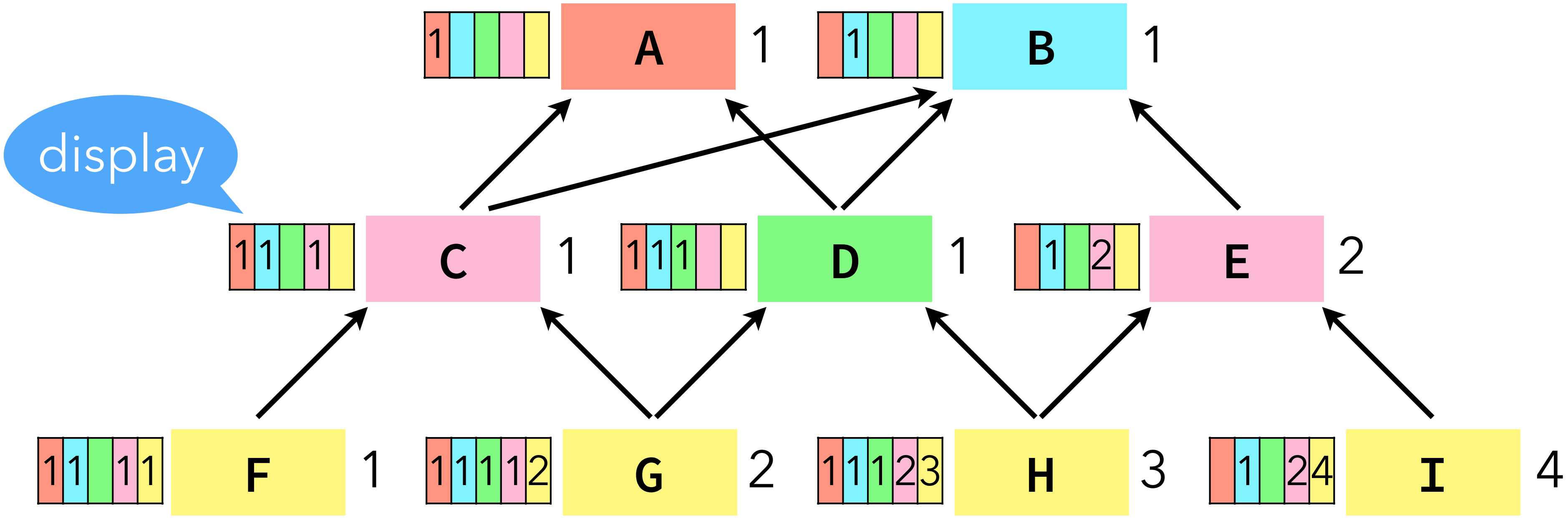
Packed encoding example



Packed encoding example



Packed encoding example



x **instanceof** B ⇔ x.display[1] == 1

Cohen's/packed encoding

It is easy to see that Cohen’s encoding is a special case of packed encoding, in which levels play the role of slices.
In a single inheritance setting, it is always valid to use levels as slices, since it is impossible for a type to have two ancestors at the same level – i.e. in the same slice.

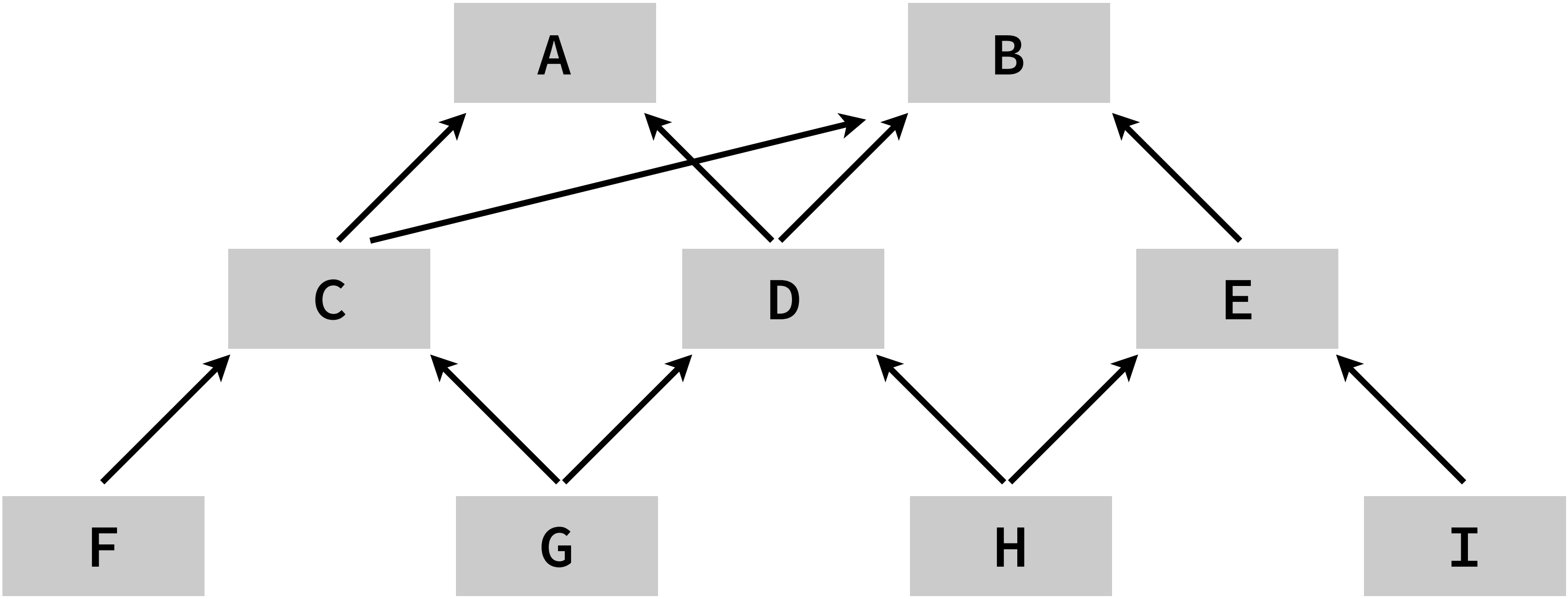
PQ encoding

PQ encoding borrows ideas from packed encoding and relative numbering.

It partitions types into slices – as few as possible – and attributes to types one unique identity per slice. The numbering of types is done so that the following property holds:

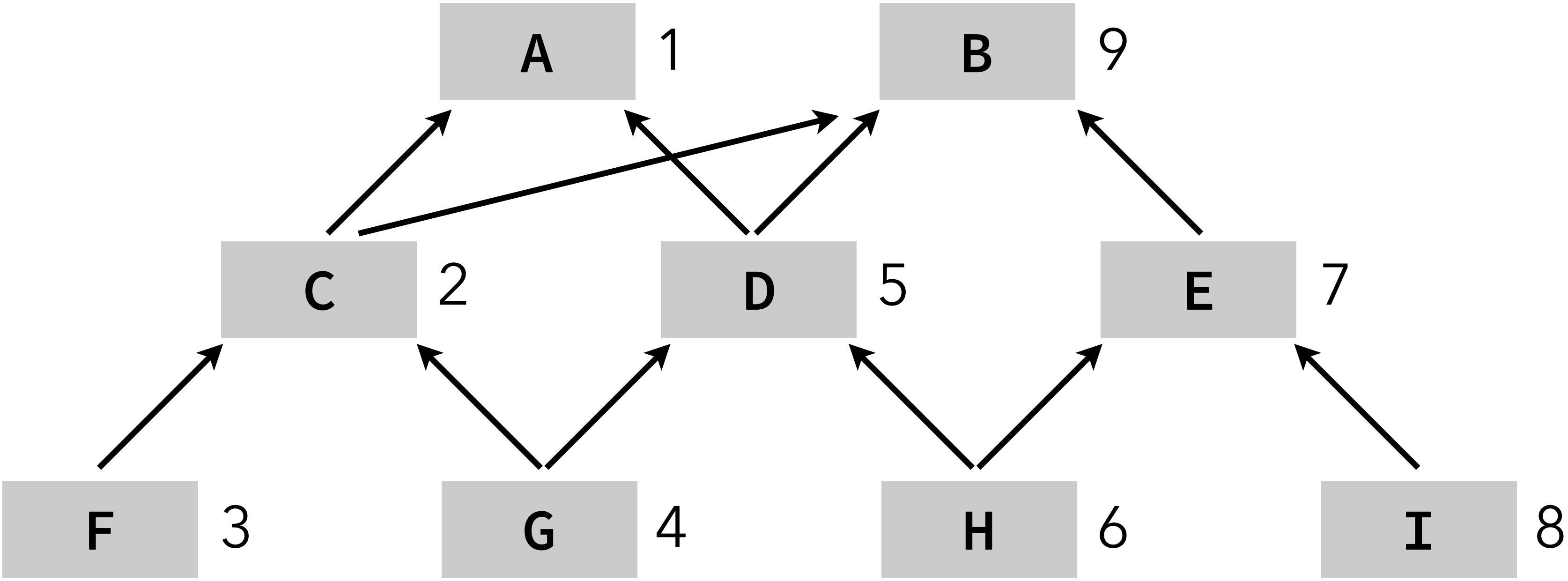
> For all types T in a slice S, all descendants of T – independently of their slice – are numbered consecutively in slice S.

PQ encoding example 1



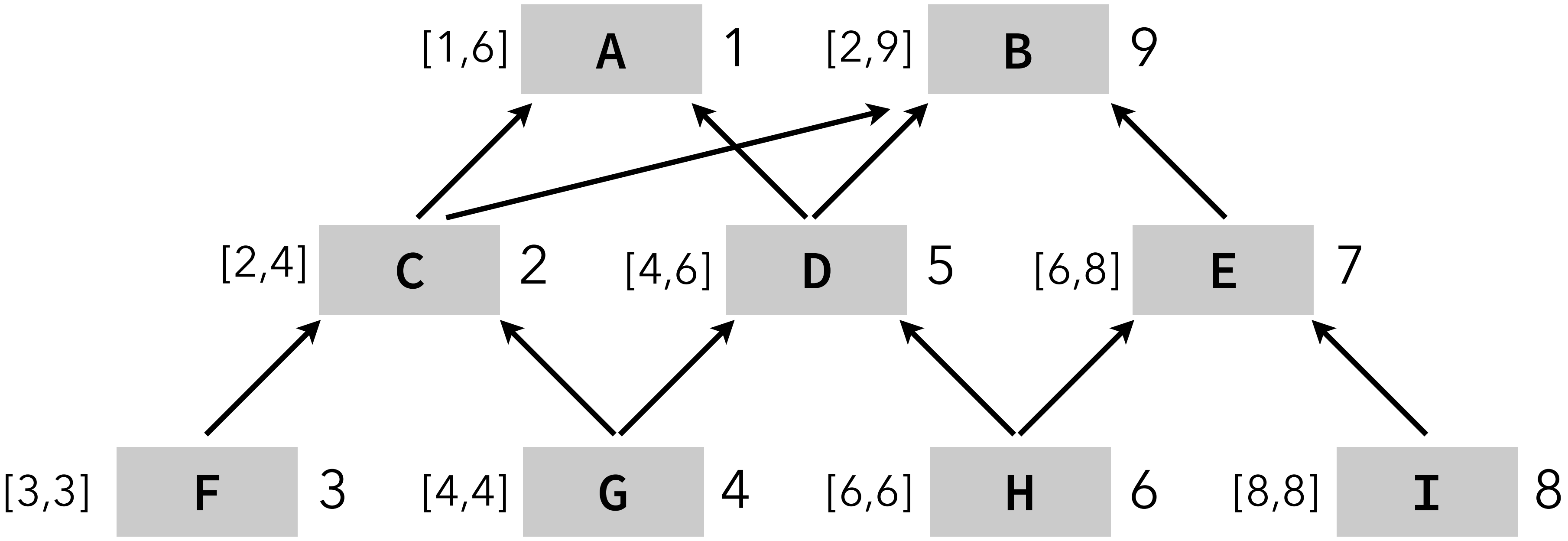
Note: a single slice is sufficient for this hierarchy.

PQ encoding example 1



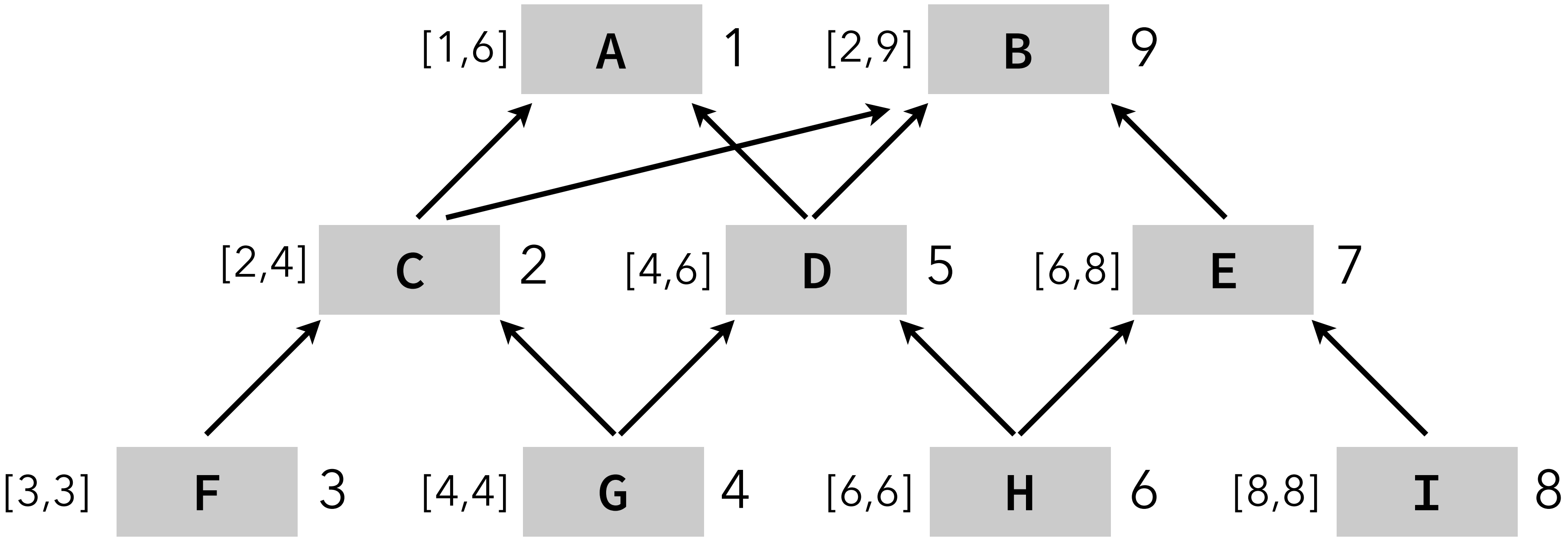
Note: a single slice is sufficient for this hierarchy.

PQ encoding example 1



Note: a single slice is sufficient for this hierarchy.

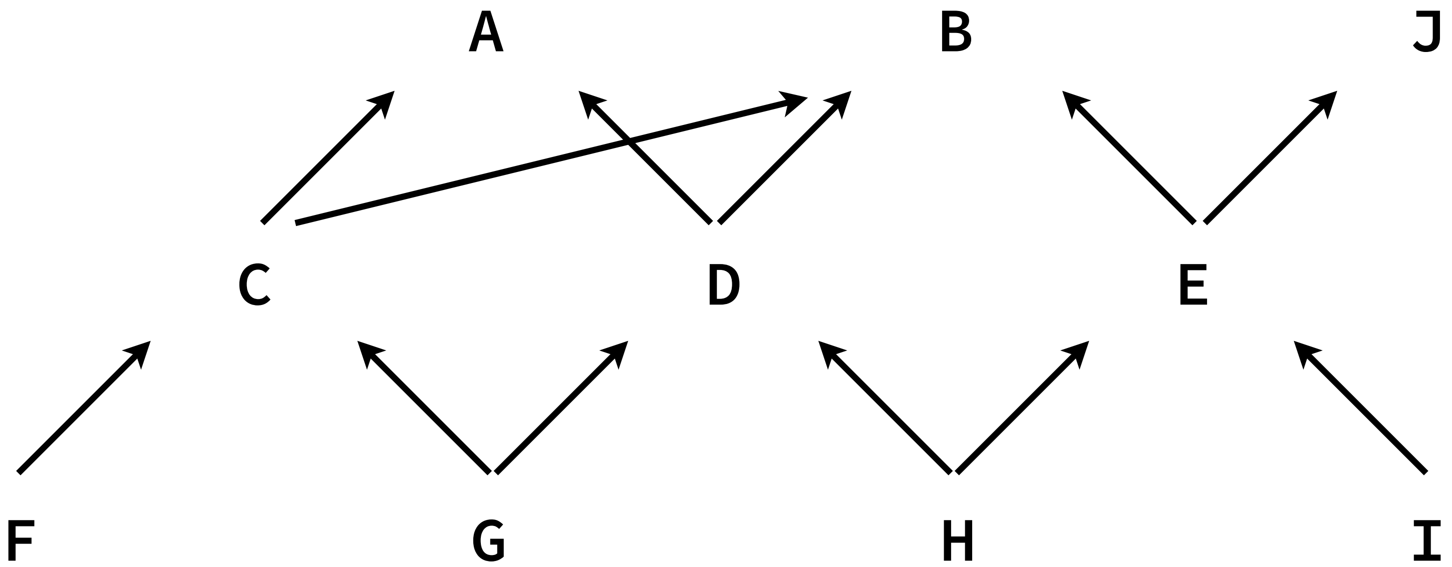
PQ encoding example 1



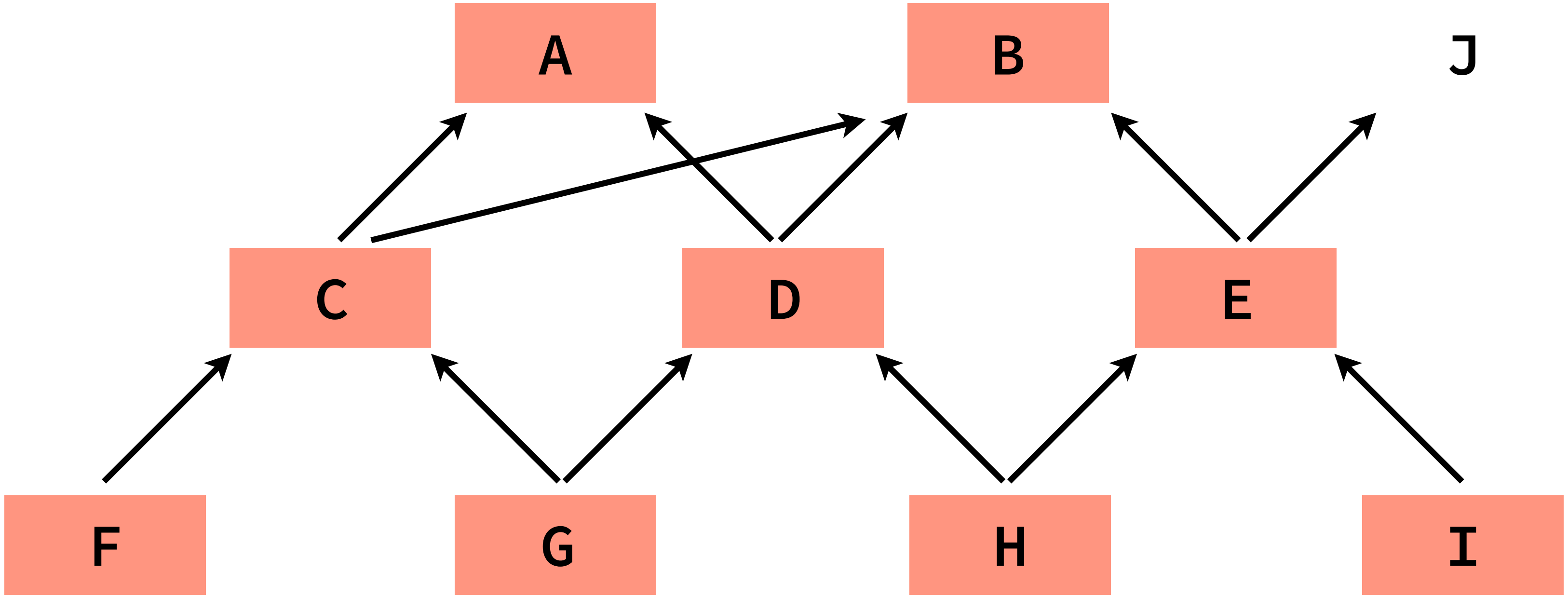
`x instanceof B ⇔ x.tid ∈ [2,9]`

Note: a single slice is sufficient for this hierarchy.

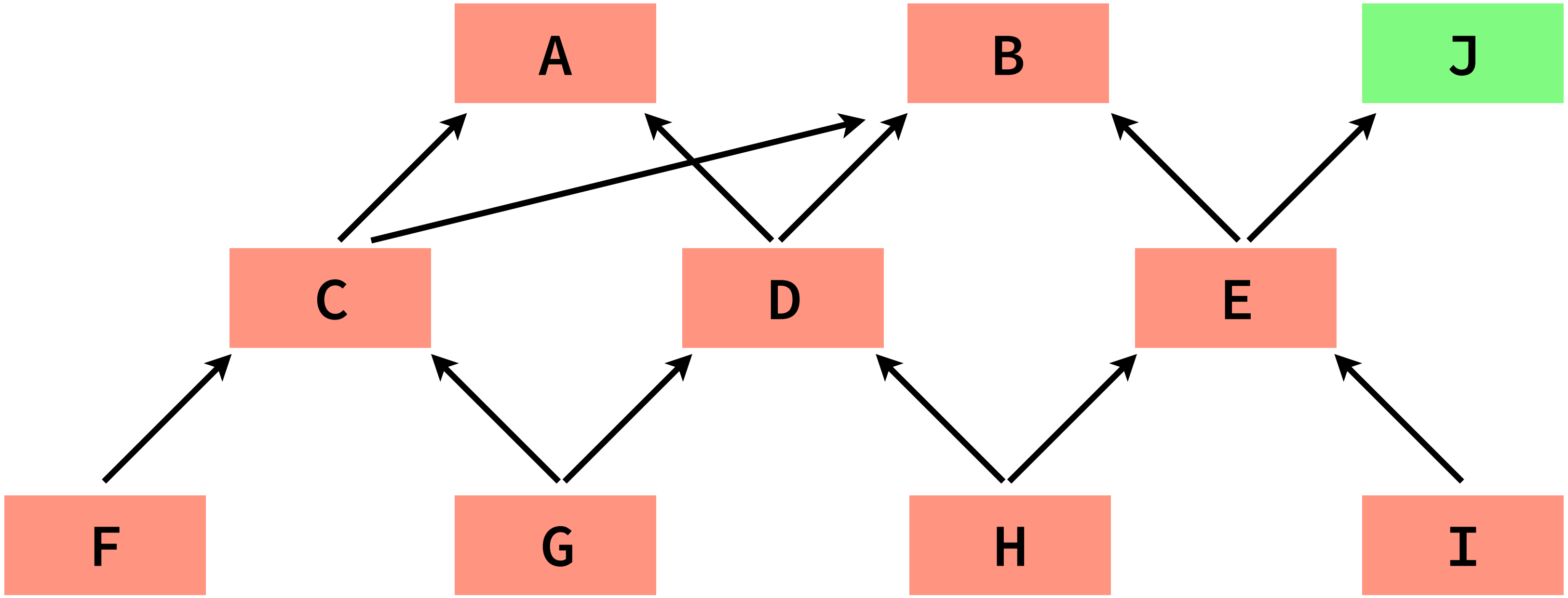
PQ encoding example 2



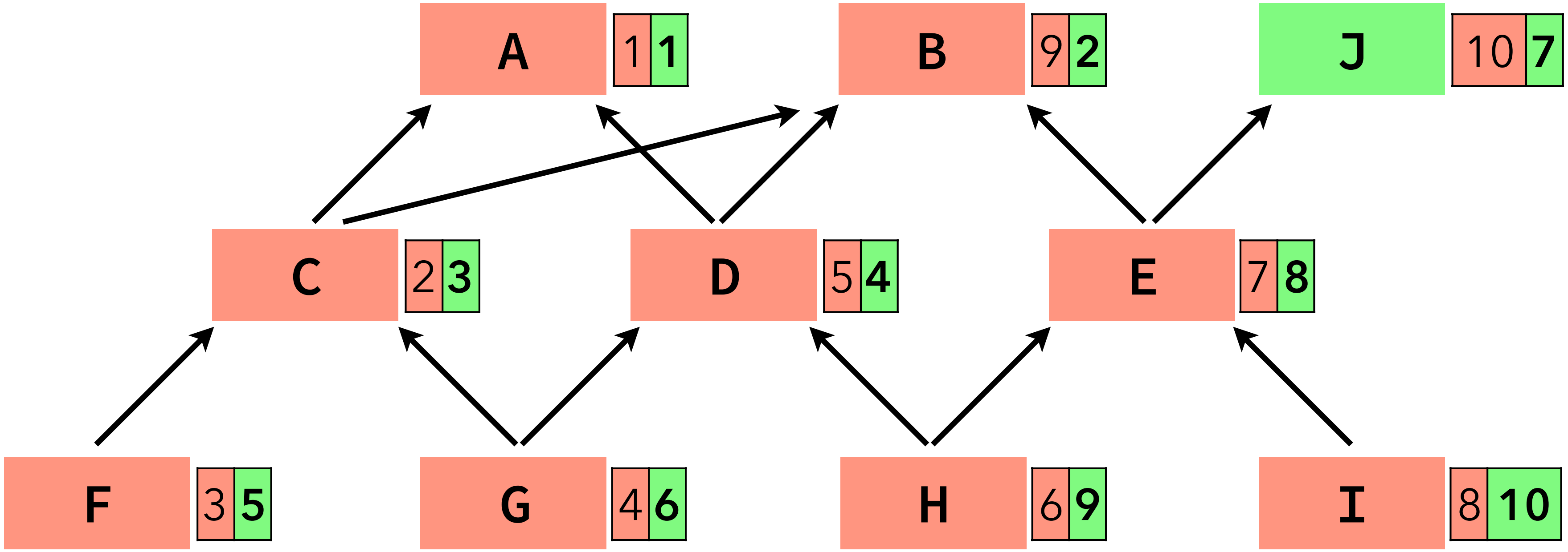
PQ encoding example 2



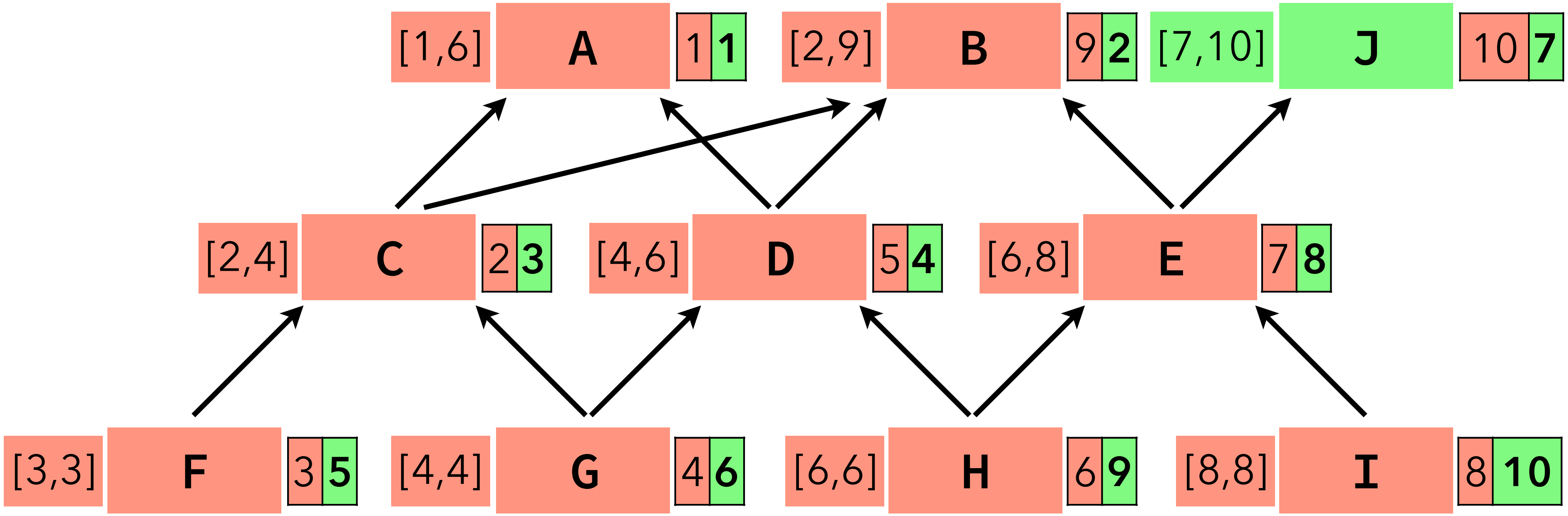
PQ encoding example 2



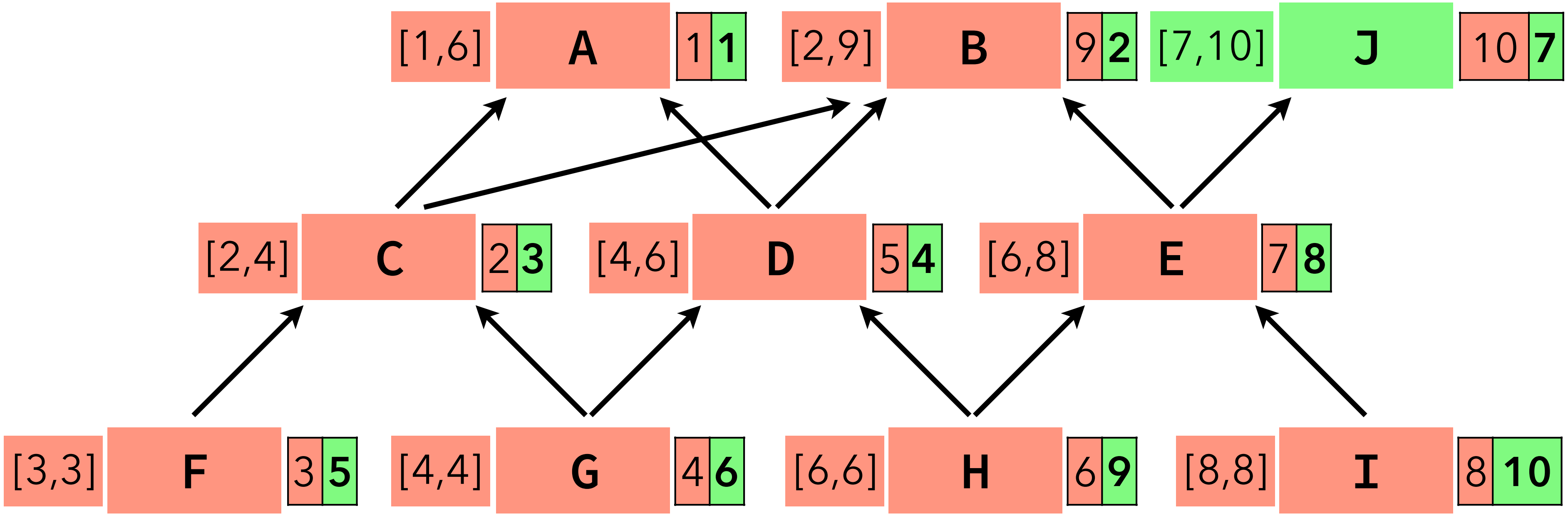
PQ encoding example 2



PQ encoding example 2



PQ encoding example 2



x **instanceof** B ⇔ x.tid[0] ∈ [2,9]

x **instanceof** J ⇔ x.tid[1] ∈ [7,10]

Hybrid techniques

Like for the dispatch problem, it is perfectly possible to combine several solutions to the membership test problem. For example, a Java implementation could use Cohen’s encoding to handle membership tests for classes, and PQ encoding for interfaces.

Membership test summary

In a single subtyping context, two simple solutions to the membership test exist: relative numbering and Cohen's encoding. Only the latter is incremental, in that it supports the incremental addition of types at the bottom of the hierarchy.

Generalizations of these techniques exist for multiple subtyping contexts: range compression, packed and PQ encoding. Those techniques enable the membership test to be solved efficiently, but the building of the supporting data structures is relatively complicated. Moreover, they require the whole hierarchy to be available, and are not incremental.