Object-Oriented Programming

CSCI-UA 0470-001
Class 24
Instructor: Randy Shepherd

Definition Inheritance

Implementation Inheritance

- James Gosling, the inventor of Java, was once asked “If you could do Java over again, what would you change?”, he replied, ”I’d leave out classes.”

- He went on to explain that the real problem wasn't classes per se, but rather implementation inheritance (the *extends* relationship).

- Everything we have talked w.r.t. inheritance in this course thus far relates to implementation inheritance (aka class inheritance).

Disadvantages of Implementation Inheritance

- A few lectures ago we noted a few problems with this type of inheritance…
  - E*ncapsulation is broken*; subclass exposed to implementation details of superclass.
  - Tight coupling between super and subclasses.
  - Only good for true ‘is-a’ relationships.
  - False hierarchies common.

Implementation Vs Definition Inheritance

- All of those disadvantages can be understood as problems with inheritance of behavior and state.
- “Definition inheritance” implies that we only inherit class contracts, no state or behavior.
- In Java, definition inheritance comes in the form of ‘interfaces’.

What are Interfaces?

- Objects define their interaction with the outside world through the methods that they expose.
    - Moreover, methods form the object's interface with the outside world.
- An *interface* is a formalization of this concept as a language primitive.
    - Moreover, a Java *interface* is group of related methods with empty bodies and *static* variables*

* not the whole story, but we’ll get there

What are Interfaces?

- Implementing an interface allows a class to be explicit about the behavior it promises to provide, its *contract*.
- This contract is enforced by the compiler.
- If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will compile.

Interface Constraints

- Interfaces can only define method *signatures*.
- Interfaces can only define public static fields.
- Interfaces cannot have constructors.
- A class that implements an interface must implement *all* the methods defined in the interface.

Interface Example

```java
public interface Bicycle {
  //  wheel revolutions per minute
  void changeCadence(int newValue);

  void changeGear(int newValue);

  void speedUp(int increment);

  void applyBrakes(int decrement);
}
```

```java
public class ACMEBicycle implements Bicycle {
  private int cadence = 0;
  private int speed = 0;
  private int gear = 1;

 // The compiler will now require that methods
 // changeCadence, changeGear, speedUp, and applyBrakes
 // all be implemented. Compilation will fail if those
 // methods are missing from this class.

  void changeCadence(int newValue) {
       cadence = newValue;
  }

  void changeGear(int newValue) {
       gear = newValue;
  }

  void speedUp(int increment) {
       speed = speed + increment;
  }

  void applyBrakes(int decrement) {
       speed = speed - decrement;
  }
}
```

Adapted from The Java™ Tutorials

Abstract Classes

- But wait.. isn’t this just a ‘pure’ abstract class?
- In fact, abstract classes sound better..
  - Abstract classes can have method implementations.
  - Abstract classes can have instance variables.
  - Abstract classes can have constructors.

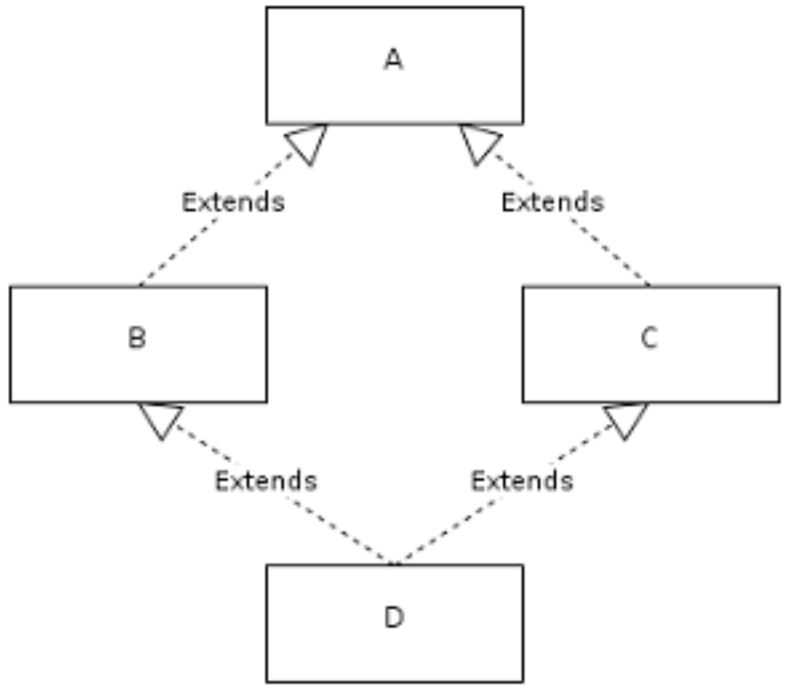
Abstract Classes

- Interfaces provide a significant feature that Abstract Classes do not.
- A class can *implement* many interfaces but can have *extend* one superclass.
- Java does not support multiple inheritance for classes!
- Why?

Multiple Inheritance

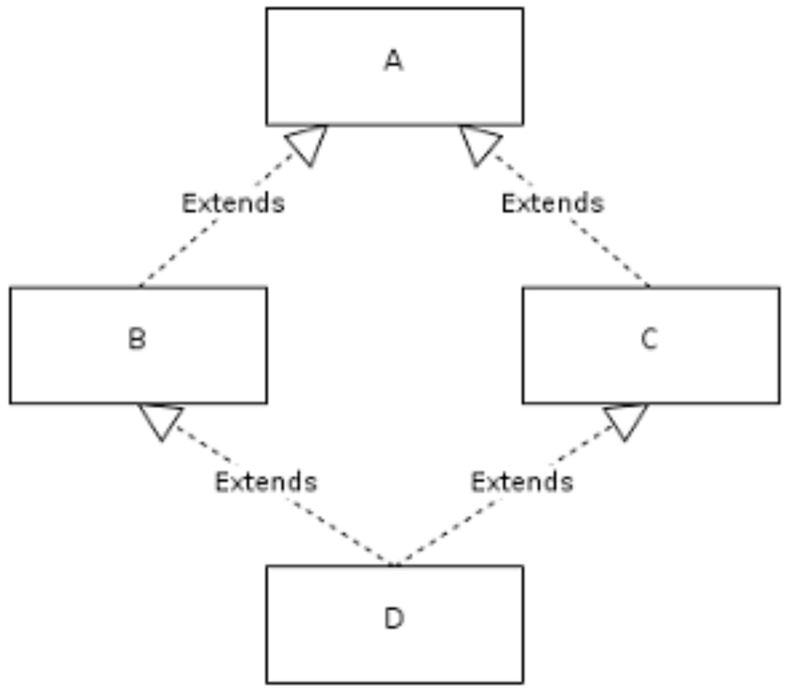
The Diamond Problem

- The "diamond problem" is an ambiguity that arises in situations like the following:
  - Two classes B and C inherit from A
  - Class D inherits from both B and C
  - There is a method in A that B and C has overridden, but D does not override it
  - Which version of the method does D inherit: that of B, or that of C?



The Diamond Problem

- There is some difficulty for the language implementer as well.
- What does the data layout of B look like? A's variables + B's variables.
- What does the data layout of C look like? A's variables + C's variables.
- What does D's data layout look like?



C++ & Multiple Inheritance

- Consider the relatively simple case of multiple inheritance.
- Note that Top is inherited from *twice* from Bottom via Left and Right.
- This means that an object of type Bottom will have two attributes called ‘a’.

```cpp
class Top {
public:
    int a;
};

class Left : public Top {
public:
    int b;
};

class Right : public Top {
public:
    int c;
};

class Bottom : public Left, public Right {
public:
    int d;
};
```

C++ & Multiple Inheritance

- How are Left, Right and Botton laid out in memory?

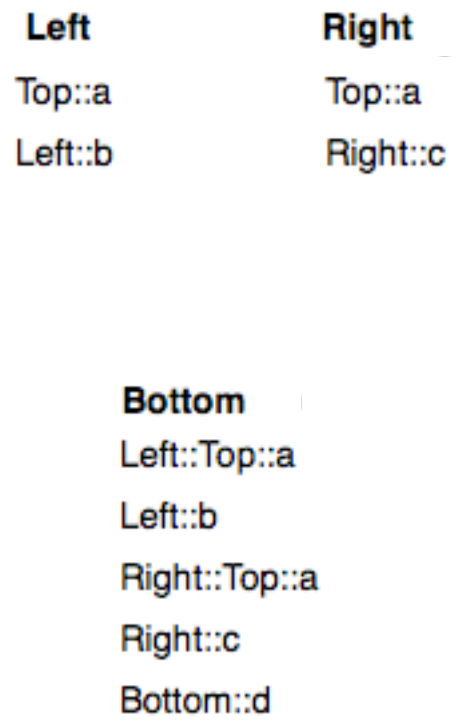
- If we take the approach we are familiar, we just ‘stack’ subclass data below superclass data.

Left	**Right**
Top::a	Top::a
Left::b	Right::c

Bottom
Left::Top::a
Left::b
Right::Top::a
Right::c
Bottom::d

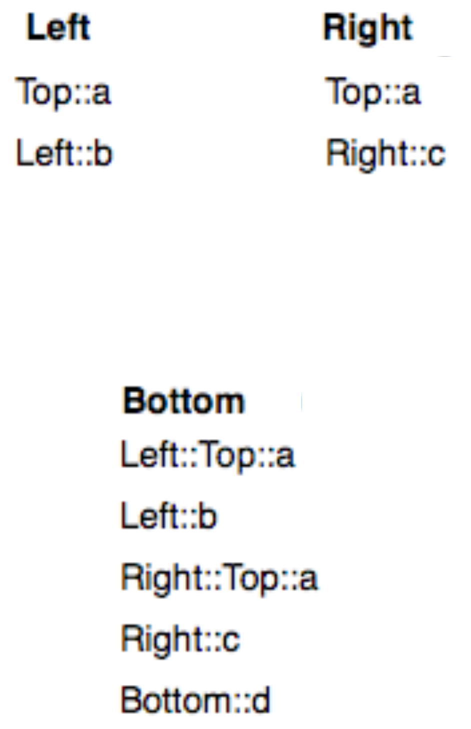
C++ & Multiple Inheritance

- Now what happens when we upcast a Bottom pointer?
  - Ex. Left* left = bottom;
- No problem.
- We can treat an object of type Bottom as if it were an object of type Left, because the memory layout of both classes agree.



C++ & Multiple Inheritance

- However, what happens when we upcast to Right?
  - Ex. Right* right = bottom;



Left	**Right**
Top::a	Top::a
Left::b	Right::c

Bottom
Left::Top::a
Left::b
Right::Top::a
Right::c
Bottom::d

C++ & Multiple Inheritance

- However, what happens when we upcast to Right?
  - Ex. Right* right = bottom;
- Uh-oh. The memory layouts are different!
- We cannot use an instance of a Bottom as an instance of a Right, polymorphism is broken!

Left	**Right**
Top::a	Top::a
Left::b	Right::c

Bottom
Left::Top::a
Left::b
Right::Top::a
Right::c
Bottom::d

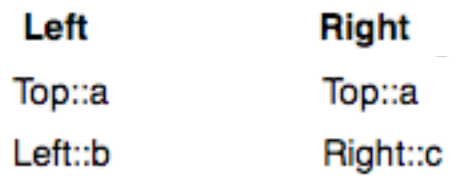
C++ & Multiple Inheritance

- For this to work, we have to adjust the pointer value during assignment to make it point to the corresponding section of the Bottom layout.

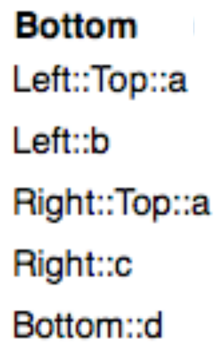
- So when we do this..

  - Right* right = bottom;

- …right points to the right section in the data layout.



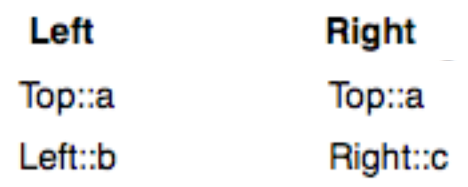
Left	Right
Top::a	Top::a
Left::b	Right::c

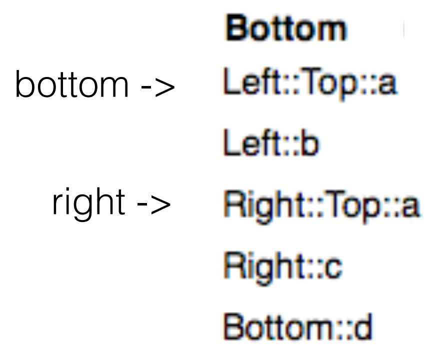


	Bottom
	Left::Top::a
	Left::b
right ->	Right::Top::a
	Right::c
	Bottom::d

C++ & Multiple Inheritance

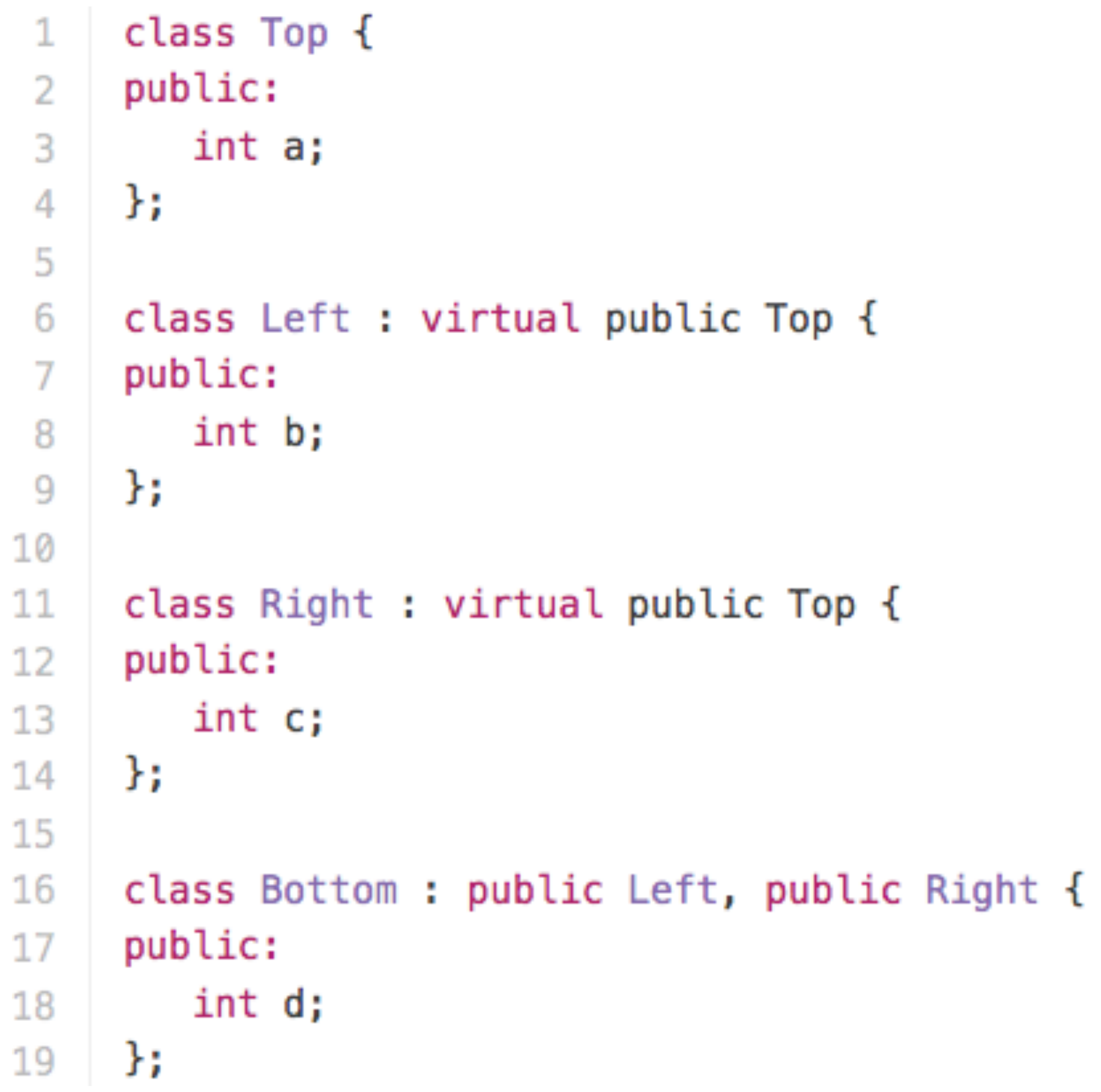
- After this adjustment, we can access bottom through the right pointer as a normal Right object.
- However, bottom and right now point to different memory locations!!
- Also, and more fundamentally, we have two ‘a’s!!





Virtual Inheritance

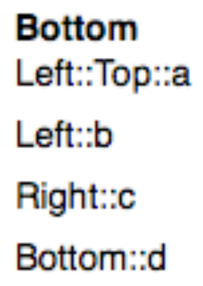
- As a programmer, to avoid the repeated inheritance of Top, we inherit *virtually* from Top.
- Using the keyword ‘virtual’ at lines 6 and lines 11 prevent the duplication of Top’s data.
- While this may seem more obvious and simpler from a programmer's point of view, from the compiler's point of view this is vastly more complicated.



```
 1  class Top {
 2  public:
 3      int a;
 4  };
 5
 6  class Left : virtual public Top {
 7  public:
 8      int b;
 9  };
10
11  class Right : virtual public Top {
12  public:
13      int c;
14  };
15
16  class Bottom : public Left, public Right {
17  public:
18      int d;
19  };
```

Virtual Inheritance

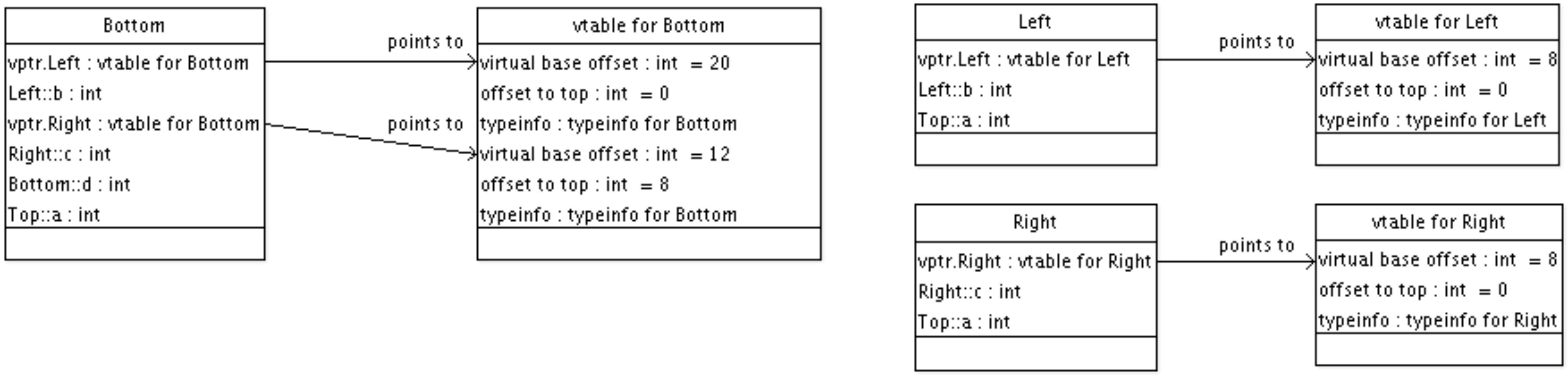
- Given the virtual keyword, consider a possible layout of Bottom:



Bottom
Left::Top::a
Left::b
Right::c
Bottom::d

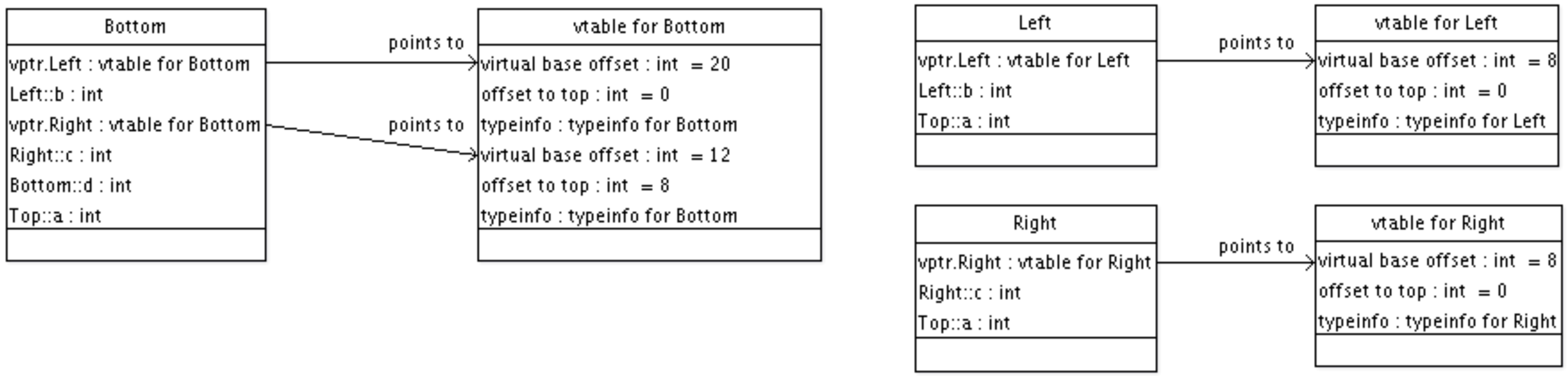
- However, what happens when we upcast to Right?
- The same problem. The memory layout of Right itself is completely different.
- Moreover, no simple layout for Bottom will work.

Virtual Inheritance


Bottom
vptr.Left : vtable for Bottom
Left::b : int
vptr.Right : vtable for Bottom
Right::c : int
Bottom::d : int
Top::a : int
points to
points to
vtable for Bottom
virtual base offset : int = 20
offset to top : int = 0
typeinfo : typeinfo for Bottom
virtual base offset : int = 12
offset to top : int = 8
typeinfo : typeinfo for Bottom
Left
vptr.Left : vtable for Left
Left::b : int
Top::a : int
points to
vtable for Left
virtual base offset : int = 8
offset to top : int = 0
typeinfo : typeinfo for Left
Right
vptr.Right : vtable for Right
Right::c : int
Top::a : int
points to
vtable for Right
virtual base offset : int = 8
offset to top : int = 0
typeinfo : typeinfo for Right

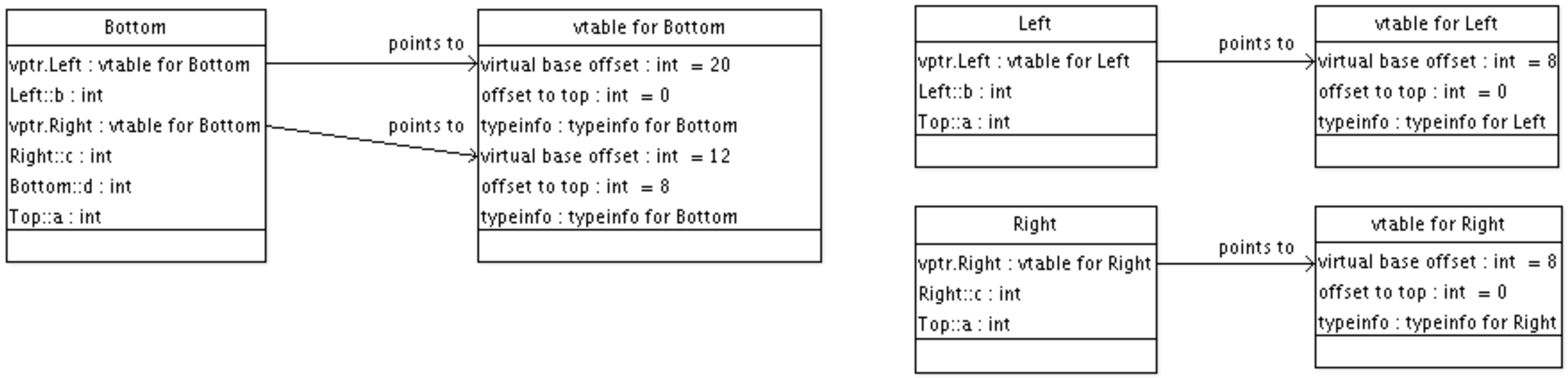
- The solution is non-trivial.
- Note two things about these new data layouts:
  - The order of the fields is reversed.
  - There are new pointers, ex vptr.Left

Virtual Inheritance


Bottom
vptr.Left : vtable for Bottom
Left::b : int
vptr.Right : vtable for Bottom
Right::c : int
Bottom::d : int
Top::a : int
points to
points to
vtable for Bottom
virtual base offset : int = 20
offset to top : int = 0
typeinfo : typeinfo for Bottom
virtual base offset : int = 12
offset to top : int = 8
typeinfo : typeinfo for Bottom
Left
vptr.Left : vtable for Left
Left::b : int
Top::a : int
points to
vtable for Left
virtual base offset : int = 8
offset to top : int = 0
typeinfo : typeinfo for Left
Right
vptr.Right : vtable for Right
Right::c : int
Top::a : int
points to
vtable for Right
virtual base offset : int = 8
offset to top : int = 0
typeinfo : typeinfo for Right

- The vptrs index new vtables specific to this purpose.
- There is a vptr for every *virtual* base class.
- The compiler inserts these when virtual inheritance is used.

Virtual Inheritance

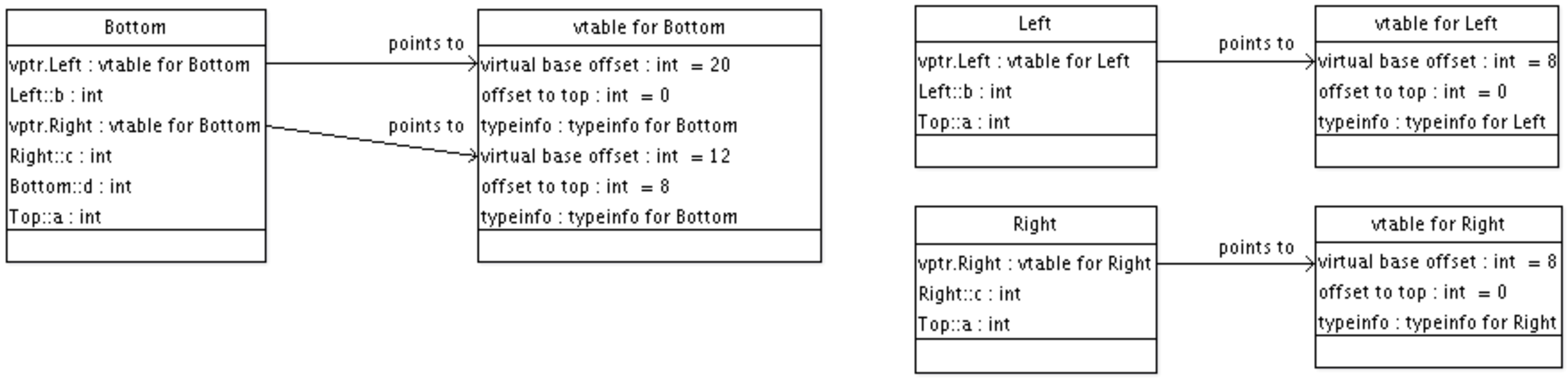

Bottom
vptr.Left : vtable for Bottom
Left::b : int
vptr.Right : vtable for Bottom
Right::c : int
Bottom::d : int
Top::a : int
points to
points to
vtable for Bottom
virtual base offset : int = 20
offset to top : int = 0
typeinfo : typeinfo for Bottom
virtual base offset : int = 12
offset to top : int = 8
typeinfo : typeinfo for Bottom
Left
vptr.Left : vtable for Left
Left::b : int
Top::a : int
points to
vtable for Left
virtual base offset : int = 8
offset to top : int = 0
typeinfo : typeinfo for Left
Right
vptr.Right : vtable for Right
Right::c : int
Top::a : int
points to
vtable for Right
virtual base offset : int = 8
offset to top : int = 0
typeinfo : typeinfo for Right

- Consider the following C++ code.

```
Bottom* bottom = new Bottom();
Left* left = bottom;
int p = left->a;
```

- The second line makes left point to the same address as bottom (i.e., it points to the “top” of the Bottom object)

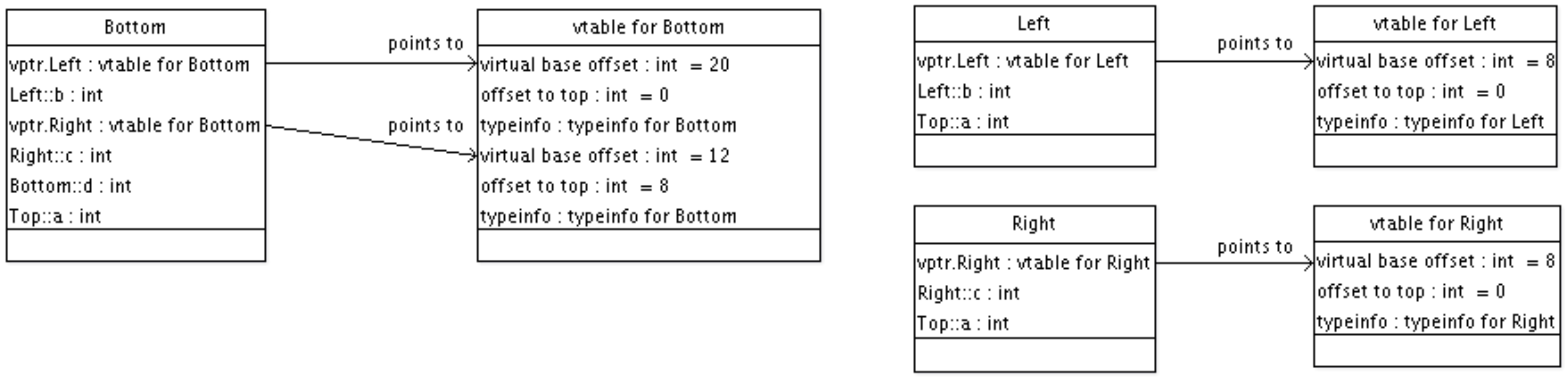
Virtual Inheritance


Bottom
vptr.Left : vtable for Bottom
Left::b : int
vptr.Right : vtable for Bottom
Right::c : int
Bottom::d : int
Top::a : int
points to
points to
vtable for Bottom
virtual base offset : int = 20
offset to top : int = 0
typeinfo : typeinfo for Bottom
virtual base offset : int = 12
offset to top : int = 8
typeinfo : typeinfo for Bottom
Left
vptr.Left : vtable for Left
Left::b : int
Top::a : int
points to
vtable for Left
virtual base offset : int = 8
offset to top : int = 0
typeinfo : typeinfo for Left
Right
vptr.Right : vtable for Right
Right::c : int
Top::a : int
points to
vtable for Right
virtual base offset : int = 8
offset to top : int = 0
typeinfo : typeinfo for Right

- We use left to index the virtual table and obtain the “virtual base offset” (vbase).

- This offset is then added to left, which is then used to index the Top section of the Bottom object.

Virtual Inheritance

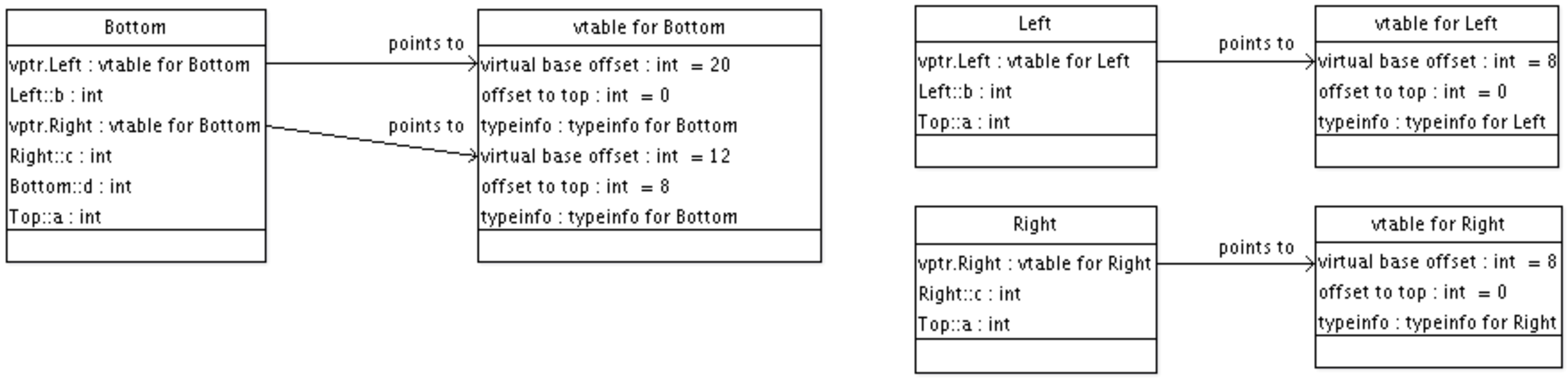

Bottom
vptr.Left : vtable for Bottom
Left::b : int
vptr.Right : vtable for Bottom
Right::c : int
Bottom::d : int
Top::a : int
points to
points to
vtable for Bottom
virtual base offset : int = 20
offset to top : int = 0
typeinfo : typeinfo for Bottom
virtual base offset : int = 12
offset to top : int = 8
typeinfo : typeinfo for Bottom
Left
vptr.Left : vtable for Left
Left::b : int
Top::a : int
points to
vtable for Left
virtual base offset : int = 8
offset to top : int = 0
typeinfo : typeinfo for Left
Right
vptr.Right : vtable for Right
Right::c : int
Top::a : int
points to
vtable for Right
virtual base offset : int = 8
offset to top : int = 0
typeinfo : typeinfo for Right

- This works for the following cast as well.

```
Bottom* bottom = new Bottom();
Right* right = bottom;
int p = right->a;
```

- ‘right’ will now point to the vptr for Right (3rd entry).

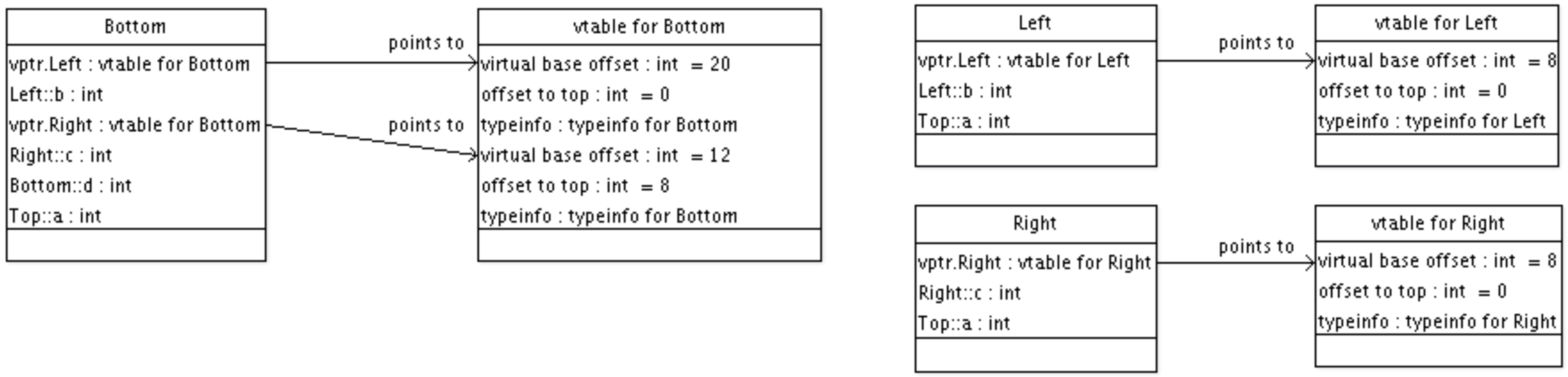
Virtual Inheritance


Bottom
vptr.Left : vtable for Bottom
Left::b : int
vptr.Right : vtable for Bottom
Right::c : int
Bottom::d : int
Top::a : int
points to
points to
vtable for Bottom
virtual base offset : int = 20
offset to top : int = 0
typeinfo : typeinfo for Bottom
virtual base offset : int = 12
offset to top : int = 8
typeinfo : typeinfo for Bottom
Left
vptr.Left : vtable for Left
Left::b : int
Top::a : int
points to
vtable for Left
virtual base offset : int = 8
offset to top : int = 0
typeinfo : typeinfo for Left
Right
vptr.Right : vtable for Right
Right::c : int
Top::a : int
points to
vtable for Right
virtual base offset : int = 8
offset to top : int = 0
typeinfo : typeinfo for Right

- Like with left, the offset is fetched from the table and then added to right, which is then used to index the Top section of the Bottom object.

- When we compare the ‘right’ and ‘bottom’ pointers the ‘offset to top’ is added to ‘right’ before the comparison is performed. This way pointer comparisons still give the correct result.

Virtual Inheritance


Bottom
vptr.Left : vtable for Bottom
Left::b : int
vptr.Right : vtable for Bottom
Right::c : int
Bottom::d : int
Top::a : int
points to
points to
vtable for Bottom
virtual base offset : int = 20
offset to top : int = 0
typeinfo : typeinfo for Bottom
virtual base offset : int = 12
offset to top : int = 8
typeinfo : typeinfo for Bottom
Left
vptr.Left : vtable for Left
Left::b : int
Top::a : int
points to
vtable for Left
virtual base offset : int = 8
offset to top : int = 0
typeinfo : typeinfo for Left
Right
vptr.Right : vtable for Right
Right::c : int
Top::a : int
points to
vtable for Right
virtual base offset : int = 8
offset to top : int = 0
typeinfo : typeinfo for Right

- Of course, the point of the exercise was to be able to access Right objects the same way as upcasted Bottom objects.
- Therefore, we have to introduce vptrs in the layout of Right (and Left) too.
- Now we can access a Bottom object through a Right and Left pointers without further difficulty.

Virtual Inheritance

- Ok we have multiple inheritance. However, it has come at rather large expense.
- We needed to…
  - introduce new virtual tables
  - classes needed to be extended with >=1 virtual pointers
  - simple lookups in an object now needs two indirections
- All this for a relatively rarely-used feature that has a lot of complexity for the end-user.

Java & Multiple Inheritance

- Simple answer, don't do it.*
- Multiple inheritance is only allowed for interfaces.
- Interfaces disallow instance variables, so data layout issues a non-issue.
- Much of the complexity we just saw disappears for the compiler author and programmer.

* not the whole story, but we'll get there

Java Versus C++

"Java omits many rarely used, poorly understood, confusing features of C++ that in our experience bring more grief than benefit. This primarily consists of operator overloading, multiple inheritance, and extensive automatic coercions."

- James Gosling *"Java: an Overview"* 1995

Abstract Classes Vs Interfaces

Adapted from Effective Java by Joshua Bloch

Differences

- As we've seen Java provides two mechanisms for defining a type that permits multiple implementations: interfaces and abstract classes.

- The most obvious difference between the two mechanisms is that abstract classes are permitted to contain implementations for some methods while interfaces are not.

- A more important difference is that to implement the type defined by an abstract class, a class *must be a subclass* of the abstract class.

Differences

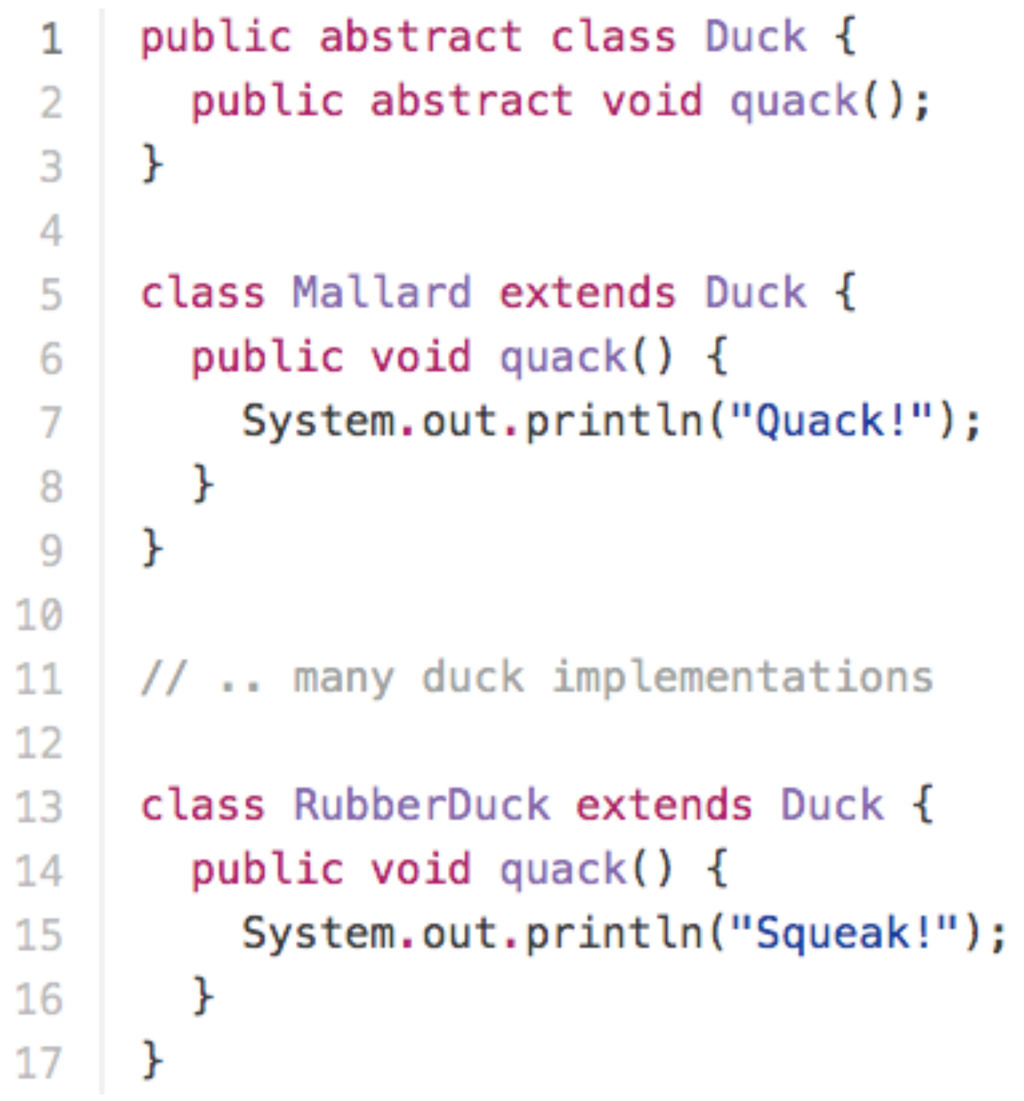
- Any class that defines all of the required methods is permitted to implement an interface, *regardless of where the class resides in the class hierarchy*.
- Because Java permits only single inheritance with classes, abstract classes are severely constrained in their usefulness.

Advantages of Interfaces: Retrofitting

- Existing classes can be easily retrofitted to implement a new interface.
  - Simply add the required methods and an implements clause to the class declaration.
  - In order for two classes extend the same abstract class, you must place the abstract class high up in the type hierarchy where it subclasses an ancestor of both classes.
  - This causes great collateral damage to the type hierarchy, forcing all descendants of the common ancestor to extend the new abstract class whether or not it is appropriate for them to do so.

Advantages of Interfaces: Retrofitting

- Suppose I have this class representing a duck.
- All ducks quack so no problem to have it as part of an abstract class.
- But what if I want to have ducks fly?



```java
public abstract class Duck {
  public abstract void quack();
}

class Mallard extends Duck {
  public void quack() {
    System.out.println("Quack!");
  }
}

// .. many duck implementations

class RubberDuck extends Duck {
  public void quack() {
    System.out.println("Squeak!");
  }
}
```

Advantages of Interfaces: Retrofitting

- Not all ducks fly so adding to the abstract class doesn't work.
- Perfect use case for an interface, simple to retrofit.
- Note that the Flyable could be used in things other than ducks!

```java
public interface Flyable {
  public void fly();
}

public abstract class Duck {
  public abstract void quack();
}

class Mallard extends Duck implements Flyable {
  public void quack() {
    System.out.println("Quack!");
  }

  public void fly() {
    System.out.println("Headed south!");
  }
}

// .. many duck implementations

class RubberDuck extends Duck {
  public void quack() {
    System.out.println("Squeak!");
  }
}
```

Advantages of Interfaces: Mixins

- Interfaces are ideal for defining ‘mixins’.
  - a ‘mixin’ is a type that a class can implement in addition to its “primary type” to declare that it provides some optional behavior. (like we just saw)
  - For example, Comparable is a ‘mixin’ interface that allows a class to declare that its instances are ordered with respect to other instances.
  - Abstract classes can’t be used to define mixins for the same reason that they can’t be retrofitted onto existing classes.

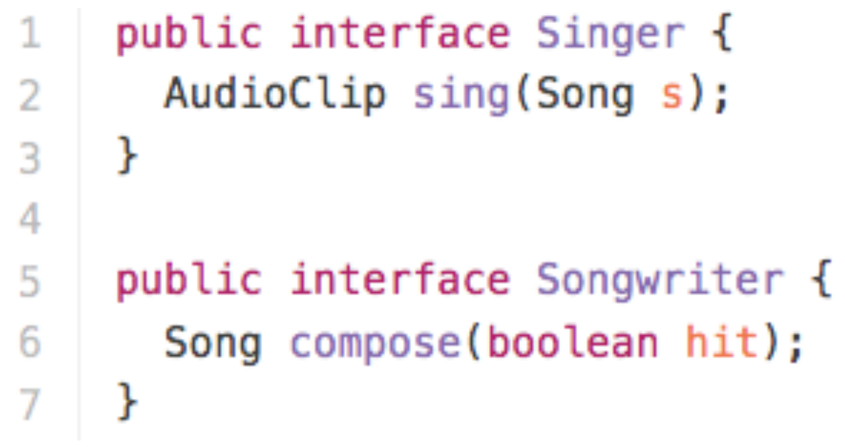
Advantages of Interfaces: Nonhierarchical Types

- Interfaces allow the construction of nonhierarchical type frameworks.
  - Type hierarchies are great for organizing some things, but other things don’t fall neatly into a rigid hierarchy.

Advantages of Interfaces: Nonhierarchical Types

- Suppose we have an interface representing a ‘singer’ and another representing a ‘songwriter’.

- Some singers are also songwriters.

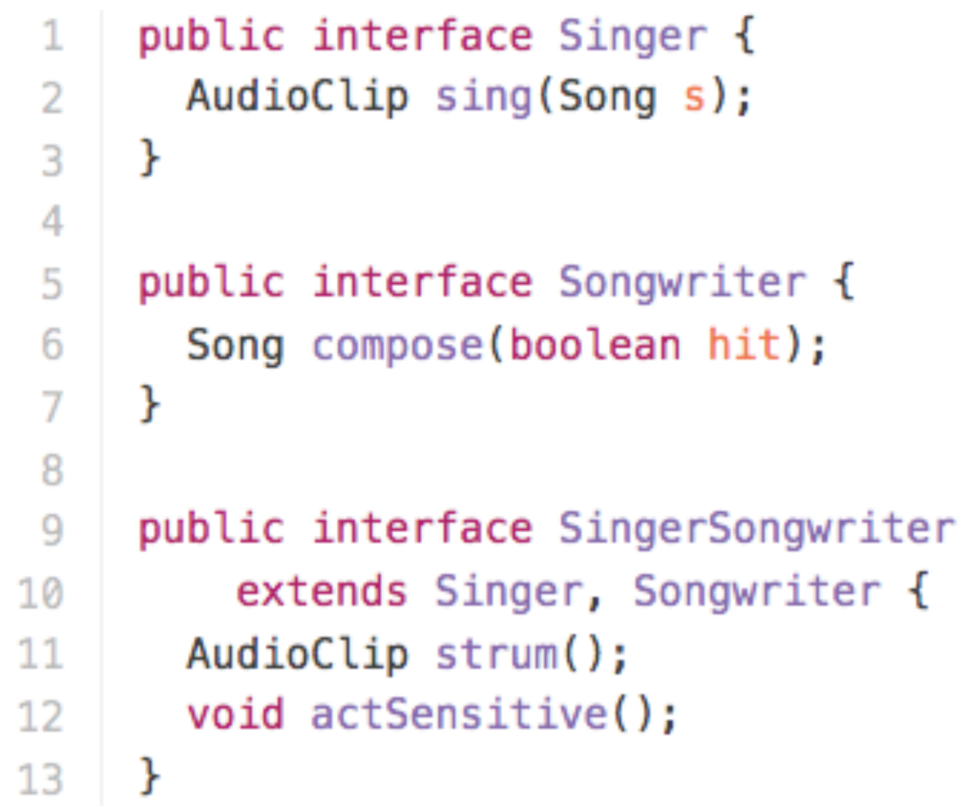


```java
public interface Singer {
  AudioClip sing(Song s);
}

public interface Songwriter {
  Song compose(boolean hit);
}
```

Advantages of Interfaces: Nonhierarchical Types

- Because we used interfaces, it is permissible for a single class to implement both Singer and Songwriter.

- In fact, we can define a third interface that extends both Singer and Songwriter and adds new methods that are appropriate to the combination.



```java
public interface Singer {
  AudioClip sing(Song s);
}

public interface Songwriter {
  Song compose(boolean hit);
}

public interface SingerSongwriter
    extends Singer, Songwriter {
  AudioClip strum();
  void actSensitive();
}
```

Advantages of Interfaces: Nonhierarchical Types

- You don’t always need this level of flexibility, but when you do, interfaces are a lifesaver.
- The alternative is a bloated class hierarchy containing a separate class for every supported combination of attributes.
- If there are n attributes in the type system, there are ~2^n possible combinations that you may have to support.

Implementation Support

- Using interfaces to define types does not prevent you from providing implementation assistance to programmers using your interfaces.
- You can combine the virtues of interfaces and abstract classes by providing an abstract *skeletal implementation* class to go with each nontrivial interface.
- The interface still defines the type, but the skeletal implementation takes all of the work out of implementing it.

* not the whole story, but we'll get there

Skeletal Implementations

- By convention, skeletal implementations are called Abstract*Interface*, where *Interface* is the name of the interface they provide implementation for.

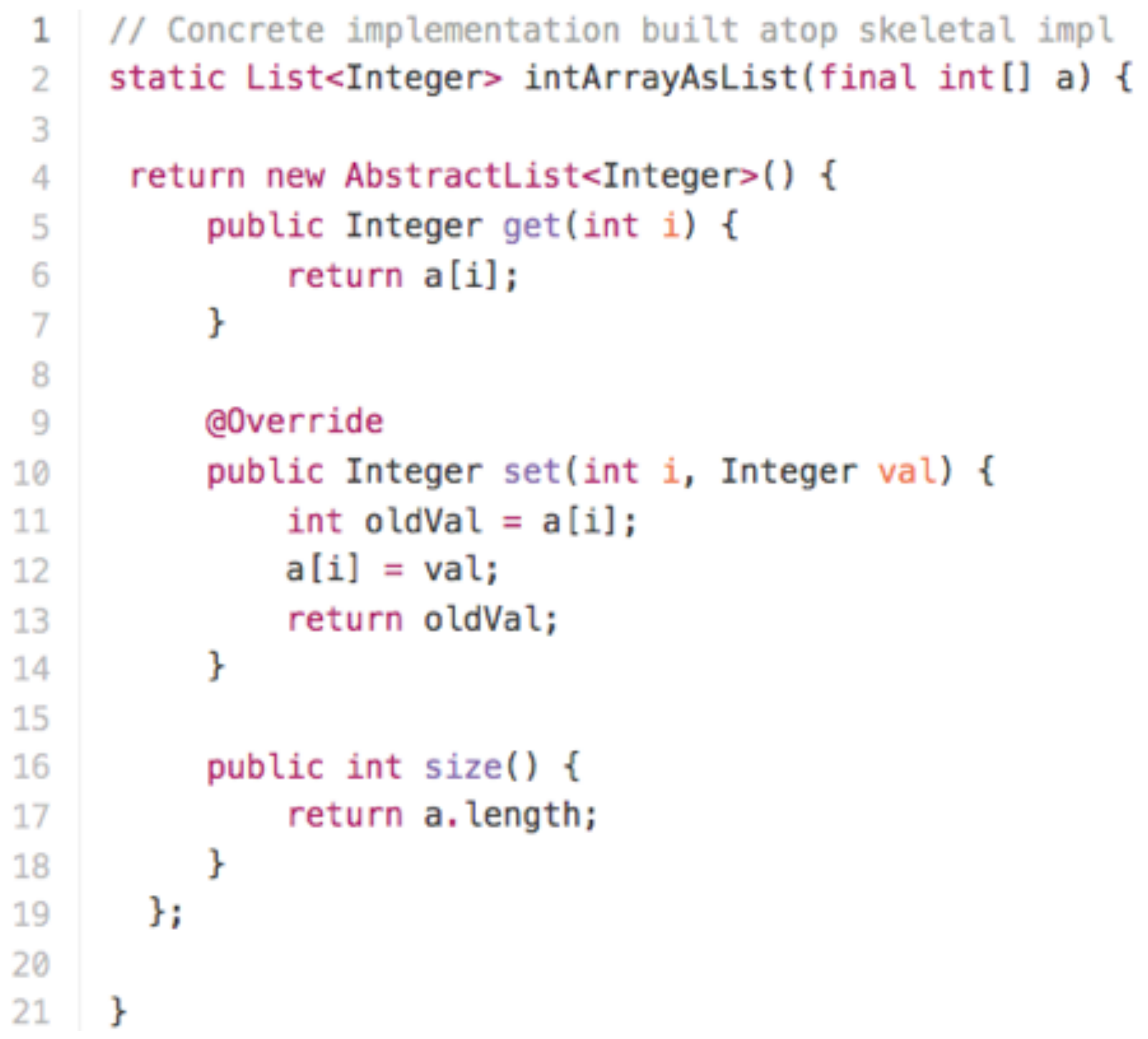
- For example, the Collections Framework provides a skeletal implementation to go along with each main collection interface: AbstractCollection, AbstractSet, AbstractList, and AbstractMap.

- These implementations take care of the ‘grunt’ work for the data structure. Logic that is consistent across all implementations of the abstract data type.

Skeletal Implementation Example

- When properly designed, skeletal implementations can make it very easy for programmers to provide their own implementations of your interfaces.

- For example, here's a static factory method containing a complete, fully functional List implementation using Java's AbstractList skeleton.

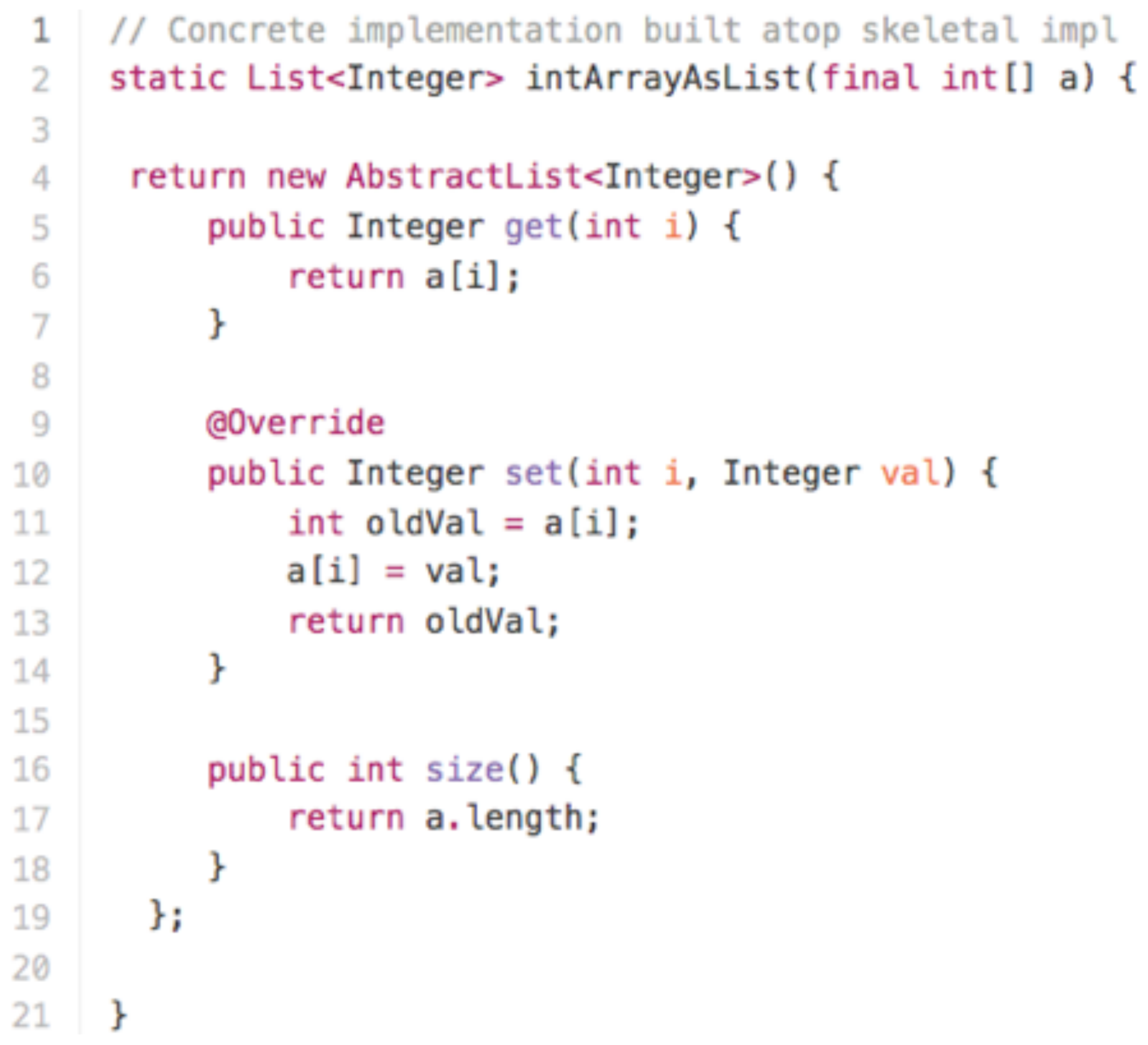


```java
// Concrete implementation built atop skeletal impl
static List<Integer> intArrayAsList(final int[] a) {

 return new AbstractList<Integer>() {
     public Integer get(int i) {
         return a[i];
     }

     @Override
     public Integer set(int i, Integer val) {
         int oldVal = a[i];
         a[i] = val;
         return oldVal;
     }

     public int size() {
         return a.length;
     }
  };

}
```

Skeletal Implementation Example

- When you consider all that a List does for you, this is an impressive demo of the power of skeletal implementations.

- Also, this is an example of the ‘*Adapter pattern*’ since it allows an int array to be viewed as a list of Integers.



```
// Concrete implementation built atop skeletal impl
static List<Integer> intArrayAsList(final int[] a) {

 return new AbstractList<Integer>() {
     public Integer get(int i) {
         return a[i];
     }

     @Override
     public Integer set(int i, Integer val) {
         int oldVal = a[i];
         a[i] = val;
         return oldVal;
     }

     public int size() {
         return a.length;
     }
  };

}
```

Skeletal Implementations

- Skeletal implementations provide the implementation assistance of abstract classes without imposing the constraints that abstract classes impose.

- For most implementors of an interface, extending the skeletal implementation is the obvious choice, but it is strictly optional.

- If a preexisting class cannot be made to extend the skeletal implementation, the class can always implement the interface manually.

Advantages of Abstract Classes: State

- If sharing state is important, then you must use abstract classes.
    - If a crucial part of your class is its data and that data needs to be shared with subclasses, then interfaces will not work.
    - Interfaces cannot have any instance variables!

Advantages of Abstract Classes: Class Evolution

- It is easier to evolve an abstract class than an interface.
  - If you want to add a new method to an abstract class, you can always add a concrete method containing a reasonable default implementation.
  - Adding a method to an interface will break all existing classes that implement the interface. (Classes that previously implemented the interface will be missing the new method and won't compile.)*

* not the whole story, but we'll get there

Java 8

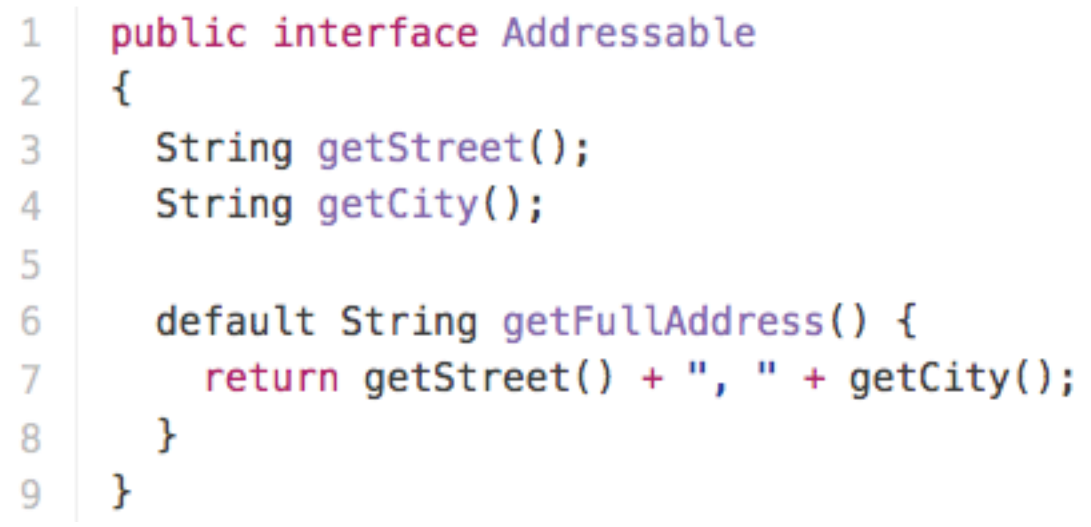
- Remember I said ‘interfaces only contain method declarations, no implementations!”. Thats not actually entirely true, at least anymore.
- Java 8, released in 2015, introduced many new features including ‘default’ and static methods for interfaces.

Default Methods

- A default method is an *instance* method defined in an interface whose method header begins with the default keyword

- It also provides an implementation.

- Every class that implements the interface inherits the interface's default methods and *can override them*.



```java
public interface Addressable
{
  String getStreet();
  String getCity();

  default String getFullAddress() {
    return getStreet() + ", " + getCity();
  }
}
```

Default Methods

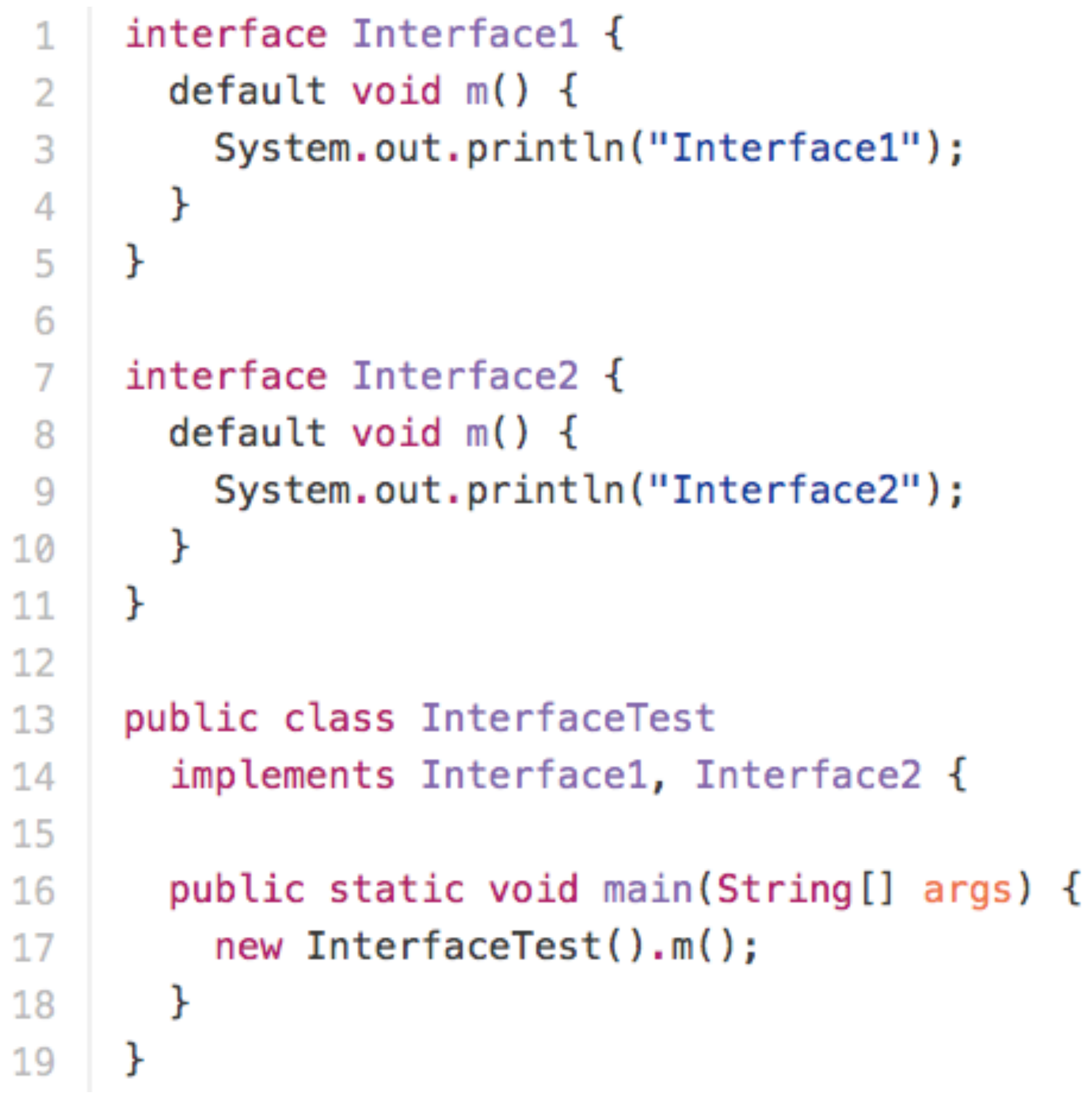
- So what happens in this case?

```java
interface Interface1 {
  default void m() {
    System.out.println("Interface1");
  }
}

interface Interface2 {
  default void m() {
    System.out.println("Interface2");
  }
}

public class InterfaceTest
  implements Interface1, Interface2 {

  public static void main(String[] args) {
    new InterfaceTest().m();
  }
}
```

Default Methods

- So what happens in this case?

- Compile error!



```java
interface Interface1 {
  default void m() {
    System.out.println("Interface1");
  }
}

interface Interface2 {
  default void m() {
    System.out.println("Interface2");
  }
}

public class InterfaceTest
  implements Interface1, Interface2 {

  public static void main(String[] args) {
    new InterfaceTest().m();
  }
}
```

Default Method Use Cases

- Evolving existing interfaces…
  - As we talked about earlier, one problem with interfaces has historically been evolving them and breaking existing implementations.
  - Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.
  - With default methods, a major drawback of interfaces is now eliminated, as they behave like abstract classes in this case.

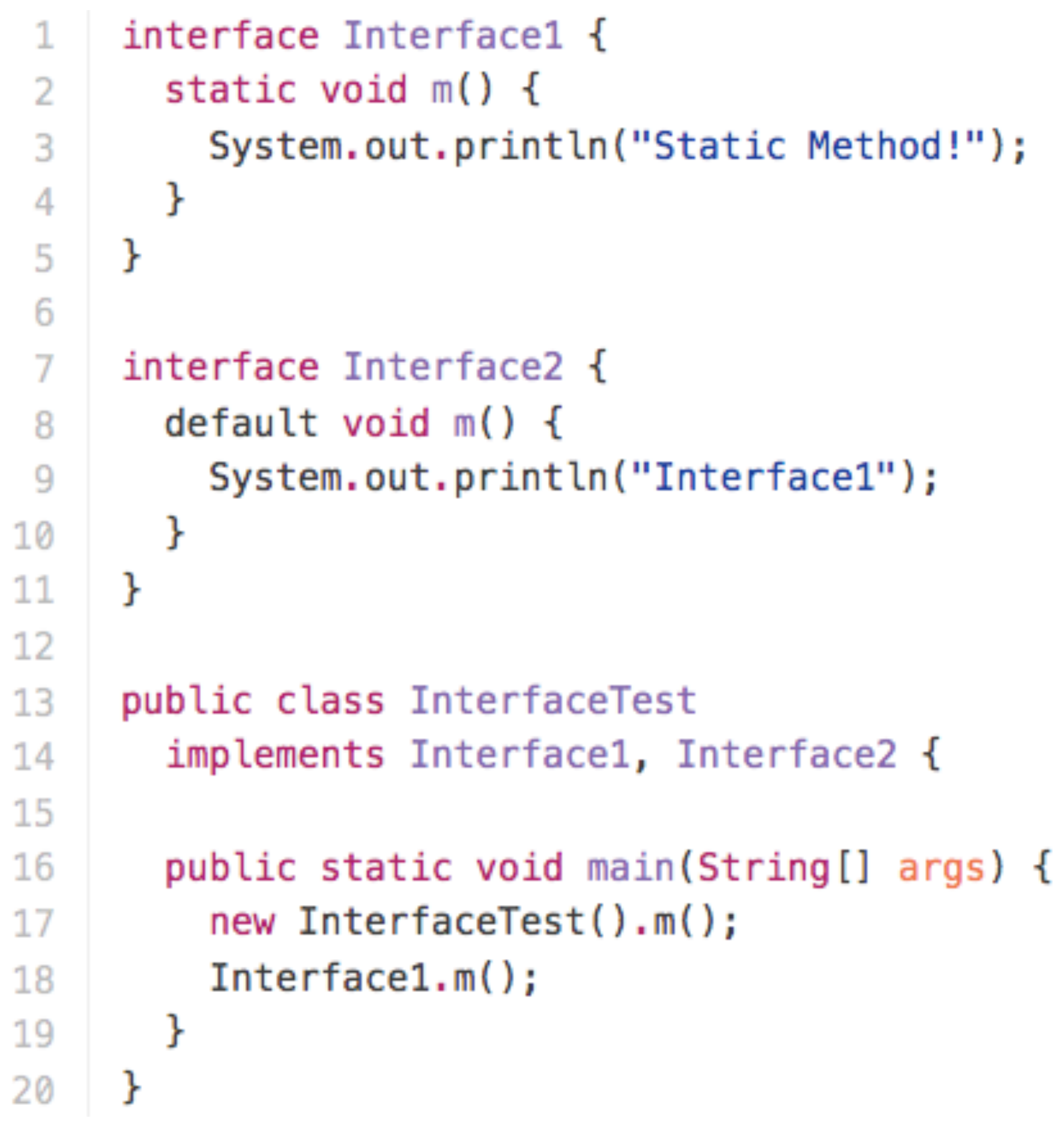
Default Method Use Cases

- Increased design flexibility
    - For many cases, it's no longer necessary to create skeletal implementations.
    - Instead, you can add a default method for ‘default’ behavior override these methods as necessary.
    - Note that skeletal implementations may still be necessary if the class must have *state that it shares with subclasses.*

Static Methods

- Java 8 also introduced the ability to have static methods in interfaces.

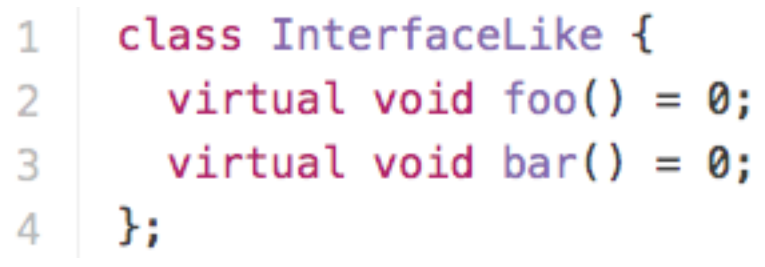
- Does this code compile?



```java
interface Interface1 {
  static void m() {
    System.out.println("Static Method!");
  }
}

interface Interface2 {
  default void m() {
    System.out.println("Interface1");
  }
}

public class InterfaceTest
  implements Interface1, Interface2 {

  public static void main(String[] args) {
    new InterfaceTest().m();
    Interface1.m();
  }
}
```

Interfaces in C++

- .. do not really exist.
- You can approximate their functionality however with ‘pure virtual abstract classes’
- We won’t get into the details, but you can search that term if you want more information.
- They look something like this…



```cpp
class InterfaceLike {
  virtual void foo() = 0;
  virtual void bar() = 0;
};
```

Conclusion

- An interface is generally the best way to define a type that permits multiple implementations.

- An exception to this rule is when you have a very constrained problem in which Coad’s Rule are satisfied, most importantly:

  1. A subclass would express a *"is a special kind of"* and not *"is a role played by a” relationship.*

  2. An instance of a subclass never needs to behave as an object of another class.

- When in doubt use definition inheritance!