Compilation Issues in Objected-Oriented Language

- OO language features
- Single inheritance
- Multiple inheritance

Object-Oriented Programming

- OO programming represents real-world objects into software objects
  - Real-world objects have states and behaviors which are represented by instance variables and methods in software objects
- OO programming languages support encapsulation and inheritance

Encapsulation

- Information hiding and modularity
  - Instance variables are not accessible outside of the object
  - They are accessible only through the methods

Classes

- A software blueprint for the same kind of objects is called a class
  - A car class: variable declarations and method implementations
  - Must instantiate the car class to create a car object

Inheritance

- Classes can be defined in terms of other classes
  - Hierarchy of classes
  - Each subclass inherits variables and methods from superclass
  - Subclass can also add its own variables and methods
  - Subclass can override inherited methods and provide specialized implementation for it

Polymorphism

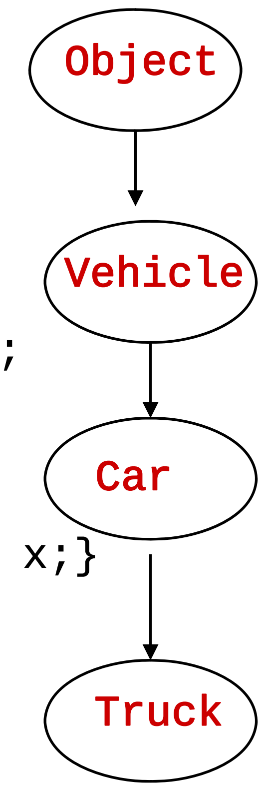
- A subclass instance can be used anywhere that one of its superclass is expected
  - As the value assigned to a variable or an argument
  - As the object on which a method is invoked

```
public class Car-demonstrate {
    public static void main(String argv[]) {
        Vehicle x = new Car();
        x.move();
    }
}
```

An Example OO Program

```
class Vehicle extends Object {
    int position=0;
    void move( int x ) { position = position + x; }
}
class Car extends Vehicle {
    int passenger=0;
    void await( Vehicle v ) { if (v.position < position)
                               v.move(position-v.position);
                              else move(10); }
}
class Truck extends Vehicle {
    void move( int x ) { if (x < 55) position = position + x;}
    void load( int x ) { ……  }

Truck t = new Truck(); Car c = new Car(); Vehicle v = c;
c.passenger=2;
c.move( 60 );
v.move( 70 ); c.await( t );
```



OO Compilation Issues

- How to layout class data fields and how to generate code to access them
- How to layout the method table and how to generate code to access them
- Compile-time binding vs. run-time binding
- How to support multiple inheritance

Class Descriptor

- As in non-OO languages, compiler needs to collect information on classes such as deciding data fields layout and recording the addresses of methods included in them
  - The information is saved in a class descriptor
    - Offsets of data fields
    - Addresses of methods
    - Compiler consult class descriptor for code generation

Access to Data Fields

- Data fields are located at objects separately
- For `Vehicle v; v.position` must be compiled into a load from the object pointed to by `v`
  - Offset of `position` can be found from a symbol table where `vehicle` class information is saved
  - However, v can also point to a `car` object and if the offset of `position` in a `car` object is different, we do not know how to compile `v.position`
    - Example: `Vehicle`: position (offset 0)
      `Car`: passenger (offset 0), position (offset 1)

Data Layout with Single Inheritance

- If each class can extend only one parent class (as in java), prefixing of data fields is used
  - When B extends A, those fields that are inherited from A are laid out in the B object at the beginning, in the same order as they appear in A, then B's fields are laid
    - Then, each field will have a unique offset no matter which object it is included
  - Access of a field for an object: since the compiler knows the offset of a field, it is a single memory access (e.g., r = load (address_object + offset))

An Example of Prefixing

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

```
 A      B      C      D
---    ---    ---    ---
 a      a      a      a
        b      d      b
        c             c
                      e
```

Access to Methods

- Need to know the method address for jump
- Method addresses are located at the class descriptor since they can be shared
- There exist differences between
  - Class method: address can be known at compile time
  - Instance method: address can be decided at run time

Class (Static) Methods

- Compiler searches across class hierarchy
  - For `Car c; c.f()`, for example, compiler searches for `f()` in the `Car` class; if not there, searches for its parent class; if the compiler finds `f()` in a superclass, say A, then `c.f()` is compiled into a jump to A_f()
    - Although `c` can point to a subclass object (e.g., `Truck`), `f()` must be a method available at c's class

Instance (Dynamic) Methods

- Due to polymorphism and overriding, it is impossible to decide at compile-time which method will be called at run-time

```
class A extends Object int x {int x=0; method f()}
class B extends A            {method g()}
class C extends B            {method g()}
class D extends C            {int y; method f()}
main() {C c; ... printf( c.f() ); ...}
```

- In this example, a method call **`c.f()`** will be a call to A_f() if the variable points an object of C (i.e., **`c = new(C);`**) while it is a call to D_f() if it points to an object of D (i.e., **`c = new(D);`**)

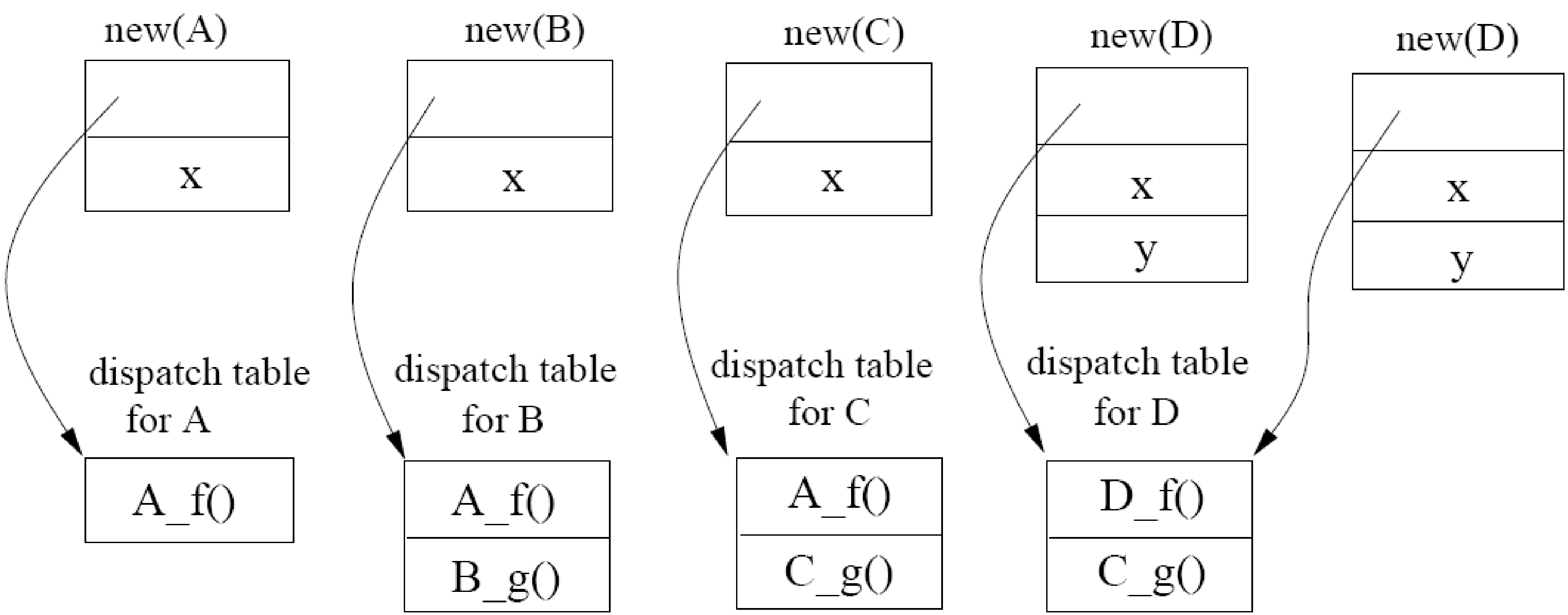
Dispatch Table: runtime data structure

- Compiler must generate a dispatch table for each class, which contains addresses for all methods available in the class (saved in code or global area)
- Each object will have a pointer to this table
- An instance method call `c.f()` will be translated to
  - Load the start address of the dispatch table
    - `R1 = load(c + offset_of_pointer_to_dispatch_table)`
  - Load the method address
    - `R2 = load(R1 + offset_of_f())`
  - Jump `R2`
- Can we know `offset_of_f()` at compile time?
  - If `A_f()` and `D_f()` have different offsets, it would cause a trouble

Method Layout with Single Inheritance

- Employ a similar layout as prefixing
  - When class B extends class A, B's dispatch table starts with entries for all method names known to A and then continues with new methods declared by B
  - An overridden method points to a different method-instance address
  - Creation of an object will keep a pointer to the dispatch table for the corresponding class that is `newed`

Example Dispatch Tables



- Offsets
  - f(): 0, g(): 1

Dealing with Multiple Inheritance

- For languages that allow class D to extend several parent classes A, B, and C, finding data field offsets and method instances is more difficult
  - E.g., it is impossible to put all of both A's fields and B's fields at the beginning of D, for the example below:

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

One Solution: Graph Coloring

- Statically analyze all classes at once, find some unique field offset for each field name, which can be used in every object containing the field
  - Can model this as a graph-coloring problem
    - There is a node for each "distinct" field name and an edge between two nodes which co-exist in the same class
    - The offsets 0, 1, 2, .. are the colors
    - Distinct name does not mean simple equivalence of strings; each fresh, non-overriding declaration of x is a distinct name
  - Access of a field is still a single memory access since the compiler can determine the offset

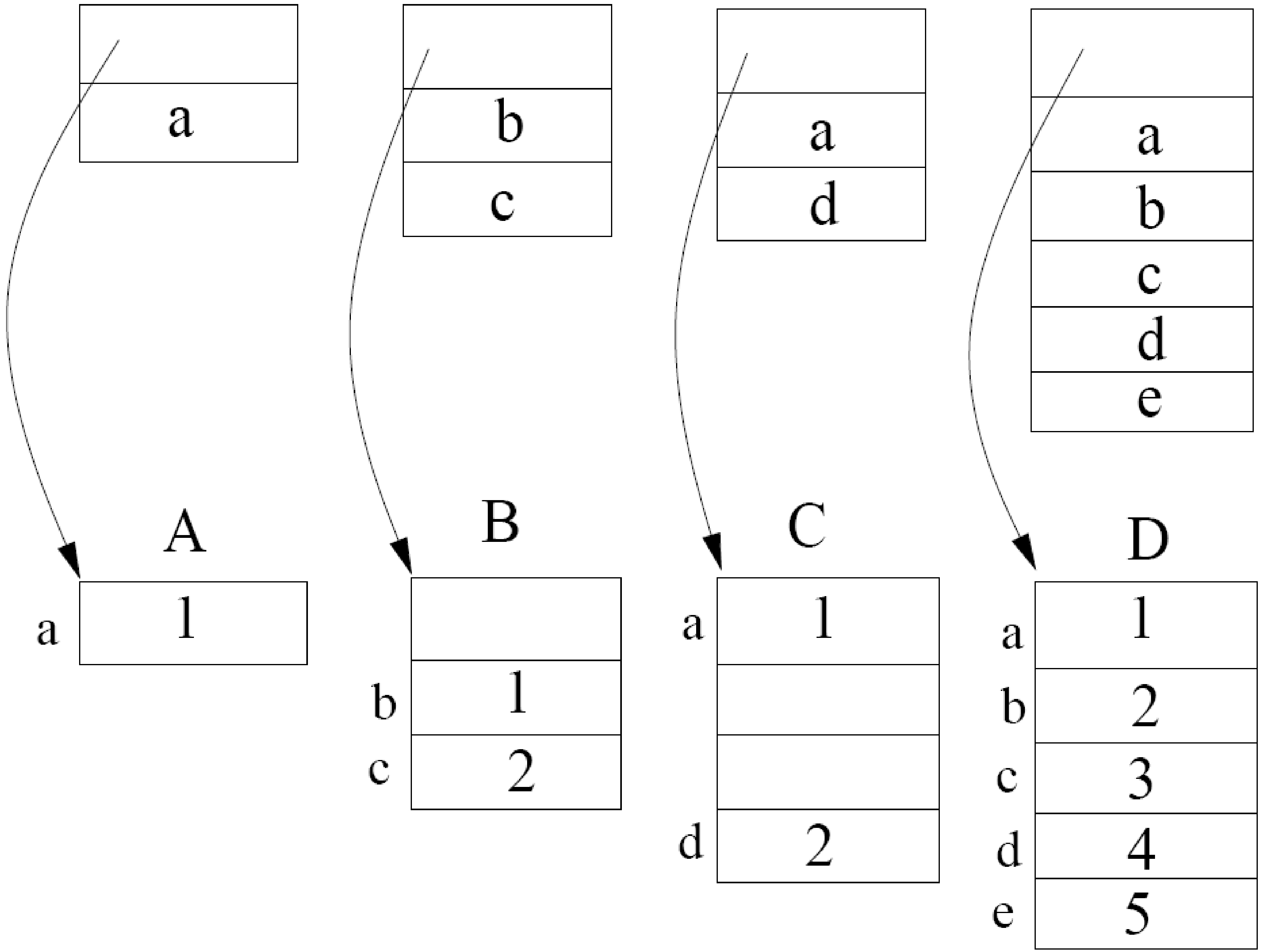
Graph Coloring Solution for the Example

- Offsets: a (0), b (1), c (2), d (3), e (4)

		A	B	C	D
		---	---	---	---
class A extends Object	{int a=0;}	a		a	a
class B extends Object	{int b=0, c=0;}		b		b
			c		c
class C extends A	{int d=0;}			d	d
class D extends A,B,C	{int e=0;}				e

Problems of Graph Coloring

- There are empty slots in the middle of an object since we cannot color the N fields of a class with the first N offsets
- Solution: pact the fields of each object and have the class descriptor tell where each field is located (now colors are offsets within a descriptor, not within an object)
  - Since the number of descriptors is much smaller compared to that of objects, the empty slots within the descriptor are acceptable
  - Access of a field requires three memory accesses, though
    - load descriptor pointer, load field-offset value, load/store the data


a
b
c
a
d
a
b
c
d
e
A
a
1
B
b
1
c
2
C
a
1
d
2
D
a
1
b
2
c
3
d
4
e
5

Multiple Inheritance with Dynamic Linking

- Dynamic linking systems like Java resolve references (change name into offsets) at run-time
  - Single inheritance does not cause any problem
- Graph coloring for multiple inheritance has problem
  - Dynamic linking system allows loading of new classes into a running system; those classes may be subclasses of classes already in use
  - Run-time graph coloring poses many problems