

Compilation

0368-3133 2018/19

Lecture 10

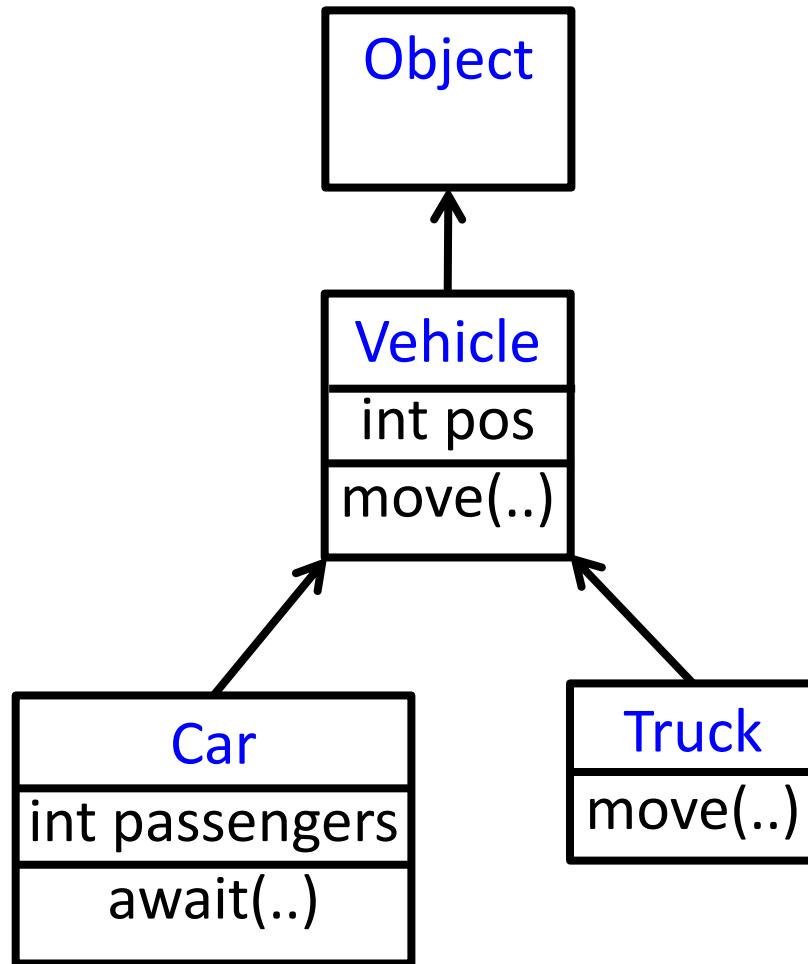
Compiling Object-Oriented Programs

Noam Rinetzky

Object Oriented Programs

- C++, Java, C#, Python, ...
- Main abstraction: **Objects** (usually of type called class)
 - Code
 - Data
- Naturally supports **Abstract Data Type** implementations
- Information hiding
- Evolution & reusability
- Important characteristic: Extension/Inheritance

A Simple Example



A Simple Example

```
class Vehicle extends Object {  
    int pos = 10;  
    void move(int x) {  
        pos = pos + x ;  
    }  
}
```

```
class Truck extends Vehicle {  
    void move(int x){  
        if (x < 55)  
            pos = pos + x;  
    }  
}
```

```
class Car extends Vehicle {  
    int passengers = 0;  
    void await(vehicle v){  
        if (v.pos < pos)  
            v.move(pos - v.pos);  
        else  
            this.move(10);  
    }  
}
```

```
class main extends Object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle o = c;  
        o.move(60);  
        t.move(70);  
        c.await(t);  
    }  
}
```

A Simple Example

```
class Vehicle extends object {  
    int pos = 10;  
    void move(int x) {  
        position = position + x ;  
    }  
}
```

```
class Truck extends Vehicle {  
    void move(int x){  
        if (x < 55)  
            pos = pos + x;  
    }  
}
```

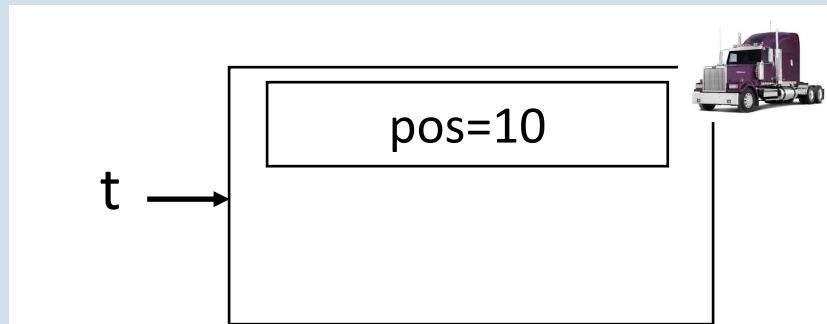
```
class Car extends Vehicle {  
    int passengers = 0;  
    void await(vehicle v){  
        if (v.pos < pos)  
            v.move(pos - v.pos);  
        else  
            this.move(10);  
    }  
}
```

```
class main extends Object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle o = c;  
        o.move(60);  
        t.move(70);  
        c.await(t);  
    }  
}
```

A Simple Example

```
class Vehicle extends object {  
    int pos = 10;  
    void move(int x) {  
        position = position + x ;  
    }  
}  
  
class Truck extends Vehicle {  
    void move(int x){  
        if (x < 55)  
            pos = pos + x;  
    }  
}  
  
class Car extends Vehicle {  
    int passengers = 0;  
    void await(vehicle v){  
        if (v.pos < pos)  
            v.move(pos - v.pos);  
        else  
            this.move(10);  
    }  
}
```

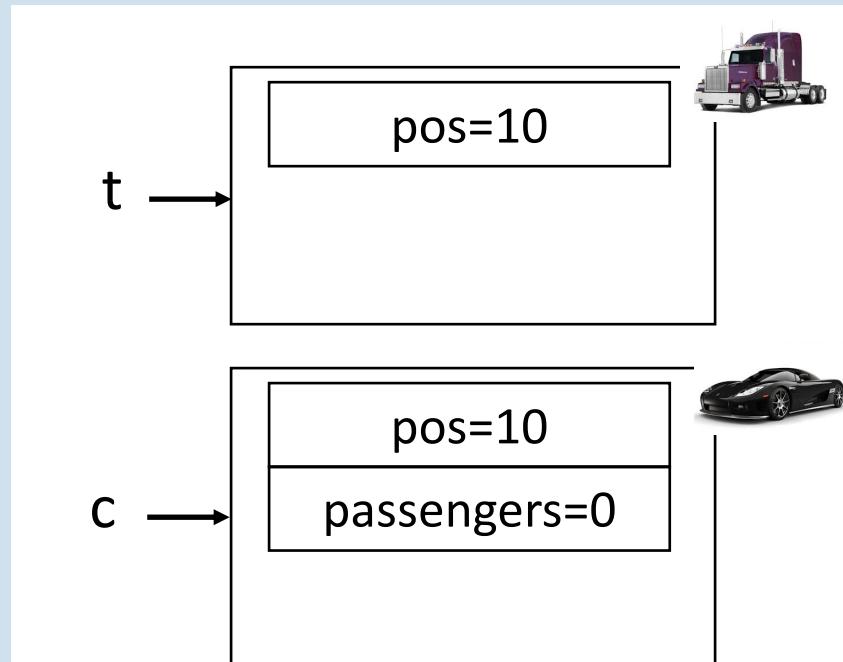
```
class main extends Object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle o = c;  
        o.move(60);  
        t.move(70);  
        c.await(t);  
    }  
}
```



A Simple Example

```
class Vehicle extends object {  
    int pos = 10;  
    void move(int x) {  
        pos = pos + x ;  
    }  
}  
  
class Truck extends Vehicle {  
    void move(int x){  
        if (x < 55)  
            pos = pos + x;  
    }  
}  
  
class Car extends Vehicle {  
    int passengers = 0;  
    void await(vehicle v){  
        if (v.pos < pos)  
            v.move(pos - v.pos);  
        else  
            this.move(10);  
    }  
}
```

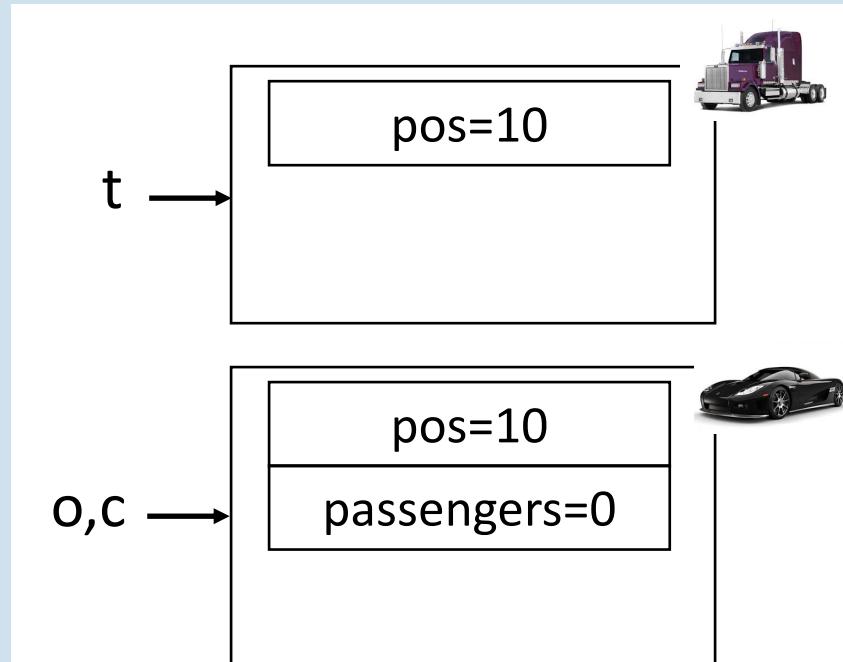
```
class main extends Object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle o = c;  
        o.move(60);  
        t.move(70);  
        c.await(t);  
    }  
}
```



A Simple Example

```
class Vehicle extends object {  
    int pos = 10;  
    void move(int x) {  
        pos = pos + x ;  
    }  
}  
  
class Truck extends Vehicle {  
    void move(int x){  
        if (x < 55)  
            pos = pos + x;  
    }  
}  
  
class Car extends Vehicle {  
    int passengers = 0;  
    void await(vehicle v){  
        if (v.pos < pos)  
            v.move(pos - v.pos);  
        else  
            this.move(10);  
    }  
}
```

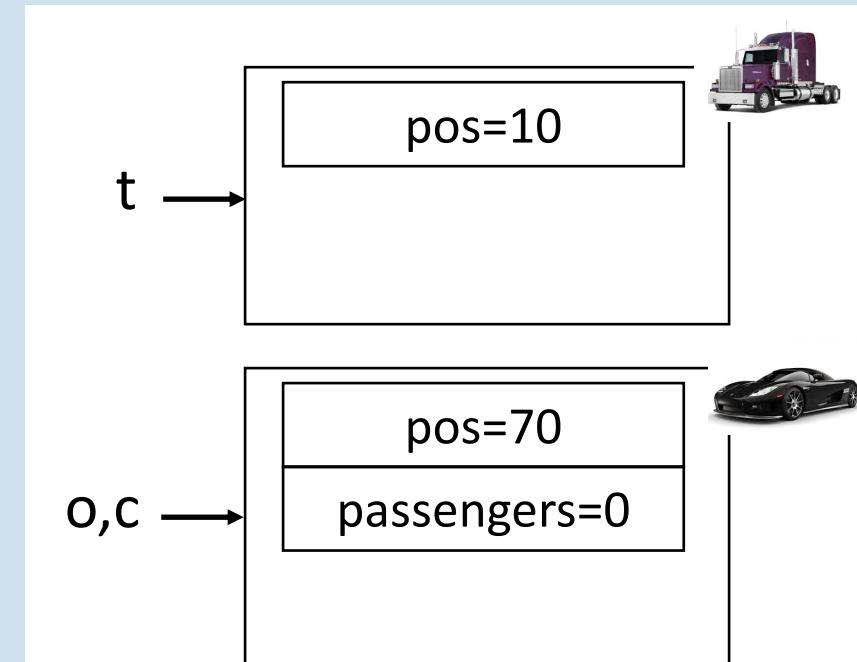
```
class main extends Object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle o = c;  
        o.move(60);  
        t.move(70);  
        c.await(t);  
    }  
}
```



A Simple Example

```
class Vehicle extends object {  
    int pos = 10;  
    void move(int x) {  
        pos = pos + x;  
    }  
}  
  
class Truck extends Vehicle {  
    void move(int x){  
        if (x < 55)  
            pos = pos + x;  
    }  
}  
  
class Car extends Vehicle {  
    int passengers = 0;  
    void await(vehicle v){  
        if (v.pos < pos)  
            v.move(pos - v.pos);  
        else  
            this.move(10);  
    }  
}
```

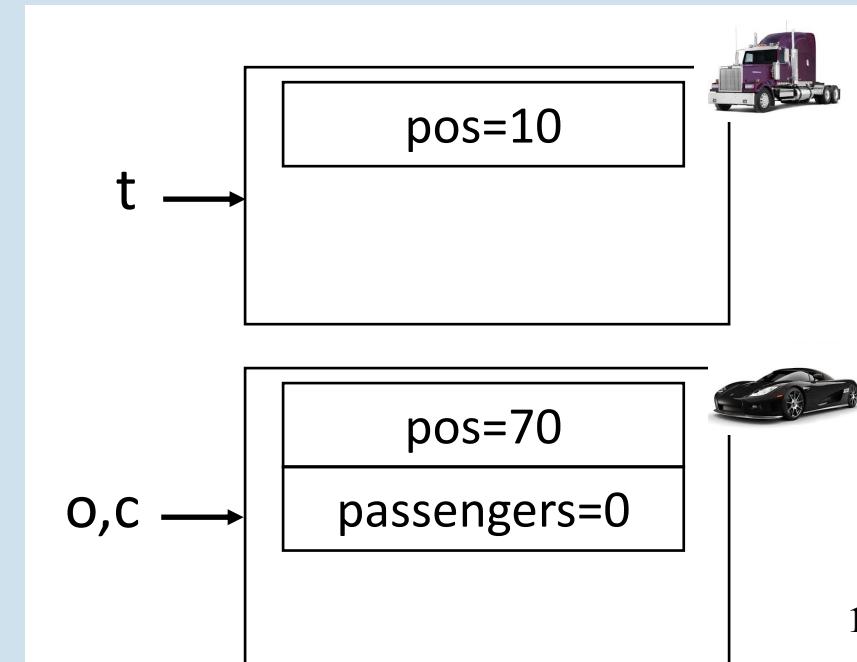
```
class main extends Object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle o = c;  
        o.move(60);  
        t.move(70);  
        c.await(t);  
    }  
}
```



A Simple Example

```
class Vehicle extends object {  
    int pos = 10;  
    void move(int x) {  
        pos = pos + x;  
    }  
}  
  
class Truck extends Vehicle {  
    void move(int x){  
        if (x < 55)  
            pos = pos + x;  
    }  
}  
  
class Car extends Vehicle {  
    int passengers = 0;  
    void await(vehicle v){  
        if (v.pos < pos)  
            v.move(pos - v.pos);  
        else  
            this.move(10);  
    }  
}
```

```
class main extends Object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle o = c;  
        o.move(60);  
        t.move(70);  
        c.await(t);  
    }  
}
```



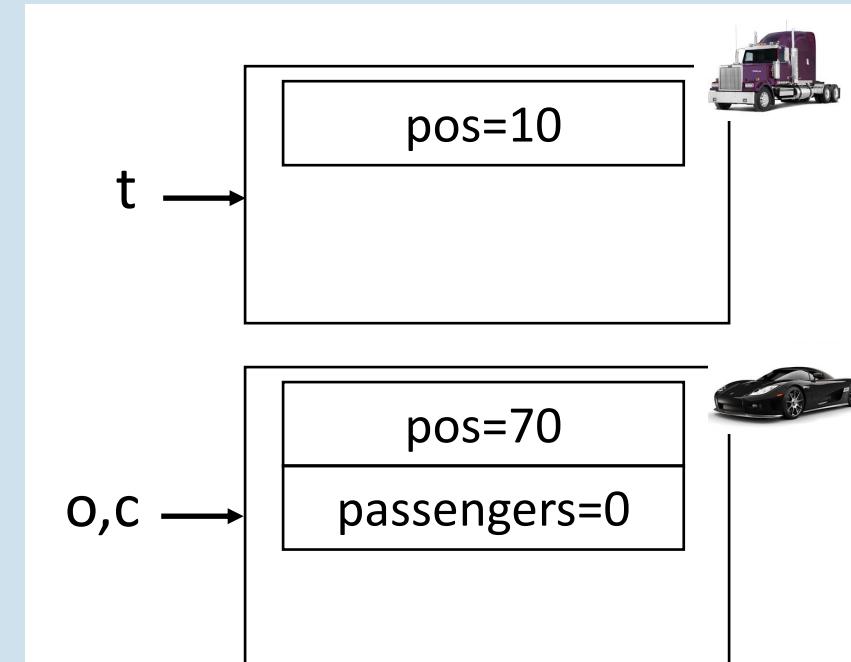
A Simple Example

```
class Vehicle extends object {  
    int pos = 10;  
    void move(int x) {  
        pos = pos + x;  
    }  
}
```

```
class Truck extends Vehicle {  
    void move(int x){  
        if (x < 55)  
            pos = pos + x;  
    }  
}
```

```
class Car extends Vehicle {  
    int passengers = 0;  
    void await(vehicle v){  
        if (v.pos < pos)  
            v.move(pos - v.pos);  
        else  
            this.move(10);  
    }  
}
```

```
class main extends Object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle o = c;  
        o.move(60);  
        t.move(70);  
        c.await(t);  
    }  
}
```



Translation into C

Translation into C (Vehicle)

```
class Vehicle extends Object {  
    int pos = 10;  
    void move(int x) {  
        pos = pos + x ;  
    }  
}
```

```
typedef struct Vehicle {  
    int pos;  
} Ve;
```

Translation into C (Vehicle)

```
class Vehicle extends Object {  
    int pos = 10;  
    void move(int x) {  
        pos = pos + x ;  
    }  
}
```

```
typedef struct Vehicle {  
    int pos;  
} Ve;  
  
void NewVe(Ve *this){  
    this→pos = 10;  
}  
  
void moveVe(Ve *this, int x){  
    this→pos = this→pos + x;  
}
```

Translation into C (Truck)

```
class Truck extends Vehicle {  
void move(int x){  
    if (x < 55)  
        pos = pos + x;  
}  
}
```

```
typedef struct Truck {  
    int pos;  
} Tr;  
  
void NewTr(Tr *this){  
    this→pos = 10;  
}  
  
void moveTr(Ve *this, int x){  
    if (x<55)  
        this→pos = this→pos + x;  
}
```

Naïve Translation into C (Car)

```
class Car extends Vehicle {  
    int passengers = 0;  
    void await(vehicle v){  
        if (v.pos < pos)  
            v.move(pos - v.pos);  
        else  
            this.move(10);  
    }  
}
```

```
typedef struct Car{  
    int pos;  
    int passengers;  
} Ca;  
  
void NewCa (Ca *this){  
    this→pos = 10;  
    this→passengers = 0;  
}  
  
void awaitCa(Ca *this, Ve *v){  
    if (v→pos < this→pos)  
        moveVe(v, this→pos - v→pos)  
    else  
        MoveCa(this, 10)  
}
```

Naïve Translation into C (Main)

```
class main extends object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle v = c;  
        v.move(60);  
        t.move(70);  
        c.await(t);  
    }  
}
```

```
void mainMa(){  
    Tr *t = malloc(sizeof(Tr));  
    Ca *c = malloc(sizeof(Ca));  
    Ve *v = (Ve*) c;  
    moveVe(v, 60);  
    moveVe(t, 70);  
    awaitCa(c, (Ve*) t);  
}
```

Naïve Translation into C (Main)

```
class main extends object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle v = c;  
        v.move(60);  
        t.move(70);  
        c.await(t);  
    }  
}
```

```
void mainMa(){  
    Tr *t = malloc(sizeof(Tr));  
    Ca *c = malloc(sizeof(Ca));  
    Ve *v = (Ve*) c;  
    moveVe(v, 60);  
    moveVe(t, 70);  
    awaitCa(c, (Ve*) t);  
}
```

```
void moveCa() ?
```

Naïve Translation into C (Main)

```
class main extends object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle v = c;  
        v.move(60);  
        t.move(70);  
        c.await(t);  
    }  
}
```

```
void mainMa(){  
    Tr *t = malloc(sizeof(Tr));  
    Ca *c = malloc(sizeof(Ca));  
    Ve *v = (Ve*) c;  
    moveVe(v, 60);  
    moveVe(t, 70);  
    awaitCa(c, (Ve*) t);  
}
```

```
void moveCa() ?
```

```
void moveVe(Ve *this, int x){  
    this->pos = this->pos + x;  
}
```

Naïve Translation into C (Main)

```
class main extends object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle v = c;  
        v.move(60);  
        t.move(70);  
        c.await(t);  
    }  
}
```

```
typedef struct Vehicle {  
    int pos;  
} Ve;
```

```
typedef struct Car{  
    int pos;  
    int passengers;  
} Ca;
```

```
void mainMa(){  
    Tr *t = malloc(sizeof(Tr));  
    Ca *c = malloc(sizeof(Ca));  
    Ve *v = (Ve*) c;  
    moveVe(v, 60);  
    moveVe(t, 70);  
    awaitCa(c, (Ve*) t);  
}
```

```
void moveCa() ?
```

```
void moveVe(Ve *this, int x){  
    this->pos = this->pos + x;  
}
```

Naïve Translation into C (Main)

```
class main extends object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle v = c;  
        v.move(60);  
        t.move(70);  
        c.await(t);  
    }  
}
```

Vehicle x = t;
x.move(20);

```
void mainMa(){  
    Tr *t = malloc(sizeof(Tr));  
    Ca *c = malloc(sizeof(Ca));  
    Ve *v = (Ve*) c;  
    moveVe(v, 60);  
    moveVe(t, 70);  
    awaitCa(c, (Ve*) t);  
}
```

```
Ve *x = t;  
moveTr((Tr*)x, 20);
```

Naïve Translation into C (Main)

```
class main extends object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle v = c;  
        v.move(60);  
        t.move(70);  
        c.await(t);  
    }  
}
```

Vehicle x = t;
x.move(20);

```
void mainMa(){  
    Tr *t = malloc(sizeof(Tr));  
    Ca *c = malloc(sizeof(Ca));  
    Ve *v = (Ve*) c;  
    moveVe(v, 60);  
    moveVe(t, 70);  
    awaitCa(c, (Ve*) t);  
}
```

```
Ve *x = t;  
moveTr((Tr*)x, 70);
```

```
void moveVe(Ve *this, int x){...}
```

```
void moveTr(Ve *this, int x){...}
```

Translation into C

Compiling Simple Classes

- Fields are handled as records
- Methods have unique names

```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
    method m2(int i) {...}  
}
```

Runtime object

a1
a2

Compile-Time Table

m1A
m2A

```
void m2A(classA *this, int i) {  
    // Body of m2 with any object  
    // field f as this?f  
  
    ...  
}
```

Compiling Simple Classes

- Fields are handled as records
- Methods have unique names

```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
    method m2(int i) {...}  
}  
  
a.m2(5)  
  
m2A(a,5)
```

Runtime object

a1
a2

Compile-Time Table

m1A
m2A

```
void m2A(classA *this, int i) {  
    // Body of m2 with any  
    // object-field f as this?f  
    ...  
}
```

Features of OO languages

- **Inheritance**
 - Subclass gets (inherits) properties of superclass
- **Method overriding**
 - Multiple methods with the same name with different signatures
- **Abstract (aka virtual) methods**
- **Polymorphism**
 - Multiple methods with the same name and different signatures but with different implementations
- **Dynamic dispatch**
 - Lookup methods by (runtime) type of target object

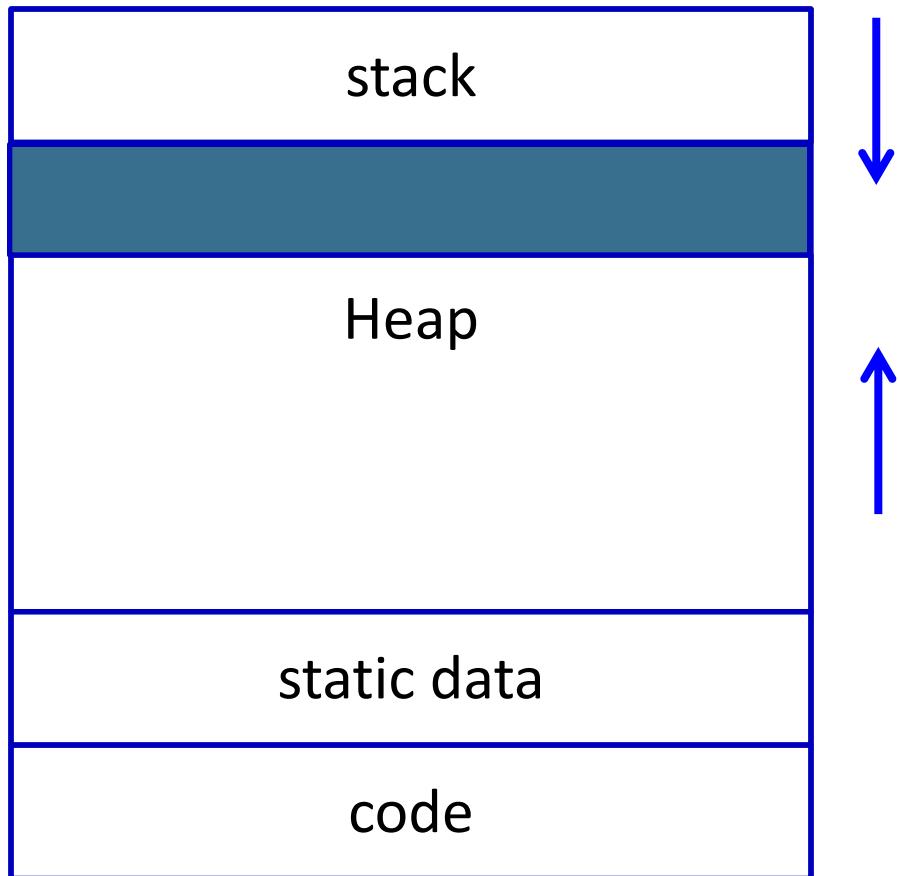
Compiling OO languages

- “Translation into C”
- Powerful runtime environment
- Adding “gluing” code

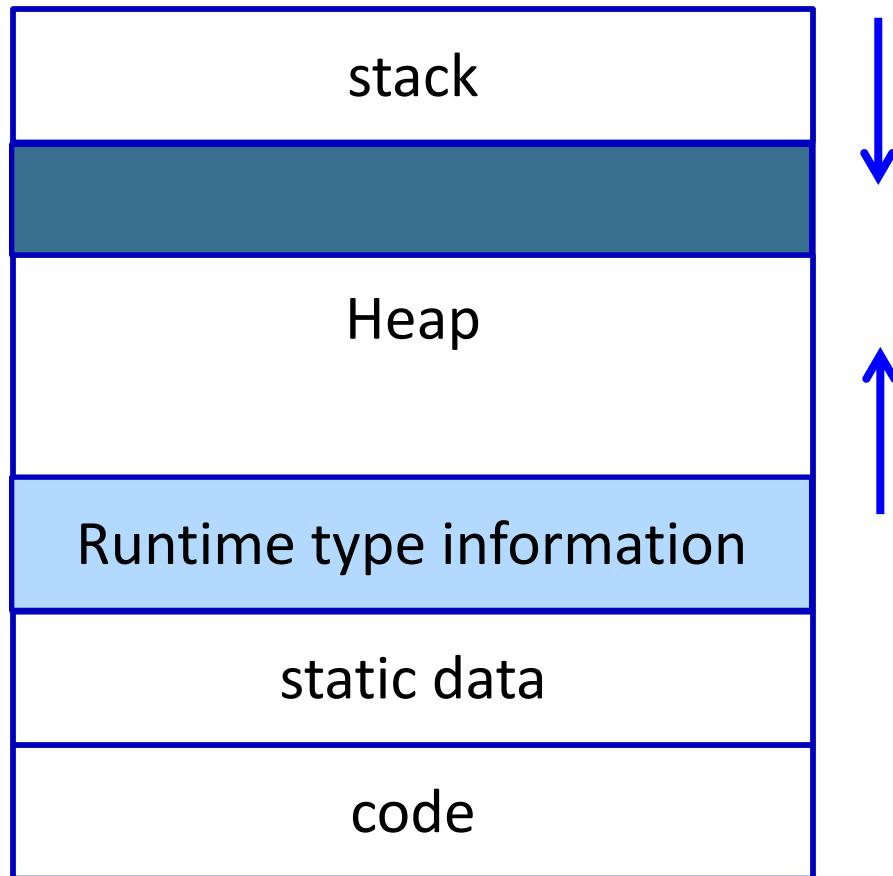
Runtime Environment

- Mediates between the OS and the programming language
- Hides details of the machine from the programmer
 - Ranges from simple support functions all the way to a full-fledged virtual machine
- Handles common tasks
 - Runtime stack (activation records)
 - Memory management
- **Runtime type information**
 - **Method invocation**
 - **Type conversions**

Memory Layout



Memory Layout



Handling Single Inheritance

- Simple type extension

```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
    method m2() {...}  
}
```

```
class B extends A {  
    field b1;  
    method m3() {...}  
}
```

Adding fields

Fields aka Data members, instance variables

- Adds more information to the inherited class
 - “Prefixing” fields ensures consistency

```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
    method m2() {...}  
}
```

```
typedef struct {  
    field a1;  
    field a2;  
} A;  
  
void m1A_A(A* this){...}  
void m2A_A(A* this){...}
```

```
class B extends A {  
    field b1;  
    method m2() {...}  
    method m3() {...}  
}
```

```
typedef struct {  
    field a1;  
    field a2;  
    field b1;  
} B;  
  
void m2A_B(B* this) {...}  
void m3B_B(B* this) {...}
```

Method Overriding

- Redefines functionality
 - More specific
 - Can access additional fields

```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
    method m2() {...}  
}
```

```
class B extends A {  
    field b1;  
    method m2() {  
        ... b1 ...  
    }  
    method m3() {...}  
}
```

Method Overriding

- Redefines functionality
 - More specific
 - Can access additional fields

```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
method m2() {...}  
}
```

m2 is declared and defined

m2 is redefined

```
class B extends A {  
    field b1;  
method m2() {  
        ... b1 ...  
}  
    method m3() {...}  
}
```

Method Overriding

- Redefines functionality
- Affects semantic analysis

```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
    method m2() {...}  
}
```

```
class B extends A {  
    field b1;  
    method m2() {  
        ... b1 ...  
    }  
    method m3() {...}  
}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Method Overriding

- Redefines functionality
- Affects semantic analysis

```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
    method m2() {...}  
}
```

```
class B extends A {  
    field b1;  
    method m2() {  
        ... b1 ...  
    }  
    method m3() {...}  
}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

declared

defined

Method Overriding

a.m2(5) // class(a) = A

m2A_A(a, 5)

```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
    method m2() {...}  
}
```

b.m2(5) // class(b) = B

m2A_B(b, 5)

```
class B extends A {  
    field b1;  
    method m2() {  
        ... b1 ...  
    }  
    method m3() {...}  
}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Method Overriding

```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
    method m2() {...}  
}
```

```
typedef struct {  
    field a1;  
    field a2;  
} A;  
  
void m1A_A(A* this){...}  
void m2A_A(A* this){...}
```

```
class B extends A {  
    field b1;  
    method m2() {  
        ... b1 ...  
    }  
    method m3() {...}  
}
```

```
typedef struct {  
    field a1;  
    field a2;  
    field b1;  
} B;  
  
void m2A_B(B* this) {...}  
void m3B_B(B* this) {...}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Method Overriding

a.m2(5) // class(a) = A

m2A_A(a, 5)

b.m2(5) // class(b) = B

m2A_B(b, 5)

```
typedef struct {  
    field a1;  
    field a2;  
} A;  
  
void m1A_A(A* this){...}  
void m2A_A(A* this){...}
```

```
typedef struct {  
    field a1;  
    field a2;  
    field b1;  
} B;  
  
void m2A_B(B* this) {...}  
void m3B_B(B* this) {...}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Abstract methods & classes

- Abstract methods
 - Declared separately, defined in child classes
 - E.g., C++ pure virtual methods, abstract methods in Java
- Abstract classes = class may have abstract methods
 - E.G., Java/C++ abstract classes
 - Abstract classes **cannot be instantiated**
- Abstract aka “virtual”
- Inheriting abstract class handled like regular inheritance
 - Compiler checks abstract classes are not allocated

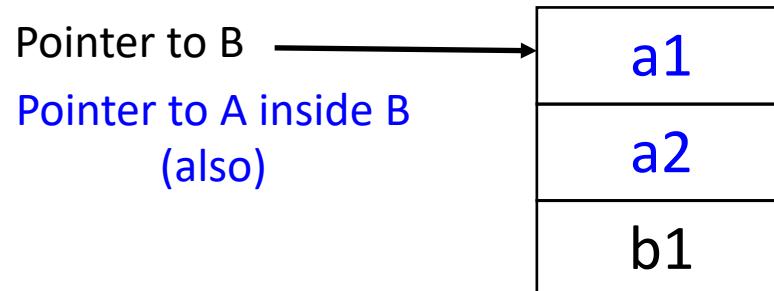
Handling Polymorphism

- When a class B extends a class A
 - variable of type pointer to A may actually refer to object of type B
- Upcasting from a subclass to a superclass
- Prefixing fields guarantees validity

```
class B *b = ...;
```

```
class A *a = b ;
```

```
classA *a = convert_ptr_to_B_to_ptr_A(b) ;
```



Dynamic Binding

- An object (“pointer”) declared to be of class A can actually be (“refer”) to a class B
- What does ‘o.m()’ mean?
 - Static binding
 - Dynamic binding
- Depends on the programming language rules
- How to implement dynamic binding?
 - The invoked function is not known at compile time
 - Need to operate on data of the B and A in consistent way

Conceptual Impl. of Dynamic Binding

```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
    method m2() {...}  
}
```

```
typedef struct {  
    field a1;  
    field a2;  
} A;  
  
void m1A_A(A* this){...}  
void m2A_A(A* this){...}
```

```
class B extends A {  
    field b1;  
    method m2() {  
        ... a3 ...  
    }  
    method m3() {...}  
}
```

```
typedef struct {  
    field a1;  
    field a2;  
    field b1;  
} B;  
  
void m2A_B(B* this) {...}  
void m3B_B(B* this) {...}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Conceptual Impl. of Dynamic Binding

"p.m2(3)"

```
switch(dynamic_type(p)) {  
    case Dynamic_class_A: m2_A_A(p, 3);  
    case Dynamic_class_B: m2_A_B(convert_ptr_to_A_to_ptr_B(p), 3);  
}
```

```
typedef struct {  
    field a1;  
    field a2;  
} A;  
  
void m1A_A(A* this){...}  
void m2A_A(A* this){...}
```

```
typedef struct {  
    field a1;  
    field a2;  
    field b1;  
} B;  
  
void m2A_B(B* this) {...}  
void m3B_B(B* this) {...}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Conceptual Impl. of Dynamic Binding

"p.m2(3)"

```
switch(dynamic_type(p)) {  
    case Dynamic_class_A: m2_A_A(p, 3);  
    case Dynamic_class_B: m2_A_B(convert_ptr_to_A_to_ptr_B(p), 3);  
}
```

```
typedef struct {  
    field a1;  
    field a2;  
} A;  
  
void m1A_A(A* this){...}  
void m2A_A(A* this){...}
```

```
typedef struct {  
    field a1;  
    field a2;  
    field b1;  
} B;  
  
void m2A_B(B* this) {...}  
void m3B_B(B* this) {...}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

More efficient implementation

- Apply pointer conversion in subclasses
 - Use dispatch table to invoke functions
 - Similar to table implementation of case

```
void m2A_B(classA *this_A) {  
    Class_B *this = convert_ptr_to_A_ptr_to_A_B(this_A);  
    ...  
}
```

More efficient implementation

```
typedef struct {  
    field a1;  
    field a2;  
} A;
```

```
void m1A_A(A* this){...}  
void m2A_A(A* this, int x){...}
```

```
typedef struct {  
    field a1;  
    field a2;  
    field b1;  
} B;
```

```
void m2A_B(A* thisA, int x){  
    Class_B *this =  
        convert_ptr_to_A_to_ptr_to_B(thisA);  
    ...  
}  
  
void m3B_B(B* this){...}
```

More efficient implementation

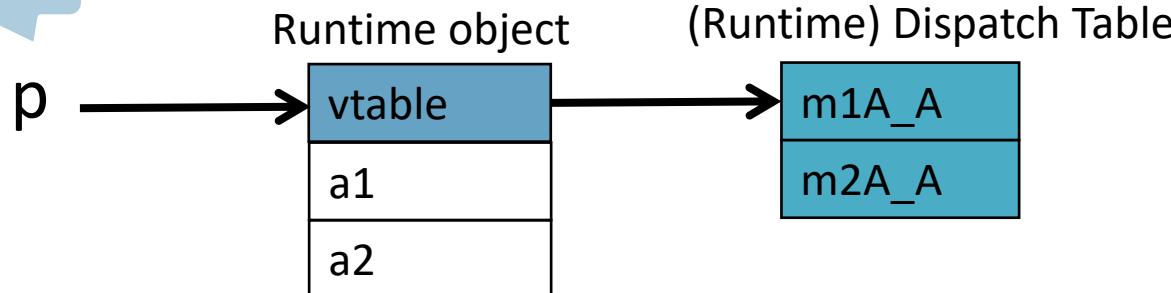
```
typedef struct {  
    field a1;  
    field a2;  
} A;
```

```
void m1A_A(A* this){...}  
void m2A_A(A* this, int x){...}
```

```
typedef struct {  
    field a1;  
    field a2;  
    field b1;  
} B;
```

```
void m2A_B(A* thisA, int x){  
    Class_B *this =  
        convert_ptr_to_A_to_ptr_to_B(thisA);  
    ...  
}  
  
void m3B_B(B* this){...}
```

classA *p;



More efficient implementation

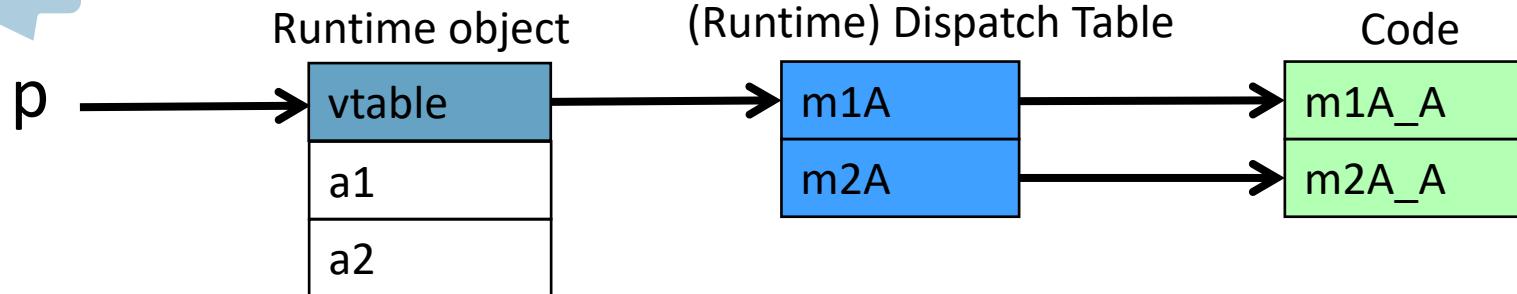
```
typedef struct {  
    field a1;  
    field a2;  
} A;
```

```
void m1A_A(A* this){...}  
void m2A_A(A* this, int x){...}
```

```
typedef struct {  
    field a1;  
    field a2;  
    field b1;  
} B;
```

```
void m2A_B(A* thisA, int x){  
    Class_B *this =  
        convert_ptr_to_A_to_ptr_to_B(thisA);  
    ...  
}  
  
void m3B_B(B* this){...}
```

classA *p;



More efficient implementation

```
typedef struct {  
    field a1;  
    field a2;  
} A;
```

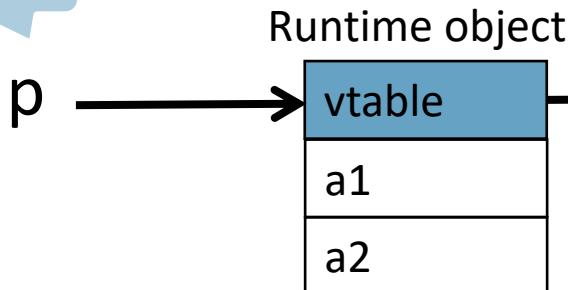
```
void m1A_A(A* this){...}  
void m2A_A(A* this, int x){...}
```

```
typedef struct {  
    field a1;  
    field a2;  
    field b1;  
} B;
```

```
void m2A_B(A* thisA, int x){  
    Class_B *this =  
        convert_ptr_to_A_to_ptr_to_B(thisA);  
    ...  
}  
  
void m3B_B(B* this){...}
```

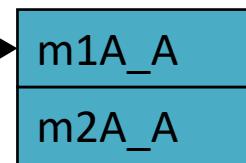
classA *p;

p.m2(3);



p→dispatch_table→m2A(p, 3);

(Runtime) Dispatch Table



More efficient implementation

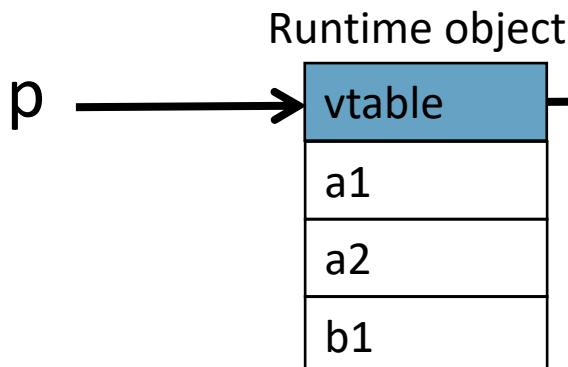
```
typedef struct {  
    field a1;  
    field a2;  
} A;
```

```
void m1A_A(A* this){...}  
void m2A_A(A* this, int x){...}
```

```
typedef struct {  
    field a1;  
    field a2;  
    field b1;  
} B;
```

```
void m2A_B(A* thisA, int x){  
    Class_B *this =  
        convert_ptr_to_A_to_ptr_to_B(thisA);  
    ...  
}  
  
void m3B_B(B* this){...}
```

p.m2(3);



p → dispatch_table → m2A(p, 3);

(Runtime) Dispatch Table

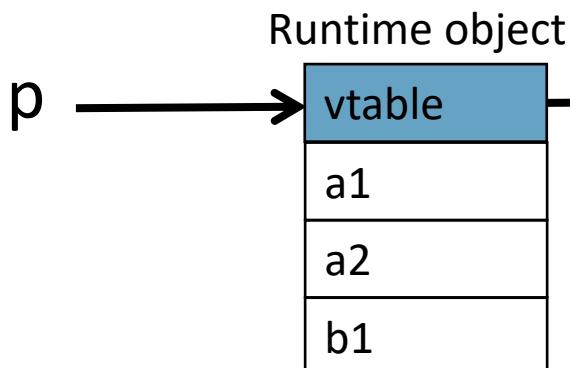
m1A_A
m2A_B
m3B_B

More efficient implementation

```
typedef struct {  
    field a1;  
    field a2;  
} A;  
  
void m1A_A(A* this){...}  
void m2A_A(A* this, int x){...}
```

```
typedef struct {  
    field a1;  
    field a2;  
    field b1;  
} B;  
  
void m2A_B(A* thisA, int x){  
    Class_B *this =  
        convert_ptr_to_A_to_ptr_to_B(thisA);  
    ...  
}  
  
void m3B_B(B* this){...}  
    convert_ptr_to_B_to_ptr_to_A(p)
```

p.m2(3);



(Runtime) Dispatch Table

m1A_A
m2A_B
m3B_B

Allocating objects (runtime)

- `x = new A()`
- Allocate memory for A's fields + pointer to vtable
- Initialize vtable to point to A's vtable
- initialize A's fields (call constructor)
- return object

Executing a.m2(5)

- Fetch the class descriptor at offset 0 from a
- Fetch the method-instance pointer p from the constant m2 offset of a
- call m2 (jump to address p)

Multiple Inheritance

```
class C {  
    field c1;  
    field c2;  
    method m1() {...}  
    method m2() {...}  
}  
  
class D {  
    field d1;  
    method m3() {...}  
    method m4() {...}  
}  
  
class E extends C, D {  
    field e1;  
  
    method m2() {...}  
    method m4() {...}  
    method m5() {...}  
}
```

Multiple Inheritance

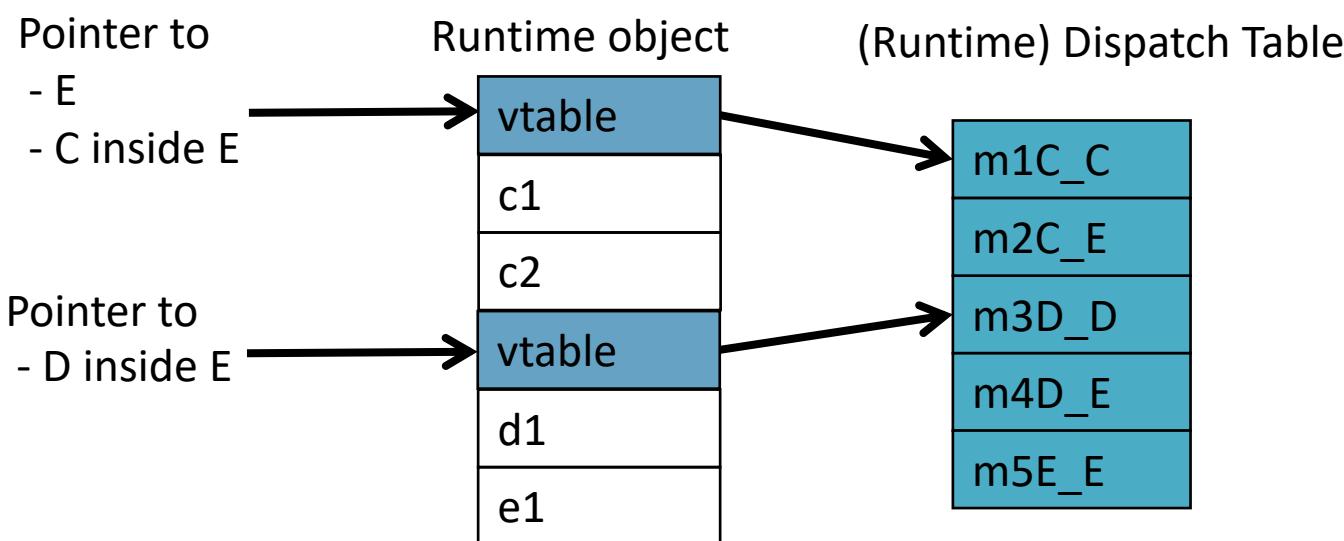
- Allows unifying behaviors
- But raises semantic difficulties
 - Ambiguity of classes
 - Repeated inheritance
- Hard to implement
 - Semantic analysis
 - Code generation
 - Prefixing no longer work
 - Need to generate code for downcasts
- Hard to use

A simple implementation

- Merge dispatch tables of superclasses
- Generate code for upcasts and downcasts

A simple implementation

```
class C {  
    field c1;  
    field c2;  
    method m1() {...}  
    method m2() {...}  
}  
  
class D {  
    field d1;  
    method m3() {...}  
    method m4() {...}  
}  
  
class E extends C, D {  
    field e1;  
    method m2() {...}  
    method m4() {...}  
    method m5() {...}  
}
```

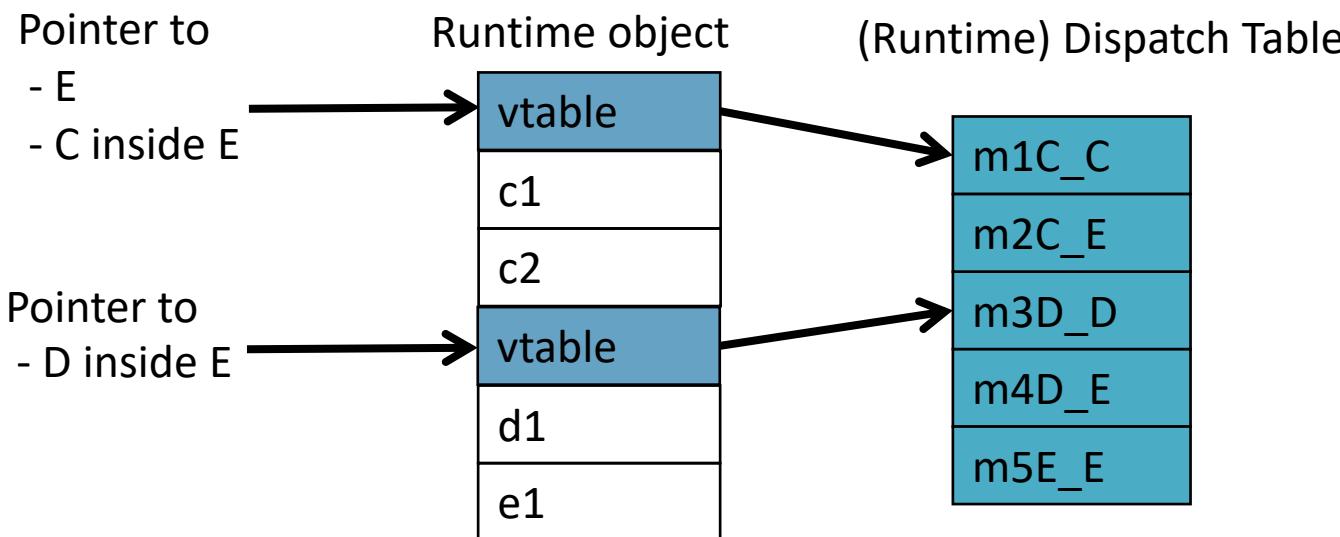


Downcasting (E → C,D)

```
class C {  
    field c1;  
    field c2;  
    method m1() {...}  
    method m2() {...}  
}  
  
class D {  
    field d1;  
    method m3() {...}  
    method m4() {...}  
}  
  
class E extends C, D {  
    field e1;  
    method m2() {...}  
    method m4() {...}  
    method m5() {...}  
}
```

`convert_ptr_to_E_to_ptr_to_C(e) = e;`

`convert_ptr_to_E_to_ptr_to_D(e) = e + sizeof(C);`

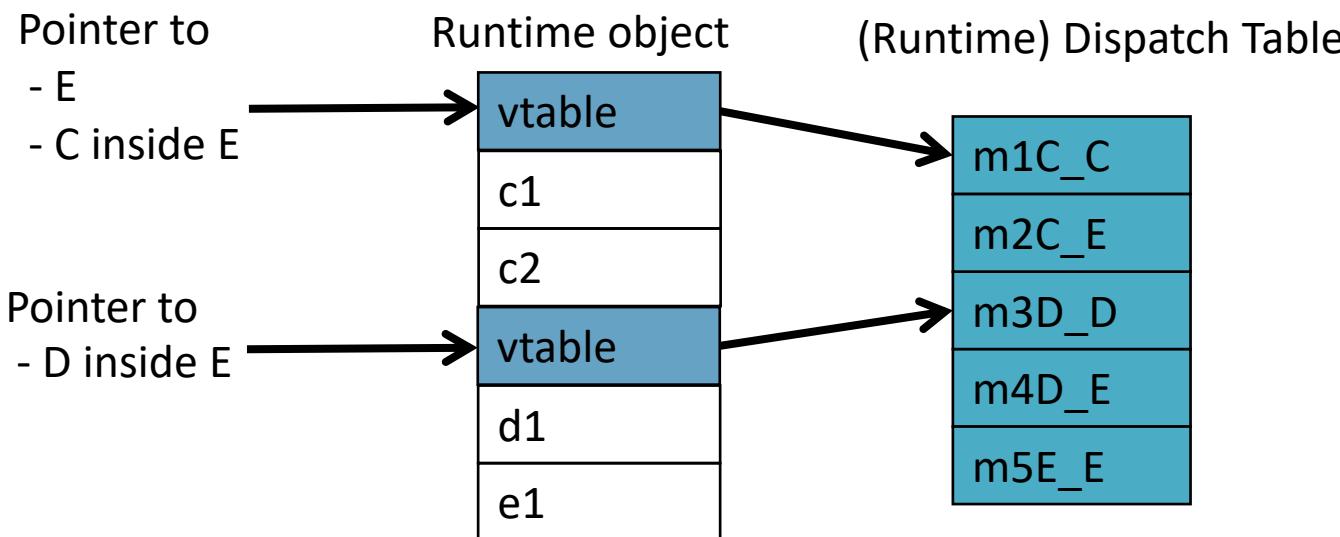


Upcasting (C,D→E)

```
class C {  
    field c1;  
    field c2;  
    method m1() {...}  
    method m2() {...}  
}  
  
class D {  
    field d1;  
    method m3() {...}  
    method m4() {...}  
}  
  
class E extends C, D {  
    field e1;  
    method m2() {...}  
    method m4() {...}  
    method m5() {...}  
}
```

`convert_ptr_to_C_to_ptr_to_E(c) = c;`

`convert_ptr_to_D_to_ptr_to_E(d) = d - sizeof(C);`



Independent multiple Inheritance

```
class A{  
    field a1;  
    field a2;  
    method m1() {...}  
    method m3() {...}  
}
```

```
class C extends A {  
    field c1;  
    field c2;  
    method m1() {...}  
    method m2() {...}  
}
```

```
class E extends C, D {  
    field e1;
```

```
class A{  
    field a1;  
    field a2;  
    method m1() {...}  
    method m3() {...}  
}
```

```
class D extends A {  
    field d1;  
    method m3() {...}  
    method m4() {...}  
}
```

```
method m2() {...}  
method m4() {...}  
method m5() {...}
```

Independent multiple Inheritance

```
class A{  
    field a1;  
    field a2;  
    method m1() {...}  
    method m3() {...}  
}
```

```
class C extends A {  
    field c1;  
    field c2;  
    method m1() {...}  
    method m2() {...}  
}
```

```
class E extends C, D {  
    field e1;
```

```
    method m2() {...}  
    method m4() {...}  
    method m5() {...}  
}
```

```
class A{  
    field a1;  
    field a2;  
    method m1() {...}  
    method m3() {...}  
}
```

```
class D extends A {  
    field d1;  
    method m3() {...}  
    method m4() {...}  
}
```

Independent multiple Inheritance

```
class A{  
    field a1;  
    field a2;  
    method m1() {...}  
    method m3() {...}  
}
```

```
class C extends A {  
    field c1;  
    field c2;  
    method m1() {...}  
    method m2() {...}  
}
```

```
class E extends C, D {  
    field e1;
```

```
    method m2() {...}  
    method m4() {...}  
    method m5() {...}  
}
```

```
class A{  
    field a1;  
    field a2;  
    method m1() {...}  
    method m3() {...}  
}
```

```
class D extends A {  
    field d1;  
    method m3() {...}  
    method m4() {...}  
}
```

Independent multiple Inheritance

```
class A{  
    field a1;  
    field a2;  
    method m1() {...}  
    method m3() {...}  
}
```

```
class C extends A {  
    field c1;  
    field c2;  
    method m1() {...}  
    method m2() {...}  
}
```

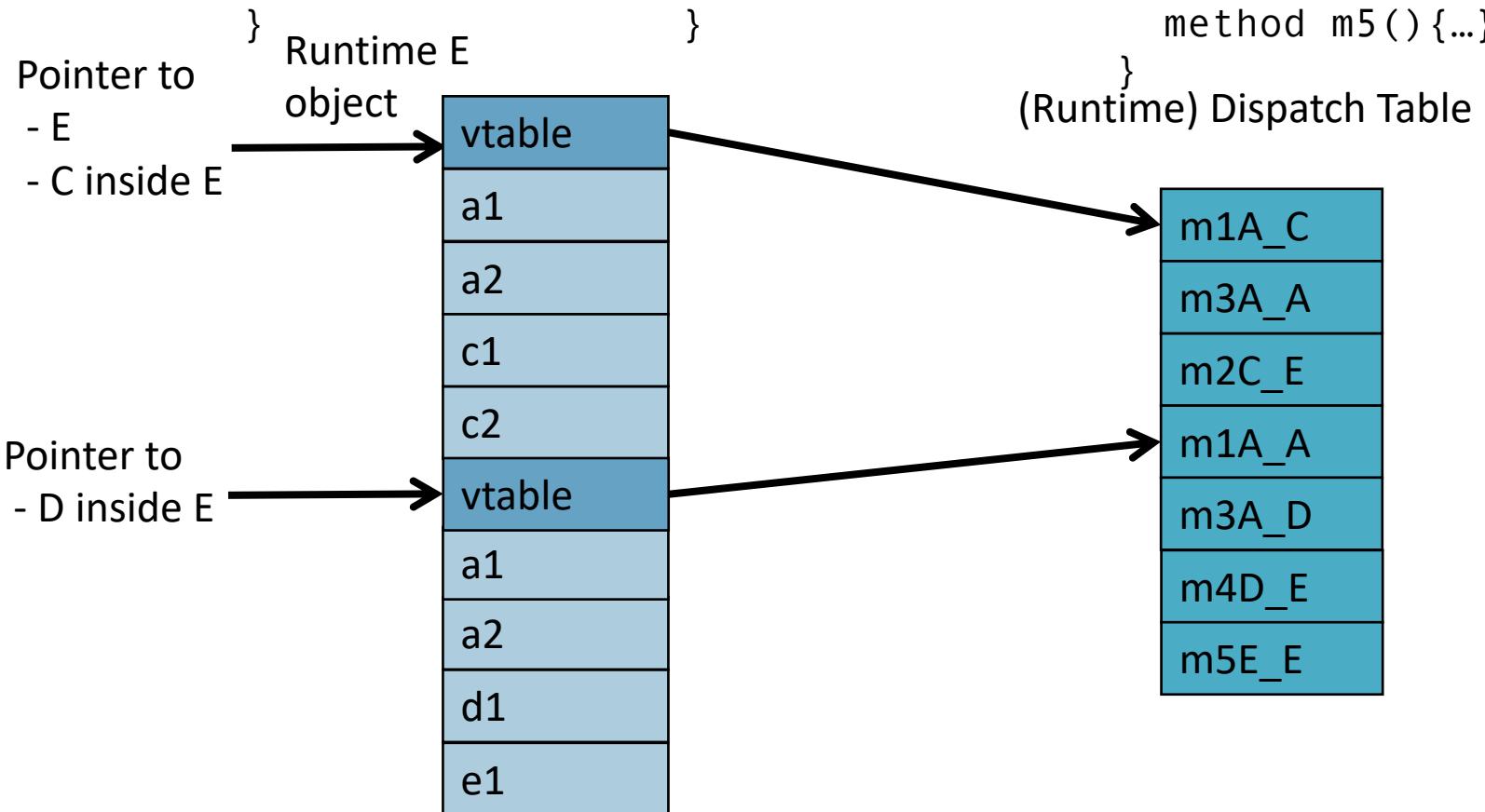
```
class E extends C, D {  
    field e1;  
    method m3() {...} //alt explicit qualification  
    method m2() {...}  
    method m4() {...}  
    method m5() {...}  
}
```

```
class A{  
    field a1;  
    field a2;  
    method m1() {...}  
    method m3() {...}  
}
```

```
class D extends A {  
    field d1;  
    method m3() {...}  
    method m4() {...}  
}
```

Independent Inheritance

```
class A{  
    field a1;  
    field a2;  
    method m1() {...}  
    method m3() {...}  
}  
  
class C extends A{  
    field c1;  
    field c2;  
    method m1() {...}  
    method m2() {...}  
}  
  
class D extends A{  
    field d1;  
    method m3() {...}  
    method m4() {...}  
}  
  
class E extends C,D{  
    field e1;  
    method m2() {...}  
    method m4() {...}  
    method m5() {...}  
}
```



Dependent multiple inheritance

- Superclasses share their own superclass
- The simple solution does not work
- The positions of nested fields do not agree

Dependent multiple Inheritance

```
class A{  
    field a1;  
    field a2;  
    method m1() {...}  
    method m3() {...}  
}  
  
class C extends A {  
    field c1;  
    field c2;  
    method m1() {...}  
    method m2() {...}  
}  
  
class D extends A {  
    field d1;  
    method m3() {...}  
    method m4() {...}  
}  
  
class E extends C, D {  
    field e1;  
  
    method m2() {...}  
    method m4() {...}  
    method m5() {...}  
}
```

Dependent multiple Inheritance

```
class A{  
    field a1;  
    field a2;  
    method m1() {...}  
    method m3() {...}  
}  
  
class C extends A {  
    field c1;  
    field c2;  
    method m1() {...}  
    method m2() {...}  
}  
  
class D extends A {  
    field d1;  
    method m3() {...}  
    method m4() {...}  
}  
  
class E extends C, D {  
    field e1;  
  
    method m2() {...}  
    method m4() {...}  
    method m5() {...}  
}
```

Dependent multiple Inheritance

```
class A{
    field a1;
    field a2;
    method m1() {...}
    method m3() {...}
}

class C extends A {
    field c1;
    field c2;
    method m1() {...}
    method m2() {...}
}

class D extends A {
    field d1;
    method m3() {...}
    method m4() {...}
}

class E extends C, D {
    field e1;

    method m2() {...}
    method m4() {...}
    method m5() {...}
}
```

Dependent Inheritance

- Superclasses share their own superclass
- The simple solution does not work
- The positions of nested fields do not agree

Implementation

- Use an index table to access fields
- Access offsets indirectly

Implementation

```
class A{  
    field a1;  
    field a2;  
    method m1() {...}  
    method m3() {...}  
}
```

```
class C  
    extends A{  
        field c1;  
        field c2;  
        method m1() {...}  
        method m2() {...}  
    }
```

```
class D  
    extends A{  
        field d1;  
        method m3() {...}  
        method m4() {...}  
    }
```

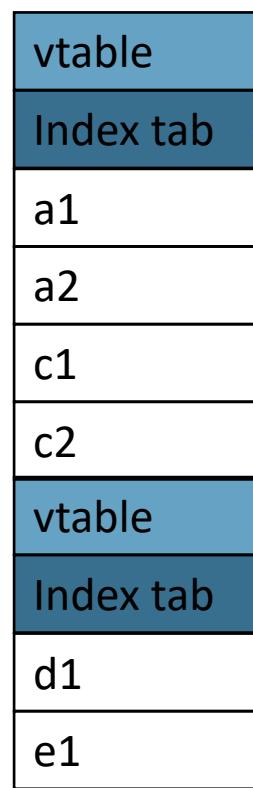
```
class E  
    extends C,D{  
        field e1;  
        method m2() {...}  
        method m4() {...}  
        method m5() {...}  
    }
```

Runtime E
object

Pointer to
- E
- C inside E



Pointer to
- D inside E



}
(Runtime) Dispatch Table



Index
tables
73

Class Descriptors

- Runtime information associated with instances
- Dispatch tables
 - Invoked methods
- Index tables
- Shared between instances of the same class
- Can have more (reflection)

Interface Types

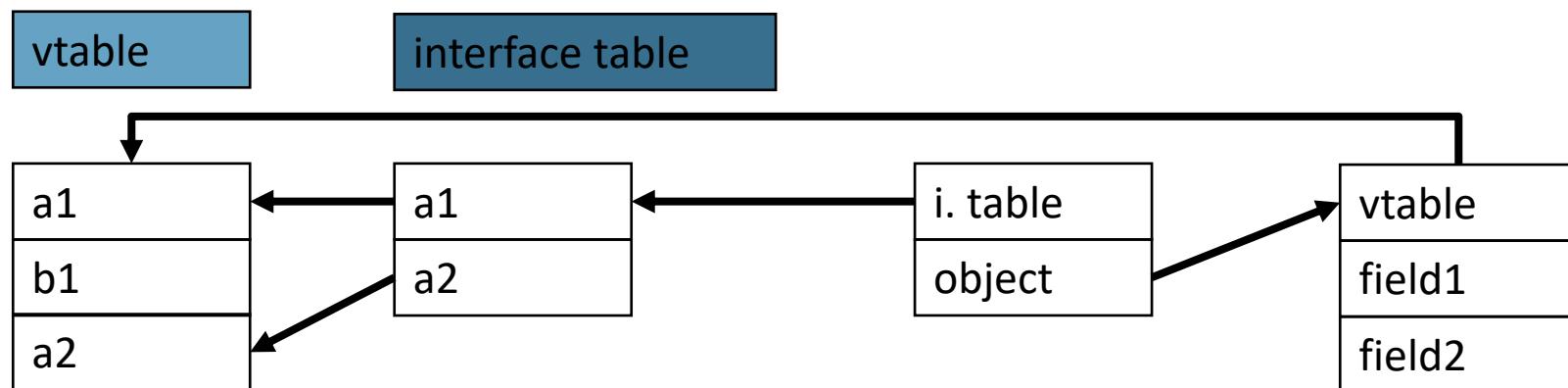
- Java supports limited form of multiple inheritance
- Interface consists of several methods but no fields

```
public interface Comparable {  
    public int compare(Comparable o);  
}
```

- A class can implement multiple interfaces
Simpler to implement/understand/use

Interface Types

- Implementation: record with 2 pointers:
 - A separate dispatch table per interface
 - A pointer to the object



Dynamic Class Loading

- Supported by some OO languages (Java)
- At compile time
 - the actual class of a given object at a given program point may not be known
- Some addresses have to be resolved at runtime
 - problem when multiple inheritance is allowed
 - Can use hash table to map fields name to pointers to functions

Other OO Features

- Information hiding
 - private/public/protected fields
 - Semantic analysis (context handling)
- Testing class membership
 - Single inheritance: look up the chain of “supers”
 - If inheritance is bounded
 - Allocate parent array at every class descriptors
 - look up at “depth-of-inheritance” index
- Up casting: (Possibly) runtime checks
 - $B\ x = (B)a$ // B extends A

Optimizing OO languages

- Hide additional costs

- Eliminate runtime checks

- ```
A x = new B() // B extends A
```

- ```
B y = (B) x
```

- Eliminate dead fields

- Replace dynamic by static binding when possible

- ```
x = new B() // B extends A and overrides f
```

- ```
x.f() // invoke B_f(x)
```

- Type propagation analysis ~ reaching definitions

Optimizing OO languages

- Simultaneously generate code for multiple classes

```
class A {      class B extends A {      class B extends A {  
    g() {}        g() {}        g()  
    f() { g() }    }          }  
}                }
```

- Generate code `B_f()` and `C_f()`
- Code space is an issue

Summary

- OO is a programming/design paradigm
- OO features complicates compilation
 - Semantic analysis
 - Code generation
 - Runtime
 - Memory management
- Understanding compilation of OO can be useful for programmers