

CS558 Programming Languages

Winter 2008

Lecture 16

REPRESENTATION OF OBJECTS

In an naive **interpreted** implementation, each object is represented by a heap-allocated record, containing

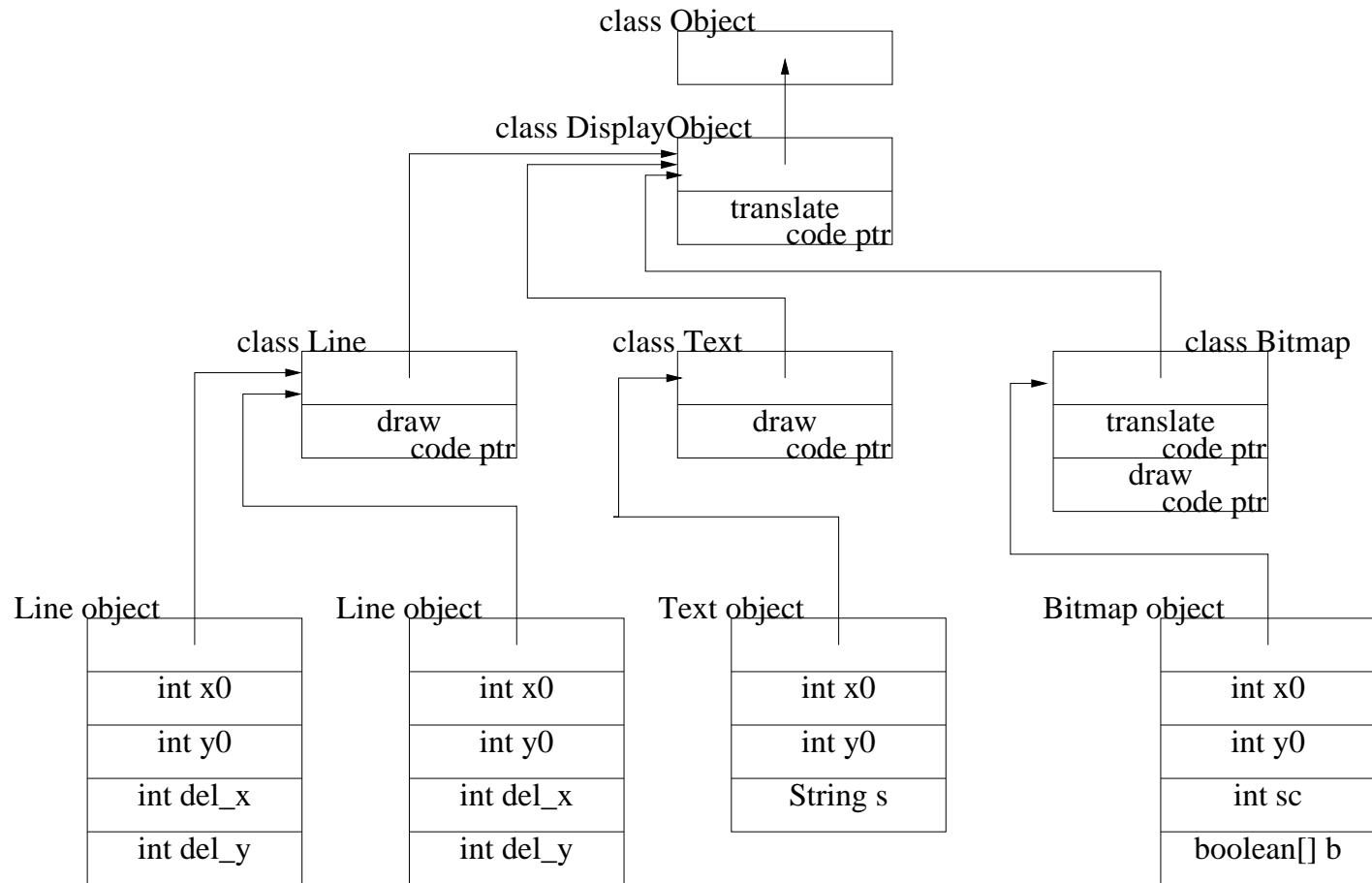
- Name and values of each instance variable.
- Pointer to class description record.

Each class is represented by a (essentially static) record with:

- Name and code pointer for each class method.
- Pointer to super-class's record.

EXAMPLE

(based on the code from slides 12,13,17 of previous lecture)



INTERPRETED IMPLEMENTATION OF OPERATIONS

To perform a message **send** (function call) at runtime, the interpreter does a method lookup, starting from the receiver object, as follows:

- Use class pointer to find class description record.
- Search for method in class record. If found, invoke it; otherwise, continue search in superclass record.
- If no method found, issue “Message Not Understood” error. (Can’t happen in strongly-typed languages; more later.)

Instance variables are accessed in the object record. Pseudo-variable `this` always points to the receiver object record; `super` always points to the superclass.

How about “compiling” OO languages?

Dynamic binding makes compilation difficult:

- method code doesn't know the precise class to which the object it is manipulating belongs,
- nor the precise method that will execute when it sends a message.

Instance variables are not so hard.

- Code that refers to instance variables of a given class will actually operate on objects of that class or of a subclass.
- Since a subclass always **extends** the set of instance variables defined in its superclass, compiler can consistently assign each instance variable a fixed (static) offset in the object record; this offset will be the same in every object record for that class and any of its subclasses.
- Compiled methods can then reference variables by offset rather than by name, just like ordinary record field offsets.

(Multiple inheritance schemes cause problems.)

COMPILATION (CONT.)

Handling message sends is harder, because methods can be overridden by subclasses.

Simple approach: keep a per-class static **method table** (or **vtable**) and “compile” message sends into indirect jumps through fixed offsets in this table.

Example: Classes in slides 12,13,17 of previous lecture all have this vtable structure:

```
-----  
(offset 0) | draw code ptr.      |  
-----  
(offset 1) | translate code ptr. |  
-----
```

These tables can get large, and much of their contents will be duplicated between a class and its superclasses. Still, this approach is used by C++, Java. (Again, multiple inheritance – and Java interfaces – cause complications.)

C++ VALUE MODEL AND INHERITANCE

Unlike Java, C++ represents objects as unboxed records.

So in declarations like this

```
DisplayObject x(10,20);  
Line y(30,40,2,4);
```

variables `x` and `y` represent actual instance records, **not** pointers to records. In order to allocate storage space for such instances, the compiler uses the **declared type** of the variable, so `x` has space for only `x0` and `y0`, not `del_x` and `del_y`.

Now, an assignment like

```
... x = y; ...
```

works by **truncating** the `Line` to fit in the space of a `DisplayObject`; the `del_x` and `del_y` fields are lost.

Not likely to be what you wanted!

C++ POINTERS

So, in practice, C++ programmers routinely use explicit **pointers** to objects instead:

```
DisplayObject *x = new DisplayObject(10,20);  
Line *y = new Line(30,40,2,4);
```

Now assignment

```
... x = y; ...
```

is a pointer assignment rather than a record copy.