Object-Oriented Languages

Object-oriented languages extend imperative languages with:

1. A classification scheme that allows us to specify is-a as well as has-a relationships. Has-a is supported by Pascal, where we can declare that one data item has another item (a record variable has-a record field). Object-Pascal, Oberon, etc, extends this capability with inheritance which allows us to state that one data item is (an extension of) another item.

2. Late binding, which allows us to select between different implementations of the same abstract data type at run-time.

Object-Oriented Languages...

3. Polymorphism, which is the ability of a variable to store values of different types. OO languages support a special kind of polymorphism, called inclusion polymorphism, that restricts the values that can be stored in a variable of type $T$ to values of type $T$ or subtypes of $T$.

4. Data encapsulation. Data (instance variables) and operations (methods) are defined together.

5. Templates and objects. A template (class or prototype) describes how to create new objects (instances of abstract data types).

Compiling OO Languages

- Runtime type checking (a variable of type $\text{ref } T$ may only reference objects of type $T$ or $T$’s subtypes).

- Because of the polymorphic nature of OO languages, we can’t always know (at compile-time) the type of the object that a given variable will refer to at run-time. When we invoke a method we can’t actually know which piece of code we should execute. Finding the right piece of code is called method lookup. It can be done by name (Objective-C) or number (C++).

- Most OO languages rely on dynamic allocation. Garbage collection is a necessary part of the runtime system of a compiler for an OO language (C++ non-withstanding). This requires runtime type description.
Object-Oriented Example

TYPE Shape = CLASS
  x, y : REAL;
  METHODS
    draw() : BEGIN ... END;
    move(X, Y:REAL) : BEGIN x := x+X; END;
END;
TYPE Square = Shape CLASS
  side : REAL;
  METHODS
    draw() : BEGIN ... END;
END;
TYPE Circle = Shape CLASS
  radius : REAL;
  METHODS
    draw() : BEGIN ... END;
    area() : REAL; BEGIN ... END;
END;

Example in Java

// Example in Java

class Shape {
  double x, y;
  void draw(); { ... }
  void move(double X, double Y); { x = x+X; }
}
class Square extends Shape {
  double side;
  void draw(); { ... }
  double area(); { ... }
}

Example in Modula-3 (A)

(* Example in Modula-3 *)
TYPE Shape = OBJECT
  x, y : REAL
  METHODS
    draw() := DefaultDraw; move(X, Y : REAL) := Move;
END;
Square = Shape OBJECT
  side : REAL
  METHODS
    draw() := SquareDraw
END;
Circle = Shape OBJECT
  radius : REAL
  METHODS
    draw() := CircleDraw; area() := ComputeArea
END;

Example in Modula-3 (B)

(* Example in Modula-3 (continued) *)
PROCEDURE Move (Self : Shape; X, Y : REAL) =
BEGIN ... END Move;
PROCEDURE DefaultDraw (Self : Shape) =
BEGIN ... END DefaultDraw;
PROCEDURE SquareDraw (Self : Square) =
BEGIN ... END SquareDraw;
PROCEDURE CircleDraw (Self : Circle) =
BEGIN ... END CircleDraw;
PROCEDURE ComputeArea (Self : Circle) : REAL =
BEGIN ... END ComputeArea;
Example in Oberon-2

TYPE  Shape = RECORD x, y : REAL END;
Square = RECORD (Shape) side : REAL END;
Circle = RECORD (Shape) radius : REAL END;
PROCEDURE (Self : Shape) Move (X, Y : REAL) =
BEGIN ... END Move;
PROCEDURE (Self : Shape) DefaultDraw () =
BEGIN ... END DefaultDraw;
PROCEDURE (Self : Square) SquareDraw () =
BEGIN ... END SquareDraw;
PROCEDURE (Self : Circle) CircleDraw () =
BEGIN ... END CircleDraw;
PROCEDURE (Self : Circle) ComputeArea () : REAL =
BEGIN ... END ComputeArea;

Record Layout

Single inheritance is implemented by concatenation, i.e.
the instance variables of class $C$ are
1. the variables of $C$'s supertype, followed by
2. the variables that $C$ declares itself.

Record Layout

The offsets of the variables that $C$ inherits from its
supertype will be the same as in the supertype itself.
In this example, $C_3$ inherits from $C_2$ which inherits from
$C_1$.
$C_3$ will have the fields from $C_1$ followed by the fields from
$C_2$ followed by $C_3$'s own fields. The order is significant.

Record Layout

TYPE  Shape = RECORD x, y : REAL END;
CLASS x, y : REAL; END;
TYPE  Square = Shape
CLASS side : REAL; END;
TYPE  Circle = Shape
CLASS radius : REAL; END;
VAR  S:Shape;
VAR  Q:Square;
VAR  C:Circle;
Record Layout...

An OO language compiler would translate the declarations in the previous slide into something similar to this:

```plaintext
TYPE Shape = POINTER TO RECORD
  x, y: REAL;
END;
TYPE Square = POINTER TO RECORD
  x, y: REAL;
  side: REAL;
END;
TYPE Circle = POINTER TO RECORD
  x, y: REAL;
  radius: REAL;
END;
VAR S: Shape; Q: Square; C: Circle;
```

Class Templates...

Square’s x, y fields are inherited from Shape. Their offsets are the same as in Shape.

```plaintext
TYPE $TemplateT = POINTER TO RECORD
  parent: $TemplateT;
  move: ADDRESS;
  draw: ADDRESS;
END;
TYPE Square = POINTER TO RECORD
  $template: $TemplateT;
  x, y: REAL;
  side: REAL;
END;
CONST Square$Template: $TemplateT = [ parent= ADDR(Shape$Template);
  move = ADDR(Shape$move);
  draw = ADDR(Square$draw); ];
```

Class Templates...

To support late binding, runtime type checking, etc, each class is represented by a template at runtime. Each template has pointers to the class’s methods and supertype.

```plaintext
PROCEDURE Shape$move (SELF : Shape; X, Y: REAL);
BEGIN
  SELF^.x := SELF^.x + X;
  SELF^.y := SELF^.y + X;
END;
```

Class Templates...

Each method is a procedure with an extra argument (SELF), a pointer to the object through which the method was invoked.

```plaintext
TYPE Shape = CLASS
  x, y : REAL;
END;
METHOD draw (); BEGIN ...;
METHOD move (X, Y : REAL);
BEGIN x := x+X; ... END;
```
Method Invocation

Sending the message draw to Q:
1. Get Q’s template, T.
2. Get draw’s address at offset 4 in T.
3. Jump to draw’s address, with Q as the first argument.

Inclusion Polymorphism

Consider the last two lines of the example in the following slide:

- In L₁, S points to a Shape object, but it could just as well have pointed to an object of any one of Shape’s subtypes, Square and Circle.
- If, for example, S had been a Circle, the assignment C := S would have been perfectly OK. In L₂, however, S is a Shape and the assignment C := S is illegal (a Shape isn’t a Circle).
Typechecking Rules

TYPE
T = CLASS Φ END;
U = T CLASS Φ END;
S = T CLASS Φ END;

VAR
t, r : T; u : U; s : S;

- A variable of type T may refer to an object of T or one of T's subtypes.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Compile-time</th>
<th>Run-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>t := r;</td>
<td>Legal</td>
<td>Legal</td>
</tr>
<tr>
<td>t := u;</td>
<td>Legal</td>
<td>Legal</td>
</tr>
<tr>
<td>u := t;</td>
<td>Legal</td>
<td>Check</td>
</tr>
<tr>
<td>s := u;</td>
<td>Illegal</td>
<td></td>
</tr>
</tbody>
</table>

Run-time Type Checking

Modula-3 Type-test Primitives:

ISTYPE(object, T) Is object's type a subtype of T?
NARROW(object, T) If object's type is not a subtype of T, then issue a run-time type error. Otherwise return object, typecast to T.
TYPECASE Expr OF Perform different actions depending on the runtime type of Expr.

- The assignment s := t is compiled into s := NARROW(t, TYPE(s)).

Run-time Type Checking...

- The Modula-3 runtime-system has three functions that are used to implement typetests, casts, and the TYPECASE statement
- NARROW takes a template and an object as parameter. It checks that the type of the object is a subtype of the type of the template. If it is not, a run-time error message is generated. Otherwise, NARROW returns the object itself.

1. ISTYPE(S,T : Template) : BOOLEAN;
2. NARROW(Object, Template) : Object;
3. TYPECODE(Object) : CARDINAL;

Run-time Checks

- Casts are turned into calls to NARROW, when necessary:

VAR S : Shape; VAR C : Circle;
BEGIN
 S := NEW (Shape); C := S;
END;

VAR S : Shape; VAR C : Circle;
BEGIN
 S := malloc (SIZE(Shape));
 C := NARROW(S, Circle$Template);
END;
Implementing ISTYPE

We follow the object’s template pointer, and immediately (through the templates’ parent pointers) gain access to its place in the inheritance hierarchy.

PROCEDURE ISTYPE (S, T : TemplatePtr) : BOOLEAN;
BEGIN
  LOOP
    IF S = T THEN RETURN TRUE; ENDIF;
    S := S^ parent;
    IF S = ROOT THEN RETURN FALSE; ENDIF;
  ENDLOOP
END ISTYPE;

Implementing NARROW

NARROW uses ISTYPE to check if S is a subtype of T. Of so, S is returned. If not, an exception is thrown.

PROCEDURE NARROW(T:TemplatePtr; S:Object):Object;
BEGIN
  IF ISTYPE(S^.template, T) THEN
    RETURN S (* OK *);
  ELSE WRITE "Type error"; HALT;
  ENDIF;
END NARROW;

Run-time Checks — Example

TYPE T = CLASS [⋯];
S = T CLASS [⋯];
U = T CLASS [⋯];
V = U CLASS [⋯];
X = S CLASS [⋯];
Y = U CLASS [⋯];
Z = U CLASS [⋯];
VAR x : X;

Run-time Checks — Example...

ISTYPE(x, T) →

ISTYPE(S, T) →

ISTYPE(U, T) →

ISTYPE(V, T) →

ISTYPE(Z, T) →
Run-time Checks – An $O(1)$ Algorithm

The time for a type test is proportional to the depth of the inheritance hierarchy. Two algorithms do type tests in constant time:

1. Norman Cohen, “Type-Extension Type Tests can be Performed in Constant Time.”
2. Paul F. Dietz, “Maintaining Order in a Linked List”.

The second is more efficient, but requires the entire type hierarchy to be known. This is a problem in separately compiled languages.

SRC Modula-3 uses Dietz’ method and builds type hierarchies of separately compiled modules at link-time.

These algorithms only work for single inheritance.

Run-time Checks – Alg. II (b)

In the Compiler (or Linker):

1. Build the inheritance tree.
2. Perform a preorder traversal and assign preorder numbers to each node.
3. Similarly, assign postorder numbers to each node.
4. Store T’s pre- and postorder numbers in T’s template.

In the Runtime System:

PROCEDURE ISTYPE (S, T : TemplatePtr) : BOOLEAN;
BEGIN
RETURN (T.pre ≤ S.pre) AND (T.post ≥ S.post);
END ISTYPE;

Run-time Checks – Alg. II (c)

\[
\begin{align*}
T & = \text{CLASS} \quad \cdots; \\
S & = T \quad \text{CLASS} \quad \cdots; \\
U & = T \quad \text{CLASS} \quad \cdots; \\
V & = U \quad \text{CLASS} \quad \cdots; \\
X & = S \quad \text{CLASS} \quad \cdots; \\
Y & = U \quad \text{CLASS} \quad \cdots; \\
Z & = U \quad \text{CLASS} \quad \cdots;
\end{align*}
\]

\[
\begin{align*}
\sqrt{\text{ISTYPE}(Y,U)} & \quad U.pre \leq Y.pre \quad U.post \geq Y.post \\
\text{ISTYPE}(Z,S) & \quad S.pre \leq Z.pre \quad S.post \geq Z.post \\
\sqrt{\text{ISTYPE}(Z,T)} & \quad T.pre \leq Z.pre \quad T.post \geq Z.post
\end{align*}
\]

Run-time Checks – Alg. II (d)

Consider U:

1. U’s pre-number is $\leq$ all it’s children’s pre numbers.
2. U’s post-number is $\geq$ all it’s children’s post numbers.

\([U.pre, U.post] \text{ “covers” (in the sense that } U.pre \leq \text{ pre and } U.post \geq \text{ post) the [pre,post] of all it’s children.}

S is not a subtype of U since \([U.pre, U.post] \text{ does not cover } [S.pre, S.post] (S.post \leq U.post \text{ but } S.pre \not\geq U.pre)\).
Multiple Inheritance

In some languages (C++, Eiffel) a class can have more than one superclass.

```java
class Person { Name : STRING; }
class Student extends Person {
    Advisor : Teacher;
}
class Teacher extends Person {
    Salary : INTEGER;
    method Rich () : BOOLEAN;
        return Salary > 50000;
}
class Tutor extends Student, Teacher {
    Boss : Teacher;
}
```

Multiple Inheritance...

We’d like to be able to call `m.Rich()` for any `Teacher` object, including a `Tutor`:

```java
PROCEDURE Rich (SELF : Teacher) : BOOLEAN;
    RETURN SELFˆ^Salary > 50000;
```

Teacher Knuth = new Teacher;
Tutor Lucy = new Tutor;
boolean k = Knuth.Rich();
boolean l = Lucy.Rich();

In order for this to work, the `Salary` field in a `Tutor` record must be at the same offset as the `Salary` field in the `Teacher` record.

Multiple Inheritance...

But, if our record layout uses simple concatenation of parent classes (like with single inheritance), we get:

```java
Person
  0:Name:STRING

Student
  0:Name:STRING
  4:Advisor:Teacher

Teacher
  0:Name:STRING
  4:Salary:INT

Tutor
  0:Name:STRING
  4:Advisor:Teacher
  8:Salary:INT

From Student
  0:Name:STRING
  4:Advisor:Teacher
  0:Salary:INT
```

The `Salary` field in a `Teacher` record is at offset 4, but the `Salary` field in the `Tutor` record is at offset 8.
Multiple Inheritance...

An inefficient implementation might do:

```
PROCEDURE Rich (SELF : Teacher) : BOOLEAN;
  RETURN IF ISTYPE(SELF,Teacher)
    THEN (SELF-4)^>50000 ELSE (SELF+8)^>50000;
```

Or we could insert extra space to align the fields properly:

```
Person
  0:Name:String
  4:Advisor:Teacher
  8:Salary:INT

Student
  0:Name:String
  4:Advisor:Teacher
  12:Wasted:4-bytes
  16:Salary:INT

Teacher
  0:Name:String
  4:Wasted:4-bytes

Tutor
  0:Name:String
  4:Advisor:Teacher
  8:Salary:INT
```

With multi-directional layouts, we place variables at both positive and negative offsets:

```
Person
  0:Name:String
  4:Advisor:Teacher
  8:Salary:INT

Student
  0:Name:String
  4:Advisor:Teacher
  12:Wasted:4-bytes
  16:Salary:INT

Teacher
  0:Name:String
  4:Wasted:4-bytes

Tutor
  0:Name:String
  4:Advisor:Teacher
  8:Salary:INT
```

How does the language deal with the same field inherited through more than one path? A Tutor inherits Name twice, once from Student and once from Teacher:

```
class Person { Name : STRING; }
class Student extends Person {...}
class Teacher extends Person {...}
class Tutor extends Student,Teacher {...}
```

The Salary-field is always at the same offset, regardless of what type of object:

```
PROCEDURE Rich (SELF : Teacher) : BOOLEAN;
  RETURN (SELF-4)^>50000;
```

Should Tutor have one or two copies of Name?

In Trellis/Owl you always get just one copy of Name.

In C++ you can choose. If you declare a superclass virtual, Tutor only gets one copy of Name, otherwise two.
Multiple Inheritance...

How does the language deal with different fields/methods with the same type/signature inherited from different classes?

```plaintext
class Student {Name : STRING; ... }
class Teacher {Name : STRING; ... }
class Tutor extends Student, Teacher {...}
Tutor T = new Tutor();
T.Name = "Knuth"; /* Which Name? */
```

In Eiffel, the programmer has to rename fields until there are no more conflicts, using a `rename` clause:

```plaintext
class Tutor extends Student,
                   Teacher rename Name⇒TName {...}
```

In C++, conflicts are resolved when the field/method is used:

```plaintext
Tutor T = new Tutor();
Teacher::T.Name = "Knuth";
```

Exam Problem

In the following object-oriented program

- "$\text{TYPE } U = T \text{ CLASS}\"$ means that $U$ inherits from $T$.
- $\text{NEW } T$ means that a new object of type $T$ is created.
- All methods are virtual, i.e. a method in a subclass overrides a method with the same name in a superclass.

```plaintext
PROGRAM X;
TYPE T = CLASS [
  v : INTEGER; c : CHAR;
  METHOD P (x:INTEGER); BEGIN ... END P;
  METHOD Q (x:CHAR); BEGIN ... END Q;
];
VAR t : T; u : U;
BEGIN
  t := NEW T; u := NEW U; ◇
END
```

1. Draw a figure that describes the state of the program at point ◇. It should have one element for each item stored in memory (i.e. global/heap variables, templates, method object code, etc.) and should explicitly describe what each pointer points to.
Read Scott: 529–551, 554–561, 564–573
For information on constructing layouts for multiple inheritance, see
William Pugh and Grant Weddell: “Two-directional record layout for multiple inheritance.”
The time for a type test is proportional to the depth of the inheritance hierarchy. Many algorithms do type tests in constant time:
1. Norman Cohen, “Type-Extension Type Tests can be Performed in Constant Time.”
2. Paul F. Dietz, “Maintaining Order in a Linked List”.

For single inheritance languages, an instance of a class \( C \) consists of (in order):
1. A pointer to \( C \)’s template.
2. The instance variables of \( C \)’s ancestors.
3. \( C \)’s instance variables.

For single inheritance languages, subtype checks can be done in \( O(1) \) time.
Method invocation is transformed to an indirect call through the template.
If we can determine the exact type of an object variable at compile time, then method invocations through that variable can be turned into “normal” procedure calls.

A template for class \( C \) consists of (in order):
1. A pointer to the template of \( C \)’s parent.
2. The method addresses of \( C \)’s ancestors.
3. Addresses of \( C \)’s methods.
4. Other information needed by the runtime system, such as
   - The size of a \( C \) instance.
   - \( C \)’s pre- and postorder numbers, if the \( O(1) \) subtype test algorithm is used.
   - \( C \)’s type code.
   - A type description of \( C \)’s instance variables. Needed by the garbage collector.

What happens when both a class and its subclass have an instance variable with the same name?
The subclass gets both variables. You can get at both of them, directly or by casting. Here’s an example in Java:

```java
class C1 {int a;}
class C2 extends C1 {double a;}
class C {
    static public void main(String[] arg) {
        C1 x = new C1(); C2 y = new C2();
        x.a = 5; y.a = 5.5;
        ((C1)y).a = 5;
    }
}
```