Object-oriented Programming

Object-oriented Languages

object-based: language that supports objects

class-based: language that supports classes

object-oriented: object-based language that additionally supports

the definition of abstract data types,
inheritance,
inclusion polymorphism (subtyping),
dynamic binding of routine invocations to routines

terminology:

objects are instances of classes,
objects contain object/instance variables and support methods,
messages are sent to objects

Derived Class in C++

```
class stack {
   public:
      void push(int) { elements[top++] = i; }
      int pop() { return elements[--top]; }
   private:
      int elements[100];
      int top = 0;
};

class counting_stack : public stack {
    public:
      int size(); //return number of elements on the stack
};
```

Polymorphism and Dynamic Binding

```
stack s;                    // not polymorph
counting_stack cs;
stack* sp = new stack();  //polymorph
counting_stack* csp = new counting_stack();
...
sp = csp;          // supported
csp = sp;          // not supported; statically unknown
s = cs;            // supported; converted to stack
cs = s;            // not supported; cs.size() undefined
...
sp->push(...);  //stack::push
csp->push(...); //counting_stack::push
sp = csp;
sp->push(...);  //which push?
```

Subclass Versus Subtype

principle of substitutability: an instance of a subtype can be used where an instance of a supertype is expected

subclass is a subtype if (simplified):

- object variables and methods are only added (not removed)
- and methods are redefined only in a compatible way:
  - they have the same or a compatible signature
    (statically checkable, but not always sufficient)
  - and methods behave equivalently
    (not statically checkable)

Strong Types and Polymorphism

```
class base {...};
class derived: public base {...};
...
base* b;
derived *d;
```

under which condition can an assignment `b = d` not lead to a type error?

example: `derived` is subtype of `base` if `derived` can be used wherever `base` is expected such that no type error can arise

are these conditions statically checkable?

Method Overriding

```
class polygon {
   public:
      polygon(...) {...}  // constructor
      virtual float perimeter() {...}; //virtual
      ...
}

class square: public polygon {
   public:
      square(...) {...}   // constructor
      float perimeter() {...};  // virtual, overridden
      ...
}
```

Contra-variant Parameter Types

```
class base {
   public:
      virtual void fnc(s1 par) {...}
};
class derived: public base {
   public:
      void fnc(s2 par) {...}
};
...
    base* b;
    derived* d;
    s1 v1;
    s2 v2;
    if(...) b = d;
    b->fnc(v1);    // what if b is of type derived?
```

Co-variant Result Types

```
class base {
   public:
      virtual t1 fnc(...) {...}
};
class derived: public base {
   public:
      t2 fnc(...) {...}
};
...
    base* b;
    derived* d;
    t0 v0;
    if(...) b = d;
    v0 = b->fnc(...);  // what if b is of type derived?
```

Co-variant Problem

```
class point {
    public:
        float x, y;
        bool equal (point p)
            { return x == p.x && y == p.y; }
};

class colorPoint : public point {
    public:
        int color;
        bool equal (colorPoint p)  // error, co-variant
            { return x == p.x && y == p.y
                              && color == p.color; }
};
```

Characteristics of C++

constructors initialize objects (base class first)

destructors clean up (base class last)

automatic object: automatically allocated and deallocated on stack

objects on heap explicitly allocated and deallocated;
corresponding functions `new` and `delete` are programmable

static variables shared by all objects of class, static allocation

assignment and equality operations programmable by overloading of = and ==

Virtual Funtions in C++

```
class student {
   public:
      virtual void print() {...}
};
class college_student: public student {
   void print() {...}
};
...
    student* s;
    college_student* cs;
    ...
    s->print();  // calls student::print()
    s = sc;      // okay
    s->print();  // calls college_student::print()
```

Pure Virtual Funtions in C++

```
class shape {
    public:
        void draw() = 0;  // pure virtual function
        void move(...) = 0;
        void hide() = 0;
        point center;
};

class rectangle : public shape {
    private:
        float length, width;
    public:
        void draw() { ... };  // impl. of derived function
        void move(...) { ... };
        void hide() { ... };
};
```

Inheritance and Visibility in C++

```
class stack {
    public:
        stack() { top = 0 }; // constructor
        void push(int i) { s[top++] = i; };
        int pop() { return s[--top]; };
    protected:
        int top;
    private:
        int s[100];
};

class counting_stack : public stack {
    public:
        int size() { return top; };
};
```

Polymorphism in C++

genericity (= parametric polymorphism) by templates
— statically resolved (by heterogeneous translation)

inclusion polymorphism supported by multiple inheritance
— statically and dynamically resolved

overloading of functions and operators
— statically resolved

explicit and implicit casts, run-time type information (rtti)
— semantic action at run-time

Package and Tagged Record in Ada

```
package Planan_Objects is
    type Planar_Object is tagged
        record
            X, Y: Float := 0.0;
        end record;
    function Distance (O: Planar_Object) return Float;
    procedure Move (O: in out Planar_Object; DX, DY: Float);
    procedure Draw (O: Planar_Object);
end Planar_Objects;
```

Extending Tagged Record in Ada

```
with Planar_Objects; use Planar_Objects;
package Special_Planar_Shapes is
    type Point in new Planar_Object with null record;
    procedure Draw (P: Point);

    type Circle is new Planar_Object with
        record
            Radius: Float;
        end record;
    procedure Draw (C: Circle);

    type Rectangle is new Planar_Object with
        record
            Length, Width: Float;
        end record;
    procedure Draw (T: Rectangle);
end Special_Planar_Shapes;
```

Use of Tagged Records in Ada

```
O1: Planar_Object;
O2: Planar_Object := (1.0, 1.0);
C: Circle := (3.0, 4.5, 6.7);
...
O1 := Planar_Object(C);   // type cast
C := (O2 with 6.7)

procedure Process_Shapes (O: Planar_Object'Class) is
    ...
begin
    ... Draw(O); ...   // dynamic binding
end Process_Shapes;

type Planar_Object_Ptr is access Planar_Object'Class;
```

Abstract Type in Ada

```
package Planan_Objects is
    type Planar_Object is abstract tagged null record;
    function Distance (O: Planar_Object) return Float
        is abstract;
    procedure Move (O: in out Planar_Object; DX, DY: Float)
        is abstract;
    procedure Draw (O: Planar_Object) is abstract;
end Planar_Objects;
```

Polymorphism in Ada

supports genericity including bounded genericity
— statically resolved

inclusion polymorphism supported by single inheritance
— statically and dynamically resolved (by class-wide types)

overloading of routines and operators
— statically resolved

only explicit casts, type comparison at run-time
— semantic actions at run-time

Classes and Inheritance in Eiffel

```
class A feature
    fnc(t: TI): TO is ... do ... end -- fnc
end -- class A

class B inherit A redefine fnc feature
    fnc(s: SI): SO is ... do ... end -- fnc
    -- supports co-variant parameter types, not type-safe
end -- class B


    ...
    a: A;
    b: B;
    ...
    create b;           -- old syntax: !!b;
    a := b;
    ... a.fnc(x) ...    -- dynamic binding
```

Polymorphism in Eiffel

supports genericity including bounded genericity
— statically resolved

inclusion polymorphism supported by multiple inheritance
— dynamically resolved (expanded objects statically)
— co-variant parameter types can result in exceptions

overloading of routines and operators
— statically resolved

checked assignment attempt (`x ?= y`)
— semantic action at run-time

Characteristics of Smalltalk

classes are objects that specify variables and methods of their instances

variables are untyped references $\longrightarrow$ natural polymorphism

single inheritance; methods corresponding to a messages are dynamically searched from bottom to top in class hierarchy

syntax: object followed by message; examples:

declaration: `value`	call: `setZero value`	(unary)
declaration: `> param`	call: `x > y`	(binary)
declaration: `from: f to: t`	call: `s from: x to: y`	(keyword)

objects are created explicitly (`myPoint <- point new`),
deallocation by garbage collector

Examples in Smalltalk

```
[x <- 0. y <- 0] value

setZero <- [x <- 0. y <- 0].
setZero value

x > y
   ifTrue:  [max <- x]
   ifFalse: [max <- y]

10 timesRepeat: [x <- x + a]

[x < b] whileTrue: [x <- x + a]
```

Polymorphism in Smalltalk

genericity not necessary because variables have no declared type
— dynamically resolved as side-effect of dynamic binding

inclusion polymorphism supported independent from inheritance
— dynamically resolved by method search

no overloading

cast not necessary, type comparison at run-time
— semantic actions at run-time

Characteristics of Java

all objects allocated on heap, deallocated by garbage collector

programs translated to portable JVM code
(idea was to use Java as language in networks, but rarely used that way)

single inheritance of classes and multiple inheritance of interfaces

variables and methods belong to objects or classes

differentiation between classes and packages

support of `final` classes and methods

predefined class `Object` contains `equals`, `clone`, ...

Clases and Inheritance in Java

```
abstract class PlanarObject {
    protected float x, y;
    public float distance() { ... }
    public void move(float x, y) { ... }
    public abstract void draw();
}

class Circle extends PlanarObject {
    private float radius;
    public void draw() { ... }
}

class Rectangle extends PlanarObject {
    private float length, width;
    public void draw() { ... }
}
```

Interfaces in Java

```
interface Dictionary {
    void insert(String c; int i);
    int lookup(String c);
}

class ArrayDict implements Dictionary {
    private String[] names;
    private int[] values;
    private int size = 0;
    public void insert(String c; int i) { ... }
    public int lookup(String c) { ... }
}

class ListDict extends SomeList implements Dictionary { ... }
```

Polymorphism in Java

genericity including bounded genericity since version 1.5
— statically resolved

inclusion polymorphism supported by single/multiple inheritance
— dynamically resolved (statically when private, final or static)

overloading of routines, but no operators
— statically resolved

checked casts, type comparison at run-time
— semantic actions at run-time