CS 617 Object Oriented Systems Lecture 9 Polymorphism: Mere Syntactic Vs. Dynamic Binding, Subtyping, Subsumption Covariance, Contravariance 3:30-5:00 pm Thu, Jan 31

Rushikesh K Joshi

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Outline

Outline

Dynamic Binding and Polymorphism I

```
class A {
public:
        virtual void f () { cout « "A.f "; };
        virtual void g () { cout « "A.g "; };
        virtual void h () { cout « "A.h "; };
        virtual void k () { cout « "A.k "; };
};
class B : public A {
public:
        virtual void g () { cout « "B.g "; };
        virtual void h () { cout « "B.h "; };
};
class C : public B {
public:
        virtual void h () { cout « "C.h "; };
        virtual void k () { cout « "C.k "; };
};
```

Dynamic Binding and Polymorphism II

```
main () {
C *cp = new C;
B* bp = cp;
A* a1 = cp;
A* a2 = bp;
A* a3 = new B;
    cp->f(); cp->g(); cp->h(); cp->k();
    bp->f(); bp->g(); bp->h(); bp->k();
    a1->f(); a1->g(); a1->h(); a1->k();
    a2->f(); a2->g(); a2->h(); a2->k();
    a3->f(); a3->g(); a3->h(); a3->k();
}
```

Outline

Member Function Overloading I

```
class Complex {
        int i ; // real component
        int j ; // imaginary component
public:
        Complex (int x, int y) { i=x; j=y; }
        Complex add (Complex a) {
                i += a.i ; j += a.j;
                Complex tmp (i,j);
                return (tmp); }
        Complex add (int i) {
                Complex tmp (i,0);
                add (tmp); }
        void printState () { ... print c1 and c2 ... };
};
```

Member Function Overloading II

```
int main () {
Complex c1 (2,3), c2(4,6);
        c1.printState();
        c2.printState();
        c1.add (c2);
        c1.add (100);
        c1.printState();
        c2.printState();
}
```

Operator Overloading I

```
class Complex {
private:
        int i ; // real component
        int j ; // imaginary component
public:
        Complex (int x, int y) { i=x; j=y; }
        Complex operator + (Complex a) {
                Complex tmp (i+a.i,j+a.j);
                return (tmp); }
        Complex operator + (int x) {
                Complex tmp (i+x,j);
                return (tmp); }
        void printState () { ... }
};
```

Operator Overloading II

```
int main () {
Complex c1 (2,3), c2(4,6);
        ..print c1 and c2 before the addition..
        c1 = c1+c2;
        c1 = c1+100;
        c1+c2;
        c1+100;
        ..print c1 and c2 after c1=c1+c2; c1=c1+100; c1+c2; c1+100; ..
}
```

Templates I

```
template <class T>
class Node {
 public:
  T element;
  Node<T> *next;
  Node<T> *previous;
  Node (T e) { element = e; next=previous=NULL;}
};
```

Templates II

```
template <class T>
class List {

protected:

int len; // cardinality
Node<T>  *head;
Node<T>  *tail;

public:

    List ();
    List <T>& in (T element);
    // attach given elem at beginning
```

Templates III

```
T out ();
// take away front elem and return it.
// receiver list is the pruned one

List <T>& push (T element);
// attach given elem at end

T pop ();
// take away last node and return it.
//receiver list is the pruned one

List <T>& operator << (T element);
// same as in ; receiver list returned
```

Templates IV

```
List <T>& operator + (T element);
// same as push ; receiver list returned

T operator -  () ;
// same as out; element returned: unary prefix

T operator ~ ();
// same as pop; element returned: unary prefix

void read_visit (ListVisitor<T> *visitor);
// visitor object gets to reads all elements
```

Templates V

```
    void rw_visit (ListVisitor<T> *visitor);
    //visitor object gets to read/write
    // transformed elements are to be returned

    int length () {return len;}

    List <T> & operator = (List <T> inputlist);
    // copy constructor

    void nullify ();
    // nullifies the given list by terminating it

};
```

Templates VI

```
int main (void) {

 List <char> l,m,n;
 List <Account> la;

 //....

}
```

Syntactic Polymorphism

- Polymorphism Merely syntactic
- Compiler can remove polymorphism during compile time through a type analysis
- For example: all calls to overloaded functions are resolved
- Same type list is used to hold elements of different types, but the compiler generates two different implementations for two different types
- No dynamic binding in syntactic polymorphism

Polymorphism at Runtime

Can we use a value of one type where a value of another type is expected?

Outline

Relating Two Different Types

- Principle of Safe Substitution: A value of one type can safely used where a value of another type is expected
- When can you say a value of type T_1 can be used where a value of type T_2 is expected?

```
A a;
B b;
C c;
R f(B val) {.. use val here ..}
   ..
   ..
   f(a); when is this permitted?
   f(c); when is this permitted?
```

Consider Some Types which are Finite Sets of Integers

we know something about type int:
int = {-MAXINT .. 0 ... +MAXINT }

Now Let's define types A, B, C as below
Type A = {1,2,3,4,5}
Type B = {1,2,3}
Type C = {1,2}

What can we say about type safety of the above program?

What about acceptability of returned parameter?

```
A a;
B b;
C c;
R f(B val) {.. use val here ..}
..
..
a = f(x); when is this permitted?
b = f(x); when is this permitted?
c = f(x); when is this permitted?
```

The Subtype Relation

$S <: T$ (Meaning: S is a subtype of T)

It's safe to use a value of a subtype where a value of a supertype is expected.

i.e. $\frac{s:S, S<:T}{s:T}$
(called The Rule of Subsumption: The latter subsumes (includes) the former)

Formulate Rules for Subtyping for simple types, structures, functions, and now **Object Types**

Outline

Set Types

Subsets are Subtypes

Record Types

One Rule (depth rule):

$$\frac{\textit{for each } i{\in}1..n \;\; S_i{<:}T_i, \;\; s{:}S_{1..n}}{s{:}T_{1..n}}$$

$S_{1..n}, T_{1..n}$ are two records

Formulate a rule based on width of records?

Function Types

$f : T_1 \rightarrow T_2$
$g : S_1 \rightarrow S_2$

When is $g <: f$?

Now to Subtyping induced by Class Inheritance

```
class A {
        public T2 f(T1);
        public T4 g(T3);
}
class B inherits A {
        public S2 f(S1);
        public S4 g(S3);
}
main () {
        A a = new B
        X x = new X
        Y y
        y = a.f(x) ← when will this statement work safely?
}
```

Covariance and Contravariance

Which one is type-safe?

At what point of time do you guarantee type safety?

Subtyping and Subsumption put to Use

- Code written in terms of supertype works on all its subtypes
- Code written in terms of an interface will work on all classes implementing the interface
- code written in terms of a superclass will work on all its subclasses
- Provided that subtyping is established between the base and the derived entities