Unit - 5
Object Oriented Programming (B.Tech II-I)

Inheritance as Categorization

In one sense, the process of inheritance is a form of categorization.

A TextWindow is a type of Window, so class TextWindow inherits from class Window.

But in the real world, most objects can be categorized in a variety of ways. The author of the textbook is

- North American
- Male
- Professor
- Parent

None of these are proper subsets of the other, and we cannot make a single rooted inheritance hierarchy out of them.

Inheritance as Combination

Instead, real world objects are combinations of features from different classification schemes, each category giving some new insight into the whole:

- Author is North American, and
- Author is Male, and
- Author is a Professor, and
- Author is a Parent.

Note that we have not lost the is-a relationship; it still applies in each case.

CS Example - Complex Numbers

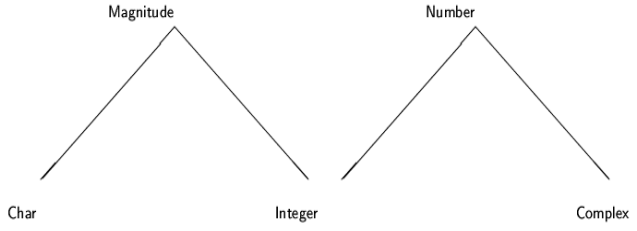
Two abstract classifications

Magnitude - things that can be compared to each other.
Number - things that can perform arithmetic
Three specific classes
Integer - comparable and arithmetic
Char - comparable but not arithmetic
Complex - arithmetic but not comparable

Possible Solutions

1. Make Number subclass of Magnitude, but redefine comparison operators in class complex to give error message if used. (subclassing for limitation)
2. Don't use inheritance at all - redefine all operators in all classes. (flattening the inheritance tree).
3. Use part inheritance, but simulate others - use Number, but have each number implement all relational operators.
4. Make Number and Magnitude independent, and have Integer inherit from both. (multiple inheritance).

Inheritance as a form of Combination:



Another Example - Walking Menus

- A Menu is a structure charged with displaying itself when selected by the user.
- A Menu maintains a collection of MenuItems.
- Each MenuItem knows how to respond when selected.
- A *cascading menu* is both a MenuItem and a Menu.

Problem with MI - Name Ambiguity:

What happens when same name is used in both parent classes.

- A CardDeck knows how to draw a Card.
- A GraphicalItem knows how to draw an image on a screen.
- A GraphicalCardDeck should be able to draw. which?

One Solution: Redefinition

One solution is to redefine one or the other operation in the child class.

```
class GraphicalCardDeck : public CardDeck, public GraphicalObject {
public:
        virtual void draw () { return CardDeck::draw(); }
        virtual void paint () { GraphicalObject::draw(); }
}

        GraphicalCardDeck gcd;
        gcd->draw(); // selects CardDeck draw
        gcd->paint(); // selects GraphicalObject draw
```

Problem with Redefinition Solution:

The redefinition solution is not without cost, however.

Now what happens when we run up against the principle of substitution?

```
        GraphicalObject * g = new GraphicalCardDeck();
        g->draw(); // opps, doing wrong method!
```

This problem can be mitigated, but the solution is complex and not perfect.

Other Approaches to Name Ambiguity

Other languages use different approaches to solving the problem of ambiguous names

- Eiffel uses the ability to rename features from the parent class. A polymorphic variable accessing through the parents name will access the renamed feature in the child.
- CLOS and Python resolve ambiguous names by the order in which the parent classes are listed. The first occurrence of the name found in a systematic search is the one selected.

Multiple Inheritance of Interfaces

Multiple inheritance of interfaces does not present the same problem of name ambiguity as does multiple inheritance of classes.

- Either the ambiguous methods in the parent classes have different type signatures, in which case there is no problem, or
- The ambiguous methods in the parent classes have the same signature. Still no problem, since what is inherited

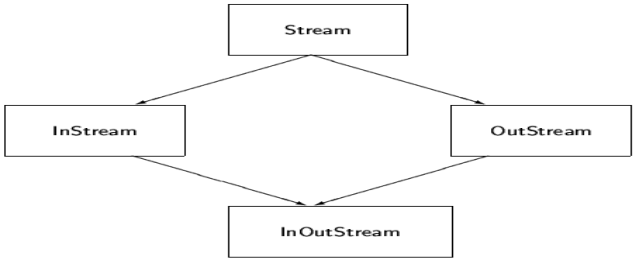
is only a specification, not an implementation.

This is why Java permits multiple inheritance of interfaces, not of classes.

Nevertheless, C# does not permit the same method name to be inherited from two parent interfaces.

Inheritance from Common Ancestors:

Another problem with MI occurs when parent classes have a common root ancestor. Does the new object have one or two instances of the ancestor?



Data Field in Common Ancestor

Imagine that the common ancestor declares a data member. Should the child class have one copy of this data field, or two?

Both answers can be justified with examples.

C++ gets around this by introducing the idea of a **virtual** parent class. If your parent is virtual there is one copy, and if not there is two.

Not a perfect solution, and makes the language complicated.

Inner Classes

The ability to next classes in C++ and Java provides a mechanism that is nearly equivalent to multiple inheritance, without the semantic problems.

```
class Child extends ParentOne {
        ...
        class InnerChild extends ParentTwo {
            ...
            // can access both parents
        }
}
```

Within the inner class you can access methods from both parent classes. This idiom is used a lot in Java. Solves many problems that would otherwise be addressed using MI. Not exactly equivalent to MI, but very close.

Definition of Polymorphic

Polymorphous: Having, or assuming, various forms, characters, or styles.

From greek routes, poly = many, and Morphos = form (Morphus was the greek god of sleep, who could assume many forms, and from which we derive the name Morphine, among other things).

A polymorphic compound can crystalize in many forms, such as carbon, which can be graphite, diamonds, or fullerenes.

In programming languages, used for a variety of different mechanisms.

Usage of the term is confused by the fact that it means something slightly different in the functional programming community than it does in the OO world.)

Major Forms of Polymorphism in Object Oriented Languages

There are four major forms of polymorphism in object-oriented languages:

- Overloading (ad hoc polymorphism) -- one name that refers to two or more different implementations.
- Overriding (inclusion polymorphism) -- A child class redefining a method inherited from a parent class.
- The Polymorphic Variable (assignment polymorphism) -- A variable that can hold different types of values during the course of execution. It is called Pure Polymorphism when a polymorphic variable is used as a parameter.
- Generics (or Templates) -- A way of creating general tools or classes by parameterizing on types.

Two Approaches to Software Reuse

One of the major goals of OOP is software reuse. We can illustrate this by considering two different approaches to reuse:

- Inheritance -- the *is-a* relationship.
- Composition -- the *has-a* relationship.

We do this by considering an example problem that could use either mechanism.

Example - Building Sets from Lists

Suppose we have already a **List** data type with the following behavior:

Want to build the Set data type (elements are unique).

```
class List {
public:
        void add (int);
        int  includes (int);
        void remove (int);
        int firstElement ();
};
```

Using Inheritance

Only need specify what is *new* - the addition method. Everything else is given for free.

```
class Set : public List {
public:
        void add(int);
};

void Set::add(int x)
{
        if (! includes(x)) // only include if not already there
            List::add(x);
}
```

Using Composition

Everything must be redefined, but implementation can make use of the list data structure.

```
class Set {
public:
        void add(int);
        int  includes(int);
        void remove(int);
        int firstElement();
private:
        List data;
};

void Set::add (int x)
        {
            if (! data.includes(x))
                    data.add(x);
        }

        int Set::includes (int x)
            { return data.includes(x); }

        void Set::remove (int x)
            { data.remove(x); }

        int Set::firstElement ()
            { return data.firstElement(); }
```

Advantages and Disadvantages of Each Mechanism

- Composition is simpler, and clearly indicates what operations are provided.
- Inheritance makes for shorter code, possibly increased functionality, but makes it more difficult to understand what behavior is being provided.
- Inheritance may open to the door for unintended usage, by means of unintended inheritance of behavior.
- Easier to change underlying details using composition (i.e., change the data representation).
- Inheritance may permit polymorphism
- Understandability is a toss-up, each has different complexity issues (size versus inheritance tree depth)
- Very small execution time advantage for inheritance

A Definition of Overloading

We say a term is *overloaded* if it has two or more meanings.

Most words in natural languages are overloaded, and confusion is resolved by means of context.

Same is true of OO languages. There are two important classes of context that are used to resolve overloaded names

- Overloading based on Scopes
- Overloading based on Type Signatures

Overloading Based on Scopes

A *name scope* defines the portion of a program in which a name can be used, or the way it can be used. Scopes are introduced using lots of different mechanisms:

- Classes or interfaces
- Packages or Units
- Procedures or Functions
- in some languages, even Blocks

An advantage of scopes is that the same name can appear in two or more scopes with no ambiguity.

Florist Example from Chapter 1

```
Procedure Friend.sendFlowersTo (anAddress : address);
begin
        go to florist;
        give florist message sendFlowersTo(anAddress);
end;

Procedure Florist.sendFlowersTo (anAddress : address);
begin
        if address is nearby then
                make up flower arrangement
                tell delivery person sendFlowersTo(anAddress);
        else
                look up florist near anAddress
                phone florist
                give florist message sendFlowersTo(anAddress)
end;
```

Resolving Overloaded Names

This type of overloading is resolved by looking at the type of the receiver.

Allows the same name to be used in unrelated classes.

Since names need not be distinct, allows short, easy to remember, meaningful names.

Overloading Based on Type Signatures

A different type of overloading allows multiple implementations in the same scope to be resolved using type signatures.

```
class Example {
                // same name, three different methods
        int sum (int a) { return a; }
        int sum (int a, int b) { return a + b; }
        int sum (int a, int b, int c) { return a + b + c; }
}
```

A type signature is the combination of argument types and return type. By looking at the signature of a call, can tell which version is intended.

Resolution Performed at Compile Time

Note that resolution is almost always performed at compile time, based on static types, and not dynamic values.

```
class Parent { ... };
class Child : public Parent { ... };
void Test(Parent * p) { cout << "in parent" << endl; }
void Test(Child * c) { cout << "in child" << endl }
        Parent * value = new Child();
        Test(value);
```

Example will, perhaps surprizingly, execute parent function.

Easy to Extend

Since output uses overloading, it is very easy to extend to new types.

```
class Fraction {
public:
        Fraction (int top, int bottom) { t = top; b = bottom; }

        int numerator() { return t; }
        int denominator() { return b; }

private:
        int t, b;
};

ostream & operator << (ostream & destination, Fraction & source)
{
        destination << source.numerator() << "/" << source.denominator();
        return destination;
}

Fraction f(3, 4);
cout << "The value of f is " << f << '\n';
```

Conversion and Coercisons

When one adds conversions into the mix, resolving overloaded function or method calls can get very complex. Many different types of conversions:

- Implicit value changing conversion (such as integer to real)
- Implicit conversion that does not change value (pointer to child class converted into pointer to parent)
- Explicit conversions (casts)
- Conversion operators (C++ and the like)

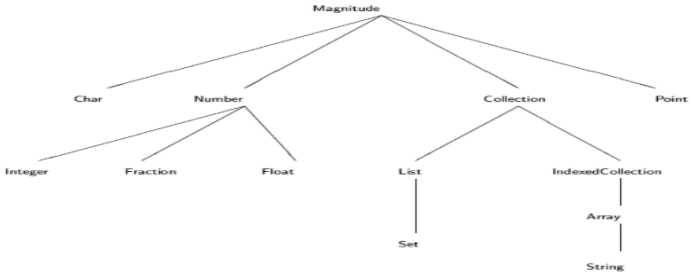
See text for illustration of the complex rules.

Difference from Overloading

Like overloading, there are two distinct methods with the same name. But there are differences:

- Overriding only occurs in the context of the parent/child relationship
- The type signatures must match
- Overridden methods are sometimes combined together
- Overriding is resolved at run-time, not at compile time.

Subclasses of Magnitude in Little Smalltalk



Overridden Relationals

Child classes need only override *one* method (for example <) to get effect of all relationals.

- Overridden in class Integer to mean integer less than.
- Overridden in class Char to be ascii ordering sequence.
- Overridden in class String to mean lexicalgraphic ordering.
- Overridden in class Point to mean lower-left quadrant.

Overriding

- In some languages (Smalltalk, Java) overriding occurs automatically when a child class redefines a method with the same name and type signature.
- In some languages (C++) overriding will only occur if the parent class has declared the method in some special way (example, keyword **virtual**).
- In some languages (Object Pascal) overriding will only occur if the child class declares the method in some special way (example, keyword **override**).
- In some languages (C#, Delphi) overriding will only occur if both the parent and the child class declare the method in some special way.

Replacement and Refinement

There are actually two different ways that overriding can be handled:

- A *replacement* totally and completely replaces the code in the parent class the code in the child class.
- A *refinement* executes the code in the parent class, and adds to it the code in the child class.

Most languages use both types of semantics in different situations. Constructors, for example, almost always use refinement.

Reasons to use Replacement

There are a number of reasons to use replacement of methods.

- The method in the parent class is abstract, it must be replaced.
- The method in the parent class is a default method, not appropriate for all situations.
- The method in the parent can be more efficiently executed in the child.

Constructors use Refinement

In most languages that have constructors, a constructor will always use refinement.

This guarantees that whatever initialization the parent class performs will always be included as part of the initialization of the child class.

Constructors use Refinement

In most languages that have constructors, a constructor will always use refinement.

This guarantees that whatever initialization the parent class performs will always be included as part of the initialization of the child class.

```
class Silly {
        private int x; // an instance variable named x

        public void example (int x) { // x shadows
instance variable
            int a = x+1;
            while (a > 3) {
                int x = 1; // local variable
shadows parameter
                a = a - x;
            }
        }
}
```

Overriding, Shadowing and Redefinition

Overriding	The type signatures are the same in both parent and child classes, and the method is declared as virtual in the parent class.
Shadowing	The type signatures are the same in both parent and child classes, but the method was not declared as virtual in the parent class.
Redefinition	The type signature in the child class differs from that given in the parent class.

C++ virtual function:

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

```
#include <iostream>
using namespace std;
class A
{
  int x=5;
   public:
   void display()
   {
      std::cout << "Value of x is : " << x<<std::endl;
   }
};
class B: public A
{
   int y = 10;
   public:
   void display()
   {
      std::cout << "Value of y is : " <<y<< std::endl;
   }
};
int main()
{
   A *a;
   B b;
   a = &b;
  a->display();
   return 0;

          }
```

Output:

```
Value of x is : 5
```

Overriding Comparisons in Smalltalk

An interesting example of overriding in found in class Magnitude in Smalltalk.

```
<= arg
        ^ self < arg or: [ self = arg ]

>= arg
        ^ arg <= self

< arg
        ^ self <= arg and: [ self ~= arg ]

> arg
        ^ arg < self
```

Notice how these definitions are circular.

Covariance and Contravariance

Frequently it seems like it would be nice if when a method is overridden we could change the argument types or return types. A change that moves down the inheritance hierarchy, making it more specific, is said to be covariant. A change that moves up the inheritance hierarchy is said to be contravariant.

```
class Parent {
        void test (covar : Mammal, contravar : Mammal) : boolean
}

class Child extends Parent {
        void test (covar : Cat, contravar : Animal) : boolean
}
```

While appealing, this idea runs into trouble with the principle of substitution.

```
Parent aValue = new Child();
aValue.text(new Dog(), new Mammal()); // is this legal??
```

Contravariant Return Types

To see how a contravariant change can get you into trouble, consider changing the return types:

```
class Parent {
    Mammal test ( ) {
            return new Cat();
    }
}

class Child extends Parent {
    Animal test () {
            return new Bird();
    }
}

Parent aParent = new Child();
Mammal result = aValue.test(); // is this legal?
```

Most languages subscribe to the novariance rule: no change in type signatures.

A Safe Variance Change

C++ allows the following type of change in signature:

```
class Parent {
public:
        Parent * clone () { return new Parent(); }
};
class Child : public Parent {
public:
        Child * clone () { return new Child(); }
};
```

No type errors can result from this change.

Variations on Overriding:

Overriding is used when you have two classes with very similar code. Some of the code is different so you want to have some common code and some abstract methods that can be changed from variation to variation. The variations will turn into abstract classes that extend your base class.

Virtual methods in java:

In object-oriented programming, a virtual function or virtual method is a function or method whose behaviour can be overridden within an inheriting class by a function with the same signature to provide the polymorphic behavior.

```
class Vehicle{
void make(){
System.out.println("heavy duty");
}
}
public class Trucks extends Vehicle{
void make(){
System.out.println("Transport vehicle for heavy duty");
}
public static void main(String args[]){
Vehicle ob1 = new Trucks();
ob1.make();
}
}
```

Output: Transport

Every non-static method in Java is a virtual function except for final and private methods. The methods that cannot be used for the polymorphism is not considered as a virtual function.

A static function is not considered a virtual function because a static method is bound to the class itself. So we cannot call the static method from object name or class for polymorphism. Even when we override the static method it does not resonate with the concept of polymorphism.