Overloading, Overriding and Method Dispatch

Upasana Pujari
POLYMORPHISM

<table>
<thead>
<tr>
<th>poly</th>
<th>morphos</th>
</tr>
</thead>
<tbody>
<tr>
<td>many</td>
<td>forms</td>
</tr>
</tbody>
</table>

- Overloading – single method name having several alternative implementations.

- Overriding – child class provides alternative implementation for parent class method.

- Polymorphic variable – a variable that is declared as one type but holds a value of a different type.
Polymorphic Variable

Example:

```java
Class Shape {
...
}

Class Triangle extends Shape {
...
}

Shape s = new Triangle;
```

- Java – all variables can be polymorphic.
- C++ – only pointers and references can be polymorphic.
Method Binding

• Determining the method to execute in response to a message.
• Binding can be accomplished either statically or dynamically.

Static Binding –
• Also known as “Early Binding”.
• Resolved at compile time.
• Resolution based on static type of the object(s).

Dynamic Binding –
• Also known as “Late Binding”.
• Resolved at run-time.
• Resolution based on the dynamic type of the object(s).
• Uses method dispatch table or Virtual function table.
Method Binding Example

Class Shape {
public:
virtual void Draw() { cout << "Shape Draw!" << endl; }
}

Class Triangle : public Shape {
public:
void Draw() { cout << "Triangle Draw!" << endl; }
}

Shape * sptr = new Triangle();
sptr->Draw(); // Triangle Draw!
Overloading

- Overloading Based on Scopes
- Overloading based on Type Signatures
Overloading

Overloading Based on Scopes

• same method name in different scopes.
• the scopes cannot overlap.
• No restriction on semantic similarity.
• No restriction on type signatures.
• Resolution of overloaded names based on class of receiver.

Example

```java
Class SomeCards {
    Draw() {...} // Paint the face of the card
}

Class SomeGame {
    Draw() {...} // Remove a card from the deck of cards
}
```
Overloading Based on Type Signatures

- same method name with different implementations having different type signatures.
- Resolution of overloaded names is based on type signatures.
- Occurs in object-oriented languages (C++, Java, C#, Delphi Pascal)
- Occurs in imperative languages (Ada), and many functional languages.

```java
Class Example {
    Add(int a) { return a; }
    Add(int a, int b) { return a + b; }
    Add(int a, int b, int c) { return a + b + c; }
}
```

- C++ allows methods as well as operators to be overloaded.
- Java does not allow operators to be overloaded.
Overloading and Method Binding

Resolution of Overloaded Methods
• Method binding at compile time.
• Based on static types of argument values
• Methods cannot be overloaded based on differences in their return types alone.

```cpp
Class SomeParent {…}
Class SomeChild : public SomeParent {…}

void Test(SomeParent *sp) { cout << "In Parent"; } 
void Test(SomeChild *sc) { cout << "In Child"; }
SomeParent *value = new SomeChild(); 

Test(value); // "In Parent"
```
Overloading Example

Overloading can be used to extend library functions and operators so they can work with user-defined data types.

Class Fraction
private:
  int t, b;

public:
  Fraction (int num, int denum) { t = num; b = denum; }
  int numerator() { return t; }
  int denominator() { return b; }

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

- Coercion and Conversion
- Redefinition
- Polyadicity
- Multi-Methods
Coercion and Conversion

- Used when actual arguments of a method do not match the formal parameter specifications, but can be converted into a form that will match.

- Coercion - implicitly implemented
  Example floatvar = intvar;

- Conversion - explicitly requested by the programmer
  Example floatvar = (double) intvar;
Substitution as Conversion

- Used when there is parent-child relationship between formal and actual parameters of a method

```
Dessert

void order ( Dessert d, Cake c );
void order ( Pie p, Dessert d );
void order ( ApplePie a, Cake c );

Pie          Cake

ApplePie     ChocolateCake

order (aDessert, aCake);
order (anApplePie, aDessert)
order (aDessert, aDessert);       // illegal
order (anApplePie, aChocolateCake)
order (aPie, aCake);
```
Substitution as Conversion

**Resolution rules** (when substitution is used as conversion in overloaded methods)

- If there is an exact match, execute that method.
- If there are more than one matching methods, execute the method that has the most specific formal parameters.
- If there are two or more methods that are equally applicable, the method invocation is ambiguous, so generate compiler error.
- If there is no matching method, generate compiler error.
Conversion

Conversion operators in C++
(these are the user supplied conversions)

• *One-argument constructor* : to convert from argument type to class type.
  ```cpp
  Fraction (int value)
  {
    t = value; b = 1; // Converts int into Fraction
  }
  ```

• *Operator with type name as its name* : to convert class type to named type.
  ```cpp
  operator double ()
  {
    return numerator() / (double) denominator;
  }
  ```
Conversion

Rules for Resolution of Overloaded methods
(taking into account all of the various conversion mechanisms)

• execute method whose formal parameters are an exact match for the actual parameters

• match using standard type promotions (e.g. integer to float)

• match using standard substitution (e.g. child types as parent types)

• match using user-supplied conversions (e.g. one-argument constructor, type name operator)

• if no match found, or more than one method matches, generate compiler error
Redefinition

When a child class defines a method with the same name as a method in the parent class but with a different type signature.

Class Parent {
    public void Test (int a) {...}
}

Class Child extends Parent {
    public void Test (int a, int b) {...}
}

Child aChild = new Child();
aChild.Test(5);

How is it different from overriding?
Different type signature in Child class.
Redefinition

Two approaches to resolution

Merge model
- used by Java, C#
- method implementations found in all currently active scopes are merged into one list and the closest match from this list is executed.
- in the example, parent class method will be executed.

Hierarchical model
- used by C++
- each currently active scope is examined in turn to find the closest matching method
- in the example, compilation error in Hierarchical model

Delphi Pascal - can choose which model is used
merge model - if overload modifier is used with child class method.
Hierarchical model - otherwise.
Polyadicity

Polyadic method - method that can take a variable number of arguments.

```c
printf("%s", strvar);
printf("%s, %d", strvar, intvar);
```

- Easy to use, difficult to implement
- `printf` in C and C++; `writeln` in Pascal; `+` operator in CLOS

```c
#include <stdarg.h>
int sum (int argcnt, ...) // C++ uses a data structure called
{                          // variable argument list
    va_list ap;
    int result = 0;
    va_start(ap, argcnt);
    while (argcnt > 0) {
        result += va_arg(ap, int);
        argcnt--;
    }
    va_end(ap);
    return result;
}
```
Optional Parameters

Another technique for writing Polyadic methods.

- Provide default values for some parameters.
- If values for these parameters are provided then use them, else use the default values.
- Found in C++ and Delphi Pascal

\begin{verbatim}
  AmtDue(int fixedCharge);
  AmtDue(int fixedCharge, int fines);
  AmtDue(int fixedCharge, int fines, int missc);
\end{verbatim}

same as

\begin{verbatim}
  AmtDue(int fixedCharge, int fines = 0, int missc = 0);
\end{verbatim}
Multi-Methods

• combines the concepts of overloading and overriding.
• Method resolution based on the types of all arguments and not just the type of the receiver.
• Resolved at runtime.

The classes integer and real are derived from the parent class number.

```
function add (Integer a, Integer b) : Integer { ... }
function add (Integer a, Real b) : Real { ... }
function add (Real a, Integer b) : Real { ... }
function add (Real a, Real b) : Real { ... }
```

```
Number x = ... ; // x and y are assigned some unknown values
Number y = ... ;
Real r = 3.14;

Real r2 = add(r, x); // which method to execute
Real r3 = add(x, y); // this is not type safe
```
Multi-Methods

Double dispatch
• a message can be used to determine the type of a receiver.

• To determine the types of two values, the same message is sent twice, using each value as receiver in turn.

• Then execute the appropriate method.
Overloading Based on Values

Overloading based on values
- overload a method based on argument values and not just types.
- Occurs only in Lisp-based languages - CLOS, Dylan.
- High cost of method selection algorithm.

Example

```plaintext
function sum(a : integer, b : integer) {return a + b;

}  
function sum(a : integer = 0, b : integer) {return b;

}
```

The second method will be executed if the first argument is the constant value zero, otherwise the first method will be executed.
Overriding

A method in child class overrides a method in parent class if they have the same name and type signature.

- **Overriding**
  - classes in which methods are defined must be in a parent-child relationship.
  - Type signatures must match.
  - Dynamic binding of messages.
  - Runtime mechanism based on the dynamic type of the receiver.
  - Contributes to code sharing (non-overriding classes share same method).
Overriding Notation

C++

class Parent {
    public:
        virtual int test (int a) { ... }
}
class Child : public Parent {
    public:
        int test (int a) { ... }
}

C#

class Parent {
    public virtual int test (int a) { ... }
}
class Child : Parent {
    public override int test (int a) { ... }
}
Overriding Notation

Java

class Parent {
    public int test (int a) { ... }
}
class Child extends Parent {
    public int test (int a) { ... }
}

Object Pascal

type
    Parent = object
        function test(int) : integer;
    end;
    Child = object (Parent)
        function test(int) : integer; override;
    end;
Overriding

Overriding as Replacement
• child class method totally overwrites parent class method.
• Parent class method not executed at all.
• Smalltalk, C++.

Overriding as Refinement
• Parent class method executed within child class method.
• Behavior of parent class method is preserved and augmented.
• Simula, Beta

Constructors always use the refinement semantics of overriding.
Replacement in SmallTalk

In support of code reuse

Code Reuse

Person
  └── GenerateReport

Director
Manager
Trainee
  └── GenerateReport

Overridden method as replacement
Replacement in SmallTalk

In support of code optimization

```
"class boolean"
{&} right
  self ifTrue: [right ifTrue: [^true] ].
  ^ false

"class True"
{&} right
  ^ right

"class False"
{&} right
  ^ false
```
Refinement in Beta

• Always code from parent class is executed first.
• When ‘inner’ statement is encountered, code from child class is executed.
• If parent class has no subclass, then ‘inner’ statement does nothing.

Example

class Parent {
    public void printResult () {
        print('< Parent Result; ');
        inner;
        print('>' );
    }
}

Parent p = new Child();
p.printResult();

< Parent Result; Child Result; >
Simulation of Refinement using Replacement

C++

```c++
void Parent::test () {
    cout << "in parent \n" ;
}
void Child::test () {
    Parent::test();
    cout << "in child \n"
}
```

Object Pascal

```object_pascal
procedure Parent.test ();
begin
    writeln("in parent");
end;
procedure Child.test ();
begin
    inherited test ();
    writeln("in child");
end;
```

Java

```java
class Parent {
    void test () {System.out.println("in parent");}
}
class Child extends Parent {
    void test () {
        super.test();
        System.out.println("in child");
    }
}
```
Refinement Vs Replacement

Refinement
• Conceptually very elegant mechanism
• Preserves the behavior of parent.
  (impossible to write a subclass that is not also a subtype)
• Cannot simulate replacement using refinement.

Replacement
• No guarantee that behavior of parent will be preserved.
  (it is possible to write a subclass that is not also a subtype).
• Can be used to support code reuse and code optimization
• Can simulate refinement using replacement.
Wrappers in CLOS

This mechanism can be used to simulate refinement.
A subclass overrides parent method and specifies a wrapping method.
Wrapping method can be
• ‘before’ method
• ‘after’ method
• ‘around’ method

(defclass parent () () )
(defclass child (parent) )
(defmethod test ((x parent)) (print "test parent"))
(defmethod atest :after ((x child)) (print "atest child"))
(defmethod btest :before ((x child)) (print "btest child"))
(defmethod rtest :around ((x child))
  (list "rtest chld before" (call-next-method) "rtest chld after"))

(defvar aChild (make-instance 'child))
(atest aChild) "atest child" "test parent"
(atest aChild) "test parent" "btest child"
(atest aChild) "rtest chld before" "test parent" "rtest chld after"
Deferred Methods

- Defined but not implemented in parent class.
- Also known as abstract method (Java) and pure virtual method (C++)
- Associates an activity with an abstraction at a higher level than it actually is.

- Used to avoid compilation error in statically typed languages.
Deferred Method Example

C++
class Shape {
    public:
        virtual void Draw () = 0;
}

Java
abstract class Shape {
    abstract public void Draw ();
}

Smalltalk
Draw
   " child class should override this"
   ^ self subclassResponsibility

(Smalltalk does implement the deferred method in parent class but when invoked will raise an error)
Shadowing

Child class implementation shadows the parent class implementation of a method.

- As example in C++, when overridden methods are not declared with ‘virtual’ keyword.
- Resolution is at compile time based on static type of the receiver.

```cpp
class Parent {
    public:
        void test () { cout << “in Parent” << endl; }
    }

class Child : public Parent {
    public:
        void test () { cout << “in Child” << endl; }
    }

    Parent *p = new Parent();
p->test(); // in Parent

    Child *c new Child();
c->test(); // in Child

    p = c;
p->test(); // in Parent
```
Overriding, Shadowing and Redefinition

**Overriding**
- Same type signature and method name in both parent and child classes.
- Method declared with language dependent keywords indicating overriding.

**Shadowing**
- Same type signature and method name in both parent and child classes.
- Method not declared with language dependent keywords indicating overriding.

**Redefinition**
- Same method name in both parent and child classes.
- Type signature in child class different from that in parent class.
Covariance and Contravariance

- An overridden method in child class has a different type signature than that in the parent class.
- Difference in type signature is in moving up or down the type hierarchy.

class Parent {
    public void test (Shape s, Square sq) {
        ... }
}

class Child extends Parent {
    public void test (Square sq, Shape s) {
        ... }
}
Covariance and Contravariance

• Covariant change - when the type moves down the type hierarchy in the same direction as the child class.

```java
Parent aValue = new Child();
aValue.func(aTriangle, aSquare);   // Run-time error
                                // No compile-time error
```

• Contravariant change - when the type moves in the direction opposite to the direction of subclassing.

```java
Parent aValue = new Child();
aValue.func(aSquare, aSquare);    // No errors
```
Covariance and Contravariance

• Covariant change in return type

    Shape func () { return new Triangle(); } // In Parent Class  
    Square func () { return new Square(); } // In Child Class  

    Parent aValue = new Child();  
    Shape aShape = aValue.func(); // No compile-time or Run-Time errors  

• Contravariant change in return type

    Square func () { return new Square(); } // In Parent Class  
    Shape func () { return new Triangle(); } // In Child Class  

    Parent aValue = new Child();  
    Square aSquare = aValue.func();  
                                // No compile-time errors  
                                // Run-Time error  

• C++ allows covariant change in return type.  
• Eiffel, Sather allows both covariant and contravariant overriding  
• Most other languages employ novariance
And Finally...

Java
• ‘final’ keyword applied to functions prohibits overriding.
• ‘final’ keyword applied to classes prohibits subclassing.

C#
• ‘sealed’ keyword applied to classes prohibits subclassing.
• ‘sealed’ keyword cannot be applied to individual functions.