Overloading, Overriding

Jessie Li

2006-11-10

Outline

Polymorphism, Method binding

Overloading

- **Overloading Based on Scopes**
- **Overloading based on Type Signatures**
- **Coercion and Conversion**
- **Redefinition**
- **Polyadicity**
- **Multi-Methods**

Overriding

- **Notating Overriding**
- **Replacement vs. Refinement**
- **Deferred Methods**
- **Overriding vs. Shadowing**
- **Covariance and Contravariance**
- **Variations on Overriding**

Polymorphism

- *Polymorphism* translates from Greek as many forms
  (poly: many morph: forms)
- *Polymorphic variable*: a variable that is declared as one type but holds a value of a different type.

Example :

```
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 objects.

Dynamic Binding

- Also known as “*Late Binding*”.
- Resolved at run-time.
- Resolution based on the dynamic type of the objects.

Scopes and Type Signatures

- *What is Scope?*
  - A scope defines the portion of a program in which a name can be used or the way in which the name can be used.

- *What is Type Signature?*
  - is a description of the argument types associated with a function, the order of arguments, and the return type.

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

```
Class Cards {
  Draw(){…} //Draw an image of the card on the screen
}

Class Game {
  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.

```
Class Example {
    //same name, three different methods
    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++ permits any method, procedure, or operator to be overloaded parametrically.
- Java does not allow operators to be overloaded.
- In Delphi Pascal “overload” must be explicitly declared.

Delphi Pascal: explicitly declare overload

```
Type
   example = class
   pubic
     function sum(a:Integer): Interger; overload;
     function sum(a,b:Integer): Integer; overload;
   end;
```

Overloading and Method Binding

Resolution of Overloaded Methods

- Method binding at compile time.
- Based on static types of argument values.
- Methods can’t be overloaded based on different return types alone.

```
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);
What is the output?
// “In Parent”
```

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 – an implicitly change in type

  Example

  ```
  double x = 2.5;
  int i = 3;
  x = i + x;  //integer i will be converted to real
  ```

- Conversion – a change in type explicitly requested by the programmer

  Example

  ```
  x = ( ( double )  i ) + x;
  ```

- When do Overloading and Coercion happen?

  Example:

  ```
  1. integer + integer
  2. integer + real           1+2+3+4  (overloading only)
  3. real + integer           1+4      (combination)
  4. real + real              4        (coercion only)
  ```

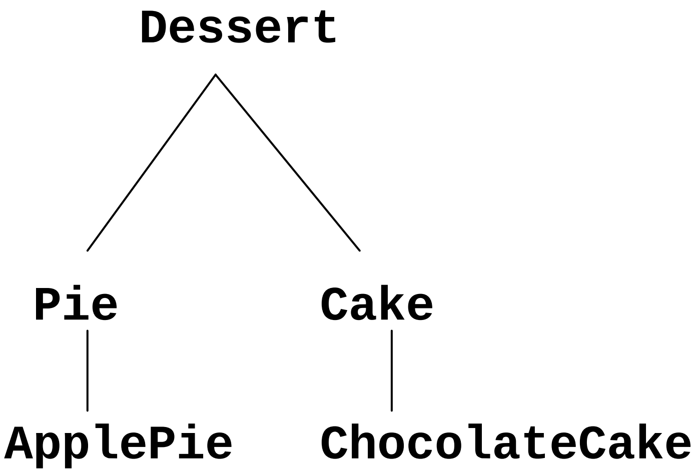
Substitution as Conversion

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, and a compiler error will be reported.
- If there is no matching method, a compiler error will be reported.

Substitution as Conversion

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



Dessert

Pie

Cake

ApplePie

ChocolateCake

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

```
order (aDessert, aCake);
order (anApplePie, aDessert)
order (aDessert, aDessert);   // compiler error, no match
order (aPie, aCake);          // compiler error, two match
order (anApplePie, aChocolateCake)
```

Redefinition

When a child class defines a method using 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 overloading?

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 a single collection 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 (redefining both methods in the child class solve the error)

Delphi Pascal - can choose which model is used

merge model - if *overload* modifier is used with child class method.

Hierarchical model - otherwise.

```
 type
    Parent = class
    Public
            procedure Example(A: Integer);
    end;
    ChildWithOneMethod = class (Parent)
    public
            procedure Example (A, B: Integer);
    end;
    ChildWithTwoMethod = class (Parent)
    public
            procedure Example (A, B: Integer); overload;
    end;
var
    C1: ChildWithOneMethod; C2: ChildWithTwoMethod;
begin
    C1 := ChildWithOneMethod.Create;
    C2 := ChildWithTwoMethod.Create;
    C1.Example(42); // error:not enough parameters
    C2.Example(42); // OK
end
```

Polyadicity

- What is Polyadicity?

  Polyadic function: that can take a variable number of arguments.

```
printf(“%s”, strvar);
printf(“%s, %d”, strvar, intvar);
```

- Easy to use, difficult to implement

- Example:
  - ***printf*** in C and C++;
  - ***writeln*** in Pascal;
  - **+ *operator*** in CLOS

    (+ 2 3)

    (+ 2 3 4 5 6)

Optional Parameters

One technique for writing Polyadic functions.

- 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

```
function Count (A, B: Integer; C: Integer  0; D: Integer = 0);
begin
    Result:= A + B + C + D;
end

begin
    Writeln (Count(2, 3, 4, 5)); //can use four arguments
    Writeln (Count(2, 3, 4)); // or three
    Writeln (Cound(2, 3)); // or two
end
```

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.

Resolution of overloaded function by the types of all arguments would introduce problem:

```
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);    // is the assignment type-safe?
```

Multi-Methods

How to solve the problem? **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

- 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

```
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.

Overloading Summary

- Overloading is the compile time matching of a function invocation to one of many similar named methods
- Two categories of overloading: scope based, type signature based
- Similar concepts: conversion and redefinition
- An alternative to overloading is the creation of polyadic functions

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

Java (smalltalk, object-c)

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

C++

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

Overriding Notation

Object Pascal

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

C# (Delphi Pascal)

```
class Parent {
  public virtual int test (int a) { … }
}
class Child : Parent {
  public override int test (int a) { … }
}
```

Replacement vs. Refinement

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.

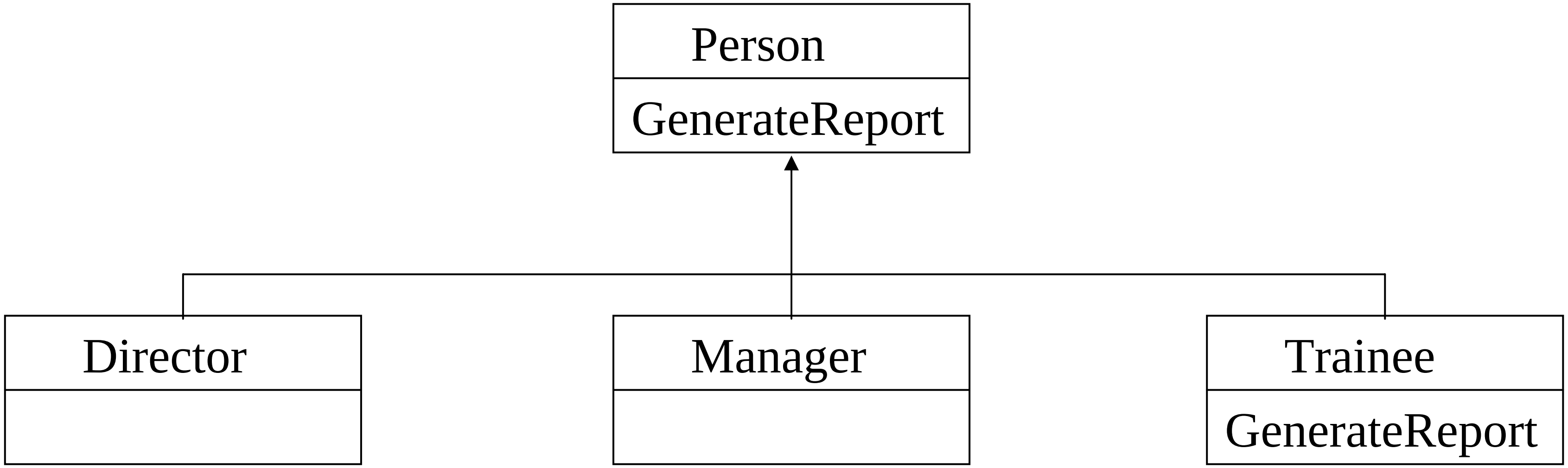
Replacement

Two major reasons for using replacement:

- in support of code reuse
- as a technique for optimization

Replacement in SmallTalk

In support of code reuse



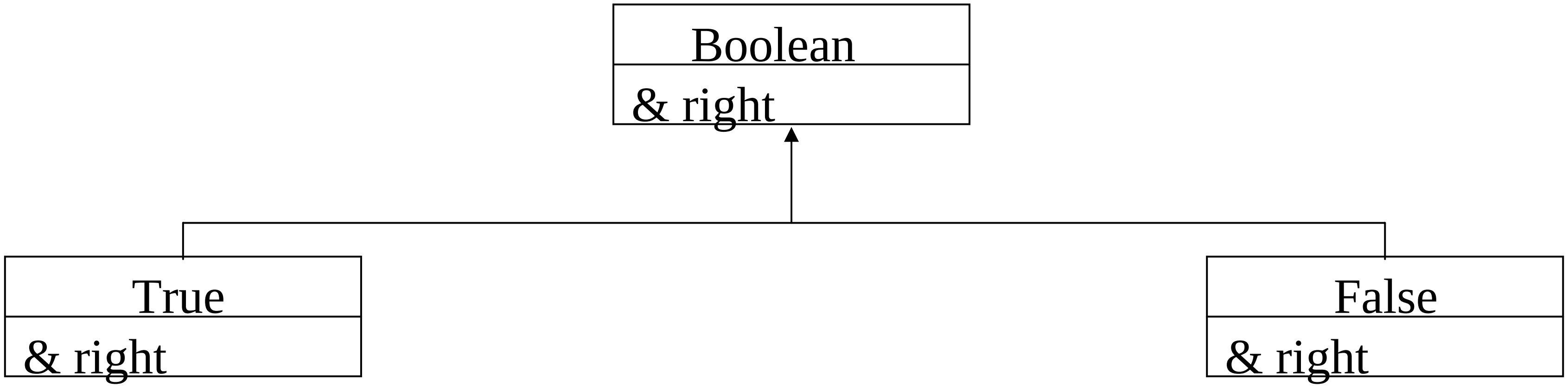
<------------------ Code Reuse ----------------------->

<----------------------->

Overriden 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(‘>’);
  }
}
```

```
class Child extends Parent {
  public void printResult () {
    print(‘Child Result; ’);
    inner;
  }
}
```

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

```
< Parent Result; Child Result; >
```

Simulation of Refinement using Replacement

C++

```
void Parent::example (int a) {
  cout << “in parent \n” ;        }

void Child::example (int a) {
  Parent::example(12); //do parent action
  cout << “in child \n”; //then child action   }
```

Java

```
class Parent {
    void example (int a) {
         System.out.println(“in parent”);} }
class Child extends Parent {
    void example (int a) {
         super.example(12); //super refers to parent class
         System.out.println(“in child”);  } }
```

Java: *super* refers to parent class, (Smalltalk, Object-C)

C#: uses keyword *base.*

Object Pascal, Delphi Pascal: use keyword *inherited*

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.

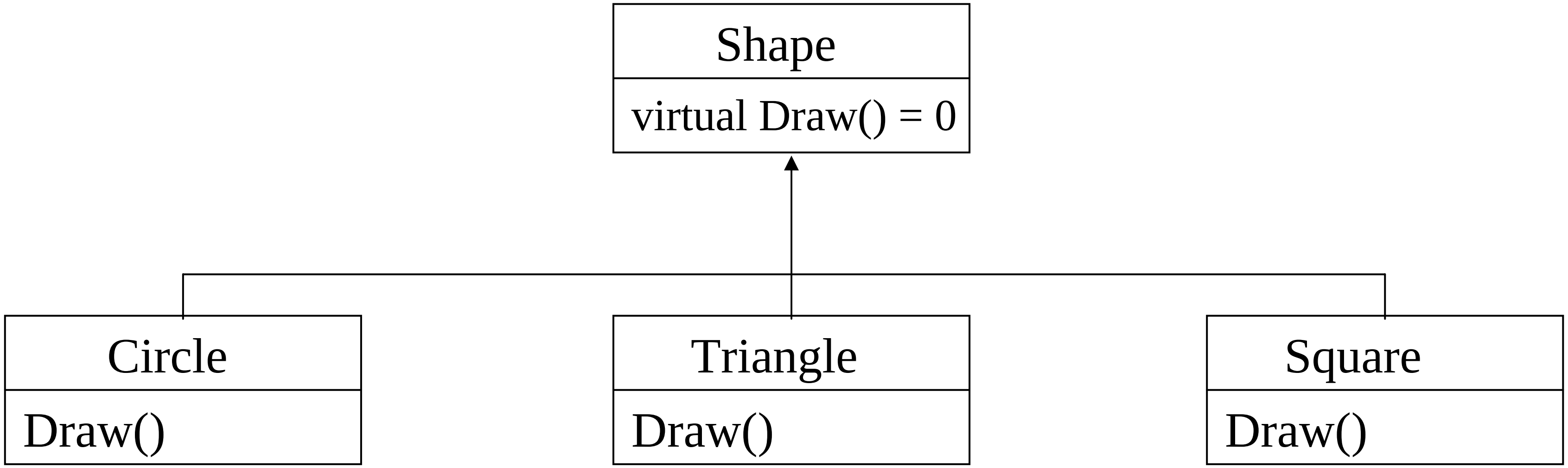
Replacement

- 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.

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 (C# and Delphi are similar)

```
abstract class Shape {
  abstract public void Draw ();
```

Smalltalk (Objective-C is similar)

```
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

What is shadowing?

```
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;
        }
    }
}
```

Shadowing vs. Overriding

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

- A shadowing performed if no keyword provided for indication of overloading
- Resolution is at compile time based on static types

```
class Parent {
public: // no virtual keyword
  void example () { cout << “in Parent” << endl; }
}
class Child : public Parent {
public:
  void example () { cout << “in Child” << endl; }
}

Parent *p = new Parent();
p->example();                 // in Parent
Child *c new Child();
c->example();                 // in Child
p = c; // be careful here!
p->example();                 // 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.

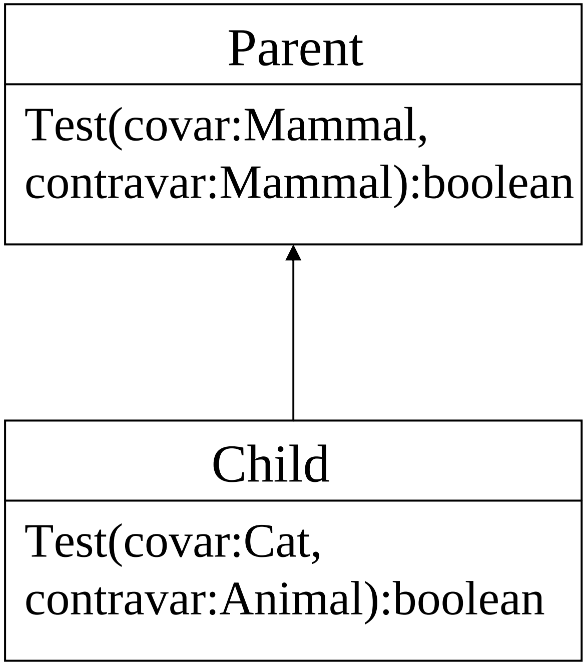
Redefinition

- 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.

- Covariant change - when the type moves down the type hierarchy in the same direction as the child class.
- Contravariant change - when the type moves in the direction opposite to the direction of subclassing.



Parent

Test(covar:Mammal,
contravar:Mammal):boolean

Child

Test(covar:Cat,
contravar:Animal):boolean

Covariance and Contravariance

- Covariant change to a by-value parameter

```
Parent aValue = new Child();
aValue.test(new Dog(), new Mammal());// Run-time error
                               // No compile-time error
```

- Contravariance change to a by-value parameter

```
No errors
```

Covariance and Contravariance

- Covariant change in return type

```
// No compile-time or Run-Time errors
```

- Contravariant change in return type

```
Class Parent {
    Mammal test () {
       return new Cat();} }
Class Child extends Parent {
   Animal test () {
       return new Bird();}}
Parent aValue = new child();
Mammal result = aValue.test(); // error: a bird is not a mammal
```

- C++ allows covariant change in return type.
- Eiffel allows both covariant and contravariant overriding
- Most other languages employ novariance to avoid this problem.

Variation on Overriding

Java

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

Example:

```
Class Parent {
    public final void aMethod (int) {…}
}
Class Child extends Parent {
    // compiler error, not allowed to override final method
    public void aMethod (int) {…}
}
```

C#

- ‘sealed’ keyword applied to classes prohibits subclassing.
- ‘sealed’ keyword cannot be applied to individual functions.

Overriding Summary

- Method in Child class use the same name and type signature as that in parent class
- Overriding is resolved at run time. (overloading at compile time)
- Replacement replaces the parent’s code; Refinement combines the code.
- Deferred method is a form of overriding where no implementation in parent and implementation in child.
- A name can shadow another use of the same name if it temporarily hides access to the previous meaning.
- A covariant change in parameter or return type is a change the moves down the class hierarchy in the same direction as the child class.
- A contravariant change moves a parameter or return type up the class hierarchy in the opposite direction from the child class.

Reference

An Introduction to Object-Oriented Programming, Third Edition

by Timothy A. Budd

Thanks!