Classes and Interfaces

Inheritance
Polymorphism

Class Declaration

```
class  Point  {
       int  x, y;
       void move (int dX, int dY) {
              x += dX;
              y += dY;
       }
}
```

- A class defines the *structure* and the *behavior* of its instances (objects).
- An instance of `Point` has two `int` fields (that store its *state)*.
- An instance of `Point` can be a target of the `move` method (that can change its *state*).

Variable

- Each class declaration adds a *type*.
- Each ***variable*** of class type holds a reference to an object. (Cf. *pointer*)
- Variable Declaration: `Point p1, p;`
  - `p1` and `p` can hold references to `Point` instances.
- Object creation: `Point p = new Point();`
  - `new` allocates storage for an object and the default constructor `Point()` initializes the `Point`-object.
  - `int`-fields `x,y` are automatically initialized to `0`.

- Assignment : `p1 = p;`
  - The object reference is copied. Now the object is accessible through both `p1` and `p`.
- Initialization :
  - An instance field of a class type is initialized, *by default*, to `null`. However, a local variable in a method declaration is *not* initialized automatically.
- Implementation:
  - The formal parameters and the local variables in a method are allocated on ***run-time stack***.
  - The objects are allocated on the ***heap***.

- Accessing fields
  - `p.x`
- Invoking methods
  - `p.move(1,5)`
- `this`
  - The keyword `this` refers to the object the method is called *on*.

```
void move (int ax, int by) {
     this.x += ax;
     this.y += by;
}
```


Method Overloading

```
void move ( Point  p ) {
       x = p.x;
       y = p.y;
}
```

- The same name can refer to two different methods if the method *signatures* are different, where the ***signature*** is the sequence of types of formal parameters (and result) of the method.
- In Java, it is illegal for two methods in a class to have the same signature, even if the return types are different. (*Cf.* Ada, C++)

Resolution: "Most Specific" Algorithm

- Find all "overloaded" methods applicable to a call. If one of them matches exactly for all arguments, invoke that method.
- Otherwise, eliminate all methods that are less specific (more general) than others.
- If exactly one method remains, then invoke it. Else, report ambiguity.

Parameter passing mechanism

- A primitive type formal parameter is initialized by the value of the actual argument. So a method cannot modify the value of the actual argument.
  - types: `int, float, char,` etc
- A class type formal parameter is initialized by the the ***reference*** value of the actual argument. So a method cannot modify the reference, but can modify the object accessible through it.
  - types: `Point`, `Object`, etc
- Wrapper classes in Java 1.4 and autoboxing in Java 5 facilitate passing of primitive types by reference.
  - types: `Integer, Boolean, Character`, etc

Overloaded Constructors

```
Point ( ) {
   super();
}
```

– Default constructor (call to parent's constructor)

```
Point( Point p ) {
  this(p.x, p.y);
}
```

– `this` invokes another constructor

```
Point(int a,int b) {
    this.x = a;
    this.y = b;
}
```

– Overloaded constructor

- *A constructor gets invoked* after *storage is allocated for the object and its fields initialized, by default, but* before *the* `new` *returns.*

Subclass Declaration

```
import java.awt.Color;
class  ColoredPoint  extends Point {
        Color c = Color.RED;
        ColoredPoint(Color ic) {
               super(0, 0);
               c = ic;
        }
}
```

– A subclass ***extends*** a class by adding new fields and new methods.

– Subclass `ColoredPoint` inherits fields `x`, `y`, and method `move` from class `Point`.

Overloading *vs* Overriding

- To override a method ***m*** of the parent class, define a method ***m*** in the subclass that has *identical* signature.
- If the signature of the method ***m*** in the subclass is different from that of the method ***m*** in the class, then the method ***m*** is overloaded in the subclass.

Method Overriding : *Efficiency*

```
class Rectangle {
            ...
       float diagonal() {
          return Math.sqrt( a*a + b*b );
       }
}
class Square extends Rectangle {
       float diagonal() {
          return  (1.4142 * a);
       }
}
```

Method Overriding : *Customization by Reuse*

```
class  Stack {
            ...
}
class  BoundedStack extends Stack {
    int bound = 10;
    void push (int i) {
      if (top < bound)
            super.push(i);
      else
            System.out.println("eRrOr");
   }
}
```


Method Overriding : *Abnormality*

```
class Birds {
            ...
        boolean fly () {
          return  true;
        }
}
class Penguins extends Birds {
        boolean fly () {
          return  false;
        }
}
```


Dynamic Dispatching : Specificity

```
Bird   b  = new Bird();
Penguin p = new Penguin();
Bird  bp  = p;

   b.fly()  ==  true
   p.fly()  ==  false
   bp.fly() ==  false
```

The *method* run on the object referred to by `bp` for the *message* `fly()` is determined by the ***run-time type*** of the object (a penguin), and not by the ***compile-time type*** of the variable `bp` (a bird).

Polymorphism and Dynamic Binding

```
Bird[] bArray =
     { new Bird(), new Penguin(), new Bird() };
for (i = 0; i < bArray.length ; i++ ) {
   ...    bArray[i].fly();  ...
}
```

Motivation:

Enables reuse of existing code libraries to call methods on instances of new subclasses *without recompilation*.

Vendors can distribute classes in binary form and yet let users customize them.

Binding and Type System

Dynamic Binding in Java

```
class P {
   public void f(P p) {
     System.out.println("f(P) in P. ");
   }
}
class C extends P {
   public void f(P p) {
     System.out.println("f(P) in C. ");
   }
   public void f(C cp) {
     System.out.println("f(C) in C. ");
   }
}
```



```
class DynamicBinding {
  public static void
     main(String[] args) {
        P pp  = new P();
        C cc  = new C();
        P pc  = cc;
      pp.f(pp);   pp.f(cc);
      pc.f(pp);   pc.f(cc);
      cc.f(pp);   cc.f(cc);
  }
}
```


Abbreviated Example

```
class P {
  public void f(P p){}
}

class C extends P {
  public void f(P p){}
  public void f(C c){}
}


P pp  = new P();
C cc  = new C();
P pc  = cc;
```

```
pp.f(pp);
pp.f(cc);

pc.f(pp);
pc.f(cc);

cc.f(pp);
cc.f(cc);
```

Compile-time vs Run-time (binding)

pp.f(pp);	>=P f(P) {}	P f(P) {}
pp.f(cc);	>=P f(P) {}	P f(P) {}
	(coercion)	(coercion)
pc.f(pp);	>=P f(P) {}	**C f(P) {}**
pc.f(cc);	>=P f(P) {}	**C f(P) {}**
	(coercion)	(coercion)
cc.f(pp);	>=**C f(P) {}**	**C f(P) {}**
cc.f(cc);	>=*C f(C) {}*	*C f(C) {}*

Dynamic Binding in C#

```
class P {
   public virtual void f(P p) {
     System.Console.WriteLine("f(P) in P. ");
   }
}
class C : P {
   public override void f(P p) {
     System.Console.WriteLine("f(P) in C. ");
   }
   public void f(C cp) {
     System.Console.WriteLine("f(C) in C. ");
   }
}
```



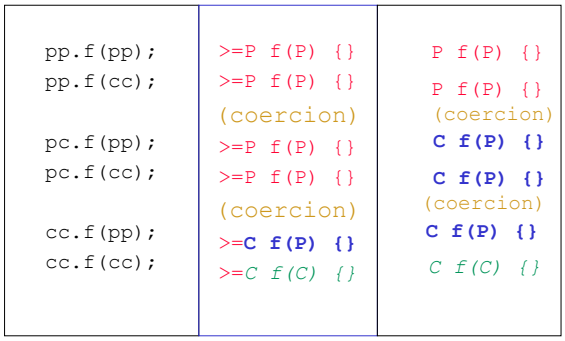
```
class DynamicBinding {
  public static void
     Main(string[] args) {
        P pp  = new P();
        C cc  = new C();
        P pc  = cc;
      pp.f(pp);   pp.f(cc);
      pc.f(pp);   pc.f(cc);
      cc.f(pp);   cc.f(cc);
  }
}
```


Abbreviated Example

```
class P {
  public virtual void
  f(P p){}
}

class C extends P {
  public override void
  f(P p){}
  public void f(C c){}
}


P pp  = new P();
C cc  = new C();
P pc  = cc;
```

```
pp.f(pp);
pp.f(cc);

pc.f(pp);
pc.f(cc);

cc.f(pp);
cc.f(cc);
```

Compile-time vs Run-time (binding)



Static Binding in C# (with **new** (*no* **virtual**))

```
class P {
   public void f(P p) {
     System.Console.WriteLine("f(P) in P. ");
   }
}
class C : P {
   public new void f(P p) {
     System.Console.WriteLine("f(P) in C. ");
   }
   public void f(C cp) {
     System.Console.WriteLine("f(C) in C. ");
   }
}
```



```
class Binding {
  public static void
     Main(string[] args) {
        P pp  = new P();
        C cc  = new C();
        P pc  = cc;
      pp.f(pp);   pp.f(cc);
      pc.f(pp);   pc.f(cc);
      cc.f(pp);   cc.f(cc);
  }
}
```


Abbreviated Example

```
class P {
  public void f(P p){}
}

class C extends P {
  public new void f(P
  p){}
  public void f(C c){}
}


P pp  = new P();
C cc  = new C();
P pc  = cc;
```

```
pp.f(pp);
pp.f(cc);

pc.f(pp);
pc.f(cc);

cc.f(pp);
cc.f(cc);
```

Compile-time (binding) vs Run-time

pp.f(pp); pp.f(cc); pc.f(pp); pc.f(cc); cc.f(pp); cc.f(cc);	P f(P) {} P f(P) {} (coercion) P f(P) {} P f(P) {} (coercion) **C f(P) {}** *C f(C) {}*	P f(P) {} P f(P) {} (coercion) P f(P) {} P f(P) {} **C f(P) {}** *C f(C) {}*

Inheritance – Overloading

C++ *vs* Java


```
class Parent {
    String method(int i) {
           return "Parent.method(int)"; }
}
class Child extends Parent {
    String method(int i, boolean b) {
           return "Child.method(int,boolean)"; }
}
class GrandChild extends Child {
    String method(int i, boolean b) {
            return "GrandChild.method(int,boolean)"; }
}
class Overload {
  public static void main(String[] args) {
    Child c = new Child();
    GrandChild gc = new GrandChild();
    System.out.println(c.method(5,true) + "\t" + c.method(6));
    System.out.println(gc.method(1,false)+ "\t" + gc.method(2));
  }
}
```



```
#include <iostream>
class Parent {
public:
    char* method(int i) {
                return "Parent.method(int)"; }
};
class Child : public Parent {
public:
    char* method(int i, bool b) {
                 return "Child.method(int,boolean)"; }
};
class GrandChild : public Child {
public:
    char* method(int i, bool b) {
                 return "GrandChild.method(int,boolean)"; }
};

int main() {
   Child* c = new Child();
   GrandChild* gc = new GrandChild();

   cout << c->method(5,true) << "\t" << c->method(6);       // ERROR
   cout << gc->method(1,false) << "\t" << gc->method(2);    // ERROR
}
```

- `class Object` is the root of the tree-structured class hierarchy.
  - `class Point extends Object { ... }`
- ***Multiple inheritance*** of classes is prohibited.
  - There is no consensus about how to resolve name conflicts, automatically.
  - This can improve performance.
    - `super()`, in default constructor, is unambiguous.
- In C++ parlance, all Java methods are ***virtual.***

keyword `final`

- `final` field
  - primitive type: constant value
  - reference type: fixed object
    - mutable type : object state changeable
      - e.g., Point
    - immutable type : object state constant
      - e.g., String
- `final` instance method
  - method cannot be over-ridden in a subclass.
  - method call can be optimized by ***in-lining***.
- `final` class

Abstract Classes

- Classes *must implement* all the methods.
- Abstract classes can *implement some methods,* leaving the rest to subclasses.
- Interfaces *must specify only the signatures* of all the methods.
  - Interfaces and abstract classes cannot be instantiated (to create objects).
  - Abstract classes provide a *framework* by factoring commonality among classes.

Interface Declaration

```
interface Table {
      boolean   isEmpty();
      void      insert(int  key, Object x);
      Object    lookUp(int  key);
}
```

- An interface specifies constants and signatures of the (public) methods.
- An interface `extends` super-interfaces.
- A class `implements` interfaces.

```
import  java.util.Hashtable;
class  MyTable  implements  Table {
      Hashtable  v = new  Hashtable();
      public boolean isEmpty() {
              return   v.isEmpty();
      }
      public void    insert(int  key, Object x) {
              v.put(new Integer( key ), x);
      }
      public Object  lookUp(int  key) {
              return v.get(new Integer( key ));
      }
}
```


Applications of Interfaces

- Integrating multiple implementations of an abstract data type
  - Unifying classes that are heterogeneous but that exhibit a common behavior.
  - Enables reuse of "higher-level" code or test code.
    - Dynamic method lookup.
- Support for multiple inheritance
  - A class can implement multiple interfaces.
  - Enables code sharing by approximating multiple inheritance of classes.
- Approximating enumerated types
  - Java 5 supports enumerated types explicitly.

Odds and ends

- In Java, the class declaration and the implementation of methods are stored in the same place for ease of maintenance.
- In Java 1.0, *object references* point at an intermediate data structure which stores the ***type*** information and the ***current heap address*** of the actual instance data. In Java 1.1, there are direct pointers to heap (that may be relocated by GC).

- Java does not allow explicit manipulation of pointers.
- `finalize()` method is defined to reclaim non-Java resources (such as file handles).
  It is called by Java runtime system automatically before an object is garbage collected.
- In C++, pointers can refer to both the heap and the run-time stack.

Static Variables and Static Methods

```
class  Point  {
       int  x, y = 0;
       Point() {
            count++ ;
       }

  static int count;
  static {
             count = 0;
  }
  static Point origin = new Point();
}
```


- Each class has a single copy of the static variables. These can be accessed by all instances of a class via methods.
- Static variables are "encapsulated" global variables that can be accessed by prefixing them with *classname*.
- The different instances of a class can *communicate* through these "shared" variables.
- Static blocks are used to initialize static variables and are run when a class is loaded.
- Static methods can refer only to static variables (and not to instance variables).

Covariant Typing in Java 5

SKIP


```
class A { A m() {};}

class B extends A { B m() {}; }

Pro:

B x = new B(). m();

Con:

Eiffel example of device-printer-CD
```