Practice of Programming using Java
Lecture 6 June 23, 2006. 6-8 pm

Rushikesh K Joshi
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Interfaces

Specify externally observable member functions

Only the prototype declarations are specified

implementation is not specified and left for implementor class/classes.

No private data members

No control constructs needed for interface specifications

Many implementations of the same interface

Part of application
code that knows the
interface and not the class can use ----------> instance of class that
implements the interface

Interfaces are implemented by classes

```
interface Collection {
        public Object fetch();
        public void add(Object o);
}
class Stack implements Collection {
        ...
        public Object fetch () {..}
        public void add (Object o) {...}
}
class Queue implements Collection {
        public Object fetch () {...}
        public void add (Objet o) {...}
}
```

Fields in Interfaces

```
interface MathsFunctions {
        double PI = 3.142;
        public double sqrt (double num);
        ...
}
```

fields are static and final.

Final field: value is set exactly once and cannot be changed.

(final fields are initialized before use)

Extending Classes

```
class A {
        private int i;
        A () {i=10;}
        public void f() {System.out.println("A::f " + i);}
}
class B extends A {
        private int j;
        B() {j=20;}
        public void g() {System.out.println("B::g " + j);}
}
```

Overriding Member Functions

```
class A {
        private int i;
        A () {i=10;}
        public void f() {System.out.println("A::i " + i);}
}
class B extends A {
        private int j;
        B() {j=20;}
        public void f() {System.out.println("B::i", + j);}
        public void g() {System.out.println("B::g " + j);}
}
```

Polymorphism

Try the below code----------

```
class Polymorphism {
        public static void main (String args[]) {
                A a = new B();
                a.f();
                // a.g(); <---  compile time error
        }
}
```

Observe the dynamic binding of message dispatched, to method defined in the class of instance that is being used.

Accessibility depends upon the type being used to refer to the instance

Final Methods cannot be overriden

```
class A {
        public final void f() {System.out.println ("A::f");}
}

class B extends A {
        public void f() {System.out.println("B::f");} <-- not allowed

}
```

Final Classes cannot be extended

```
Final class A {
        public final void f() {System.out.println ("A::f");}
}

class B extends A {  <--- not allowed
        public void g() {System.out.println("B::g");}

}
```

Super and this

Can be used in nonstatic methods

super.f() will dispatch to superclass's implementation of f

```
class B extends A {

        public B f() {
                System.out.println ("A::f");
                super.f();
                return this;
        }
}
```

Abstract Classes

Specify partial behavior

Some behavior is unknown and can be left out for variants

Mostly used when there are multiple possible variant subclasses

Cannot be instanciated

```
abstract class A {
        protected int i;
        public void m() {i=20;}
        public abstract int n();
}
class B extends A {
        public int n() {return i;}
}
```

Abstract classes can provide partial implementations

```
interface Collection {
        public Object fetch();
        public void add(Object o);
        public int size();
}
abstract class CollectionPartImple implements Collection {
        protected Vector collection;
        public int size () {return collection.size();}
        .....
}
class Stack extends CollectionPartImple implements Collection {    ..}
class Queue implements Collection { ..}
```