

Programming Language Concepts: Lecture 4

Madhavan Mukund

Chennai Mathematical Institute

`madhavan@cmi.ac.in`

`http://www.cmi.ac.in/~madhavan/courses/pl2009`

PLC 2009, Lecture 4, 28 January 2009

Class hierarchy

- ▶ Subclasses **inherit** attributes from parent class
- ▶ Subclasses can add functionality
 - ▶ A subclass is more specific than its parent
 - ▶ Subclasses can be used in place of the parent class
- ▶ `class Employee {...}`

```
class Manager extends Employee{  
    private String secretary;  
    public boolean setSecretary(name s){ ... }  
    public String getSecretary(){ ... }  
}
```

```
Employee e = new Manager()
```

Overriding and dynamic dispatch

- ▶ Subclass can **override** parent class method
 - ▶ Function name and signature must both match
 - ▶ `public equals(Date d)` does not override `public equals(Object o)`
- ▶ **Dynamic dispatch** allows each object to “know” which method to use.

```
class Employee { ... public double bonus(double p) ...}
```

```
class Manager extends Employee{  
    ... public double bonus(double p) ...  
}
```

```
Employee e = new Manager();  
...  
print(e.bonus(x));
```

Java class hierarchy

- ▶ No multiple inheritance — tree-like
- ▶ Universal superclass `Object`
- ▶ Useful methods defined in `Object`

```
boolean equals(Object o) // defaults to pointer equality
```

```
String toString() // converts the values of the  
// instance variable to String
```

- ▶ To print `o`, use `System.out.println(o+"");`

Subclasses, subtyping and inheritance

- ▶ Class hierarchy provides both **subtyping** and **inheritance**
- ▶ **Subtyping**
 - ▶ Compatibility of interfaces.
 - ▶ **B** is a subtype of **A** if every function that can be invoked on an object of type **A** can also be invoked on an object of type **B**.
- ▶ **Inheritance**
 - ▶ Reuse of implementations.
 - ▶ **B** inherits from **A** if some functions for **B** are written in terms of functions of **A**.

Subtyping vs inheritance

Consider the following classes

- ▶ `queue`, with methods `insert-rear`, `delete-front`
- ▶ `stack`, with methods `insert-front`, `delete-front`
- ▶ `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`

Subtyping vs inheritance

Consider the following classes

- ▶ `queue`, with methods `insert-rear`, `delete-front`
- ▶ `stack`, with methods `insert-front`, `delete-front`
- ▶ `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`

What are the subtype and inheritance relationships between these classes?

Subtyping vs inheritance

- ▶ `queue`, with methods `insert-rear`, `delete-front`
- ▶ `stack`, with methods `insert-front`, `delete-front`
- ▶ `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`

Subtyping vs inheritance

- ▶ `queue`, with methods `insert-rear`, `delete-front`
- ▶ `stack`, with methods `insert-front`, `delete-front`
- ▶ `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`

Subtyping

- ▶ `deque` has more functionality than `queue` or `stack`
- ▶ `deque` is a subtype of both these types

Subtyping vs inheritance

- ▶ `queue`, with methods `insert-rear`, `delete-front`
- ▶ `stack`, with methods `insert-front`, `delete-front`
- ▶ `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`

Subtyping

- ▶ `deque` has more functionality than `queue` or `stack`
- ▶ `deque` is a subtype of both these types

Inheritance

- ▶ Can suppress two functions in a `deque` and use it as a `queue` or `stack`
- ▶ Both `queue` and `stack` inherit from `deque`

Subclasses, subtyping and inheritance

- ▶ Class hierarchy provides both **subtyping** and **inheritance**
- ▶ **Subtyping**
 - ▶ Compatibility of interfaces.
 - ▶ **B** is a subtype of **A** if every function that can be invoked on an object of type **A** can also be invoked on an object of type **B**.
- ▶ **Inheritance**
 - ▶ Reuse of implementations.
 - ▶ **B** inherits from **A** if some functions for **B** are written in terms of functions of **A**.

Using one idea (hierarchical classes) to implement both concepts blurs the distinction between the two

Abstract classes

- ▶ Collect together classes under a common heading
- ▶ Classes `Circle`, `Square` and `Rectangle` are all shapes
- ▶ Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`

Abstract classes

- ▶ Collect together classes under a common heading
- ▶ Classes `Circle`, `Square` and `Rectangle` are all shapes
- ▶ Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- ▶ We want to force every shape to define a function
`public double perimeter()`

Abstract classes

- ▶ Collect together classes under a common heading
- ▶ Classes `Circle`, `Square` and `Rectangle` are all shapes
- ▶ Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- ▶ We want to force every shape to define a function

```
public double perimeter()
```

- ▶ Define a function in `Shape` that returns an absurd value

```
public double perimeter() { return -1.0; }
```

- ▶ Rely on the subclass to redefine this function

Abstract classes ...

- ▶ A better solution
 - ▶ Provide an **abstract definition** in `Shape`
`public abstract double perimeter();`
- ▶ Forces subclasses to provide a concrete implementation

Abstract classes ...

- ▶ A better solution
 - ▶ Provide an **abstract definition** in **Shape**
`public abstract double perimeter();`
- ▶ Forces subclasses to provide a concrete implementation
- ▶ Cannot create objects from a class that has abstract functions
- ▶ **Shape** must itself be declared to be **abstract**

```
abstract class Shape{  
    ...  
    public abstract double perimeter();  
    ...  
}
```


Abstract classes ...

- Can still declare variables whose type is an abstract class

```
Shape sarr[] = new Shape[3];
```

```
Circle c = new Circle(...);  sarr[0] = c;
```

```
Square s = new Square(...);  sarr[1] = s;
```

```
Rectangle r = new Rectangle(...);  sarr[2] = r;
```

```
for (i = 0; i < 2; i++){  
    size = sarr[i].perimeter();  
    // each sarr[i] calls the appropriate method  
    ...  
}
```

Generic functions

- Use abstract classes to specify generic properties

```
abstract class Comparable{  
    public abstract int cmp(Comparable s);  
    // return -1 if this < s, 0 if this == 0,  
    //          +1 if this > s  
}
```

Generic functions

- Use abstract classes to specify generic properties

```
abstract class Comparable{  
    public abstract int cmp(Comparable s);  
    // return -1 if this < s, 0 if this == 0,  
    //           +1 if this > s  
}
```

- Now we can sort any array of objects that extend `Comparable`

```
class Sortfunctions{  
    public static void quicksort(Comparable[] a){  
        ...  
        // Usual code for quicksort, except that  
        // to compare a[i] and a[j] we use a[i].cmp(a[j])  
    }  
}
```

Generic functions ...

```
► class Sortfunctions{  
    public static void quicksort(Comparable[] a){  
        ...  
    }  
}
```

► To use this definition of `quicksort`, we write

```
class Myclass extends Comparable{  
    double size;    // quantity used for comparison  
    ...  
    public int cmp(Comparable s){  
        if (s instanceof Myclass){  
            // compare this.size and ((Myclass) s).size  
            // Note the cast to access s.size  
            ...  
        }  
    }  
}
```

Mutiple inheritance

- ▶ How do can we sort `Circle` objects?
 - ▶ `Circle` already extends `Shape`
 - ▶ Java does not allow `Circle` to also extend `Comparable`!

Mutiple inheritance

- ▶ How do can we sort `Circle` objects?
 - ▶ `Circle` already extends `Shape`
 - ▶ Java does not allow `Circle` to also extend `Comparable`!
- ▶ An `interface` is an abstract class with no concrete components

```
interface Comparable{  
    public abstract int cmp(Comparable s);  
}
```

Mutiple inheritance

- ▶ How do can we sort `Circle` objects?
 - ▶ `Circle` already extends `Shape`
 - ▶ Java does not allow `Circle` to also extend `Comparable`!
- ▶ An **interface** is an abstract class with no concrete components

```
interface Comparable{  
    public abstract int cmp(Comparable s);  
}
```

- ▶ A class that extends an interface is said to “implement” it:

```
class Circle extends Shape implements Comparable{  
    public double perimeter(){...}  
    public int cmp(Comparable s){...}  
    ...  
}
```

Multiple inheritance

- ▶ How do we sort `Circle` objects?
 - ▶ `Circle` already extends `Shape`
 - ▶ Java does not allow `Circle` to also extend `Comparable`!
- ▶ An **interface** is an abstract class with no concrete components

```
interface Comparable{  
    public abstract int cmp(Comparable s);  
}
```

- ▶ A class that extends an interface is said to “implement” it:

```
class Circle extends Shape implements Comparable{  
    public double perimeter(){...}  
    public int cmp(Comparable s){...}  
    ...  
}
```

- ▶ Can implement multiple interfaces