Object – Oriented Technology

Chapter 1: Typed Objects

Suad Alagić

Springer 2016

```
interface TwoDPoint {
  int getX();
  int setX(int x);
  int getY();
  int setY(int y);
}
```

```
class TwoDPoints implements TwoDPoint {
  private int x;
  private int y;
  public int getX(){
        return x;
  }
  public void setX(int x) {
        this.x=x;
  }
  public int getY() {
        return y;
  }
  public void setY(int y) {
        this.y=y;
  }
}
```

OBJECT PROPERTIES

- object identity
- object state
- methods applicable to the object

MESSAGES

TwoDPoint p;
p.setX(5); p.setY(10);

a.m(a1,a2,. . .,an)

CREATING OBJECTS

```
TwoDPoint obj=new TwoDPoints();
obj.setX(5); obj.setY(10);
```

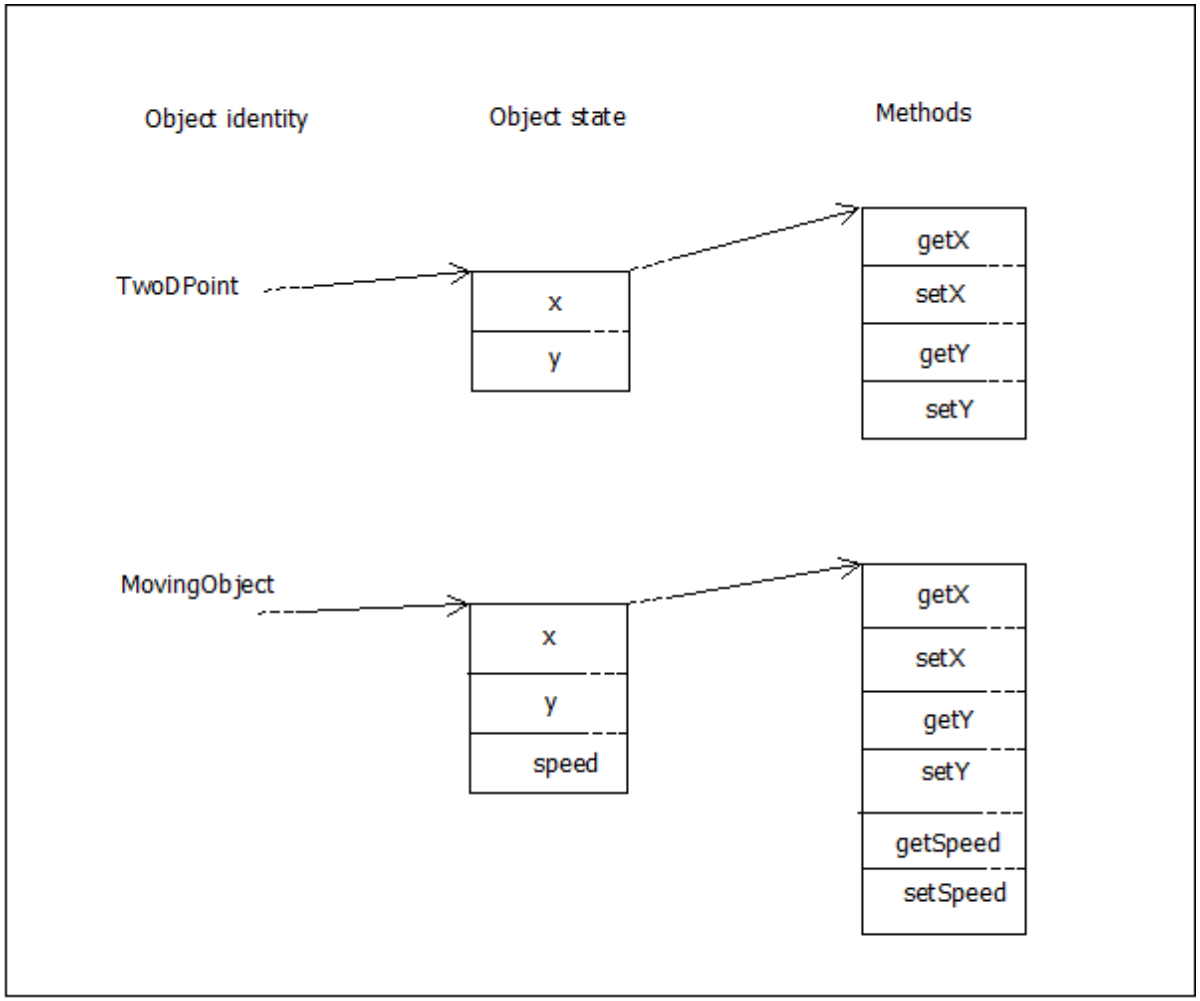
```
TwoDPoints(int xVal, int yVal) {
  x=xVal; y=yVal;
}
```

```
TwoDPoint obj= new TwoDPoints(5,100);
```

```
interface MovingObject extends TwoDPoint {
  float getSpeed();
  void setSpeed(float newSpeed);
}
```

```
class MovingObjects extends TwoDPoints {
  private float speed;
  public float getSpeed() {
        return speed;
  }
  public void setSpeed(float newSpeed) {
        speed=newSpeed;
  }
}
```

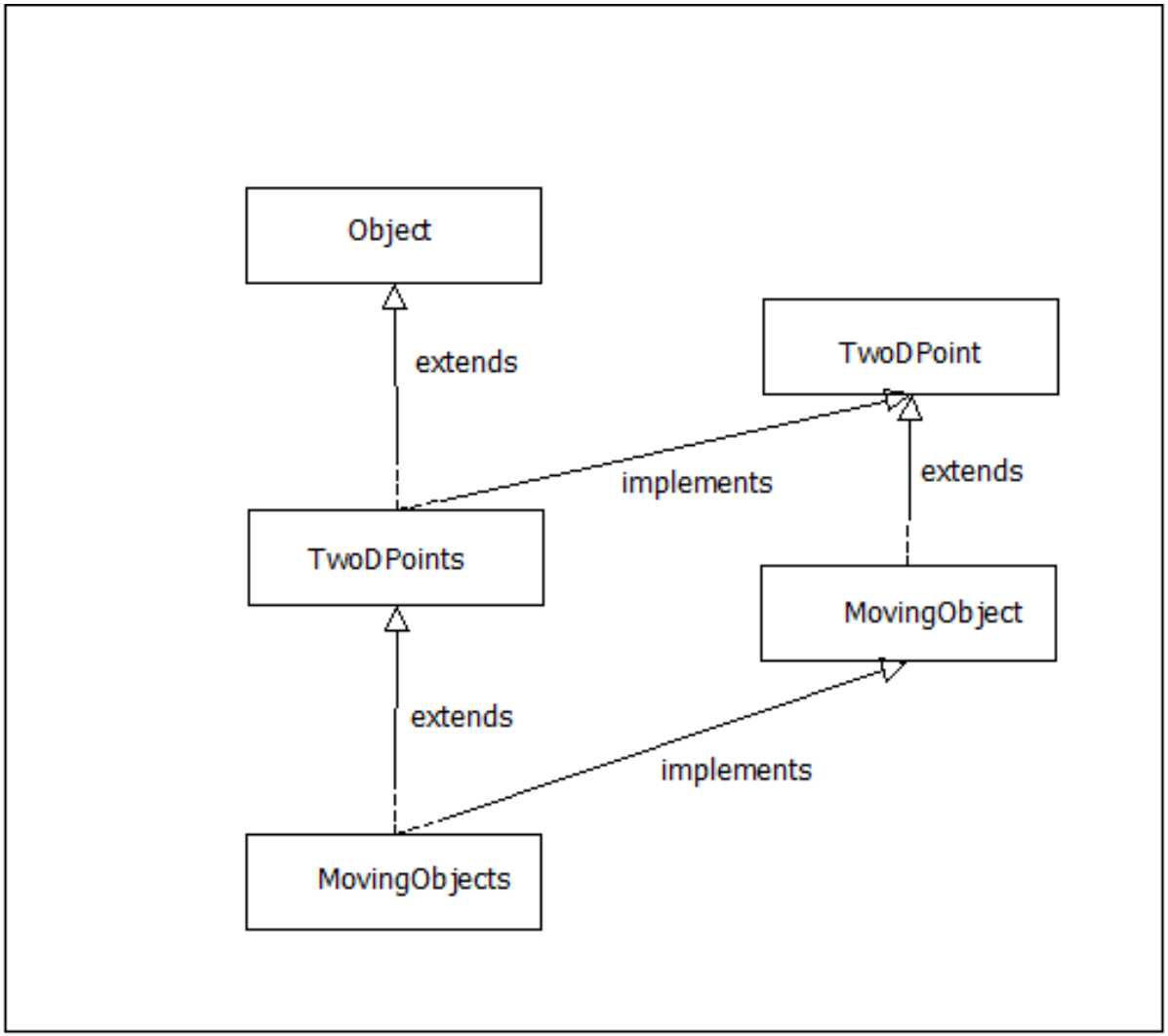
OBJECT: STATE and METHODS



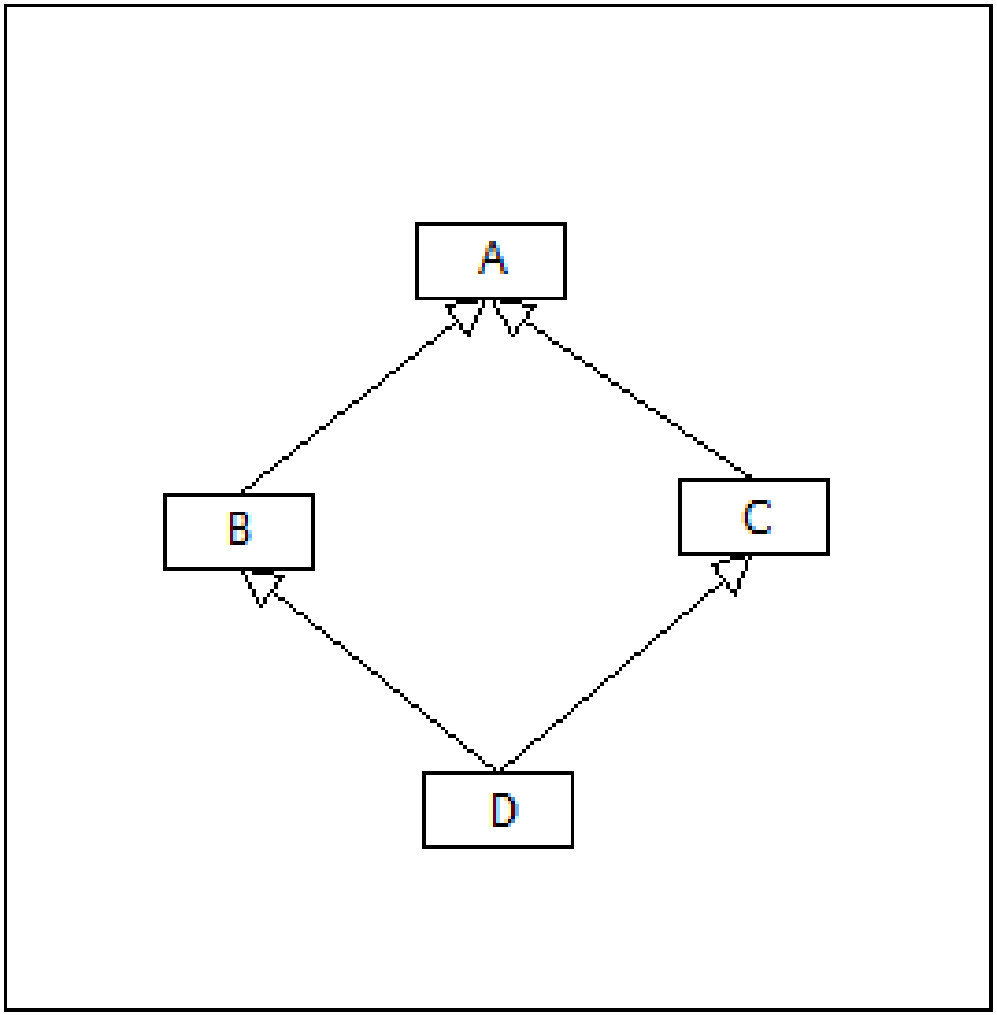
CLASS Object

```
public class Object
  public boolean equals(Object x);
  public Class getClass();
  // other methods
}
```

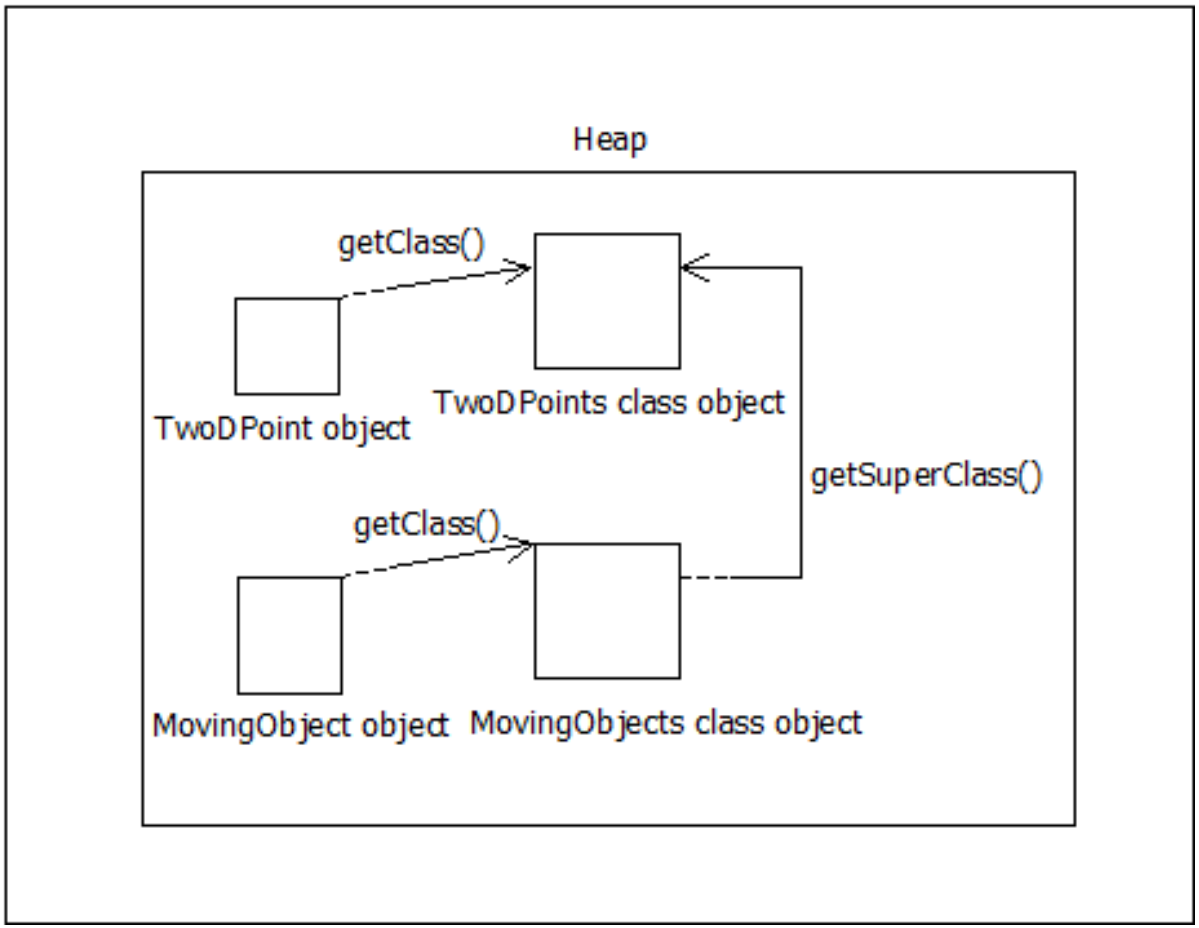
INHERITANCE



MULTIPLE INHERITANCE



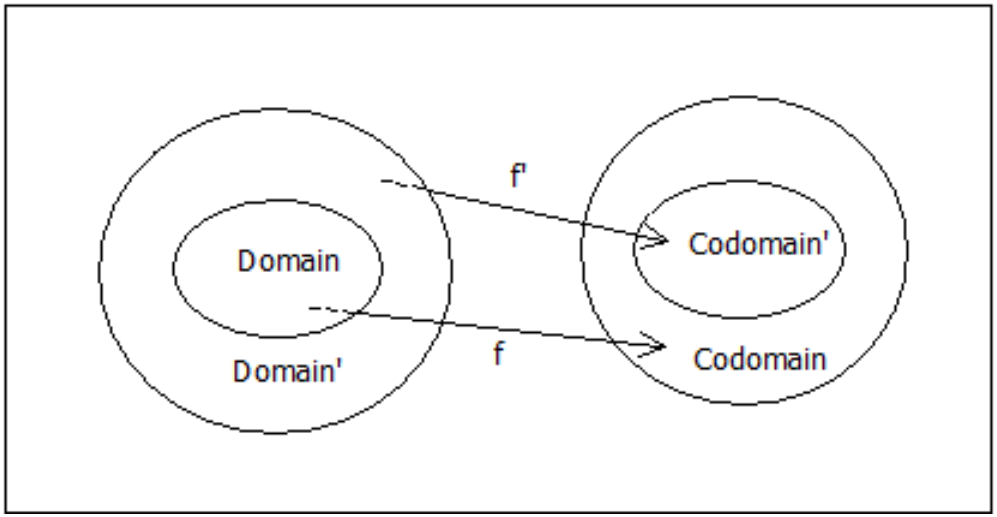
SUPERCLASS


Heap
getClass()
TwoDPoint object
TwoDPoints class object
getSuperClass()
getClass()
MovingObject object
MovingObjects class object

CLASS Class

```
public final class Class {
  // methods for accessing field signatures
  // methods for accessing constructor signatures
  // methods for accessing method signatures
  public Class getSuperClass();
}
```

FUNCTIONS



FUNCTION SUBTYPING

f: Domain $\rightarrow$ CoDomain

f′: Domain′ $\rightarrow$ CoDomain′

Domain $\subseteq$ Domain′

CoDomain′ $\subseteq$ CoDomain

FUNCTION SUBTYPING

$T1' \rightarrow T2'$

$T1 \rightarrow T2$

$T1 <: T1'$ (contravariance) and $T2' <: T2$ (covariance)

COVARIANCE

```
public class Object {
  public Object clone()
// other methods
}
```

```
public class TwoDPoints {
  public TwoDPoints clone()
// other methods
}
```

FIELD INHERITANCE

```
class A {
  Ta f;
  Ta getf(){return f; }
  void setf(Ta value){ f = value; }
}

class B extends A {
  Tb f;
  Tb getf(){return f; }
  void setf(Tb value){f = value; }
}
```

Tb $<:$ Ta and Ta $<:$ Tb

ASSIGNMENT

```
TwoDPoint x = new TwoDPoints();
MovingObject y = new MovingObjects();
x=y;
```

DYNAMIC BINDING

```
public class Vehicle {
  private int VIN;
  private String make;
  public boolean equals(Object x) {
        return (VIN = (Vehicle)x.VIN);
  }
  // other methods
}

Object x = new Object();
Object y = new Object();
Vehicle xV = new Vehicle();
Vehicle yV = new Vehicle();
x=xV; y=yV;
. . . x.equals(y). . .
```

VIRTUAL METHODS

```
public class Object {
  public virtual boolean equals(Object x);
  // other methods
}
```

```
public class Vehicle {
  private int VIN;
  private String make;
  public override boolean equals(Object x) {
        return (VIN = (Vehicle)x.VIN);
  }
  // other methods
}
```

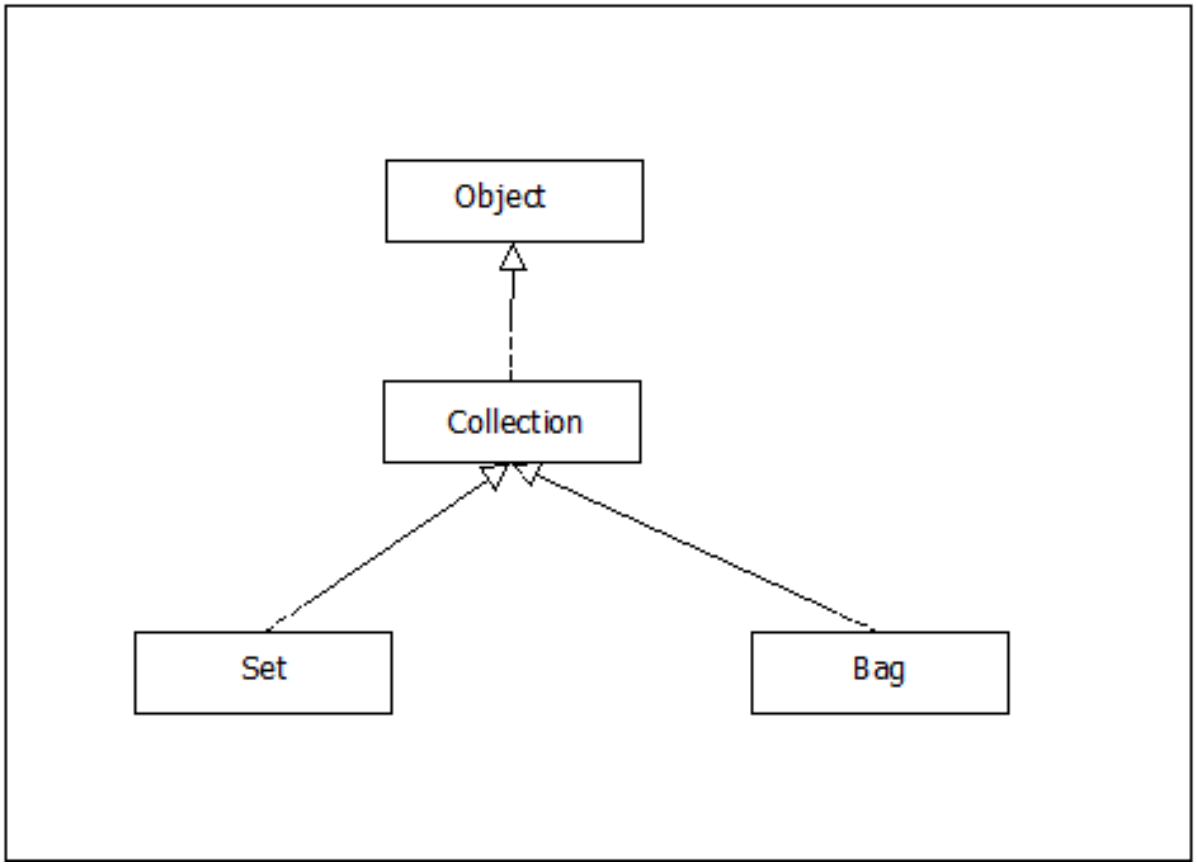
FINAL METHODS

```
public class Object {
  public final Class getClass();
  // other methods
}
```

STATIC METHODS

```
public class Vehicle {
// fields;
  public static int numberOfVehicles();
  // other methods
}
```

COLLECTION TYPES



COLLECTION TYPES

```
public interface Collection{
  public boolean isMember(Object x);
  public void add(Object x);
  public void remove(Object x);
}

public interface Set extends Collection {
  public Set union(Set s);
  public Set intersection(Set s);
}
```

PROBLEMS with COLLECTIONS

Collection employees;

Emloyee emp = **new** Employee();
employees.add(emp);

Department dept = **new** Department();
employees.add(dept); ???

PROBLEMS with COLLECTIONS

```
for (Employee emp: employees)
  emp.displaySalary(); ???

for (Object emp: employees)
  emp.displaySalary(); ???

for (Object emp: employees)
  (Employee)emp.displaySalary();
```

EXCEPTION HANDLING

```
try {
        for (Object emp: employees)
        (Employee)emp.displaySalary();
        }
  catch (ClassCastException classEx )
        {exception handling }
```

```
public interface Collection<T> {
public boolean isMember(T x);
public void add(T x);
public void remove(T x);
}
```

```
Collection<Employee> employees;
Employee emp = new Employee();
employees.add(emp)

Collection<Employee> employees;
Department dept = new Department();
employees.add(dept) ???

for (Employee emp: employees)
  emp.displaySalary();
```

BOUNDED PARAMETRIC TYPES

OrderedCollection<T>

OrderedCollection<T **extends** Comparable<T>>

```
public interface Comparable<T> {
  public int compareTo(T x);
// other comparison methods
}

  public interface OrderedCollection<T extends Comparable<T>>
                    extends Collection<T> {
// . . .
}
```

Employee **implements** Comparable<Employee>

OrderedCollection<Employee>

IMPLEMENTED PARAMETRIC CLASS

```
public class OrderedCollection<T extends Comparable<T>>
                    implements Collection<T> {
  private LinkedList<T> elements;
  public OrderedCollection() {
        elements = new LinkedList<T>();}
  public boolean isMember(Object e) {
        return elements.contains(e); }
  public void add(T e) {
  if (¬ elements.contains(e)) {
  for (int i = 0; i < elements.size() - 1; i++) {
        if (elements.get(i).compareTo(e) ≤ 0 ∧
        elements.get(i + 1).compareTo(e) > 0)
        elements.add(e); }
  public void remove(Object e) {
        if (elements.contains(e))
        elements.remove(e); }
}
```

ABSTRACT CLASSES

```
public abstract class AbstractBag<T> {
  public abstract int size();
  public abstract int occurrences(T x);
  protected class Member<T> {. . . }
  // code for union, intersection and copy
}
```

ABSTRACT CLASSES

```
protected class Member<T> {
  T value;
  int count;
  @Override
  public boolean equals(Object other) {
        Member mem = (Member) other;
        return this.value.equals(mem.value);
  }
  // other methods
}
```

ABSTRACT CLASSES

```
  public abstract < B extends AbstractBag<T>> B copy(); {
// code for this method
}
  public < B extends AbstractBag<T>> B union(B otherBag) {
  // code for this method
}
  public <B extends AbstractBag<T>> B intersection(B otherBag){
  //code for this method
}
```

OrderedBag< T **extends** Comparable<T>> **extends**
AbstractBag<T>
Employee **implements** Comparable<Employee>

OrderedBag<Employee>.

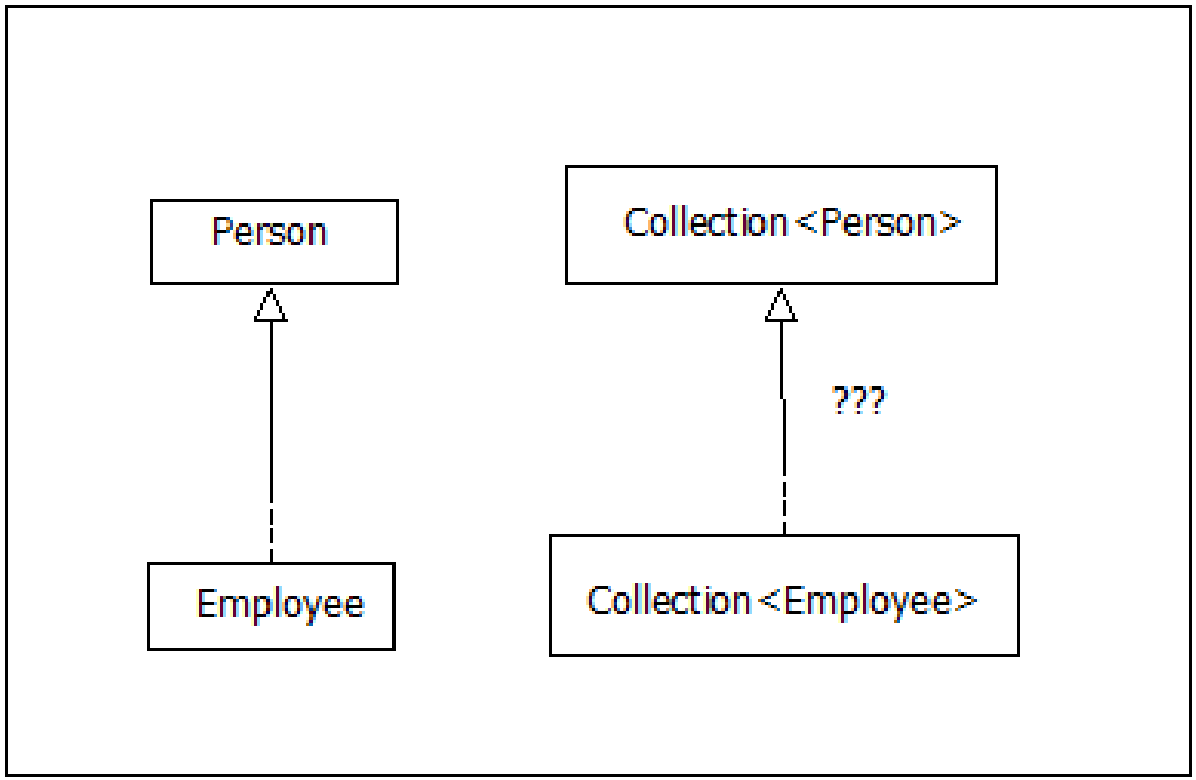
ITERATOR

```
public interface Iterator<T> {
  boolean hasNext();
  T next();
  void remove();
}
```

ABSTRACT IMPLEMENTATION

```
public < B extends AbstractBag<T>> B union(B otherBag) {
        B result, other;
if (this.size() < otherBag.size()) {
        result = otherBag.copy();
        other = this; }
else {
        result = this.copy();
        other = otherBag; }
Iterator<Member> itr = other.iterator();
        Member m, n;
        while (itr.hasNext()) {
        m = itr.next();
        n = other.getMember(m);
        if (n ≠ null)
        set count of n to max(m,n);
        else result.addMember(m);
        }
return result;
}
```

PARAMETRIC TYPES AND INHERITANCE



class Employee **extends** Person {. . .} .

Employee $<:$ Person

Collection<Employee> $<:$ Collection<Person> ? ? ?

Employee[] $\not<:$ Person[]