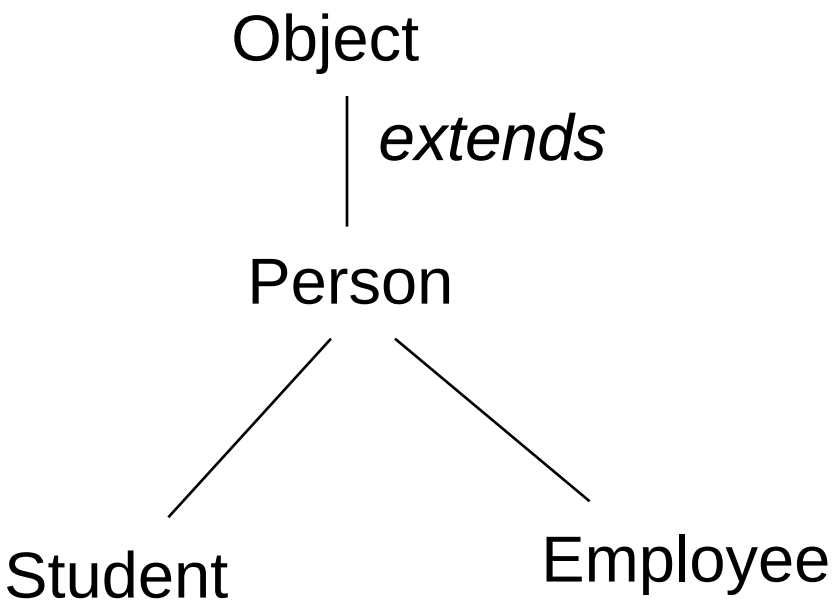
Method Resolution

- **Inheritance**
  - **inheriting methods**
  - **overriding methods**
  - **use of super keyword**
- **Interfaces**
  - **examples**
  - **cloning: shallow and deep**

Inheritance

- **Extending a class by a subclass allows for code sharing**
  - **Objects in subclass can *inherit* methods from the superclass**
  - **e.g., Binary_Expr and Plus_Expr**
- **Overriding**
  - **Methods with same name and signature in subclass can *override* the same named method in the superclass, for subclass objects**

Overriding Methods



Person *extends* Object means Person is a *subclass* of Object class; Object is a *superclass* of Person class

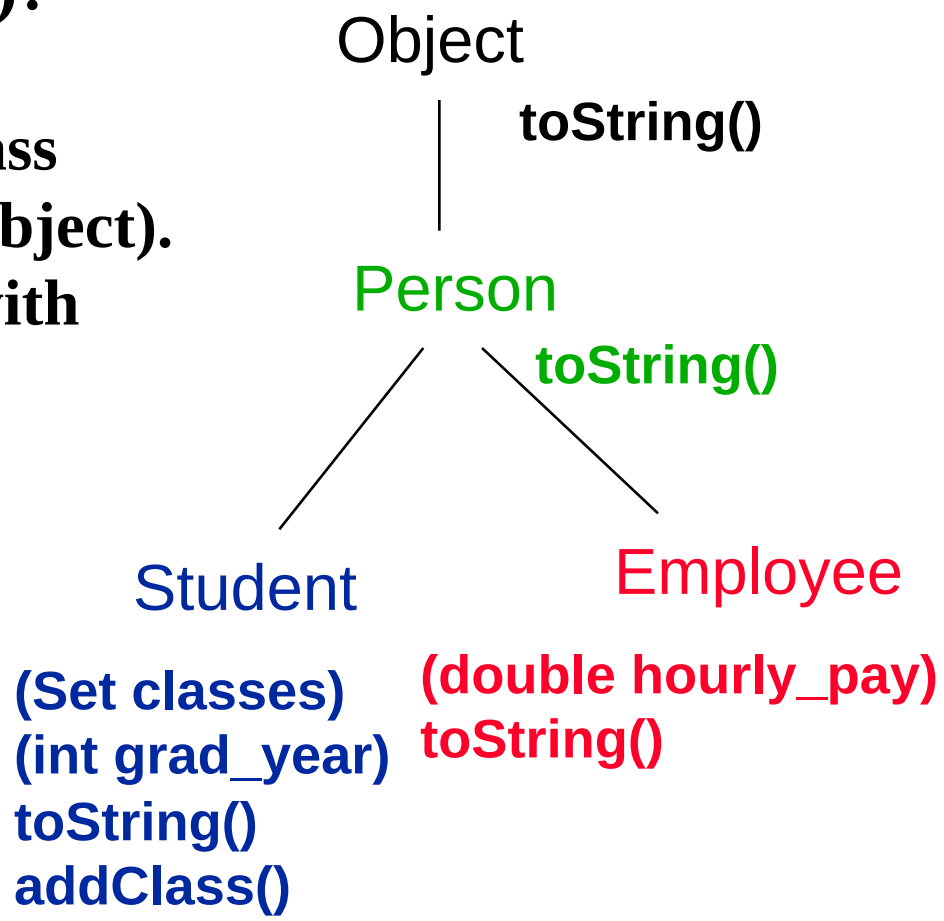
Inheritance Hierarchy

Method resolution performed at run-time, using the hierarchy

Method Overriding or Dynamic Binding

How to resolve x.toString()?

1. At run-time, determine class of the object (e.g., x is an A object).
2. Start lookup for method with same signature in class A.
3. Proceed up inheritance hierarchy until find closest superclass with same signature (i.e., method toString()); this may be class A itself



Use of Super

super.method is bound at compile-time to method selected as though receiver were of the superclass type of the class in which the invocation appears

```
public class Employee extends Person //using assn4's
{  private double hourly_pay;          //Person class
   public Employee(String name, Person mother,
       Person father, String title, int born, int died,
       char gender,double pay)
   {   super(name,mother,father,title,born,died,
             gender);//calls superclass constructor
       hourly_pay = pay;
   }
   public String toString()
                     //uses Person class toString()
   {   String s = "\n" + super.toString();
       if (hourly_pay != 0) s = s + " Hourly pay is " +
             hourly_pay + "\n";
       return s;
   }
```

Class Student

```
import cs111.util.Set.*;
public class Student extends Person
{  private Set classes;
   private int grad_year;

   public Student(String name, Person mother,
       Person father, String title, int born,
       int died, char gender,int year)
   {   super(name,mother,father,title,born,
             died,gender);
       grad_year = year;
       classes = new Set();
   }
```

Class Student

```
public void addClass(String cl)
   {   classes.addTo(cl);
   }
   public String toString()
   {   //uses Person's toString()
       String s = "\n" + super.toString();
       if (grad_year != 0) s = s +
             "\n  Graduation year is " + grad_year +
       "\n";
       if (!(classes.isEmpty()))
             s = s + "  Taking classes: \n" +
               classes.toString();//Set's toString()
       return s;
   }
}
```

main() in Employee Class

```
public static void main(String args[])
  { Employee bgr = new Employee("Barbara Ryder", null,
      null,"Professor of Computer Science",1947,0,'F',
      .50);
  Employee vs = new Employee("Vince Sgro",null,null,
            "Adjunct Professor",0,0,'M',.25);
  Student bar = new Student("Beth Ryder",bgr,null,
      "Med School Student",1973,0,'F',1999);
  bar.addClass("101");
  bar.addClass("123");
  Student aer = new Student("Andrew Ryder",bgr,null,
      "CS major",1975,0,'M',1996);
  }
```

/studempl/*java

Output

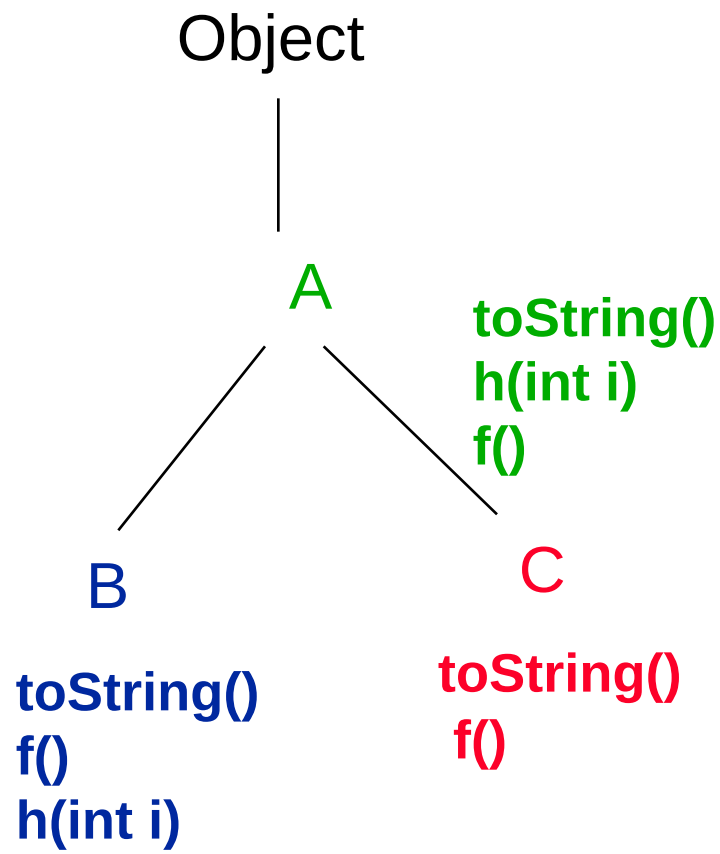
```
Beth Ryder, Med School Student
  Graduation year is 1999
  Taking classes:
        101
        123
Barbara Ryder, Professor of Computer Science Hourly
   pay is 0.5

Andrew Ryder, CS major
  Graduation year is 1996

Vince Sgro, Adjunct Professor Hourly pay is 0.25
```

Example

/overrid/*.java



Class A

```
import java.io.*;
class A{
   public A() {}
   public void f(){
       System.out.println("in A's f");
       return;}
   public void h(int i){
       System.out.println("in A's h, i= " + i);
       return;}
   public void s(double d){
       System.out.println("in A's s, d= " + d);
       return;}
...
```

Class B

```
import java.io.*;
class B extends A{
   public B() {}
   public void f(){
       System.out.println("in B's f");
       return;}
   public void h(int i){
       System.out.println("in B's h, i= " + i);
       return;}
   public void s(int j){
       System.out.println("in B's s, j= " + j);
       return;}
}
```

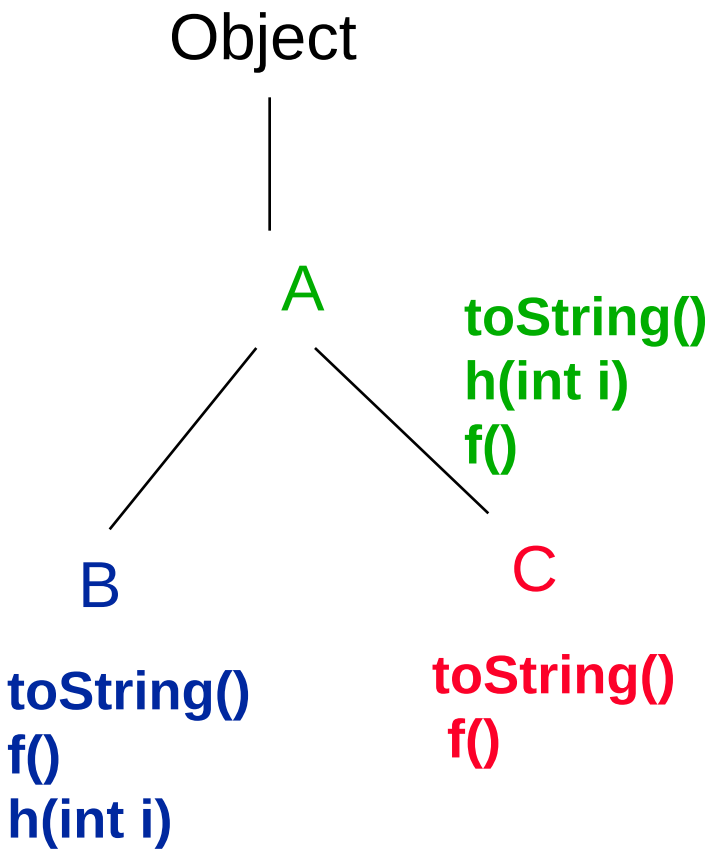
Class C

```
import java.io.*;

class C extends A{
   public C() {}
   public void f()
   {
       System.out.println("in C's f");
       return
   }
}
```

main method

```
//overriding - fcns have
//same signature
A a1 = new B();
A a2 = new C();
B b = new B();
A a = new A();
a.f();//A's f()
a1.f();//B's f()
a2.f();//C's f()
b.h(0);//B's h()
a1.h(2);//B's h()
a2.h(1);//A's h()
a.h(3);//A's h()
```



Example

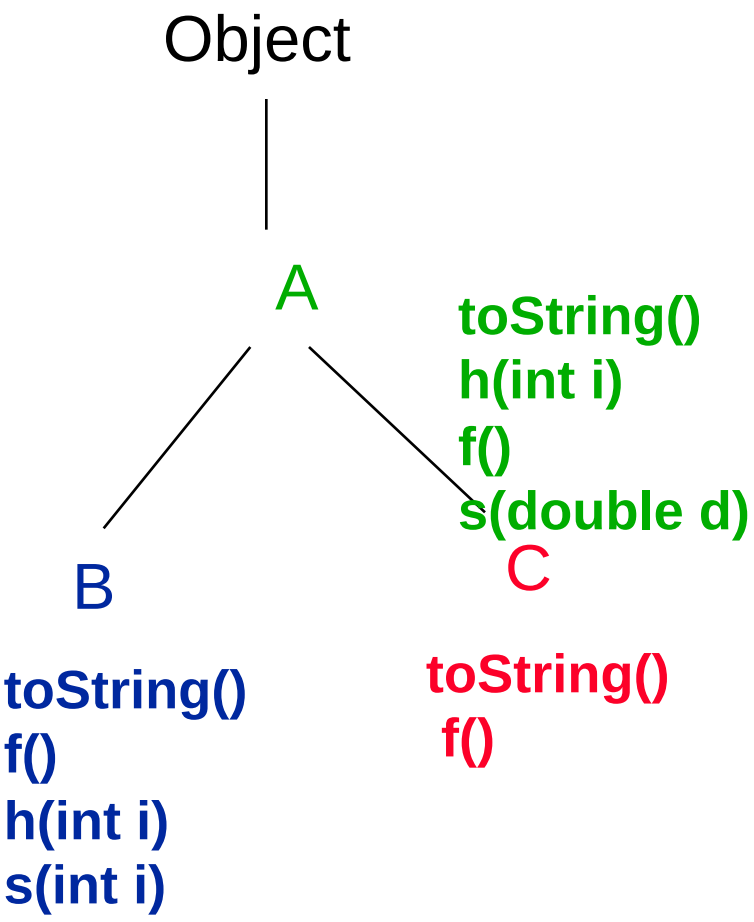
```
A a1 = new B();
A a2 = new C();
B b = new B();
A a = new A();
```

```
//overloading -when signatures
//not same, must look at type
//matching between arg and
//param
a.s(3.);//A's s()

a1.s(3.);//A's s() because
//arg is a double and B's
//s() expects an int

b.s(0);//B's s()
b.s(1.0);//A's s()

//matching rules are not
//always straight-forward
a1.s(0);//A's s()
```



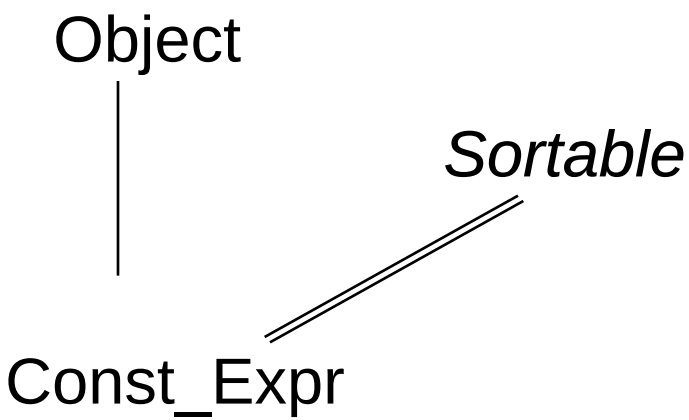
Interfaces

- **Interface specifies functionality without providing implementation**
  - **Inhertance of promised functionality**
- **A class which *implements* an interface must define those methods specified in the interface**
- **Interfaces provide a way to simulate multiple inheritance to some extent in Java**

Interfaces

- **Examples**
  - ***Enumeration* interface promises two functions *hasMoreElements()* and *nextElement()***
  - ***Sortable* promises that pairs of values can be compared using *lessThan(..)***
  - ***Clonable* promises there is a copying method called *clone()* which creates a copy of its receiver object and returns it**

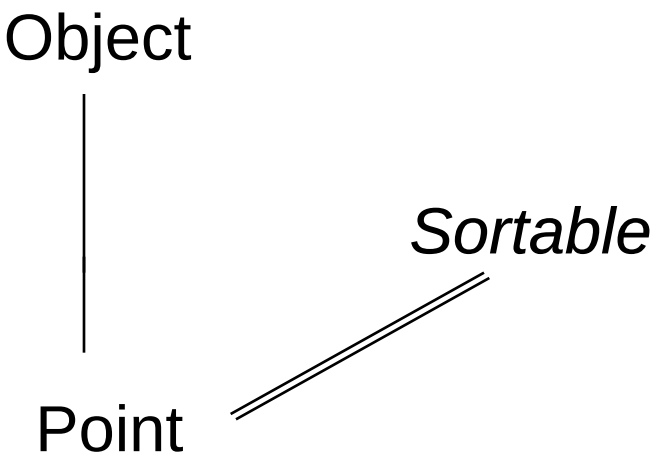
Interface Examples



can implement lessThan(Const_Expr other) in Const_Expr as a comparison of Const_Expr values:

```
 if (this.eval() < other.eval()) return this;
else return other;
```

Interface Examples



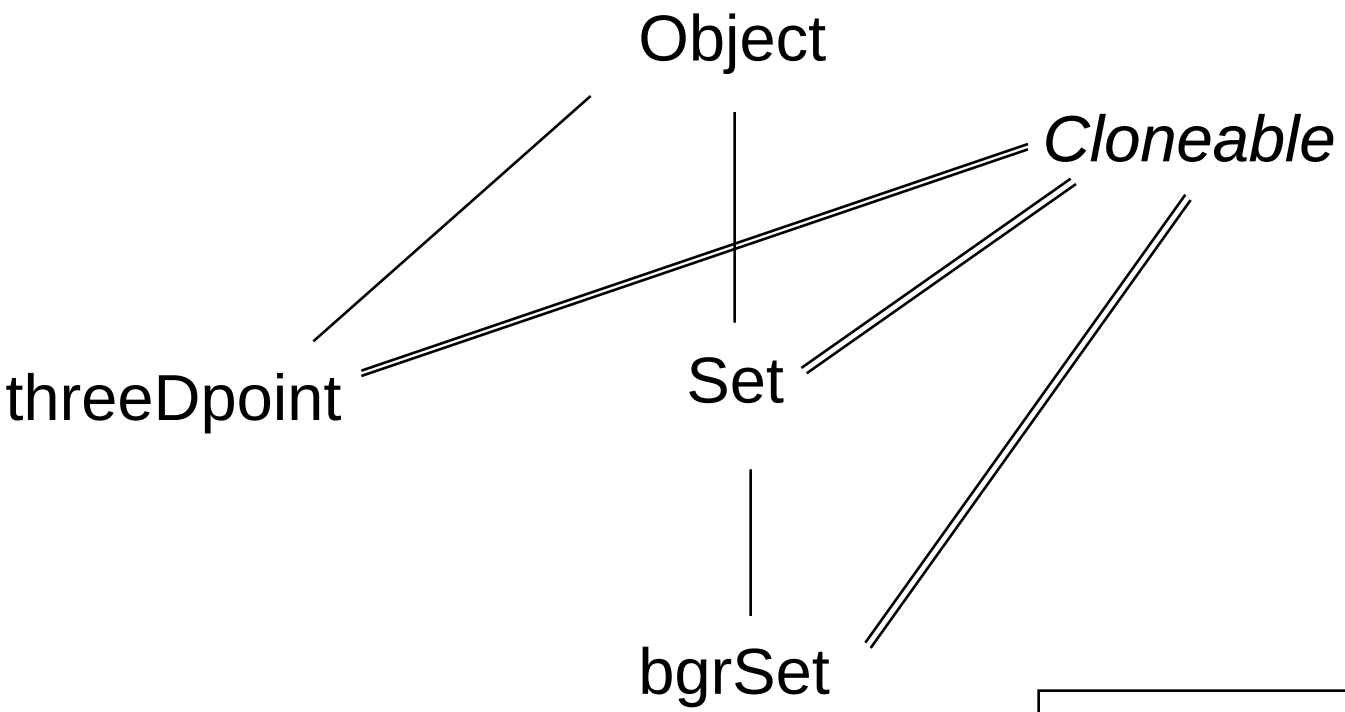
can implement lessThan(Point other) in Point as a distance to the origin comparison:

```
double xthis = this.getX(), ythis =this.getY(),
       xother = other.getX(), yother = other.getY();
if (xthis*xthis+ythis*ythis < xother*xother+yother*yother)
       return this
else return other;
```

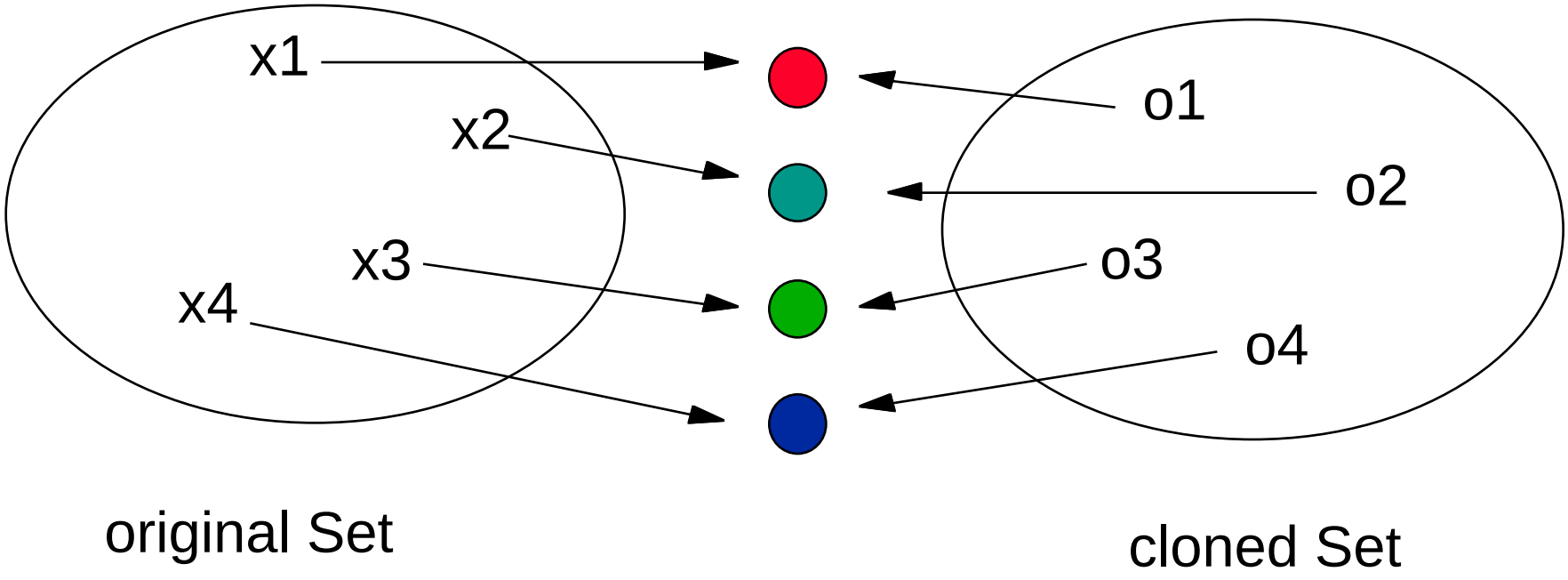
Interface Cloneable

- ***Object cloning*** **- creating a copy of an object**
  - **Can do a *shallow* or *deep* copy**
  - **clone() creates a new object with same values as old one**
- **Manner of performing cloning determines whether or not mutating the value of the old object will affect the value of the clone**

Cloneable Example

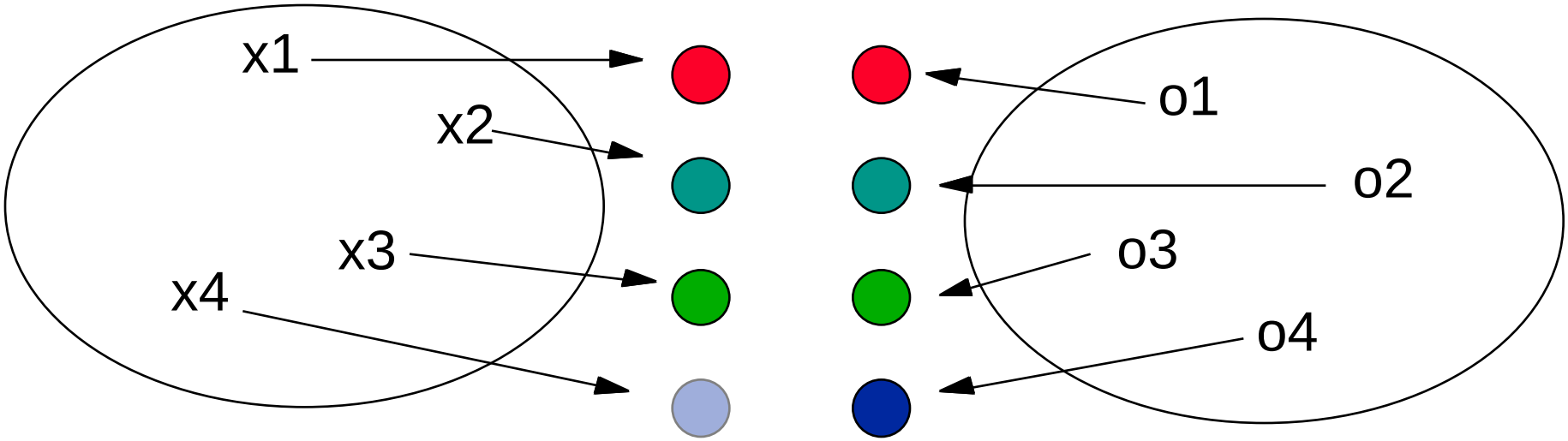


Shallow Cloning - in Set class



Problem: if you mutate value of an object in the original Set you will observe the mutation in the cloned Set.

Deep Cloning in bgrSet class



original bgrSet

cloned bgrSet

Mutation of original bgrSet does not affect cloned bgrSet.

Shallow Cloning - in Set

```
protected Object clone()
  {
    Set cl = new Set();
    Object o;
    LinkedListEnumeration lle = new
                  LinkedListEnumeration(list);
    while(lle.hasMoreElements())
    { o = lle.nextElement();
      cl.addTo(o);//copies references to objects
                  //not objects themselves
    }
    return(cl);
  }
```

Deep Cloning in bgrSet class

```
public class bgrSet extends Set implements Cloneable
{  bgrSet()
   { super();}
//clone does a deep copy in bgrSet unlike the
// shallow copy in Set; elements in bgrSet
//must be Cloneable too;assume never extend bgrSet
   protected Object clone() {
       bgrSet b = new bgrSet();
       SetEnumeration se = new SetEnumeration(this);
       while (se.hasMoreElements())
       {     threeDpoint oo = (threeDpoint)
                              se.nextElement();
             Object ooclone = oo.clone();
             //copies (clones) objects themselves
             b.addTo(ooclone);
       }
       return b;
   }
```

threeDpoint class

```
class threeDpoint extends Object implements Cloneable
{  private double x;
   private double y;
   private double z;
//clone() copies all fields of threeDpoint object
//creating an exact copy of it
//methods defined:
   protected Object clone();
   public void getX(),getY(), getZ()
   public void setX(), setY(), setZ()
   public String toString()
...
}
```

main from bgrSet class

```
   public static void main(String []args)
   {Set b = new Set();
   bgrSet bgr = new bgrSet();
//create two different sets
   threeDpoint origin = new threeDpoint(0.,0.,0.);
   threeDpoint one = new threeDpoint(1.,1.,1.);
   b.addTo(one);
   b.addTo(new threeDpoint(2.,2.,2.));
   bgr.addTo(origin);
   bgr.addTo(new threeDpoint(0.,1.,2.));
   bgr.addTo(new threeDpoint(0.,3.,4.));
   System.out.println ("original sets were b=\n" +
                       b.toString() + "\n bgr = " +
                       bgr.toString());
```

Output 1

```
original sets were b=
        threeDpoint [1.0,1.0,1.0]
        threeDpoint [2.0,2.0,2.0]
 bgr =  threeDpoint [0.0,0.0,0.0]
        threeDpoint [0.0,1.0,2.0]
        threeDpoint [0.0,3.0,4.0]
```

more of main method

```
  Set bclone = (Set) b.clone();
  bgrSet bgrclone = (bgrSet) bgr.clone();
//mutate bgrSets b and bgr
  origin.setZ(-1.0);
  one.setZ(-2.0);

  System.out.println (
  "shallow clone in Set shows the mutation of one”
           + “\n" + bclone.toString());

  System.out.println(
  "deep clone in bgrSet does not show mutation of” +
     “origin \n" + bgrclone.toString());
  }
```

Output 2

```
shallow clone in Set shows the mutation of one
        threeDpoint [1.0,1.0,-2.0]
        threeDpoint [2.0,2.0,2.0]
deep clone in bgrSet does not show mutation of
  origin
        threeDpoint [0.0,0.0,0.0]
        threeDpoint [0.0,1.0,2.0]
        threeDpoint [0.0,3.0,4.0]
```