Prototype Pattern

- **Intent**
  - Specify the kind of objects to create using a prototypical instance, and create a new instance by copying this prototype
- **Motivation**
  - Copy & Paste function in the editor
  - Tool palette with prototype objects which can be copied

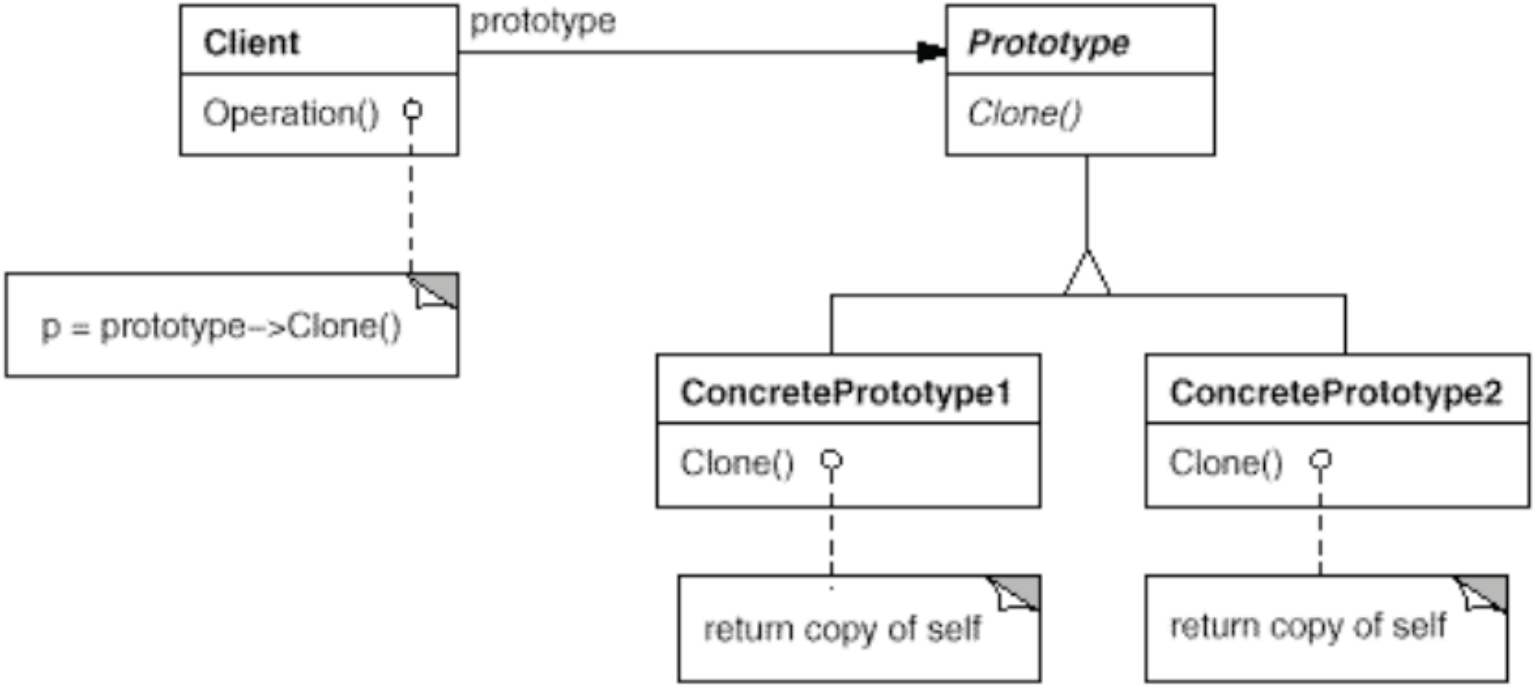
Prototype Pattern

- **Problem: Copy & Paste in JDraw**

```
Figure copyFigure(Figure f) {
   Rectangle r = f.getBounds();
   if(f instanceof RectangleFigure) {
      return new RectangleFigure(r.x, r.y, r.w, r.h);
   } else if(f instance of OvalFigure) {
      return new Ovalfigure(r.x, r.y, r.w, r.h);
   } else if(f instance of LineFigure) {
      Line line = (LineFigure) f;
      return new LineFigure(line.getFrom(), line.getTo());
   } else {
      ...
   }
}
```

  - Violates open-closed principle
  - Solution: delegate copying to instance itself

Prototype Pattern: Structure



Prototype Pattern

- **Implementation of the clone method**

```
public class Point {
   private int x, y;

   public Point(int x, int y) {
      this.x = x; this.y = y;
   }

   public Object clone() {
      ...
   }
}
```

Object.clone

- **Object.clone**

```
class Object {
   protected Object clone() throws CloneNotSupportedException {
      ...
   }
}
```

  - General intent of this method (not absolute requirements, SHOULD)
    - x.clone() != x — *new instance*
    - x.clone().getClass() == x.getClass() — *same dynamic type*
    - x.clone().equals(x) — *equal*
  - Visibility: `protected`
    - Can only be invoked if clone is overridden in a subclass with a method visible by the client
    - Cannot be invoked on objects of static type Object

Object.clone

- **Implementation**
  1. Implementation checks whether the class implements the Cloneable interface
     - Cloneable is a marker interface
     - Cloneable is used to declare that a class supports cloning
     - If Object.clone is called on a class without Cloneable

       `=> CloneNotSupportedException`

  2. New instance is created (no constructor invocation!!!)
  3. All attributes are copied (memory-copy)

Object.clone: Harmony Implementation

- **Object**

```
protected Object clone() throws CloneNotSupportedException {
   if (!(this instanceof Cloneable)) {
      throw new CloneNotSupportedException(
                    "Doesn't implement Cloneable interface!");
   }
   return VMMemoryManager.clone(this);
}
```

- **VMMemoryManager**

```
class VMMemoryManager {
   static native Object clone(Object object);
   ...
}
```

Object.clone: Harmony Implementation

- **C-Code (object_clone, simplified)**

```
jobject object_clone(JNIEnv *jenv, jobject jobj) {
   ManagedObject *result;
   ObjectHandle h = (ObjectHandle) jobj;
   VTable *vt = h->object->vt();
   size = vt->allocated_size;
   result = vt->clss->allocate_instance();

   memcpy(result, h->object, size);
   result->set_obj_info(0);
   ObjectHandle new_handle = oh_allocate_local_handle();
   new_handle->object = result;
   return (jobject) new_handle;
}
```

Object.clone: Example Point

- **Implementation of the clone method**

```
public class Point implements Cloneable {
   private int x, y;
   public Point(int x, int y) {
      this.x = x; this.y = y;
   }

   @Override
   public Object clone() {
      try {
         return super.clone();
      } catch (CloneNotSupportedException e) {
         throw new InternalError(e.getMessage());
      }
   }
}
```

Object.clone: Example Point

- **Implementation of the clone method**

```
class ColorPoint extends Point {
   private Color c;
   public ColorPoint(int x, int y, Color c) {
      super(x, y); this.c = c;
   }




}
```

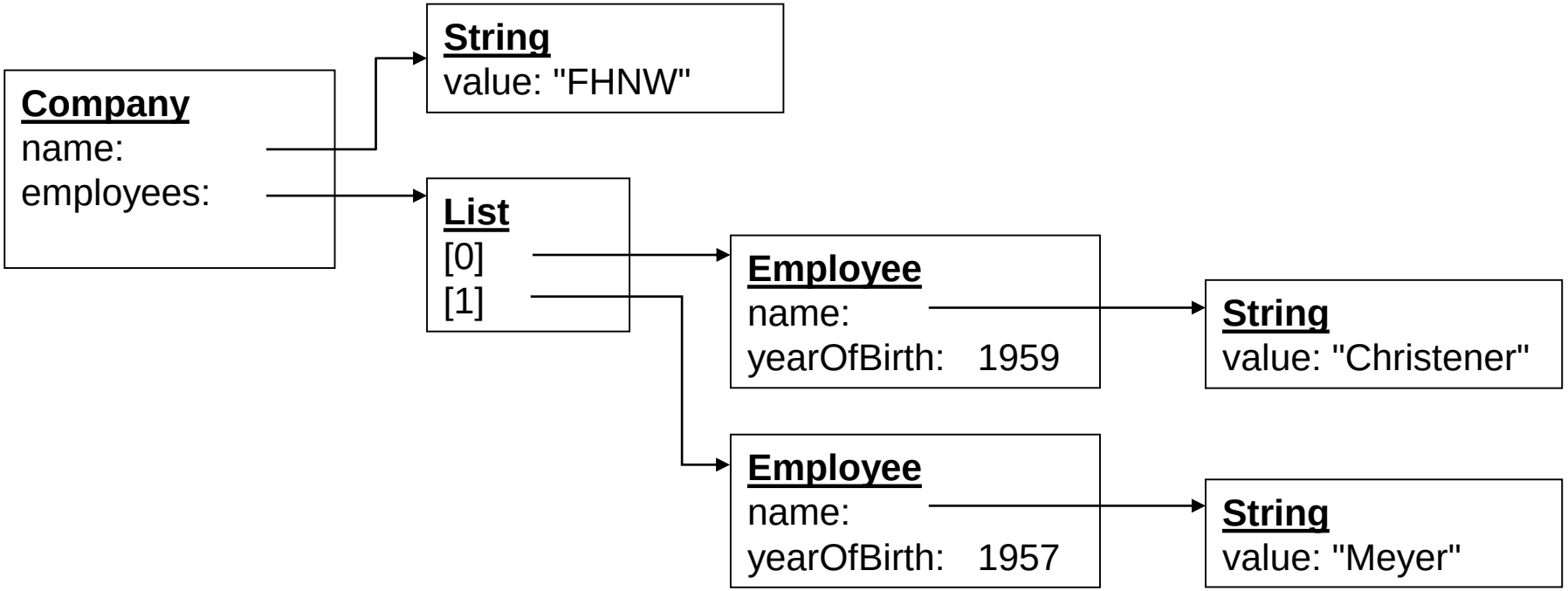
Deep vs Shallow Copy

- **Company with Employees**

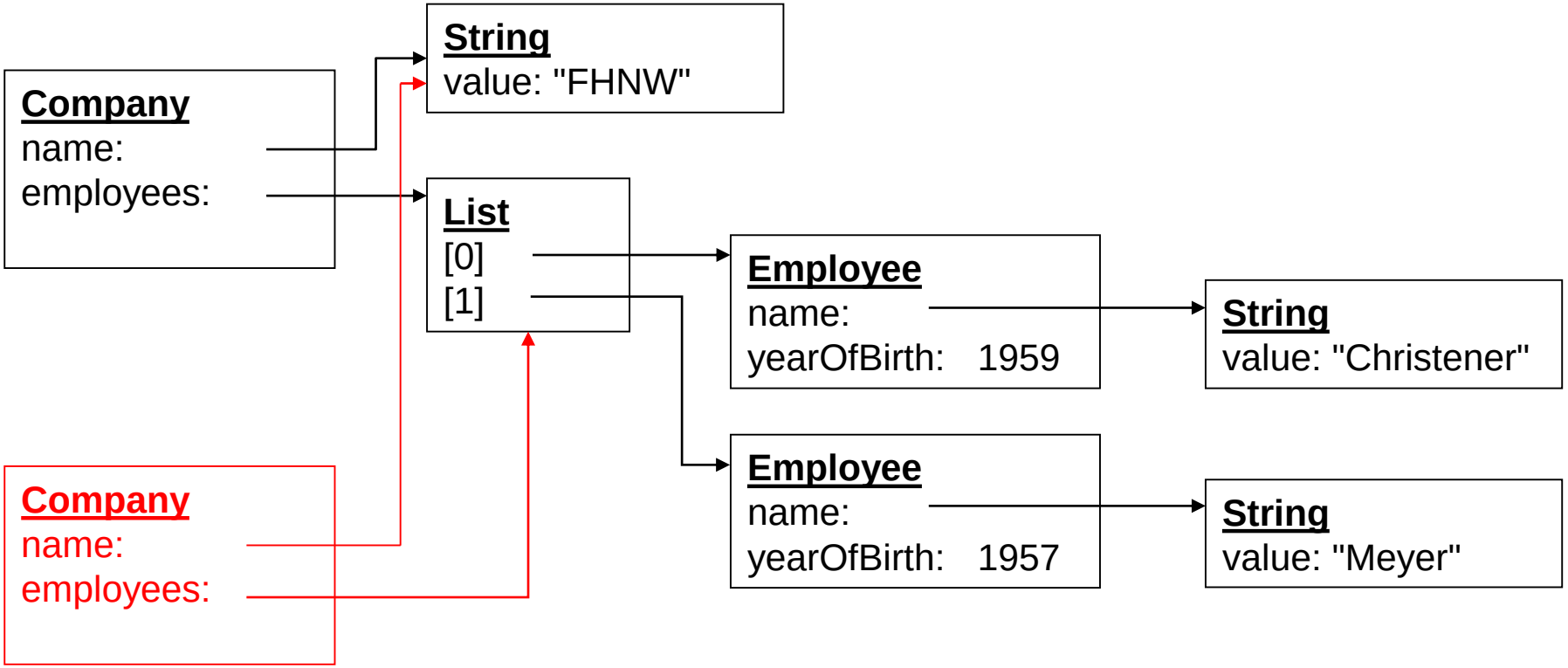
```
pubic class Company {
   private String name;
   private List<Employee> employees = new ArrayList<>();
}

public class Employee {
   private String name;
   private int yearOfBirth;
}
```

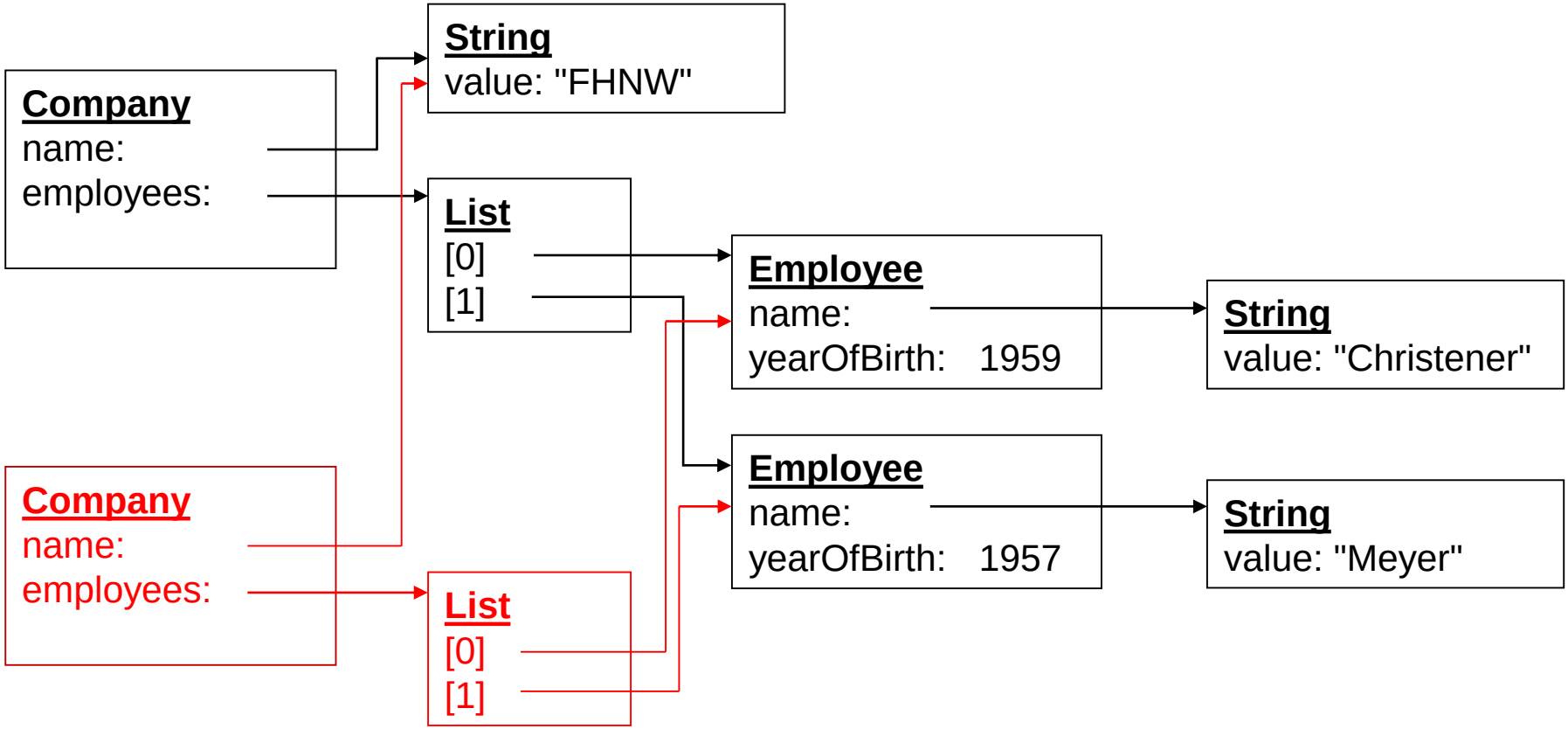
Deep vs Shallow Copy



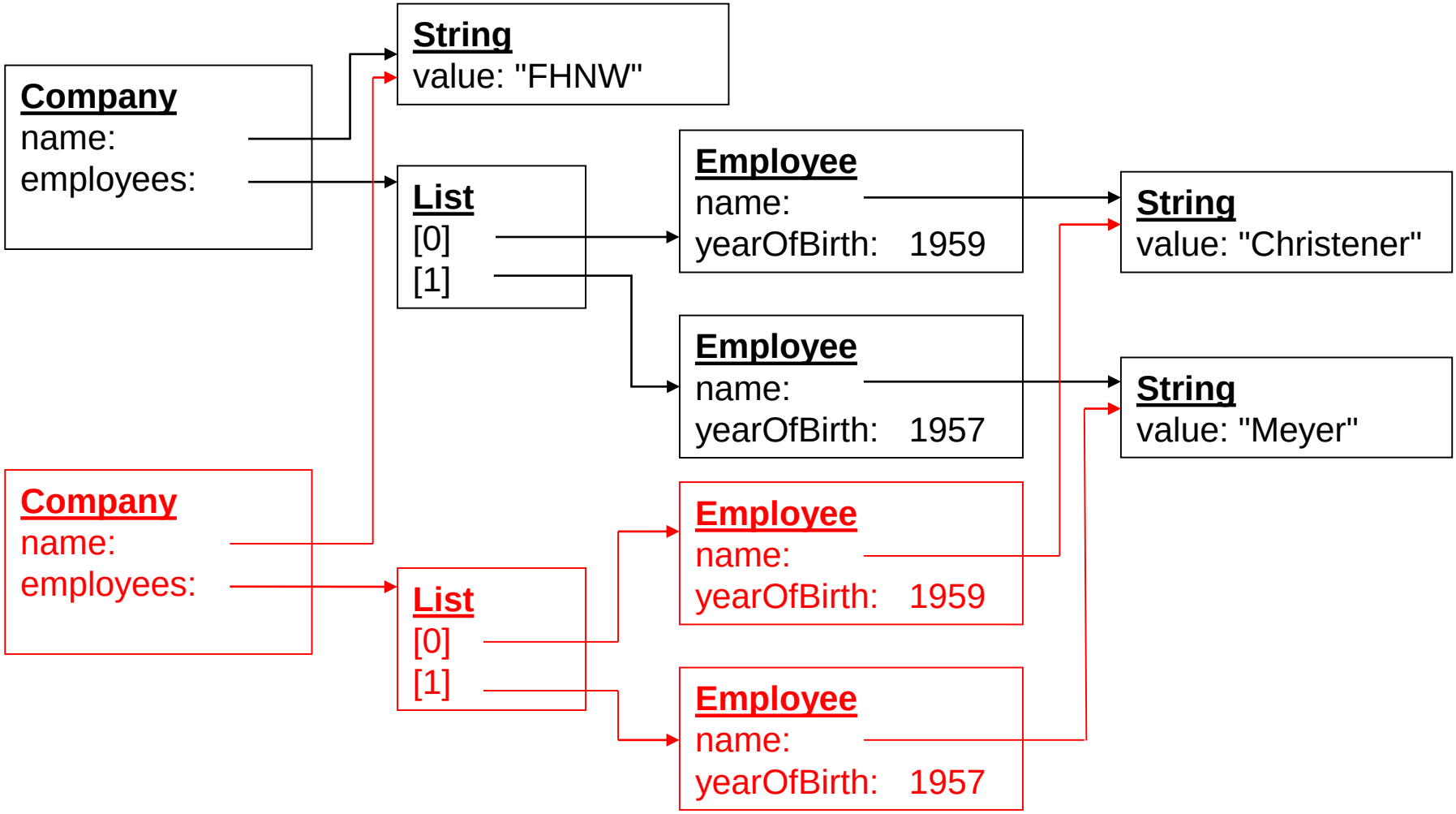
Deep vs Shallow Copy



Deep vs Shallow Copy



Deep vs Shallow Copy



Remarks

- **References to immutable objects**
  - Immutable objects need not be cloned, as they cannot be changed (and typically cannot be cloned)
- **Final fields**
  - Final fields cannot be changed in the clone method
- **Signature: return type**
  - Java allows covariant typing, e.g. type of result may be strengthened
- **Alias references**
  - Alias references (and cycles) have to be handled manually when implementing a deep-copy

Clone With Copy Constructor

- **Every cloneable class contains a copy constructor**

```
class Company {
   public Company(Company c) {
      if(c==null) throw new IllegalArgumentException();
      // initialize this with attributes of c
   }
   ...
}
```

- **Method clone invokes this copy constructor**

```
class Company {
   public Object clone() {
      return new Company(this);
   }
   ...
}
```

Clone With Copy Constructor

- **Subclass invokes copy constructor of base class**

```
class SmallBusinessCompany extends Company {
   public SmallBusinessCompany(SmallBusinessCompany c) {
      super(c);
      // initialize this with attributes of c
   }
   ...
}
```

- **Deep-Copy**
  - For a deep copy the attributes need to be copied in the constructor