Objects and Classes

- object
  - state
  - operations
  - identity
  - instance of a class
- class
  - collection of related objects
    - same operations
    - same set of possible states

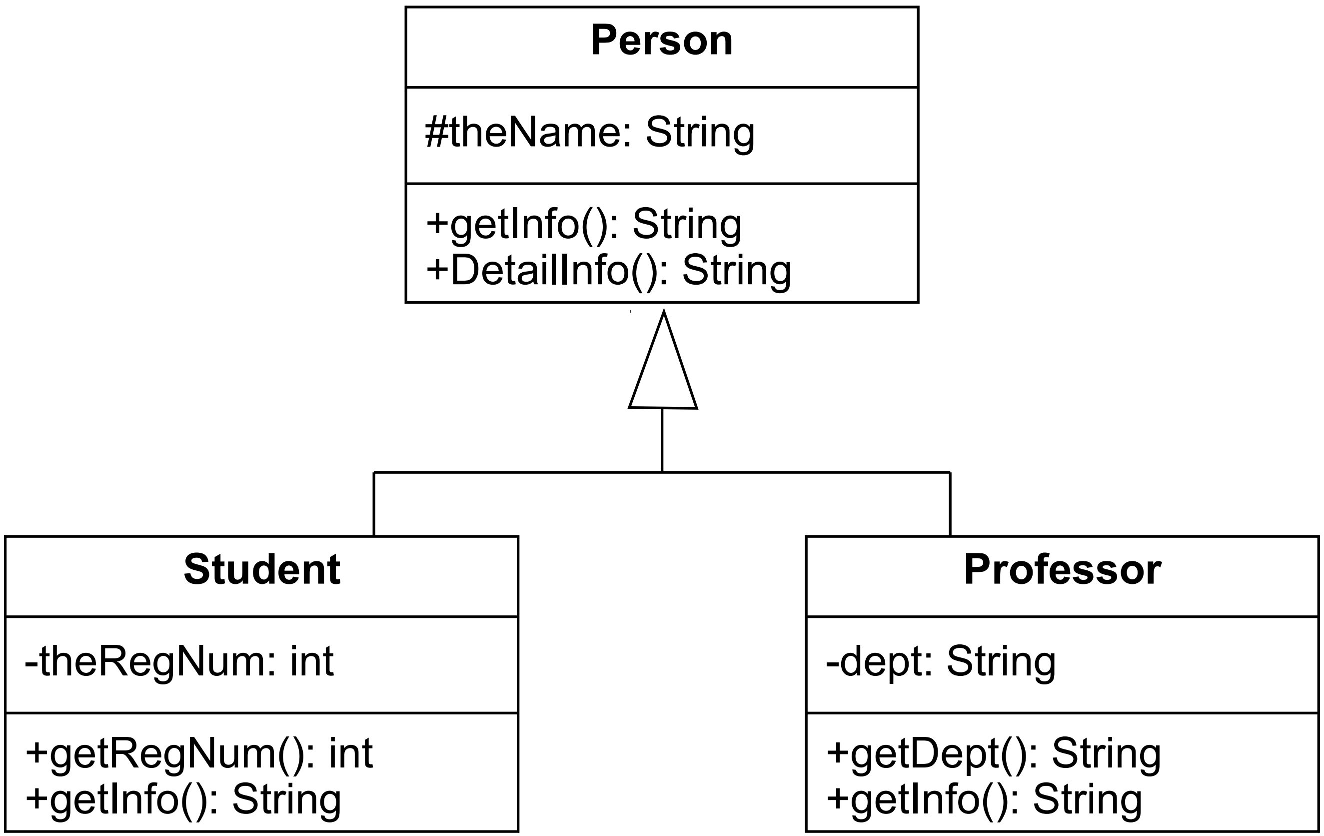
Classification of Languages

- object-based
  - data abstraction, encapsulation, information hiding
  - not what one typically means by “object-oriented”
  - e.g. VB (no inheritance/subtyping)
- class-based
  - module = class
  - class = type
  - class defines properties
  - object = instance of class
- prototype-based
  - alternate object-oriented language with objects, but not defined class-types (behavioural inheritance is via cloning existing objects, etc.)
  - mostly a few scripting languages (javascript, lua, etc.)
- object-oriented
  - inheritance
- purely object-oriented vs. hybrids
  - Smalltalk, Eiffel, Java vs C++, Delphi, Ada 95

Inheritance

- a tool for software re-use
- “is a” relationship
  - every object of subclass is conceptually also an object of superclass
- superclass (parent, base class)
- subclass (child, derived class)
  - extension, restriction
- generalization
- specialization
- Example: person, student, professor
  - inheriting attributes and operations
  - overriding operations
  - new attributes and operations
  - private vs public vs protected

Inheritance



Superclass (Java)

```
public class Person {
  protected String theName;

  public Person(String name) {
    theName = name;
  } // constructor

  public String getInfo() {
    return theName;
  } // getInfo

  public String detailInfo() {
    return “Details are: “ + this.getInfo();
  } // detailInfo
} // Person
    …
Person girl = new Person (“Sue”);
String stra = girl.getInfo();
String strb = girl.detailInfo();
```

Subclass (Java)

```
public class Student extends Person {
  private int theRegNum;

  public Student(String name, int reg) {
    super(name);
    theRegNum = reg;
  } // constructor

  public int getRegNum() {
    return theRegNum;
  } // getRegNum

  public String getInfo() {
    return super.getName() + “,” + theRegNum;
  } // getInfo
} // Student
    …
Student woman = new Student (“Mary”, 2000153);
String stra = woman.getInfo();
String strb = woman.detailInfo();
```

Inheritance

- Java
  - syntax
    - e.g. `public class Student extends Person {…}`
  - all classes are subclasses of `Object`
- C++
  - syntax
    - e.g. `class Student: public Person {…}`
  - no common superclass
- Ada
  - tagged records
  - not class-based

Superclass (Ada)

```
package Persons is
    type Person is tagged private;

   procedure personInit(p : in out Person;
                         name : in String);
   function getInfo(p : in Person) return String;
   function detailInfo(p : in Person) return String;
private
   type Person is tagged record
      theName : String;
   end record;
end Persons;

package body Persons is
   …
end Persons;
```

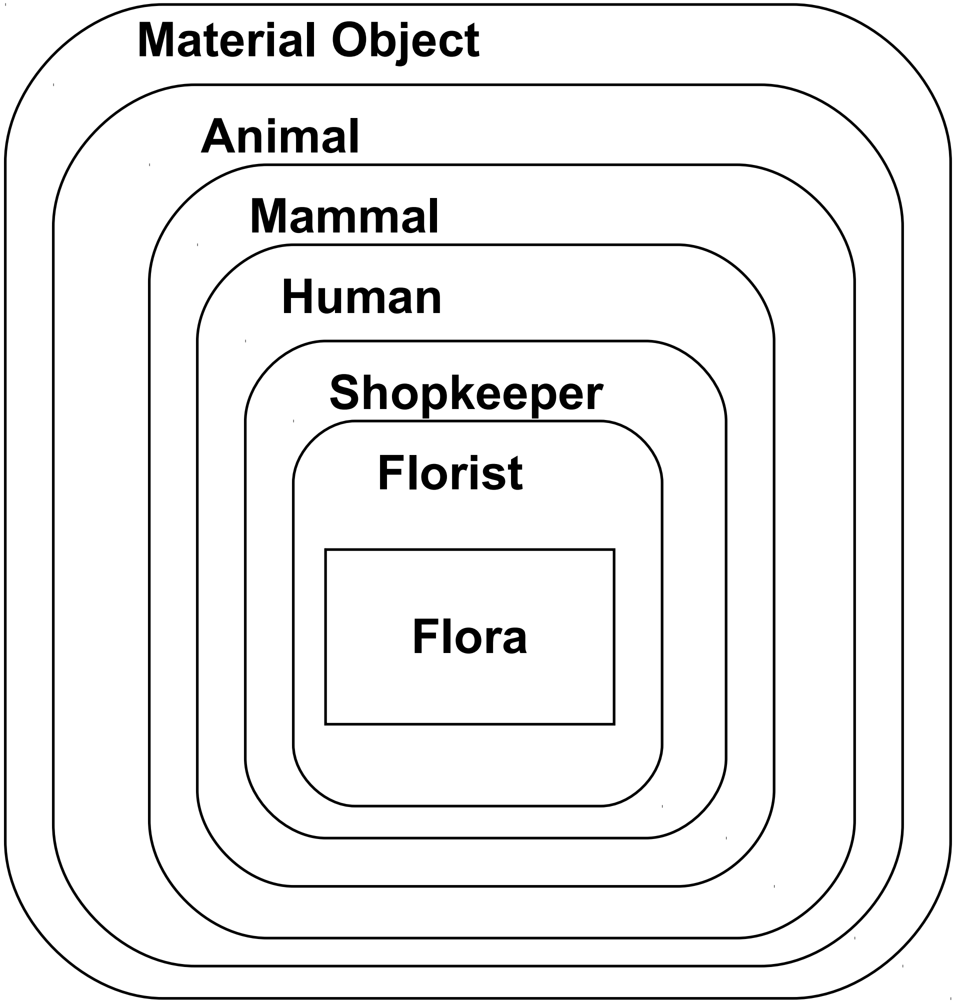
Subclass (Ada)

```
with Persons; use Persons;
package Students is
   type Student is new Person with private;
   procedure studentInit(p : in out Student;
                         name : in String;
                         reg : in Integer);
  function getRegNum(p : in Student) return Integer);
  function getIno(p : in Student) return String;
private
   type Student is new Person with record
      theRegNum : Integer;
   end record;
end Students;
```

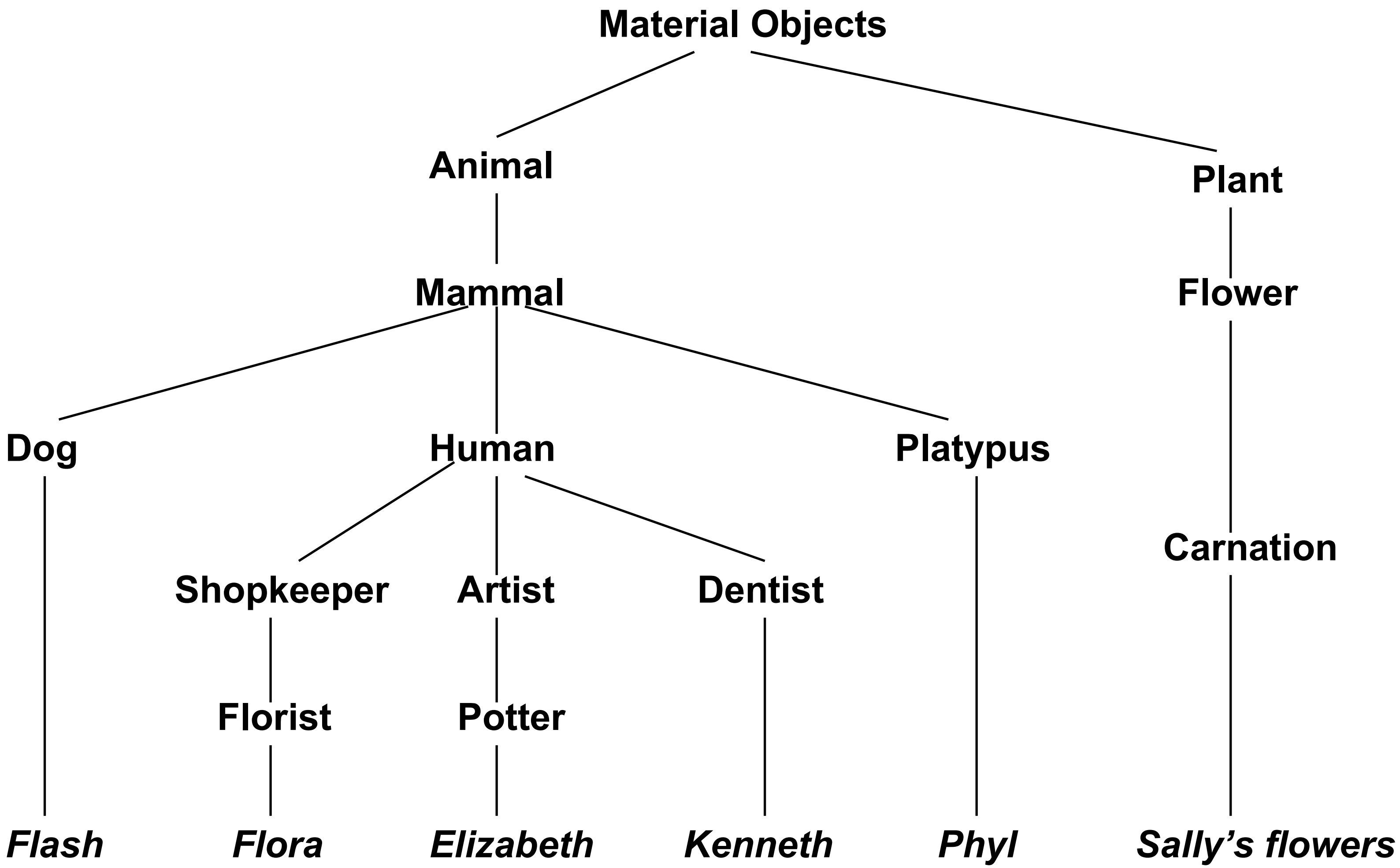
Class Hierarchies (B1.1.5-1.1.6)

- example: material objects
- inheritance is transitive
- abstract class
  - no direct instances
  - vs. concrete class
  - vs. `final`
  - abstract operations
- overriding
  - to encode exceptions to the rule
- method binding
  - compile-time error
  - run-time
    - polymorphism

Categories of Material Objects (B1.1.5)



Class Hierarchy - Material Objects (B1.1.5)



Substitutability (B8.3)

- subclass:
  - class constructed using inheritance
- subtype:
  - all data fields of parent
  - all operations of parent (may be overridden)
  - indistinguishable if substituted for parent in a similar situation
    - subtype can be assigned as a variable of parent's type
    - subtype can be used as if it were of parent's type
- substitution rule: object of subclass must be usable in place of object from superclass

Forms of Inheritance (B8.4)

- inheritance for specialization
  - child satisfies all specifications of parent, but specializes
  - e.g. Person, Student, Professor
- inheritance for specification
  - parent is an abstract class
  - Java `interface`: behaviour, not structure
- inheritance for construction
  - subclass and parent are conceptually different
  - parent has functionality required by subclass

- inheritance for extension
  - no modifications to parent's behaviour
  - new behaviour is added
- inheritance for limitation
  - behaviour of subclass is more restrictive than behaviour of parent
- inheritance for combination
  - multiple inheritance
  - approximations in Java

Specification: Java Interface (B8.4.2)

```
interface ActionListener {
   public void actionPerformed (ActionEvent e);
}


class CannonWorld extends Frame {
   …
     // a fire button listener implements the action
     // listener interface
   private class FireButtonListener implements ActionListener {
     public void actionPerformed (ActionEvent e) {
     … // action to perform in response to button press
     }
   }
}
```

Specification: Java Abstract Class (B8.4.2)

```
public abstract class Number {
   public abstract int intValue();
   public abstract long longValue();
   public abstract float floatValue();
   public abstract double doubleValue();
   public byte byteValue()
        { return (byte) intValue(); }
   public short shortValue()
        { return (short) intValue(); }
}
```

Construction (B8.4.3)

```
public class Stack<E> extends LinkedList<E> {

    public void push(E item) {
        addFirst(item);
    }

    public boolean empty() {
        return isEmpty();
    }

    public E pop() {
        return removeFirst();
    }

    public E peek() {
        return getFirst();
    }
}
```

Extension (B8.4.4)

```
public class Properties extends Hashtable<Object,Object> {
   …
   public void load(InputStream in) throws IOException {…}

   public void save(OutputStream out, String header) {…}

  public String getProperty(String key) {…}

  public Enumeration<?> propertyNames() {…}

  public void list(PrintStream out) {…}

}
```

Limitation (B8.4.5)

```
public class Set<E> extends LinkedList<E> {

    // methods add, remove, contains, isEmpty and
   // size are all inherited from LinkedList

    public int indexOf(Object obj) {
      System.out.println("Do not use Set.indexOf");
      return 0;
   }

   public E get(int index) {
      System.out.println("Do not use Set.get");
      return null;
   }

 }
```

Benefits of Inheritance (B8.7)

- software reusability
- increased reliability
- code sharing
- consistency of interface
- software components
- rapid prototyping
- polymorphism
- information hiding

Costs of Inheritance (B8.8)

- execution speed
  - general-purpose tools generally slower than specific tools
- program size
  - use of library increases size
- message-passing overhead
  - vs. invoking procedures
- program complexity
  - overuse of inheritance

Aggregation vs. Inheritance (B10.2)

- aggregation:
  - relationship between objects
  - one object is “part of” another object
- inheritance:
  - relationship between classes
  - properties of a single object
- example: `Stack` and `Vector`

LinkedList class

```
public class LinkedList<E> {

   // see if the list is empty
   public boolean isEmpty() {…}

   // return size of the list
   public int size() {…}

   // add element to the head of the list
   public void addFirst(E value) {…}

   // return the first element in the list
   public E getFirst() {…}

   // remove and return the first element
  public E removeFirst() {…}
   …
}
```

Stack Using Composition (B10.2.1)

```
public class Stack<E> {

  private LinkedList<E> theData;

  public Stack()
       { theData = new LinkedList<E>(); }

  public boolean empty()
       { return theData.isEmpty(); }

  public void push(Object item)
       { theData.addFirst(item); }

  public E peek()
       { return theData.getFirst(); }

  public E pop() {
       { return theData.removeFirst(); }
  }
}
```

Stack Using Inheritance (B10.2.2)

```
class Stack<E> extends LinkedList<E> {

   public void push (Object item)
        { addFirst(item); }

   public E peek ()
        { return getFirst(); }

   public E pop () {
        return removeFirst();
}
```

Comparison (B10.3)

- inheritance implies substitutability while composition does not
- composition is easier to understand
- inheritance extends operations (“yo-yo” problem)
- inheritance requires less writing
- inheritance cannot restrict operations
- inheritance is is-a, composition is uses-a

Implications of Inheritance (B11)

- polymorphic variables
- allocation on the heap
- assignment and parameter passing by reference semantics
- equality testing
- memory management

Polymorphism (B11.1)

- “many forms”
- polymorphic variables
  - declared as a variable of one type
    - static
  - can maintain a value of that type or any subtype
    - dynamic
  - static binding vs. dynamic binding
  - example: ShapeTest

Shape Classes (B11.1)

```
class Shape {
  protected int x;
  protected int y;

  public Shape (int ix, int iy)
  { x = ix; y = iy; }

  public String describe()
  { return “unknown shape”; }
}

class Square extends Shape {
  protected int side;

  public Square (int ix, int iy, int is)
  { super(ix, iy); side = is; }

  public String describe()
  { return “square with side ” + side; }
}
```

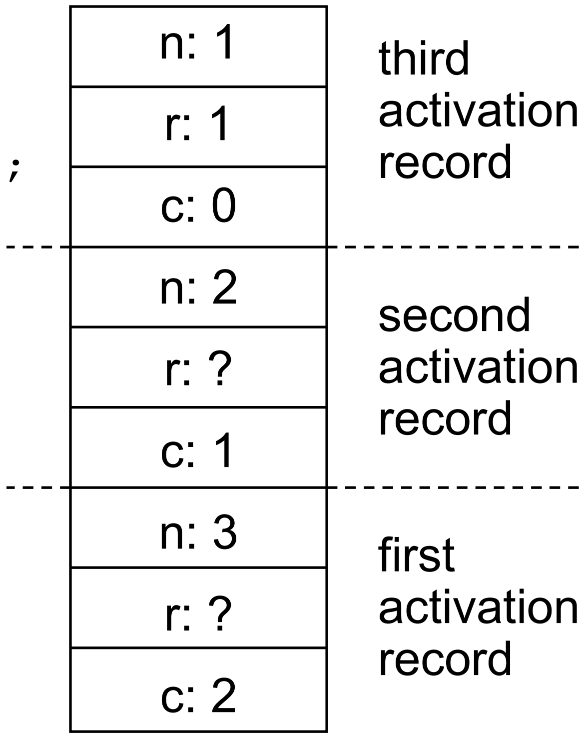
```
class Circle extends Shape {
  protected int radius;

  public Circle (int ix, int iy, int ir;)
  { super(ix, iy); radius = ir; }

  public String describe()
  { return "circle with radius " + radius; }
}

…

class ShapeTest {
  static public void main (String [] args) {
    Shape form = new Circle (10,10,5);
    System.out.println("form is " + form.describe());
    form = new Square (15,20,10);
    System.out.println("form is " + form.describe());
  }
}
```

Memory Layout (B11.2)

- stack-based
  - tied to method entry and exit
  - offsets must be known at compile-time
  - example: factorial
  - memory requirements for polymorphic variable determined at run-time
- heap-based
  - allocated when requested, i.e. at run-time
  - freed when no longer required
  - compiler needs to calculate offsets
    - memory requirements for pointers known at compile time
- C++
  - variables stored on stack
  - extra fields “sliced off”

Stack-Based Allocation (B11.2)

```
class FacTest {
  static public void main (String [] args) {
    int f = factorial(3);
    System.out.println("Factorial of 3 is "+f);
  }

  static public int factorial (int n) {
    int c = n-1;
    int r;
    if(c > 0)
      r = n* factorial(c);
    else
      r = 1;
    return r;
  }
}
```

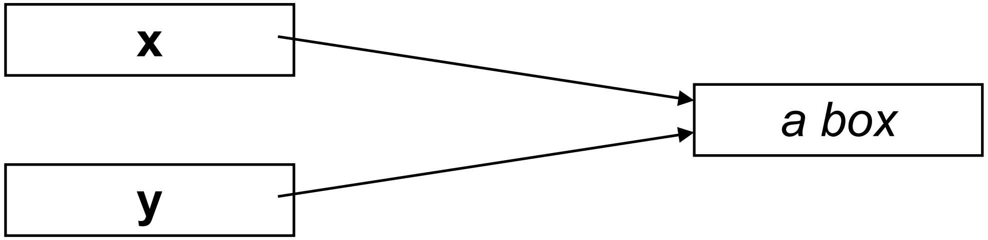


Assignment (B11.3)

- reference semantics:
  - pointer is copied
  - result: 2 references to same object
- clones
  - 2 different objects with same values
- example: Box

Reference Semantics (B11.3)

```
public class Box {
  private int value;

  public Box() { value = 0; }
  public void setValue (int v) { value = v; }
  public int getValue() { return value; }
}
...

    Box x = new Box();
    x.setValue(7);
    Box y = x;
    y.setValue(11);
    System.out.println("contents of x" + x.getValue());
    System.out.println("contents of y" + y.getVallue());
```



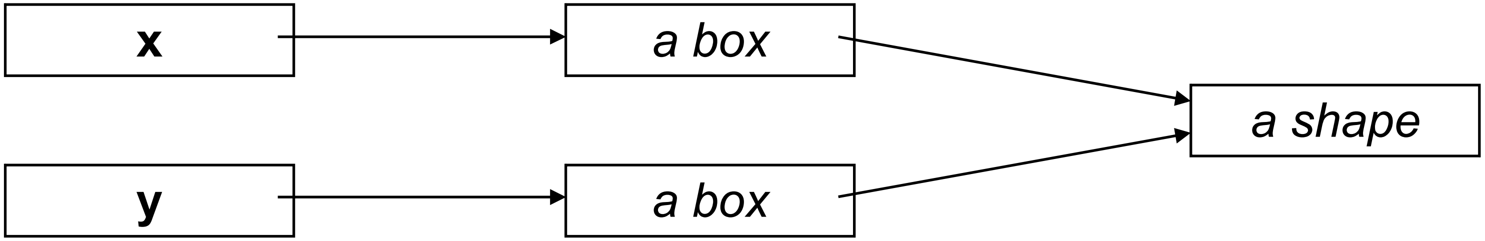
Cloning (B11.3.1)

```
public class Box implements Cloneable {
  private int value;

  public Box () { value = 0; }
  public void setValue (int v) { value v; }
  public int getValue () { return value; }

  public Object clone () {
    Box b = new Box();
    b.setValue (getValue());
    return b;
  }
}
```

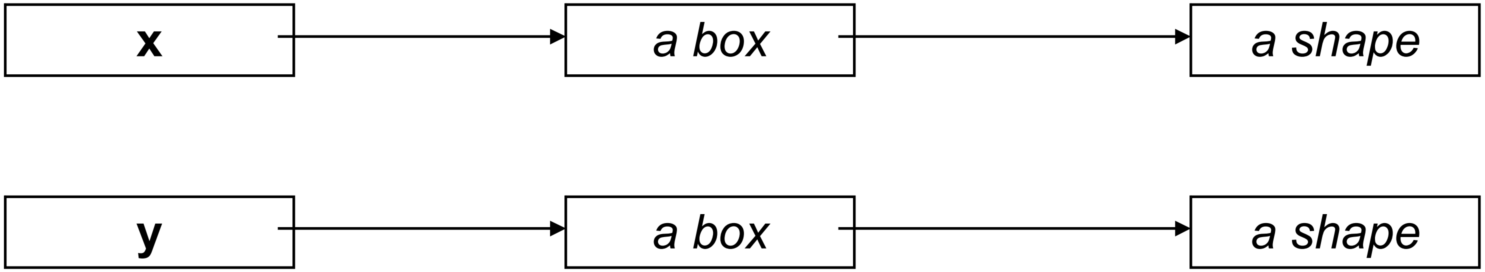
…

```
    Box x = new Box();
    x.setValue (7);
    Box y = (Box) x.clone();
    y.SetValue (11);
```

Shallow Copy vs Deep Copy (B11.3.1)



A shallow copy



A deep copy

Equality Test (B11.4)

- reference semantics (`==`, `!=` in Java)
  - compare pointers
  - identity testing
  - compare to `null`
- object equality
  - `equals` in Java
  - by default is defined as `==`
  - can be overridden to do equality
    - deep equals vs shallow equals
    - care to preserve symmetry and transitivity

Garbage Collection (B11.5)

- stack-based recovered at block (procedure) exit
- heap-based not necessarily automatically recovered
- explicit deallocation (C++) vs garbage collection (Java)