Objects

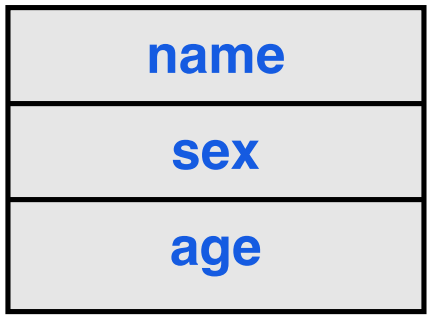
Run time direct instances of classes

Fields

- The attributes within a class are a template for a collection of **fields** in an object

```
class PERSON
feature
    name : STRING
    sex : GENDER
    age : INTEGER
end -- class PERSON
```

a_Person



Objects

- Variables with a class for a type
- Must have a name **declared** and the name must be **attached** to the object
- Using an object requires two steps: **declaration** and **creation**

```
p : PERSON          -- declare the name p
create p            -- create and attach object to p

w : PERSON                  -- declare the name w
create w.make("Me")         -- create via a function
```

Creation Operator

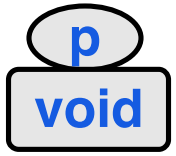
- **create** is akin to **new** in C++ and Java

- The 4 steps
  - » **Create an instance of the type**
    - **Allocate enough memory for the instance**
  - » **Initialize each field to default values**
  - » **Attach the reference to the variable**
  - » **Execute the procedure (if any) to complete initialization**

Reference types

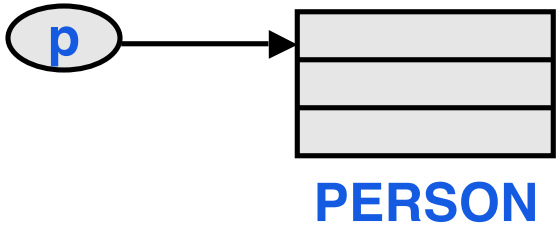
- p is used to **refer to** an instance of type **PERSON**

p : PERSON



- Create and attach object to p – **p is attached**

create p



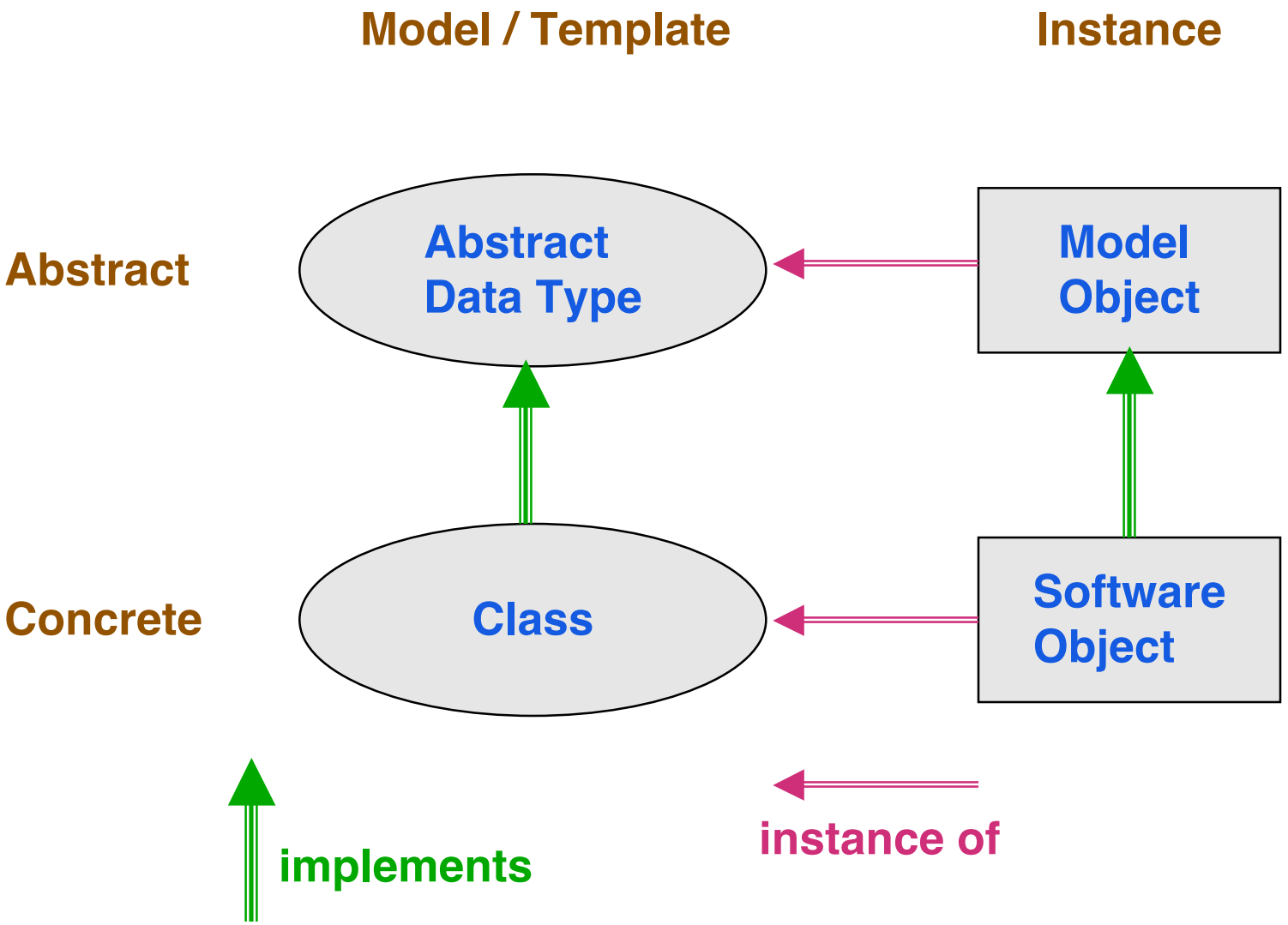
- Think of p as a pointer

For type safety, unlike C/C++, the pointer cannot be de-referenced

Reference Types – 2

- In general, declaring a type means the variable is a reference to an instance of the type
- Primitive types – **INTEGER, REAL, DOUBLE, CHAR** – are not references, they are statically allocated (**expanded**)
  - » **They are still first class objects – no repackaging as in Java**
- **Expansion** means the reference is replaced with the fields of the referenced object
- Any reference can be expanded
  - » **Provided there are no cycles**

Models & Objects



Copying

a := y – copies only the reference

a := y.twin

– shallow copy – one level copy
– new storage space is created
– y must exist

a := y.deep_twin

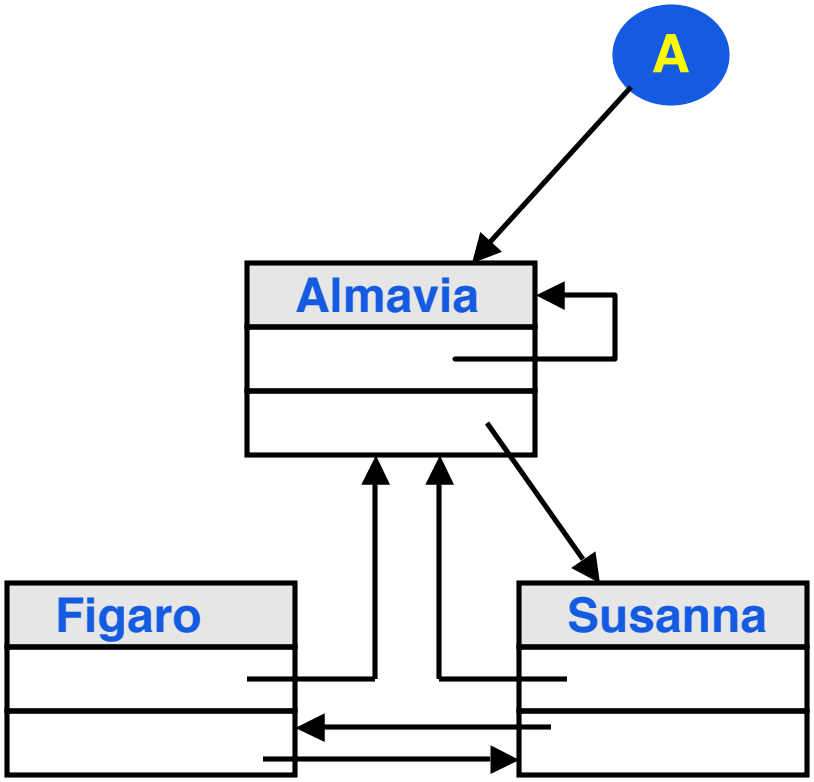
– deep copy – all levels
– new storage space is created
– y must exist

a . copy(y)

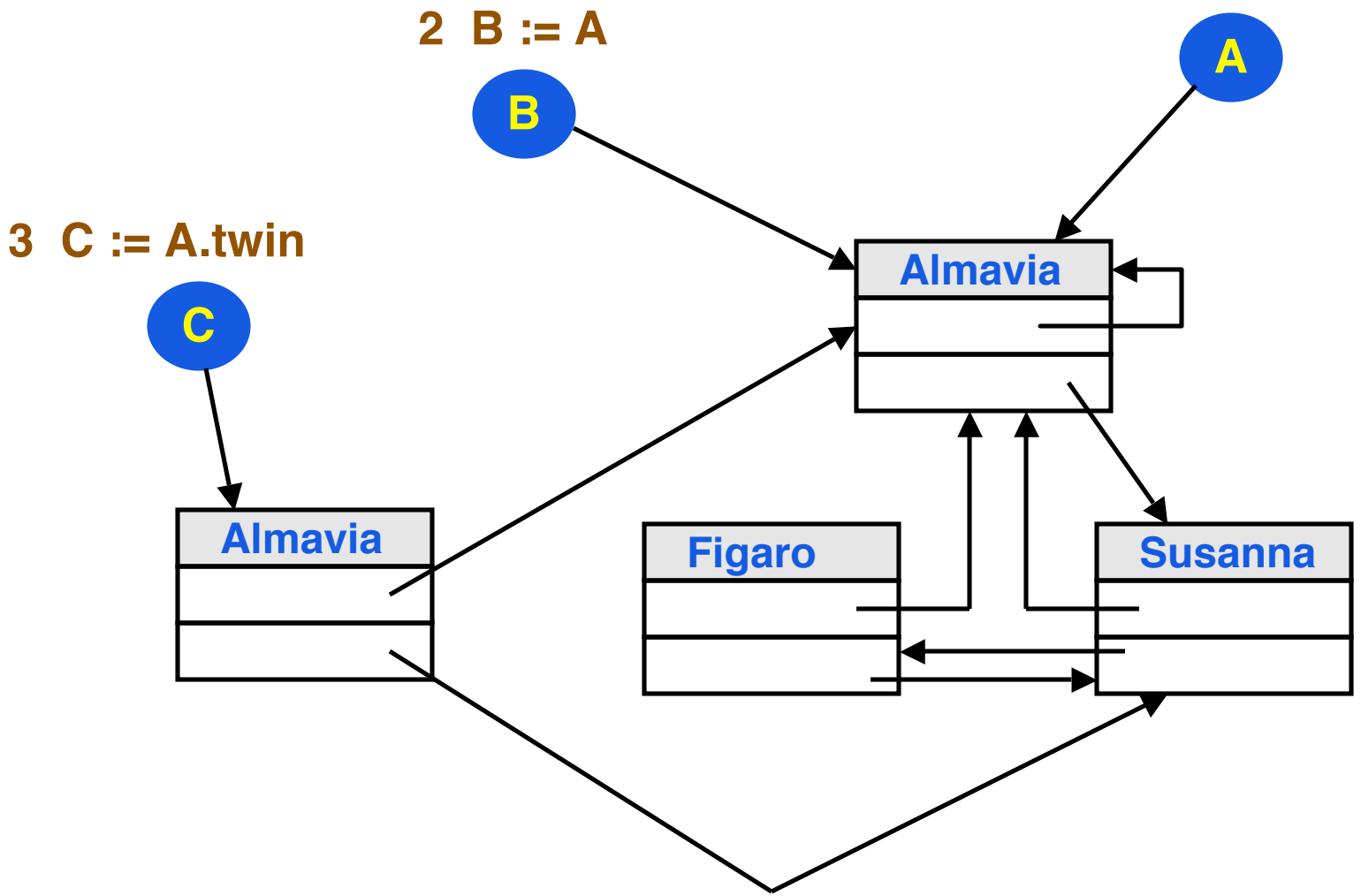
– shallow copy
– a exists, replace fields of a with those in y
– NO new storage

Copying – 2

1 A is created

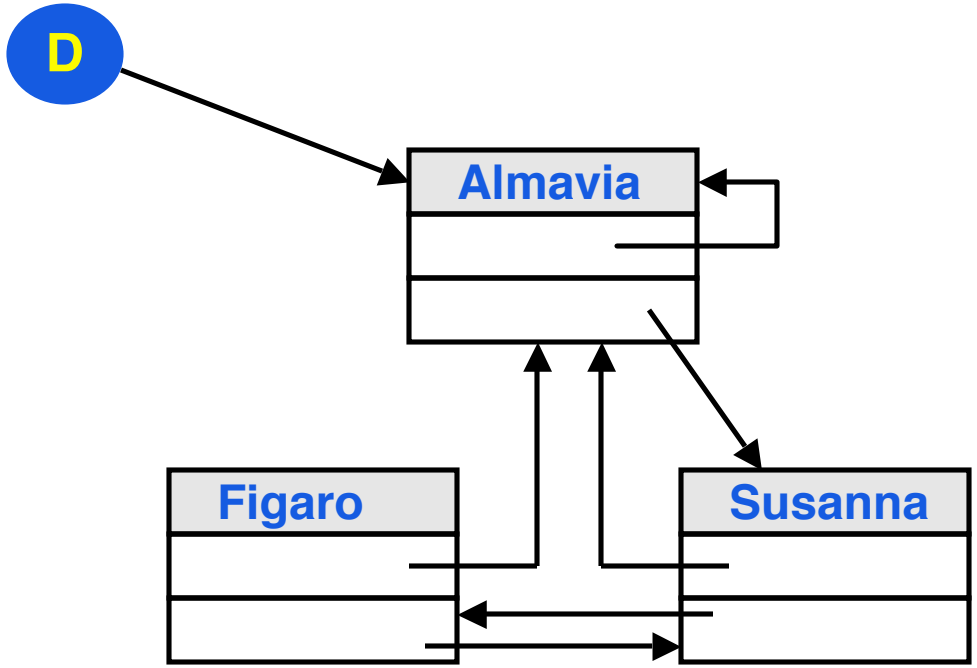


Copying – 3



Copying – 4

4 D := A.deep_twin -- all new memory locations



Equality

=	– compares references
equal (a , b)	– shallow comparison – compares one level – works if a is void
a.is_equal (b)	– compares one level – shallow comparison
deep_equal (a , b)	– compares all levels – deep comparison – works if a is void

Persistence

- Direct dependents
  - » **The direct dependents of an object are the objects attached to its references**
- Dependents
  - » **The dependents of an object are:**
    - > **The object itself**
    - > **Its dependents**
    - > **And – recursively – the dependents of its direct dependents, etc.**

Persistence Closure Principle

Whenever a storage mechanism stores an object, it must store with it the dependents of that object.

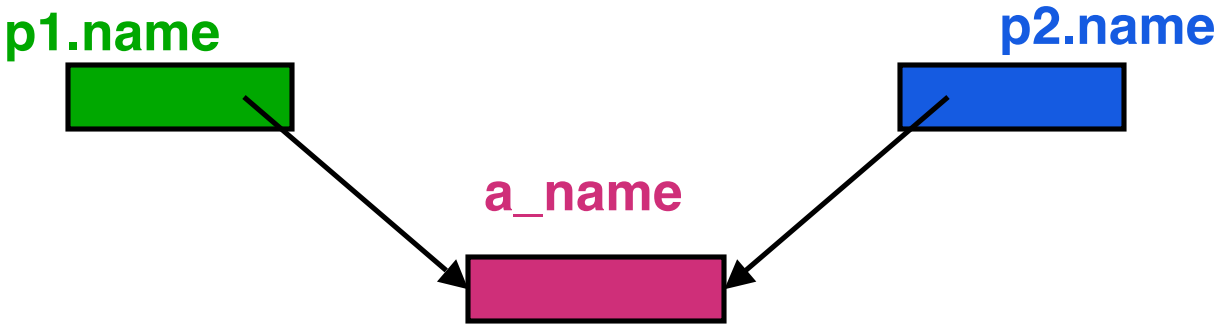
Whenever a retrieval mechanism retrieves a previously stored object, it must also retrieve any dependent of that object not already retrieved.

Composite Objects & Expanded Types

- Consider the following version of class Person

```
class PERSON
feature
    name : NAME
end -- class PERSON
```

- We say that Person **has a** NAME
- Normally NAME is a reference
  - » **Makes it possible for two or more instances of PERSON to share the same NAME**



Composite Objects & Expanded Types – 2

- Sharing references leads to **aliasing** which can lead to surprises
  - **p1.name := “John” ; p2.name := p1.name**
  - **print (p2.name) --> John**
  - **p2.name.put(‘x’ , 1)**
  - **print (p1.name) --> xohn -- Probably a surprise**
- Could be careful about names always pointing to different memory locations
  - **condition: p1.name ≠ p2.name**
  - **But could be difficult to enforce**

Composite Objects & Expanded Types – 3

- Use expanded types to enforce **aggregation**
  - » **Object has a collection of subparts that are unique to it**

```
feature
    name : expanded NAME
end -- class PERSON
```

- Now guarantee
  - » **p1.name ≠ p2.name**
  - » **Still permit: p1.name . is_equal (p2.name)**

Aliasing

- Occurs when two variables point to the same memory location
- Can lead to surprises but
  - » **Reference assignments needed to benefit from OO**
    - > **Often need two pointers to point to the same object**
  - » **Encapsulation makes it possible to avoid dangers of reference manipulations**