Aliases and shared data

- An *alias* of a variable which refers to an object is a different variable which refers to the same object.

```
K a = new K();
K b = a;
```

- Aliases are useful when used in different objects to represent *shared* information.
- Pointer (or physical) equality and structural equality are two equivalence relations between objects *of the same class*.

Pointer equality

- Pointer equality represents sameness

- Two variables `a` and `b`, of the same class, are *pointer equal*, written `a==b`, if the two variables refer to the same object, i.e. if they are aliases.

- For example, if

```
K a = new K(), b = a, c = new K();
```

- then `a==b` but `c` is not pointer-equal to neither `a` nor `b`

- It can be used to detect shared information.

Structural equality

- Structural equality represents equivalence of parts. This is, two variables `a` and `b`, of the same class, are structurally equal, if each attribute of `a` is equal to the same attribute of `b`.

- For example, if we have

```
class M {
    int q;
    String r;
}
```

- and we have

```
M a = new M(), b = new M();
```

- Then `a` is structurally equal to `b` if `a.q` is equal to `b.q` and `a.r` is equal to `b.r`.

Structural equality

- For example, in the following

```
M a = new M(), b = new M(), c = new M();

a.q = 6;
a.r = ‘‘one’’;

b.q = 3 + 3;
b.r = ‘‘o’’+‘‘ne’’;

c.q = 6;
c.r = ‘‘un’’;
```

- `a` is structurally equal to `b` but `c` is not structurally equal to either `a` or `b`.

Shallow vs. deep structural equality

```
class F {
  int i;
  String j;
  F(int i, String j)
  {
    this.i = i;
    this.j = j;
  }
  boolean equals(F other)
  {
    return this.i == other.i
        && this.j == other.j;
  }
}
```

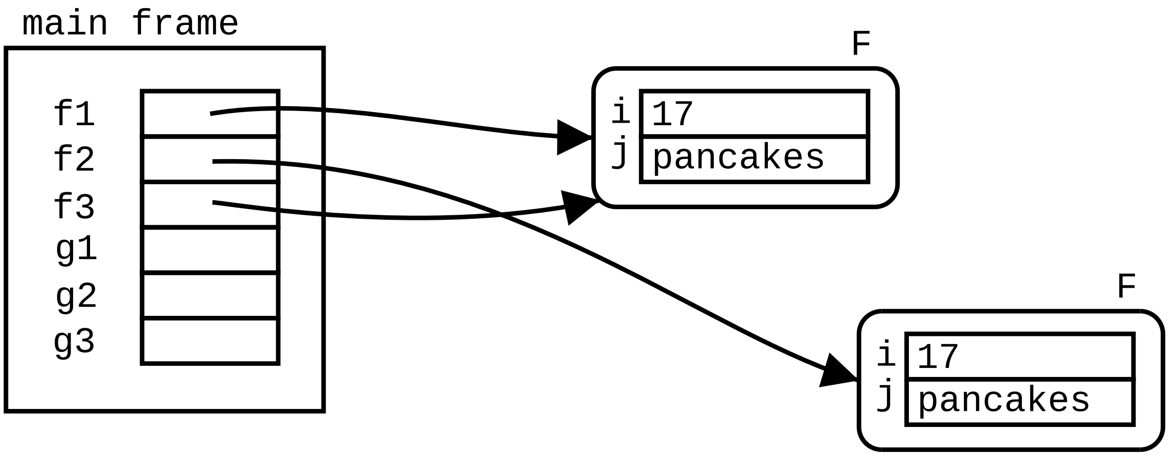
Shallow vs. deep structural equality

```
class G {
  float v;
  F u;
  G(float v, F u)
  {
    this.v = v;
    this.u = u;
  }
  boolean equals(G other)
  {
    return this.v == other.v
        && this.u == other.u; //tests for
                              //shared u
  }
}
```

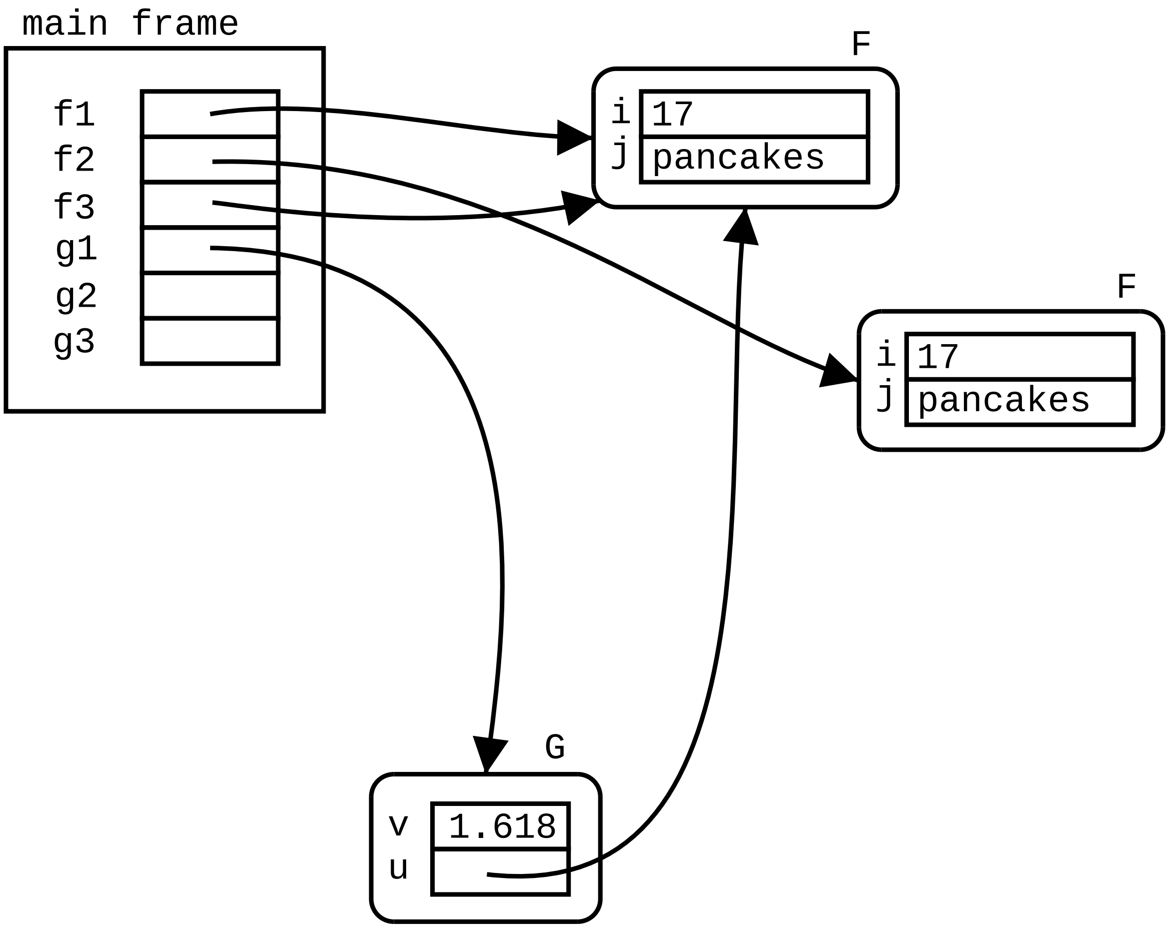
Shallow vs. deep structural equality

```
public class Test {
  public static void main(String[] args)
  {
    F f1 = new F(17, “pancakes”);
    F f2 = new F(17, “pancakes”);
    F f3 = f1;
    G g1 = new G(1.618, f1);
    G g2 = new G(1.618, f2);
    G g3 = new G(1.618, f3);
    if (g1.equals(g2))
      System.out.println("g1 equals g2");
    if (g1.equals(g3))
      System.out.println("g1 equals g3");
    if (g2.equals(g3))
      System.out.println("g2 equals g3");
  }
}
```

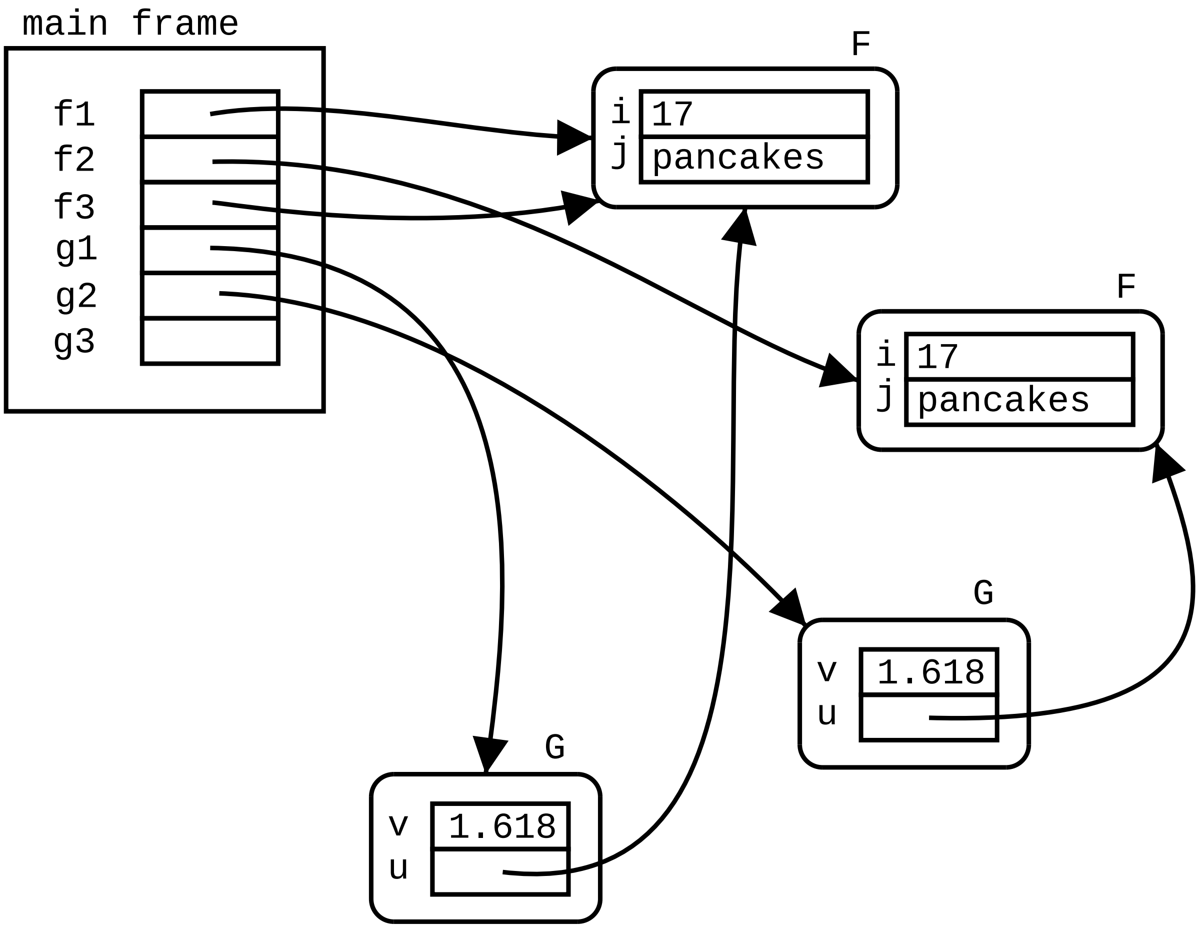
Shallow vs. deep structural equality



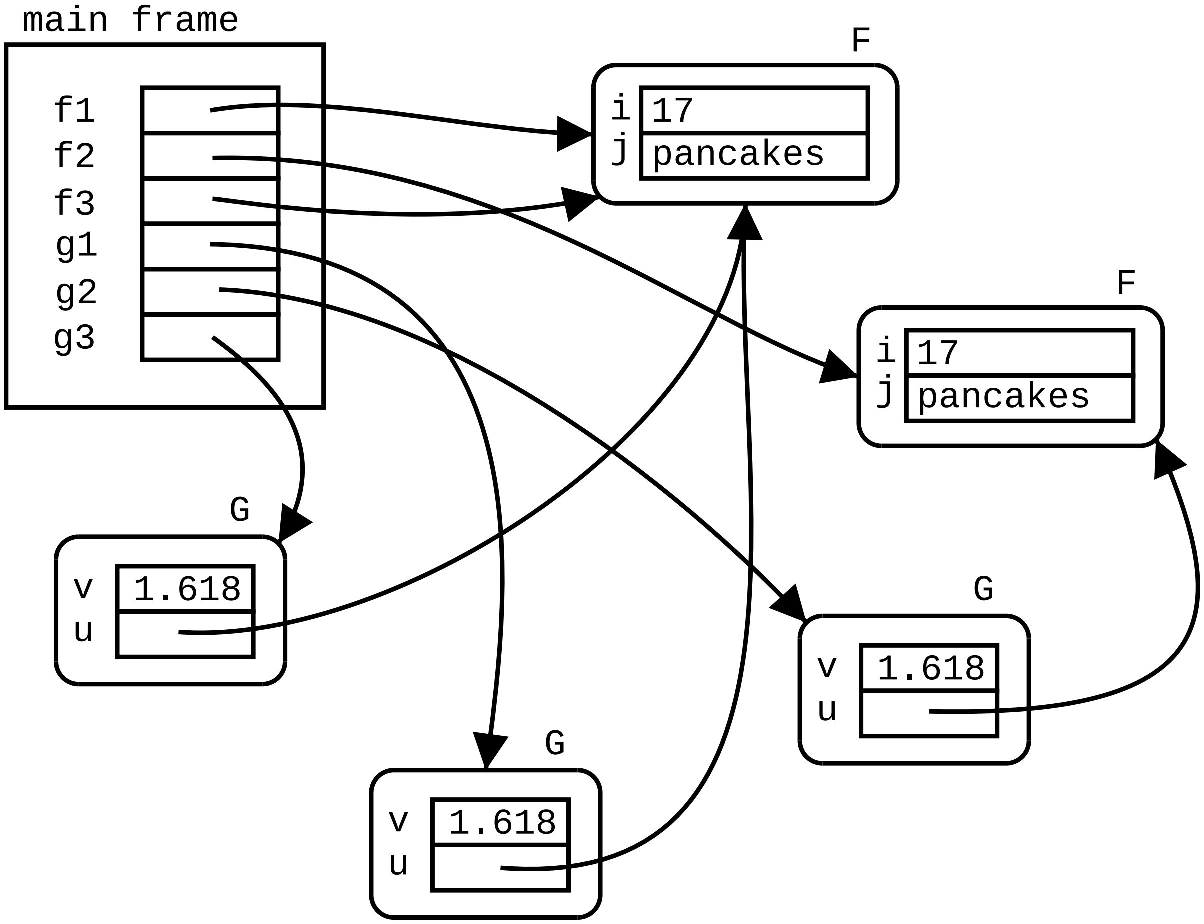
Shallow vs. deep structural equality



Shallow vs. deep structural equality



Shallow vs. deep structural equality



Shallow vs. deep structural equality

```
class G {
  float v;
  F u;
  G(float v, F u)
  {
    this.v = v;
    this.u = u;
  }
  boolean equals(G other)
  {
    return this.v == other.v
        && this.u == other.u;
  }
  boolean deep_equals(G other)
  {
    return v == other.v
        && u.equals(other.u);
  }
}
```

Shallow vs. deep structural equality

- Shallow structural equality is when the equality used to compare the parts (attributes) of the objects is pointer equality.

- Deep structural equality is when the equality used to compare the parts (attributes) of the objects is some structural equality (shallow/deep).

- Shallow equality compares only one level of indirectness, while deep equality might compare many.

Shallow vs. deep structural equality

- In the example, g1 and g3 are shallowly-structurally equal; g1, g2 and g3 are deeply-structurally equal, but g2 is not shallow-structurally equal to neither g1 nor g3. And none of g1, g2, nor g3 is pointer-equal to any of the others.

- Suppose that F had a deep_equals method, then we could have a very_deep_equals method in G:

```
boolean very_deep_equals(G other)
{
  return v == other.v
      && u.deep_equals(other.u);
}
```

Copying and cloning

- Sometimes you don't want to share information, but just give a copy.

- Hence, the purpose of copying an object is to produce a structurally equivalent object to the original, which is not pointer equivalent (i.e. a *different* object whose attributes are equal to the original.)

- For primitive data types, this is done simply by using the assignment statement:

  ```
  x = y;
  ```

- Means copy the value of `y` in the memory location of `x`.

- But, for user-defined data types (classes), one must explicitly create the copy (sometimes called clone) and copy the each of attributes of the object into the copy.

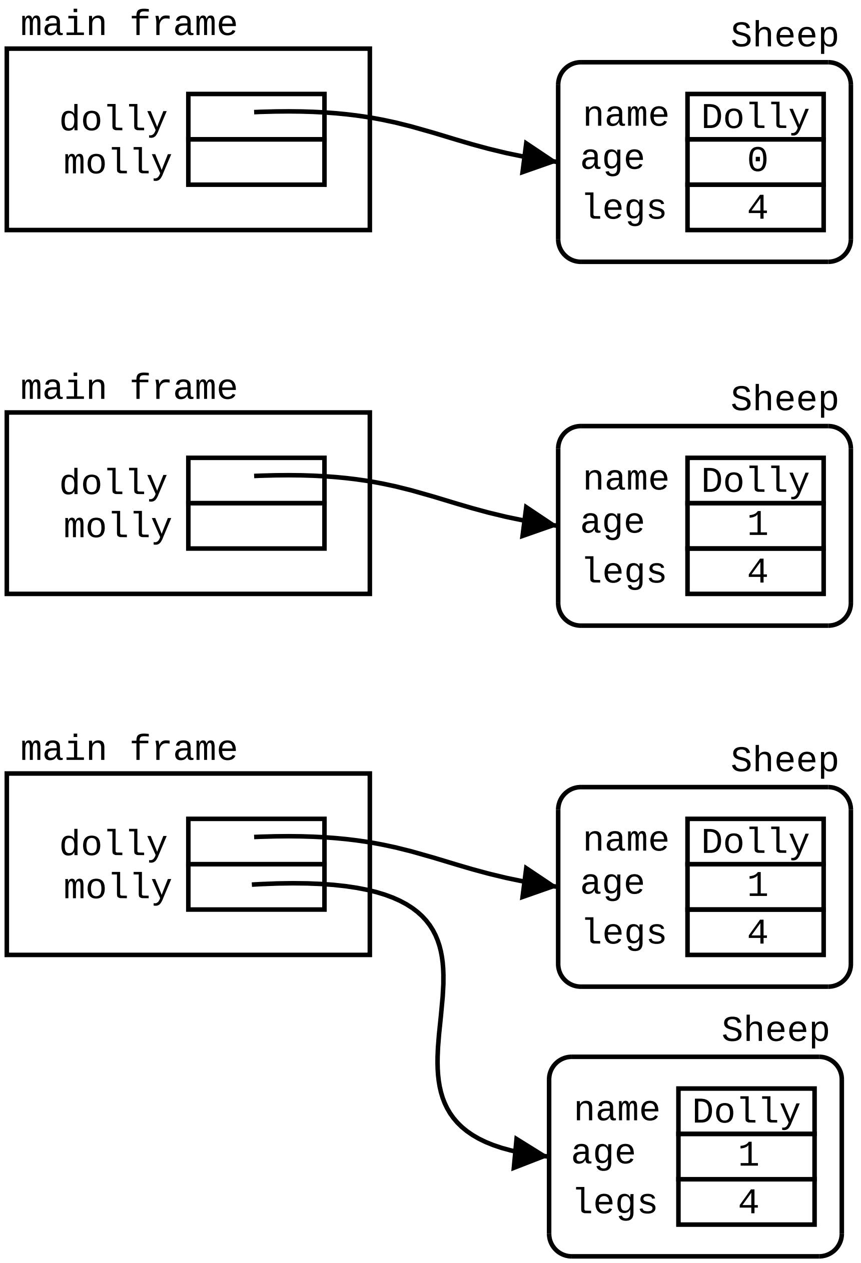
Copying and cloning

```
class Sheep {
  String name;
  int age;
  int legs;
  Sheep(String n)
  {
    name = n;
    age = 0;
    legs = 4;
  }
  void grow_up() { age++; }
  Sheep clone()
  {
    Sheep copy = new Sheep(name);
    copy.age = this.age;
    copy.legs = this.legs; //could be different
    return copy;
  }
}
```

Copying and cloning

```
public class SheepTest {
  public static void main(String[] args)
  {
    Sheep dolly = new Sheep(‘‘Dolly’’);
    dolly.grow_up();
    Sheep molly = dolly.clone();
    dolly.grow_up();
    System.out.println(dolly.age);
    System.out.println(molly.age);
  }
}
```

Copying and cloning



main frame

dolly

molly

Sheep

name Dolly

age 0

legs 4

main frame

dolly

molly

Sheep

name Dolly

age 1

legs 4

main frame

dolly

molly

Sheep

name Dolly

age 1

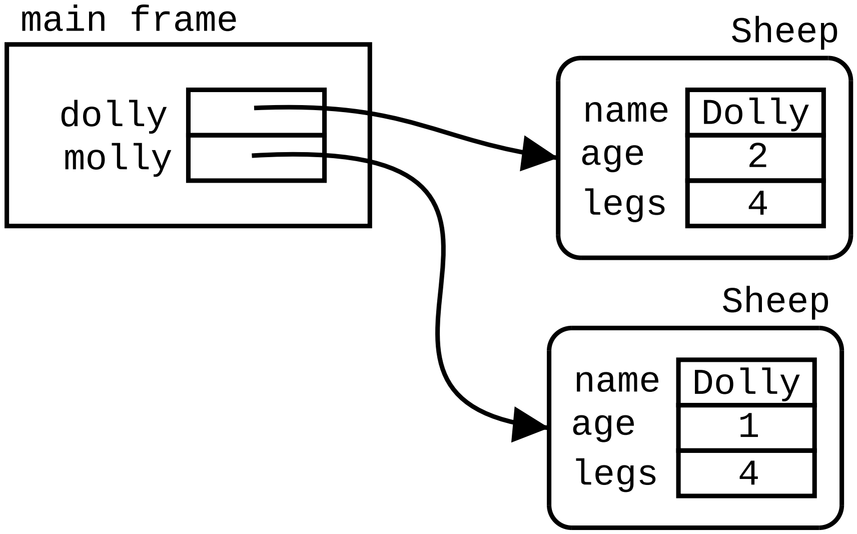
legs 4

Sheep

name Dolly

age 1

legs 4



main frame

dolly

molly

Sheep

name Dolly

age 2

legs 4

Sheep

name Dolly

age 1

legs 4

Shallow copy

```
class Brain {
  String memory;
  Brain()
  {
    memory = ‘‘’’;
  }
  void learn(String something)
  {
    memory = memory + something;
  }
}
```

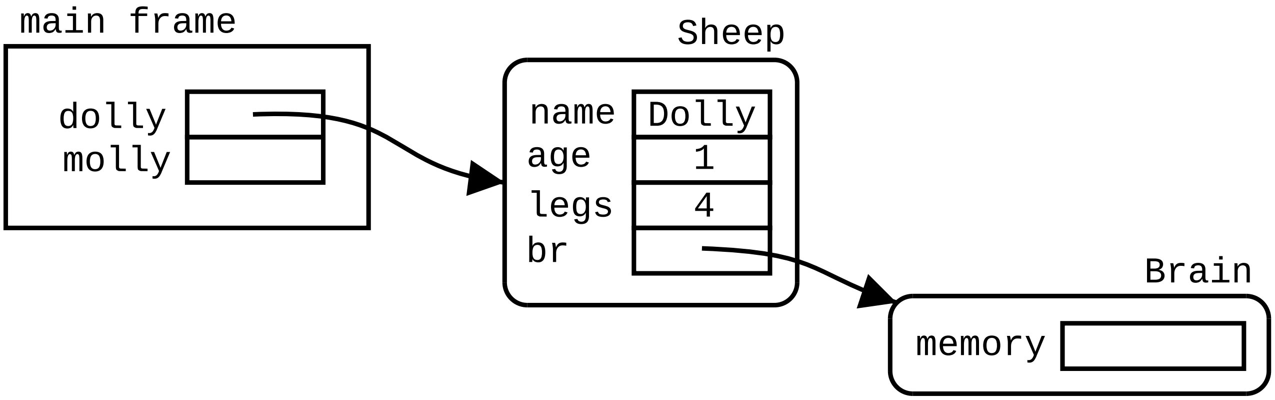
Shallow copy

```
class Sheep {
  String name;
  int age, legs;
  Brain br;
  Sheep(String n)
  {
    name = n;
    br = new Brain();
    age = 0;
    legs = 4;
  }
  void grow_up() { age++; }
  void learn(String something)
  {
    br.learn(something);
  }
  // Contintues below...
  Sheep clone()
  {
    Sheep copy = new Sheep(name);
    copy.age  = this.age;
    copy.legs = this.legs;
    copy.br   = this.br; // Making an alias
    return copy;
  }
} // End of class Sheep
```

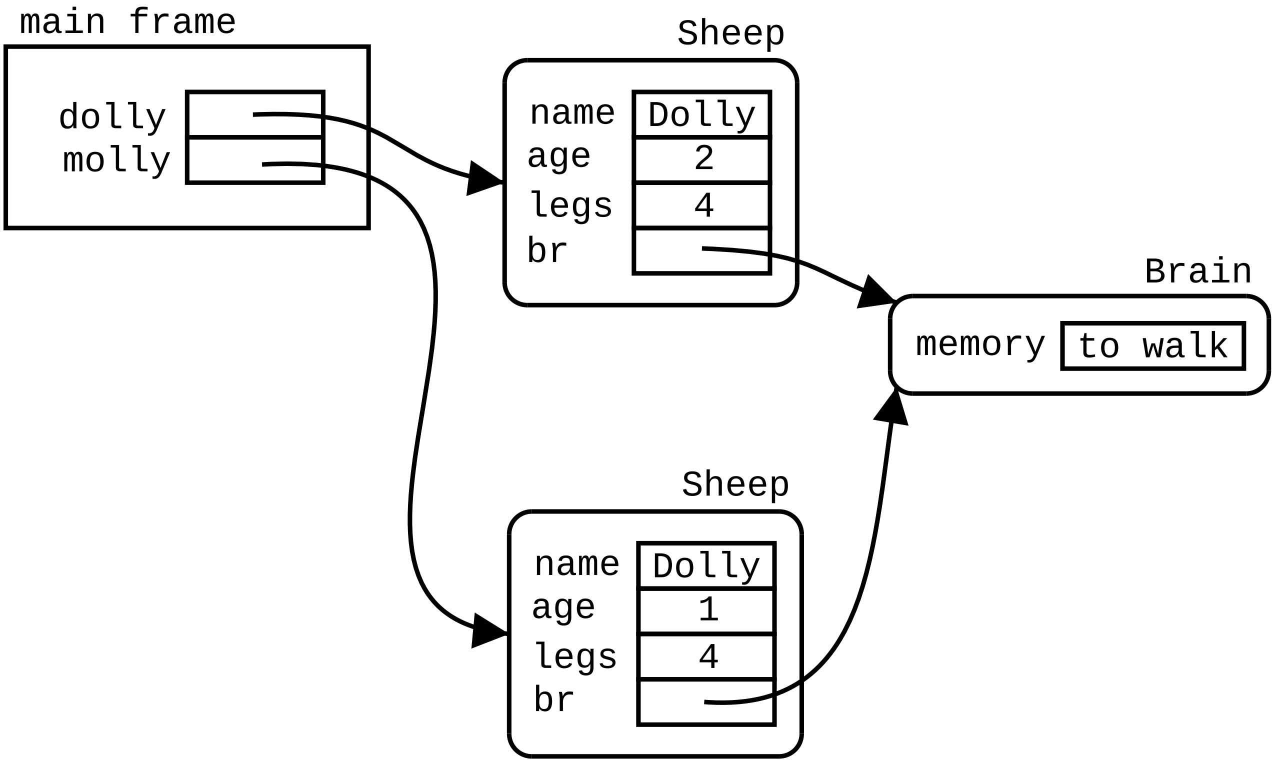
Shallow copy

```
public class SheepTest {
  public static void main(String[] args)
  {
    Sheep dolly = new Sheep(“Dolly”);
    dolly.grow_up();
    Sheep molly = dolly.clone();
    dolly.grow_up();
    molly.learn(“ to walk ”);
    System.out.println(dolly.age);
    System.out.println(molly.age);
    System.out.println(dolly.br.memory);
  }
}
```

Shallow copy



Shallow copy



Deep copy

```
class Brain {
  String memory;
  Brain()
  {
    memory = ‘‘’’;
  }
  void learn(String something)
  {
    memory = memory + something;
  }
  Brain clone()
  {
    Brain copy = new Brain();
    copy.memory = this.memory;
    return copy;
  }
}
```

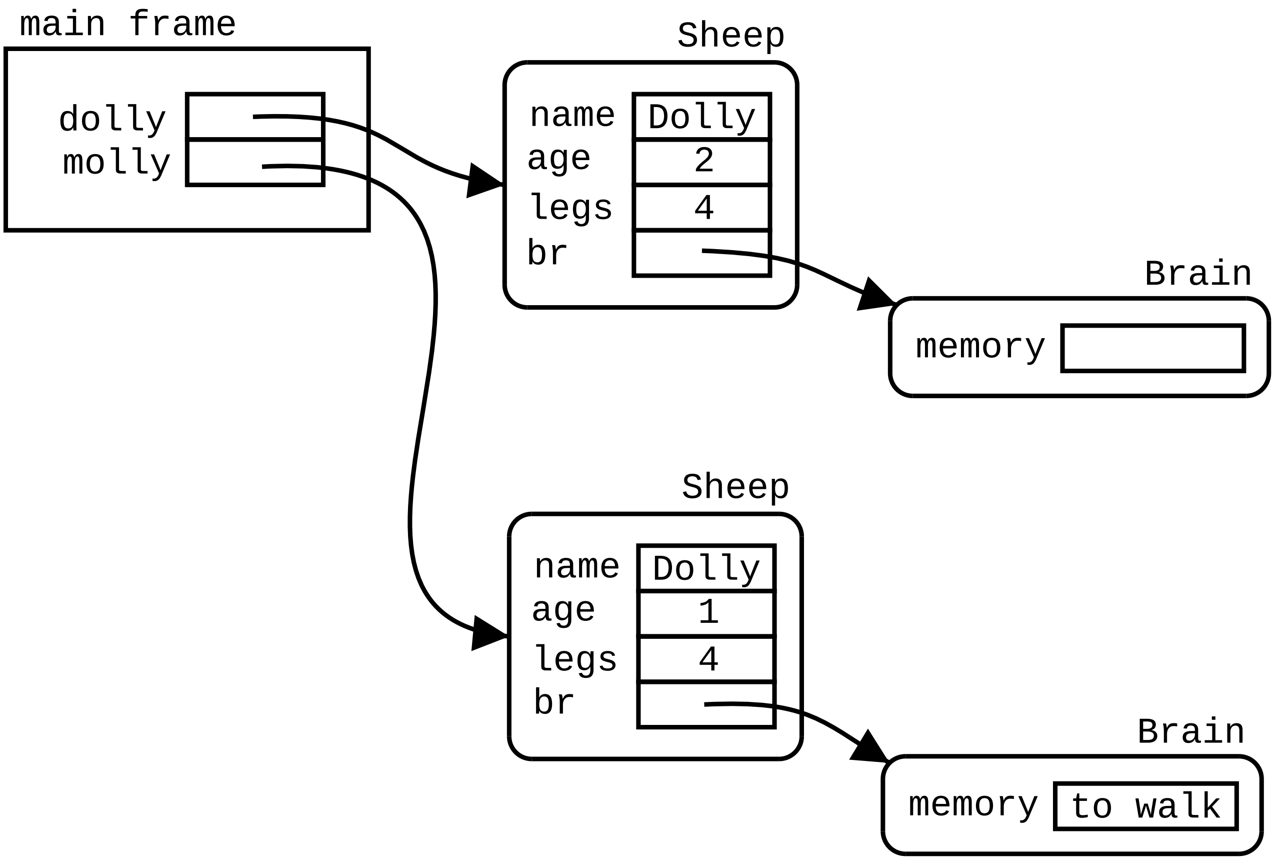
Deep copy

```
class Sheep {
  String name;
  int age, legs;
  Brain br;
  // Same as before...
  Sheep clone()
  {
    Sheep copy = new Sheep(name);
    copy.age  = this.age;
    copy.legs = this.legs;
    copy.br   = this.br; // Making an alias
    return copy;
  }
  Sheep deep_clone()
  {
    Sheep copy = new Sheep(name);
    copy.age = this.age;
    copy.legs = this.legs;
    copy.br = br.clone();
    return copy;
  }
}
```

Deep copy

```
public class SheepTest {
  public static void main(String[] args)
  {
    Sheep dolly = new Sheep(“Dolly”);
    dolly.grow_up();
    Sheep molly = dolly.deep_clone();
    dolly.grow_up();
    molly.learn(“ to walk ”);
    System.out.println(dolly.age);
    System.out.println(molly.age);
    System.out.println(dolly.br.memory);
  }
}
```

Deep copy


main frame
dolly
molly
Sheep
name
Dolly
age
2
legs
4
br
Brain
memory
Sheep
name
Dolly
age
1
legs
4
br
Brain
memory
to walk

Shallow copy vs deep copy

- Shallow copying creates a new object with exactly the same values in its attributes as the original.
  - This is, the original object and its copy are structurally equal, they are not pointer equal, but their attributes are pointer equal.

- Deep copying creates a new object where the attributes of the copy are copies (structurally equivalent) of the attributes of the original.
  - This is, the original object and its copy are structurally equal, they are not pointer equal, and their attributes are structurally equal.

Parameter passing by reference vs. by value

- A programming language that has methods, procedures, and/or functions can pass arguments to the function in several ways:
  - Passing parameters by value: The arguments received by the function are a copy (usually shallow) of the actual arguments.
  - Passing parameters by reference: The arguments received by the function are aliases of the actual arguments.
- In Java, primitive data types are passed by value, but all user-defined data types are passed by reference.

Parameter passing by reference vs. by value

```
public class PassingParameters {
  public static void main(String[] args)
  {
    Sheep dolly = new Sheep(“Dolly”);
    Sheep molly = new Sheep(“Molly”);
    do_something(dolly);
    System.out.println(dolly.br.memory);
    do_something(molly.clone());
    System.out.println(molly.br.memory);
    do_something(molly.deep_clone());
    System.out.println(molly.br.memory);
  }
  static void do_something(Sheep s)
  {
    s.learn(“ to eat ”);
  }
}
```