Defining characteristics of OOP

- A programming language is object-oriented if it supports:
  - Class definitions and class instantiation
  - Message-passing
  - Aggregation
  - Encapsulation
  - Polymorphism
  - Inheritance

Aliases

- Suppose we have

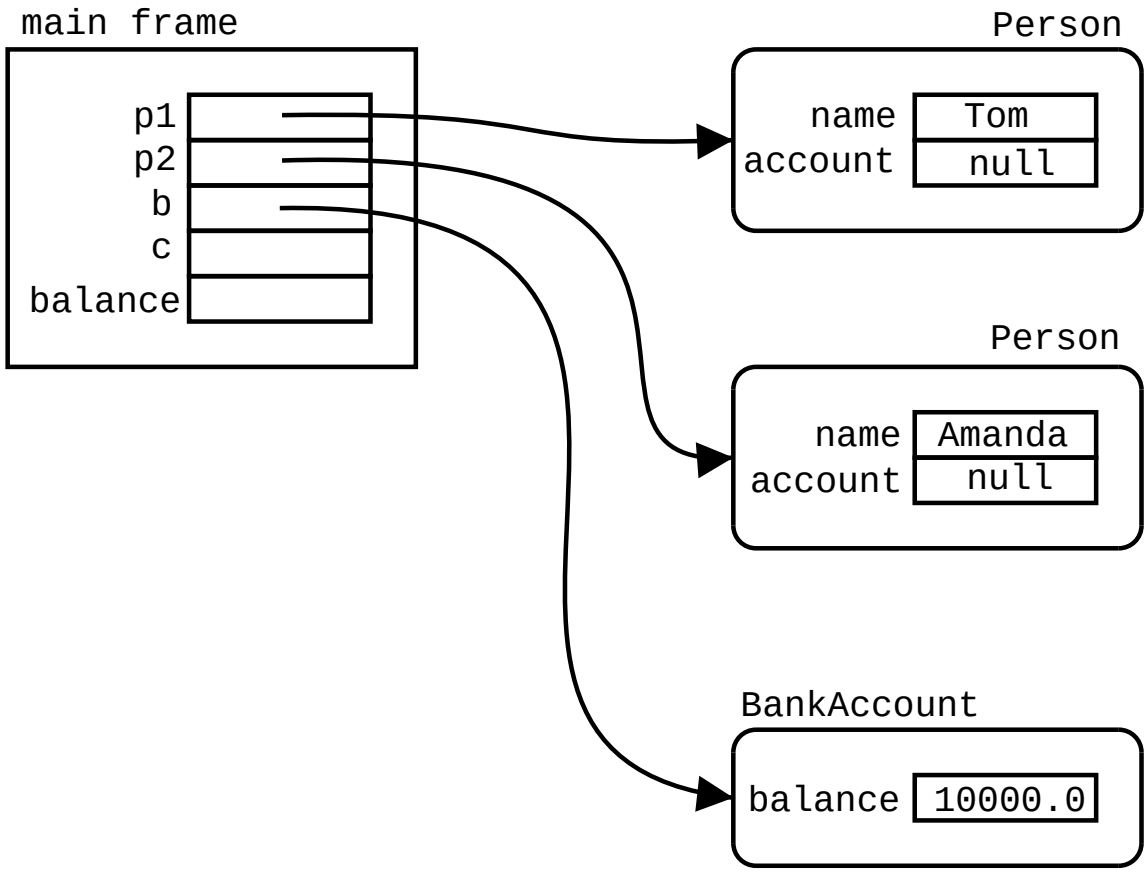
```
A x, y;
x = new A();
y = new A();
```

- Both variables `x` and `y` are `A`'s
- ... but the objects they refer to are different, individual, and independent `A`'s.
- A variable is an alias of another variable if they both point to the same object.

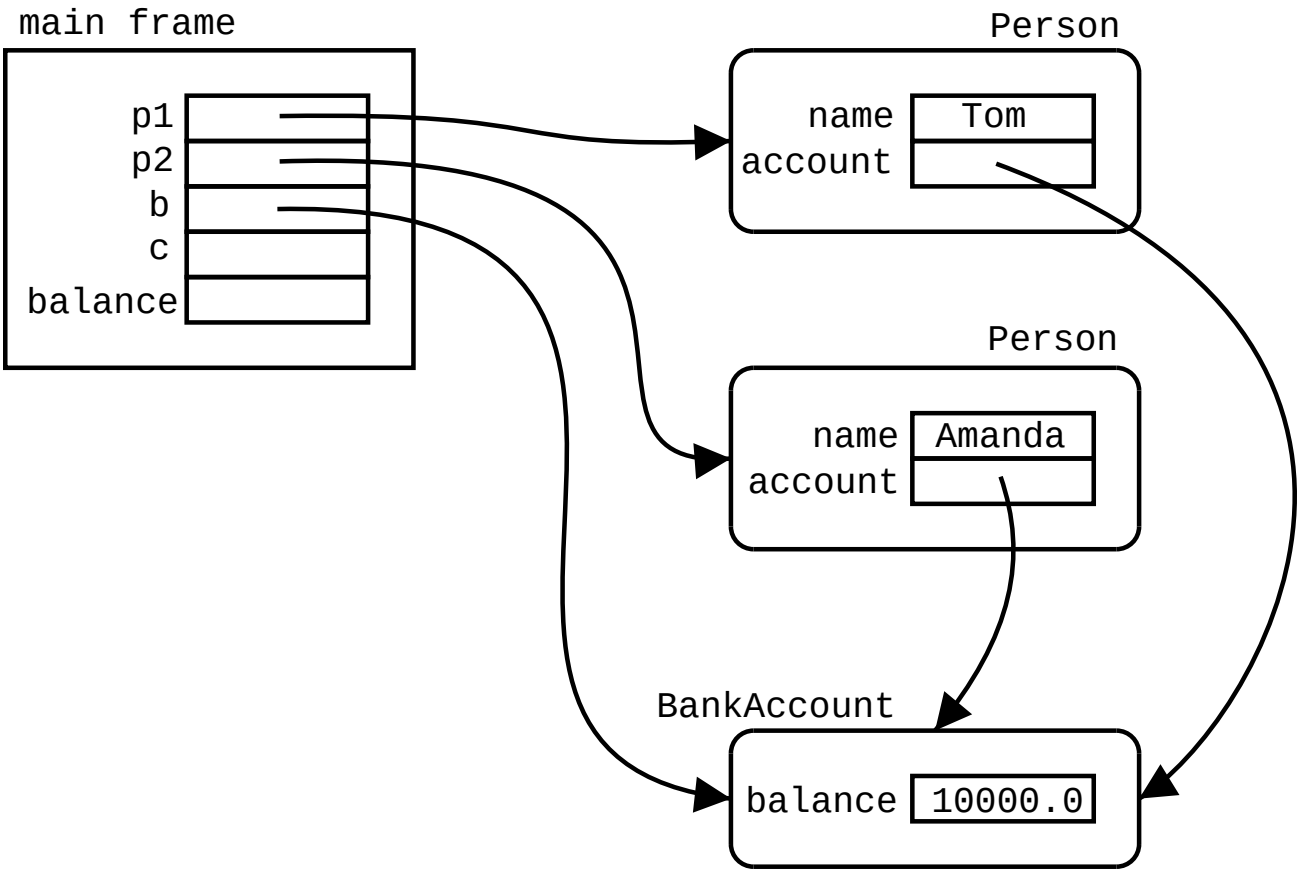
```
A x, y;
x = new A();
y = x;
```

- In this case `x` and `y` are the "same", i.e. they refer to the same object.

Shared references



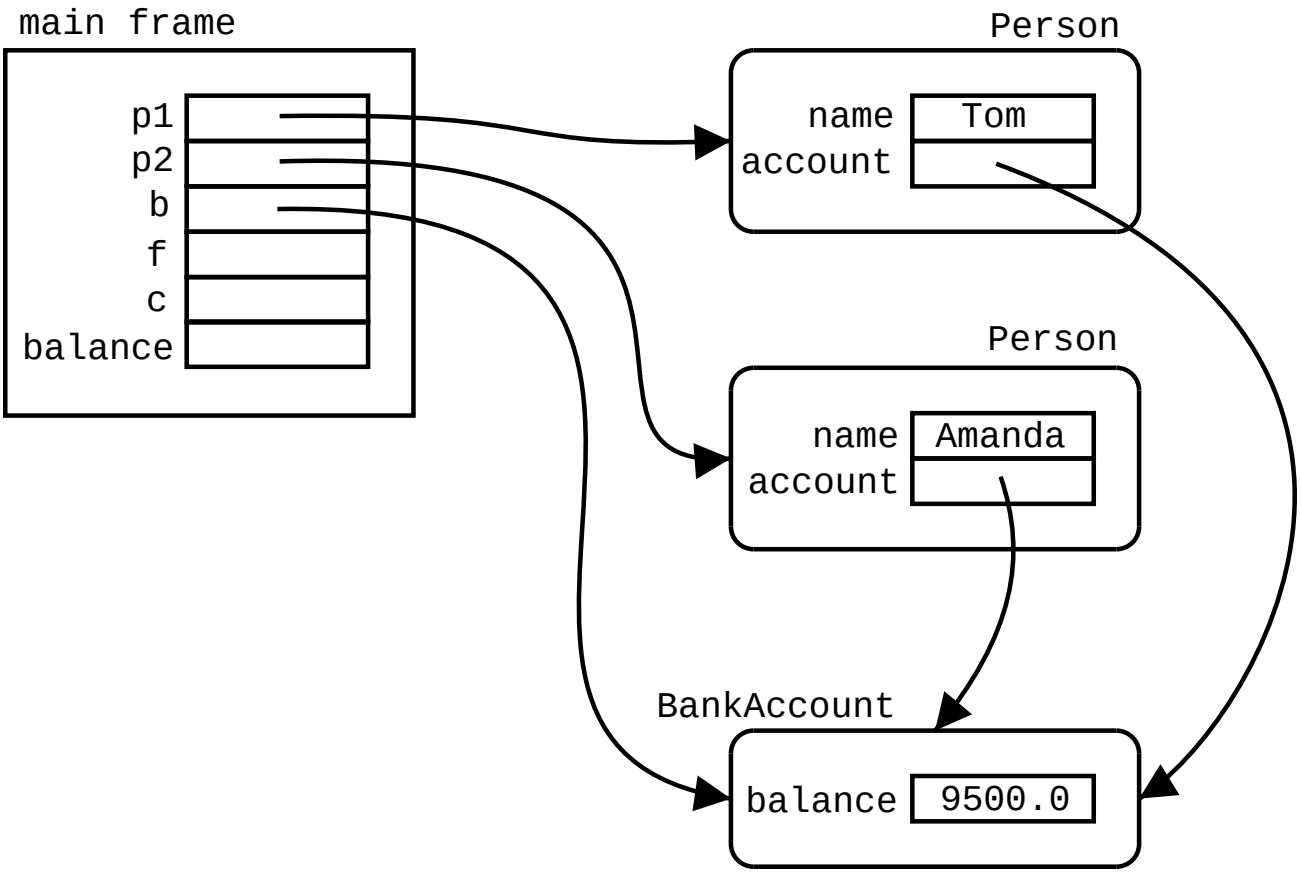
Shared references



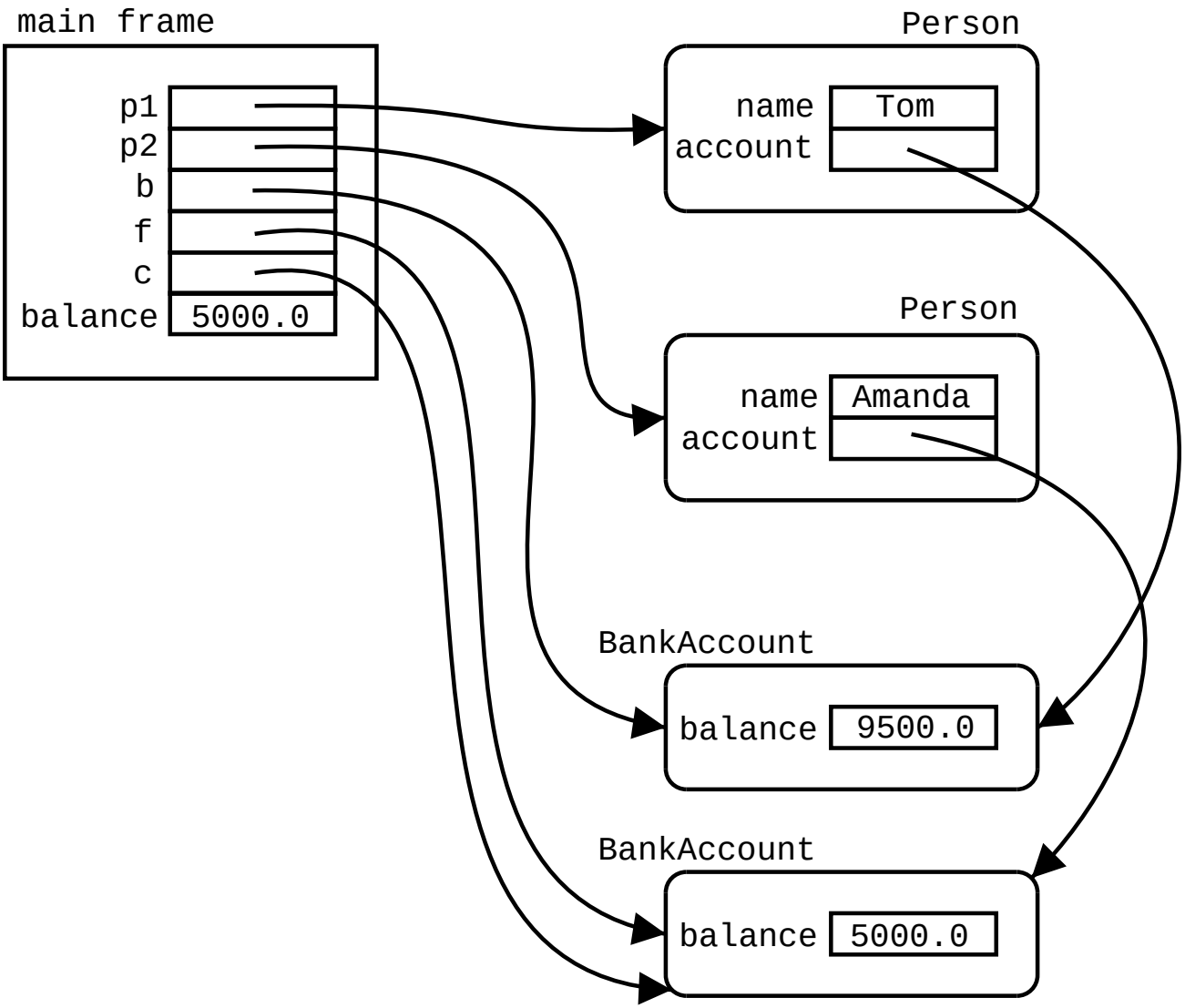
Shared references vs static variables

- In the BankingTest example b is shared between p1 and p2 only, not between all Person objects
- Static variables are like aliases, but they force all objects of the class to share the static reference, while non-static shared references are shared between specific objects.
- Furthermore, if a variable is declared as static the object it refers to is always shared between all objects in the class, while a non-static shared reference might become "unshared".

Shared references vs static variables



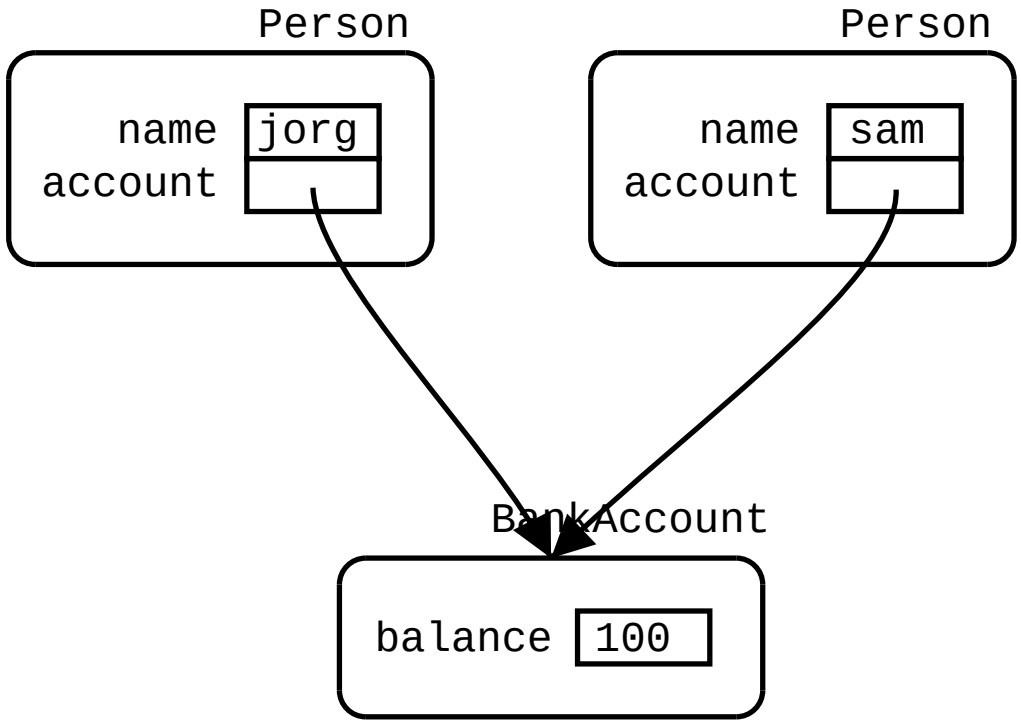
Shared references vs static variables



Equality between objects

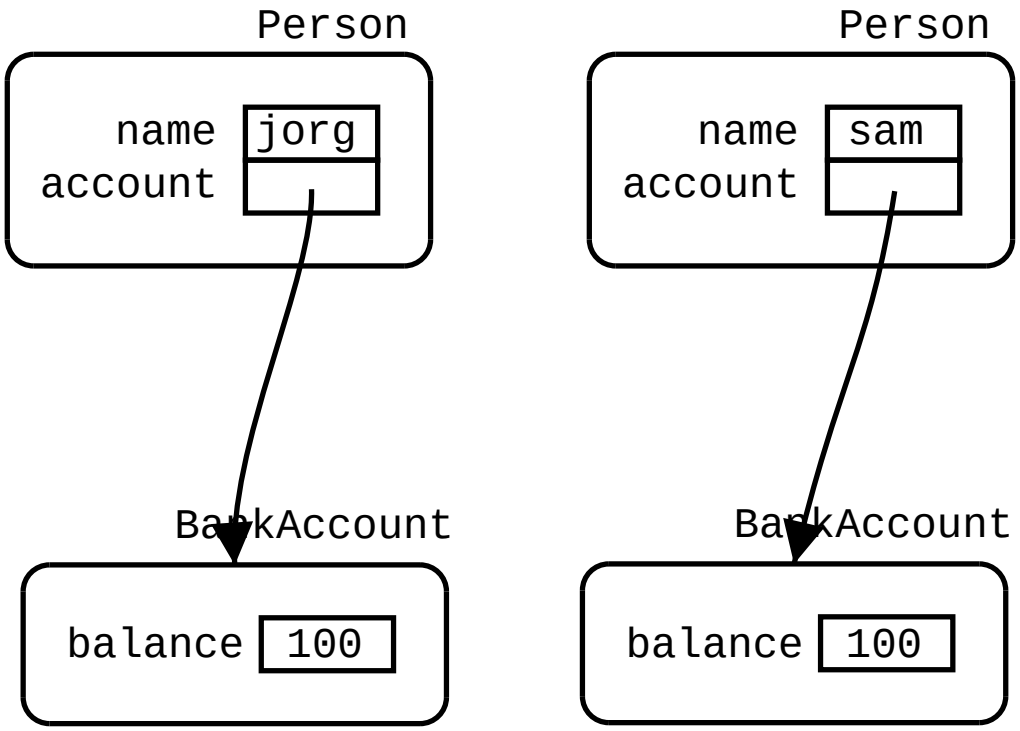
- To check whether two primitive values are equal we use ==
- How do we know if two objects are equal?
- What does it mean to say that two objects are equal?
- *Equality is not the same as "sameness".*
- Possible questions related to equality:
  - Given two people, do they share a bank account?
  - Given two people, do they have the same amount of money in their bank accounts?

Equality between objects



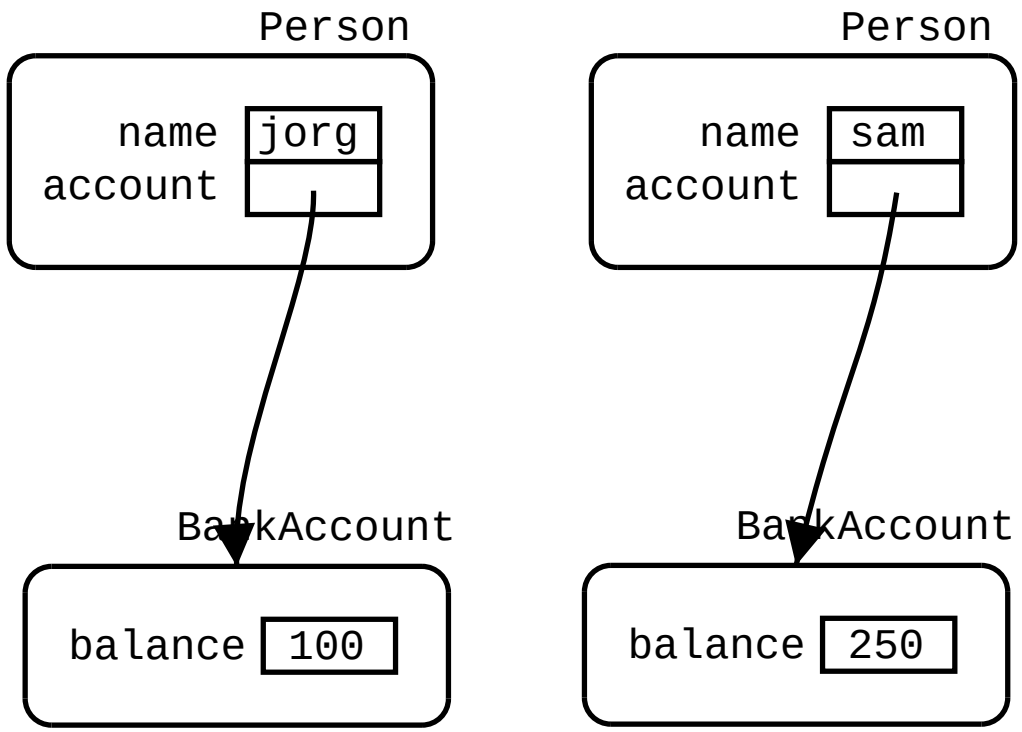
Jorg's account and Sam's account are *pointer equal*, i.e. they are the same.

Equality between objects



Jorg's account and Sam's account are *structurally equal*, i.e. the values of the attributes are the equal.

Equality between objects



Jorg's account and Sam's account are *different*, i.e. the object is not shared and values of the attributes are the different.

Pointer equality

- Pointer equality also called "physical" equality is equality (sameness) of references.
- The == operator is used for testing for pointer equality.
- Pointer equality is used to test for sameness of objects:

```
A x, y;
x = new A();
y = x;
```

- ...then `x == y` is true, but in

```
A x, y;
x = new A();
y = new A();
```

- ... `x == y` is false, even if the attributes of the objects are the same.
- Pointer equality is an equivalence between objects of the same class only.

Example

```
public class BankingTest
{
  public static void main(String[] args)
  {
    Person p1 = new Person(“Tom”);
    Person p2 = new Person(“Amanda”);
    BankAccount b = new BankAccount(10000.0f);
    p1.open_account(b);
    p2.open_account(b);

    BankAccount d = p1.account();
    d.withdraw(500.0f);
    BankAccount c = p2.account();
    if (c == d)
      System.out.println(“It’s a shared account”);
  }
}
```

Structural equality

- Structural equality: when the aggregates (parts) of two different objects are equal
- Structural equality is only between objects of the same class.
- Two objects are structurally equal if their attributes are equal
- Suppose we have a class

```
class A {
    int x, y;
    A(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Structural equality

- and there is some client with

```
A a1 = new A(17, 29);
A a2 = new A(17, 29);
A a3 = new A(17, 80);
```

- then a1 is structurally equal to a2, but a3 is not structurally equal to either a1 or a2.

- If we want to test for structural equality we must explicitely provide the code. This is usually done by writing a method called "equal" or "equals":

Structural equality

```
class A {
    int x, y;
    A(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    boolean equals(A other)
    {
        return this.x == other.a
            && this.y == other.y;
    }
}
```

Structural equality

```
public class Test
{
  public static void main(String[] args)
  {
    A a1 = new A(17, 29);
    A a2 = new A(17, 29);
    A a3 = new A(17, 80);
    if (a1.equals(a2))
      System.out.println(“a1 is equal to a2”);
    if (a2.equals(a3))
      System.out.println(“a2 is equal to a3”);
    if (a1 == a2)
      System.out.println(“a1 is the same as s2”);
  }
}
```

Structural equality vs pointer equality

- Note that
  - If two objects are the same (equal by pointer equality) then they are (structurally) equal, ...
    This is, x == y implies that x.`equals`(y) must evaluate to true.
  - ...but if two objects are structurally equal, they may not be physically the same.
    This is, it may be the case that x.`equals`(y) evaluates to true, but x == y may be false.

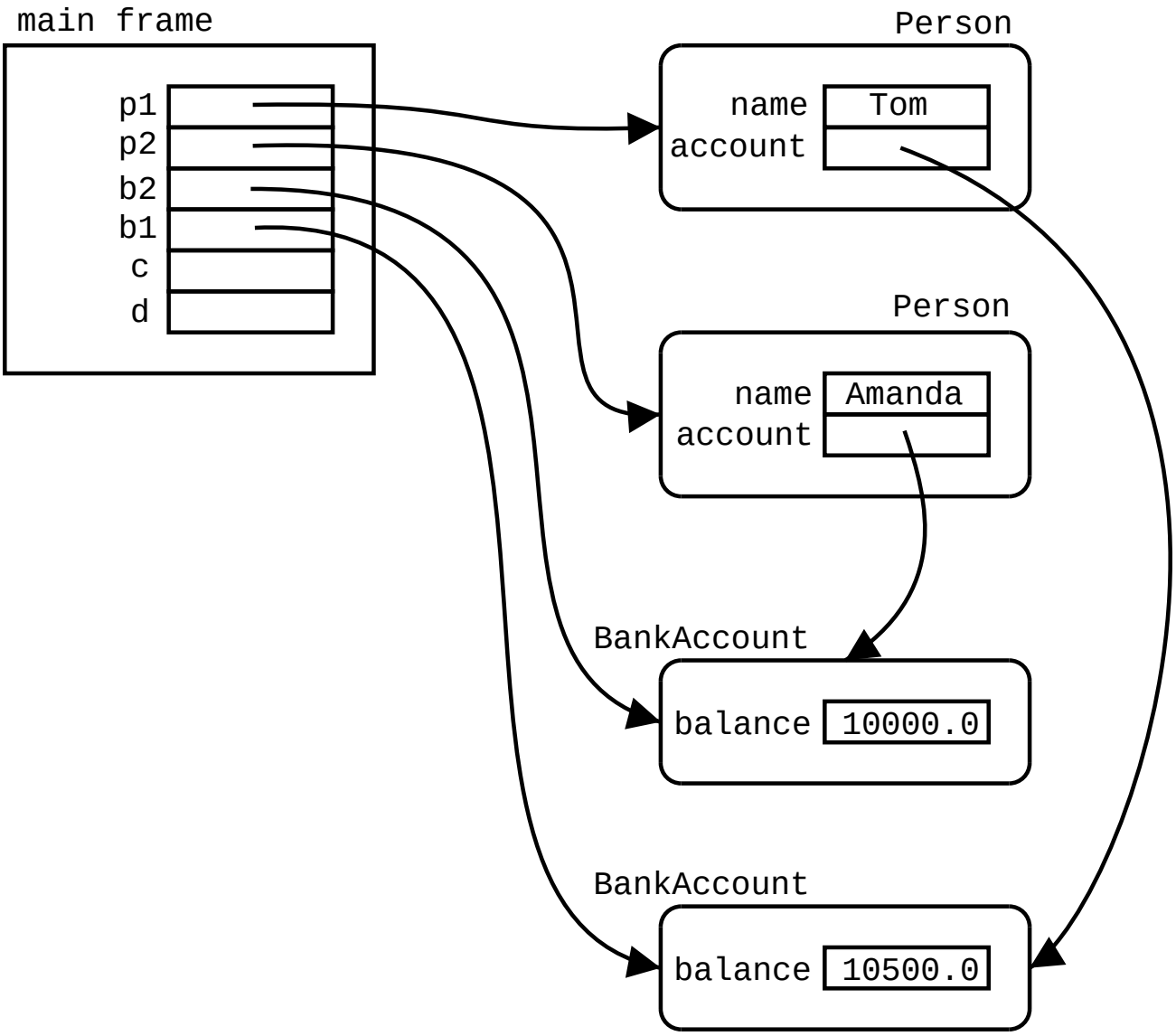
Example

```
public class BankAccount {
  private float balance;
  // ... same as before
  public boolean equals(BankAccount other)
  {
    return this.balance == other.balance;
  }
}
```

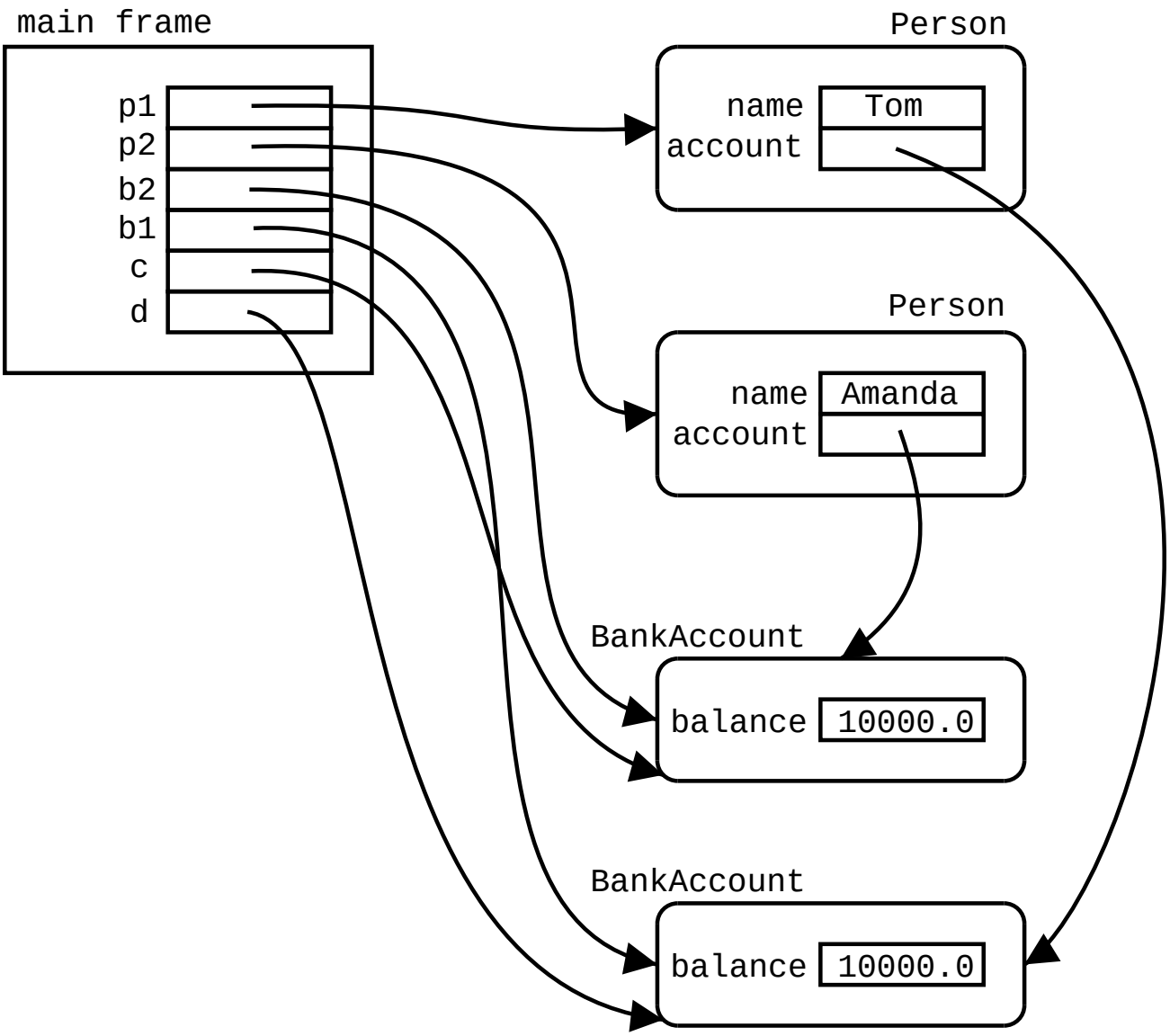
Example

```
public class BankingTest
{
  public static void main(String[] args)
  {
    Person p1 = new Person(“Tom”);
    Person p2 = new Person(“Amanda”);
    BankAccount b1 = new BankAccount(10500.0f);
    BankAccount b2 = new BankAccount(10000.0f);
    p1.open_account(b1);
    p2.open_account(b2);
    BankAccount d = p1.account();
    d.withdraw(500.0f);
    BankAccount c = p2.account();
    if (c.equals(d))
      System.out.println(“They are equal accounts”
  }
}
```

Example


main frame
p1
p2
b2
b1
c
d
Person
name
Tom
account
Person
name
Amanda
account
BankAccount
balance
10000.0
BankAccount
balance
10500.0

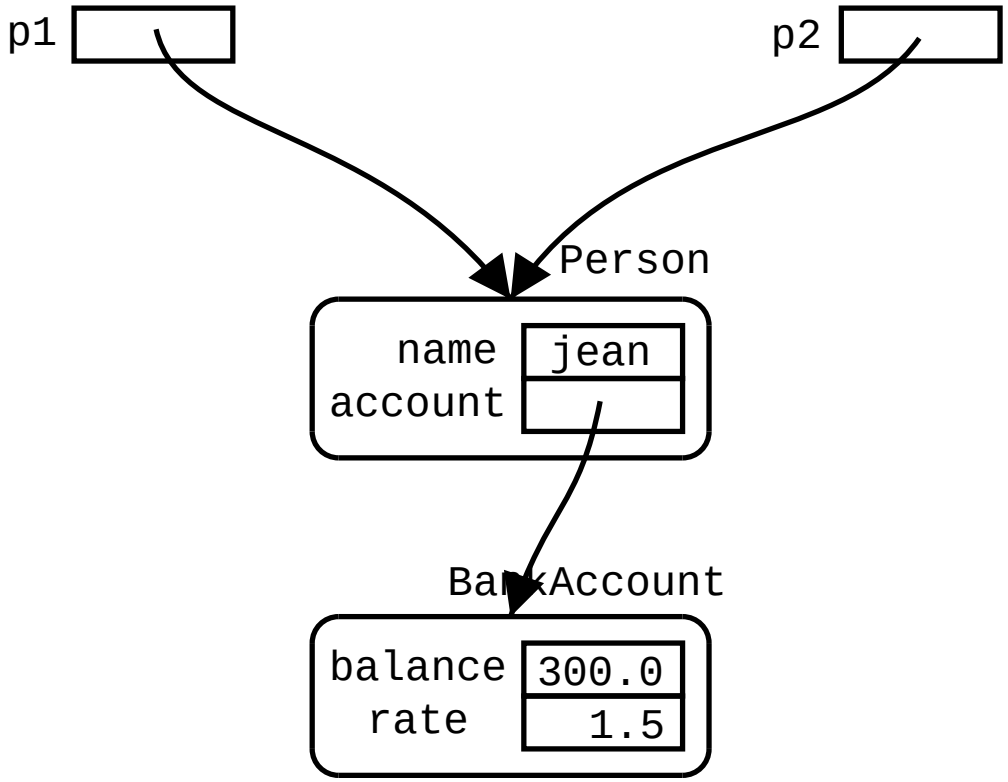
Example


main frame
p1
p2
b2
b1
c
d
Person
name
Tom
account
Person
name
Amanda
account
BankAccount
balance
10000.0
BankAccount
balance
10000.0

Shallow vs. deep structural equality

- Two objects are *structurally equal* if their attributes are *equal*
- There are two main kinds of structural equality:
  - Shallow equality
  - Deep equality
- Two objects are *shallow-equal* if their attributes are physically equal
- Two objects are *deeply-equal* if their attributes are structurally equal

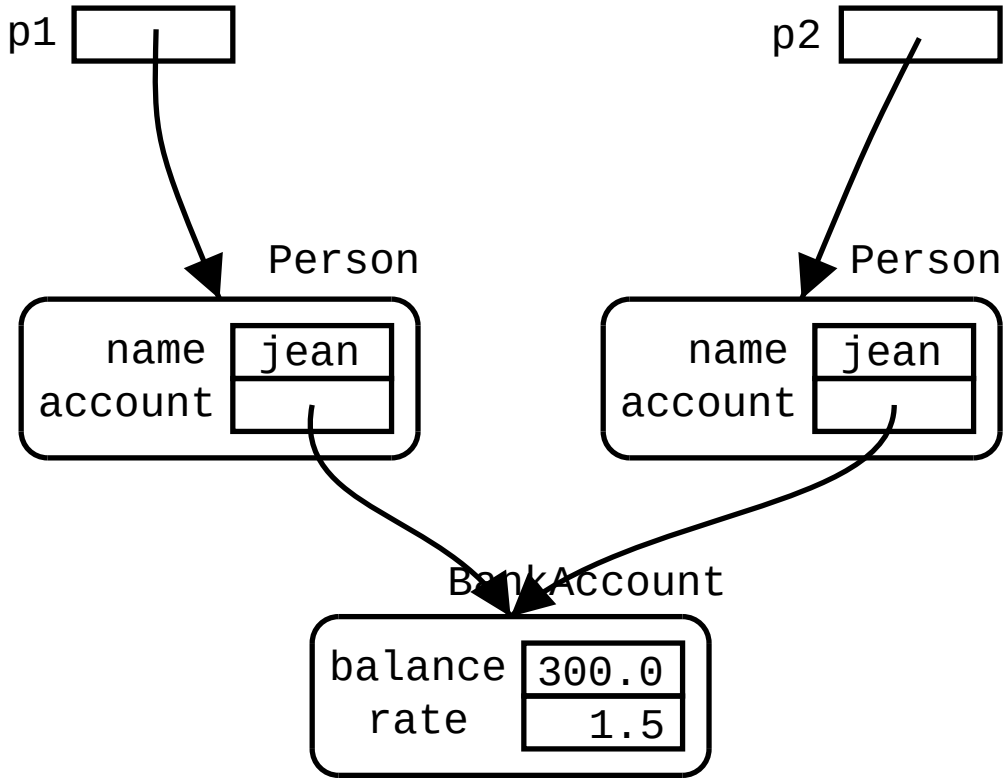
Shallow vs. deep structural equality



p1 and p2 are physically equal

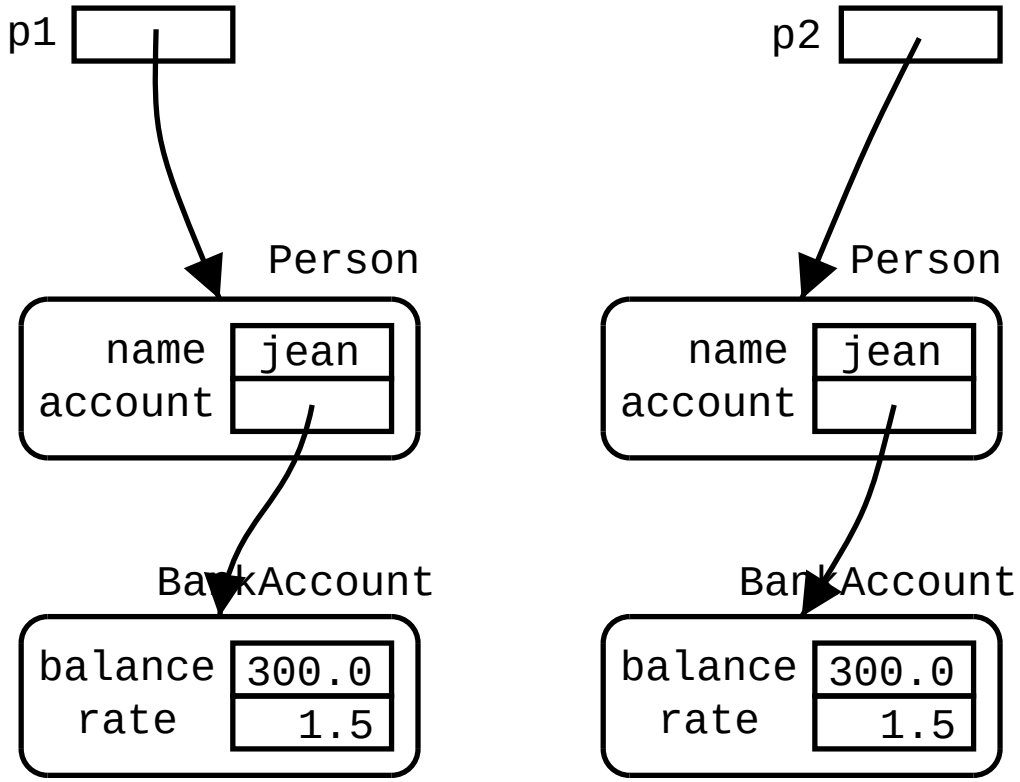
```
p1 == p2
```

Shallow vs. deep structural equality



p1 and p2 are shallow-equal

Shallow vs. deep structural equality



p1 and p2 are deeply equal

Shallow vs. deep structural equality

```
public class BankAccount {
  private double balance, rate;
  // ...
  public double getBalance()
  {
    return balance;
  }
  public double getRate()
  {
    return rate;
  }
  public boolean equals(BankAccount other)
  {
    return this.balance == other.balance
        && this.rate == other.rate;
  }
}
```

Shallow vs. deep structural equality

```
public class Person {
  private String name;
  private BankAccount account;
  Person(String name) {
    this.name = name;
    account = null;
  }
  public void openAccount(BankAccount a)
  {
    account = a;
  }
  public boolean shallow_equals(Person other)
  {
    return this.name == other.name
        && this.account == other.account;
  }
  public boolean deep_equals(Person other)
  {
    return this.name.equals(other.name)
        && this.account.equals(other.account);
  }
}
```

Shallow vs. deep structural equality

```
public class Test {
  public static void main(String[] args)
  {
    Person p1 = new Person(“Tom”);
    Person p2 = new Person(“Amanda”);
    BankAccount b1 = new BankAccount(10000.0f);
    BankAccount b2 = new BankAccount(10000.0f);
    p1.open_account(b1);
    p2.open_account(b2);
    if (p1 == p2)
      System.out.println(“Physically equal”);
    if (p1.shallow_equals(p2))
      System.out.println(“Shallow-equal”);
    if (p1.deep_equals(p2))
      System.out.println(“Deeply equal”);
  }
}
```

Shallow vs. deep structural equality

```
public class Test {
  public static void main(String[] args)
  {
    Person p1 = new Person(“Tom”);
    Person p2 = new Person(“Tom”);
    BankAccount b1 = new BankAccount(10000.0f);
    BankAccount b2 = new BankAccount(20000.0f);
    p1.open_account(b1);
    p2.open_account(b2);
    if (p1 == p2)
      System.out.println(“Physically equal”);
    if (p1.shallow_equals(p2))
      System.out.println(“Shallow-equal”);
    if (p1.deep_equals(p2))
      System.out.println(“Deeply equal”);
  }
}
```

Shallow vs. deep structural equality

```
public class Test {
  public static void main(String[] args)
  {
    Person p1 = new Person(“Tom”);
    Person p2 = new Person(“Tom”);
    BankAccount b1 = new BankAccount(10000.0f);
    BankAccount b2 = new BankAccount(10000.0f);
    p1.open_account(b1);
    p2.open_account(b2);
    if (p1 == p2)
      System.out.println(“Physically equal”);
    if (p1.shallow_equals(p2))
      System.out.println(“Shallow-equal”);
    if (p1.deep_equals(p2))
      System.out.println(“Deeply equal”);
  }
}
```

Shallow vs. deep structural equality

```
public class Test {
  public static void main(String[] args)
  {
    Person p1 = new Person(“Tom”);
    Person p2 = new Person(“Tom”);
    BankAccount b1 = new BankAccount(10000.0f);
    BankAccount b2 = b1;
    p1.open_account(b1);
    p2.open_account(b2);
    if (p1 == p2)
      System.out.println(“Physically equal”);
    if (p1.shallow_equals(p2))
      System.out.println(“Shallow-equal”);
    if (p1.deep_equals(p2))
      System.out.println(“Deeply equal”);
  }
}
```

Shallow vs. deep structural equality

```
public class Test {
  public static void main(String[] args)
  {
    Person p1 = new Person(“Tom”);
    Person p2 = p1;
    BankAccount b1 = new BankAccount(10000.0f);
    BankAccount b2 = new BankAccount(10000.0f);
    p1.open_account(b1);
    p2.open_account(b2);
    if (p1 == p2)
      System.out.println(“Physically equal”);
    if (p1.shallow_equals(p2))
      System.out.println(“Shallow-equal”);
    if (p1.deep_equals(p2))
      System.out.println(“Deeply equal”);
  }
}
```

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.equals(other.j);
  }
}
```

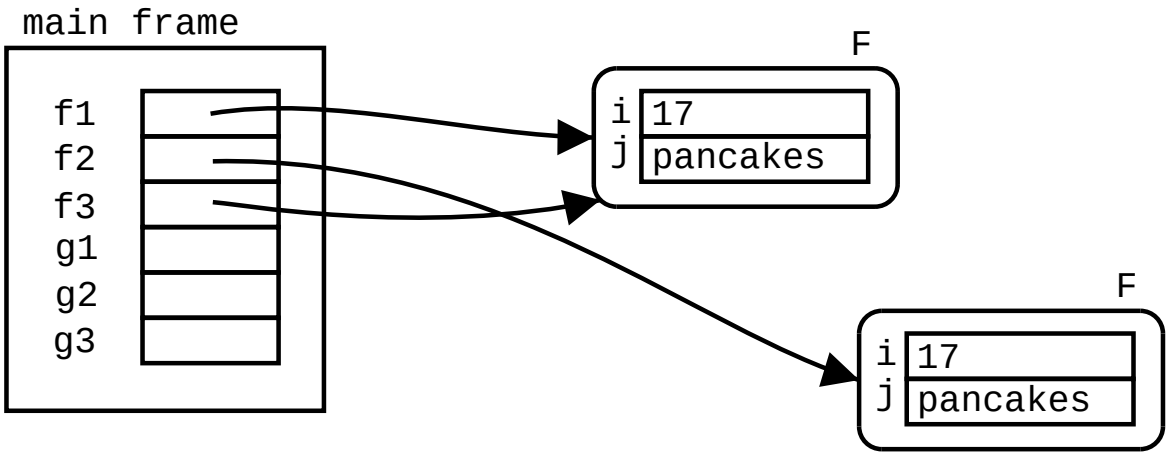
Shallow vs. deep structural equality

```
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
  }
}
```

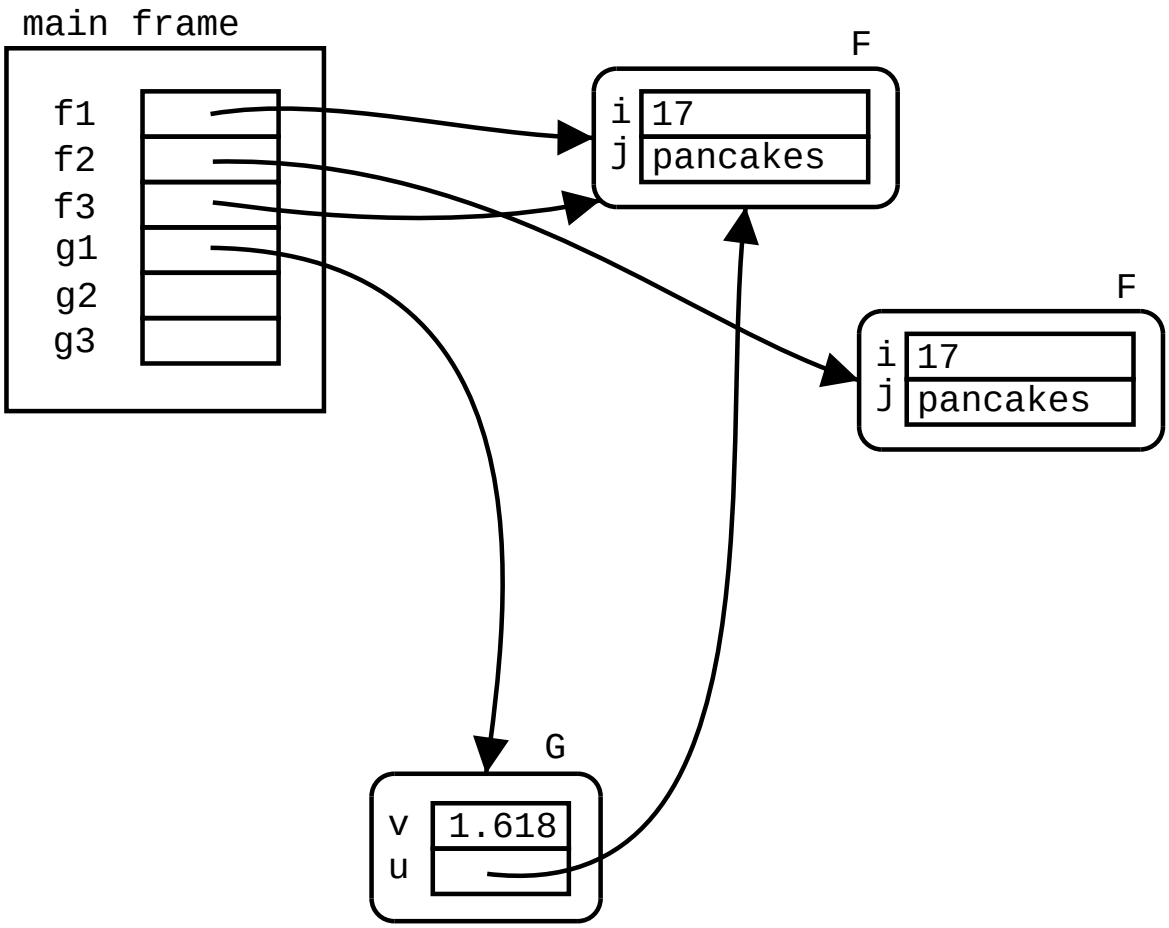
Shallow vs. deep structural equality

```
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");
  }
}
```

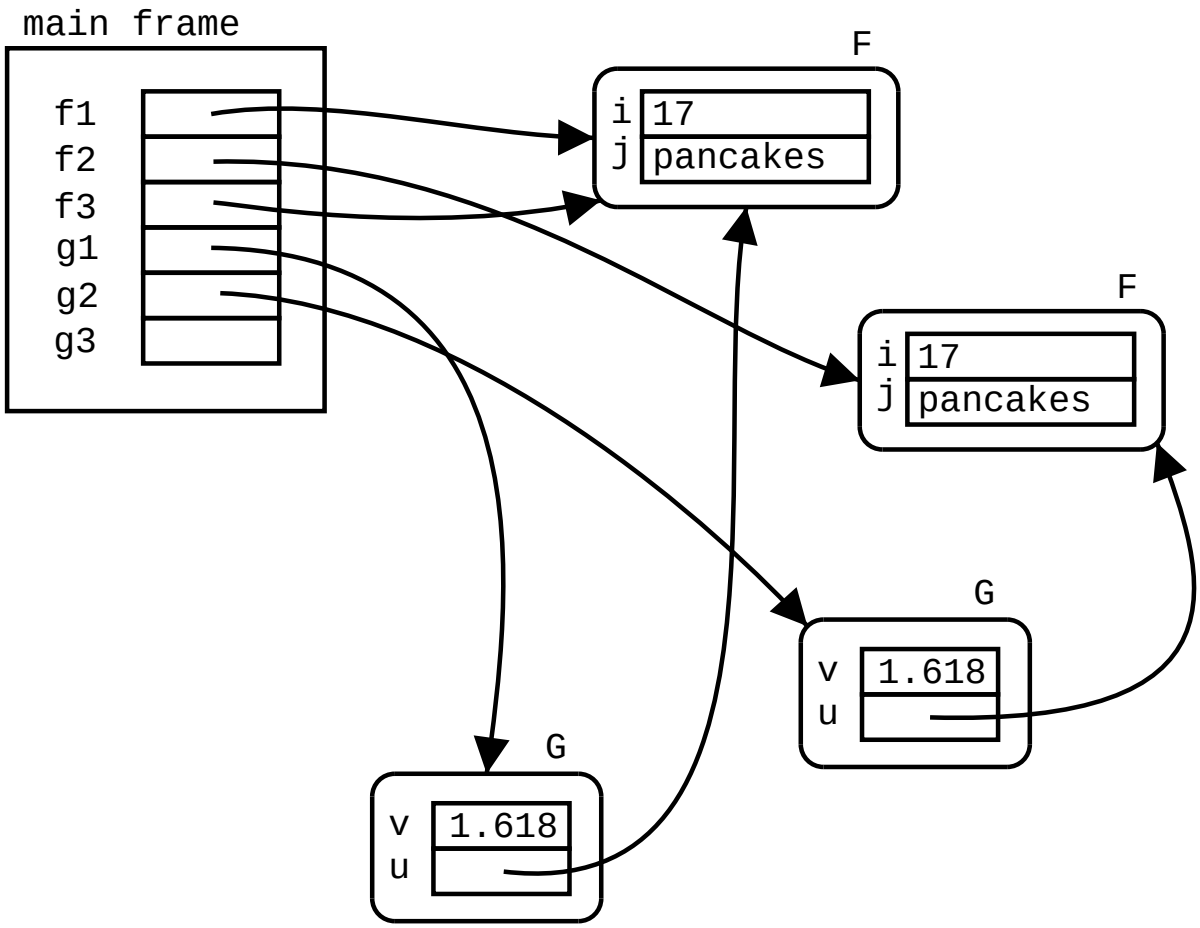
Shallow vs. deep structural equality



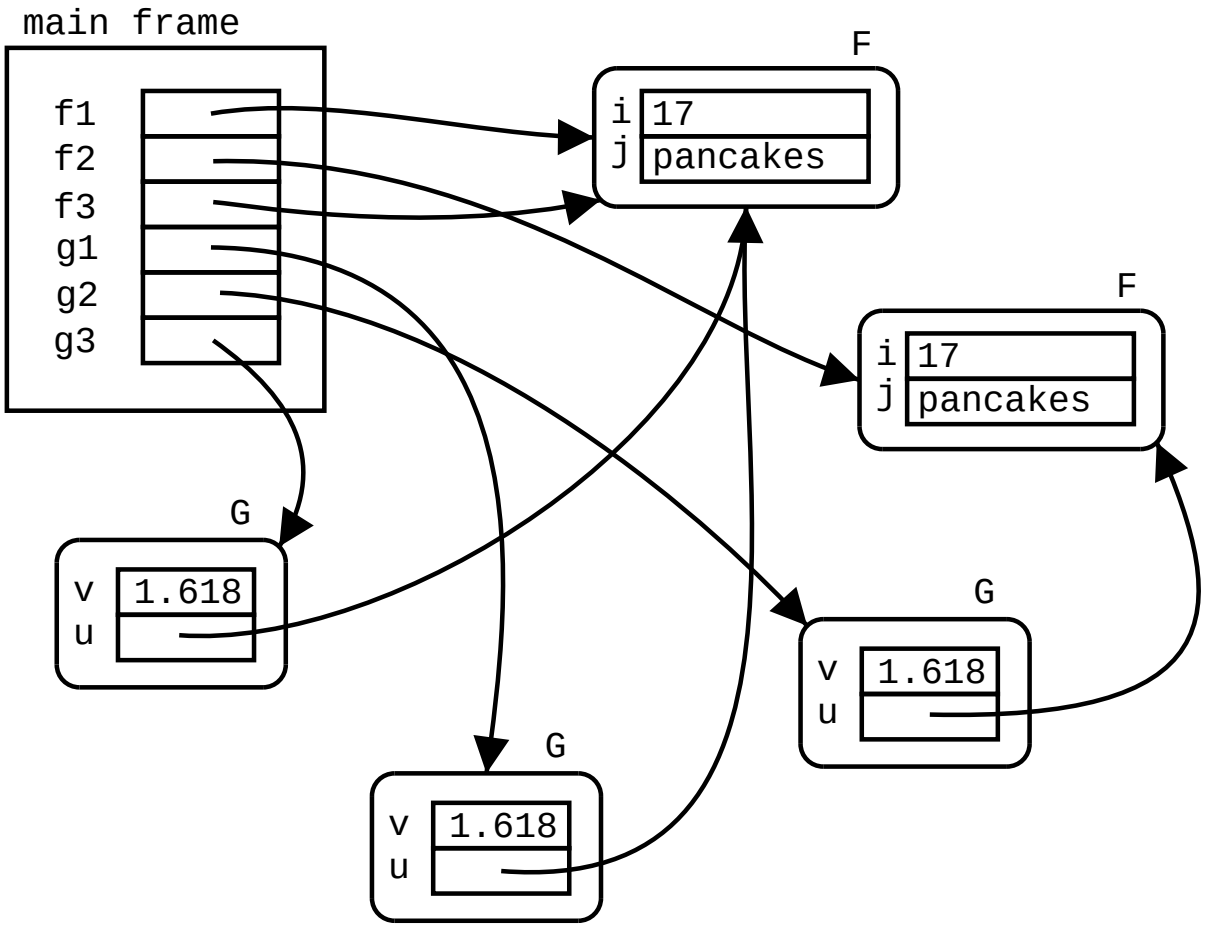
Shallow vs. deep structural equality



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.

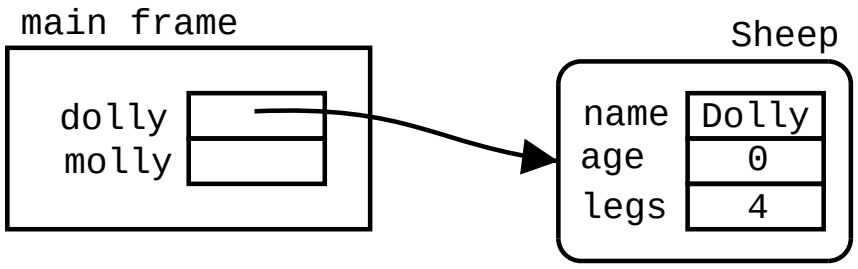
Copying and cloning

```
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;
  }
}
```

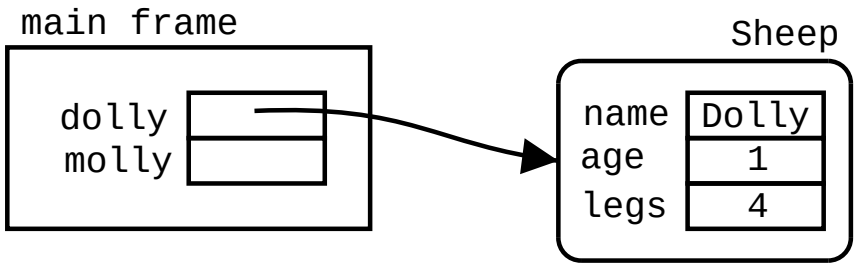
Copying and cloning

```
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);
  }
}
```

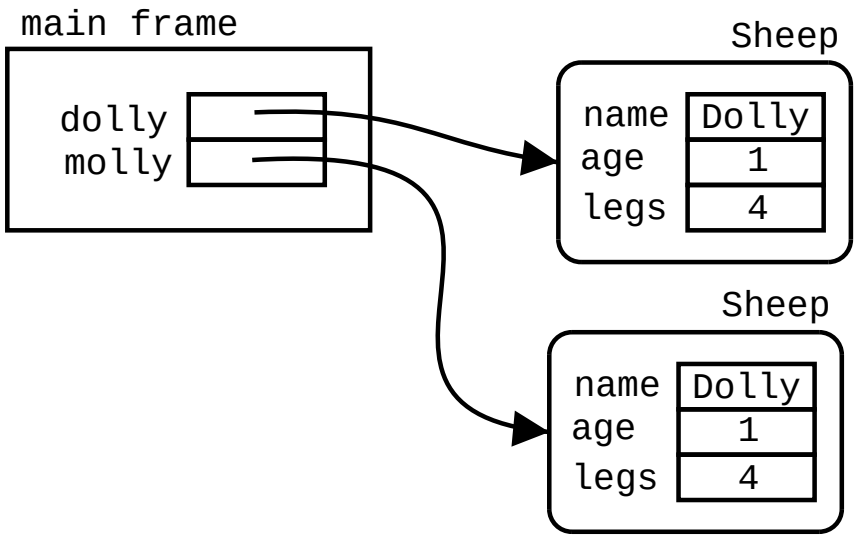
Copying and cloning



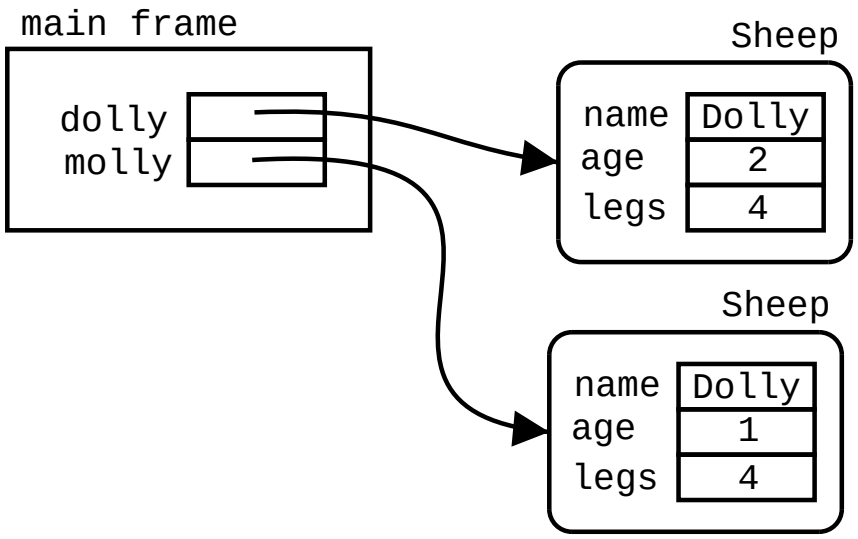
Copying and cloning



Copying and cloning



Copying and cloning



Shallow copy

```
class Brain {
  String memory;
  Brain()
  {
    memory = “”;
  }
  void learn(String something)
  {
    memory = memory + something;
  }
}
```

Shallow copy

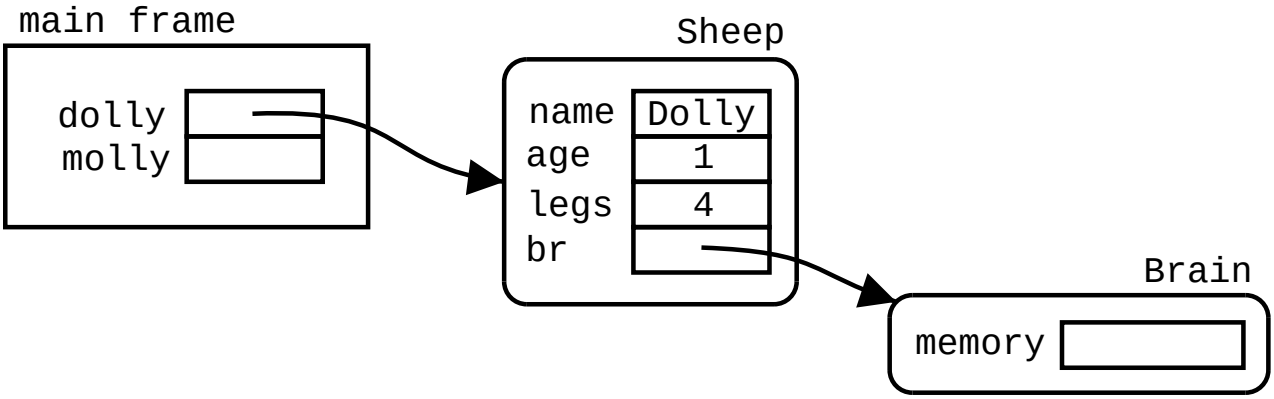
```
class Sheep {
  String name;
  int age, legs;
  Brain br;
  Sheep(String n)
  {
    name = n;
    br = new Brain();
    age = 0;
    legs = 4;
  }
  void grow_up() { age++; }
  void learn(String something)
  {
    br.learn(something);
  }
  // Contintues below...
```

```
    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
```

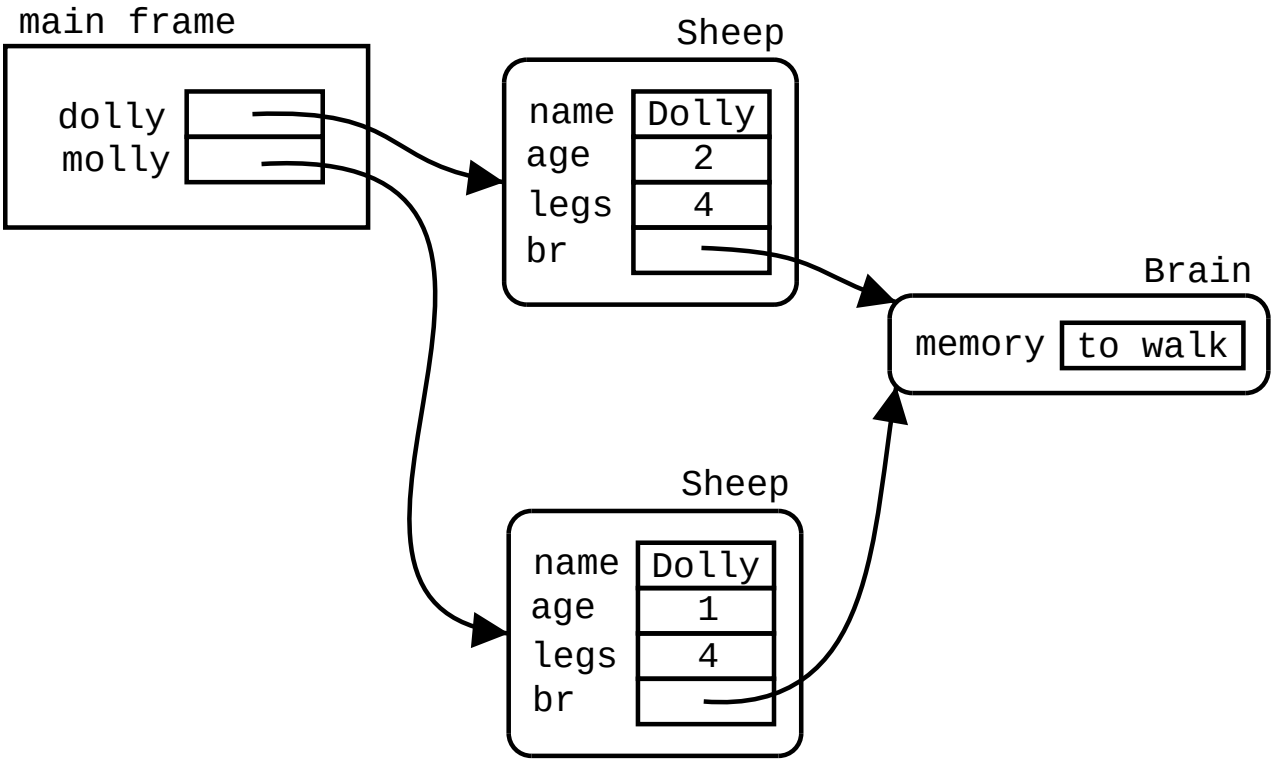
Shallow copy

```
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;
  }
}
```

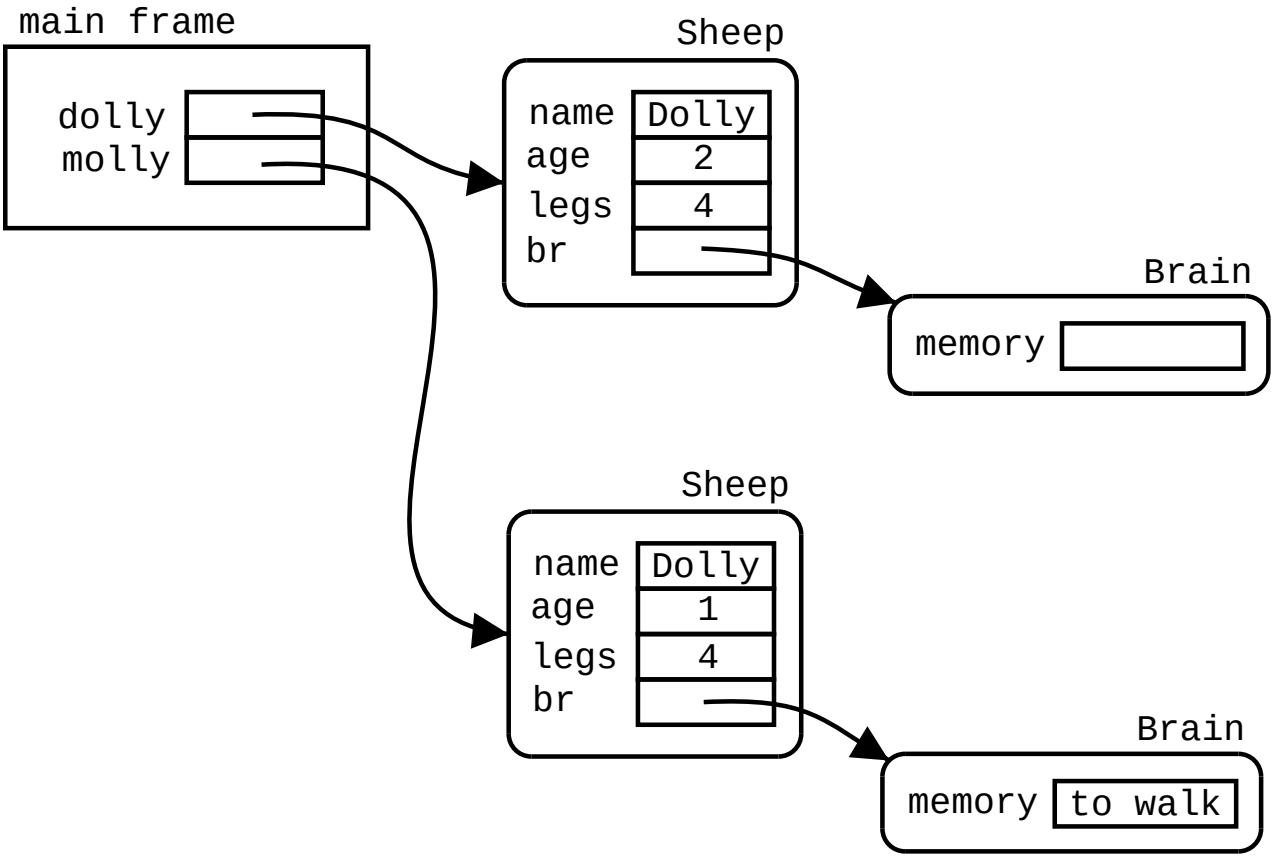
Deep copy

```
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 ”);
  }
}
```

Being of some kind

- The "is a" relationship between an object (or instance) and its class
- So if we have a class

```
class A {
  //...
}
```

- and in some client code we have

```
A x;
x = new A();
```

- Then `x` is an `A`.
- The variable `x` is of type `A`
- The value of `x` is an object of type `A`
- The object referred to by `x` is a kind of `A`.

The end