Reminder

- Assignments are INDIVIDUAL
- If you need help, ask the instructor or the TAs

Defining characteristics of OOP

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

Encapsulation and visibility

- Abstraction and visibility
- Purpose of encapsulation:
  - Hiding the state of an object, (part of) the structure of an object (attributes and/or methods,) or the internal representation of data, so that a client doesn't have to know about the internals of an object (abstraction.)
  - Security: maintaining the integrity of data. Enforcing limited visibility so that clients cannot "corrupt" the state of an object, so that only the class of the object can change the object's state.
- Visibility modifiers (for attributes and methods): `public`, `private` and `protected`.
- Visibility modifiers are orthogonal (independent) of whether the attribute or method is static or not. So they can be combined in any way.

Encapsulation to enforce integrity

```
public class BankAccount
{
  public double balance;
  public BankAccount() { balance = 0.0; }
  public void deposit(double amount)
  {
    if (amount > 0.0)
      balance = balance + amount;
  }
  public void widthdraw(double amount)
  {
    if (amount <= balance)
      balance = balance - amount;
  }
}
```

Encapsulation to enforce integrity

```
public class BankingApplication
{
  public static void main(String[] args)
  {
    BankAccount b;
    b = new BankAccount();
    b.deposit(500.0);
    b.balance = b.balance - 700.0;   // OK
  }
}
```

Encapsulation to enforce integrity

```
public class BankAccount
{
  private double balance;
  public BankAccount() { balance = 0.0; }
  public void deposit(double amount)
  {
    if (amount > 0.0)
      balance = balance + amount;
  }
  public void widthdraw(double amount)
  {
    if (amount <= balance)
      balance = balance - amount;
  }
}
```

Encapsulation to enforce integrity

```
public class BankingApplication
{
  public static void main(String[] args)
  {
    BankAccount b;
    b = new BankAccount();
    b.deposit(500.0);
    b.balance = b.balance - 700.0;   // ERROR
  }
}
```

Encapsulation to enforce integrity

```
public class BankingApplication
{
  public static void main(String[] args)
  {
    BankAccount b;
    b = new BankAccount();
    b.deposit(500.0);
    b.withdraw(700.0);   // OK
  }
}
```

Privacy is relative

```
public class BankAccount
{
  private double balance;
  public BankAccount() { balance = 0.0; }
  public void deposit(double amount)
  {
    if (amount > 0.0)
      balance = balance + amount;
  }
  public void widthdraw(double amount)
  {
    if (amount <= balance)
      balance = balance - amount;
  }
  public void transfer(BankAccount other)
  {
    this.balance = this.balance + other.balance;
    other.balance = 0.0;
  }
}
```

Privacy is relative

```
public class BankingApplication
{
  public static void main(String[] args)
  {
    BankAccount b1, b2;
    b1 = new BankAccount();
    b2 = new BankAccount();
    b1.deposit(500.0);
    b2.transfer(b1);
  }
}
```

Aggregation

- Analisys:
  - Identify relevant information: objects and types of objects (classe)
  - Identify relationships between objects

- Different kinds of relationships depending on the type of the objects involved

- For example:
  - Numeric relationships:
    - account balance > 0
    - car fuel > 10
    - hitPoints <= maxHitPoints
    - number of heads <= 2
    - number of fingers > 1
    - tax_payable = base + (income - cutoff)*rate
  - Structural relationships:
    - A bank account *has a* balance and an owner
    - A car *has an* engine
    - A person *has a* name and a head

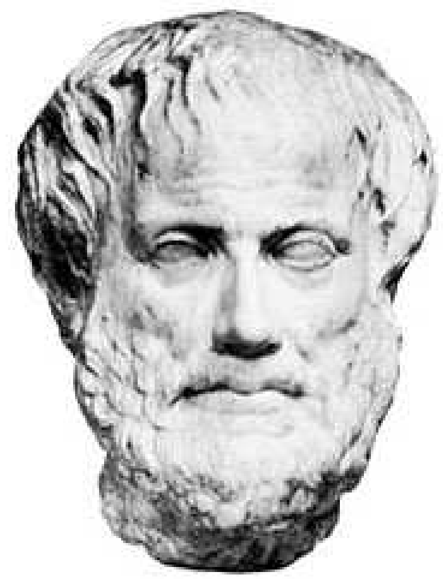
Objects and Aggregation

- Objects are data with structure: objects have attributes.
- We think of attributes as characteristics of objects in a class.
- The relation between an object and its attributes can be seen as a “has a” relationship.
- Aggregation is the composition of objects in different parts or *aggregates* (the attributes.)

Objects and Aggregation

```
public class Engine {
    // ...
}
public class Car {
    Engine engine;
    // ...
}
public class StreetSimulation {
    void p()
    {
        Car mycar = new Car();
        mycar.engine = new Engine();
    }
}
```

Aristotle



Aristotle

- Silogisms:
  - **If** every city *has a* mayor, **and** Edinburgh *is a* city, **then** Edinburgh *has a* mayor.
  - **If** every car *has an* engine, **and** this *is a* car, **then** this *has an* engine.
  - **If** every A *has a* B, **and** x *is an* A, **then** x *has a* B.
- In OOP:
  - **If** every object of type A *has an* attribute of type B **and** x *is an* A object **then** x *has an* attribute of type B.
  - **If** a class A *has an* attribute of class B, **and** x *is an* instance of A, **then** x *has an* attribute of class B.

Objects and Aggregation

- Aggregation is given by the “has a” relationship.

```
public class A {
    B u;
    // ...
}
public class C {
    void m()
    {
        A x = new A();
        ... x.u ...
    }
}
```

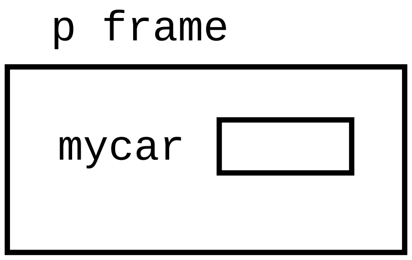
Objects and Aggregation

```
public class Mayor {
    // ...
}
public class City {
    Mayor mayor;
    // ...
}
public class Something {
    void p()
    {
        City edinburgh = new City();
        edinburgh.mayor = new Mayor();
    }
}
```

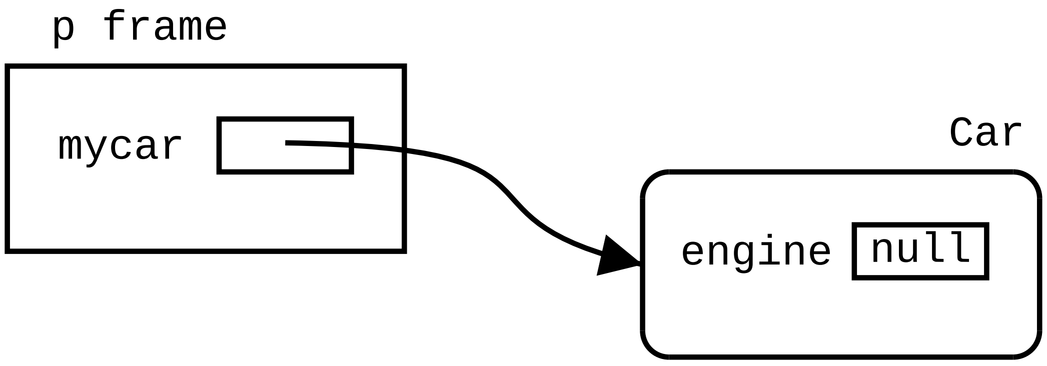
Objects and Aggregation

```
public class Engine {
    // ...
}
public class Car {
    Engine engine;
    // ...
}
public class Something {
    void p()
    {
        Car mycar = new Car();
        mycar.engine = new Engine();
    }
}
```

Objects and Aggregation

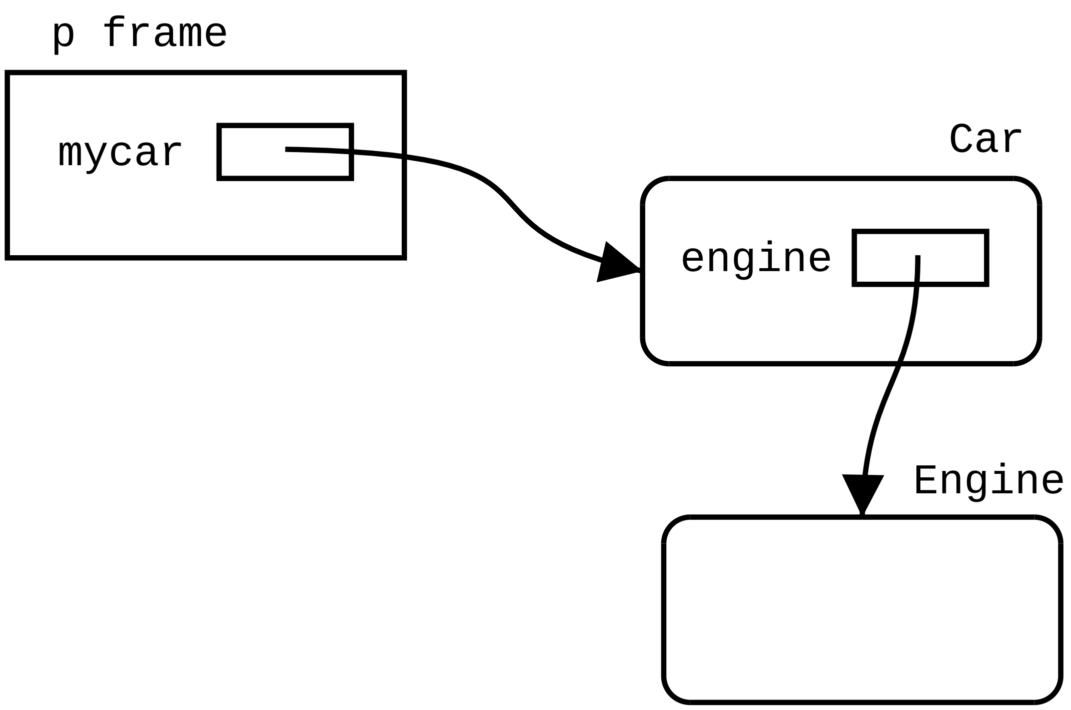


Objects and Aggregation

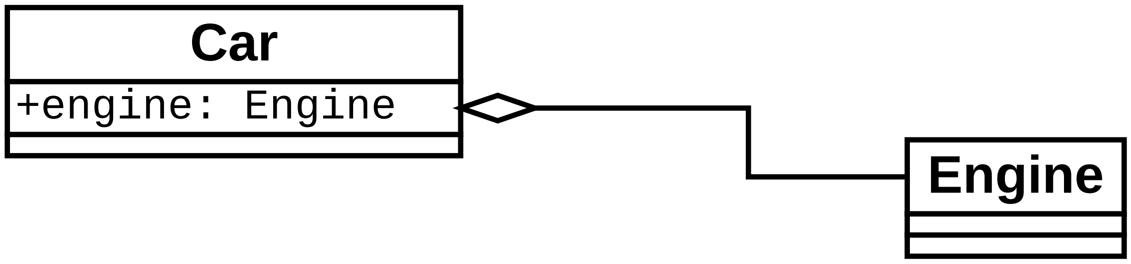


McGill

Objects and Aggregation



Objects and Aggregation



Example

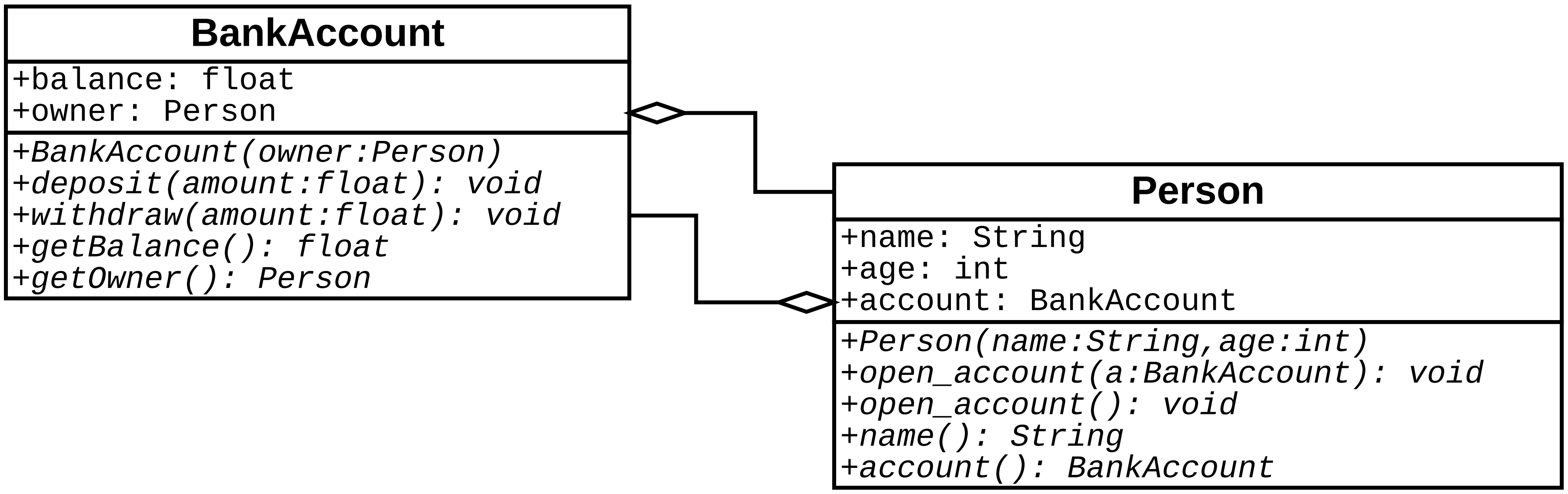
```
public class Engine
{
    private boolean on;
    private double rpm;

    public Engine()
    {
        on = false;
        rpm = 0.0;
    }
    public void turn_on()
    {
        on = true;
        rpm = 50.0;
    }
    public void accelerate()
    {
        rpm = rpm + 10.0;
    }
    public void decelerate()
    {
        rpm = rpm - 10.0;
    }
    public double get_rpm()
    {
        return rpm;
    }
}
```

Example (contd.)

```
public class Car
{
    private Engine engine;
    private double speed;

    public Car()
    {
        engine = new Engine();
        speed = 0.0;
    }
    public void turn_on()
    {
        engine.turn_on();
    }
    public void acelerate()
    {
        engine.acelerate();
        speed = speed + 10 * engine.get_rpm();
    }
}
```

Mutual references

- Mutual reference: An object A can have a reference to an object B which has a reference to A



Mutual reference

```
public class BankAccount
{
    private float balance;
    private Person owner;

    public BankAccount(Person owner)
    {
        this.owner = owner;
        balance = 0.0;
    }
    public void deposit(float amount)
    {
        balance = balance + amount;
    }
    public void withdraw(float amount)
    {
        if (amount <= balance)
            balance = balance - amount;
    }
    public float balance() { return balance; }
    public Person owner()  { return owner; }
}
```

Mutual reference

```
public class Person
{
    private String name;
    private int age;
    private BankAccount account;

    public Person(String name, int age)
    {
        this.name = name;
        this.age = age;
        account = null;
    }
    public void open_account(BankAccount a)
    {
        account = a;
    }
    public void open_account()
    {
        account = new BankAccount(this);
    }
    // Continues below...
    public String name()
    {
        return name;
    }
    public BankAccount account()
    {
        return account;
    }
}
```

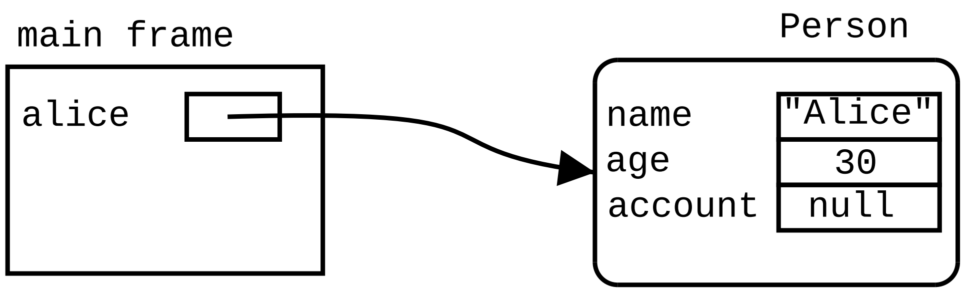
Mutual reference (contd.)

```
public class Banking
{
  public static void main(String[] args)
  {
    Person alice = new Person(“Alice”, 30);
    BankAccount a = new BankAccount(alice);
    alice.open_account(a);

    Person bob = new Person(“Bob”, 29);
    bob.open_account();

    BankAccount b = bob.account();
    b.deposit(300.0);

    alice.account().deposit(200.0f);

    System.out.println(b.balance());
    System.out.println(alice.account().balance());
    System.out.println(a.balance());
  }
}
```

Mutual reference (contd.)

```
public class Banking
{
  public static void main(String[] args)
  {
    Person alice = new Person(‘‘Alice’’, 30);
    BankAccount a = new BankAccount(alice);
    alice.open_account(a);

    Person bob = new Person(‘‘Bob’’, 29);
    bob.open_account();

    BankAccount b = bob.account();
    b.deposit(300.0);

    alice.account().deposit(200.0f);

    System.out.println(b.balance());
    System.out.println(alice.account().balance());
    System.out.println(a.balance());
  }
}
```

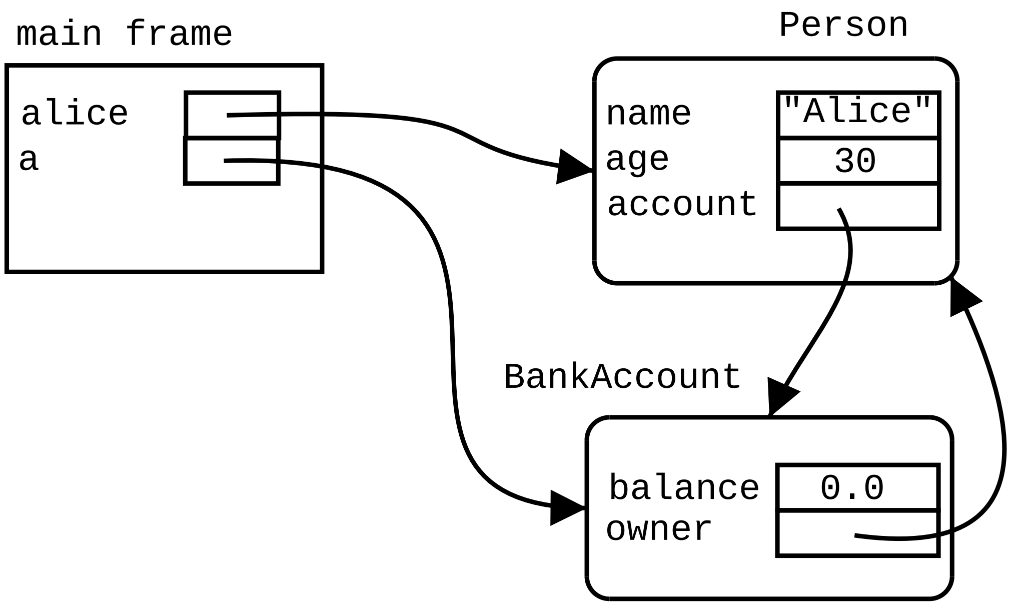
Mutual reference



Mutual reference (contd.)

```
public class Banking
{
  public static void main(String[] args)
  {
    Person alice = new Person(‘‘Alice’’, 30);
    BankAccount a = new BankAccount(alice);
    alice.open_account(a);

    Person bob = new Person(‘‘Bob’’, 29);
    bob.open_account();

    BankAccount b = bob.account();
    b.deposit(300.0);

    alice.account().deposit(200.0f);

    System.out.println(b.balance());
    System.out.println(alice.account().balance());
    System.out.println(a.balance());
  }
}
```

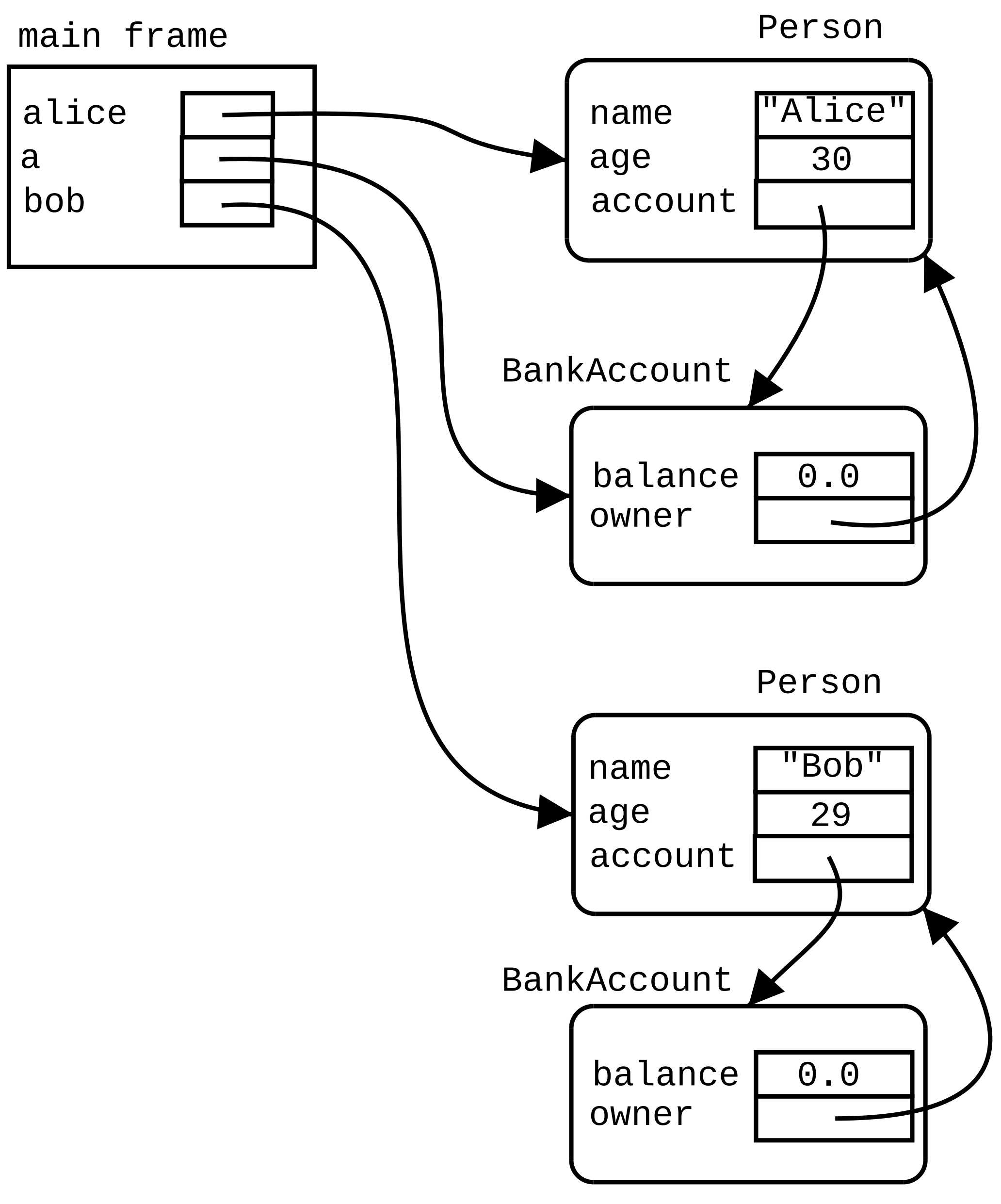
Mutual reference



Mutual reference (contd.)

```
public class Banking
{
  public static void main(String[] args)
  {
    Person alice = new Person(“Alice”, 30);
    BankAccount a = new BankAccount(alice);
    alice.open_account(a);

    Person bob = new Person(“Bob”, 29);
    bob.open_account();

    BankAccount b = bob.account();
    b.deposit(300.0);

    alice.account().deposit(200.0f);

    System.out.println(b.balance());
    System.out.println(alice.account().balance());
    System.out.println(a.balance());
  }
}
```

Mutual reference



Mutual reference

- Mutual references between objects of the same class:

```
public class Person {
    private String name;
    private Person spouse;
    public Person(String name, int age)
    {
        this.name = name;
        this.age  = age;
        this.spouse = null;
    }
    public void marry(Person someone)
    {
        this.spouse = someone;
        someone.spouse = this;
    }
    public String name()   { return name; }
    public Person spouse() { return spouse; }
}
```

Mutual reference

```
public class Marriage
{
    public static void main(String[] args)
    {
        Person a = new Person(“Alice”, 30);
        Person b = new Person(“Bob”, 29);
        a.marry(b);
        System.out.println(a.name());
        System.out.println(a.spouse().name());
        System.out.println(b.name());
        System.out.println(b.spouse().name());
    }
}
```

Aliases and shared references

- Variables and values
- If we execute:

```
x = 5;
```

- then the value of `x` is 5.
- Strictly speaking `x` is not 5; `x` is a memory location.
- So while we would informally read `x==5` as "`x` is 5", the actual meanning is the value of `x` is 5.
- Hence, after executing

```
x = 5;
y = 5;
```

  and both `x` and `y` have the same value, but they are not the same variable.

Variables and values

- For primitive data types (int, boolean, float, String, etc.)

  ```
  x = y;
  ```

  means copy the value of y in the memory location of x;

- So

  ```
  int x, y;
  x = 4;
  y = x;
  ```

  means that both `x` and `y` have value 4, but they have a separate identity because each of them is a different memory location...

x	4
y	4

Variables and values

- So the value of y is the same as the value of x, but y is not the same as x.
- ... which implies that their values are independent:

```
int x, y;
x = 4;
y = x;
x++;
// x == 5 and y == 4
```

- Variables can be changed over time by assignment.
- If x and y are two variables of a primitive data type, we say that they are equal if their values are the same.
- We can test for whether the values of two variables are the same using the == operator.

Being the “same” as something else

- 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.

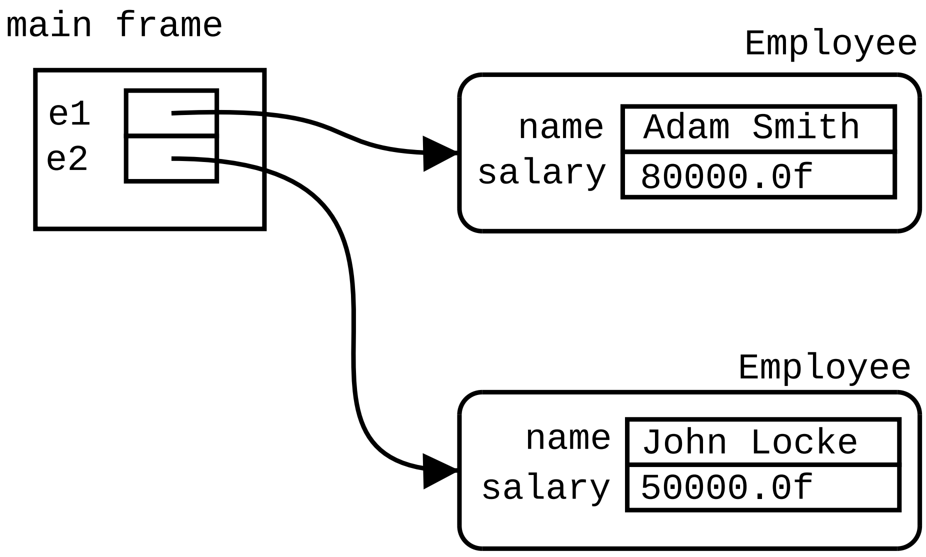
Example:

```
class Employee
{
    String name;
    float salary;
    Employee(String name, float salary)
    {
        this.name = name;
        this.salary = salary;
    }
    String name() { return name; }
    float salary() { return salary; }
    void raise_salary(float percentage)
    {
        salary = salary * (1 + percentage/100.0f);
    }
}
```

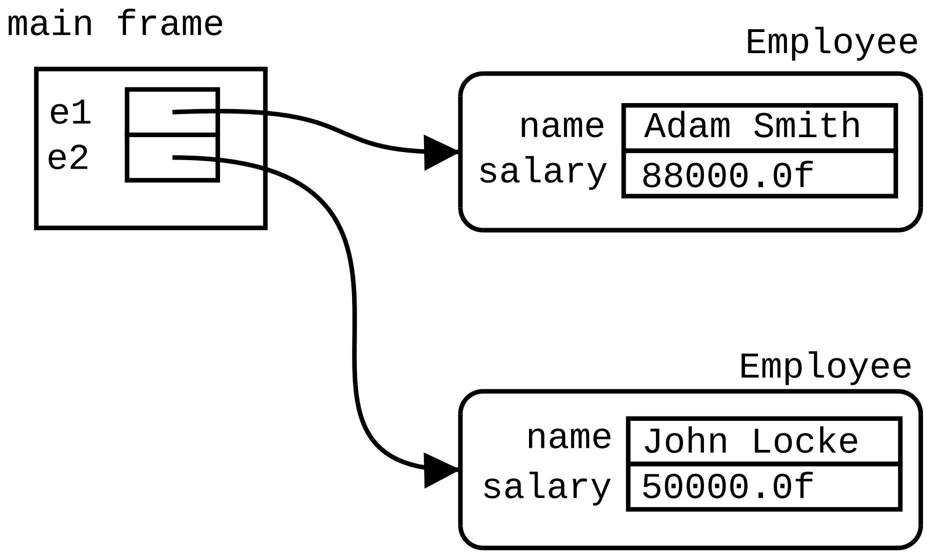
Example (contd.)

```
public class Test
{
  public static void main(String[] args)
  {
    Employee e1 = new Employee(“Adam Smith”, 80000
    Employee e2 = new Employee(“John Locke”, 50000
    e1.raise_salary(10f);
    System.out.println(e2.salary());
  }
}
```

Example (contd.)



Example (contd.)



Alias

- A variable is an alias of another variable if they both point to the same object.

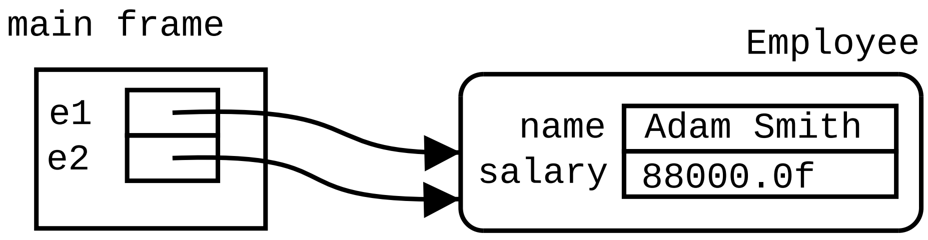
```
A x, y;
x = new A();
y = x;
```

- In this case x and y are the “same”.
- More precisely, the values of x and y are the same reference (pointer,) and therefore they refer to the same object.

Example (contd.)

```
public class Test
{
  public static void main(String[] args)
  {
    Employee e1 = new Employee(‘‘Adam Smith’’, 80000
    Employee e2 = e1;
    e1.raise_salary(10f);
    System.out.println(e2.salary());
  }
}
```

Example (contd.)



Aliases

- Compare Test with

```
int x1, x2;
x1 = 6;
x2 = x1;
x1 = x1 * 3;
```

- If two variables are aliases, whatever one does to either of them, affects the other, because they refer to the same object.

Shared references

```
public class BankAccount
{
  private float balance;
  public BankAccount(float b) { balance = b; }
  public void deposit(float amount)
  {
    balance = balance + amount;
  }
  public void withdraw(float amount)
  {
    if (balance >= amount)
      balance = balance - amount;
  }
  public float balance() { return balance; }
}
```

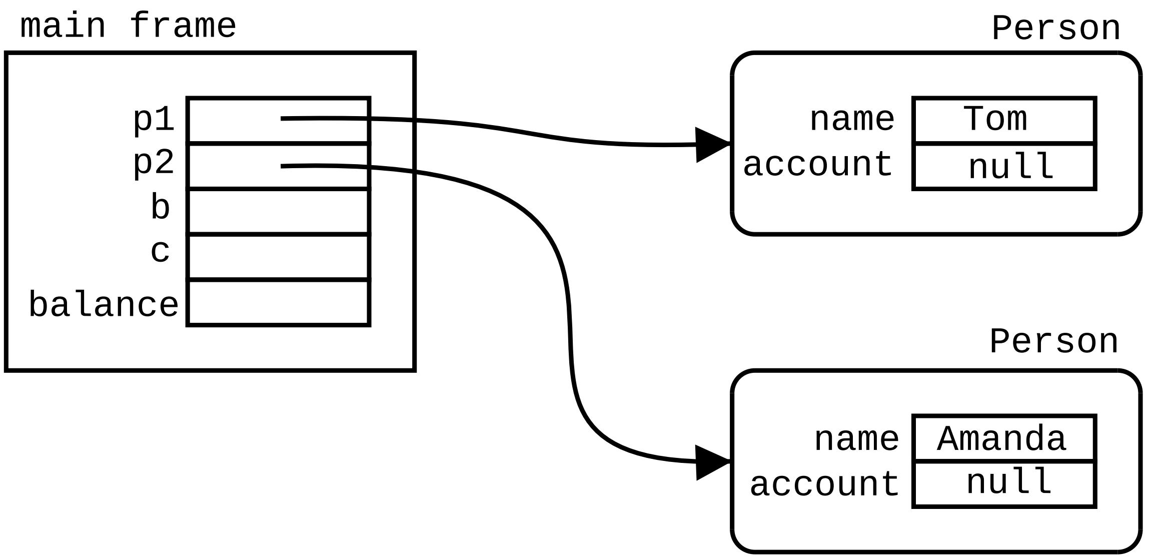
Shared references

```
public class Person
{
  private String name;
  private BankAccount account;
  public Person(String name) { this.name = name; }
  public void set_account(BankAccount a)
  {
    account = a;
  }
  public String name() { return name; }
  public BankAccount account() { return account; }
}
```

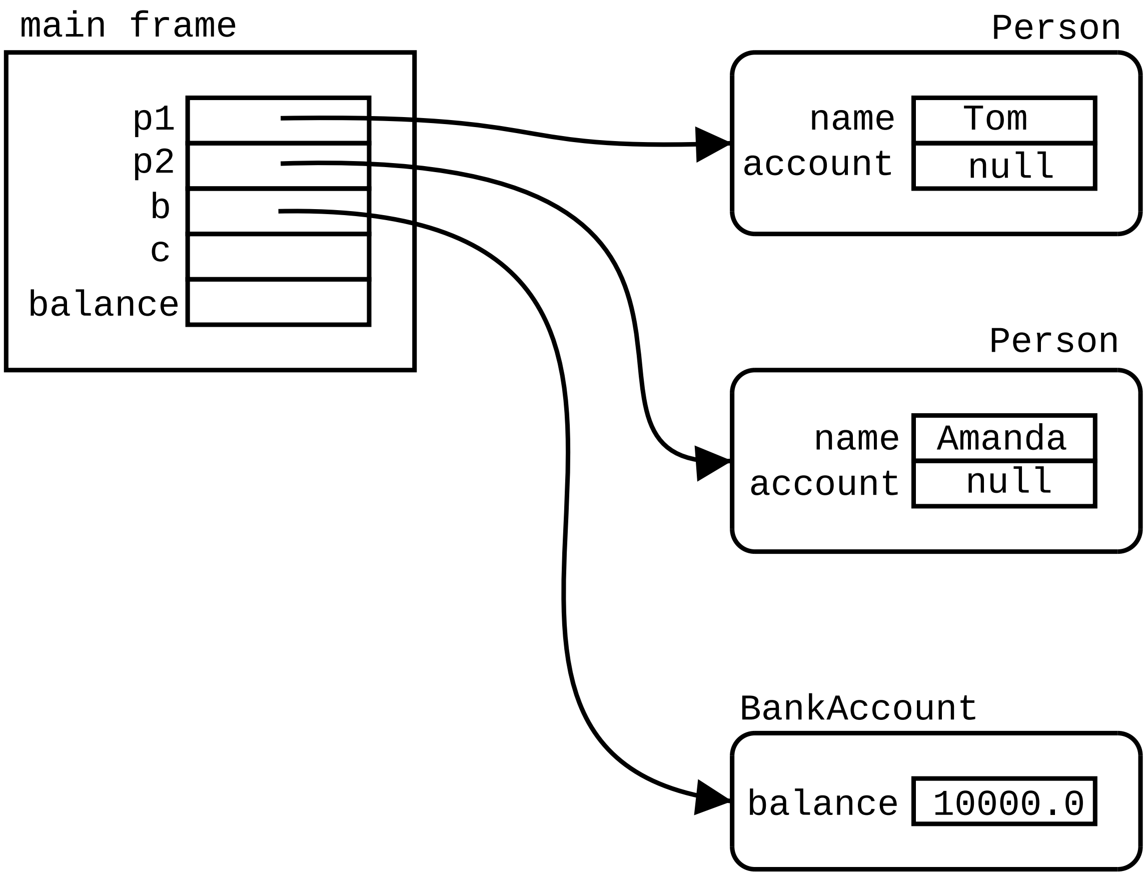
Shared references

```
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.set_account(b);
    p2.set_account(b);

    b.withdraw(500.0f);
    BankAccount c = p2.account();
    float balance = c.balance();
    System.out.println(balance);
  }
}
```

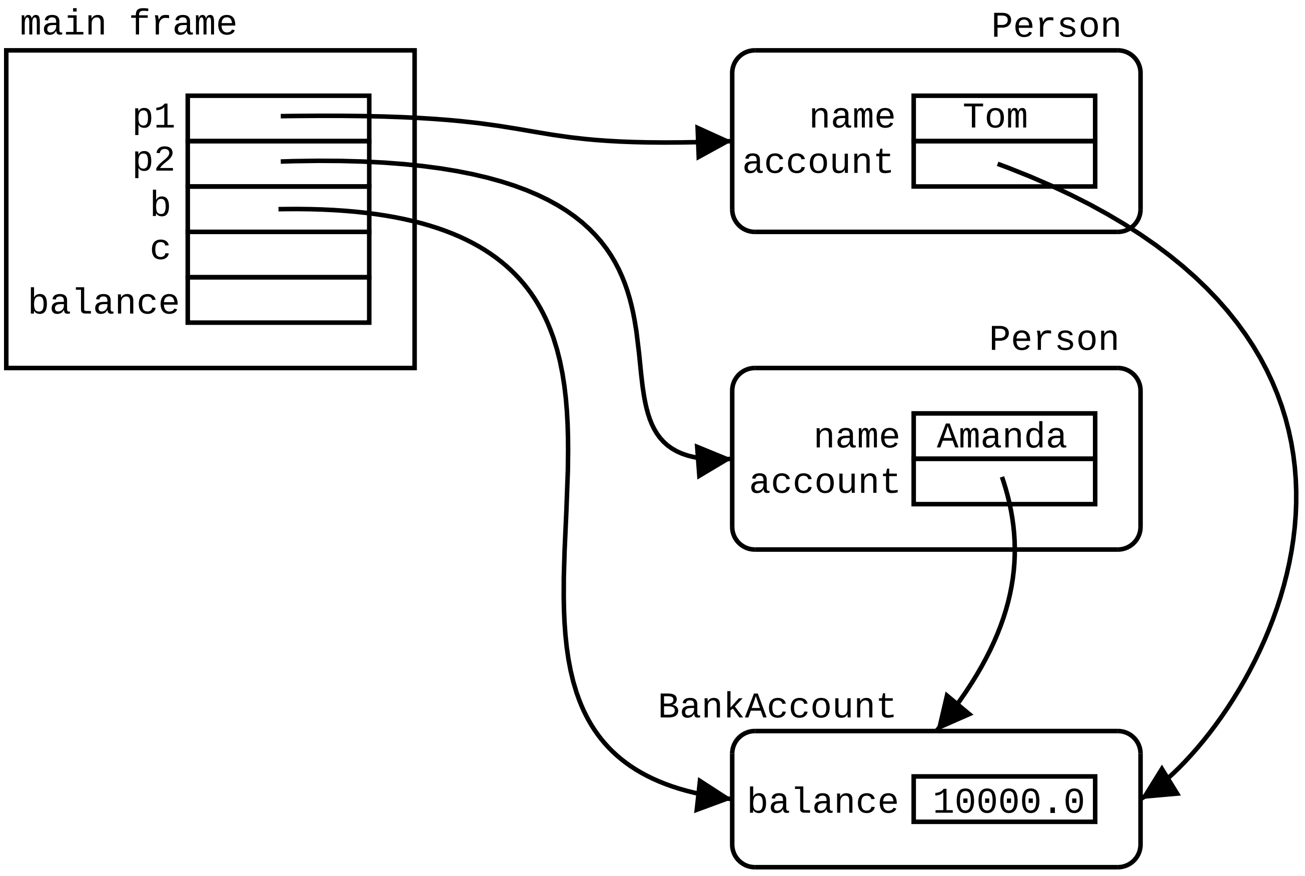
Shared references



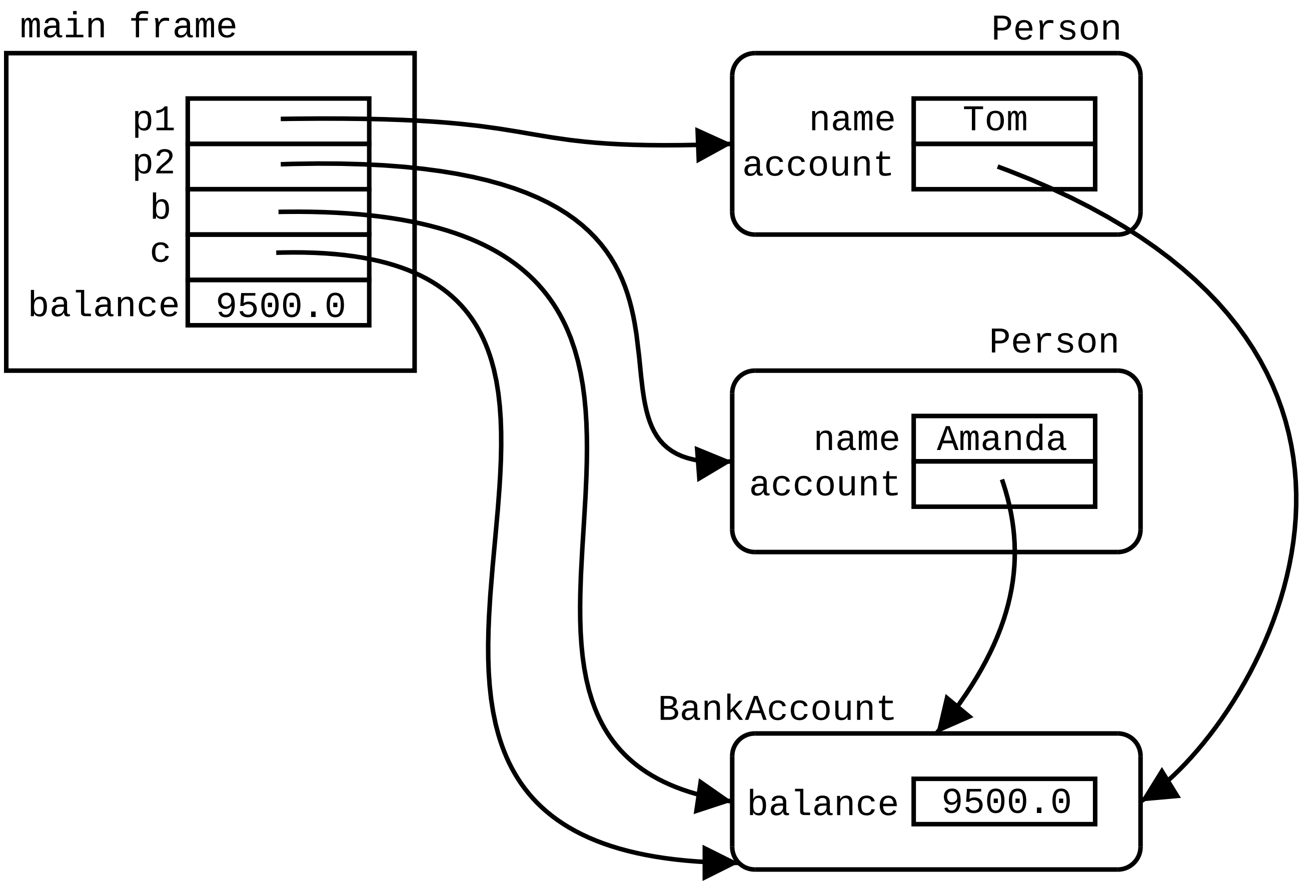
Shared references



Shared references



Shared references



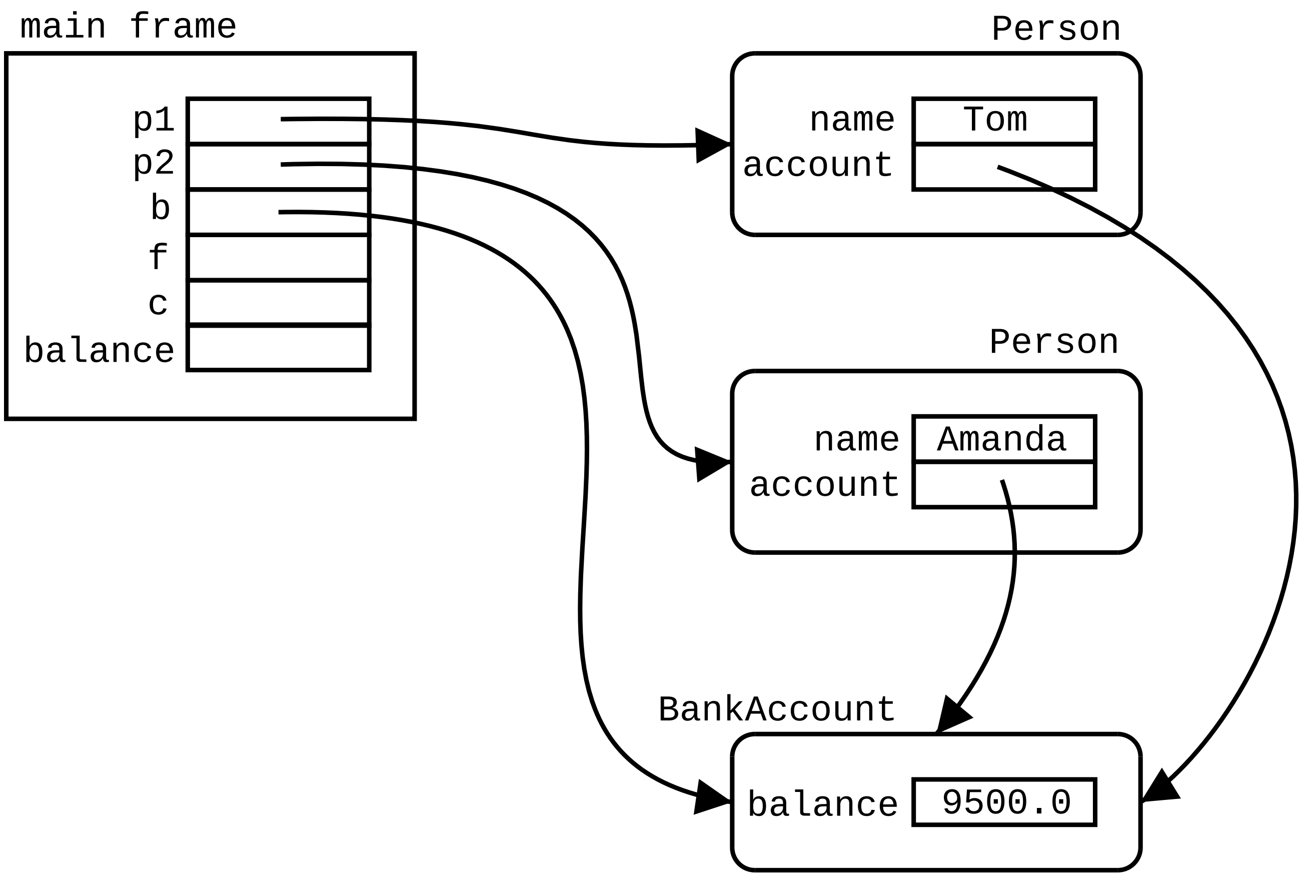
Shared references vs static variables

- 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 declares 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”.

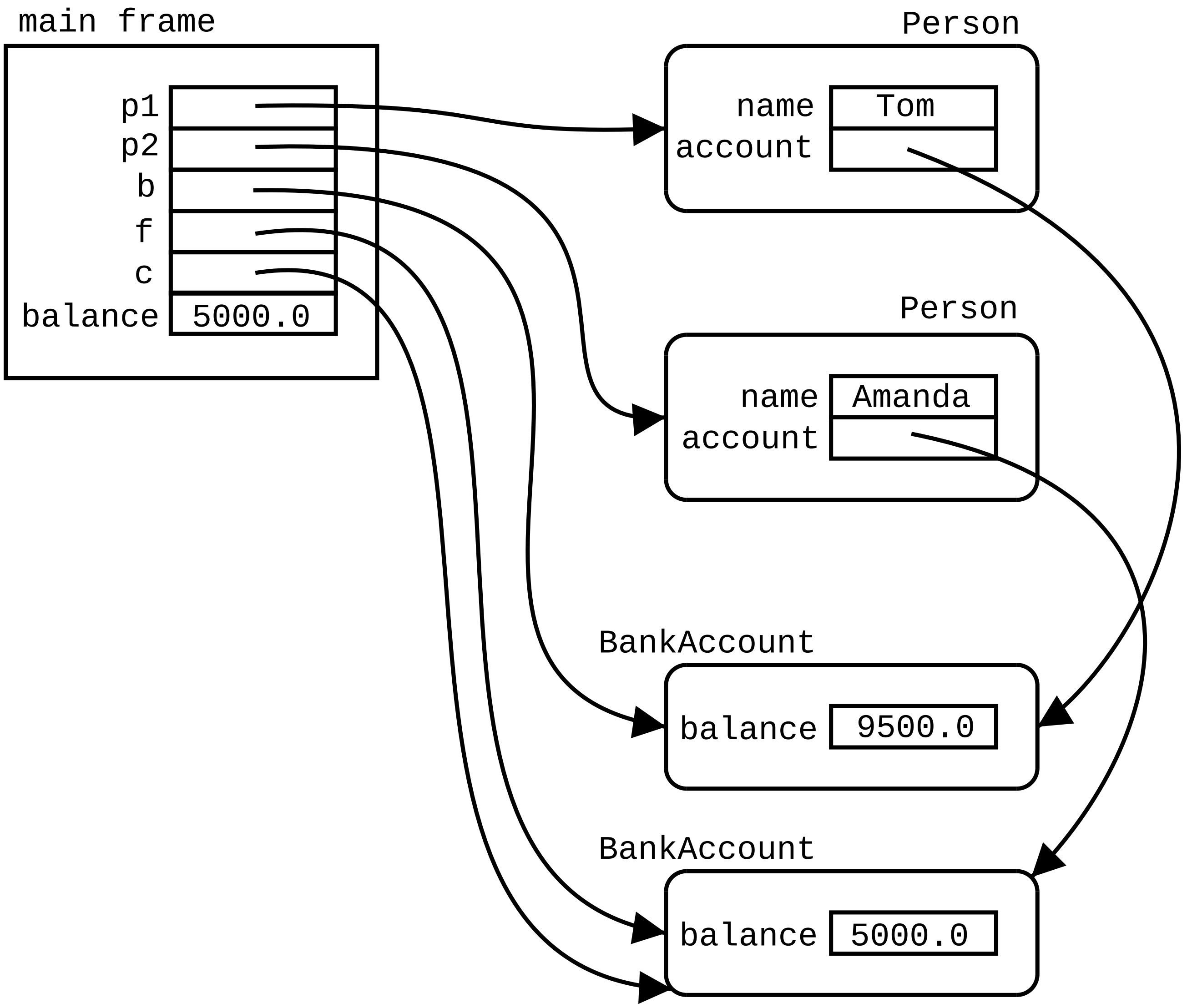
Shared references vs static variables

```
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.set_account(b);
    p2.set_account(b);
    b.withdraw(500.0f);
    BankAccount f = new BankAccount(5000.0f);
    p2.set_account(f);
    BankAccount c = p2.account();
    float balance = c.balance();
    System.out.println(balance);
  }
}
```

Shared references vs static variables



Shared references vs static variables



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.

McGill

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.set_account(b);
    p2.set_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”);
  }
}
```

Being equal to something

- 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 {
    String x, y;
    A(String x, String y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Being equal to something

- and there is some client with

```
A a1 = new A(“hello”, “bye”);
A a2 = new A(“hello”, “bye”);
A a3 = new A(“bonjour”, “bye”);
```

- 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 {
    String x, y;
    A(String x, String 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("hello", "bye");
    A a2 = new A("hello", "bye");
    A a3 = new A("bonjour", "bye");
    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.

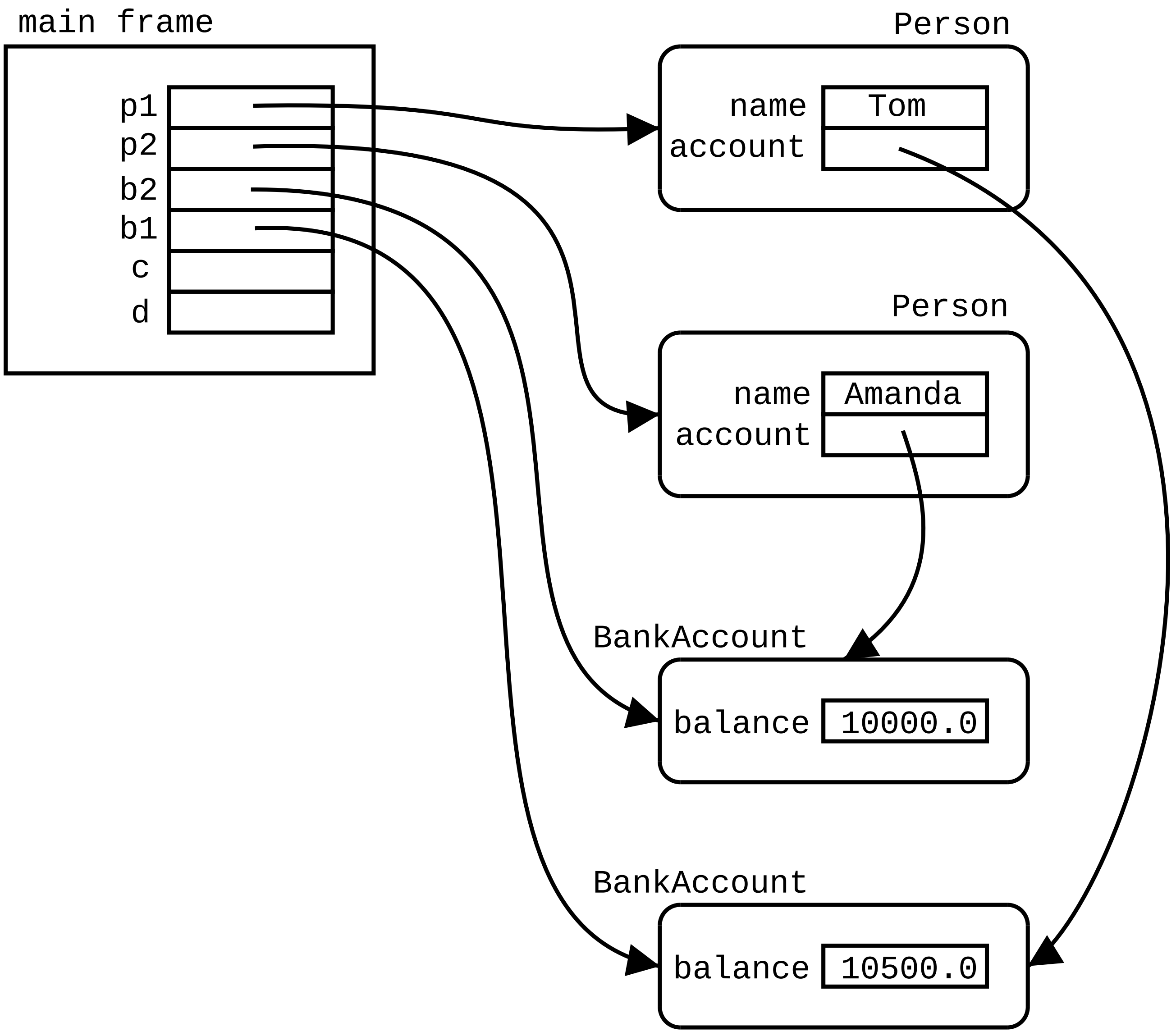
Example

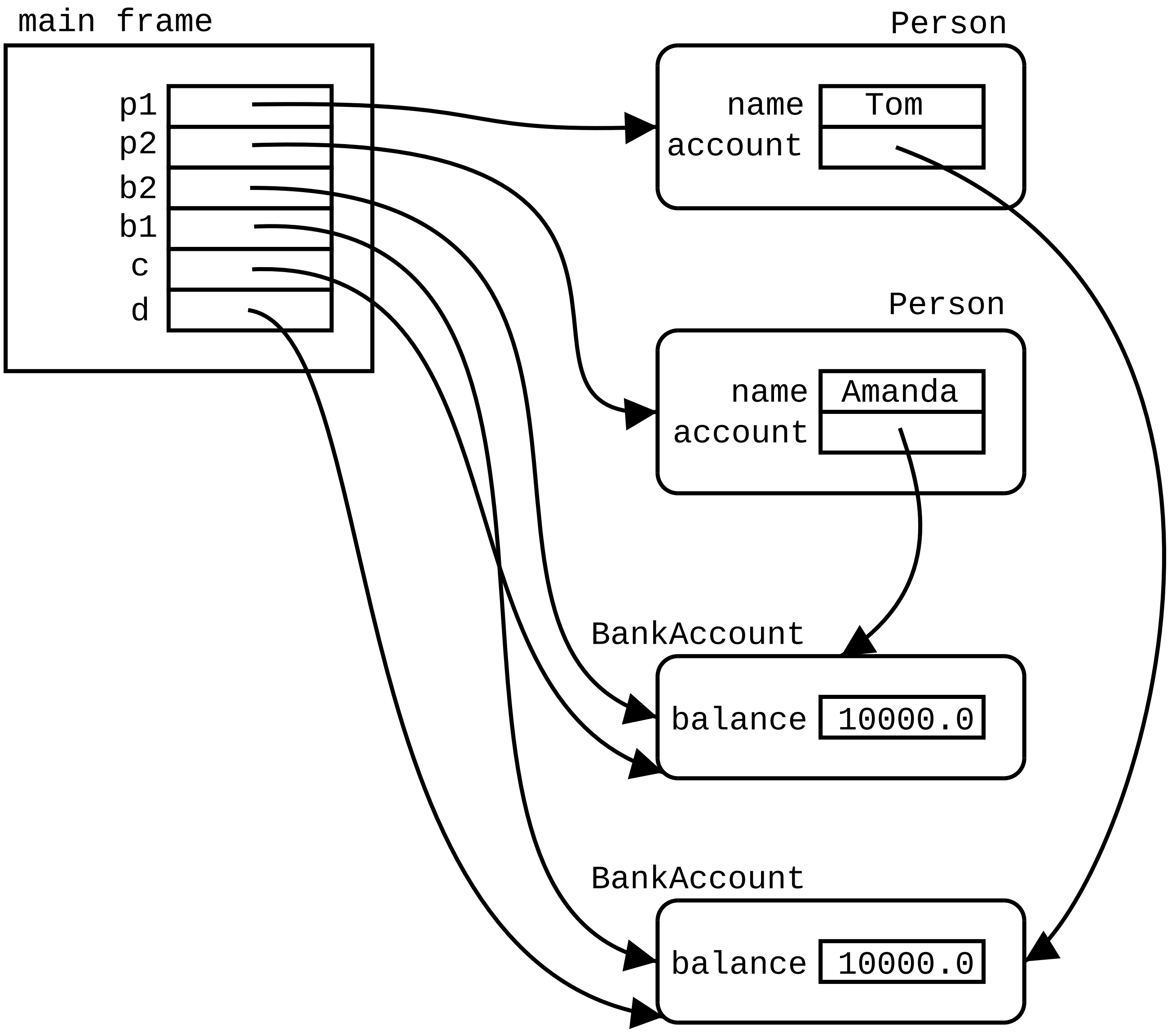
```
public class BankAccount {
  private float balance;
  // ... same as before
  public boolean equals(BankAccount other_account)
  {
    return this.balance == other_account.balance;
  }
}
```

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.set_account(b1);
    p2.set_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


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

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