Topic 3.0: Objects

Learning Goals (Week 3):

- **References to Objects**
- **The clone() method**
- **Arrays of Objects**
- **Objects as Method Parameter**
- Understand Java Garbage Collection
- Objects in Objects: Safe Creation
- Objects in Objects: Safe Method Use
- Understand Compartmentalization & Encapsulation Features

References to objects

- Every type **except double, float, long, int, short, byte, char,** or **boolean** is an Object
- This includes
  - String
  - all arrays
  - your own classes
  - any pre-supplied classes like Scanner or ArrayList
- Any variable with one of these types stores **a reference to an object, never the object itself**

Cloning objects

- To make a completely new object, identical to an existing one, you need to write a method
  - This is traditionally named **clone( )**
- Consider our **Person** class from before (with a **String name** and **int age**)
- A clone( ) method for the Person class:

```
public Person clone( ) {
 return new Person(name, age);
}
```

we want to return a **Person** object that is a clone of the current object

Cloning objects

```
public Person clone( ) {
   return new Person(name, age);
}
```

- Simpler looking method than writing the **clone()** :

```
public Person clone( ) {
   Person newPerson = new Person();
   newPerson.name = this.name;
   newPerson.age = this.age;
   return newPerson;
}
```

Cloning objects

```
public Person clone( ) {
   return new Person(name, age);
}
```

- or in **main()**

```
public static Person clone(Person p ) {
   Person newPerson = new Person();
   newPerson.setName(p.getName());
   newPerson.setAge(p.getAge());
   return newPerson;
}
```

So Cloning...

- Last week we talked Shallow vs Deep Copy of Arrays
- Today, **simple assignment (shallow copy) of Objects**
    - also known as **aliasing**

```
Person one, two;
 one = new Person("Fred", 29);
 two = one;
```

- Just like arrays, we only copy the reference. Changing one affects both.

So Cloning...

- A **clone (deep copy)** gives two independent objects

```
Person one, two;
 one = new Person("Fred", 29);
 two = one.clone();
```

- A change to one will not affect the other
- **This is not an issue with String objects (or other "immutable" objects because they can't be changed) (stay tuned for an example)**
- Neither one is right or wrong, depends on what you need: use the one that does what you want it to do. **Make sure you know**

Cloning Objects: Wild and Wacky Stuff

- What if we did:

```
Person one, two;
one = new Person("Anik", 29);
two = one.clone();
```

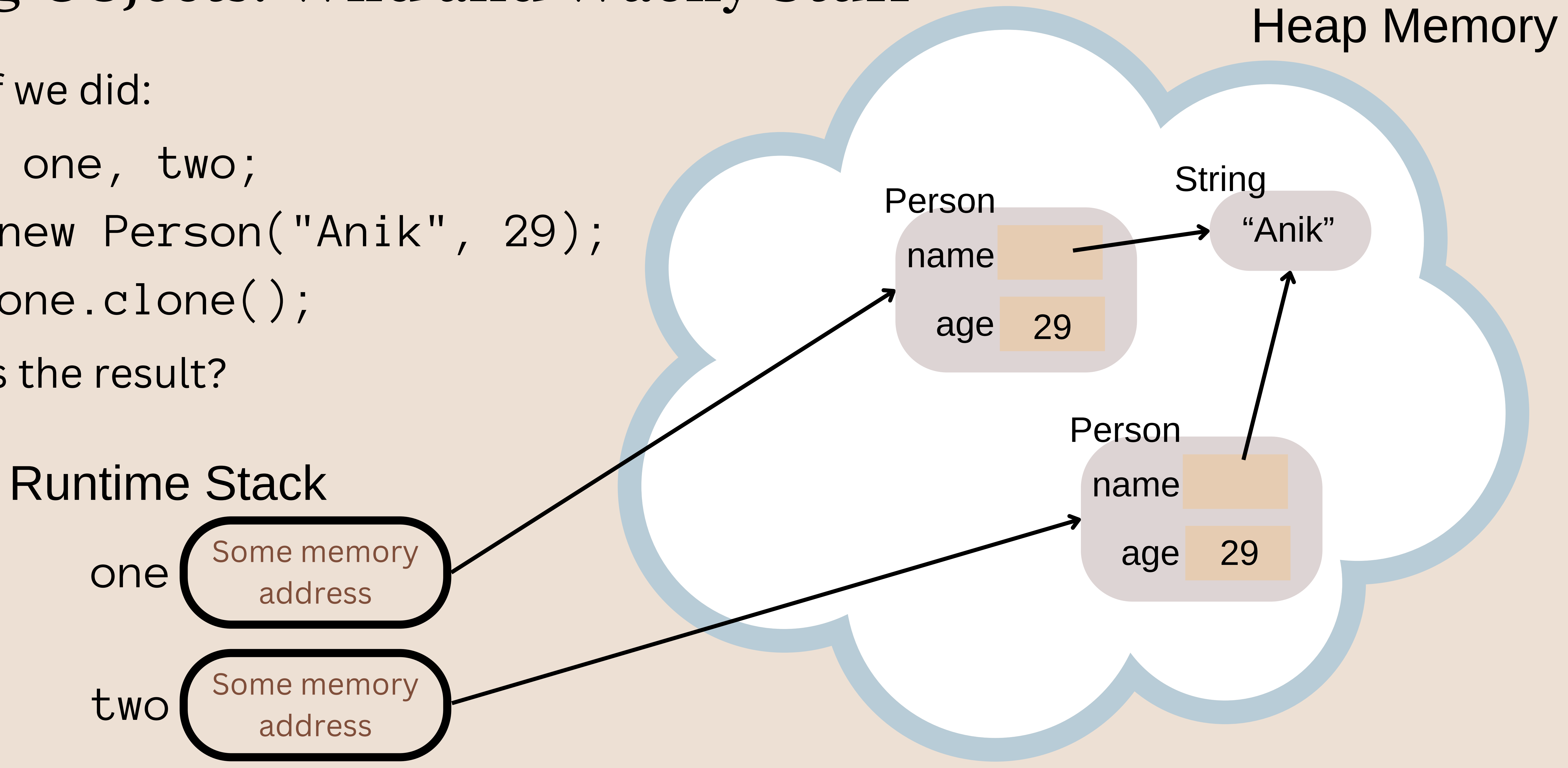
- What is the result?

Cloning Objects: Wild and Wacky Stuff

- What if we did:

```
Person one, two;
one = new Person("Anik", 29);
two = one.clone();
```

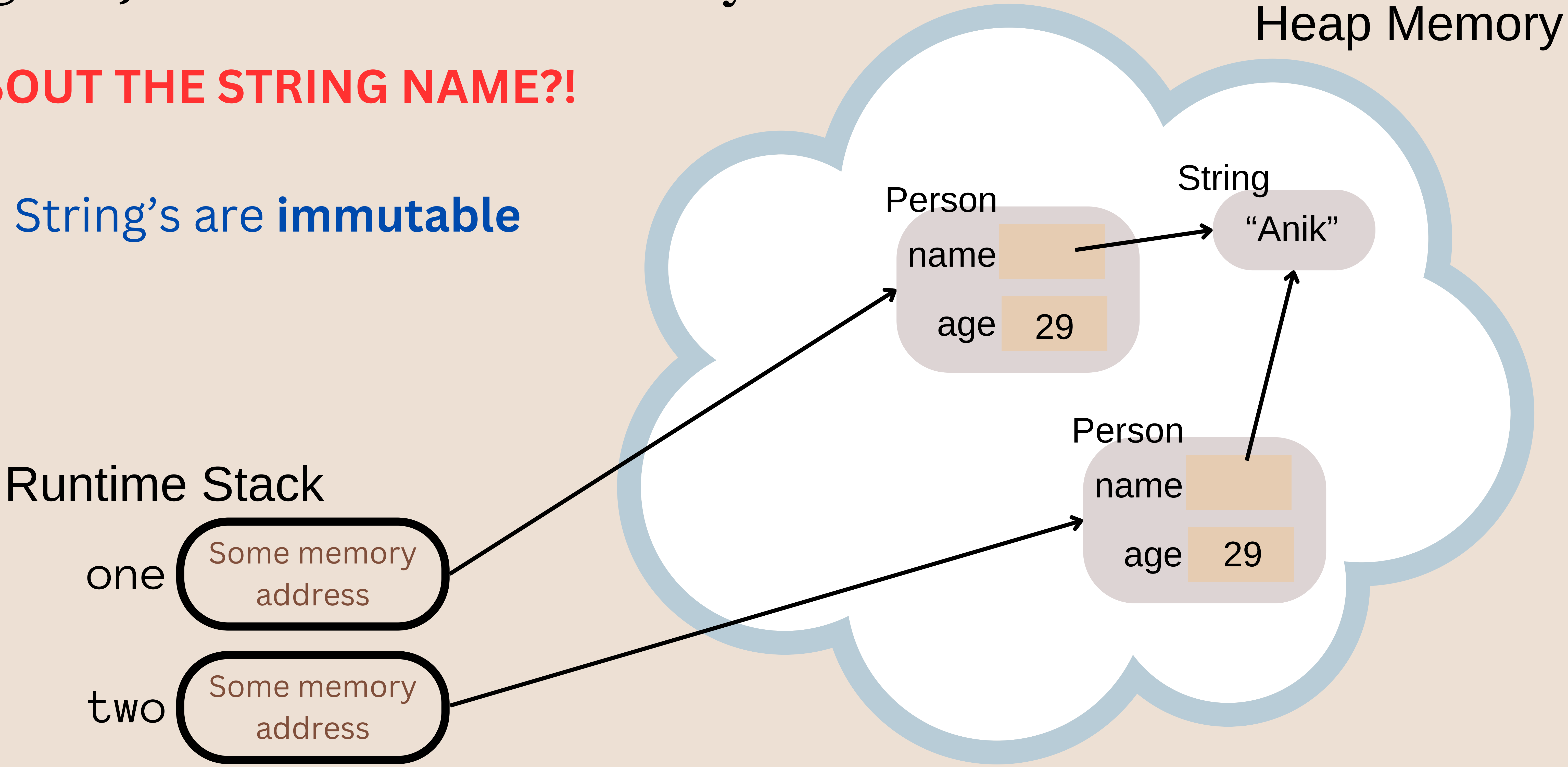
- What is the result?



Cloning Objects: Wild and Wacky Stuff

WHAT ABOUT THE STRING NAME?!

oh yeah… String's are **immutable**



Cloning Arrays: A reminder from last week

```
int[] a1 = {4,1,7};
int[] a2;
a2 = a1; // shallow

// manual deep
a2 = new int[a1.length];
for(int i=0; i<a1.length; i++)
a2[i] = a1[i];
```

```
// shortcut deep
a2 = new int[a1.length];
 System.arraycopy(a1, 0, a2, 0, a1.length);

 /* a1 and a2 must be references to existing
 * arrays, the 0's are the desired starting
 * positions, and the last parameter is the
 * number of elements to be copied. */
```

Arrays of Objects

- If we have an **array of objects**, then we have a **reference to an array of other references!**
- Now a true **"deep copy"** should make clones at two different levels!

Arrays of Objects

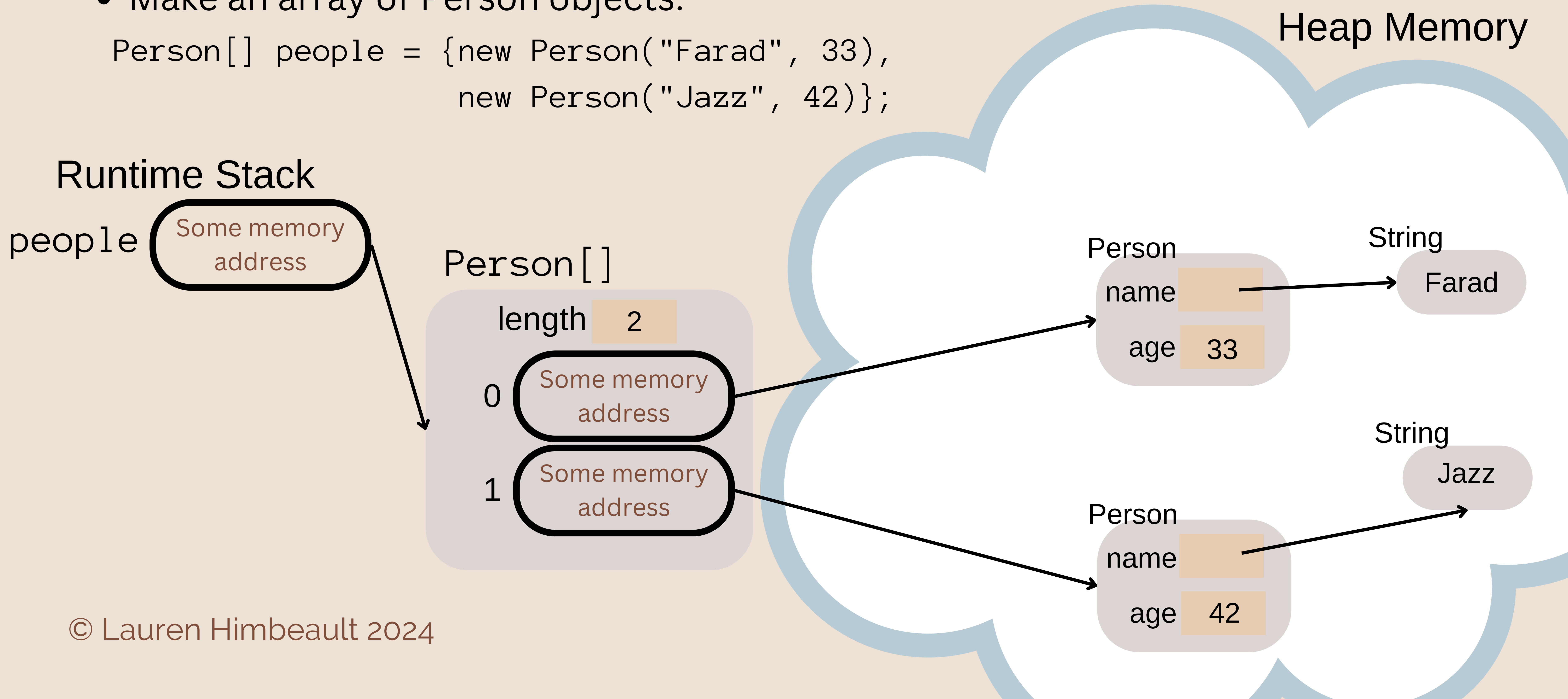
- If we have an **array of objects**, then we have a **reference to an array of other references!**
- Now a true **"deep copy"** should make clones at two different levels!

- Then what about a array of objects that contain references to other objects which contain arrays...?
  - The principles are the same
  - If every level in this situation does something correct and sensible, then the whole thing will work reliably
  -
- **Think! Plan on paper before implementing!**

Consider the following:

- Make an array of Person objects:

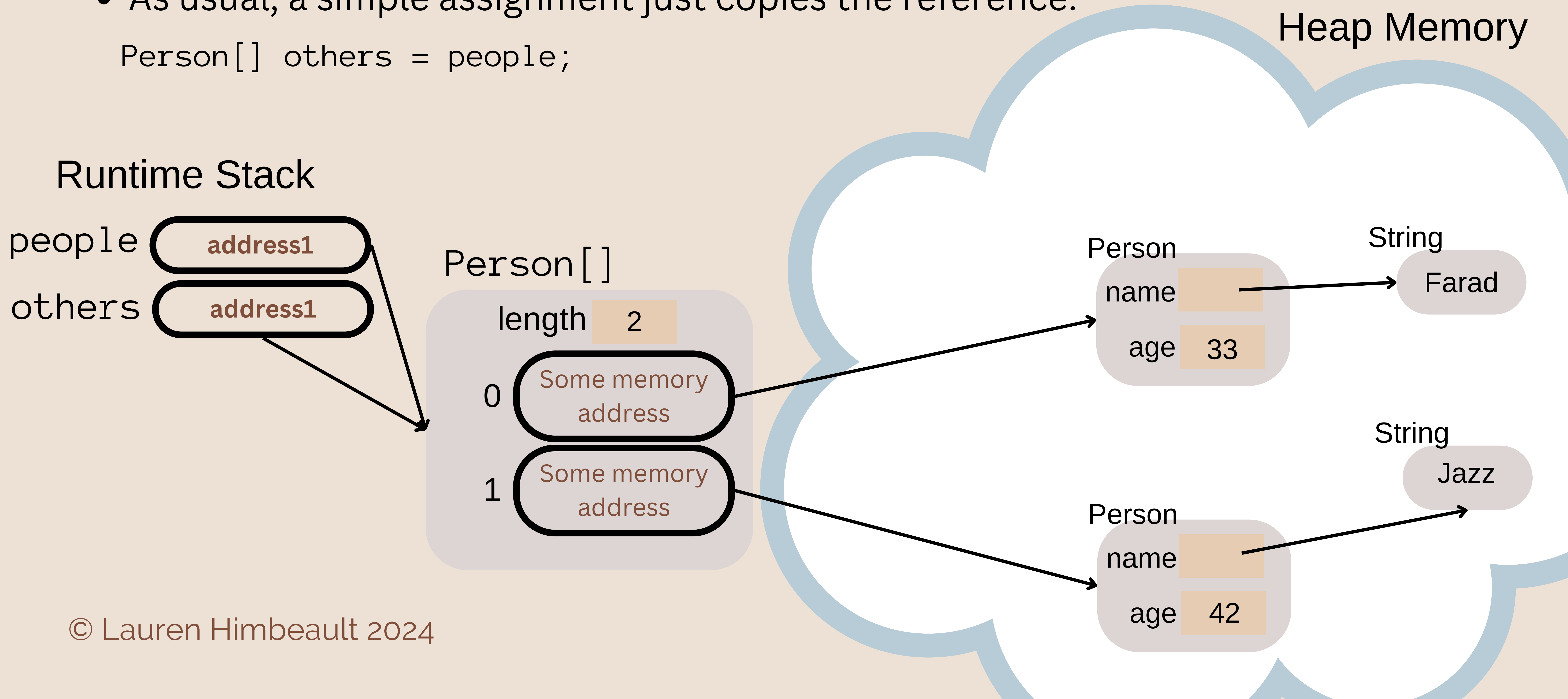
```
Person[] people = {new Person("Farad", 33),
                   new Person("Jazz", 42)};
```



Consider the following:

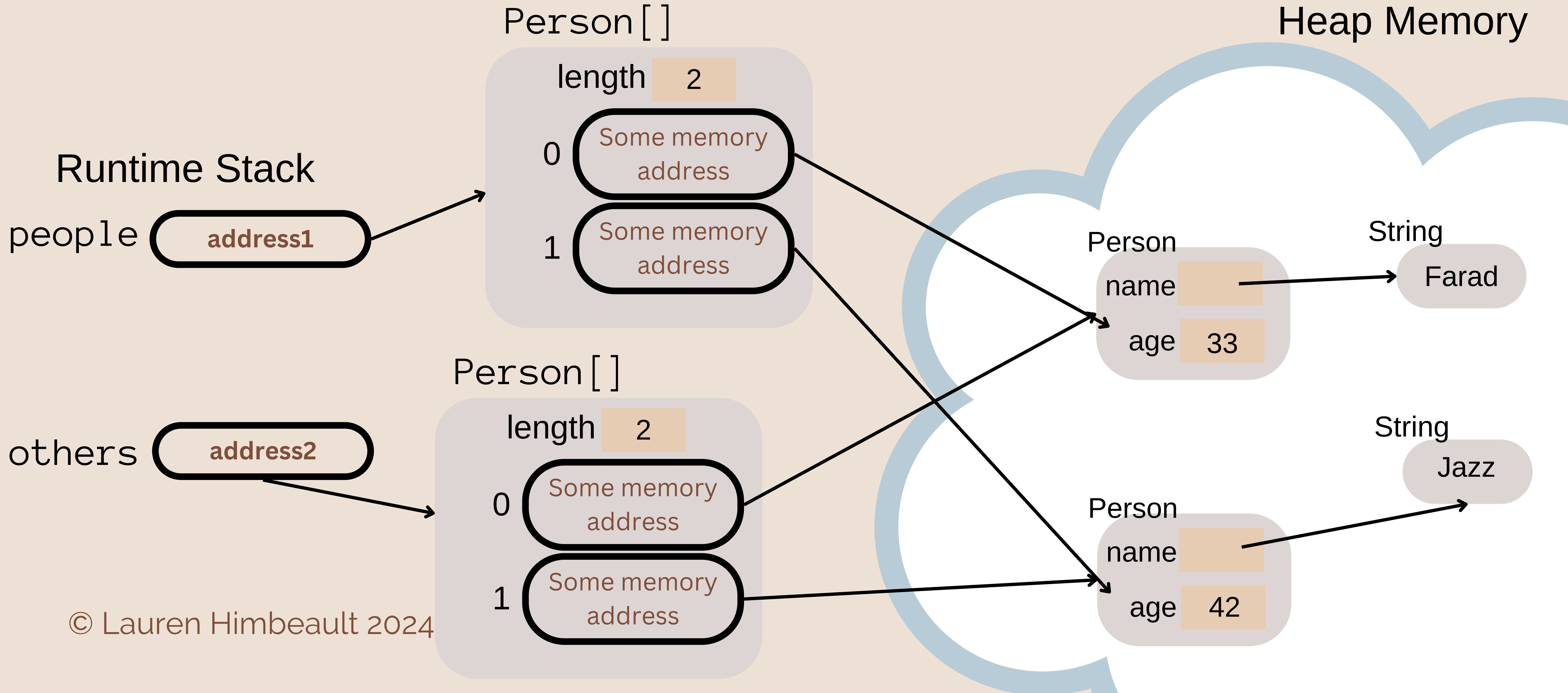
- As usual, a simple assignment just copies the reference:

```
Person[] others = people;
```



- If we use System.arraycopy (or a for loop), we'll get a new Person[ ] array:

```
Person[] others = new Person[people.length];
System.arraycopy(people, 0, others, 0, people.length);
```

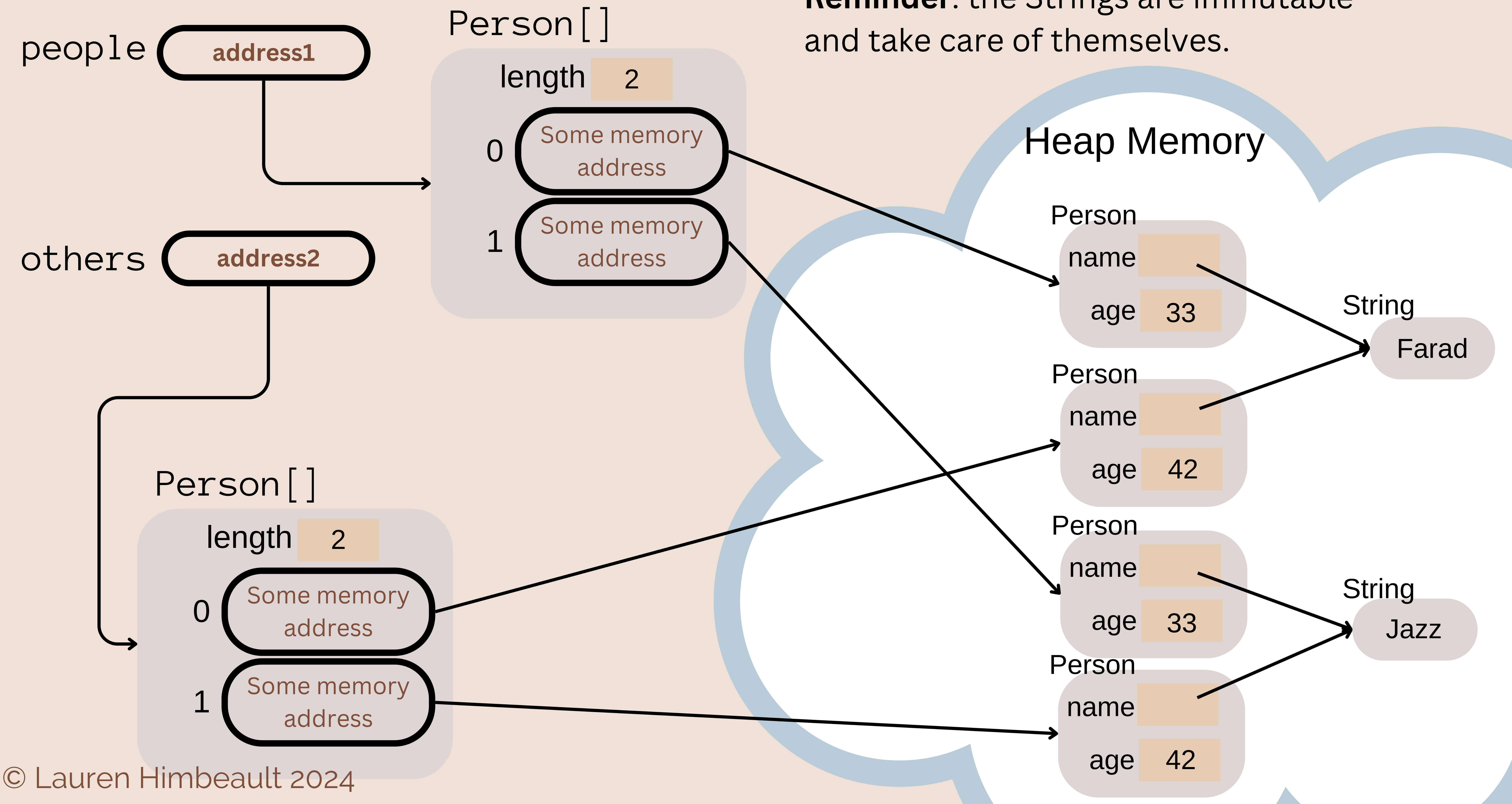


True Deep Copy

- To make **two fully independent copies,** we'd need to make clones of the Person objects, too. **(Note that this is not always what we would want)**
- We'll need to write our own for loop this time:

```
Person[] others = new Person[people.length];
for(int i=0; i<people.length; i++) {
    others[i] = people[i].clone();
}
```

Reminder: the Strings are immutable and take care of themselves.



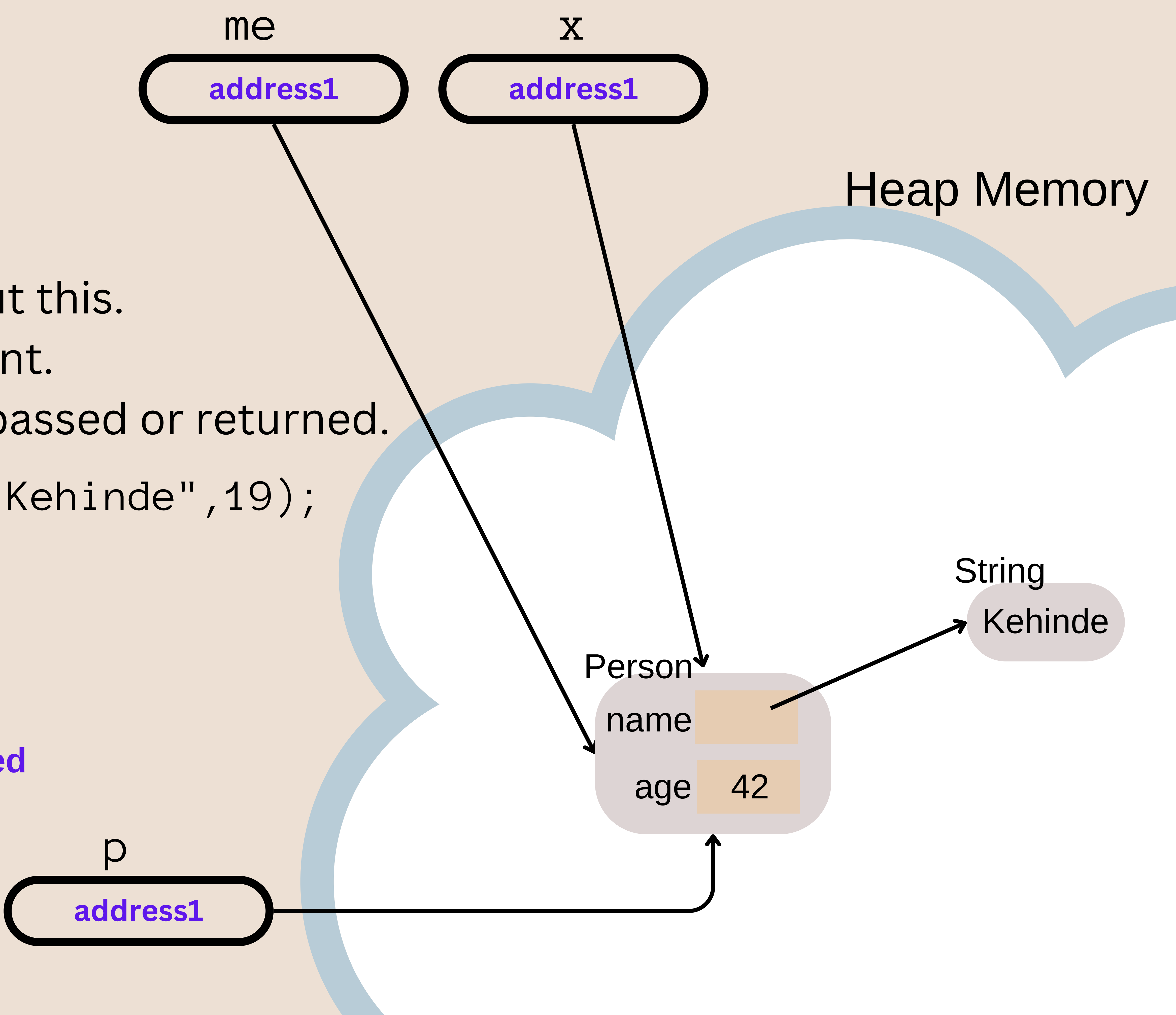
Objects as Parameters

(We should know this from Arrays)

- There is nothing special about this.
  - It's the same as assignment.
  - It's the reference that is passed or returned.

```
Person me = new Person("Kehinde",19);
Person x = me;
someMethod(me);
...
```

```
// a copy of the reference of me is passed
void someMethod(Person p){
    ...
}
```



Pause & Practice

Assume the Person class has both a copy constructor **and** a clone() method and they are implemented exactly correctly.

For each of the following code snippets (6 in total over the remaining slides), determine if obj1 == obj2 (deep vs shallow copy).

For the answers, code it up yourself
(yes you will have to code up copy constructors and clone() methods)

Pause & Practice

```
Person person1 = new Person("Aarav", 30);
Person person2 = person1; // Assigning reference
// Is person1 == person2?
```

```
Person person1 = new Person("Sara", 25);
Person person2 = new Person(person1); // copy constructor
// Is person1 == person2?
```

```
Person person1 = new Person("Ibrahim", 40);
Person person2 = person1.clone(); // clone method
// Is person1 == person2?
```

Pause & Practice

```
Person[] persons1 = { new Person("Deepak", 35),
                      new Person("Fatima", 28)};

Person[] persons2 = persons1; // Assigning reference
// Is persons1[0] == persons2[0]?
// Is persons1 == persons2?
```

Pause & Practice

```java
Person[] persons1 = { new Person("Raj", 45),
                      new Person("Layla", 32)};

Person[] persons2 = new Person[persons1.length];

for(int i = 0; i < persons1.length; i++) {
    persons2[i] = new Person(persons1[i]);
}

// Is persons1[0] == persons2[0]?
// Is persons1 == persons2?
```

Pause & Practice

```java
Person[] persons1 = { new Person(”Amir”, 50),
                      new Person(”Priya”, 27)};

Person[] persons2 = persons1.clone();

for(int i = 0; i < persons1.length; i++) {
    persons2[i] = persons1[i].clone();
}

// Is persons1[0] == persons2[0]?
// Is persons1 == persons2?
```