COMP 1020 - A01, D01 Using objects

FALL 2020

References to objects

- This has been said many times before, but let's repeat it again:
- 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

References to objects

- This has been said many times before, but let's repeat it again:
- 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

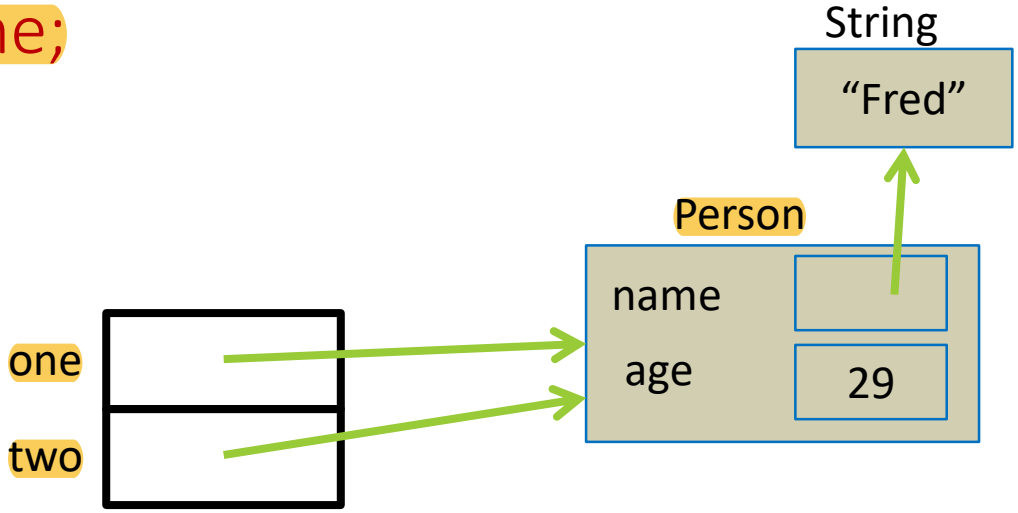
- A simple assignment statement will only copy the references, not the objects themselves (a "shallow copy"):

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

Cloning objects

- A simple assignment statement will only copy the references, not the objects themselves (a "shallow copy"):

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



Cloning objects

- To make a completely new object, identical to an existing one, you need to write a method
  - This is traditionally named clone( )

Clone() method

- A clone( ) method for the Person class:

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

Clone() method

- A clone( ) **method for the Person class:**

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

Notice the return type: Person → we want to return a Person object that is a clone of the current object

Clone() method

- A clone( ) method for the Person class:

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

- This is much simpler than:

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

Clone() method

- A clone( ) method for the Person class:

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

- This is much simpler than:

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

Lesson is: Keep it simple! Use your methods (that you defined previously)!

Clone() method

- A clone( ) method for the Person class:

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

- This is much simpler than:

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

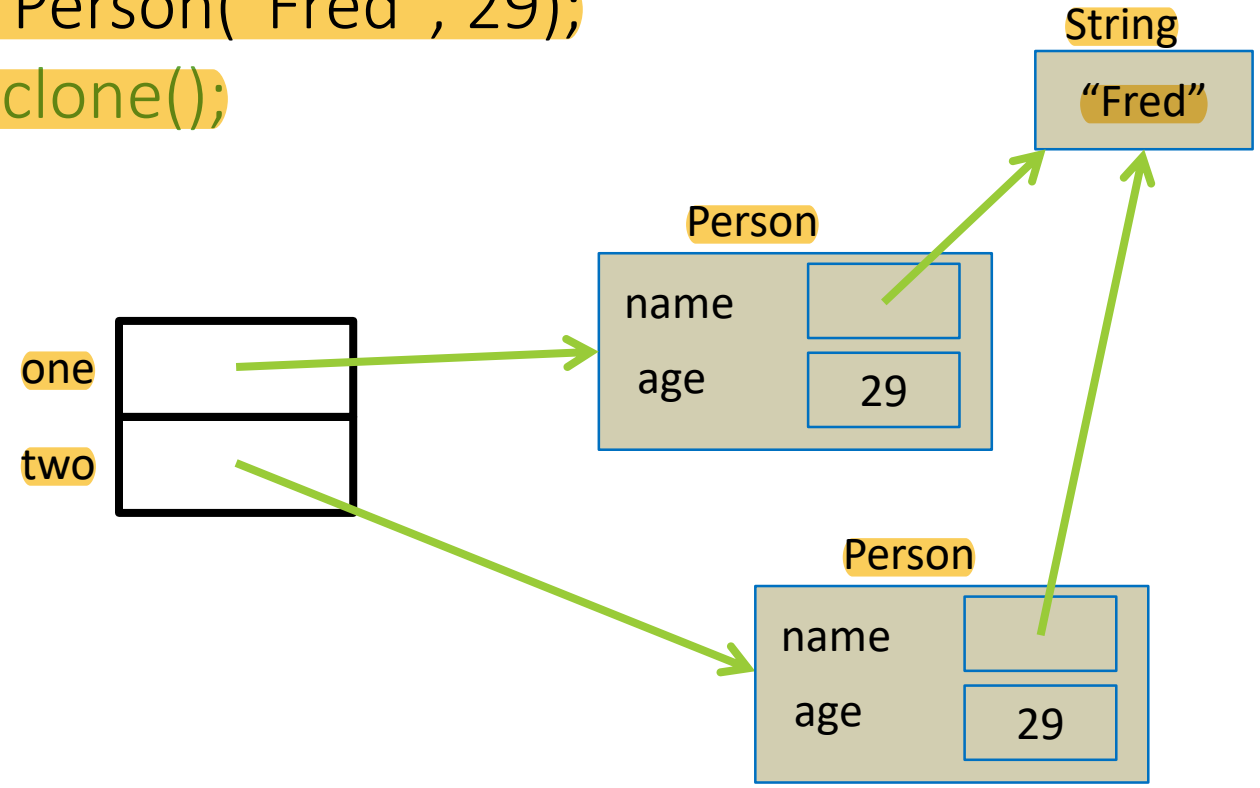
By the way: this. is not necessary here (no naming conflict), but I'm using it anyway

Cloning objects

- Now if we did:

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

- we'd get:

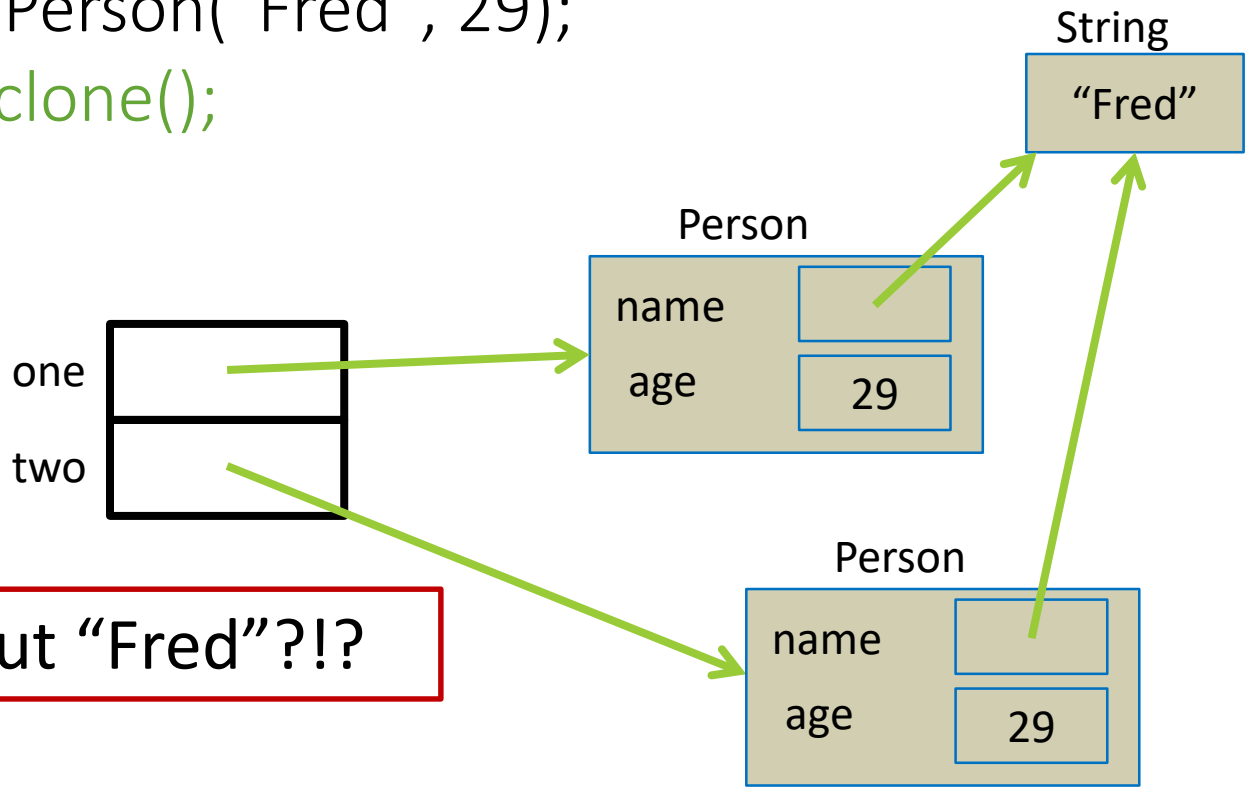


Cloning objects

- Now if we did:

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

- we'd get:



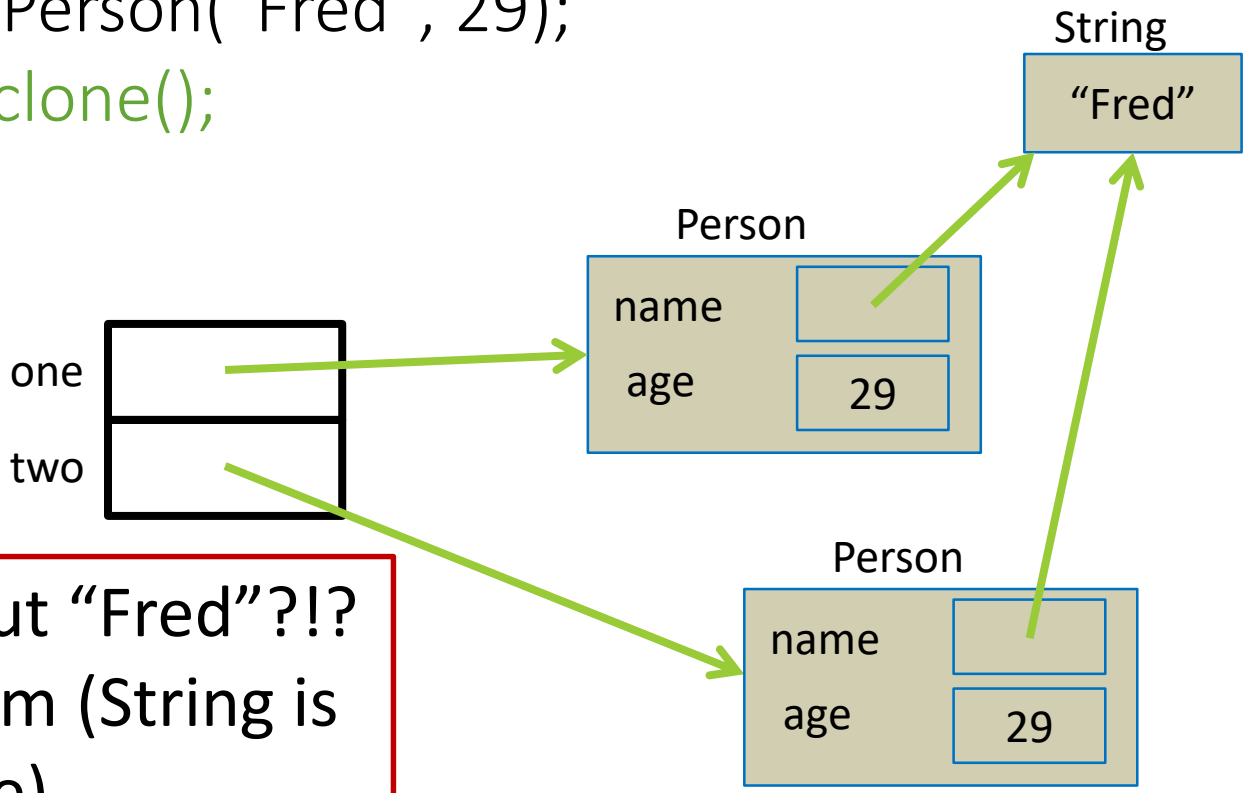
Wait! What about "Fred"?!?

Cloning objects

- Now if we did:

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

- we'd get:



Wait! What about "Fred"?!?

- No problem (String is immutable)

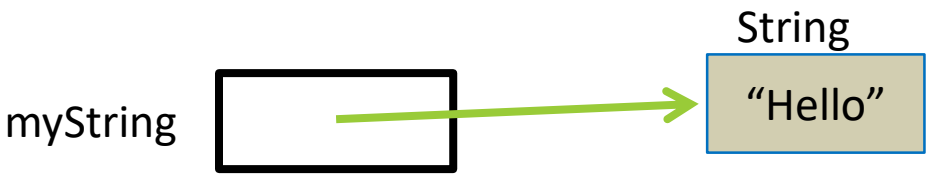
Strings are immutable

- Every String is immutable: once it's created, you cannot change its value
- That means, every time you "modify" the value of a String variable, what actually happens, behind the scenes:
  - A new String object is created, and the new reference to it is returned

Strings are immutable

- Example:

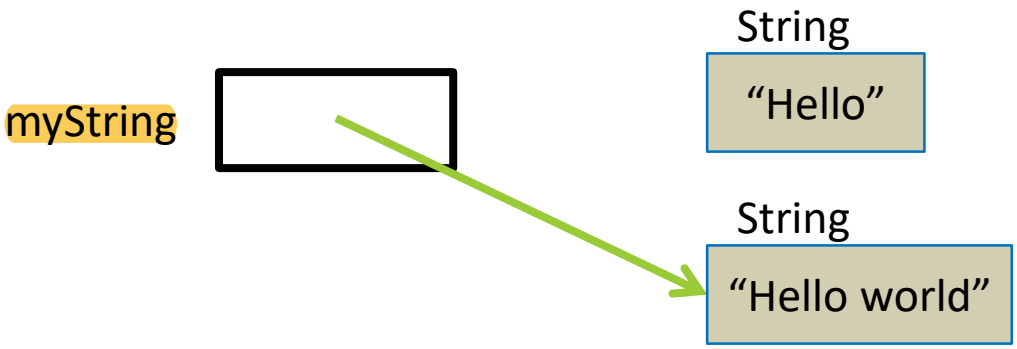
String myString = “Hello”;



Strings are immutable

- Example:

```
String myString = “Hello”;
myString = myString + “ world”;
```



- You are never modifying a String in place, you always get a new one → String is immutable

Back to clone, what's the difference?

- A simple assignment (shallow copy) gives two references to the same object

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

- This is known as an alias

- Any changes to one of them will affect the other

Back to clone, what's the difference?

- 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)

Back to clone, what's the difference?

- A clone (deep copy) gives two independent objects

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

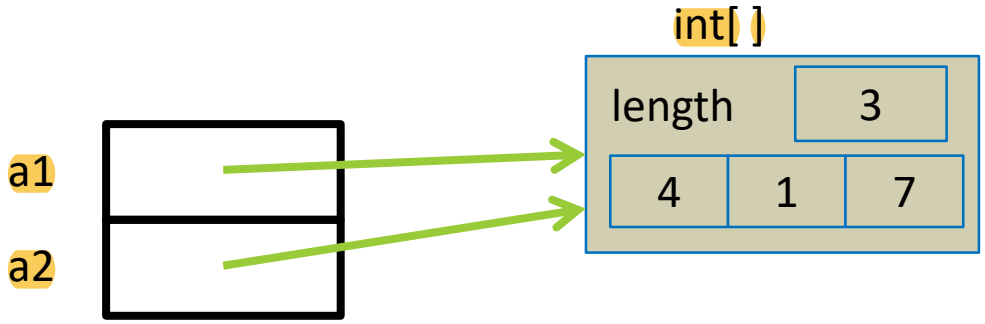
- A change to one will not affect the other
  - This is not an issue with String objects (or other "immutable" objects because they can't be changed)

- Neither one is right or wrong, depends on what you need: use the one that does what you want it to do

What about arrays?

- Arrays are objects, too. Using a simple assignment copies only the reference:

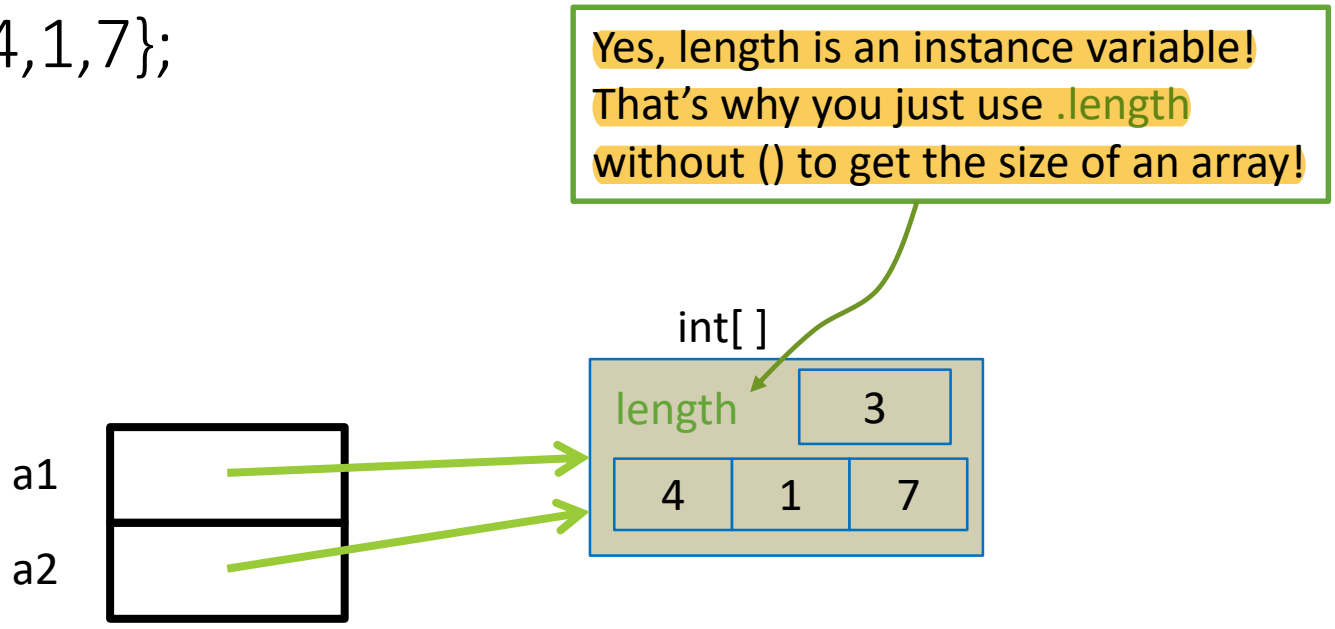
```
int[] a1 = {4,1,7};
int[] a2;
a2 = a1;
```



What about arrays?

- Arrays are objects, too. Using a simple assignment copies only the reference:

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



Cloning arrays

- We can't add a clone( ) method to the int[ ] class!
  - There is no such class, anyway.

- We have to use:

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

Cloning arrays

- Or we can take a slight shortcut:

```
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. */
```

System.arraycopy()

- The method call

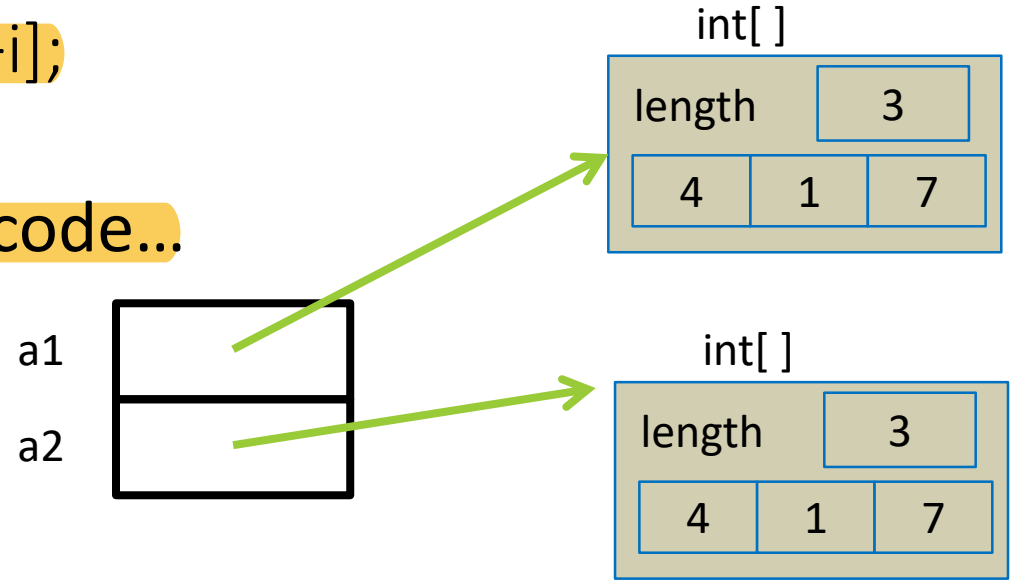
  System.arraycopy(a1, p1, a2, p2, n);

- is the same as

  for(int i=0; i<n; i++)

  a2[p2+i] = a1[p1+i];

- It doesn't save much code…



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!

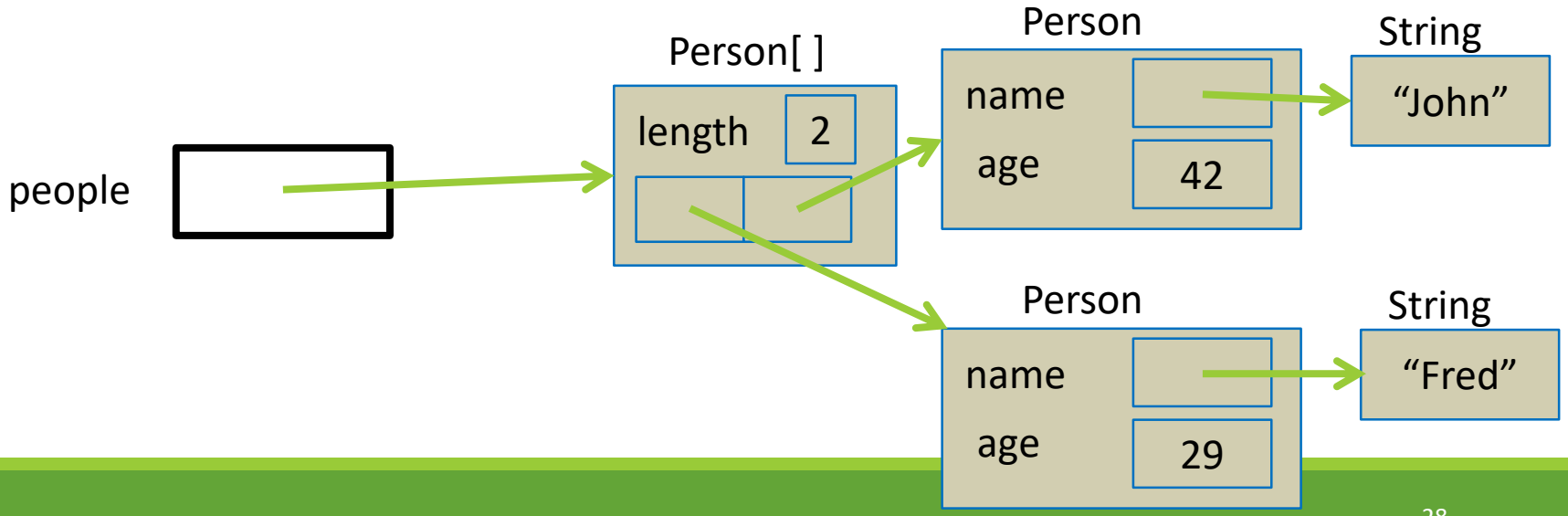
Arrays of objects

- 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
  - You might want shallow copies, you might want deep copies → every situation is different
    - Think! Plan on paper before implementing!

Array of Person objects

- Make an array of Person objects:

```
Person[] people = {new Person("Fred", 29),
                   new Person("John", 42)};
```



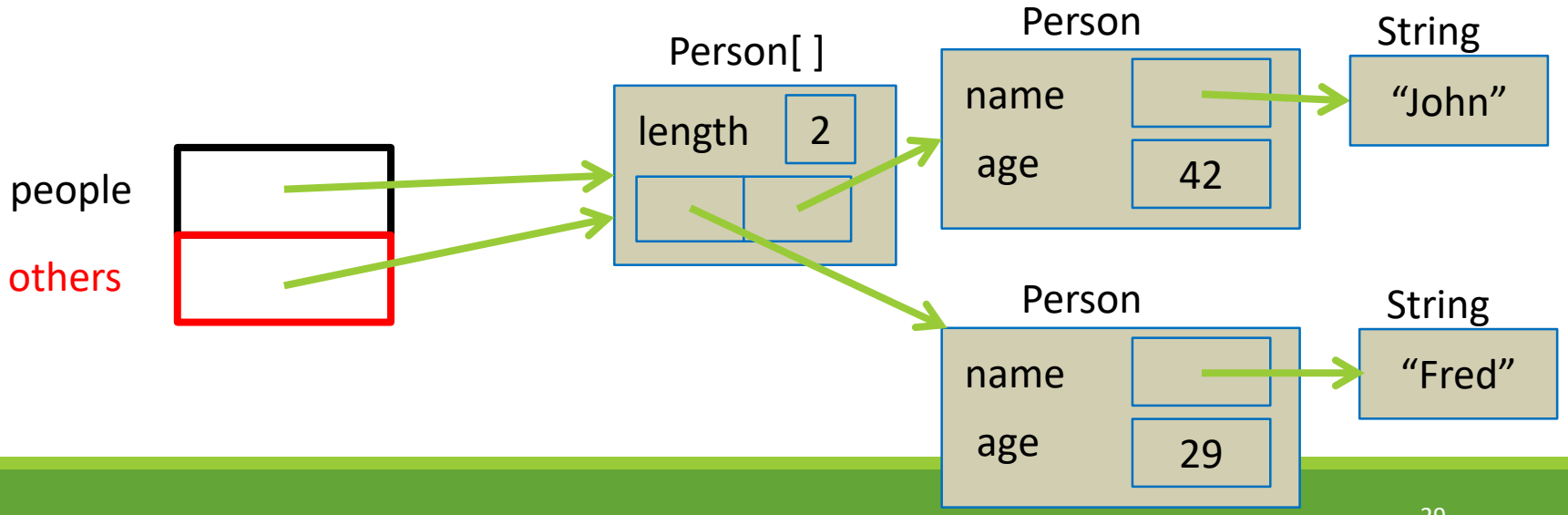
Array of Person objects

- Make an array of Person objects:

  Person[] people = {new Person("Fred", 29),
  new Person("John", 42)};

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

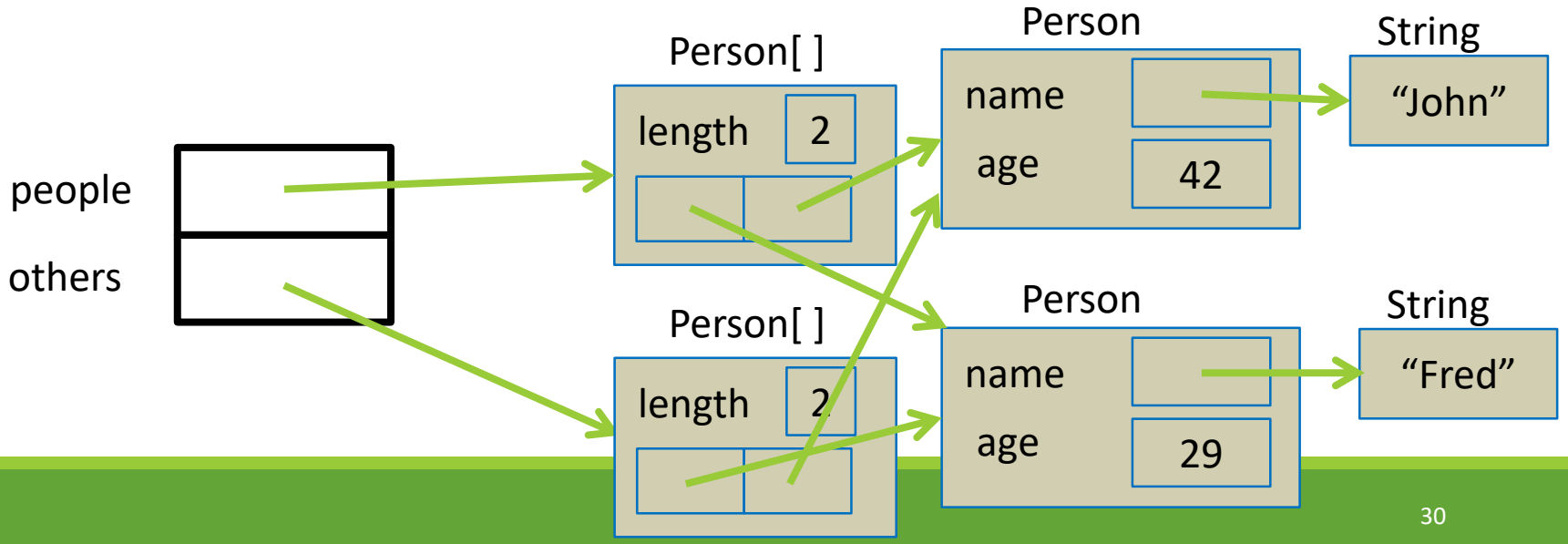
  Person[] others = people;



Array of Person objects

- 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);
```

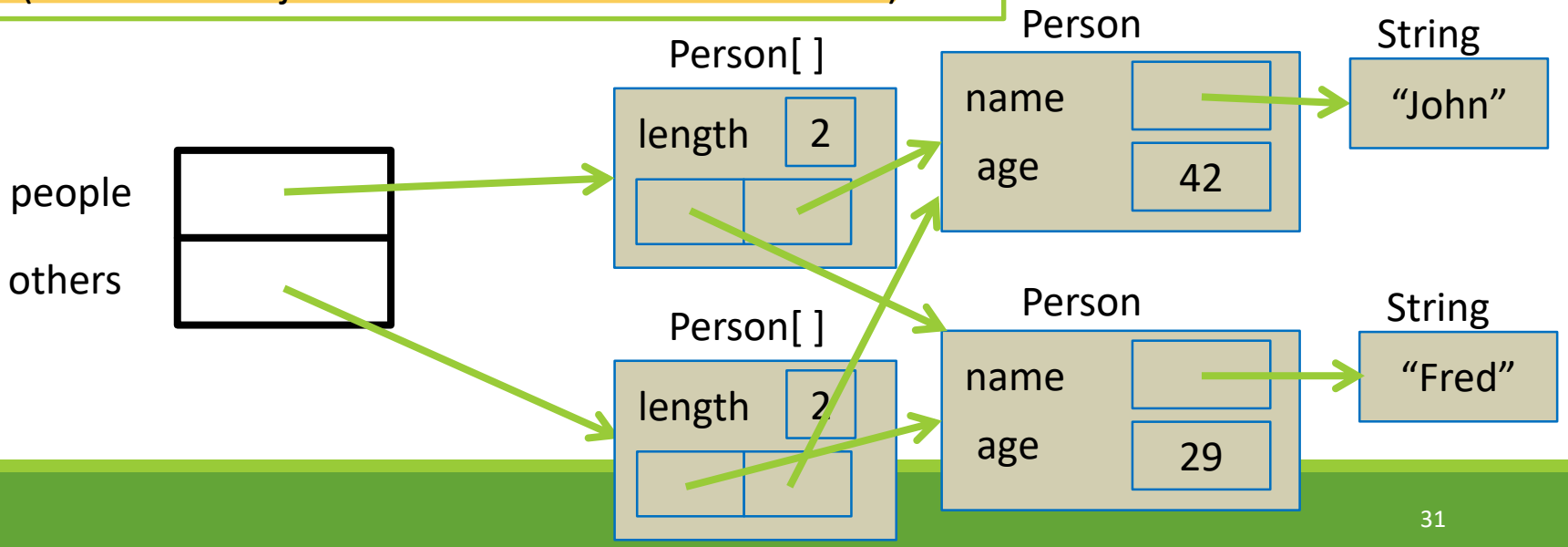


Array of Person objects

- 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);
```

Ok, so what do we have here… we get two different arrays of Person objects, but the references to the Persons were only copied (the Person objects themselves were not cloned!)



A true “deep copy”

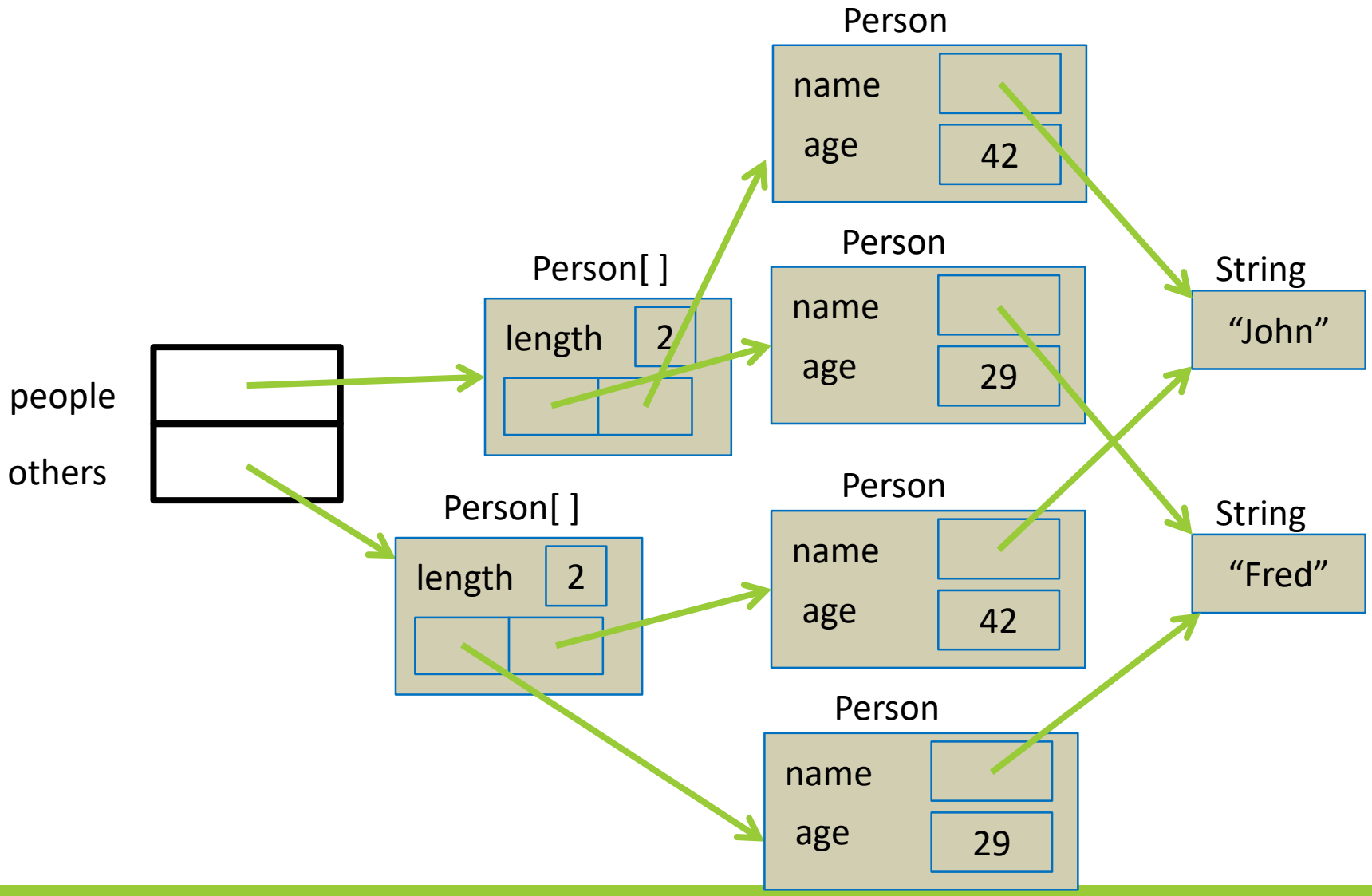
- To make two fully independent copies, we'd need to make clones of the Person objects, too. (Note that this is not always what we would want)

- 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();
```

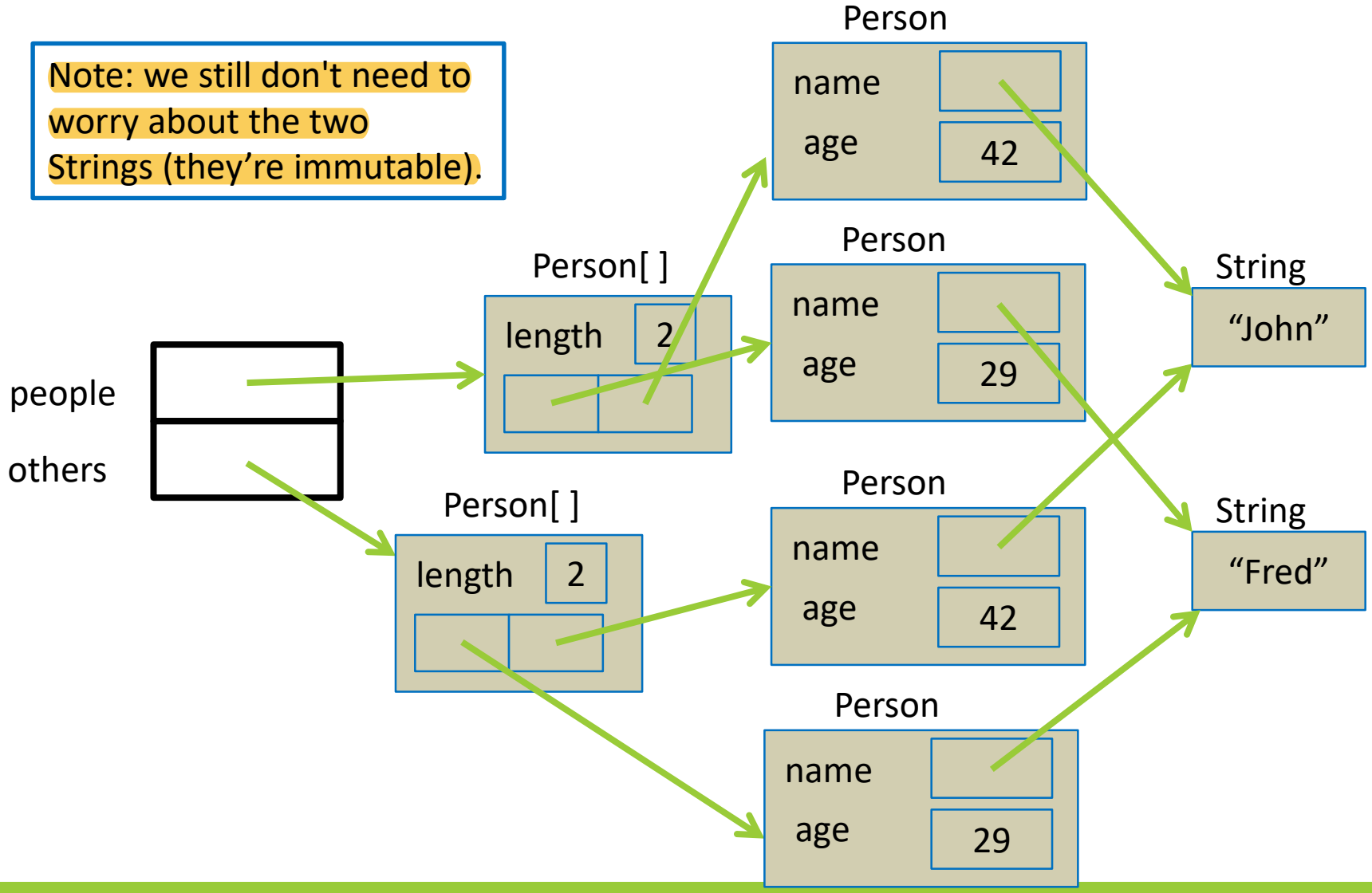
- Check the result of this on the next slide

Results of a “deep copy”



Results of a “deep copy”

Note: we still don't need to worry about the two Strings (they’re immutable).

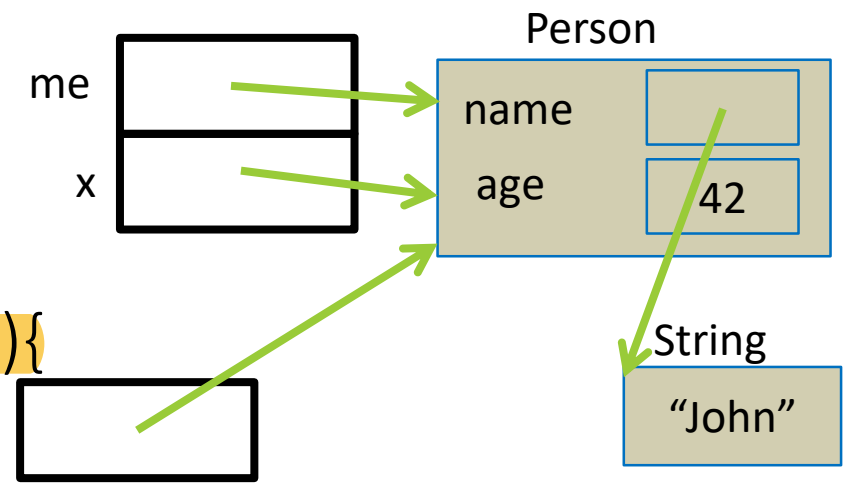


Objects as parameters / results

- 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("John",42);
Person x = me;
someMethod(me);
...

void someMethod(Person p){
    ...
}
```

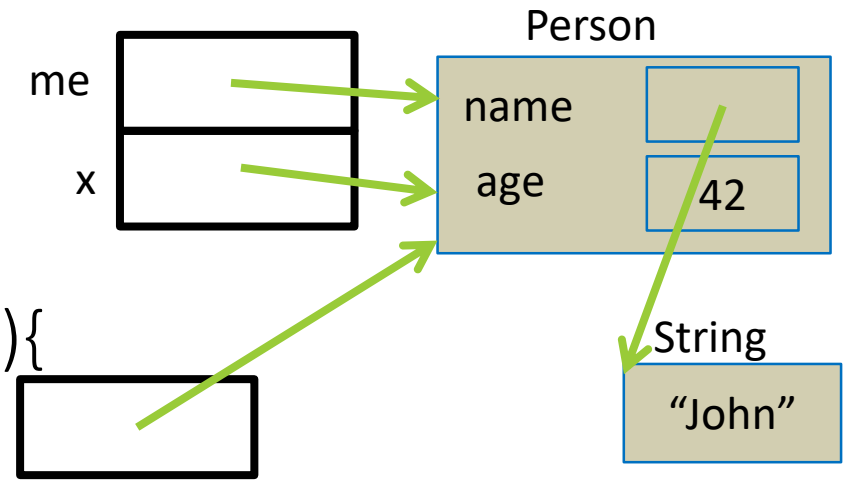


Objects as parameters / results

- 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("John",42);
Person x = me;
someMethod(me);
…
void someMethod(Person p){
    …
}
```

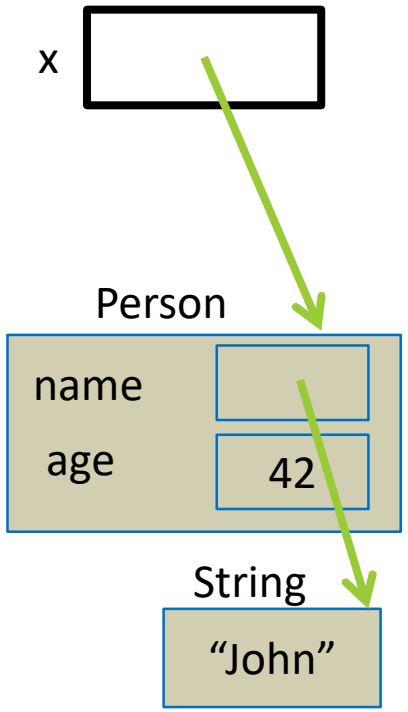
a copy of the reference of me is passed



Orphans and garbage collection

- When there are no places where the reference to an object is stored, it is no longer usable
  - It's an "orphan"

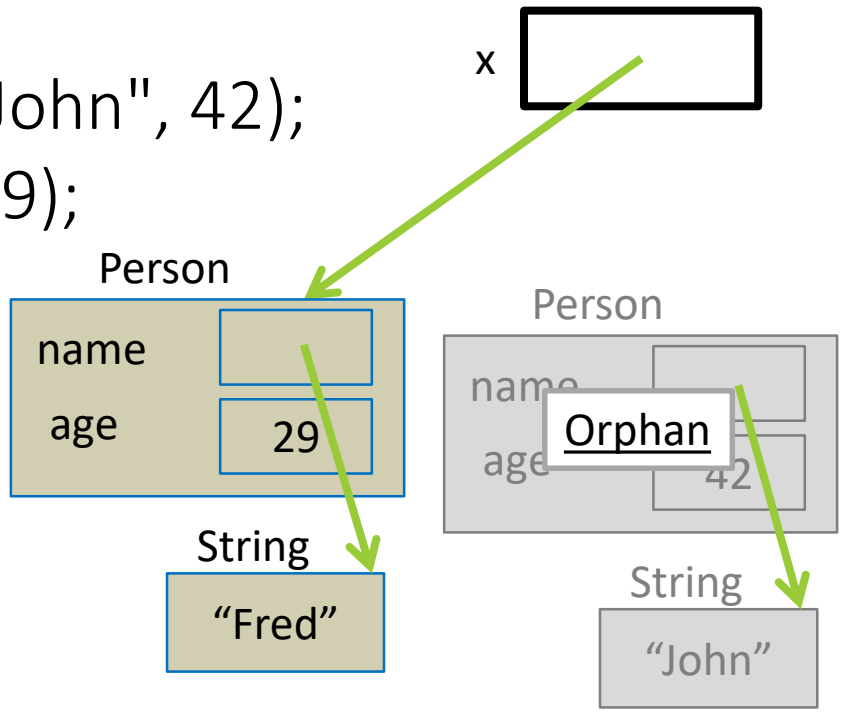
```
Person x = new Person("John", 42);
```



Orphans and garbage collection

- When there are no places where the reference to an object is stored, it is no longer usable
  - It's an "orphan"

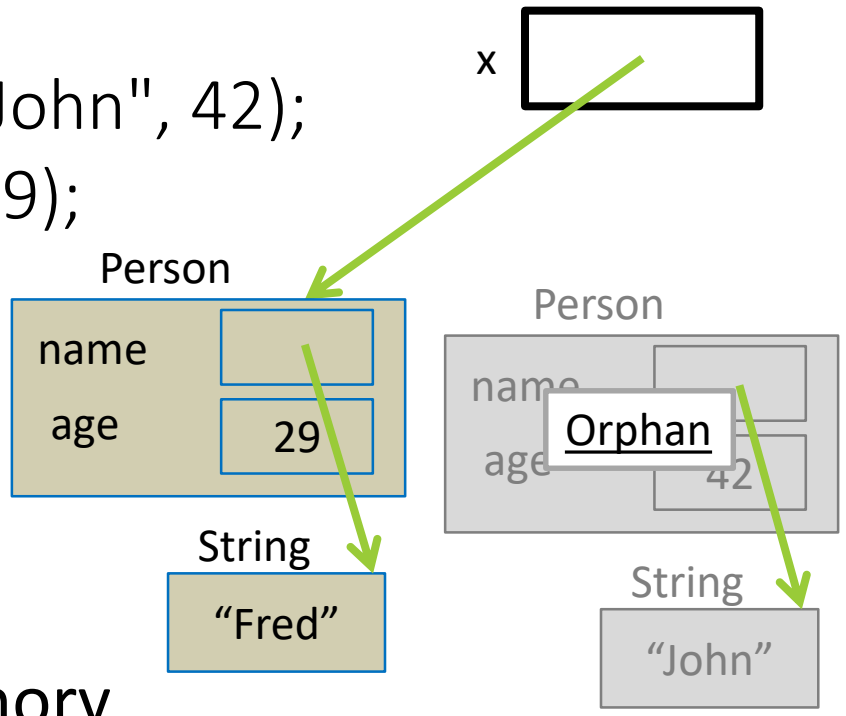
```
Person x = new Person("John", 42);
x = new Person("Fred", 29);
```



Orphans and garbage collection

- When there are no places where the reference to an object is stored, it is no longer usable
  - It's an "orphan"

```
Person x = new Person("John", 42);
x = new Person("Fred", 29);
```

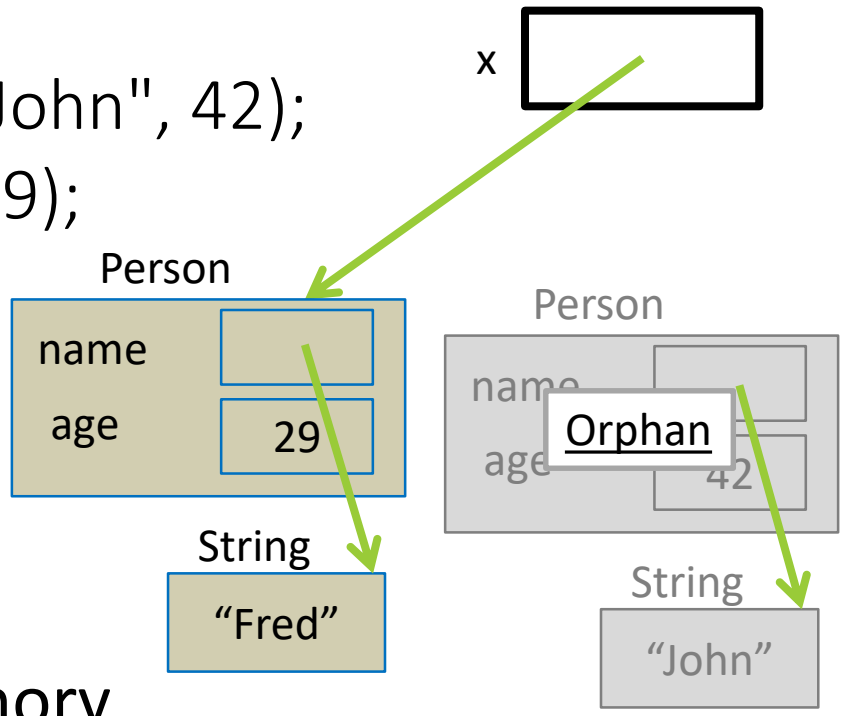


- Java will handle this
  - "garbage collection"
  - frees up any unused memory

Orphans and garbage collection

- When there are no places where the reference to an object is stored, it is no longer usable
  - It's an "orphan"

```
Person x = new Person("John", 42);
x = new Person("Fred", 29);
```



- Java will handle this
  - "garbage collection"
  - frees up any unused memory
  - "memory leaks" occur in other languages

Objects containing objects

- An instance variable in an object can be of any type, including object types
  - This means they contain a reference to some other object, not the object itself
  - This is extremely common and very powerful

Objects containing objects

- Let's change our Person object:

```
//Instance variables
private String name;
private int age;
private Person spouse; //null means no spouse
//how about Person[] children ? Sure. Later.
```

New methods are necessary

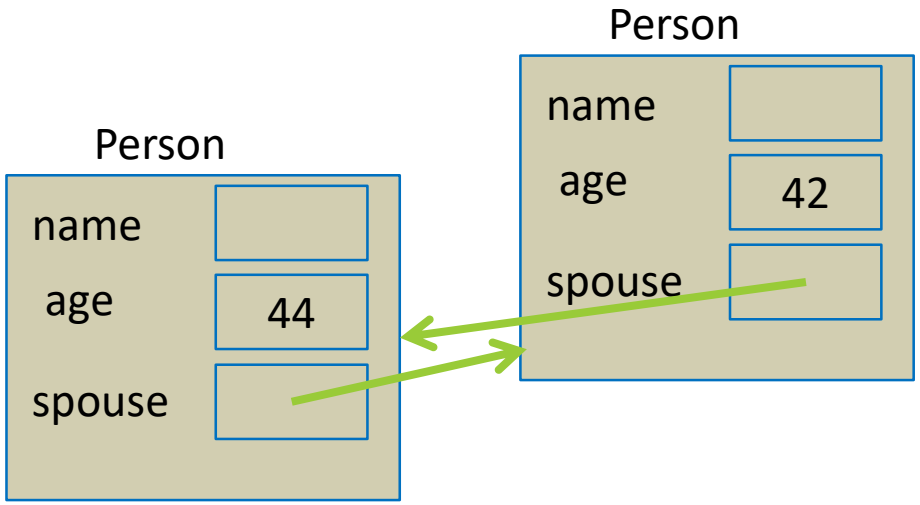
- A new constructor would be useful:

```
public Person(String who, int currentAge, Person otherHalf)
{
    name = who;
    age = currentAge;
    spouse = otherHalf;
    //make sure the other person is married, too!
    if(otherHalf != null)
        otherHalf.spouse = this;
    population++;
}//constructor
```

New methods are necessary

- How about new instance methods as well:

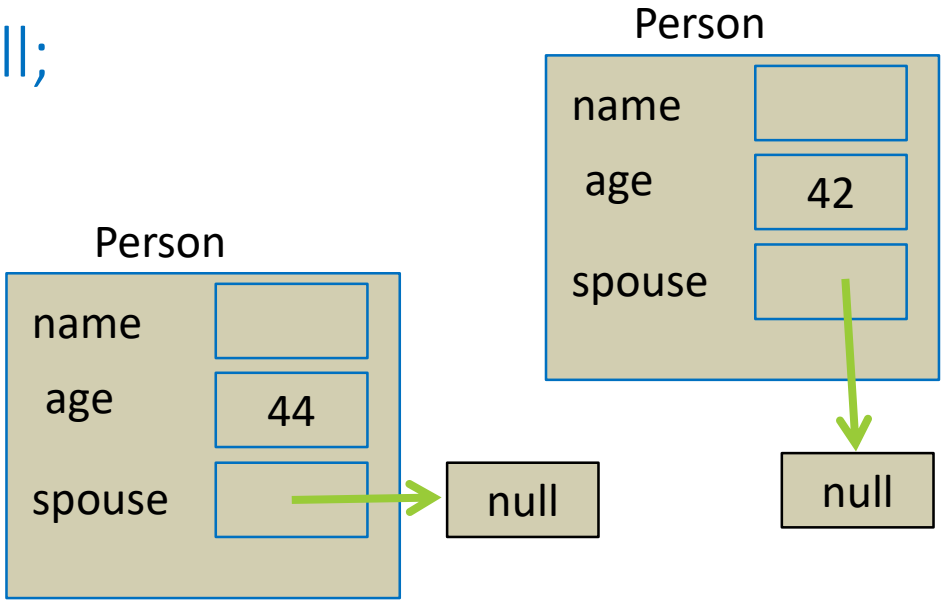
```
public void marries(Person other) {
    spouse = other;
    if (other != null)
        other.spouse = this;
}//marries
```



New methods are necessary

- How about new instance methods as well:

```
public void divorces() {
    if (spouse != null){
        spouse.spouse = null;
        spouse = null;
    }
}//divorces ☹
```

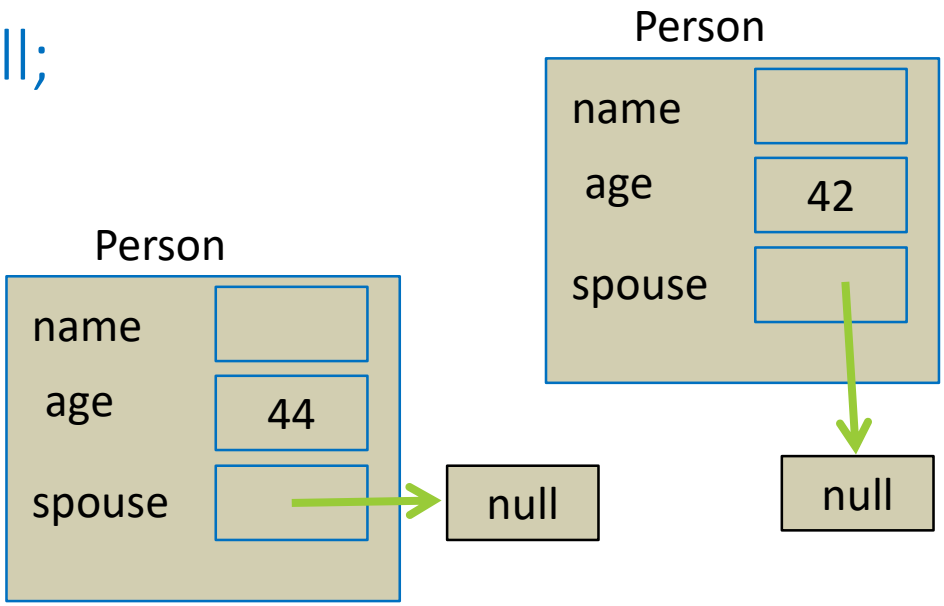


New methods are necessary

- How about new instance methods as well:

```
public void divorces() {
    if (spouse != null){
        spouse.spouse = null;
        spouse = null;
    }
}//divorces ☹
```

Order of operations is important here!
If you did it the other way around:
spouse = null;
spouse.spouse = null;
You would get a null pointer exception!



New methods are necessary

- How about new instance methods as well:

```
public boolean isMarried() {
    return spouse != null;  //don't use an IF here, useless!
}

public Person getSpouse() {
    return spouse;
}
```

New methods are necessary

- We might want to update the toString method to print the name of the spouse…
- How would we do that?
- → let's update our old Person.java example

Updating Person.java

- Note that we have a large number of very small and simple methods:
  - This is how OOP code should be
  - Results in code that is easy to maintain / change

Passing an object to a method

- We've seen earlier that it works the same way as if it was a primitive type → you just declare the type of the parameter (Person for example)

```
public void marries(Person other) {
    spouse = other;
    if (other != null)
        other.spouse = this;
}//marries
```

Passing an object to a method

- But how does passing a parameter really work in Java?

- Java always passes a copy of the variable to a method, not the variable itself

  - When passing a primitive type, a copy of the value is passed to the method

  - When passing an object, a copy of the reference is passed to the method

Passing an object to a method

- Example of passing a primitive type:

```
//In a class:
public static void main (String[] args) {
    int x = 5;
    changeValue(x);
    System.out.println(x);  //What is printed?
}

public static void changeValue(int x) {
    x += 10;
}
```

Passing an object to a method

- Example of passing a primitive type:

```
//In a class:
public static void main (String[] args) {
    int x = 5;
    changeValue(x);
    System.out.println(x);  //What is printed? 5
}

public static void changeValue(int x) {
    x += 10;
}
```

x here is just a copy of the value that was passed to the method!

Passing an object to a method

- Example of passing an object:

```
//In a class:
public static void main (String[] args) {
    Person p = new Person("George", 65);
    changeValue(p);
    System.out.println(p);  //What is printed?
}

public static void changeValue(Person p) {
    p = new Person("Janet", 48);
}
```

Passing an object to a method

- Example of passing an object:

```
//In a class:
public static void main (String[] args) {
    Person p = new Person("George", 65);
    changeValue(p);
    System.out.println(p);  //What is printed? George (65)
}

public static void changeValue(Person p) {
    p = new Person("Janet", 48);
}
```

p here is just a copy of the reference that was passed to the method! Modifying where it points to does not affect the initial reference that was passed to the method!

Passing an object to a method

- Example of passing an object:

```
//In a class:
public static void main (String[] args) {
    Person p = new Person("George", 65);
    changeValue(p);
    System.out.println(p);  //What is printed?
}

public static void changeValue(Person p) {
    p.haveBirthday();
}
```

Passing an object to a method

- Example of passing an object:

```
//In a class:
public static void main (String[] args) {
    Person p = new Person("George", 65);
    changeValue(p);
    System.out.println(p);  //What is printed? George (66)
}

public static void changeValue(Person p) {
    p.haveBirthday();
}
```

p here is still accessing the same object in memory, so calling an instance method will affect the object. Just like an alias.

One final step

- Let's add a list of children to our Person object
- But a list of people is a different thing from a Person…
  - It has its own unique actions
    - Print the whole list
    - Search for a certain Person in the list
    - Add/delete from the list (delete!? This example is becoming very dark…)

What's our best strategy?

- There should be a separate PersonList class, which will handle all these operations
- Write a PersonList class with:
  - A "partially-filled array" of Person
    - Use a generous fixed size
  - A constructor to make an empty list
  - Methods addPerson and toString

Link the two classes

- Add an instance variable PersonList children to the Person class
  - Adjust the constructors as needed

- Provide methods in Person, that will make use of the methods in PersonList
  - void addChild(Person)
  - String getListOfChildrenString( )
  - Let's build this!

What's the point of PersonList?

- Why build a PersonList class, and not just dealing with everything inside Person (Person[] as an instance variable in Person)?

What's the point of PersonList?

- Why build a PersonList class, and not just dealing with everything inside Person (Person[] as an instance variable in Person)?

- Reusability!
  - PersonList is a general class that can be reused every time you need a list of Person objects
  - Can be used for other purposes than list of children:
    - List of employees
    - List of students
    - Etc.

What's the point of PersonList?

- Why build a PersonList class, and not just dealing with everything inside Person (Person[] as an instance variable in Person)?

- Also, compartmentalization and encapsulation!
  - Dividing the work between the different objects: PersonList will take care of all operations that can be done on its data (the partially-filled array)
  - The original Person object won't have to worry about how PersonList manages the list, and just use the public methods offered by PersonList (encapsulation)