COMP 1020 - ArrayLists and collections

UNIT 3

All slides © their respective authors; redistribution is not permitted

Partially-full arrays

- We've used "partially-full arrays" a lot…
  - An array with some maximum size
  - An integer to keep track of the current size
    - i.e. the number of elements of the array currently in use

All slides © their respective authors; redistribution is not permitted

Partially-full arrays

- They have some drawbacks:
  - The maximum size is a limitation
    - It can be increased in some cases by allocating an entirely new array, and copying all of the existing elements… but that’s not ideal
  - Elements must be shifted left or right often
    - When inserting, deleting, moving, sorting, etc.
- Two variables are used for one "list“ (array and size)
  - Could make an Object out of it to solve this…

All slides © their respective authors; redistribution is not permitted

ArrayList class

- Java has the built-in ArrayList class that provides some sort of “partially-full arrays”
  - It works exactly as we would write it ourselves
  - import java.util.ArrayList is required

All slides © their respective authors; redistribution is not permitted

ArrayList class

- To create one for a specific type of data:

```
ArrayList<String> myList = new ArrayList<String>( );
```

  - This is some new syntax called a "generic"
    - We probably won't have time to cover generics in this course…

All slides © their respective authors; redistribution is not permitted

ArrayList class

- To create one for a specific type of data:

  ```
  ArrayList<String> myList = new ArrayList<String>( );
  ```

  - This is some new syntax called a "generic"
    - We probably won’t have time to cover generics in this course…

- To create one that will store any kind of (Object) data:

  ```
  ArrayList myList = new ArrayList( );
  ```

  - This is functionally the same as ArrayList<Object>.

All slides © their respective authors; redistribution is not permitted

Objects in ArrayLists

- One small disadvantage to ArrayLists:
  - They can only hold Objects, not primitive types
    - ArrayList<String> is OK
    - ArrayList<int> is an error
      - Can’t use int, double, boolean, char, float, long, byte, short

All slides © their respective authors; redistribution is not permitted

Objects in ArrayLists

- One small disadvantage to ArrayLists:
    - They can only hold Objects, not primitive types
        - ArrayList<String> is OK
        - ArrayList<int> is an error
            - Can’t use int, double, boolean, char, float, long, byte, short

- But ArrayList or ArrayList<Object> can hold any object
    - And Integer, Double, Boolean, Character, Long are classes that give Object versions of the primitive types, allowing them to be used, too!

All slides © their respective authors; redistribution is not permitted

Common ArrayList methods

- Let’s set up:

```
ArrayList<String> a = new ArrayList<String>( );
```

- Adding objects to an ArrayList:

```
a.add("testing"); //adds to the end
a.add("hippo"); //now "testing" "hippo"
a.add(1, "second"); //add “second” to index 1
    // now "testing" "second" "hippo"
a.add(10,"far"); //IndexOutOfBoundsException -
                 //can't leave "gaps"
```

Common ArrayList methods

- The add method returns true every time (?why?)

  - It's standard for “collections” like ArrayLists to return a boolean result meaning “did it change?”

All slides © their respective authors; redistribution is not permitted

Common ArrayList methods

- The add method returns true every time (?why?)

  - It's standard for “collections” like ArrayLists to return a boolean result meaning “did it change?”

- Determining the size of the list:
  a.size()

  - But arrays use .length and Strings use .length()
  - Why aren't these things ever consistent? ☹

All slides © their respective authors; redistribution is not permitted

Common ArrayList methods

- Removing objects from an ArrayList:

```
a.remove(0); //Removes the first element.
          //All others move left one place.
          //returns the deleted element
a.remove("hippo"); //removes that String
          //returns a boolean ("was it there?")
a.remove(10); //IndexOutOfBoundsException
a.clear(); //a is now empty. This method is void.
```

All slides © their respective authors; redistribution is not permitted

Common ArrayList methods

- To obtain or replace (get or set) objects:

```
a.get(0)  //Just like a[0] would be for an array
a.get(a.size()-1)  //gets the last one.
a.set(0, "new") // replaces the first one.
    //returns the old value that was deleted.
a.set(10, "new") // IndexOutOfBoundsException
```

  - Since ArrayLists contain only Objects, the result is always a reference to an Object

All slides © their respective authors; redistribution is not permitted

Common ArrayList methods

- To obtain or replace (get or set) objects:

```
a.get(0)  //Just like a[0] would be for an array
a.get(a.size()-1)  //gets the last one.
a.set(0, "new") // replaces the first one.
     //returns the old value that was deleted.
a.set(10, "new") // IndexOutOfBoundsException
```

  - Since ArrayLists contain only Objects, the result is always a reference to an Object

- The usual toString method is there:

```
a.toString( )
System.out.println(a) //This uses toString( ), too.
```

All slides © their respective authors; redistribution is not permitted

Common ArrayList methods

- Searching for things in ArrayLists:
  - int indexOf(Object)
    - returns the position of the first occurrence of that object, or -1
  - int lastIndexOf(Object)
    - searches from the other end
  - Both of the above methods will send .equals() messages to determine equality

Common ArrayList methods

- Note about equals:
  - String, Integer, Double, and most other built-in classes will all implement an equals method
  - Your classes should implement one, too
  - Remember: the default Object.equals just compares references → might not be what you want

All slides © their respective authors; redistribution is not permitted

Common ArrayList methods

- Searching for things in ArrayLists:
  - boolean contains(Object)
    - Simply detects whether it's there or not
      - Probably just does indexOf(Object) >= 0

All slides © their respective authors; redistribution is not permitted

Arrays vs. ArrayLists

Array	ArrayList
String[ ] a = new String[10];	ArrayList<String> a = new ArrayList<String>( );
a.length //cannot change	a.size( ) //changes after each modification
a[0]	a.get(0)
a[0] = "test"	a.set(0, "test")
…n/a…	a.add("new")
…n/a…	a.remove(0)
Contains any type	Contains objects only

All slides © their respective authors; redistribution is not permitted

ArrayList example 1

- Example 1: Build an alphabetical list of words
  - Approach: If you have an alphabetical list:

    ant bat cat elk frog
  - and you want to insert a new one: dog
  - Find the first one in the list that’s “bigger”:

    ant bat cat elk frog
  - Then insert the new one in that position:

    ant bat cat dog elk frog
  - If you can’t find a “bigger” one: e.g. insert **goat**
  - Then it just goes at the end:

    ant bat cat dog elk frog **goat**

See ArrayListInsert.java

ArrayList example 2

- Example 2: Remove duplicates from a list of words
  - Approach: For each word in the list:

    one two two three four three two
  - Try to find that word, searching from the far end:

    Find: one two two three four three two
  - Remove any duplicates, but not the original one:

    Remove: one two two three four three
    Find: one two two three four three
    Remove: one two three four three
    Find: one two three four three
    Leave: one two three four three

See ArrayListDuplicates.java

Wrapper classes

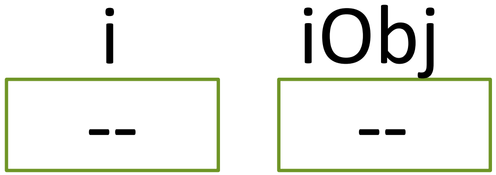
2. Use an “Integer object” – a tiny object that contains only a single integer.

- Java provides “wrapper classes” for all of the primitive types
- Primitive types: int, double, boolean, char, long, float, short, byte
- Object types: Integer, Double, Boolean, Character, Long, Float, Short, Byte
  - These store references to immutable objects (just like String)

All slides © their respective authors; redistribution is not permitted

Primitives vs Wrappers (old way)

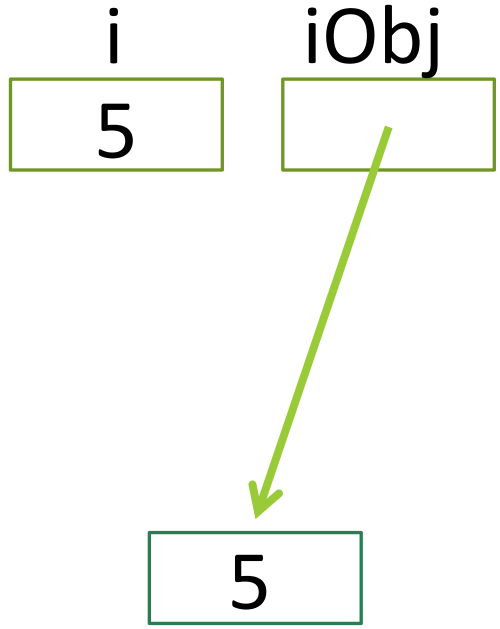
```
//Create variables
   int i;
   Integer iObj;
//Assign values
   i = 5;
   iObj = new Integer(5);
//Use the values
   i+1
   iObj.intValue( )+1
//Change the values
   i=3;
   iObj = new Integer(3);
```



All slides © their respective authors; redistribution is not permitted

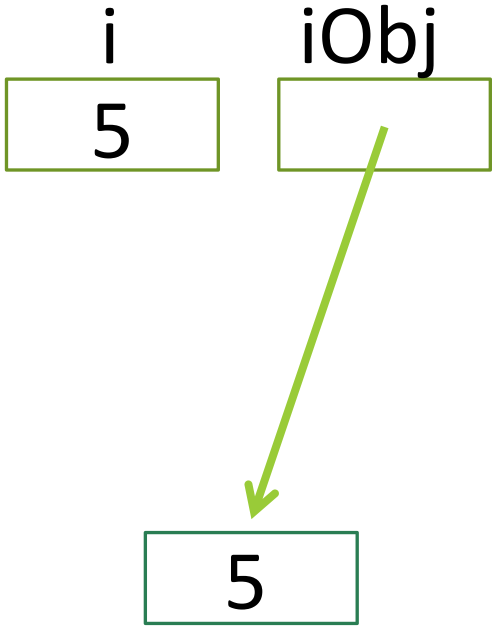
Primitives vs Wrappers (old way)

```
//Create variables
   int i;
   Integer iObj;
//Assign values
   i = 5;
   iObj = new Integer(5);
//Use the values
   i+1
   iObj.intValue( )+1
//Change the values
   i=3;
   iObj = new Integer(3);
```



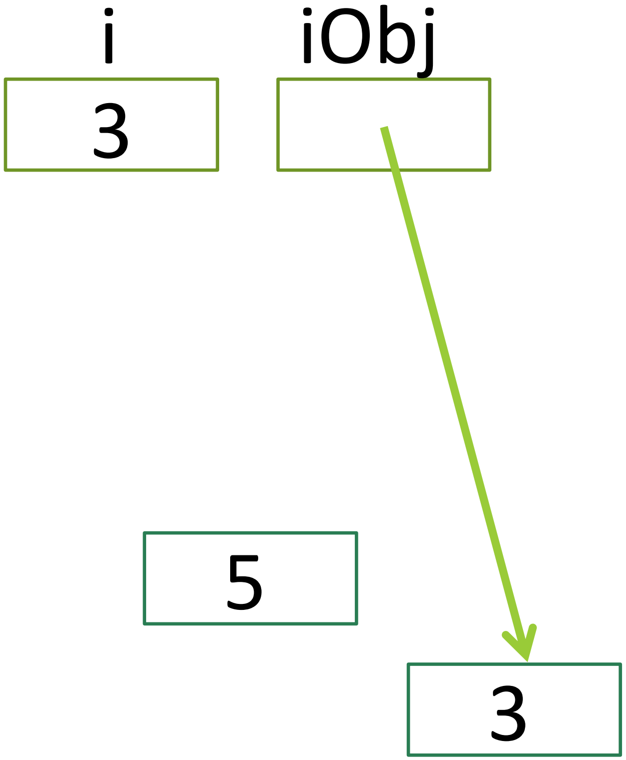
Primitives vs Wrappers (old way)

```
//Create variables
   int i;
   Integer iObj;
//Assign values
   i = 5;
   iObj = new Integer(5);
//Use the values
   i+1                  → 6
   iObj.intValue( )+1   → 6
//Change the values
   i=3;
   iObj = new Integer(3);
```



Primitives vs Wrappers (old way)

```
//Create variables
   int i;
   Integer iObj;
//Assign values
   i = 5;
   iObj = new Integer(5);
//Use the values
   i+1
   iObj.intValue( )+1
//Change the values
   i=3;
   iObj = new Integer(3);
```



The Integer object is immutable! Just like Strings!

All slides © their respective authors; redistribution is not permitted

Type casting

- Using wrappers that way is clumsy…

- Java (since SE5 – very old now) will convert freely between primitive types and their wrapper type

- If it is expecting an int value, and you use an Integer, it will “unbox” it (extract the value from it)

- If it is expecting an Integer value, and you use an int, it will “box” it (create an Integer with that value)

All slides © their respective authors; redistribution is not permitted

Type casting

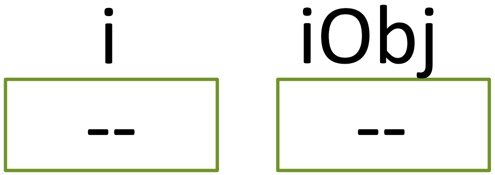
- Examples:

```
int x = new Integer(34); //silly, but legal
Integer y = 45;          //also legal.
x = y; //OK. It extracts its value.
y = x; //OK. It creates an object for you.
```

All slides © their respective authors; redistribution is not permitted

Primitives vs Wrappers (new way)

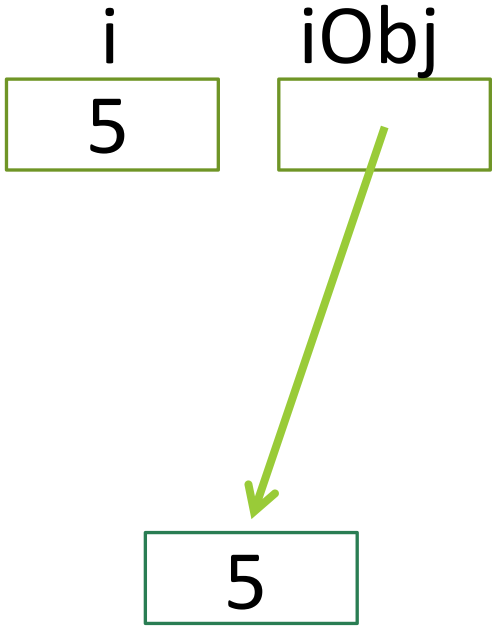
```
//Create variables
   int i;
   Integer iObj;
//Assign values
   i = 5;
   iObj = 5; //boxes
//Use the values
   i+1
   iObj+1 //unboxes
//Change the values
   i=3;
   iObj = 3; //boxes
```



All slides © their respective authors; redistribution is not permitted

Primitives vs Wrappers (new way)

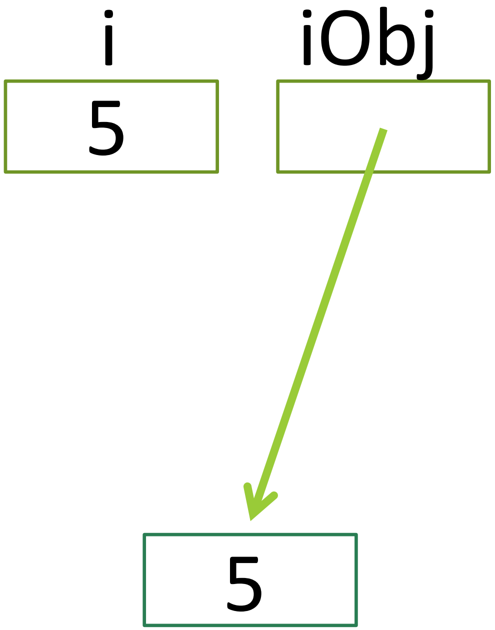
```
//Create variables
   int i;
   Integer iObj;
//Assign values
   i = 5;
   iObj = 5; //boxes
//Use the values
   i+1
   iObj+1 //unboxes
//Change the values
   i=3;
   iObj = 3; //boxes
```



All slides © their respective authors; redistribution is not permitted

Primitives vs Wrappers (new way)

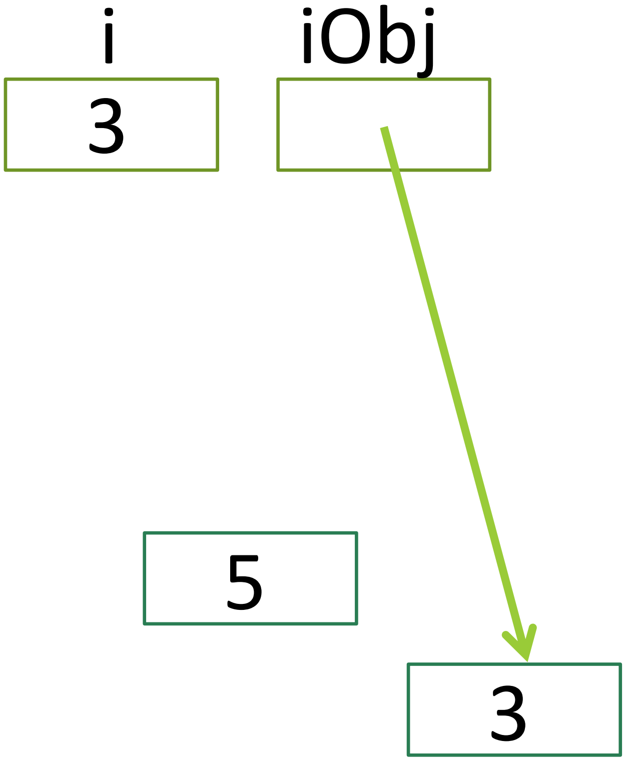
```
//Create variables
   int i;
   Integer iObj;
//Assign values
   i = 5;
   iObj = 5; //boxes
//Use the values
   i+1              → 6
   iObj+1 //unboxes → 6
//Change the values
   i=3;
   iObj = 3; //boxes
```



All slides © their respective authors; redistribution is not permitted

Primitives vs Wrappers (new way)

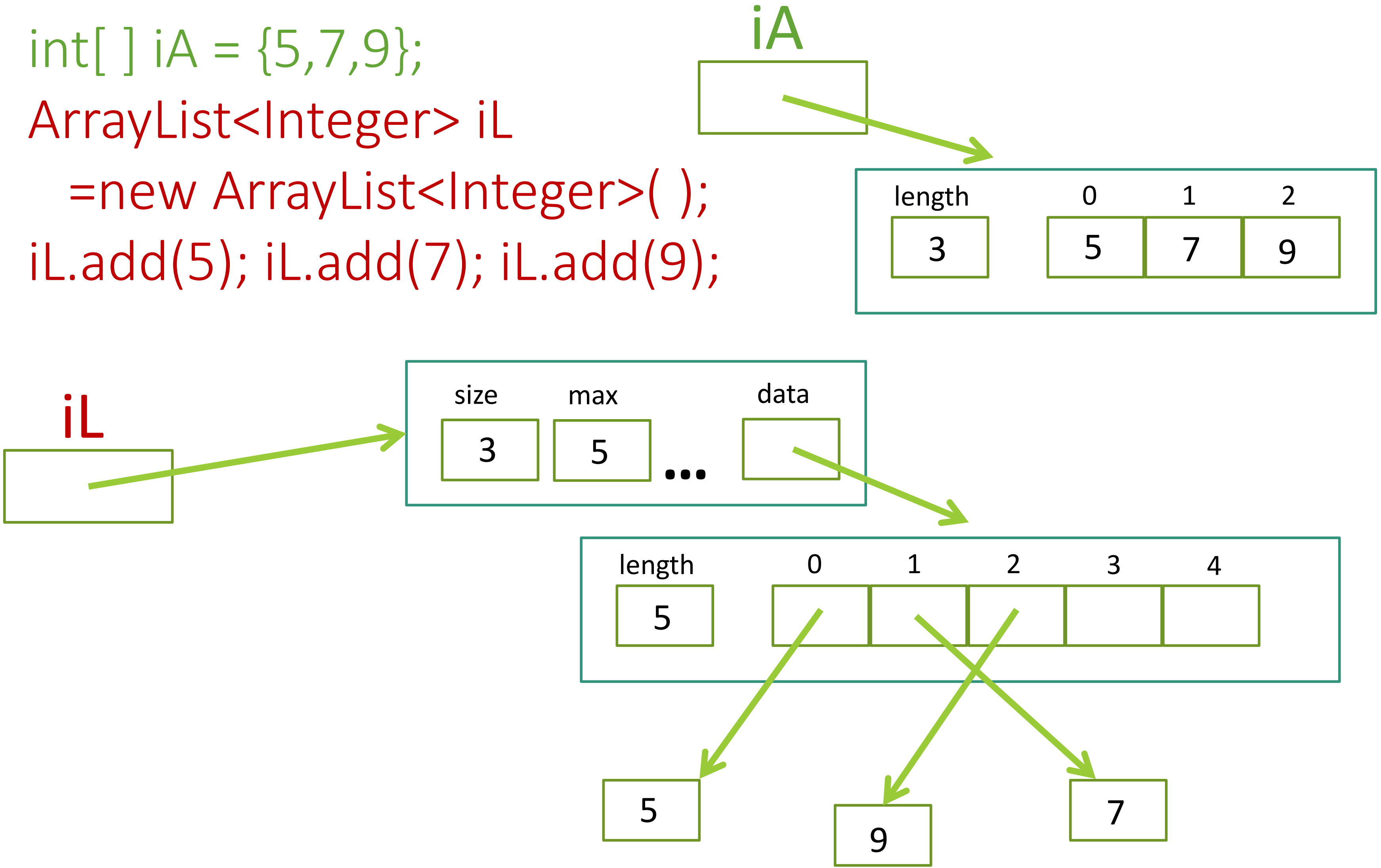
```
//Create variables
   int i;
   Integer iObj;
//Assign values
   i = 5;
   iObj = 5; //boxes
//Use the values
   i+1
   iObj+1 //unboxes
//Change the values
   i=3;
   iObj = 3; //boxes
```



All slides © their respective authors; redistribution is not permitted

int[ ] vs ArrayList<Integer>

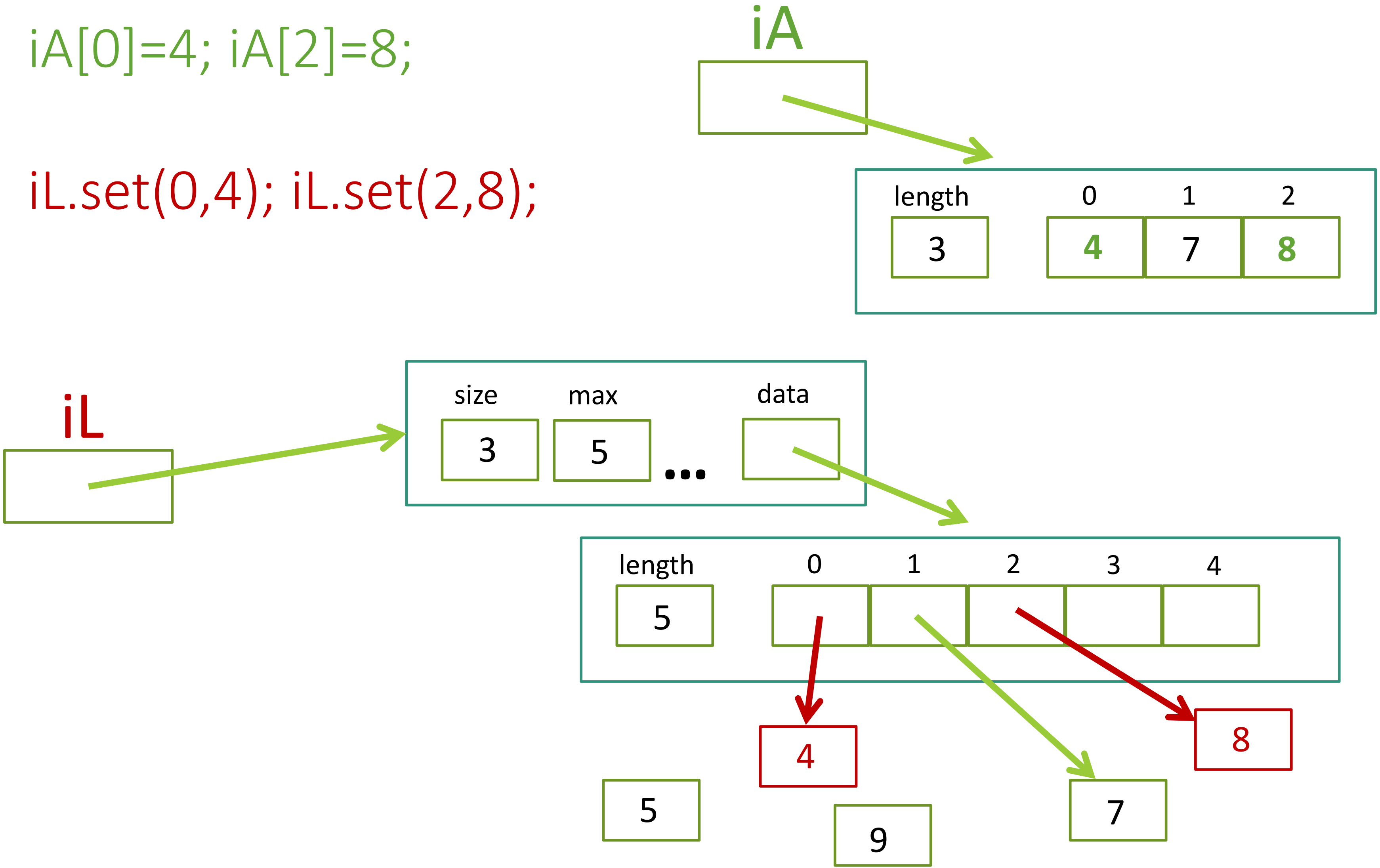
```
int[ ] iA = {5,7,9};
ArrayList<Integer> iL
  =new ArrayList<Integer>( );
iL.add(5); iL.add(7); iL.add(9);
```



int[ ] vs ArrayList<Integer>

iA[0]=4; iA[2]=8;

iL.set(0,4); iL.set(2,8);



int[] vs ArrayList<Integer>

- ArrayLists are convenient and give many useful methods to handle lists which freely grow and shrink

- But, as the last slide shows, there is internal complexity and a speed penalty

  - Usually OK, since modern processors are very fast

  - But in computationally intensive tasks, ordinary arrays would be preferred

All slides © their respective authors; redistribution is not permitted

ArrayLists of Objects

- An ArrayList<Object>, or just ArrayList, is flexible and powerful
  - It can store a list of any kind of data
  - With a mixture of types
  - This is how dynamic interpreted languages do just about everything

ArrayLists of Objects

- But...
- When you get( ) an element, you just get an Object
- You probably can't do anything with it until you (down)cast it to the correct type
- And you probably need to check instanceof before doing the cast, to do it safely

All slides © their respective authors; redistribution is not permitted

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

All slides © their respective authors; redistribution is not permitted

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

All slides © their respective authors; redistribution is not permitted

Cloning objects

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

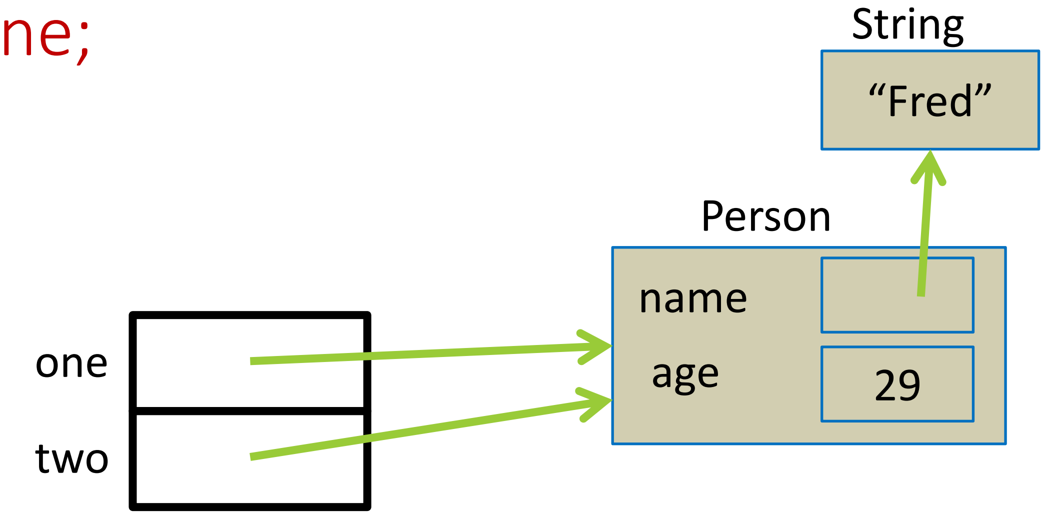
```
Person one, two;
one = new Person("Fred", 29);
two = one;
```

All slides © their respective authors; redistribution is not permitted

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

All slides © their respective authors; redistribution is not permitted

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;
}
```

All slides © their respective authors; redistribution is not permitted

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

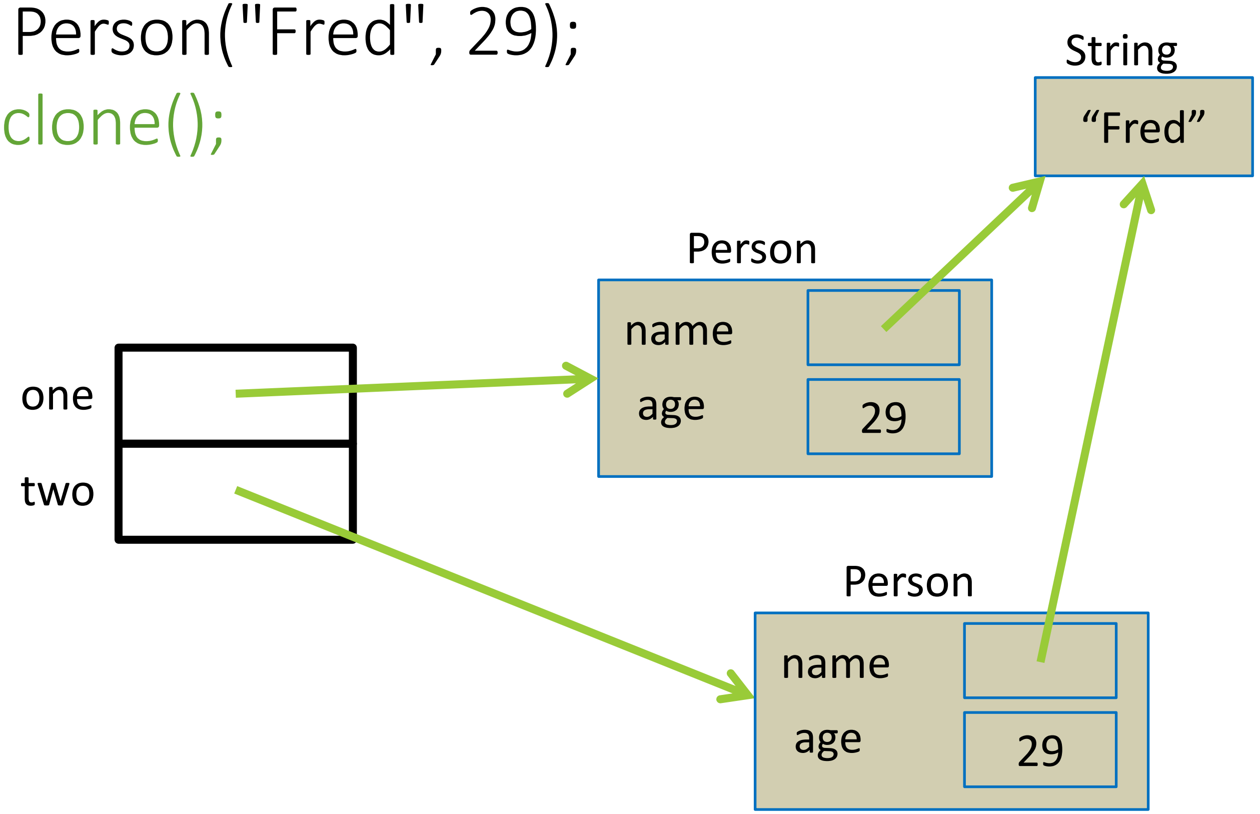
All slides © their respective authors; redistribution is not permitted

Cloning objects

- Now if we did:

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

- we'd get:



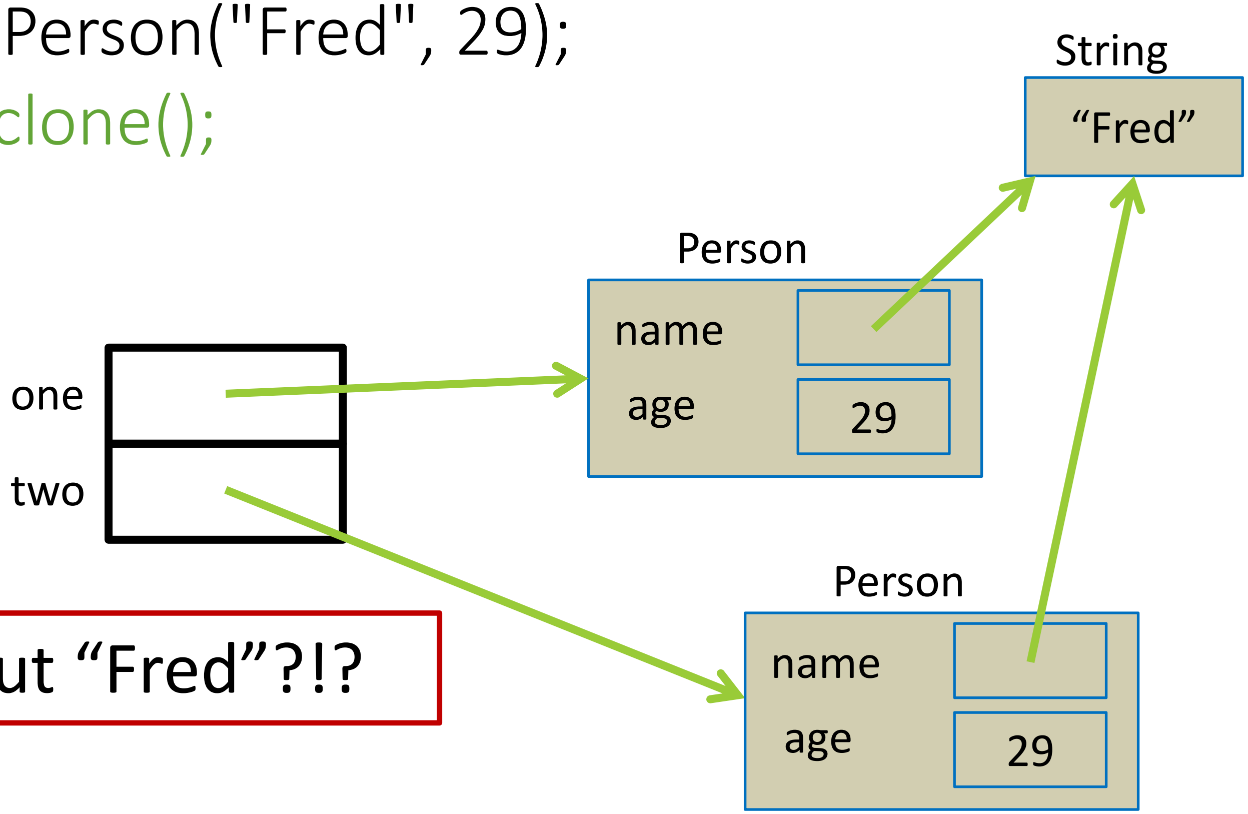
All slides © their respective authors; redistribution is not permitted

Cloning objects

- Now if we did:

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

- we'd get:



Wait! What about “Fred”?!?

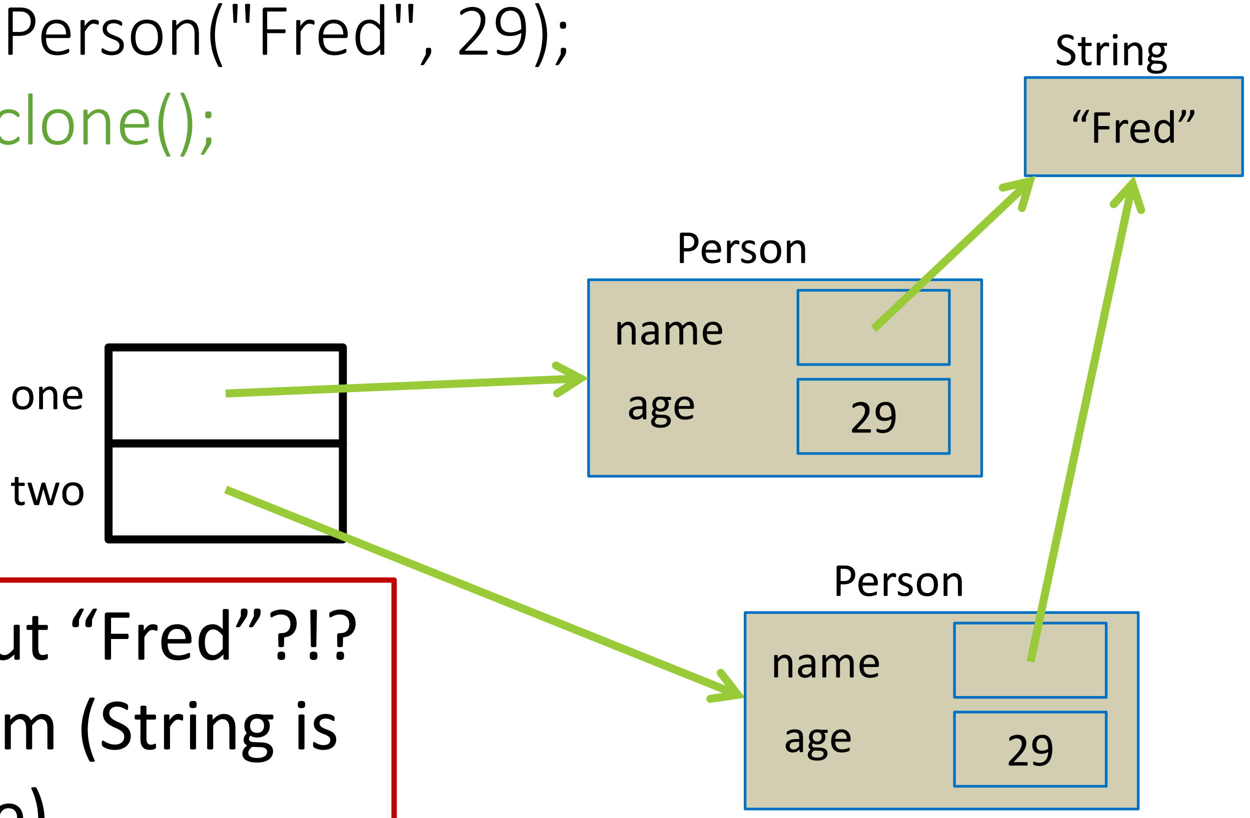
All slides © their respective authors; redistribution is not permitted

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)

All slides © their respective authors; redistribution is not permitted

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

All slides © their respective authors; redistribution is not permitted

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)

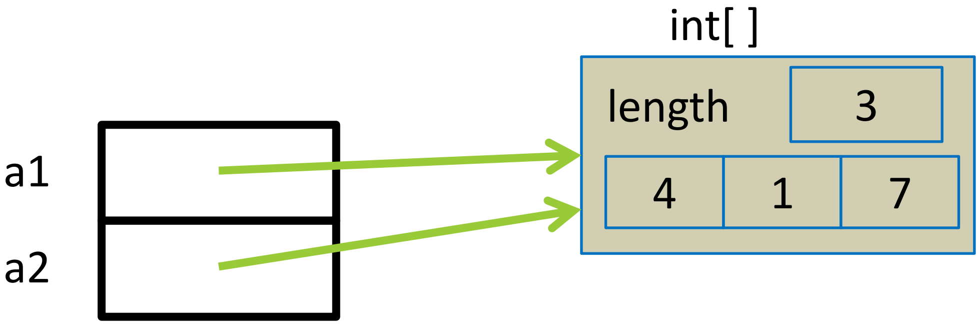
- Neither one is right or wrong, depends on what you need: use the one that does what you want it to do

All slides © their respective authors; redistribution is not permitted

What about arrays?

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

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



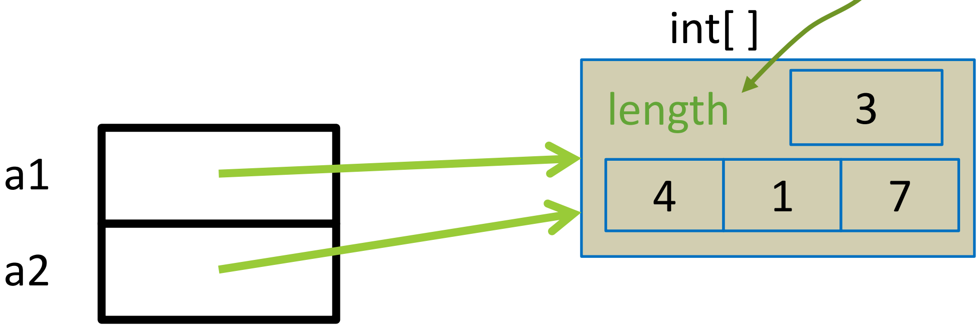
All slides © their respective authors; redistribution is not permitted

What about arrays?

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

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

Yes, length is an instance variable! That’s why you just use .length without () to get the size of an array!



All slides © their respective authors; redistribution is not permitted

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. */
```

All slides © their respective authors; redistribution is not permitted

System.arraycopy()

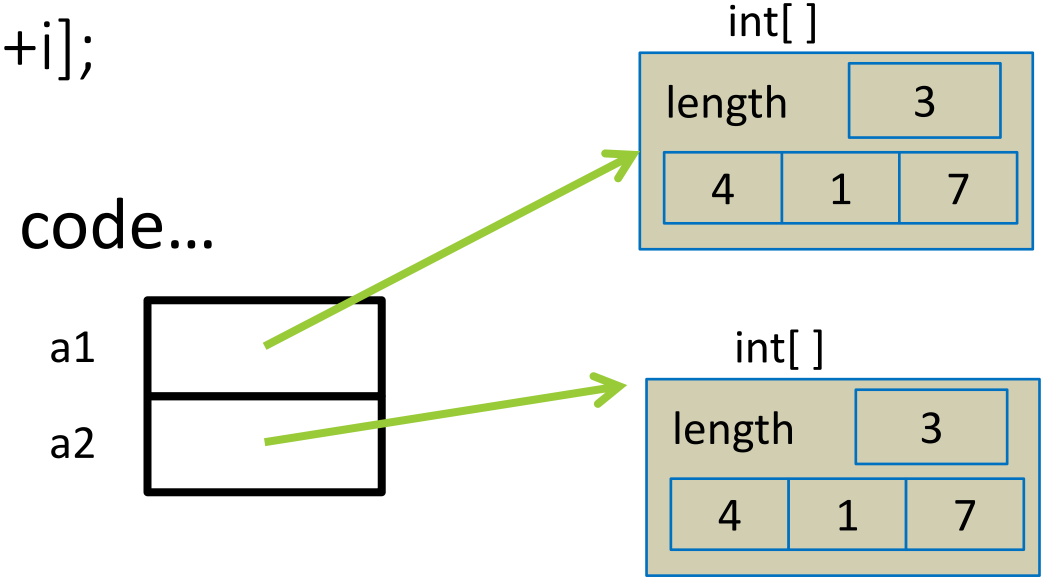
- The method call

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

- is the same as

```
for(int i=0; i<n; i++)
     a2[p2+i] = a1[p1+i];
```

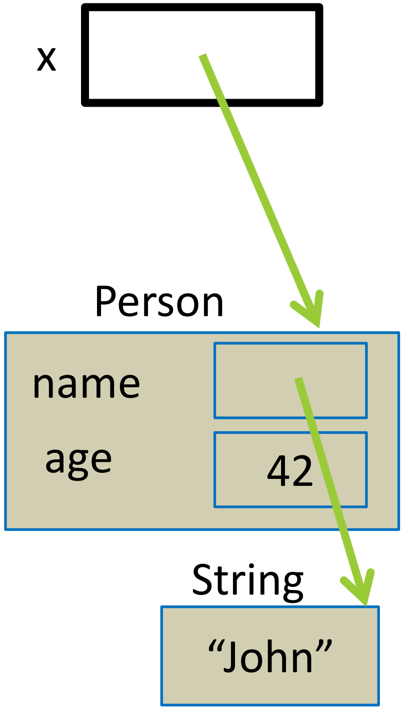
- It doesn't save much code…



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"

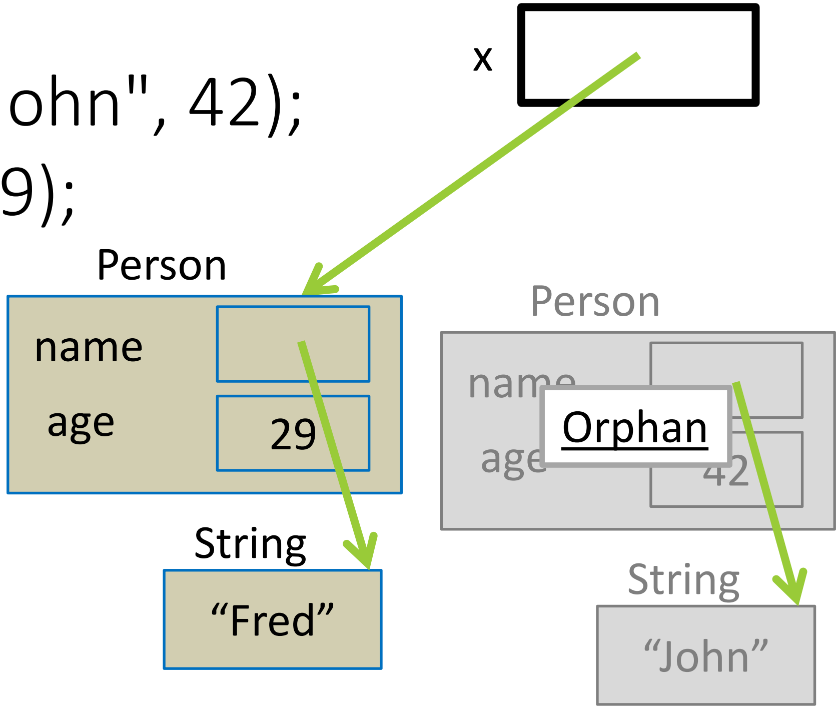
```
Person x = new Person("John", 42);
```



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);
```

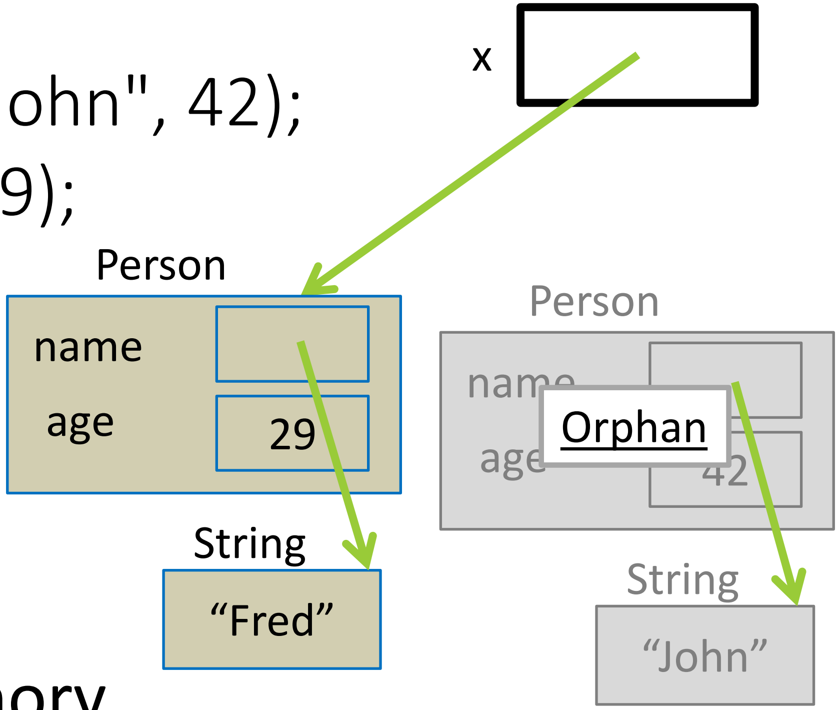


All slides © their respective authors; redistribution is not permitted

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

All slides © their respective authors; redistribution is not permitted

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

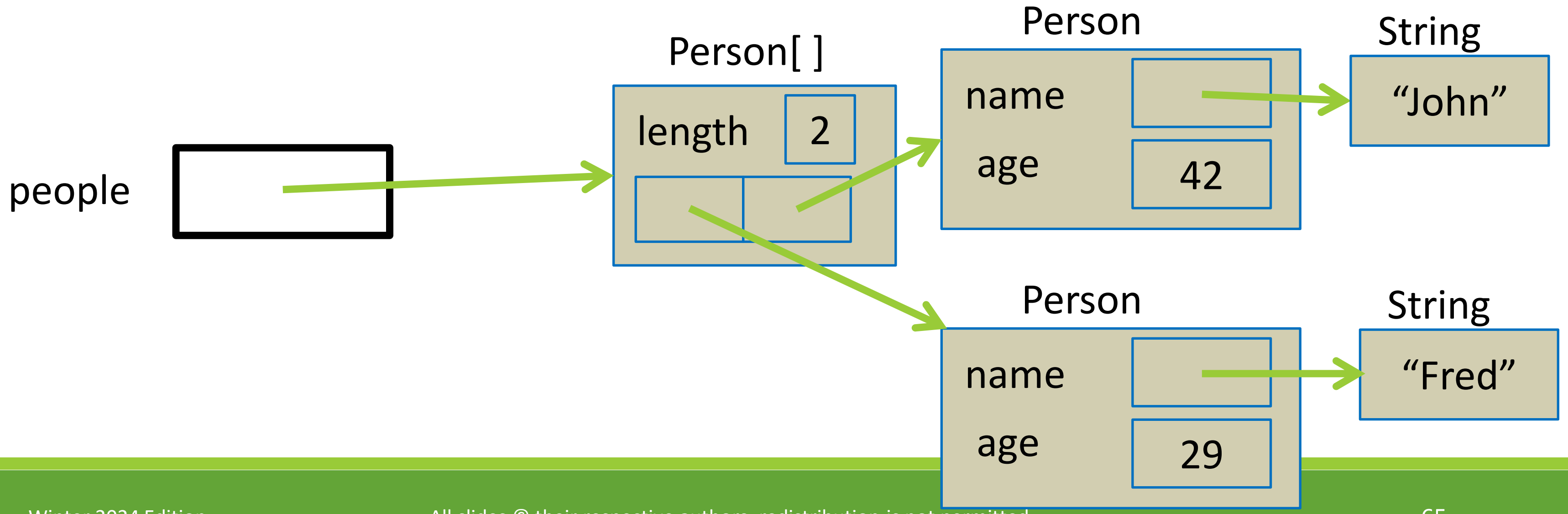
- Then what about an array of objects that contain references to other objects which contain arrays...?
  - The principles are the same
  - If every level in this situation does something correct and sensible, then the whole thing will work reliably
  - You might want shallow copies, you might want deep copies → every situation is different
    - Think! Plan on paper before implementing!

All slides © their respective authors; redistribution is not permitted

Array of Person objects

- Make an array of Person objects:

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



All slides © their respective authors; redistribution is not permitted

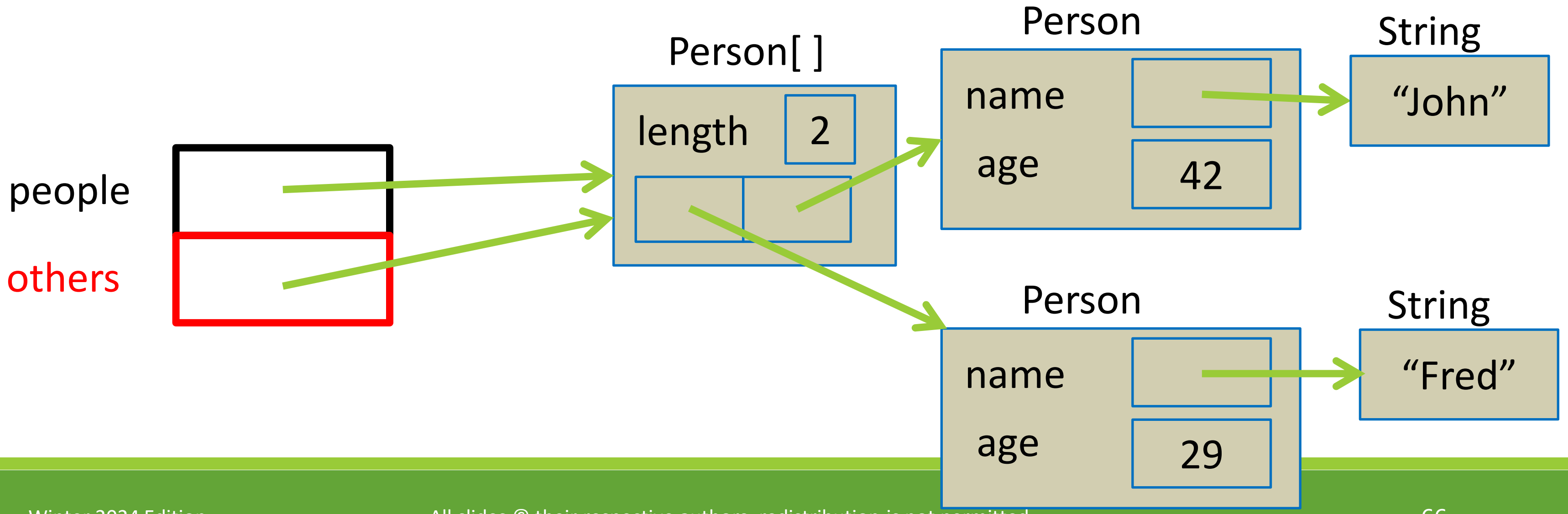
Array of Person objects

- 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;
```

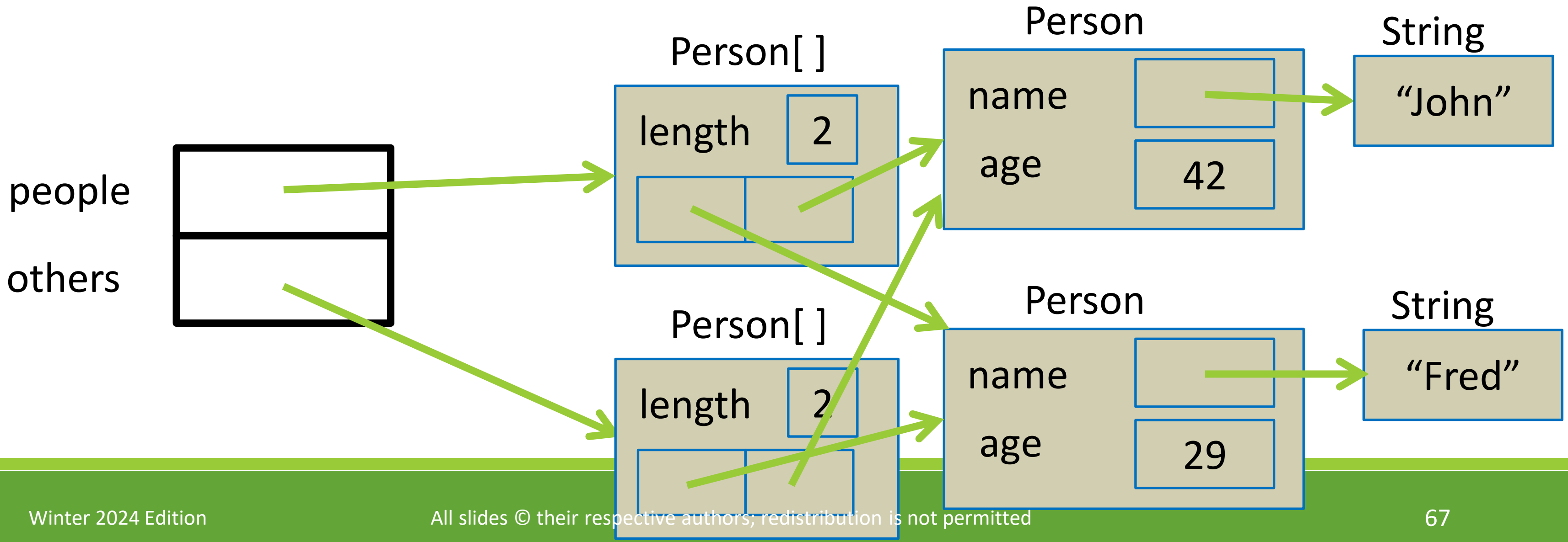


All slides © their respective authors; redistribution is not permitted

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);
```



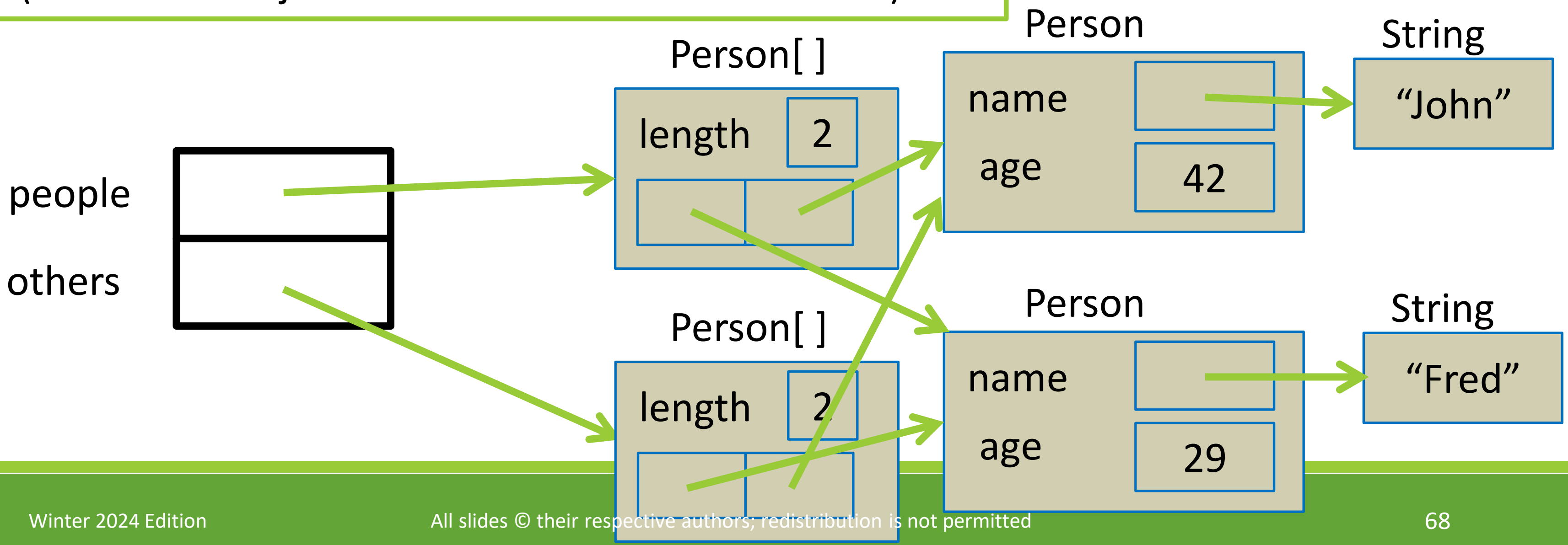
All slides © their respective authors; redistribution is not permitted

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!)



All slides © their respective authors; redistribution is not permitted

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)

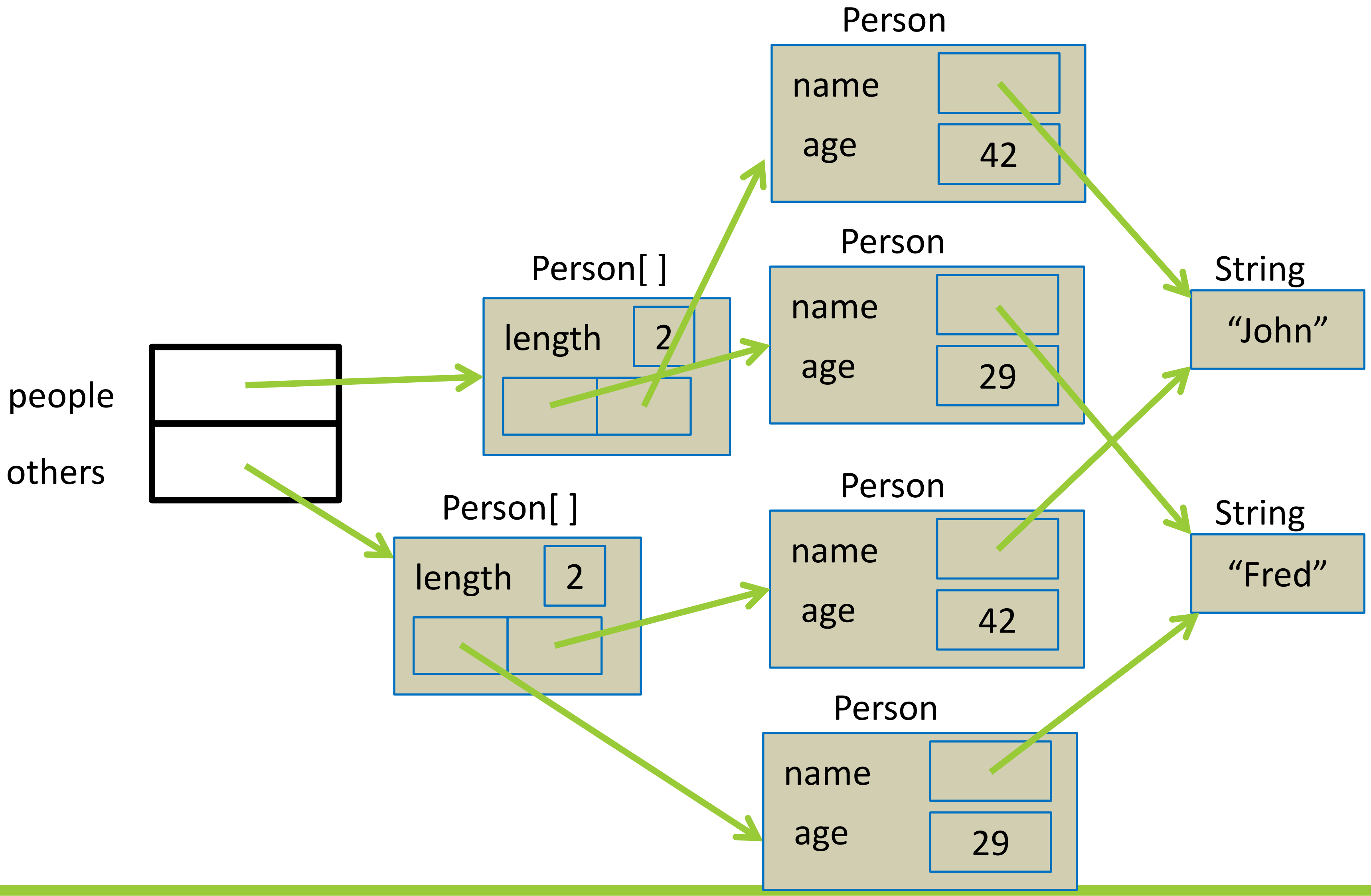
- We'll need to write our own for loop this time:

  ```
  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

All slides © their respective authors; redistribution is not permitted

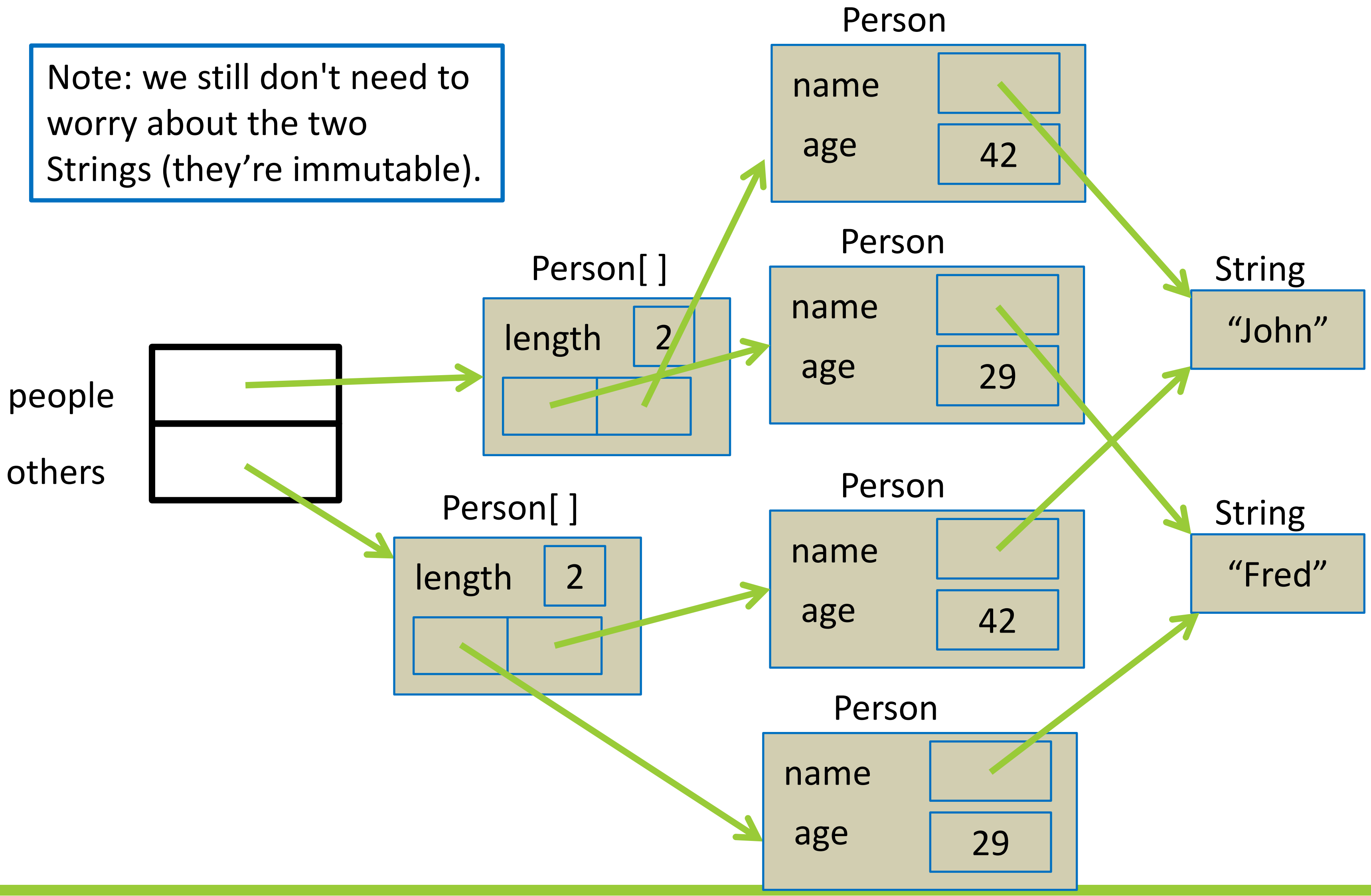
Results of a “deep copy”



All slides © their respective authors; redistribution is not permitted

Results of a “deep copy”

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



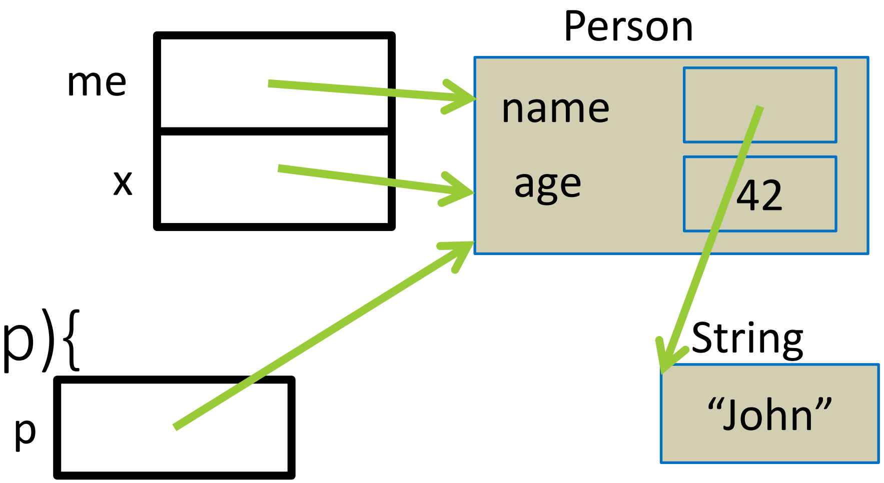
All slides © their respective authors; redistribution is not permitted

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){
    ...
}
```



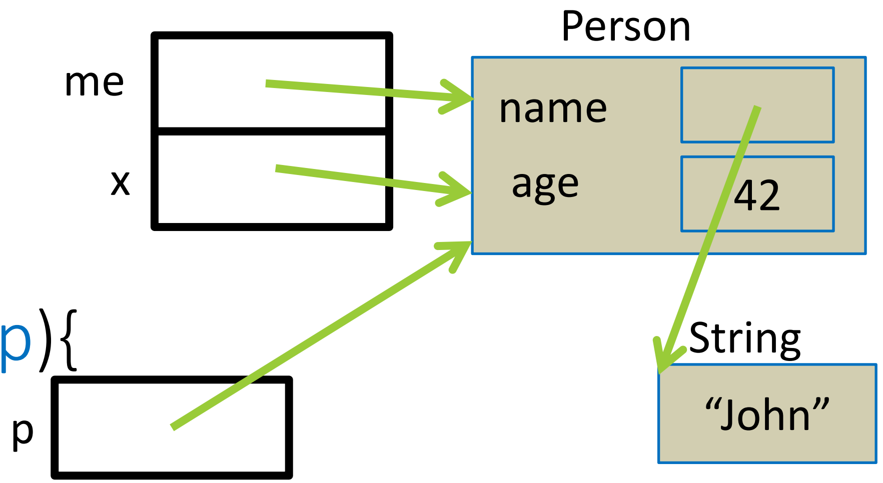
All slides © their respective authors; redistribution is not permitted

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



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

All slides © their respective authors; redistribution is not permitted

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.
```

All slides © their respective authors; redistribution is not permitted

New methods are necessary

- A new constructor would be useful:

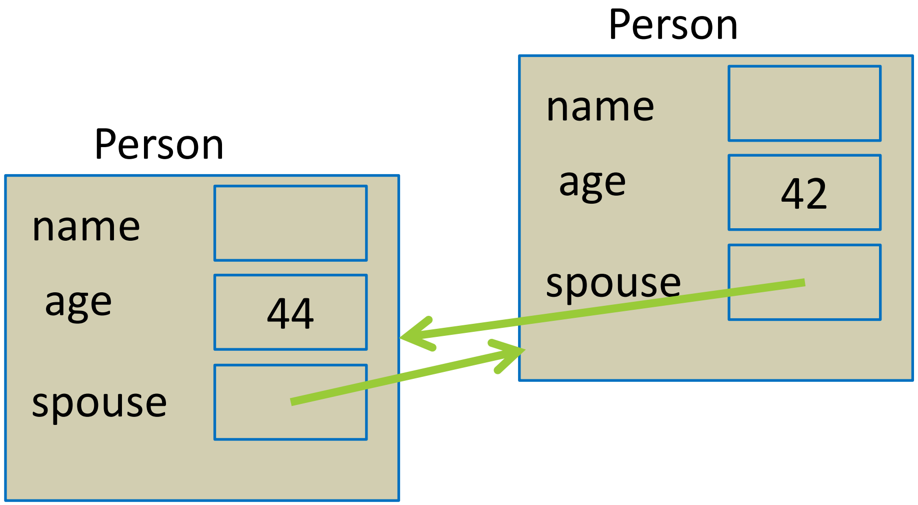
```
public Person(String who, int currentAge, Person otherHalf)
{
    name = who;
    age = currentAge;
    spouse = otherHalf;
    //make sure the other person is married, too!
    if(otherHalf != null)
        otherHalf.spouse = this;
    population++;
}//constructor
```

All slides © their respective authors; redistribution is not permitted

New methods are necessary

- How about new instance methods as well:

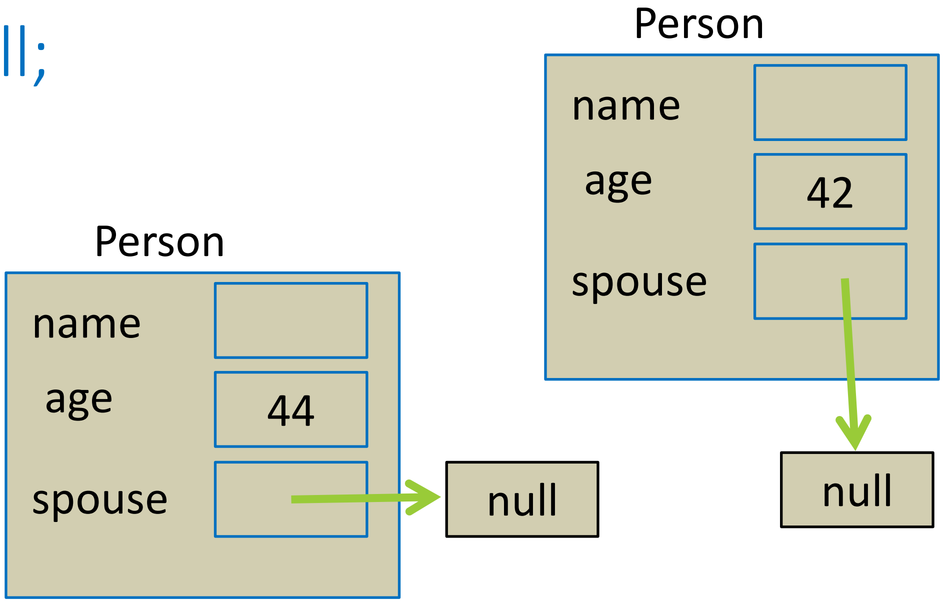
```
public void marries(Person other) {
     spouse = other;
     if (other != null)
          other.spouse = this;
}//marries
```



New methods are necessary

- How about new instance methods as well:

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



All slides © their respective authors; redistribution is not permitted

New methods are necessary

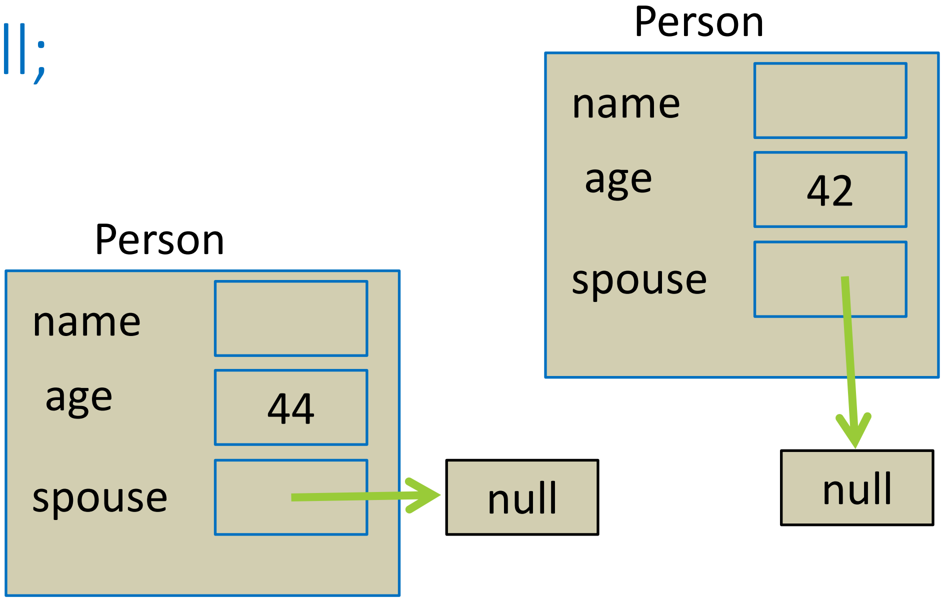
- How about new instance methods as well:

```
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!



All slides © their respective authors; redistribution is not permitted

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;
}
```

All slides © their respective authors; redistribution is not permitted

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
```

All slides © their respective authors; redistribution is not permitted

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

All slides © their respective authors; redistribution is not permitted

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;
}
```

All slides © their respective authors; redistribution is not permitted

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!

All slides © their respective authors; redistribution is not permitted

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();
}
```

All slides © their respective authors; redistribution is not permitted

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.

All slides © their respective authors; redistribution is not permitted

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…)

All slides © their respective authors; redistribution is not permitted

What’s our best strategy?

- There should be a separate PersonList class, which will handle all these operations

- A class whose primary role is to store a bunch of other objects is sometimes known as a collection class

All slides © their respective authors; redistribution is not permitted

What’s our best strategy?

- Write a PersonList class with:
  - A “partially-filled array” of Person
    - Use a generous fixed size
    - Or use an ArrayList!
  - A constructor to make an empty list
  - Methods addPerson and toString

All slides © their respective authors; redistribution is not permitted

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!

All slides © their respective authors; redistribution is not permitted

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)?

All slides © their respective authors; redistribution is not permitted

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.

All slides © their respective authors; redistribution is not permitted

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)

All slides © their respective authors; redistribution is not permitted