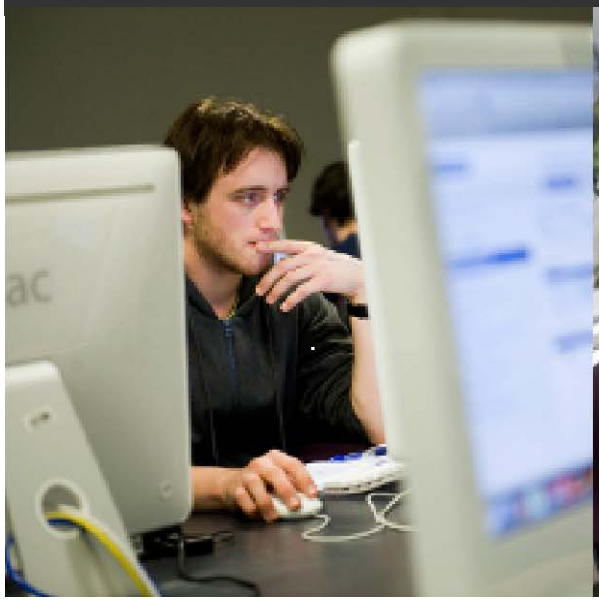
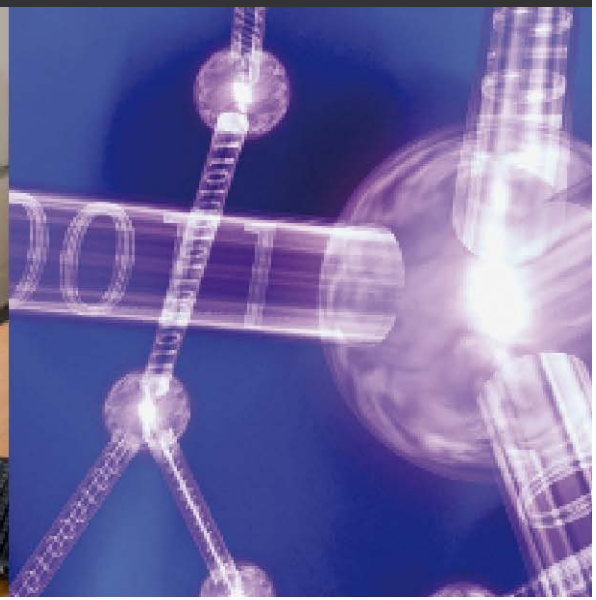
MONASH University

Information Technology

Module 5:
Understanding Object References & Relationships

FIT2034 Computer Programming 2
Faculty of Information Technology



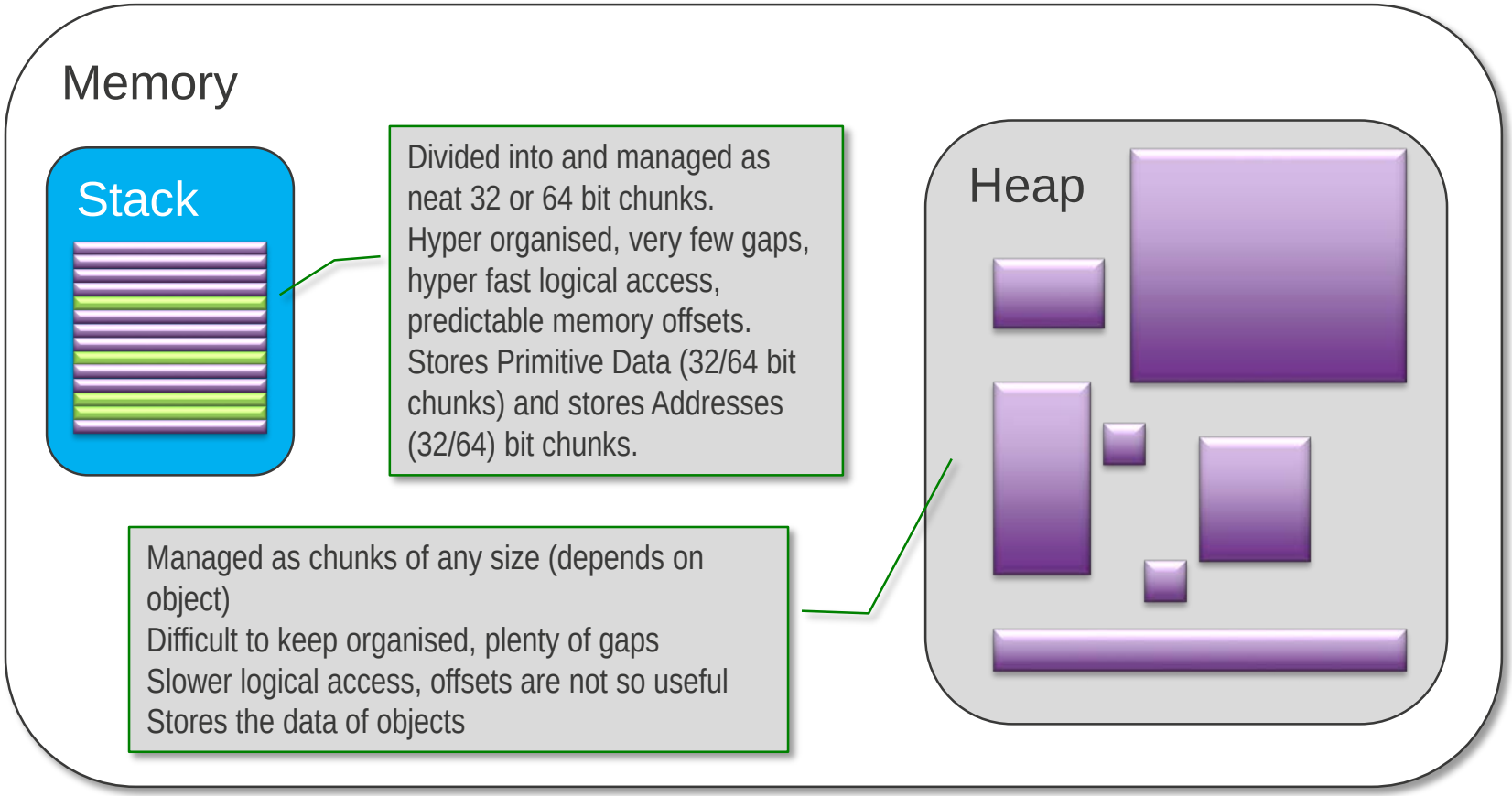


Part 1:
Object References

Why Reference Variables? – A Reminder

- Primitive type data and Address data
  - Is small and of a few predictable sizes
- Reference type data
  - Can be of any size including large

This requires 2 very different memory management strategies if memory management is to be fast and space efficient



Reference Variables as Instance Variables

- So far, all the instance variables of classes have been of primitive types
  - e.g. int, float, double, boolean, …
  - Except String which we know often behaves like a primitive data type
- Instance variables can also be of a Reference type (be an object reference)
  - i.e. they can reference an instance of some class
- Gives rise to class relationships
  - Notice that this creates a relationship between two classes (explained in more detail later today)
- Examples - See next slide

```
public class Person {
    private String name;
    private int age;
    private Car vehicle;
}
```

reference
primitive
reference

Class Relationship:
A Person has a Car

```
public class Car {
    private String type;
    private int buildYear;
    private Date regoDate;
}
```

reference
primitive
reference

Class Relationship:
A Car has a Date when it was registered (its registration date)

```
public class Date {
    private int day;
    private int month;
    private int year;
}
```

primitive
primitive
primitive

Passing Reference Variables

- Passing Reference variables as Method Parameters
  - Works EXACTLY the same way as passing variables of Primitive data types, i.e.
    - At call time, the actual parameter's value is evaluated
      - In this case the value of a single reference variable
    - This value is copied to the corresponding formal parameter
  - For Reference Variables the difference is:
    - The "evaluated value" is an ADDRESS not a value
    - The actual and formal parameter are both the same Reference Type (not the same primitive data type)
- In summary
  - Primitive data type parameters pass values
  - Reference type parameters pass addresses
    - i.e. a new alias for an object in the calling method, is created in the called method giving access to the object's data from the called method!!!

or an expression that evaluates to an address that points at an object of the appropriate type

Passing Reference Variables - Example

- Consider the following class code and driver code

```
public class Person {
  private String name;
  private int age;
  private Car vehicle;

  public Person() {
    age = 20;
    name = "nobody";
    vehicle = null;
  }

  public int getAge(){
    return age;
  }

  public void setAge(int newAge){
    if (newAge > 0)
      age = newAge;
  }
}
```

reference
primitive
reference

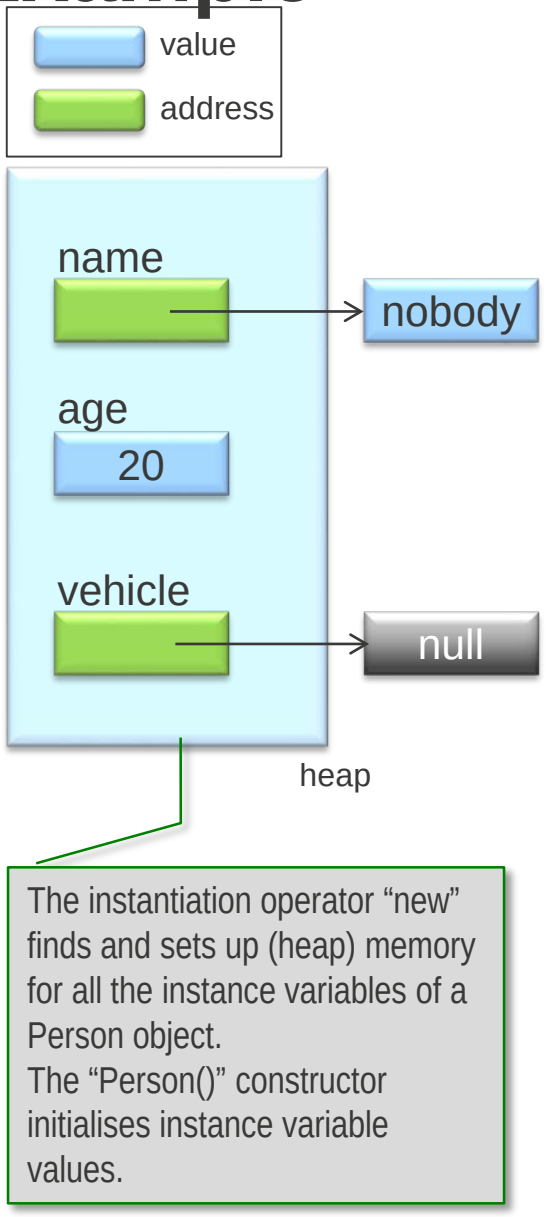
Person Class

Driver code outside of Person Class

```
public class PersonDriver {

  public static void main(String[] args) {
    Person p = new Person() ;
    System.out.println(personsAge(p));
  }

  public static int personsAge(Person p1) {
    int result = p1.getAge();
    return result;
  }
}
```

Passing Reference Variables - Example

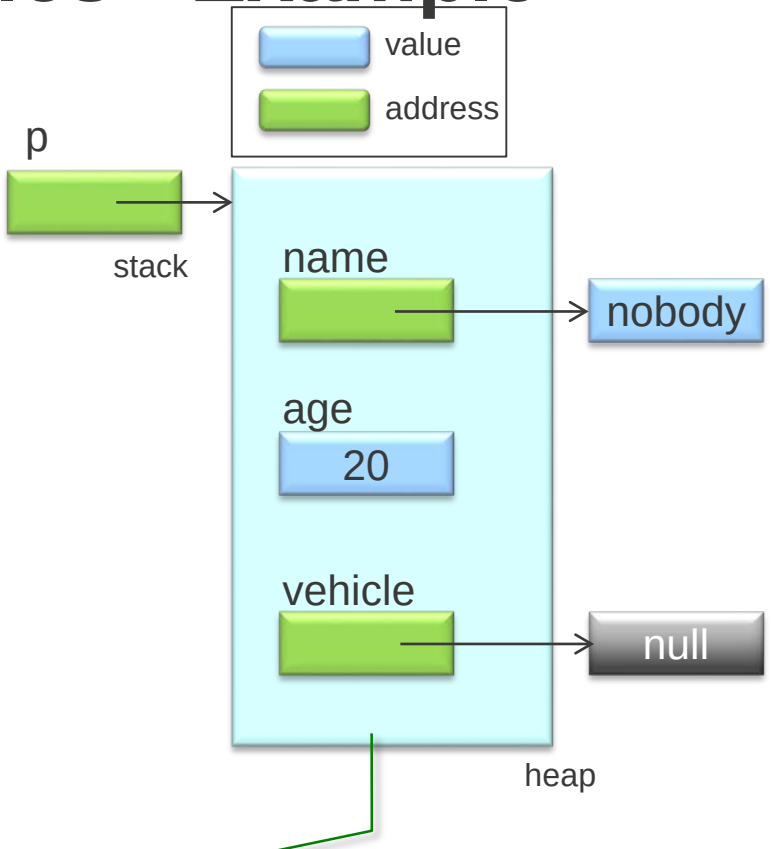
```
public class PersonDriver {

    public static void main(String[] args) {
        Person p = new Person();
        System.out.println(personsAge(p));
    }

    public static int personsAge(Person p1) {
        int result = p1.getAge();
        return result;
    }
}
```



- Let’s analyse the driver code statement-by-statement

Passing Reference Variables - Example

```
public class PersonDriver {

    public static void main(String[] args) {
        Person p = new Person();
        System.out.println(personsAge(p));
    }

    public static int personsAge(Person p1) {
        int result = p1.getAge();
        return result;
    }
}
```

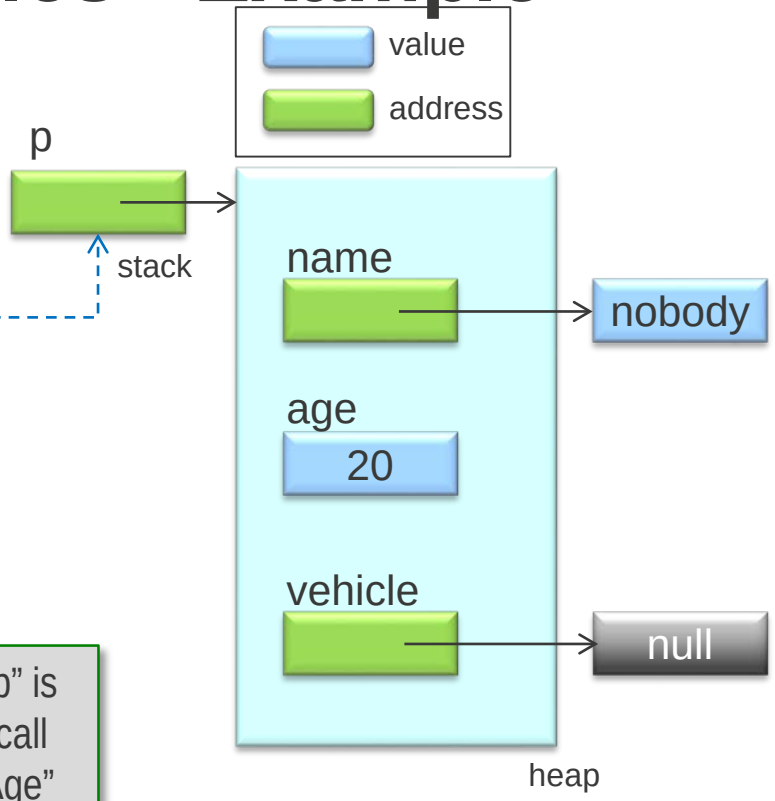


"new" returns the (heap) address of the new Person object which is assigned as the value of a new (stack) reference variable called "p".

Passing Reference Variables - Example

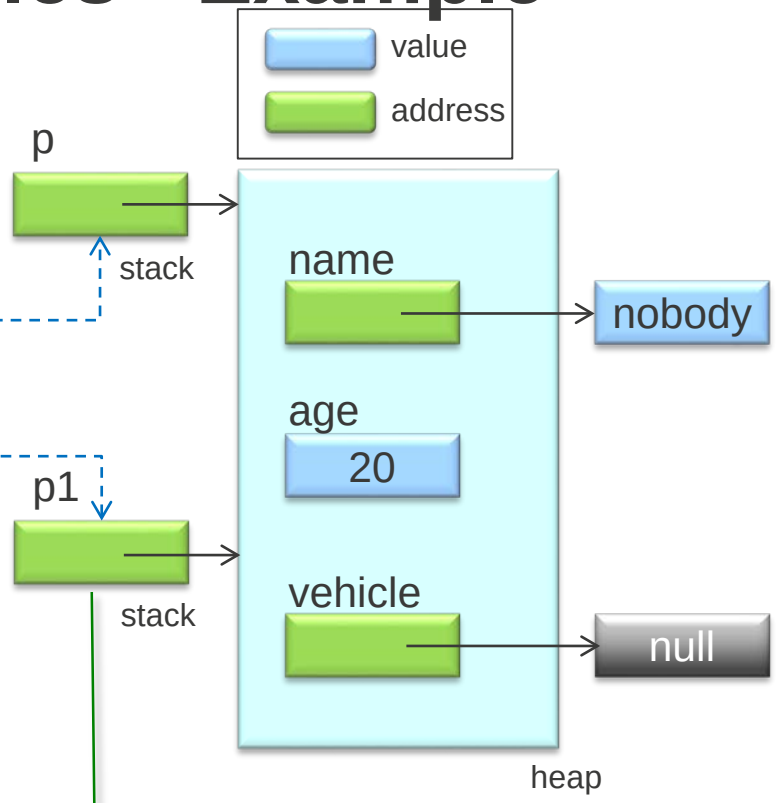
```
public class PersonDriver {

    public static void main(String[] args) {
        Person p = new Person();
        System.out.println(personsAge(p));
    }

    public static int personsAge(Person p1) {
        int result = p1.getAge();
        return result;
    }
}
```

The reference variable “p” is an actual argument in a call to the method “personsAge”



Passing Reference Variables - Example

```
public class PersonDriver {

    public static void main(String[] args) {
        Person p = new Person();
        System.out.println(personsAge(p));
    }

    public static int personsAge(Person p1) {
        int result = p1.getAge();
        return result;
    }
}
```

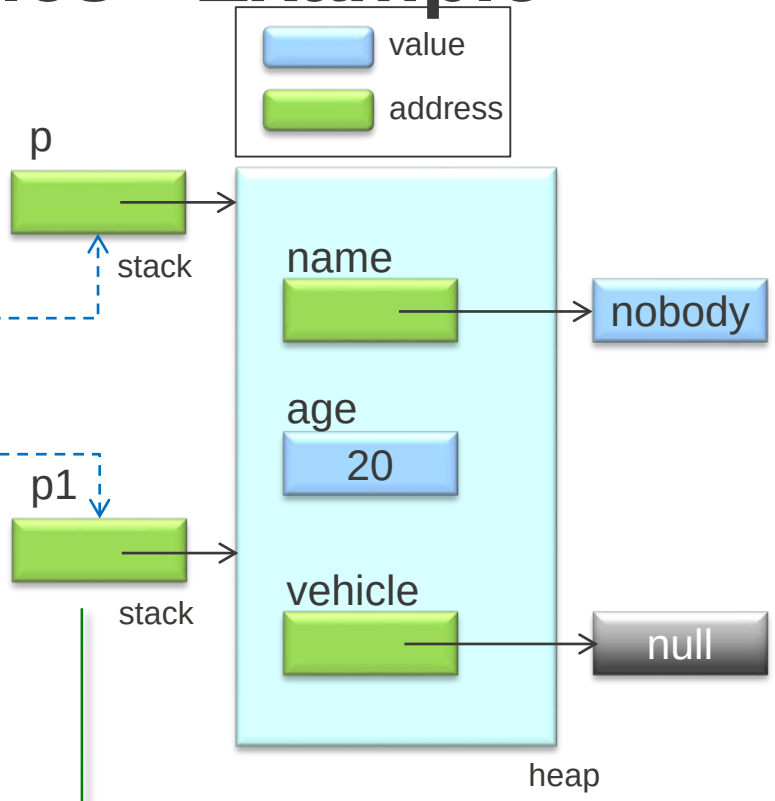


The usual parameter passing mechanism takes place i.e. a copy of the value of the actual parameter is used to initialise the corresponding formal parameter.
The difference here is that both parameters are reference variables and the value passed is an address (because the value of a reference variable is an address).

p1 is a formal parameter of the personsAge method and therefore behaves like a local variable. Therefore this reference variable is created in (stack) memory immediately before the personsAge method begins execution

Passing Reference Variables - Example

```
public class PersonDriver {

    public static void main(String[] args) {
        Person p = new Person();
        System.out.println(personsAge(p));
    }

    public static int personsAge(Person p1) {
        int result = p1.getAge();
        return result;
    }
}
```



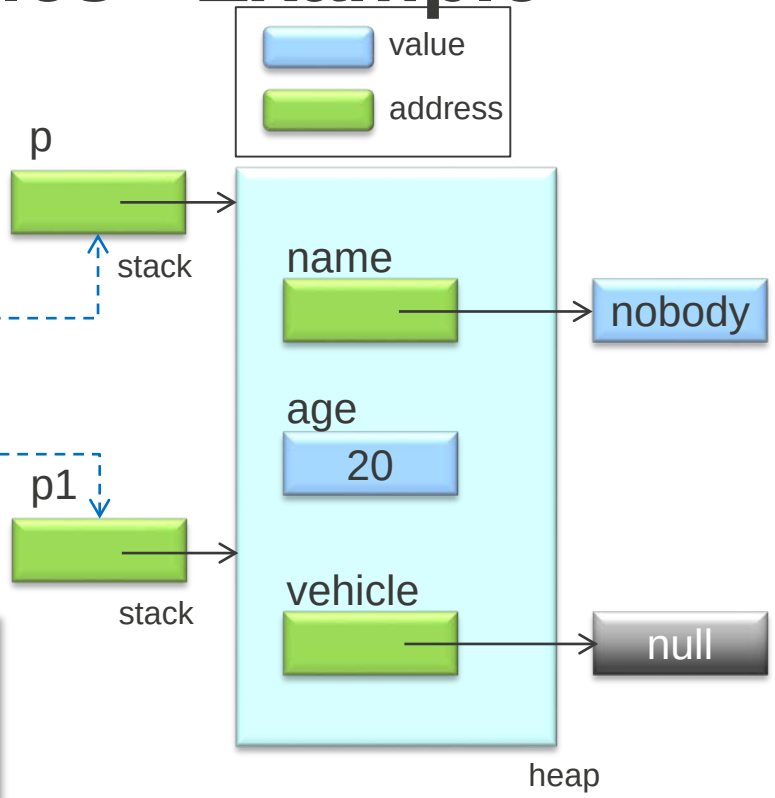
"p" and "p1" are synonyms or aliases. They both references the same object in (heap) memory.

Therefore the Accessor calls p1.getAge() and p.getAge() are the same call which returns the integer value 20.

Passing Reference Variables - Example

```
public class PersonDriver {

    public static void main(String[] args) {
        Person p = new Person();
        System.out.println(personsAge(p));
    }

    public static int personsAge(Person p1) {
        int result = p1.getAge() 20;
        return result;
    }
}
```

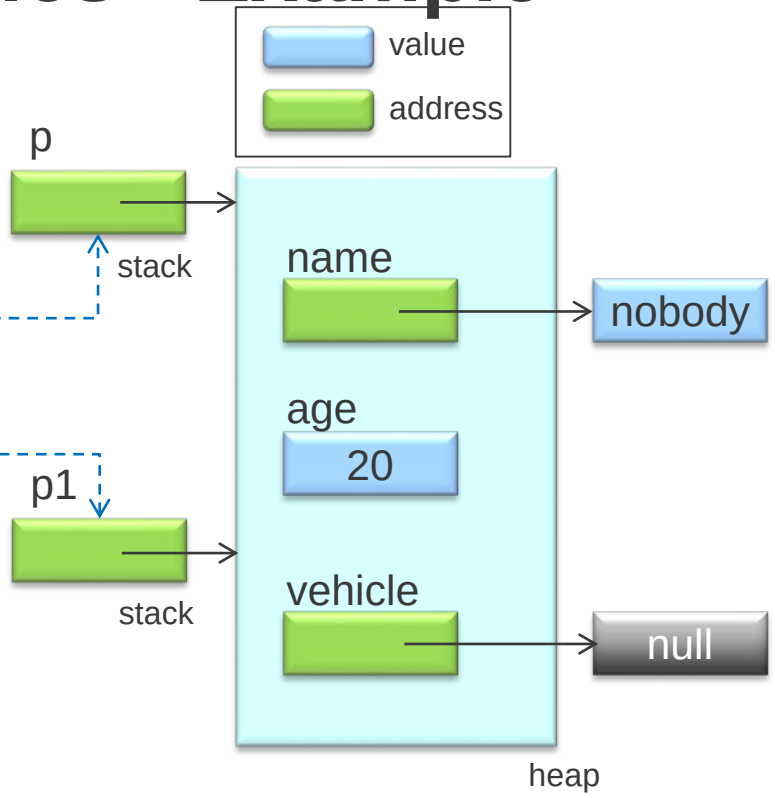
Accessor Method call p1.getAge() is replaced by its return value



Passing Reference Variables - Example

```
public class PersonDriver {

    public static void main(String[] args) {
        Person p = new Person();
        System.out.println(personsAge(p));
    }

    public static int personsAge(Person p1) {
        int result = 20;
        return result 20;
    }
}
```

Accessor's return value becomes personsAge method's return value.

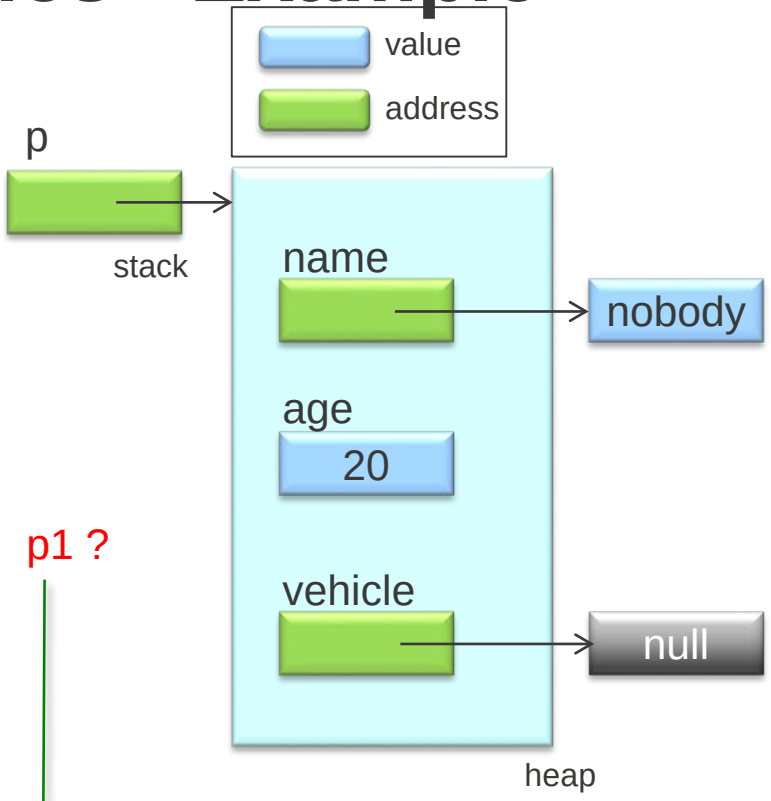


Passing Reference Variables - Example

```
public class PersonDriver {

    public static void main(String[] args) {
        Person p = new Person();
        System.out.println(personsAge(p) 20);
    }

    public static int personsAge(Person p1) {
        int result = 20;
        return 20;
    }
}
```

20

personsAge method call is replaced by its return value.



p1 ?

p1 is a formal parameter of the personsAge method and therefore behaves like a local variable of this method. Therefore since personsAge has finished executing p1 is erased from memory.

Passing null - Be careful!

- When you write methods with formal parameters of reference types
  - You should validate that any passed reference is not null before using it in any way, e.g.

```
p != null        p == null
```

- Recall
  - When a reference variable has the value null it means no object is currently being referenced (pointed at) by it
  - Invoking methods on a reference variable with value null will cause a run-time error for obvious reasons
    - So called NullPointerException

Object Reference Parameters – Side Effects

- Consider the following class code and driver code

```
public class Person {

    private String name;
    private int age;
    private Car vehicle;

    public Person() {
        age = 20;
        name = "nobody";
        vehicle = null;
    }

    public int getAge(){
        return age;
    }

    public void setAge(int newAge){
        if (newAge > 0)
            age = newAge;
    }
}
```

reference
primitive
reference

Person Class

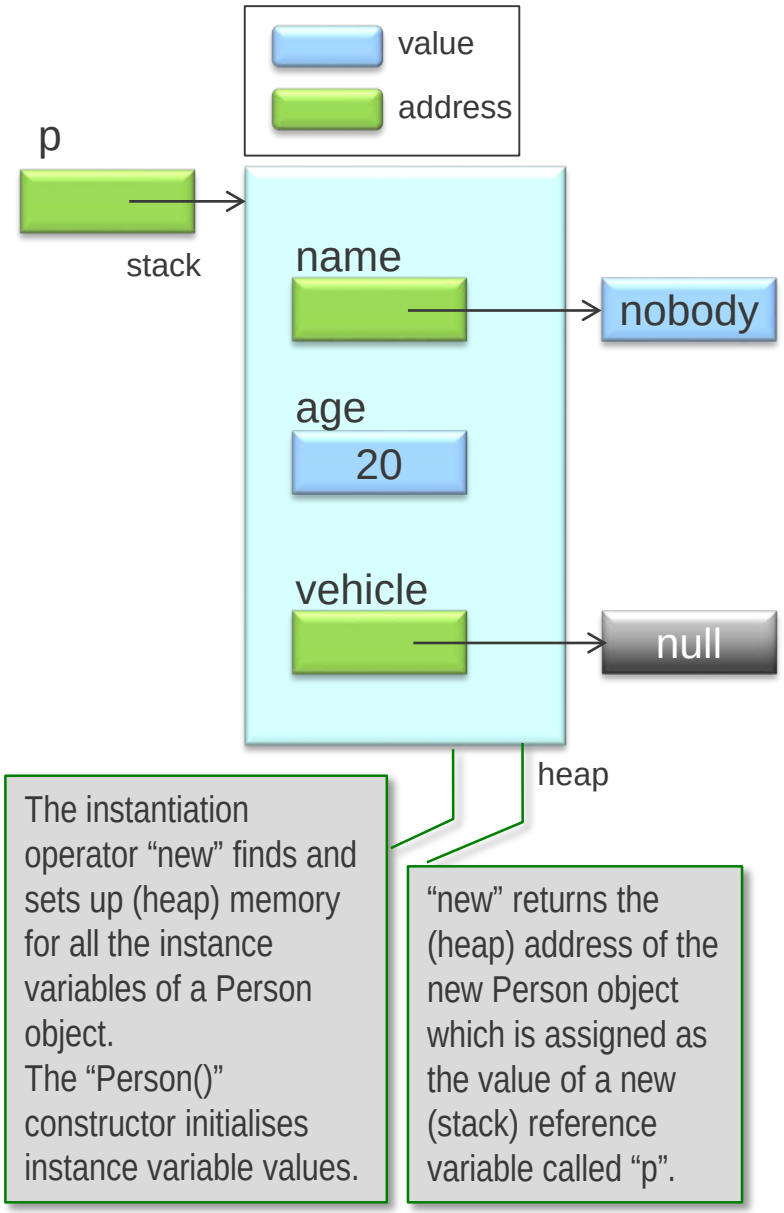
Driver code outside of Person Class

```
public class PersonDriver2{

    public static void main(String[] args){
        Person p = new Person();

        System.out.println(p.getAge());
        sideEffect(p);
        System.out.println(p.getAge());
    }

    public static void sideEffect(Person p1){
        p1.setAge(99);
    }
}
```

Unlike the previous example Driver code this actively sets (not passively gets) the value of an instance variable of the object referenced by p1. The question is does this have any (side) effect on the object referenced by p?

Object Reference Parameters – Side Effects

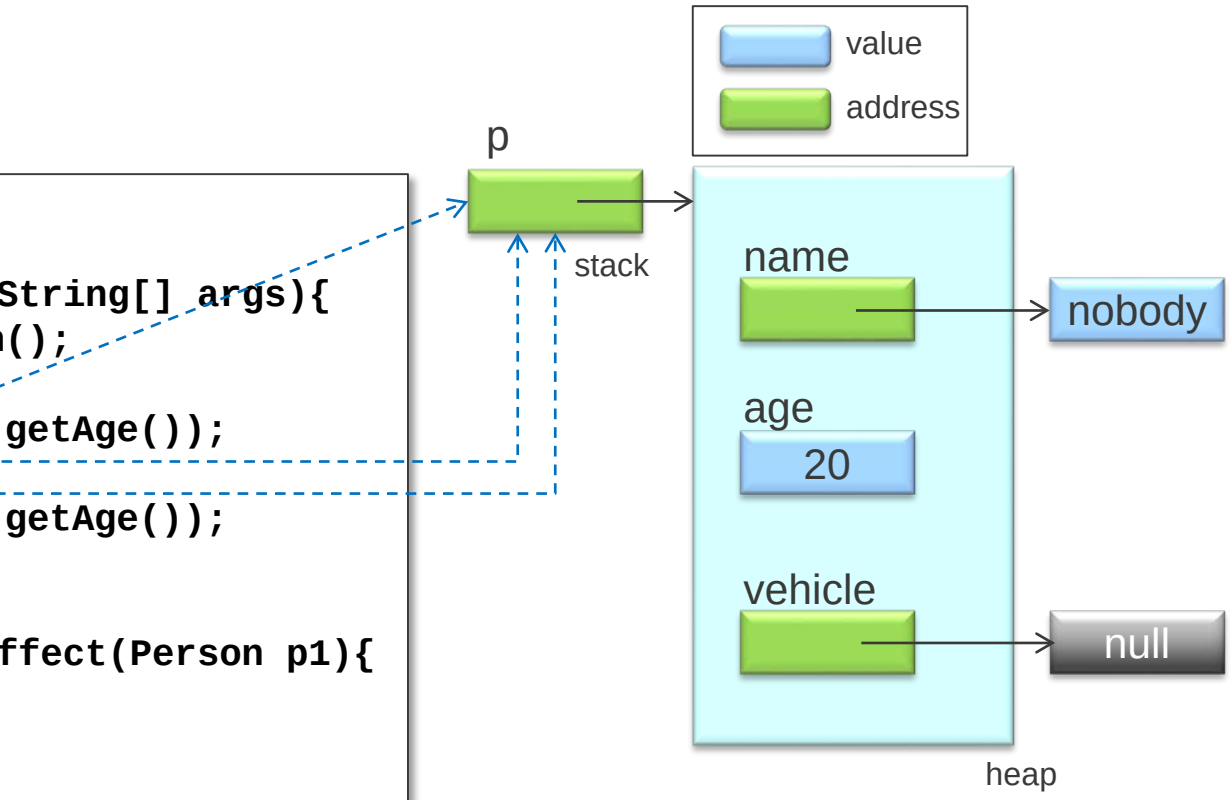
```
public class PersonDriver2{

    public static void main(String[] args){
        Person p = new Person();

        System.out.println(p.getAge());
        sideEffect(p);
        System.out.println(p.getAge());
    }

    public static void sideEffect(Person p1){
        p1.setAge(99);
    }
}
```



- In this analysis we will highlight the main points rather than take a statement-by-statement approach

Object Reference Parameters – Side Effects

```
public class PersonDriver2{

    public static void main(String[] args){
        Person p = new Person();

        System.out.println(p.getAge());
        sideEffect(p);
        System.out.println(p.getAge());
    }

    public static void sideEffect(Person p1){
        p1.setAge(99);
    }
}
```

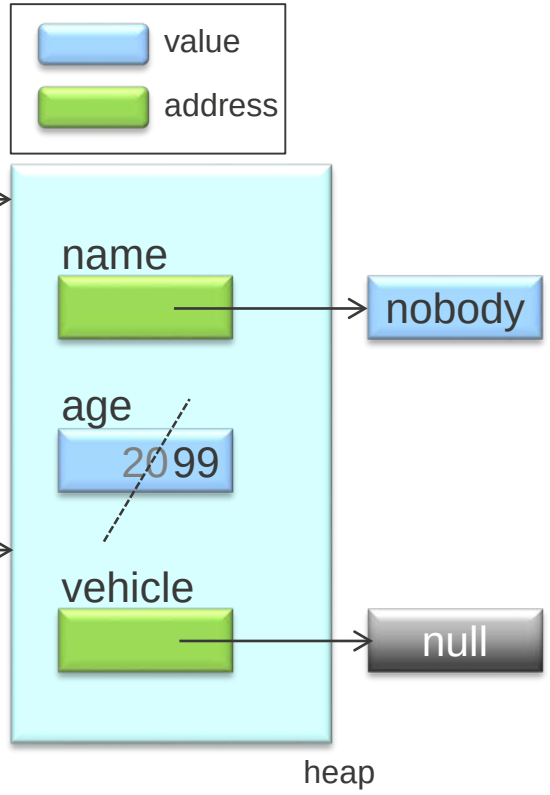


The reference variable p appears as an actual argument in 3 methods calls.
It's the second call (to the method sideEffect) that interests are here.
This is because rather than passively getting the value of an instance variable of a Person object (e.g. getAge) this method sets such a value (setAge(…)).

Object Reference Parameters – Side Effects

```
public class PersonDriver2{

    public static void main(String[] args){
        Person p = new Person();

        System.out.println(p.getAge());
        sideEffect(p);
        System.out.println(p.getAge());
    }

    public static void sideEffect(Person p1){
        p1.setAge(99);
    }
}
```

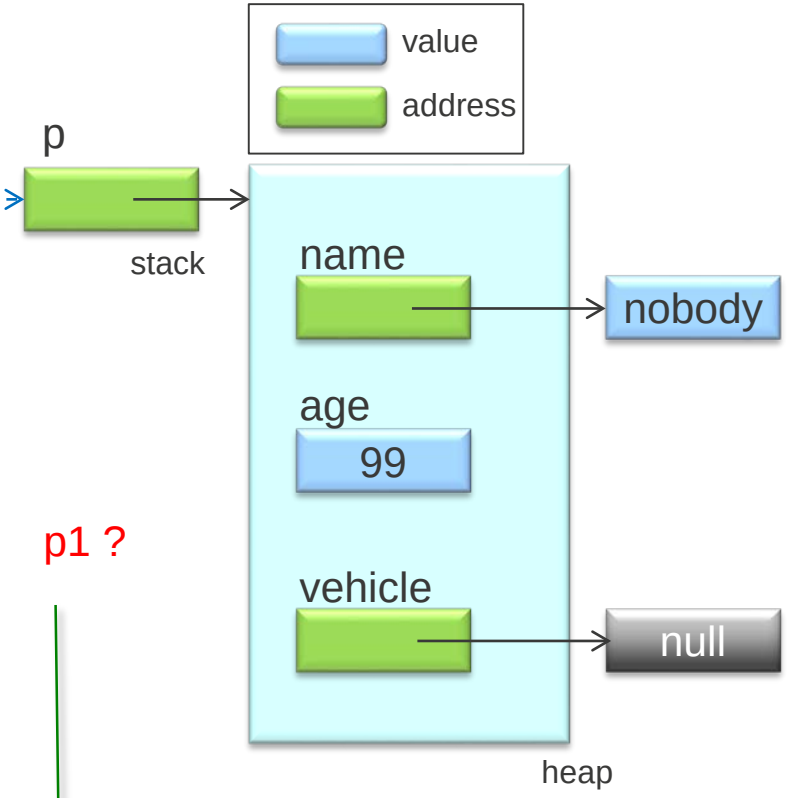
same as `p.setAge(99)`



Lets concentrate on the second call.
The usual parameter passing mechanism takes place i.e. a copy of the value of the actual parameter is used to initialise the corresponding formal parameter.
The difference here is that both parameters are reference variables and the value passed is an address. “p” and “p1” are now synonyms or aliases. They both references the same object in the heap i.e.: p1.setAge(99) is the same as p.setAge(99).

Object Reference Parameters – Side Effects

```
public class PersonDriver2{

    public static void main(String[] args){
        Person p = new Person();

        System.out.println(p.getAge());
        sideEffect(p);
        System.out.println(p.getAge() 99);
    }

    public static void sideEffect(Person p1){
        p1.setAge(99);
    }
}
```



p1 ?

getAge method call is replaced by its return value.

20
99

p1 is a formal parameter of the sideEffect method and therefore behaves like a local variable of this method. Therefore since sideEffect has finished executing p1 is erased from memory.

Object Reference Parameters – Side Effects

- When an Object Reference is passed as a parameter in a method call
  - Its value (which is *the address* of the object) is copied from actual to formal parameter
  - i.e. a copy of the object's *address* is passed
- Code that has an object's address can access the object's public methods
  - Including Mutator methods which set the value of instance variables
- Therefore code in a called method, that has been passed an object reference can call any public mutators of that object
  - i.e. the called method can effect the State (data) of an object from the calling method.
  - This is called a side effect of the called Method

Object Reference Parameters – Side Effects

- Are Side Effects bad?
  - When a called method can cause side effects in a calling method we say the methods are coupled
    - This means neither can be created or maintained in isolation which complicates such tasks

- By the way
  - If the value of the passed object reference is changed (i.e. it is set to point at another object from the same class) in the called method, then the local parameter is no longer an alias of an object in the calling method and any method coupling associated with it disappears (see next slide)

```
public class PersonDriver3 {

    public static void main(String[] args) {
        Person p = new Person();

        System.out.println(p.getAge());
        noSideEffect(p);
        System.out.println(p.getAge());
}

    public static void noSideEffect(Person p1) {
        Person p2 = new Person();

        p1 = p2;

        p1.setAge(99);
    }
}
```

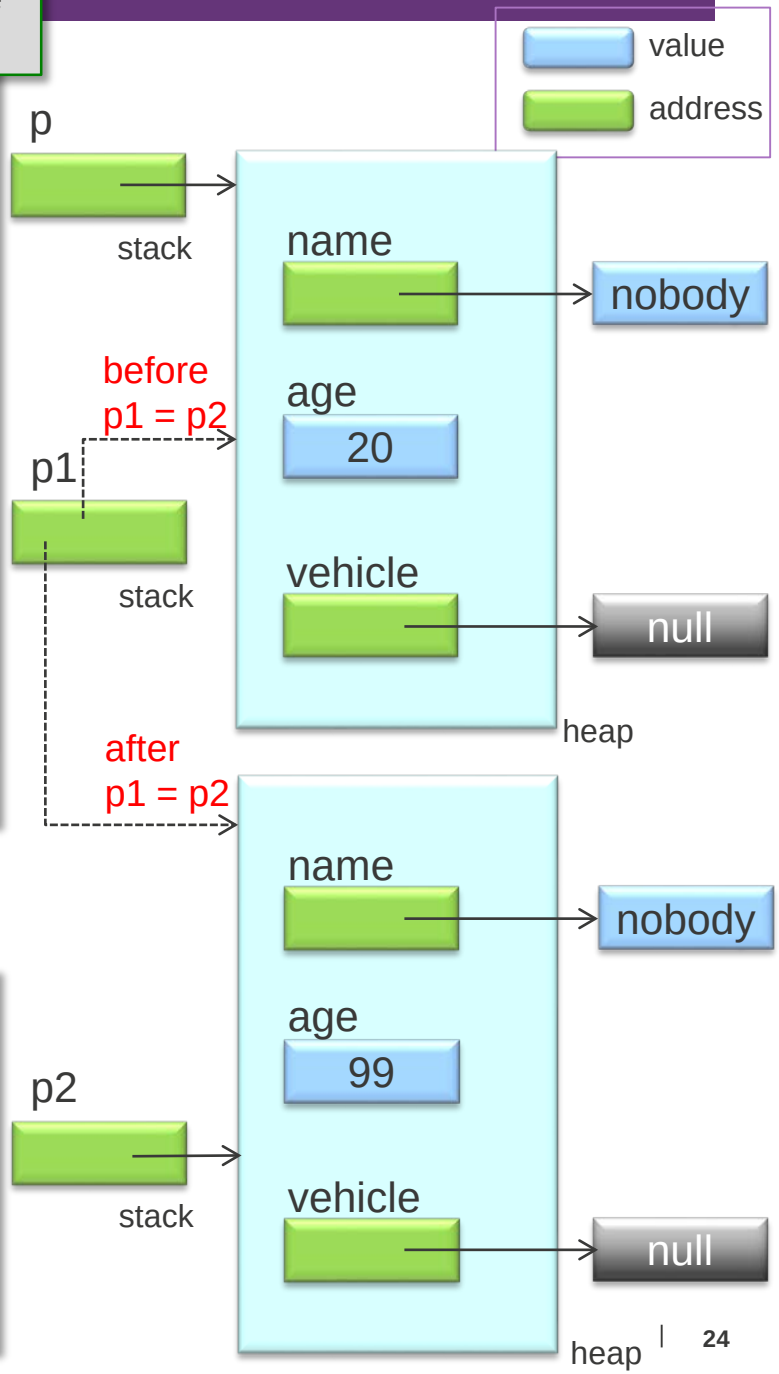
This Driver code outside the Person class

After this assignment p1 no longer references the same Person object as p. it now references the same Person object as p2.

p1.setAge(99) same as p2.setAge(99) NOT THE SAME AS p.setAge(99).

The reason why invoking methods on p1 (before p1 = p2) has an effect on p is because they both reference the same Person object in memory i.e. their value is that common Person objects' address.

During the execution of the noSideEffect method, we point p1 at another person object (p1 = p2). The connection between p1 and p is now broken and invoking methods on p1 now invokes these methods on another Person object in memory NOT the one pointed at by p i.e. It no longer has any effect on p.



Returning Object References

- A method can return an object reference
  - Obviously its specified return type (specified in its header) must be of the same reference type as the object reference returned
- The object reference must EITHER:
  - Reference (point at) an existing instantiated object of the type specified in the method's header OR
  - Be the special value `null` (which is interpreted as pointing at no objects)
    - Remember, variables of all reference types can be set to `null` but this doesn't mean they lose their type
- In the calling method
  - The returned object reference can have its class's methods invoked on it in the usual way

Part 2:
Introduction to Association Relationships

Class Relationships

- Objects of one class can be related to objects of another class
  - We say they are involved in an **association relationship** with each other
- The relationship could be momentary (e.g. only during a method call)
  - The Person objects given to the personsAge and sideEffect methods earlier today were not lasting
- or somewhat permanent (lasting beyond method call
  - Example on next slide – The car is related to the owner

Class Relationships - Association

- Every Person object can be associated to a Car object, through the 'vehicle' instance variable

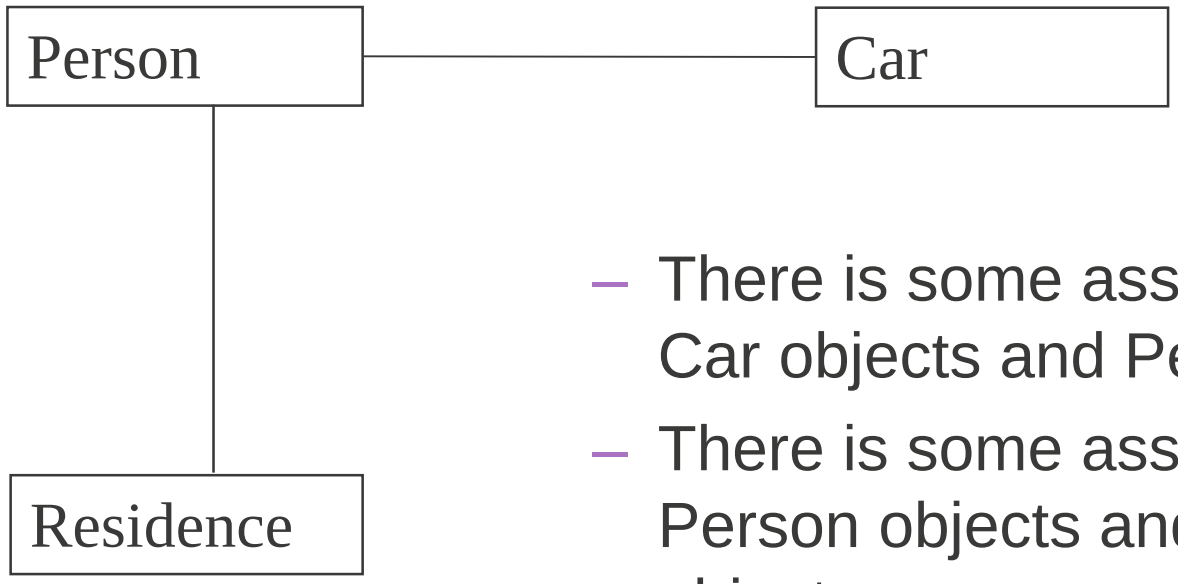
```
public class Person {
    private String name;
    private int age;
    private Car vehicle;
}
```

Class Relationship:
A Person is linked to Car

```
public class Car {
    private String type;
    private int buildYear;
    private Date regoDate;
}
```

Representing Associations

- A class diagram can show associations by connecting classes by labelled lines



  - There is some association between Car objects and Person objects
  - There is some association between Person objects and Residence objects

Links vs Associations

- association between two objects at run-time is known as a **link**
- An **association** is a specification of *the possibility* of two objects being linked
- A class is a specification of how the objects are designed to be
  - So all relationships specified in the class, can be manifested as links when an actual object is made
    - An association is an abstraction of all possible links.
  - All objects in a class are capable of the same *types* of links as each other, but they will likely each link to different actual objects

Uni-directional Association

- Arise where only one of the two participating classes ever sends messages (to the other)
  - Target – the class-type which receives the messages
  - Sender – the class-type which sends the messages
- Typically, the sender maintains references to objects of the target class type.
  - The Target does not know about the Sender object
  - The sender's reference to the target is established either:
    - In constructor of Sender class, or
    - By a mutator of Sender class, or
    - By being passed as a parameter to a method of the Sender which will call the target during the method

Uni-directional Association

- Person “knows” of the Car, but Car doesn’t “know” the Person

```
public class Person {
    private String name;
    private int age;
    private Car vehicle;
}
```

```
public class Car {
    private String type;
    private int buildYear;
    private Date regoDate;
}
```

Establishing Target by constructor

```
public class Person {
    private String name;
    private int age;
    private Car vehicle;

    public Person(String myName, int myAge, Car myVehicle)
    {
        name = myName;
        age = myAge;
        vehicle = myVehicle;
    }
}
```

Class Relationship:
A Person is linked to a Car

Note that the parameter is of type Car – it will be a reference to a Car object, or null

Establishes a somewhat permanent link from the Person, to the Car.

- After the constructor has run, the Person class can send messages to the Car at any time by using the 'vehicle' reference variable
  - Provided the myVehicle parameter was not null

Establishing/Changing Target by a mutator

```
public class Person {
    private String name;
    private int age;
    private Car vehicle;

    public void setVehicle(Car myNewVehicle)
    {
        vehicle = myNewVehicle;
    }
}
```

Note that the parameter is of type Car – it will be a reference to a Car object, or null

Previous linked Car will no longer be "known" by this Person object

Establishes a somewhat permanent link from the Person, to the Car.

- The setVehicle method changes which Vehicle that the Person has a reference to
  - Since it can be called again in the future, we say the link is semi-permanent – it is permanent unless the method is called again

Establishing/Changing Link

```
Car toyota;
Car holden;
Person linda;
...

linda = new Person("Linda",19, toyota);

...

linda.setVehicle(holden);

...

linda.setVehicle(null);
```

Various object references are declared in the driver class. The cars should be initialised (not shown here)

Constructing a Person, and providing the reference to a Car for the Person to link to

Changing the Car that the Person will now link to

Linda may sell her car, so she no longer links to any car

Momentary Associations

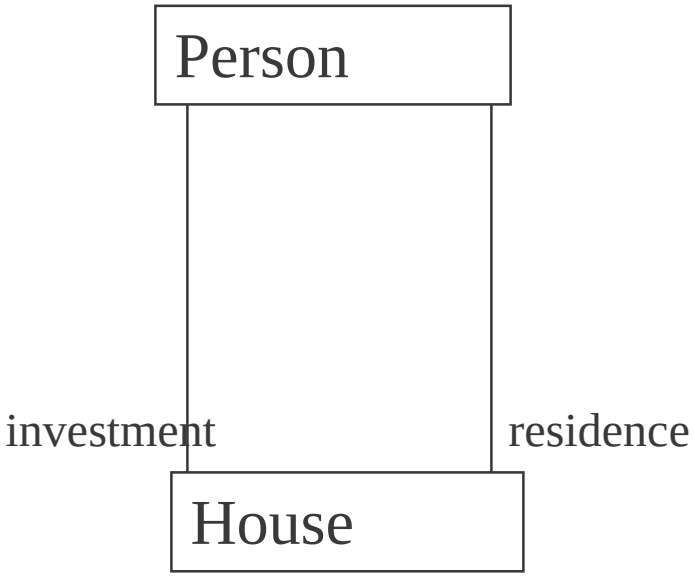
- Sometimes a method requires an object during its processing, but is not part of the class in which the method is written
- We can provide the object as a parameter, and use it, but we won't "remember" it beyond conclusion of the method

```
class CarMechanicShop
{
    public void serviceCar(Car someCar)
    {
        someCar.putPetrolIn(50);
        someCar.removeAllOil();
        someCar.putOilInMotor(10);
        someCar.putCoolantIn(25);
    }
}
```

  - We are able to use the object's public methods
  - We don't store the parameter anywhere else during the method

Multiple Associations

- There may be more than one association between two classes of the same types.
  - For a given ‘Person’, there may be different objects involved for each of the associations…



Multiple Associations – Example

```
public class Person {
    private House myInvestment; // my Investment property
    private House myResidence;  // where I live
    ...
}
```

- The Person class refers to several House objects
  - So, a Person object can send messages to each of these separate House objects independently

```
public class House {
   private Person myOwner;
   private Person myTenant;
   private String address;
   ...
}
```

  - And a given House refers to several Person objects in this example

Bi-directional Associations

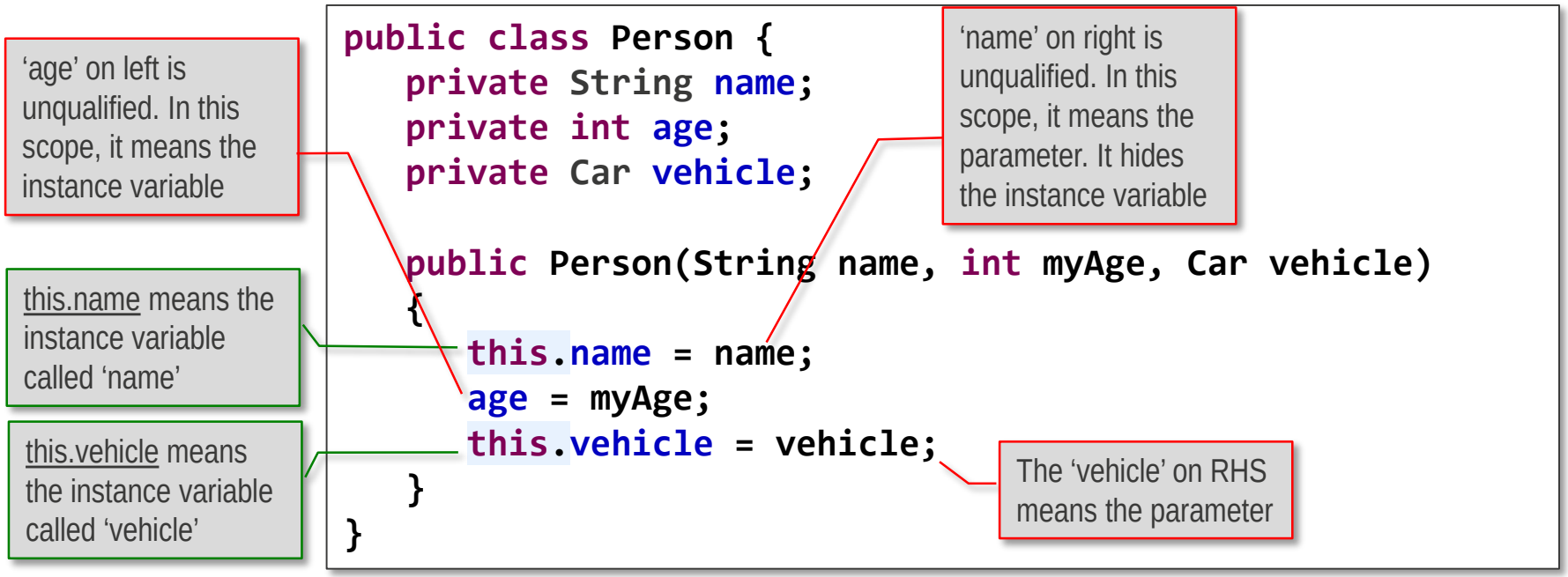
- Both objects know of each other, so either may send a message to the other one at any time
  - One will be the Target, whilst the other is being Sender
  - Target/Sender role can switch without needing to re-establish a relationship
- Establishing these links is more complicated:
  - Both objects need to exist, but one must be instantiated before the other
  - Order of creation depends on other links required
  - One class's set method will call the other class's set method.
  - May require passing the 'this' reference (on next slide) as a parameter to the other object's mutator method!!!!

The special reference: this

- The Java keyword ‘this’ is an alias for whatever object is executing the statement in which that use of the reference has been coded
  - It is only able to be used inside instance methods, not in static methods
- It Is largely redundant since it is assumed
  - Any unqualified instance variable name in a class’s method code is assumed to be preceded by “**`this`**`.`”
    - i.e. the instance variable of the object this method was invoked on
  - Any unqualified method call in a class’s method code is assumed to be preceded by “**`this`**`.`”
    - i.e. the called method is to be invoked on the object the calling method was invoked on

The special reference: this

- It is commonly used in constructors and mutators so that parameters can use the same name as an instance variable
  - Such parameters hide or shadow the instance variables due to rules about scope



Establishing a bi-directional link

```
public class Person {
    private String name;
    private int age;
    private Car vehicle;

    public void setVehicle(
      Car myNewVehicle)
    {
       vehicle = myNewVehicle;
       vehicle.setMyOwner(this);
    }
}
```

Establishes link to Vehicle, from Person

```
public class Car {
    private String type;
    private int buildYear;
    private Date regoDate;
    private Person myOwner;

    public Car(String type, int year)
    {
       this.type = type;
       this.buildYear = year;
       this.myOwner = null;
    }

    public void setMyOwner(Person who)
    {
       myOwner = who;
    }
}
```

Establishes link in other direction (to the Person)

In driver, we say:

linda.setVehicle(holden)

Time out question

- Assume we have several Person objects
  - Linda, Michael, Toby
- After the following sequence of instructions, what object will the holden Vehicle's myOwner variable refer to?

```
holden.setMyOwner(linda);
holden.setMyOwner(michael);
holden.setMyOwner(toby);
```

- After the same sequence of instructions, what would linda's vehicle variable be?

Danger when coding bi-directional links

- Trying to maintain a bi-directional link is tricky
  - If we change one side of the association, we should change the other
- Avoid using them unless necessary
- If you do need a bi-directional link, enforce a policy of “checking” the other side
  - Establish one direction’s link (e.g. Person to Car)
  - Invoke a method on the other side to set the back-link
    - In this method, perform a check that the forward link is established by calling an accessor on the parameter (see next slide)
    - If the back link is currently established to another Person, disestablish it, and then record back-link to new Person

Safer establishment of bi-directional link

```
public class Person {
   private String name;
   private int age;
   private Car vehicle;

   public void setVehicle(
     Car myNewVehicle)
   {

      vehicle = myNewVehicle;
      if (vehicle != null)
         vehicle.setMyOwner(this);
   }

   public Car getVehicle()
   {
      return vehicle;
   }
}
```

```
public class Car {
   private String type;
   private int buildYear;
   private Date regoDate;
   private Person myOwner;

   ...

   public void setMyOwner(Person who)
   {
      if ((who == null)  ||
          (who.getVehicle() == this) )
      {
         if ((myOwner != null)
          && (myOwner != who) )
            myOwner.setVehicle(null);

         myOwner = who;
      }
   }
}
```

Only call setMyOwner for non-null parameter

Checks if the parameter Person object's vehicle is referring to this very object – same address of object

Check for an existing (old) owner

Check there actually is a change of owner being proposed

Establish new link

Disestablish old link that old owner has to us

Part 3:
Composition and Privacy Leaks

Composition

- Sometimes objects are made up of parts which themselves are objects
  - Example: A chair is made up of the cushioned seat, the legs, the back-rest
  - Example: A flower is made up of petals and stem
- We say that the Chair and the Flower each are a **composite** of various other things
  - We cannot conceive of a flower without petals, or a chair without a seat and legs.
  - Composition specifies a very tight binding relationship to the parts
- Composite objects in Java require special treatment of the parts

Composite – Example

```
public class Car {
   private String type;
   private int buildYear;
   private Date regoDate;
}
```

reference
primitive
reference

A Car has a Date when it was first registered (its registration date)


```
public class Date {
   private int day;
   private int month;
   private int year;
}
```

primitive
primitive
primitive

- The ‘Date’ object used by the Car class is an attribute, or integral part of the Car
  - The Car object was *not* a ‘part’ of the Date, however

Date Class

- Assume a simple Date class coded as follows:

```
public class Date {                    // No validation checks coded
   private int day;
   private int month;
   private int year;

   public Date(int day, int month, int year){
      this.day = day;
      this.month = month;
      this.year = year;
   }
   public void setDay(int day) { this.day = day; }
   public int getDay() { return day; }
   public void setMonth(int month) { this.month = month; }
   public int getMonth() { return month; }
   public void setYear(int year) { this.year = year; }
   public int getYear() { return year; }
}
```

Composition – Protection of parts

- An object which is a composite of other objects must take extra precautions to protect the composite data items (its parts)
  - They must be private
  - They should not have mutators
  - There should not be any way to circumvent the privateness of the items
- Usually, a composite's parts are created by the constructor of the composite object
  - You do not pass them in as parameters, otherwise they would not be private to the object

Example – Establishment of Composite

```
public Car(String type, int yearMade,
  int regDay, int regMonth, int regYear)
{
    this.type = type;
    this.buildYear = yearMade;
    myOwner = null;
    regoDate = new Date(regDay,regMonth,regYear);
}
```

- This version of the Car constructor accepts parameters which are used to construct a Date object that is an intrinsic 'part' of the Car
  - There will be no mutator for the regoDate attribute, because it is a fixed fact about the object.
  - Can we have an accessor for the regoDate attribute?

Accessors for composite parts?

- You may be attempted to write the following as an accessor for the regoDate attribute:

```
public Date getRegoDate()
{
    return regoDate;
}
```

- This is considered bad, because it exposes the private attribute of the Car to now be publicly known by whichever object invoked getRegoDate()
  - The attribute may as well have been public, but this would not be following the principles of encapsulation and information hiding

Privacy Leaks

- A **privacy leak** arises whenever some data which is private to an object due to being an integral ‘part’ of the object, is exposed to other classes by the address of the data being returned to or provided by the external class
  - The reference returned on the previous slide is exposing the address of the Date object
- It is a problem, because the external object can now invoke public methods on that ‘part’ object, and make changes which the ‘composite’ will not know are being made
  - Possibly violating the integrity of the composite

Privacy leak illustrated

```
public class MainDriver {
    public static void main(String[] args)
    {
        Car ford;
        ford = new Car("Falcon",2005,1,1,2006);
        ford.printRegistration();

        Date someDate = ford.getRegoDate();

        someDate.setMonth(5);

        ford.printRegistration();
    }
}
```

Access via an accessor of private vehicle data of somebody

Direct access of private vehicle data of somebody

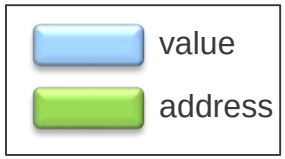
We expect this statement should print the same output as the earlier statement, because we have not changed the ford object's data by calling any mutator of the ford object

```
public class Car {
    private Date regoDate;
    ...
    public Date getRegoDate() { return regoDate; }
    public void printRegistration() {
        System.out.println(regoDate);
    }
}
```

- What is output by this driver?

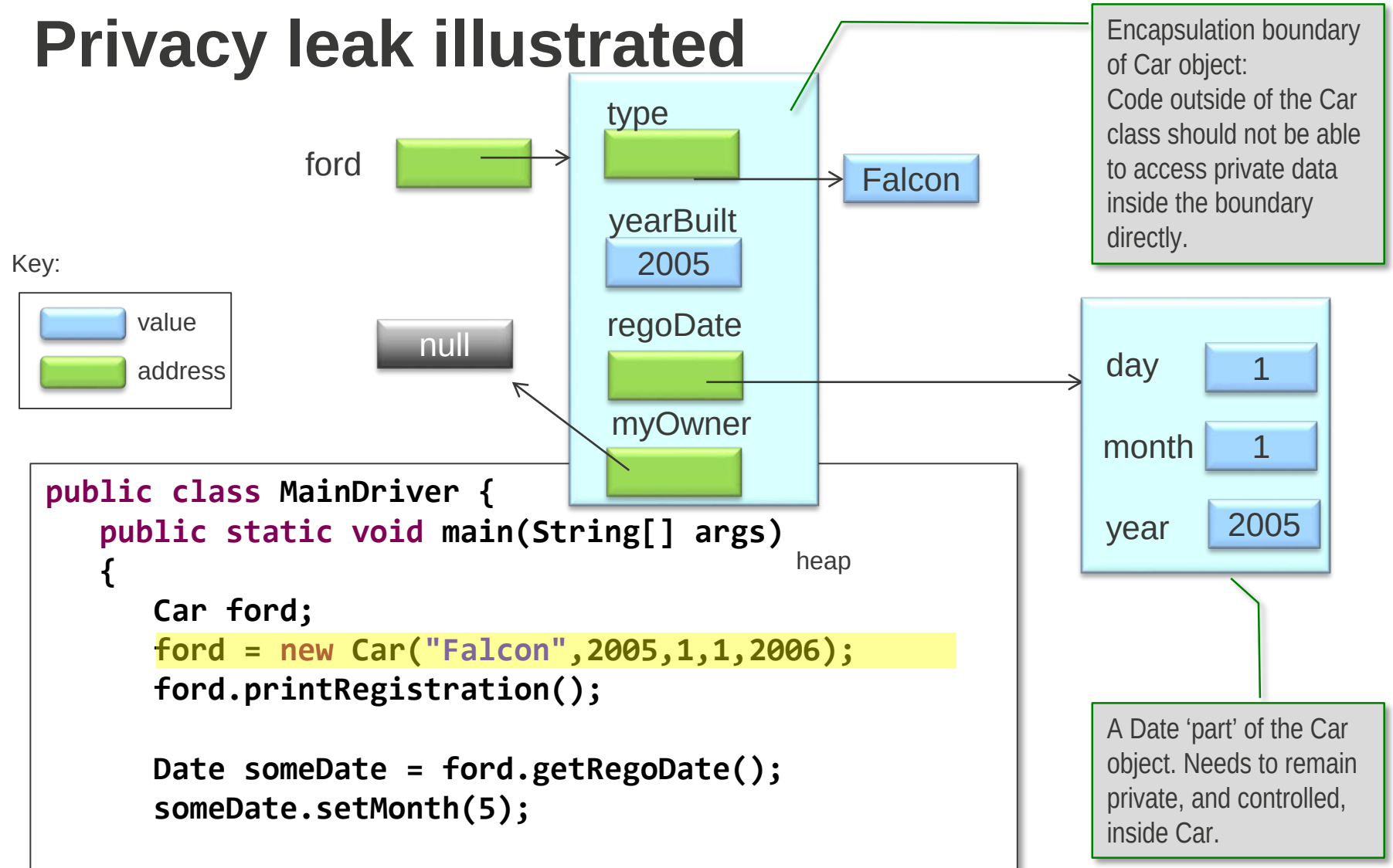
Privacy leak illustrated

Key:



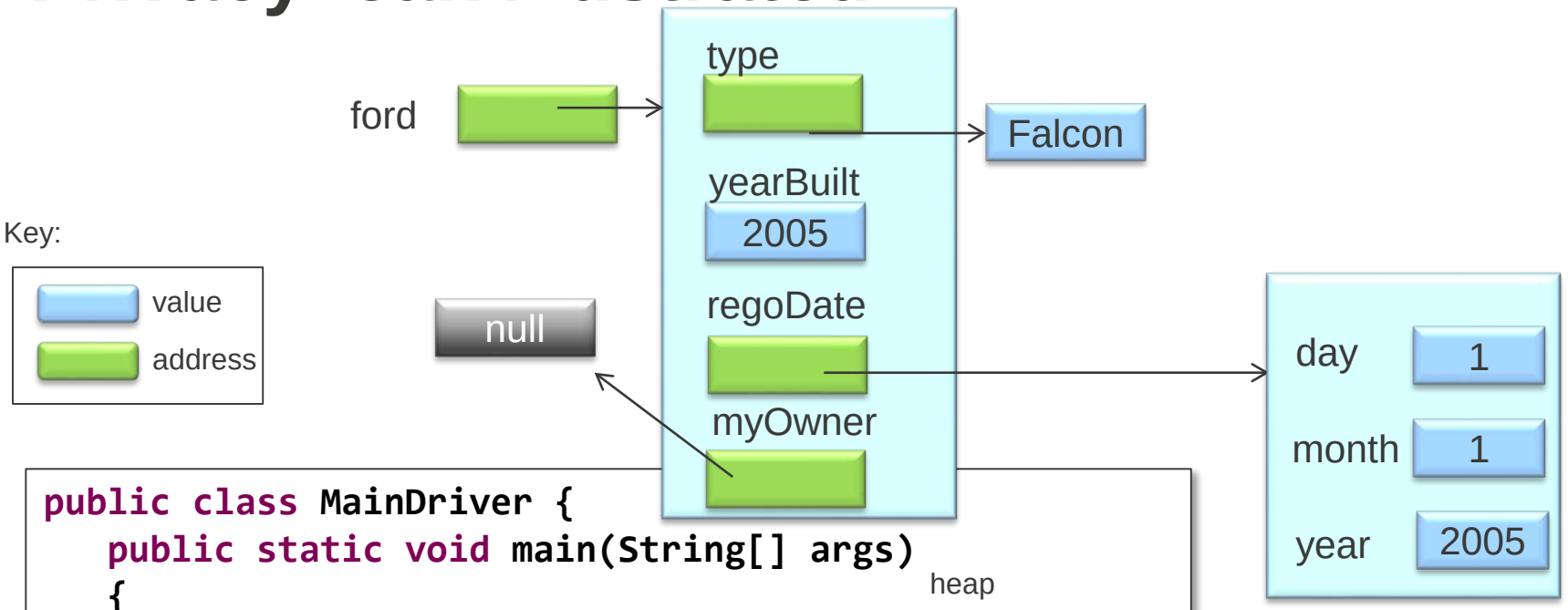

```
public class MainDriver {
    public static void main(String[] args)
    {
        Car ford;
        ford = new Car("Falcon",2005,1,1,2006);
        ford.printRegistration();

        Date someDate = ford.getRegoDate();
        someDate.setMonth(5);

        ford.printRegistration();
    }
}
```

Privacy leak illustrated




```
public class MainDriver {
    public static void main(String[] args)
    {
        Car ford;
        ford = new Car("Falcon",2005,1,1,2006);
        ford.printRegistration();

        Date someDate = ford.getRegoDate();
        someDate.setMonth(5);

        ford.printRegistration();
    }
}
```

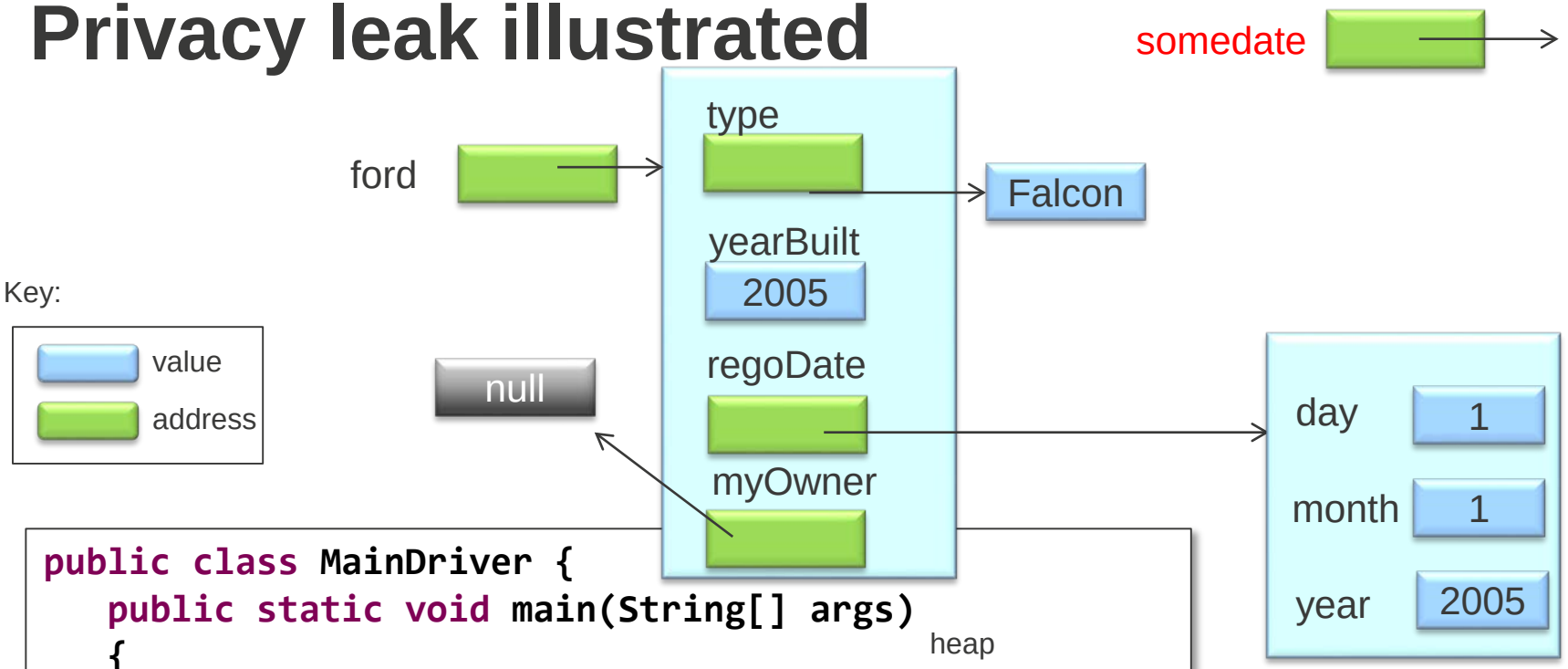
Privacy leak illustrated




```
public class MainDriver {
    public static void main(String[] args)
    {
        Car ford;
        ford = new Car("Falcon",2005,1,1,2006);
        ford.printRegistration();

        Date someDate = ford.getRegoDate();
        someDate.setMonth(5);

        ford.printRegistration();
    }
}
```

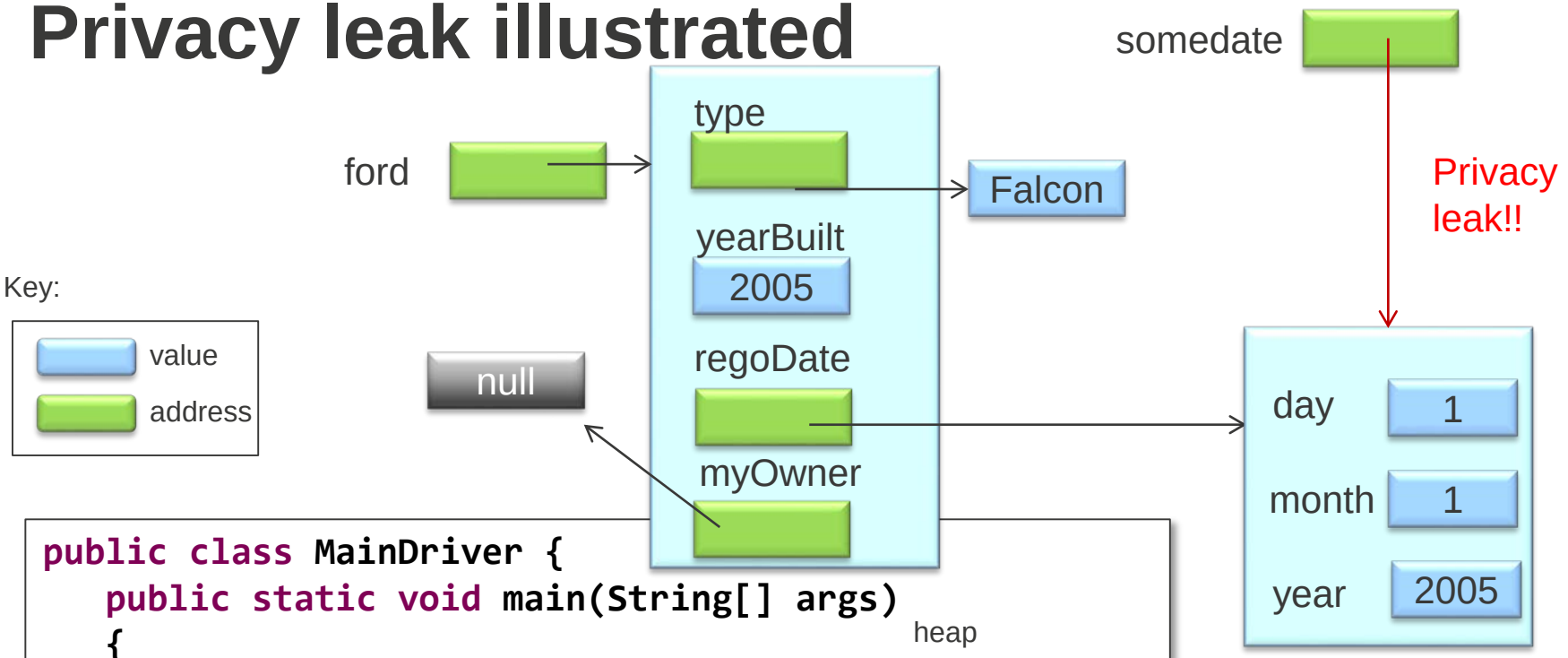
1/1/2005

Privacy leak illustrated




```
public class MainDriver {
    public static void main(String[] args)
    {
        Car ford;
        ford = new Car("Falcon",2005,1,1,2006);
        ford.printRegistration();

        Date someDate = ford.getRegoDate();
        someDate.setMonth(5);

        ford.printRegistration();
    }
}
```

Privacy leak illustrated

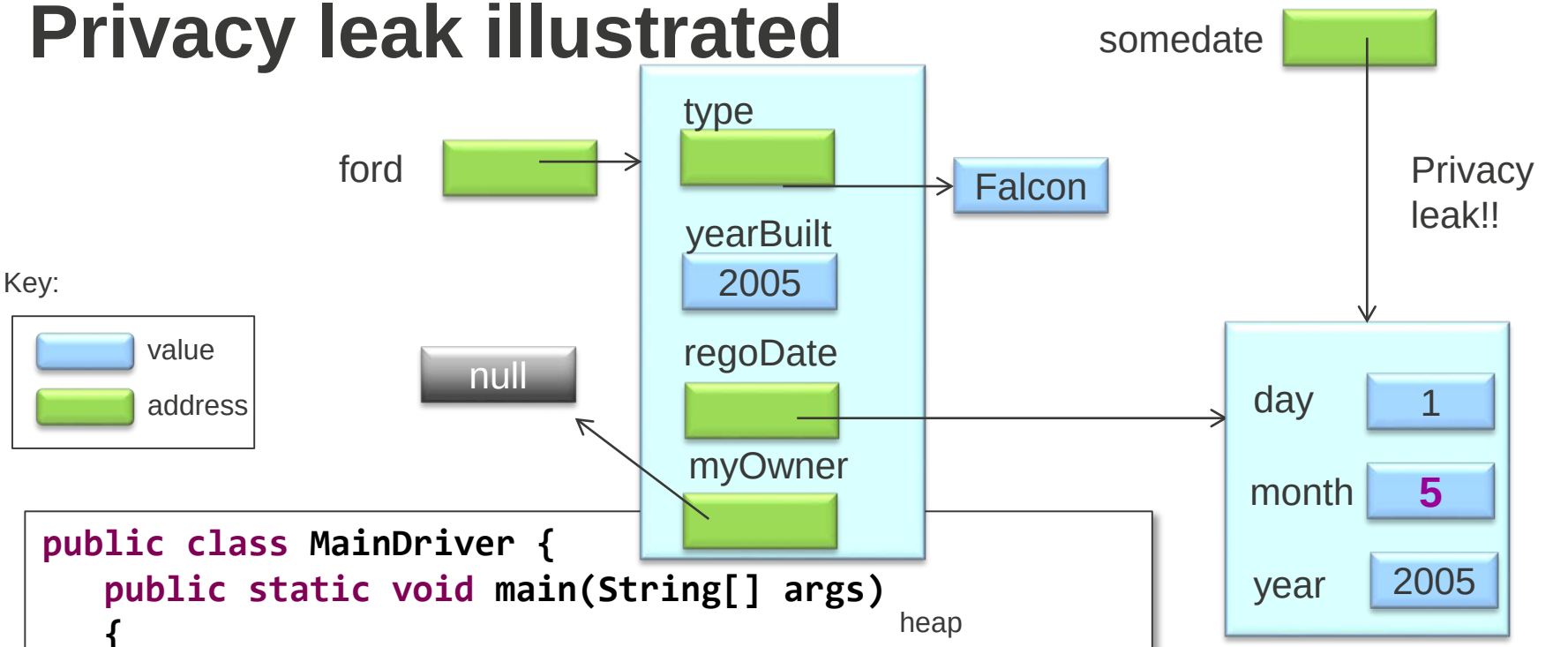



```
public class MainDriver {
   public static void main(String[] args)
   {
      Car ford;
      ford = new Car("Falcon",2005,1,1,2006);
      ford.printRegistration();

      Date someDate = ford.getRegoDate();
      someDate.setMonth(5);

      ford.pri
   }
}
```

```
public class Car {
  public Date getRegoDate() { return regoDate; }
}
```

Privacy Leak out. BAD!!!
Here an address of an instance variable of reference type is being passed out of the class allowing code outside the class to have access to that private variable's state.

Privacy leak illustrated

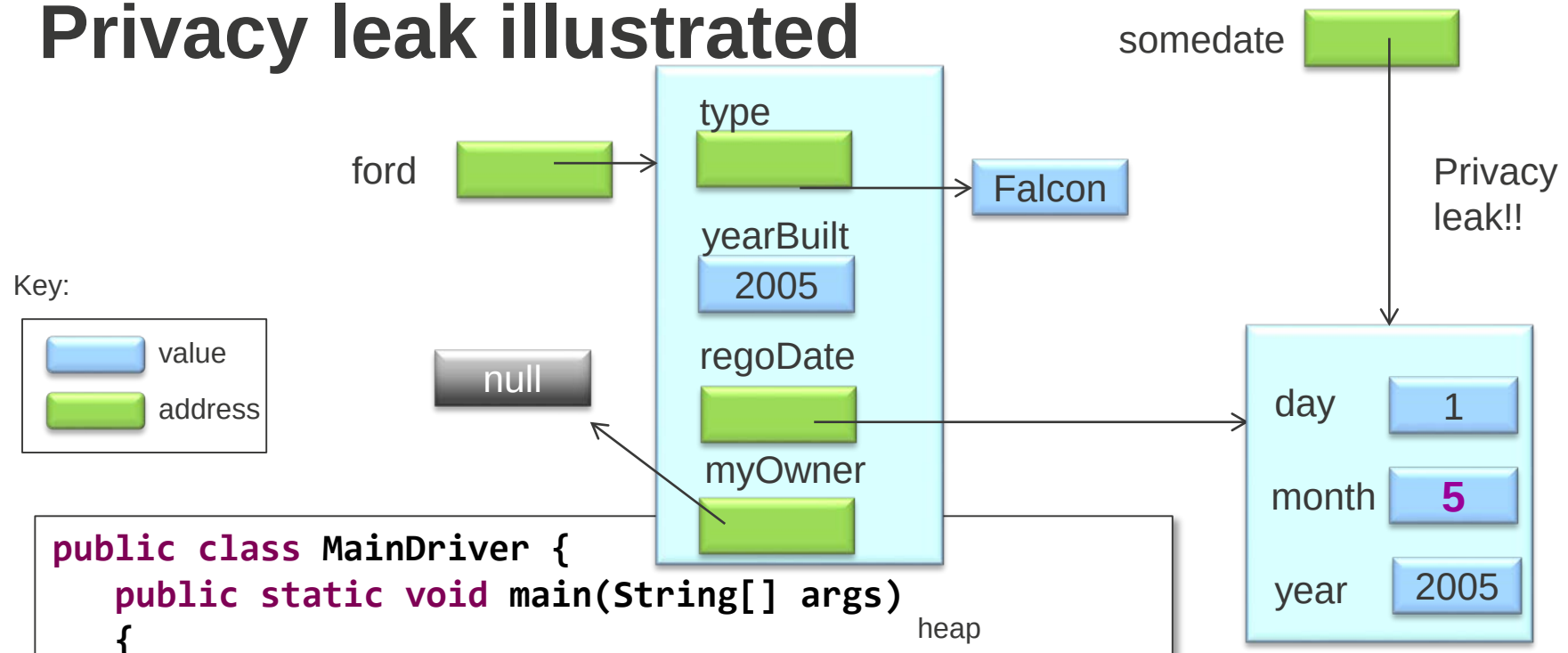



```
public class MainDriver {
    public static void main(String[] args)
    {
        Car ford;
        ford = new Car("Falcon",2005,1,1,2006);
        ford.printRegistration();

        Date someDate = ford.getRegoDate();
        someDate.setMonth(5);

        ford.p
    }
}
```

Now the leaked address (someDate which is an alias of regoDate) is used to set the value of one of regoDate's instance variables directly (i.e. bypassing ford).

```
public class Date {
    public void setMonth(int month) { this.month = month; }
}
```

Privacy leak illustrated




```
public class MainDriver {
    public static void main(String[] args)
    {
        Car ford;
        ford = new Car("Falcon",2005,1,1,2006);
        ford.printRegistration();

        Date someDate = ford.getRegoDate();
        someDate.setMonth(5);

        ford.printRegistration();
    }
}
```

1/1/2005

Here a "legal" access via ford to an instance variable of regoDate confirms the leak is present.

1/5/2005

What Just Happened?

- regoDate
  - Is a (private) instance variable of the Car object referenced by the reference variable “ford”
  - It should therefore only be manipulated through the Car class’s methods (otherwise the Car may get “confused”)
- Unfortunately its accessor (`getRegoDate()`) surrenders the address of regoDate to code outside the Car class
- This separate code can now bypass the Car class’s accessors and mutators and access the vehicle’s “private” instance variables through the Date class’s accessors and mutators

What Just Happened?

- Such Privacy Leaks are only a problem for instance variables that are of reference type
  - The accessors and mutators of instance variables of primitive type pass out and in, respectively, copies of primitive type values, not addresses
  - Having a copy of a value does not permit access to the original value (they are at two different memory locations)
- So what should happen instead?
  - Write accessors and mutators of reference variables to mimic the way instance variables of primitive type are accessed and mutated

What Should Happen?

- Accessors
  - Should not pass out the address of an instance variable of reference type
  - They should make a copy of the instance variable object and pass the copy's address out
    - There is then no connection between the 2 objects
- Mutators
  - Should not receive as a parameter (should not pass in) the address of an object and blindly assign this address to an instance variable of reference type
  - They should make a copy of the passed object and assign the copy's address to the instance variable of reference type
    - There is then no connection between the 2 objects

A Better Accessor – returning a copy

```
public class Car {
    private Date regoDate;
    ...
    public Date getRegoDate() {
        Date copyDate;
        copyDate = new Date(regoDate);
        return copyDate;
    }
}
```

Construct a new object (in separate memory location), using the regoDate as the basis to copy from

Return the copy.
No privacy leak now.

```
public class Date {
    private int day;
    private int month;
    private int year;
    ...
    public Date(Date other){
      if (other != null) {
       this.day   = other.day;
       this.month = other.month;
       this.year  = other.year;
      } else { day = month = year = 0; }
    }
}
```

This is called a Copy Constructor. It receives an object of its own type as its sole parameter, and copies the value of each field to the new object

Summary: Association versus Composition

- Composition is where objects form part of a composite
  - The part uniquely belongs to the composite and cannot be shared or become part of some other composite during the part's lifetime
- Association is simply that one object is remembering a reference to another object but this second object is not 'part' of the first one
  - The second object could be referred to by many other objects simultaneously
  - Example: one 'teacher' object referred to by many different student objects
- Privacy leaks are a concern only for reference variables that are composition parts