CSE201: Monsoon 2020
Advanced Programming

Lecture 02: Classes and Objects

Raghava Mutharaju (Section-B)

Vivek Kumar (Section-A)

Computer Science and Engineering

IIIT Delhi

raghava.mutharaju@iiitd.ac.in

Last Lecture

- Introduction to OOP
  - What, Why, and Advantages of OOP
  - Encapsulation
  - Procedural programming v/s OO programming

Today’s Lecture

- Identifying classes and objects
- Working with objects

Ideas to Program

Analysis (common sense)

Design (object oriented)

Implementation (actual programming)

Testing

- Analysis
  - ***What to do and not how to do it***
  - Decide corner cases and exact functionalities
- Design
  - Define classes, their attributes and methods, objects, and class relationships
- Implementation
  - Novice programmers often think that writing code is the heart of software development, but actually it should be the least creative step
- Testing
  - A program should be free of errors

Analysis: Identifying Classes & Responsibilities

- Identifying classes
  - Good first step: look for **nouns** in use cases. Then...
    - **Actors**- objects that perform tasks
    - **Events** - store information about events
- Identifying responsibilities
  - Good first step: look for **verbs**, **actions** in use cases
    - These actions may directly describe responsibilities, or
    - may depend on other responsibilities

Analysis: Identifying Classes

- A partial requirement document

> **The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.**

Of course, not all nouns will correspond to a class or object in the final solution

Analysis: Guidelines for Discovering Classes

- Limit responsibility of each analysis class
  - Clear purpose for existence
  - Avoid giving too many responsibilities to one class
- Use clear and consistent names
  - Class names should be nouns
  - Not finding good name implies class is too fuzzy
- Keep analysis classes simple
  - In first step don’t worry about class relationships

Exercise: Logging into Email Account

- A partial requirement document

For accessing an online email account, the customer will first click the login button on the home page of the email account. This will display the login page of email account. Once the customer gets directed to the login page, he will enter his user id and password, and then click the OK button. The email account will first validate the customer credentials and then grant access to his email account.

Exercise: Logging into Email Account

- Step -1: Identifying classes (nouns) and objects

For accessing an online email account, the customer will first click the login button on the home page of the email account. This will display the login page of email account. Once the customer gets directed to the login page, he will enter his user id and password, and then click the OK button. The email account will first validate the customer credentials and then grant access to his email account.

Exercise: Logging into Email Account

- Step -2: Identifying methods (verbs)

For accessing an online email account, the customer will first click the login button on the home page of the email account. This will display the login page of email account. Once the customer gets directed to the login page, he will enter his user id and password, and then click the OK button. The email account will first validate the customer credentials and then grant access to his email account.

Design: Classes and Objects

- Recall, class represents a group of objects with similar behaviors
  - Instantiate as many objects as you like!
- If a class becomes too complex, decompose into multiple smaller classes
- Assign responsibilities to each class
  - Every activity in a program represents methods in a class
  - In early stages, begin with primary responsibilities and evolve the design

Design: Interaction Between Objects

- Sequence diagrams
  - Interaction diagrams that details how operations are carried out in a program
  - **Messages**: Interaction between two objects is performed as a message sent from one object to another
    - Help tracing object methods and interactions
  - UML is significantly improved version of sequence diagram
    - *We will cover this in depth in later lectures*

Exercise: Logging into Email Account

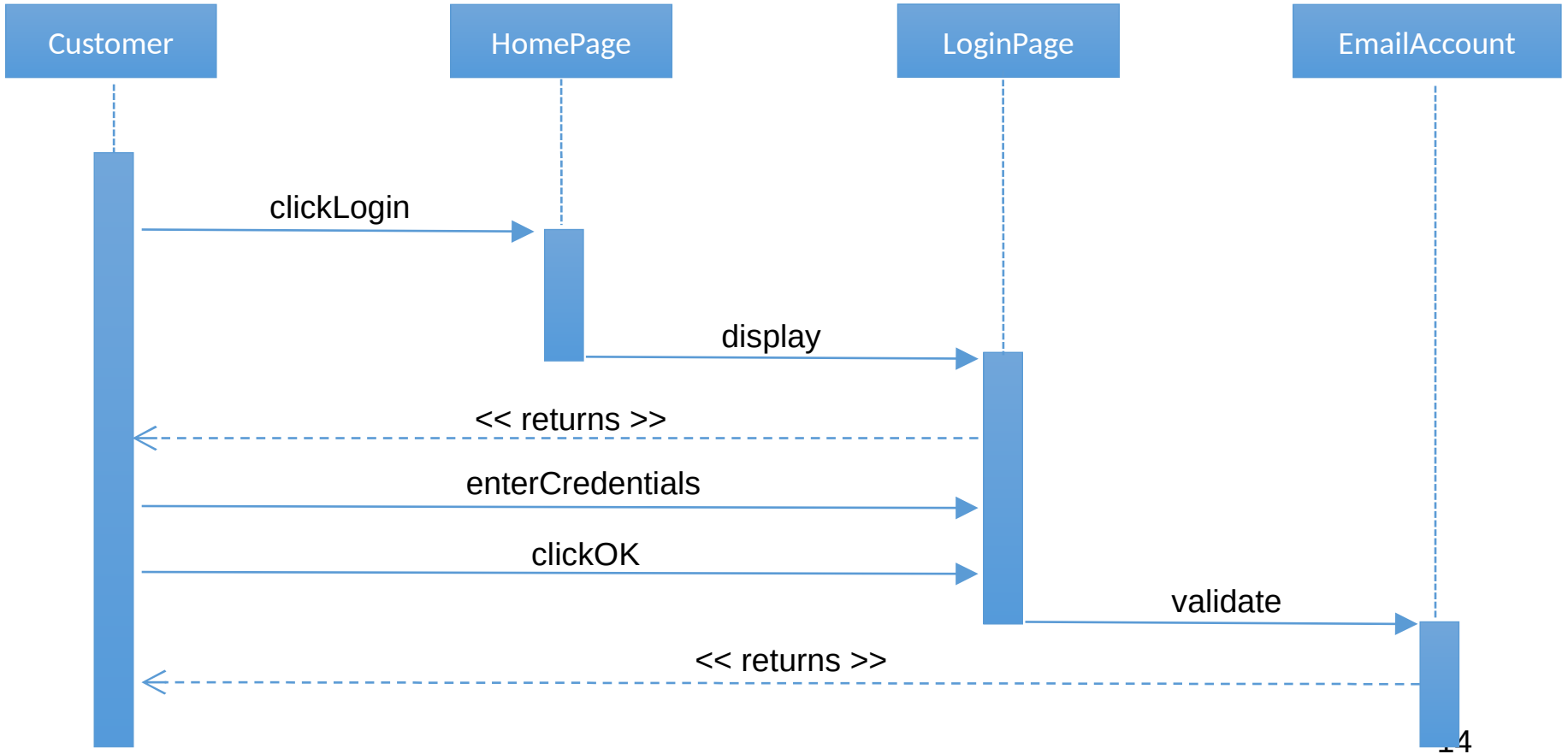
- Step-3: Draw the sequence diagram

For accessing an online email account, the customer will first click the login button on the home page of the email account. This will display the login page of email account. Once the customer gets directed to the login page, he will enter his user id and password, and then click OK button. The email account will first validate the customer credentials and then grant access to his email account.

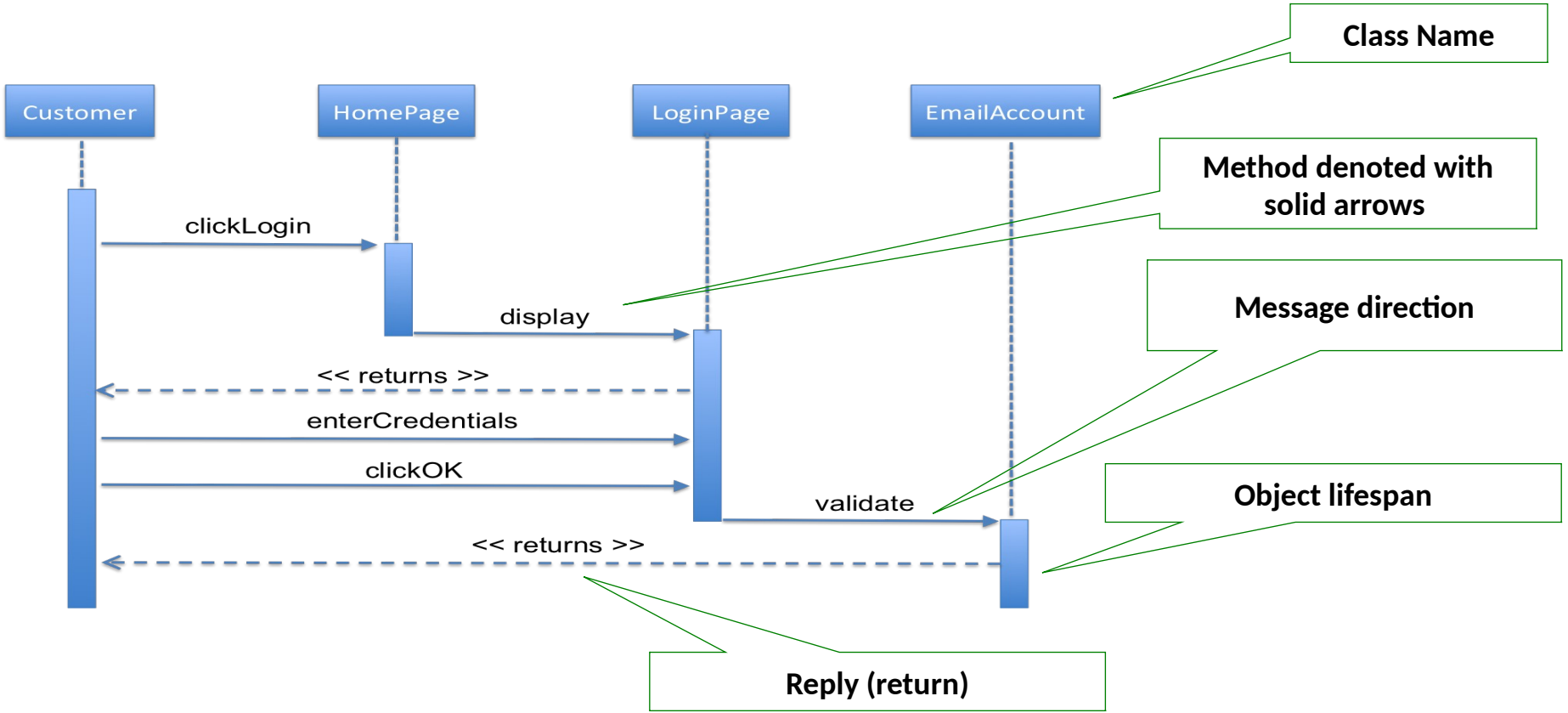
Classes
Customer
HomePage
LoginPage
EmailAccount

Methods
clickLogin
display
enterCredentials
clickOK
validate

Sequence Diagram



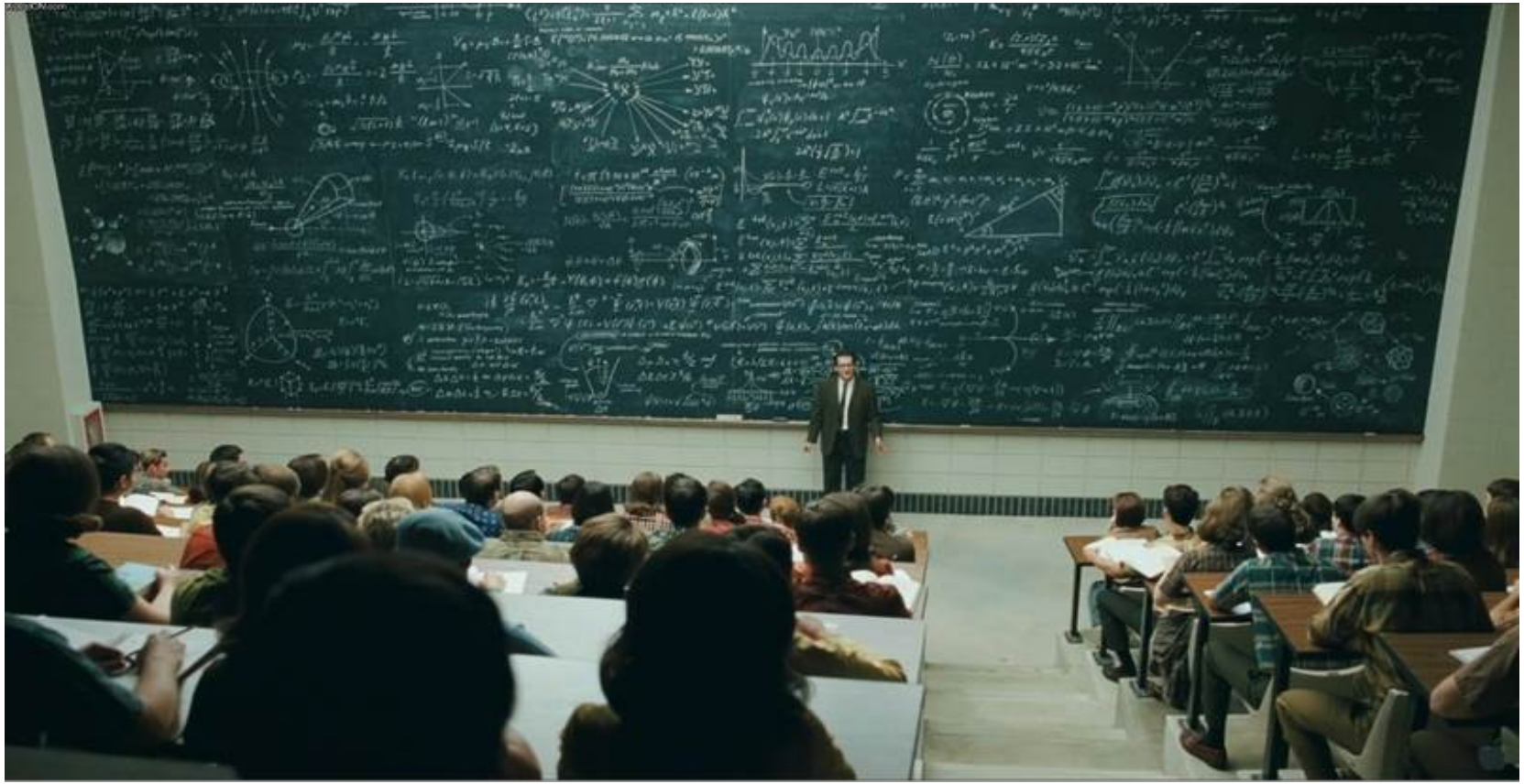
Sequence Diagram



Cohesion Between Methods

- Methods of an object should be in harmony. If a method seems out of place, then your object might be better off by giving that responsibility to somewhere else
- E.g., for *LoginPage class*, *enterCredentials()*, *clickOK()* are in harmony but not if we make *validate()* as method of *LoginPage*

Identify Classes Below

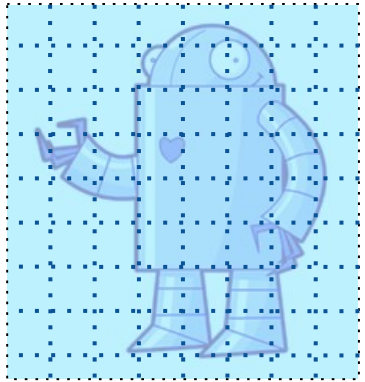


Let’s change gears…

How to Work with Objects ?

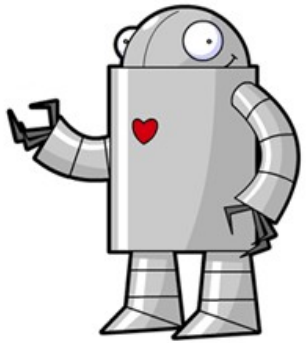
Review: Instantiation

- **Instantiation** means building an object from its class “blueprint”
- Ex: `new Robot();` creates an instance of Robot
- This calls the `Robot` class’s **constructor**: a special kind of method



The `Robot` class

`new Robot();`



instance

Review: Constructors

- A **constructor** is a method that is called to create a new object
- Let's define one for the `Dog` class
- All `Dog`s know how to bark, eat, and wag their tails

```
public class Dog {

  public Dog() {
    // this is the constructor!
  }

  public void bark(int numTimes) {
      // code for barking goes here
  }

  public void eat() {
      // code for eating goes here
  }

  public void wagTail() {
      // code for wagging tail goes here
  }
}
```

Review: Constructors

- Constructors do not specify a return type
- Name of constructor must exactly match name of class
- Now we can instantiate a Dog in some method:

```
new Dog();
```

```
public class Dog {

  public Dog() {
    // this is the constructor!
  }

  public void bark(int numTimes) {
      // code for barking goes here
  }

  public void eat() {
      // code for eating goes here
  }

  public void wagTail() {
      // code for wagging tail goes here
  }
}
```

Review: Constructors

Question:

- Find the order of execution for following statement

Dog djangho = new Dog(“Djangho”);

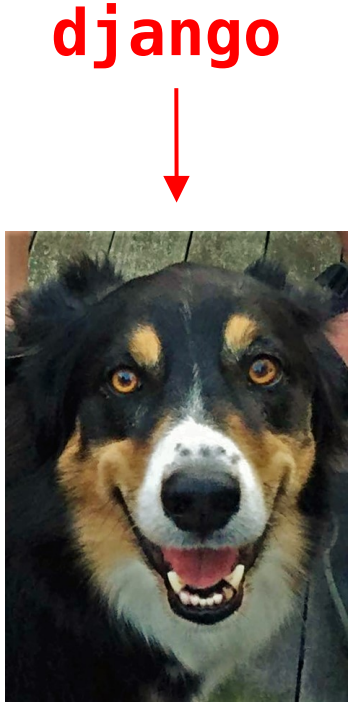
```
public class Dog {
  private String name;
  private int breed_id;
  private int rego_id;
  private static int rego_counter;

  { // initialization block
     rego_id = ++rego_counter;  //
line-z
  }
  public Dog(int _breed) {
    this.breed_id = _breed;     //
line-y
  }
  public Dog(String _name) {
    this(20);
    this.name = _name;          //
line-x
```

Variable Declaration & Assignment

```
Dog django = new
      Dog();
```

```
<type> <name> =
     <value>;
```

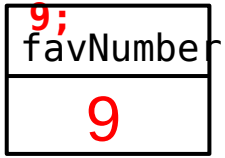
- The “=” operator **assigns** the instance of `Dog` that we created to the variable `django`. We say “`django` gets a new `Dog`”
- Note that we can reassign as many times as we like



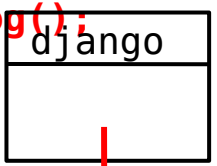
Variables Store Information: Values vs. References

- A variable stores information as either:
  - a *value of a primitive (*aka *base) type* (like `int` or `float`)
  - or a *reference to an instance* (like an instance of `Dog`) of an arbitrary type stored elsewhere in memory – we symbolize a reference with an arrow

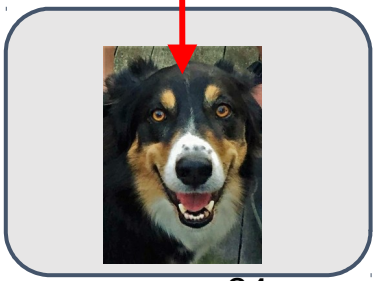
```
int favoriteNumber = 9;
```




```
Dog django = new Dog();
```



- Think of the variable like a box; storing a value or reference is like putting something into the box
- Primitives have a predictable memory size, while arbitrary objects vary in size, hence Java simplifies its memory management by having a fixed size reference to an instance elsewhere in memory
  - "one level of indirectness"



(somewhere else in memory)

Example: Instantiation

```
public class PetShop {

   /*constructor of trivial PetShop! */
   public PetShop() {
       this.testDjango();
   }

   public void testDjango() {
       Dog django = new Dog();
       django.bark(5);
       django.eat();
       django.wagTail();

   }
       …
}
```

This doesn't seems to be the job of PetShop owner! Maybe DogGroomer should be hired..

- Let's call the `testDjango()` method within the constructor of the `PetShop` class
- Whenever someone instantiates a `PetShop`, it in turn calls `testDjango()`, which in turn instantiates a `Dog`
- Then it tells the `Dog` to bark, eat, and wag its tail

Objects as Parameters (1/2)

- Methods can take in objects as parameters
- The `DogGroomer` class has a method `groom`
- `groom` method needs to know which `Dog` to groom

```
public class DogGroomer {

  public DogGroomer() {
      // this is the constructor!
  }
            type          name
  public void groom(Dog shaggyDog)
  {
      // code that grooms shaggyDog
  }
}
```

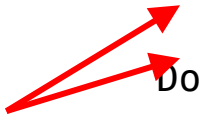
Objects as Parameters (2/2)

- How to call the `groom` method?
- Do this in the `PetShop` helper method `testGroomer()`
- `PetShop`'s call to `testGroomer()` instantiates a `Dog` and a `DogGroomer`, then tells the `DogGroomer` to groom the `Dog`

```
public class PetShop {

  public PetShop() {
      this.testGroomer();
  }

  public void testGroomer() {
      Dog django = new Dog();
      DogGroomer groomer = new
  DogGroomer();
      groomer.groom(django);
  }

}
```

What is Memory?

- Memory (system memory, not disk or other peripheral devices) is the hardware in which computers store information, both temporary and permanent
- Think of memory as a list of slots; each slot holds information (e.g., a local int variable, or a reference to an instance of a class)
- Here, two references are stored in memory: one to a Dog instance, and one to a DogGroomer instance



```
//Elsewhere in the program
Petshop petSmart = new Petshop();

public class PetShop {

  public PetShop() {
      this.testGroomer();
  }

  public void testGroomer() {
      Dog django = new Dog();
      DogGroomer groomer = new
DogGroomer();
      groomer.groom(django);
  }

}
```

Objects as Parameters: Under the Hood (1/6)

```
public class PetShop {

   public PetShop() {
        this.testGroomer();
   }

   public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
   }
```

```
public class DogGroomer {

   public DogGroomer() {
        // this is the constructor!
   }

   public void groom(Dog shaggyDog) {
        // code that grooms shaggyDog goes here!
   }
}
```

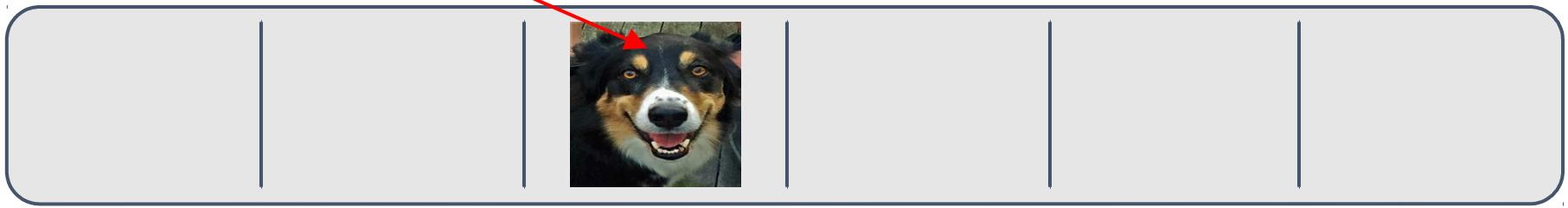
Somewhere in memory...

Objects as Parameters: Under the Hood (2/6)

```
public class PetShop {

   public PetShop() {
        this.testGroomer();
   }

   public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
   }
```

```
public class DogGroomer {

   public DogGroomer() {
        // this is the constructor!
   }

   public void groom(Dog shaggyDog) {
        // code that grooms shaggyDog goes here!
   }
}
```

Somewhere in memory...



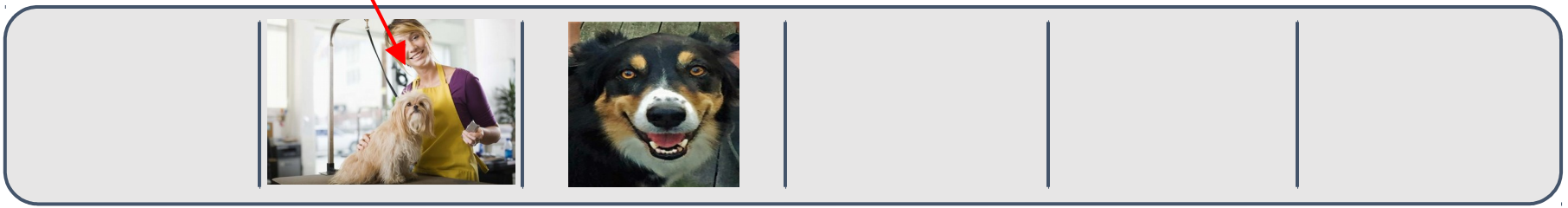
When we instantiate a `Dog`, he's stored somewhere in memory. Our `PetShop` will use the name `django` to refer to this particular `Dog`, at this particular location in memory.

Objects as Parameters: Under the Hood (3/6)

```
public class PetShop {

   public PetShop() {
        this.testGroomer();
   }

   public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
   }
```

```
public class DogGroomer {

   public DogGroomer() {
        // this is the constructor!
   }

   public void groom(Dog shaggyDog) {
        // code that grooms shaggyDog goes here!
   }
}
```

Somewhere in memory...



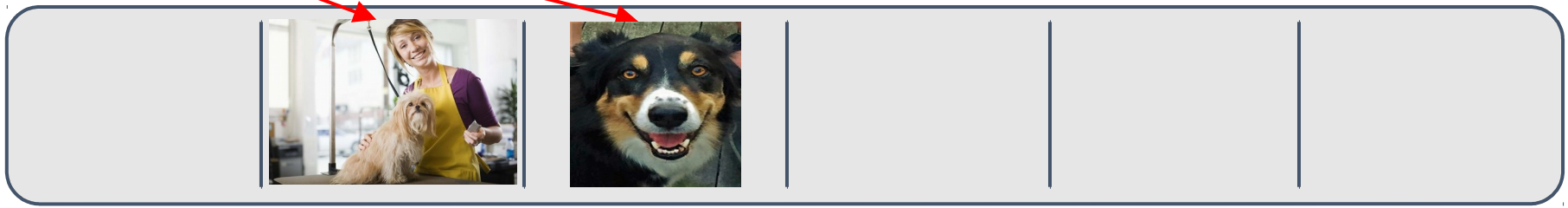
The same goes for the `DogGroomer`—we store a particular `DogGroomer` somewhere in memory. Our `PetShop` knows this `DogGroomer` by the name `groomer`.

Objects as Parameters: Under the Hood (4/6)

```
public class PetShop {

   public PetShop() {
        this.testGroomer();
   }

   public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
   }
```

```
public class DogGroomer {

   public DogGroomer() {
        // this is the constructor!
   }

   public void groom(Dog shaggyDog) {
        // code that grooms shaggyDog goes here!
   }
}
```

Somewhere in memory...



We call the `groom` method on our `DogGroomer`, `groomer`. We need to tell her which `Dog` to groom (since the `groom` method takes in a parameter of type `Dog`). We tell her to groom `django`.

Objects as Parameters: Under the Hood (5/6)

```
public class PetShop {

   public PetShop() {
        this.testGroomer();
   }

   public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
   }
```

```
public class DogGroomer {

   public DogGroomer() {
        // this is the constructor!
   }

   public void groom(Dog shaggyDog) {
        // code that grooms shaggyDog goes here!
   }
}
```

Somewhere in memory...

When we pass in `django` as an argument to the `groom` method, we're telling the `groom` method about him. When `groom` executes, it sees that it has been passed that particular `Dog`.

Objects as Parameters: Under the Hood (6/6)

```
public class PetShop {

   public PetShop() {
        this.testGroomer();
   }

   public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
   }
```

```
public class DogGroomer {

   public DogGroomer() {
        // this is the constructor!
   }

   public void groom(Dog shaggyDog) {
        // code that grooms shaggyDog goes here!
   }
}
```

Somewhere in memory...

The `groom` method doesn't really care which `Dog` it's told to groom—no matter what another object's name for the `Dog` is, `groom` is going to know it by the name `shaggyDog`.

Variable Reassignment (1/2)

- After giving a variable an initial value, we can **reassign** it (make it refer to a different object)
- What if we wanted our `DogGroomer` to `groom` two different `Dog`s when the `PetShop` opened?
- Could re-use the variable `django` to first point to one `Dog`, then another!

```
public class PetShop {

  /* This is the constructor! */
  public PetShop() {
      this.testGroomer();
  }

  public void testGroomer() {
      Dog django = new Dog();
      DogGroomer groomer =
             new DogGroomer();
      groomer.groom(django);
  }
}
```

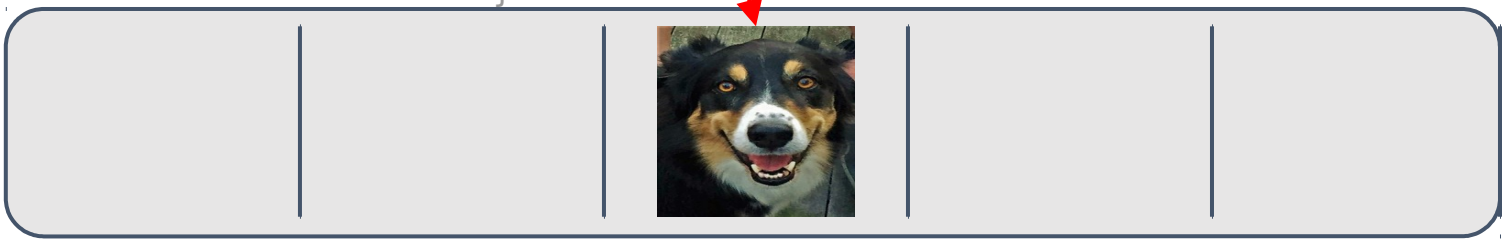
Variable Reassignment (2/2)

- First, instantiate another `Dog`, and reassign variable `django` to point to it
- Now `django` no longer refers to the first `Dog` instance we created, which has already been groomed
- We then tell `groomer` to `groom` the newer `Dog`

```
public class PetShop {

/* This is the constructor! */
public PetShop() {
    this.testGroomer();
}

public void testGroomer() {
    Dog django = new Dog();
    DogGroomer groomer =
            new DogGroomer();
    groomer.groom(django);
  django = new Dog();// reassign django
  groomer.groom(django);
}
```

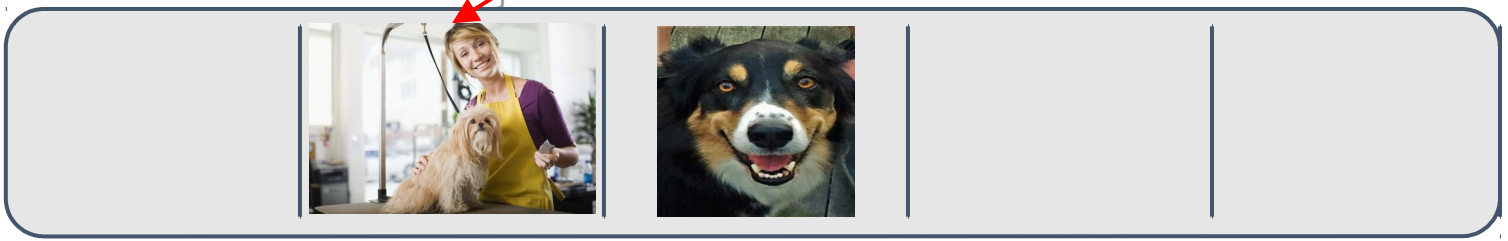
Variable Reassignment: Under the Hood (1/5)

```
public class PetShop {

   /* This is the constructor! */
   public PetShop() {
        this.testGroomer();
   }

   public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
        django = new Dog();
        groomer.groom(django);
   }
```



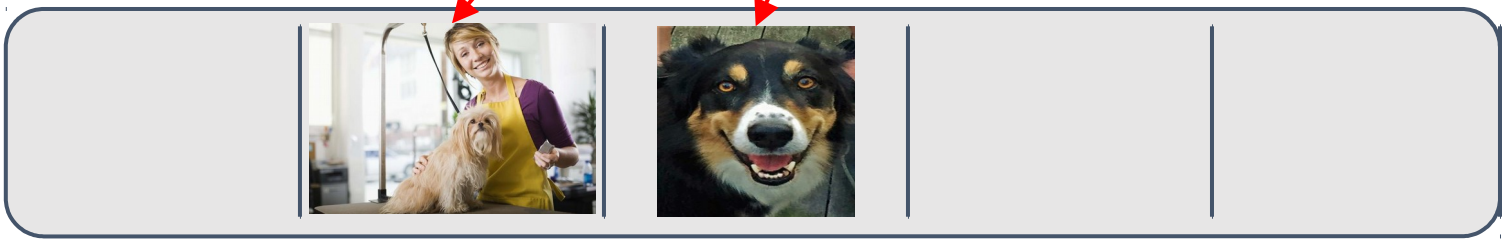
Variable Reassignment: Under the Hood (2/5)

```
public class PetShop {

   /* This is the constructor! */
   public PetShop() {
        this.testGroomer();
   }

   public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
        django = new Dog();
        groomer.groom(django);
   }
```



Variable Reassignment: Under the Hood (3/5)

```
public class PetShop {

   /* This is the constructor! */
   public PetShop() {
        this.testGroomer();
   }

   public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
        django = new Dog();
        groomer.groom(django);
   }
```



Variable Reassignment: Under the Hood (4/5)

```
public class PetShop {

   /* This is the constructor! */
   public PetShop() {
        this.testGroomer();
   }

   public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
        django = new Dog();
        groomer.groom(django);
   }
```

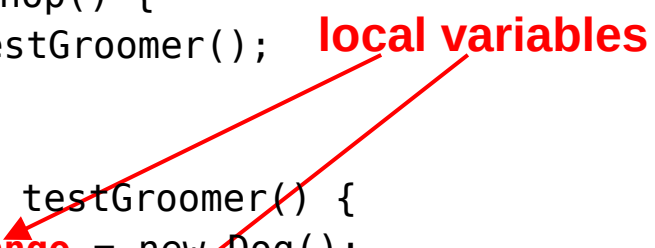
Variable Reassignment: Under the Hood (5/5)

```
public class PetShop {

   /* This is the constructor! */
   public PetShop() {
        this.testGroomer();
   }

   public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
        django = new Dog(); //old ref garbage collected
        groomer.groom(django);
   }
```

Local Variables (1/2)

- All variables we've seen so far have been **local variables**: variables declared *within a method*
- Problem: the ***scope*** of a local variable (where it is known and can be accessed) is limited to its own method—it cannot be accessed from anywhere else
  - the same is true of method parameters

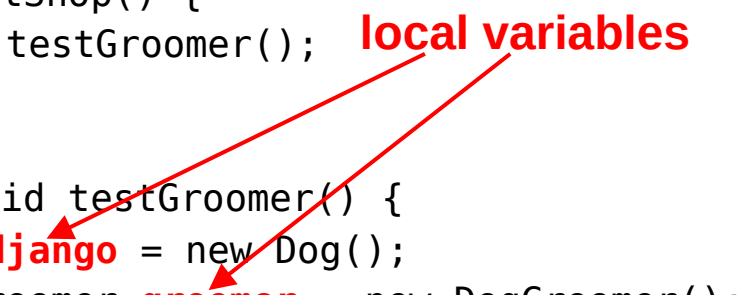
```
public class PetShop {

    /* This is the constructor! */
    public PetShop() {
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
        django = new Dog();
        groomer.groom(django);
    }

}
```



Local Variables (2/2)

- We created `groomer` and `django` in our `PetShop`'s helper method, but as far as the rest of the class is concerned, they don't exist
- What happens to django after the method is executed?
  - "Garbage Collection"

```
public class PetShop {

    /* This is the constructor! */
    public PetShop() {
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
        django = new Dog();
        groomer.groom(django);
    }

}
```



Accessing Local Variables

- If you try to access a local variable outside of it's method, you'll receive a "cannot find symbol" compilation error.

In Terminal:

```
Petshop.java:13: error: cannot find
symbol
  django.playCatch();
     ^
  symbol: variable django
  location: class PetShop
```

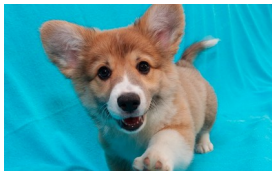
```
public class PetShop {

   private DogGroomer _groomer;

   /* This is the constructor! */
   public PetShop() {
        _groomer = new DogGroomer();
        Dog django = new Dog();
   }


    public void exerciseDjango() {
      django.playCatch();.
   }

}
```

Instance Variables for the Rescue

- Local variables aren't always what we want. We'd like every `PetShop` to come with a `DogGroomer` who exists for as long as the `PetShop` exists
- That way, as long as the `PetShop` is in business, we'll have our `DogGroomer` on hand
- We can accomplish this by storing the `DogGroomer` in an **instance variable**

What's an Instance Variable?

- An **instance variable** models a property that all instances of a class have
  - its *value* can differ from instance to instance (e.g, the dog's breed, name, color, …)
- Instance variables are declared within a class, not within a single method, and are accessible from anywhere within the class – its scope is the entire class
- Instance variables and local variables are identical in terms of what they can store—either can store a base type (like an `int`) or a reference to an object (instance of some other class)



Instance Variables

- We've modified `PetShop` example to make our `DogGroomer` an **instance variable**
- Split up declaration and assignment of instance variable:
  - **declare** instance variable
  - **initialize** the instance variable by **assigning** a value to it in the constructor
  - purpose of constructor is to initialize all instance variables so the instance has a valid initial "state" at its "birth"
  - state is the set of all values for all properties—local variables don't hold properties - they are "temporaries"

```
public class PetShop {

  private DogGroomer _groomer;

  /* This is the constructor! */
  public PetShop() {
      _groomer = new DogGroomer();
      this.testGroomer();
  }

  public void testGroomer() {
    Dog django = new Dog();//local
  var
      _groomer.groom(django);
  }

}
```

declaration

assignment

Always Remember to Initialize!

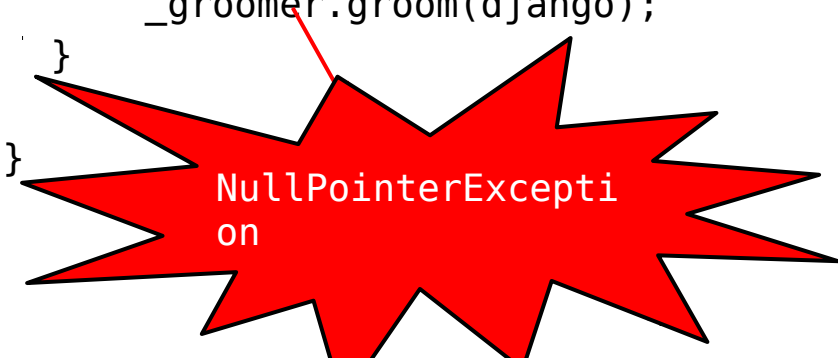
- What if you declare an instance variable, but forget to initialize it?
- The instance variable will assume a “default value”
  - if it’s an `int`, it will be 0
  - if it’s an object, it will be `null`—a special value that means your variable is not referencing any instance at the moment

```
public class PetShop {

  private DogGroomer _groomer;

  /* This is the constructor! */
  public PetShop() {
      //oops!
      this.testGroomer();
  }

  public void testGroomer() {
    Dog django = new Dog();//local
  var
      _groomer.groom(django);
  }

}
```

NullPointerExceptions

- If a variable's value is null and you try to give it a command, you'll be rewarded with a *runtime error*—you can't call a method on "nothing"!
- This particular error yields a `NullPointerException`
- When you run into one of these (we promise, you will)—edit your program to make sure you have explicitly initialized all variables

```
public class PetShop {

  private DogGroomer _groomer;

  public PetShop() {
      //oops!
      this.testGroomer();
  }

  public void testGroomer() {
    Dog django = new Dog();//local
  var
      _groomer.groom(django);
  }
}
```



Next Lecture

- Class relationships