CSE201: Advanced Programming

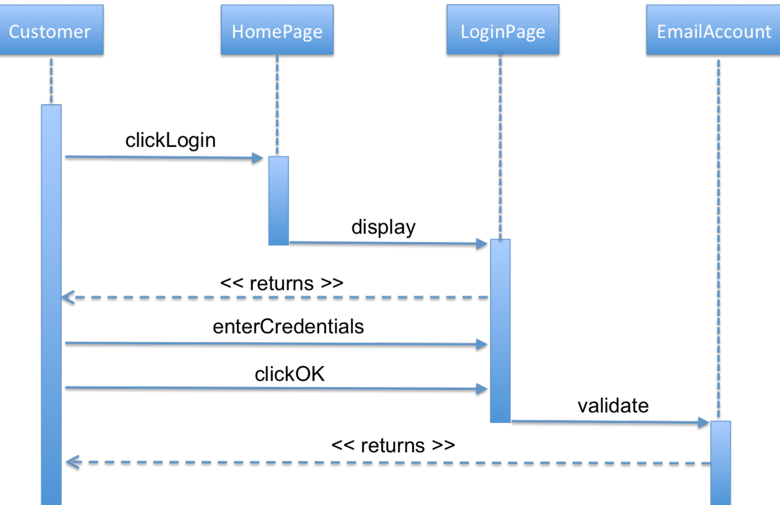
Lecture 03: Class Relationships

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

- Program development
- Identifying classes and objects
- Sequence diagrams
- Working with objects
  - Objects as parameters
  - Variable reassignment
  - Instance variables

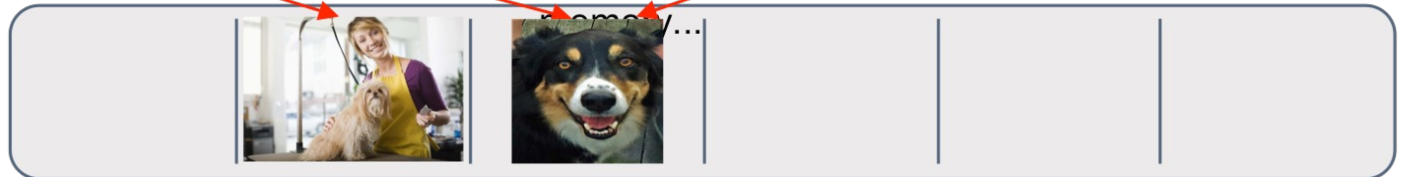
For accessing an online email account, the customer will first click the login button on the home page of the email account. This will display the login page of email account. Once the customer gets directed to the login page, he will enter his user id and password, and then click OK button. The email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




```
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



```
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);
  }
```

```
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

This Lecture

- Class relationships

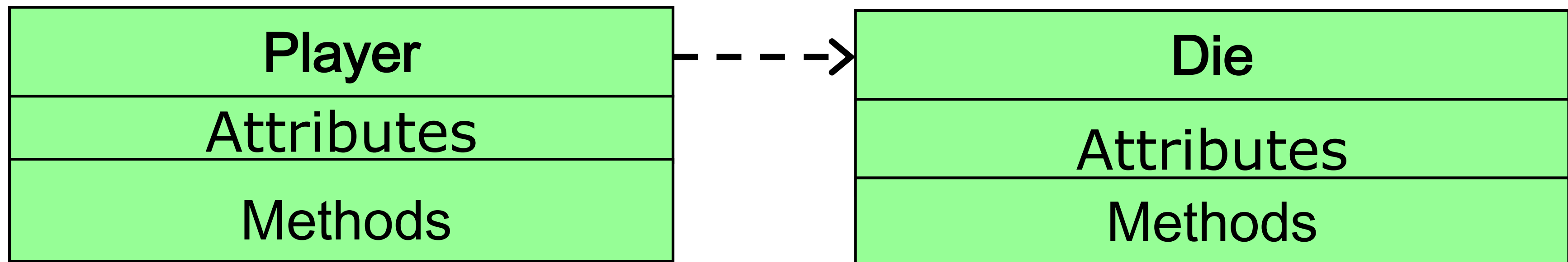
Slide acknowledgements: Internet resources + CS15, Brown University

UML: Quick Introduction

- UML stands for the *Unified Modeling Language*
  - We will cover this in depth in later lectures
- Much detailed than sequence diagrams
- *UML diagrams* show relationships among classes and objects
  - Lines connecting the classes
- A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)

© Vivek Kumar

A Sample UML Class Diagram



© Vivek Kumar

Class Relationships

- The whole point of OOP is that your code replicates real world objects, thus making your code readable and maintainable
- When we say real world, the real world has relationships
- When writing a program, need to keep in mind “big picture”—how are different classes related to each other?

© Vivek Kumar

Most Common Class Relationships

- **Composition**
  - A “contains” B
- **Association**
  - A “knows-about” B
- **Dependency**
  - A “depends on” B
- Inheritance
  - HarleyDavidson “is-a” Bike

© Vivek Kumar

Composition Relationship

- Class A **contains** object of class B
  - A **instantiate** B
- Thus A knows about B and can call methods on it
- But this is not symmetrical! B can't automatically call methods on A
- Lifetime?
  - The death relationship
  - Garbage collection of A means B also gets garbage collected

© Vivek Kumar

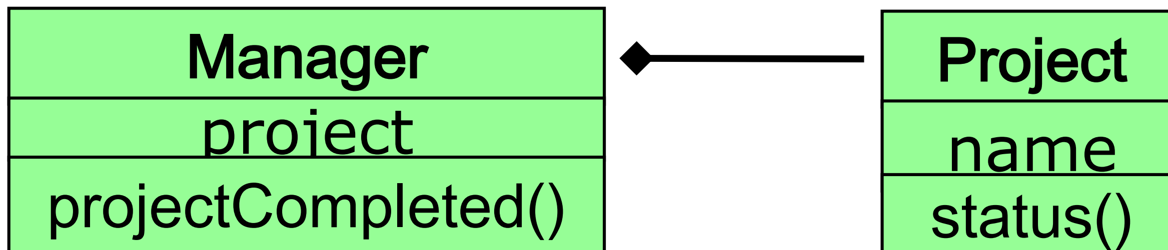
Composition in UML

- Represented by a solid arrow with diamond head
- In below UML diagram, A is composed of B

© Vivek Kumar

Composition Example (1/2)

- Manager is fixed for a project and is responsible for the timely completion of the project. If manager leaves, project is ruined




```
class Project {
   private String name;
   public boolean status() { ... }
   .....
}
// A manager is fixed for a project
class Manager {
   private Project project;
   public Manager() {
     this.project = new Project("ABC");
   }
   public boolean projectCompleted() {
     return project.status();
   }
}
```

© Vivek Kumar

Composition Example (2/2)

- PetShop **contains** a DogGroomer
- Composition relationship because PetShop itself instantiates a DogGroomer with
  "new DogGroomer();"
- Since PetShop created a DogGroomer and stored it in an instance variable, all PetShop's methods "know" about the _groomer and can access it

```
public class PetShop {

  private DogGroomer _groomer;

  public PetShop() {
    _groomer = new DogGroomer();
    this.testGroomer();
  }

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

}
```

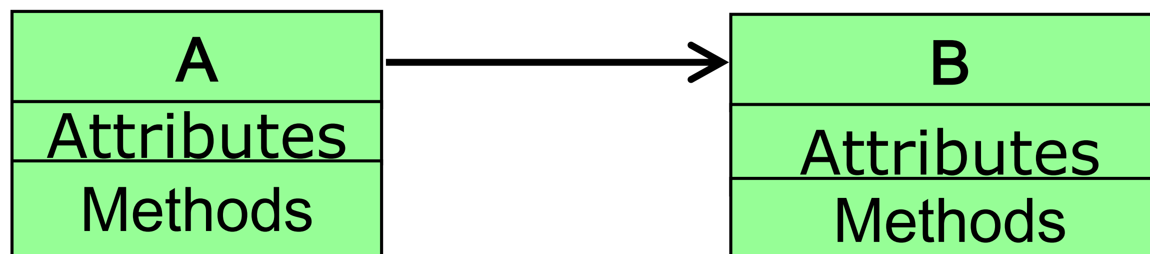

Andries van Dam © 2016 9/22/16

Association Relationship

- Association is a relationship between two objects
- Class A and class B are **associated** if A “knows about” B, but B is not a component of A
- But this is not symmetrical! B “doesn’t knows about” A
- **Class A holds a class level reference to class B**
- Lifetime?
    - Objects of class A and B have their own lifetime, i.e., they can exist without each other

Association in UML

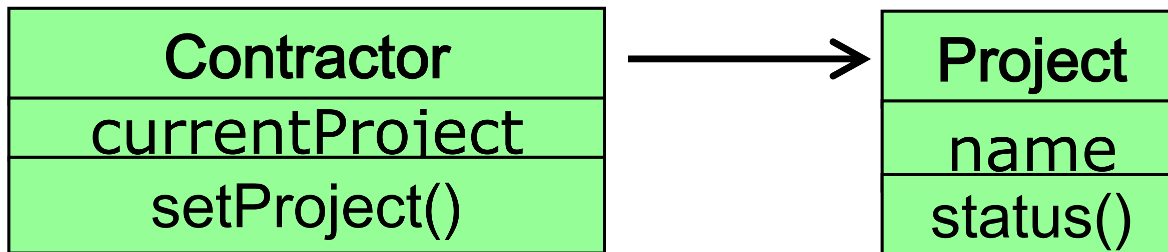
- Represented by a solid arrow
- In below UML diagram, A holds a reference of B



© Vivek Kumar

Association Example (1/4)

- A contractor’s project keep’s changing as per company’s policy and contractor’s performance




```
class Project {
   private String name;
   public boolean status() { ... }
   .....
}
// Contractor’s project keep changing
class Contractor {
   private Project currentProject;
   public Contractor(Project proj) {
     this.currentProject = proj;
   }
   public void setProject(Project proj){
     this.currentProject = proj;
   }
}
```

© Vivek Kumar

Associations Example (2/4)

- **Association** means that one object knows about another object that is not one of its components

```
public class DogGroomer {


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

  public void groom(Dog shaggyDog) {
    shaggyDog.setHairLength(1);
  }
}
```


Andries van Dam © 2016 9/22/16

Associations Example (2/4)

- As noted, PetShop contains a DogGroomer, so it can send messages to the DogGroomer
- But what if the DogGroomer needs to send messages to the PetShop she works in?
    - the DogGroomer probably needs to know several things about her PetShop: for example, operating hours, grooming supplies in stock, customers currently in the shop...

```
public class DogGroomer {



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

  public void groom(Dog shaggyDog) {
    shaggyDog.setHairLength(1);
  }
}
```


Andries van Dam © 2016 9/22/16

Associations Example (2/4)

- The PetShop keeps track of such information in its properties
- Can set up an **association** so that DogGroomer can send her PetShop messages to retrieve information she needs

```
public class DogGroomer {



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

  public void groom(Dog shaggyDog) {
    shaggyDog.setHairLength(1);
  }
}
```


Andries van Dam © 2016 9/22/16

Associations Example (2/4)

- This is what the full association looks like
- Let's break it down line by line
- **But note we're not yet making use of the association in this fragment**

```
public class DogGroomer {

  private PetShop _petShop;

  public DogGroomer(PetShop myPetShop) {
    _petShop = myPetShop; // store the assoc.
  }

  public void groom(Dog shaggyDog) {
    shaggyDog.setHairLength(1);
  }
}
```


Andries van Dam © 2016 9/22/16

Associations Example (2/4)

- We declare an instance variable named `_petShop`
- We want this variable to record the instance of `PetShop` that the `DogGroomer` belongs to

```
public class DogGroomer {

  private PetShop _petShop;

  public DogGroomer(PetShop myPetShop) {
    _petShop = myPetShop; // store the assoc.
  }

  public void groom(Dog shaggyDog) {
    shaggyDog.setHairLength(1);
  }
}
```


Andries van Dam © 2016 9/22/16

Associations Example (2/4)

- Modified DogGroomer's constructor to take in a parameter of type PetShop
- Constructor will refer to it by the name myPetShop
- Whenever we instantiate a DogGroomer, we'll need to pass it an instance of PetShop as an argument. Which? The PetShop instance that created the DogGroomer, hence use this

```
public class DogGroomer {
   private PetShop _petShop;

   public DogGroomer(PetShop myPetShop) {
      _petShop = myPetShop; // store the assoc.
   }
   //groom method elided
}
```

```
public class PetShop {
   private DogGroomer _groomer;

   public PetShop() {
      _groomer = new DogGroomer(this);
      this.testGroomer();
   }

   //testGroomer() elided
}
```


Andries van Dam © 2016 9/22/16

Associations Example (2/4)

- Now store myPetShop in instance variable _petShop
- _petShop now points to same PetShop instance passed to its constructor
- After constructor has been executed and can no longer reference myPetShop, any DogGroomer method can still access same PetShop instance by the name _petShop

```
public class DogGroomer {

  private PetShop _petShop;

  public DogGroomer(PetShop myPetShop) {
    _petShop = myPetShop; // store the assoc.
  }

  public void groom(Dog shaggyDog) {
    shaggyDog.setHairLength(1);
  }
}
```


Andries van Dam © 2016 9/22/16

Associations Example (2/4)

- Let's say we've written an accessor method and a mutator method in the `PetShop` class:
  `getClosingTime()` and
  `setNumCustomers(int customers)`
- If the `DogGroomer` ever needs to know the closing time, or needs to update the number of customers, she can do so by calling
  - `getClosingTime()`
  - `setNumCustomers(int customers)`

```
public class DogGroomer {

  private PetShop _petShop;
  private Time _closingTime;

  public DogGroomer(PetShop myPetShop) {
    _petShop = myPetShop; // store assoc.
    _closingTime = myPetShop.getClosingTime();
    _petShop.setNumCustomers(10);
  }
}
```


Andries van Dam © 2016 9/22/16

Association: Under the Hood (1/5)

```
public class PetShop {
   private DogGroomer _groomer;

   public PetShop() {
     _groomer = new DogGroomer(this);
     this.testGroomer();
   }

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

```
public class DogGroomer {
   private PetShop _petShop;

   public DogGroomer(PetShop myPetShop) {
     _petShop = myPetShop;
   }

   /* groom and other methods elided for this
   example */
}
```

Somewhere in memory...

Andries van Dam © 2016 9/22/16

Association: Under the Hood (2/5)

```
public class PetShop {
   private DogGroomer _groomer;

   public PetShop() {
      _groomer = new DogGroomer(this);
      this.testGroomer();
   }

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

```
public class DogGroomer {
   private PetShop _petShop;

   public DogGroomer(PetShop myPetShop) {
      _petShop = myPetShop;
   }

   /* groom and other methods elided for this
   example */
}
```

Somewhere in memory...

Somewhere else in our code, someone calls new PetShop(). An instance of PetShop is created somewhere in memory and PetShop's constructor initializes all its instance variables (just a DogGroomer here)

Andries van Dam © 2016 9/22/16

Association: Under the Hood (3/5)

```
public class PetShop {
   private DogGroomer _groomer;

   public PetShop() {
      _groomer = new DogGroomer(this);
      this.testGroomer();
   }

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

```
public class DogGroomer {
   private PetShop _petShop;

   public DogGroomer(PetShop myPetShop) {
      _petShop = myPetShop;
   }

   /* groom and other methods elided for this
   example */
}
```

Somewhere in memory...

The PetShop instantiates a new DogGroomer, passing itself in as an argument to the DogGroomer's constructor (remember the this keyword?)

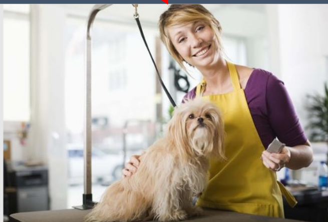
Andries van Dam © 2016 9/22/16

Association: Under the Hood (4/5)

```
public class PetShop {
   private DogGroomer _groomer;

   public PetShop() {
      _groomer = new DogGroomer(this);
      this.testGroomer();
   }

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

```
public class DogGroomer {
   private PetShop _petShop;

   public DogGroomer(PetShop myPetShop) {
      _petShop = myPetShop;
   }

   /* groom and other methods elided for this
   example */
}
```

Somewhere in memory...



When the DogGroomer's constructor is called, its parameter, myPetShop, points to the same PetShop that was passed in as an argument.

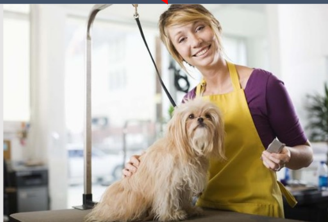
Andries van Dam © 2016 9/22/16

Association: Under the Hood (5/5)

```
public class PetShop {
   private DogGroomer _groomer;

   public PetShop() {
     _groomer = new DogGroomer(this);
     this.testGroomer();
   }

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

```
public class DogGroomer {
   private PetShop _petShop;

   public DogGroomer(PetShop myPetShop) {
     _petShop = myPetShop;
   }

   /* groom and other methods elided for this
   example */
}
```

Somewhere in memory...



The DogGroomer sets its _petShop instance variable to point to the same PetShop it received as an argument. Now it "knows about" the petShop that instantiated it! And therefore so do all its methods...

Andries van Dam © 2016 9/22/16

Associations Example (3/4)

- Here we have the class `Professor`
- We want `Professor` to know about his TAs—he didn't create them or vice versa, hence no containment – they are peer objects
- Let's set up associations!

```
public class Professor {

  // declare instance variables here
  // and here…
  // and here…
  // and here!

  public Professor(/* parameters */) {


    // initialize instance variables!
    // …
    // …
    // …
  }

  /* additional methods elided */
}
```


Andries van Dam © 2016 9/22/16

Associations Example (3/4)

- The `Professor` needs to know about 4 TAs, all of whom will be instances of the class `TA`
- Once he knows about them, he can call methods of the class TA on them: `remindTA`, `runRefresherModule`, etc.
- Take a minute and try to fill in this class

```
public class Professor {

    // declare instance variables here
    // and here…
    // and here…
    // and here!

    public Professor(/* parameters */) {


        // initialize instance variables!
        // …
        // …
        // …
    }

    /* additional methods elided */
}
```


Andries van Dam © 2016 9/22/16

Associations Example (3/4)

- Here's our solution!
- Remember, you can choose your own names for the instance variables and parameters
- The Professor can now send a message to one of his TAs like this:

```
_ta1._runRefresherModule();
```

```
public class Professor {

   private TA _ta1;
   private TA _ta2;
   private TA _ta3;
   private TA _ta4;

   public Professor(TA firstTA,
         TA secondTA, TA thirdTA
         TA fourthTA) {

      _ta1 = firstTA;
      _ta2 = secondTA;
      _ta3 = thirdTA;
      _ta4 = fourthTA;
   }

   /* additional methods elided */
}
```


Andries van Dam © 2016 9/22/16

Associations Example (3/4)

- We've got the `Professor` class down
- Now let's create a professor and TAs from a class that contains all of them: `Course`
- Try and fill in this class!
    - You can assume that the `TA` class takes no parameters in its constructor.

```
public class Course {

    // declare Professor instance var.
    // declare four TA instance vars.
    // …
    // …
    // …

    public Course() {
      // instantiate the four TAs
      // …
      // …
      // instantiate the professor!

    }
}
```


Andries van Dam © 2016 9/22/16

Associations Example (3/4)

- We declare _vivek, _akanksha, _akash, _alind and _abhiprayah as instance variables
- In the constructor, we instantiate them
- Since the constructor of Professor takes in 4 TAs, we pass in _akanksha, _akash, _alind and _abhiprayah

```
public class Course {

   private Professor _vivek;
   private TA _akanksha;
   private TA _akash;
   private TA _alind;
   private TA _abhiprayah;

   public Course() {
     _akanksha = new TA();
     _akash = new TA();
     _alind = new TA();
     _abhiprayah = new TA();
     _vivek = new Professor(_akanksha,
                 _akash , _alind , _abhiprayah);
   }
}
```


Andries van Dam © 2016 9/22/16

Associations Example (3/4)

```
public class Professor {

   private TA _ta1;
   private TA _ta2;
   private TA _ta3;
   private TA _ta4;

   public Professor(TA firstTA,
           TA secondTA, TA thirdTA
           TA fourthTA){

     _ta1 = firstTA;
     _ta2 = secondTA;
     _ta3 = thirdTA;
     _ta4 = fourthTA;
     _ta1.runRefresherModule();
   }
   /* additional methods elided */
}
```

```
public class Course {

   private Professor _vivek;
   private TA _akanksha;
   private TA _akash;
   private TA _alind;
   private TA _abhiprayah;

   public Course() {
     _akanksha = new TA();
     _akash = new TA();
     _alind = new TA();
     _abhiprayah = new TA();
     _vivek = new Professor(_akanksha,
          _akash , _alind , _abhiprayah);
   }
}
```


Andries van Dam © 2016 9/22/16

Associations Example (4/4)

- What if we want the TAs to know about `Professor` too?
- Need to set up another association
- Can we just do the same thing?

```
public class Course {

  private Professor _vivek;
  private TA _akanksha;
  private TA _akash;
  private TA _alind;
  private TA _abhiprayah;

  public Course() {
    _akanksha = new TA();
    _akash = new TA();
    _alind = new TA();
    _abhiprayah = new TA();
    _vivek = new Professor(_akanksha,
            _akash , _alind , _abhiprayah);
  }
}
```


Andries van Dam © 2016 9/22/16

Associations Example (4/4)

- This doesn't work: when we instantiate _akanksha, _akash, _alind and _abhiprayah, we would like to pass them an argument, _vivek
- But _vivek hasn't been instantiated yet! And can't initialize _vivek first because the TAs haven't been created yet…
- What can we try instead?

```
public class Course {

  private Professor _vivek;
  private TA _akanksha;
  private TA _akash;
  private TA _alind;
  private TA _abhiprayah;

  public Course() {
    _akanksha = new TA();
    _akash = new TA();
    _alind = new TA();
    _abhiprayah = new TA();
    _vivek = new Professor(_akanksha,
          _akash , _alind , _abhiprayah);
  }
}
```

Associations Example (4/4)

- Need a way to pass `_vivek` to `_akanksha`, `_akash`, `_alind` and `_abhiprayah` **after** we instantiate `_vivek`
- Use a new method, `setProf`, and pass each TA `_vivek`

```
public class Course {

  private Professor _vivek;
  private TA _akanksha;
  private TA _akash;
  private TA _alind;
  private TA _abhiprayah;

  public Course() {
    _akanksha = new TA();
    _akash = new TA();
    _alind = new TA();
    _abhiprayah = new TA();
    _vivek = new Professor(_akanksha,
    _akash , _alind , _abhiprayah);

  }
}
```

Andries van Dam © 2016 9/22/16

Associations Example (4/4)

```
public class TA {

   private Professor _professor;

   public TA() {

     //Other code elided

   }

   public void setProf(Professor prof) {
     _professor = prof;
   }
}
```

- Now each TA will know about `_vivek`!

```
public class Course {

   private Professor _vivek;
   private TA _akanksha;
   private TA _akash;
   private TA _alind;
   private TA _abhiprayah;

   public Course() {
     _akanksha = new TA();
     _akash = new TA();
     _alind = new TA();
     _abhiprayah = new TA();
     _vivek = new Professor(_akanksha,
          _akash , _alind , _abhiprayah);

     _akanksha.setProf(_vivek);
     _akash.setProf(_vivek);
     _alind.setProf(_vivek);
     _abhiprayah.setProf(_vivek);
   }
}
```


Andries van Dam © 2016 9/22/16

Question

- What happens if `setProf` is never called?
- Will the TAs be able to call methods on the `Professor`?

Andries van Dam © 2016 9/22/16

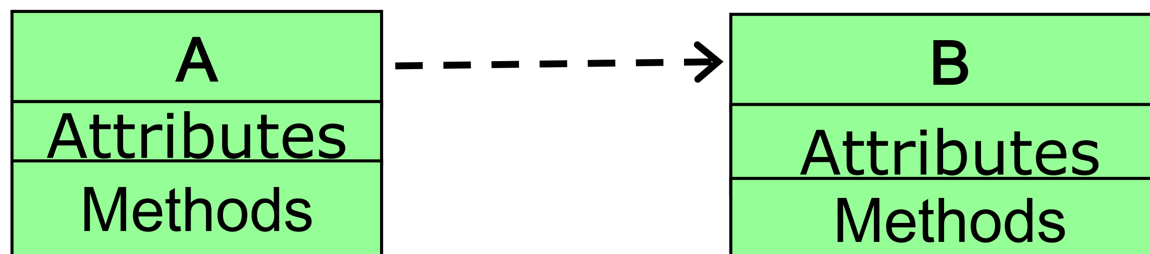
Dependency

- Class A **depends** on class B if A cannot carry out its work without B, but B is neither a component of A nor it has association with A
- **A is requesting service from an object of class B**
  - A or B **“doesn’t know”** about each other (no association)
  - A or B **“doesn’t contain”** each other (no composition)
- But this is not symmetrical! B doesn’t depends on A

© Vivek Kumar

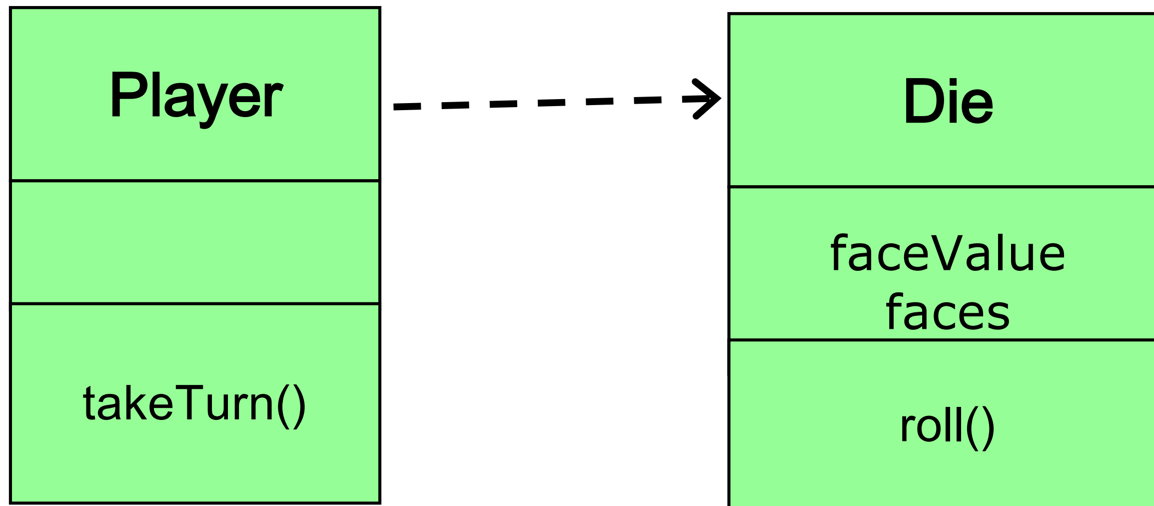
Dependency in UML

- Represented by a dashed arrow starting from the dependent class to its dependency
  - A is dependent on B
  - A is requesting service from B



© Vivek Kumar

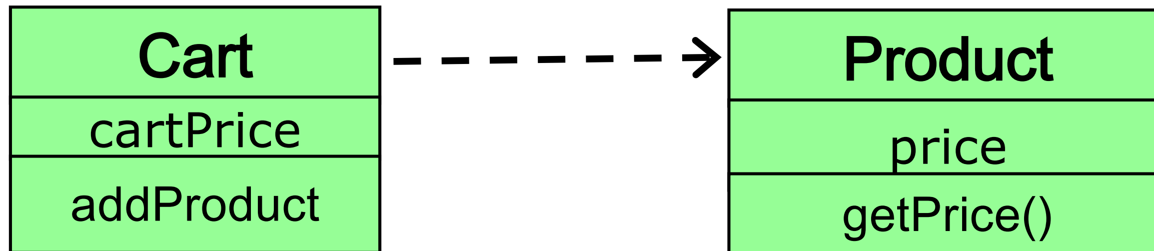
Dependency Example (1/3)




```
class Die {
  private int faceValue, faces;
  .....
  public void roll() { ..... }
}

class Player {
  public void takeTurn(Die die) {
    die.roll();
  }
}
```

© Vivek Kumar

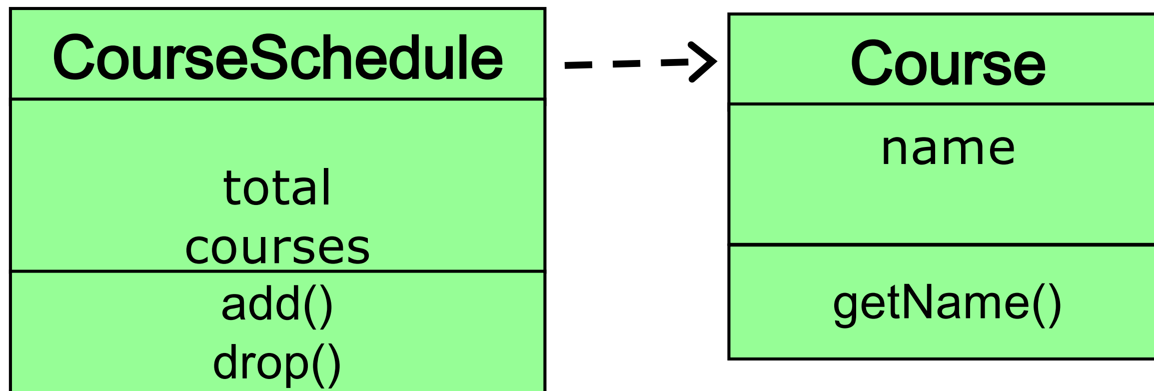
Dependency Example (2/3)




```
class Product {
   private double price;
   .....
   public double getPrice() { ..... }
}

class Cart {
   private double cartPrice;
   public void addProduct(Product p) {
     cartPrice += p.getPrice();
   }
}
```

© Vivek Kumar

Dependency Example (3/3)




```
class Course {
   private String name;
   .....
   public String getName() { ..... }
}

class CourseSchedule {
   private int total;
   private String courses[];
   public void addCourse(Course c) {
     courses[total++] = c.getName();
   }
   .....
}
```

© Vivek Kumar

Next Lecture

- Interfaces in Java
- Quiz-1
  - Syllabus: Lecture 01-03
- Assignment-1 to be announced on 09/08 evening