Object Interactions

Object Oriented Programming
2016375 - 5

Camilo López

Outline

- Events Drive Object Collaboration
- Declaring Methods
- Methods Implement Business Rules
- Objects As the Context for Method Invocation
- Method overloading
- Message Passing Between Objects
- Delegation
- Obtaining Handles on Objects
- Objects as Clients and Suppliers
- Information Hiding/Accesibility
- Accessing Private Features from Client Code
- The Power of Encapsulation Plus Information Hiding
- Exceptions to the Private/Public Rule
- Constructors

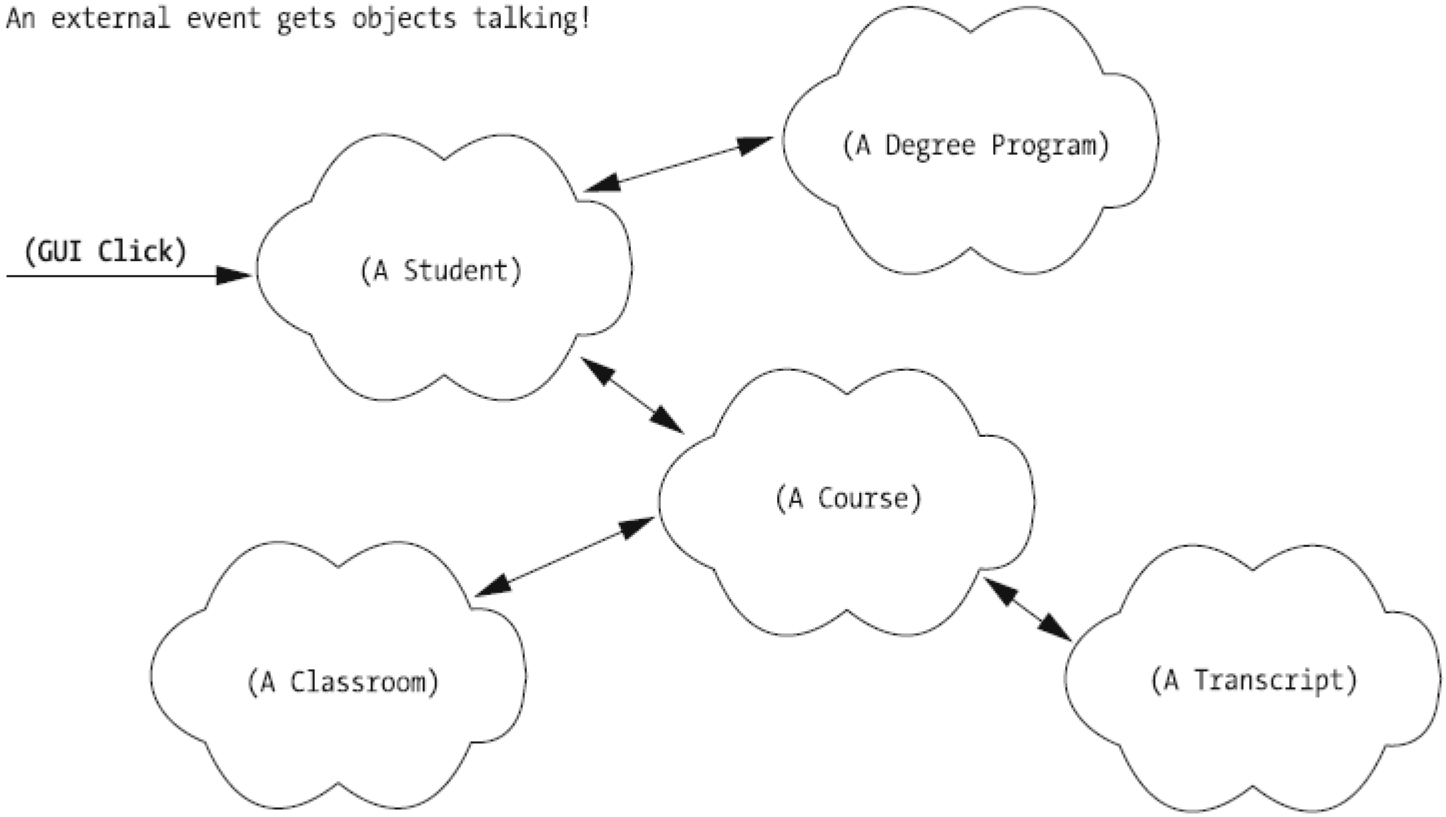
Events Drive Object Collaboration

- At its simplest, the process of object-oriented software development involves the following four basic steps:

1. Properly establishing the functional requirements for, and overall mission of, an application
2. Designing the appropriate classes—their data structures, behaviors, and relationships with one another—necessary to fulfill these requirements and mission
3. Instantiating these classes to create the appropriate types and number of object instances
4. Setting these objects in motion through **external triggering events**

Events Drive Object Collaboration

Communication between objects



request to register for a course made by a student user

Declaring Methods

If an object A wants to request some service of an object B, A needs to know the specific language to communicate with B.

- *Object A needs to be clear as to exactly which of B's methods/services A wants B to perform.*
- *Depending on the service request, object A may need to give B some additional information so that B knows exactly how to proceed.*
- *Object B in turn needs to know whether object A expects B to report back the outcome of what it has been asked to do.*

AM I WAITING FOR SOMETHING?

Declaring Methods

- Method Headers

```
boolean registerForCourse(String courseId, int secNo)
```

return type **method name** **List of formal parameters**

- Parsing Arguments to Methods
  - To provide it with the (optional) “fuel” necessary to do its job
  - To otherwise guide its behavior in some fashion
- Method Body
  - program the details of what the method is to do
  - Features may be declared in any order
  - return statements

Methods Implement Business Rules

- The logic contained within a method body defines the **business logic** for an abstraction.

```
boolean isHonorsStudent() {
    boolean result = false;
    if (gpa >= 3.5) {
        result = true;
    }
    return result;
}
```

If a student has a grade point average (GPA) of 3.5 or higher, then he or she is an honors student.

Methods Implement Business Rules

- The logic contained within a method body defines the **business logic** for an abstraction.

```
boolean isHonorsStudent() {
    boolean result = false;
    if (gpa >= 3.5 &&
        numCourses >= 3 &&
        no grades lower than a B) {
            result = true;
    }
    return result;
}
```

In order for a student to be considered an honors student, the student must

a) Have a grade point average (GPA) of 3.5 or higher

b) Have taken at least three courses

c) Have received no grade lower than "B" in any of these courses

Objects As the Context for Method Invocation

- Methods in an OOPL differ from functions in a non-OOPL in that
  - Functions are executed by the programming environment as a whole, whereas
  - Methods are executed by specific objects

```
// Instantiate two Student objects.
Student x = new Student();
Student y = new Student();

x.registerForCourse("MATH 101", 10);
```

Invoke the registerForCourse method on Student object x, asking it to register for course MATH 101, section 10; Student y is unaffected.

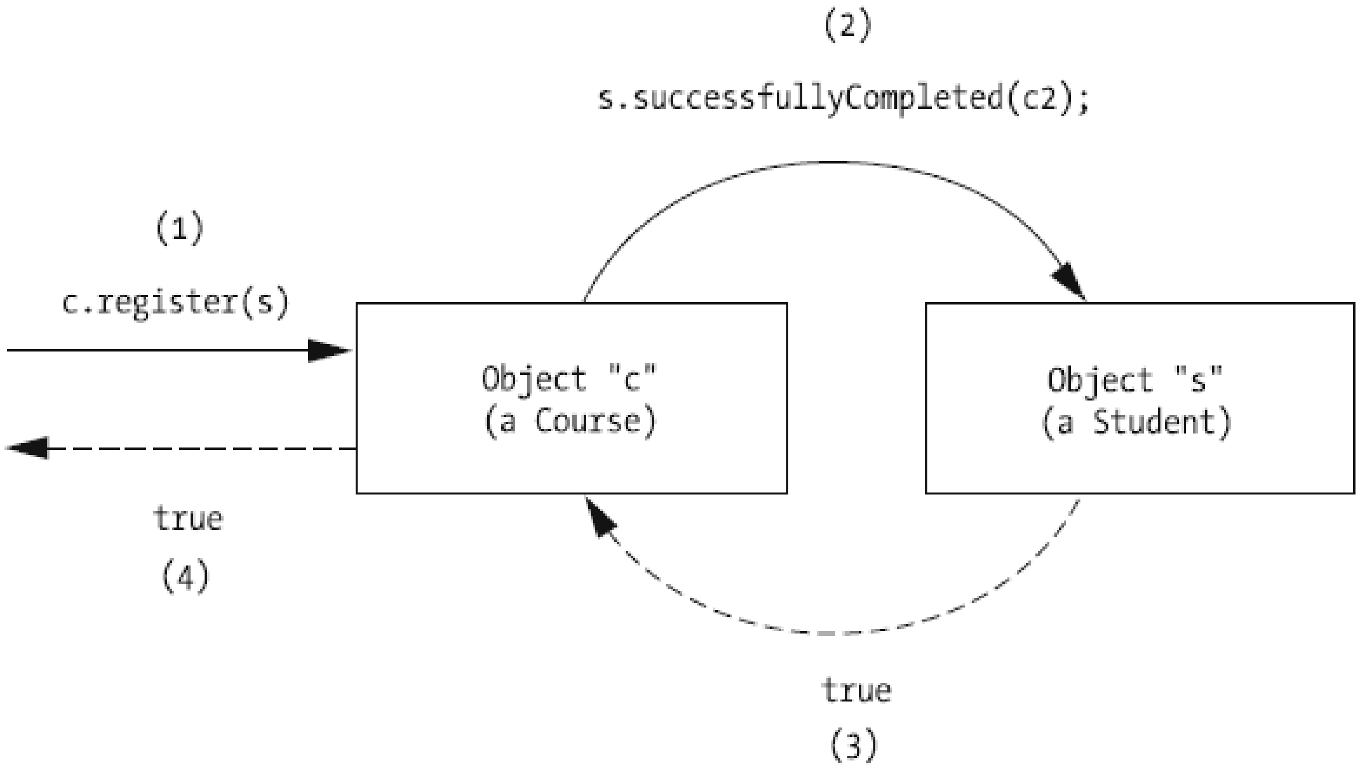
Method overloading

- **Overloading** is a language mechanism that allows two or more different methods belonging to the same class to have the ***same* name** as long as they have ***different* argument signatures.**

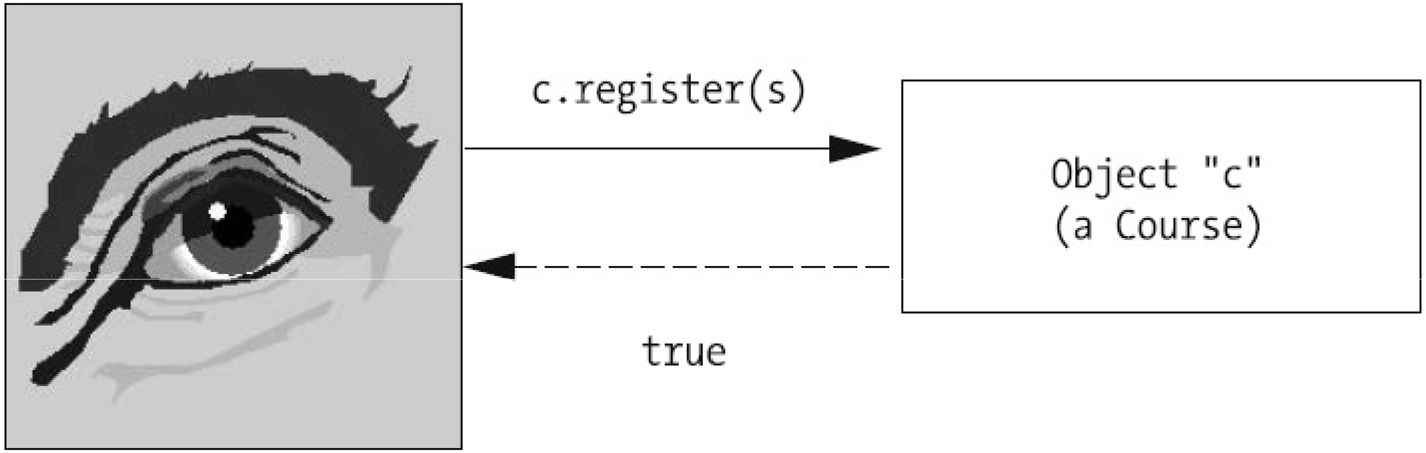
```
void print(String fileName) { ... // version #1
void print(int detailLevel) { ... // version #2
void print(int detailLevel, String fileName) { ... // version #3
int print(String reportTitle, int maxPages) { ... // version #4
boolean print() { ... // version #5
```

- Note that **there is no such thing as *attribute* overloading**; that is, if a class tries to declare two attributes with the same name

Message Passing Between Objects



Delegation



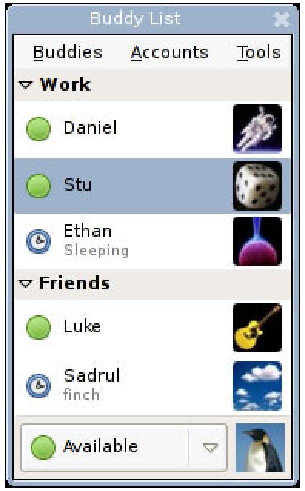
The fact that delegation has occurred between objects is often transparent to the initiator of a message, as well.

Obtaining Handles on Objects

The only way that an object A can pass a message to an object B is if A has access to a reference to/handle on B. This can happen in several different ways.

1. ***Object A might maintain a reference to B as one of A's attributes.***

```
public class Student {
    // Attributes.
    String name;
    Professor facultyAdvisor;
    // etc.
```



Obtaining Handles on Objects

The only way that an object A can pass a message to an object B is if A has access to a reference to/handle on B. This can happen in several different ways.

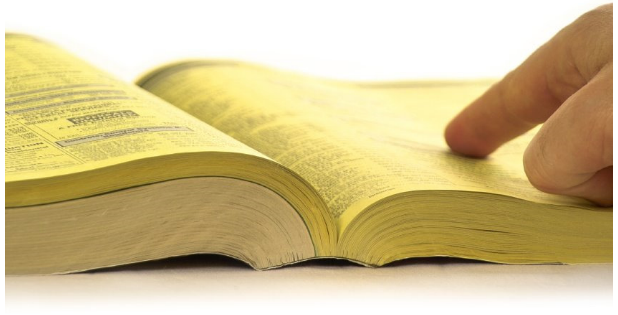
2. ***Object A may be handed a reference to B as an argument of one of A's methods.***

```
// …
   String name;
   Student student;
   Course course;
   //…
   c.register(s);
```

Obtaining Handles on Objects

The only way that an object A can pass a message to an object B is if A has access to a reference to/handle on B. This can happen in several different ways.

3. ***A reference to object B may be made "globally available" to the entire application***



Eric T. Cartman is on Facebook Sign up for Facebook to connect with Eric T. Cartman.

Eric T. Cartman

Simply the best cartoon creation ever...

Location:
South Park, Colorado.

6 of 366,543 fans

Basic Info

Location: South Park, Colorado.

Detailed Info

Website: http://www.southparkstudios.com
http://en.wikipedia.org/wiki/Eric_Cartma...

Personal Information: Eric Theodore Cartman
Age 9
Student at South Park Elementary school Colorado
.
I am out of control. Yeah, I use drugs. I can do what I waunt, biatch! Yeah, I have sex, and I don't use protection! It's my hot body; I'll do what I waunt! I don't go to school and I kill people! What-evah! I'll do what I waunt!
.

Personal Interests: Chinpokemon
Wii
Cheesy Poofs
Abusing Kyle Broflovski
Getting what a want and making everyone respect my authoritaah!
.

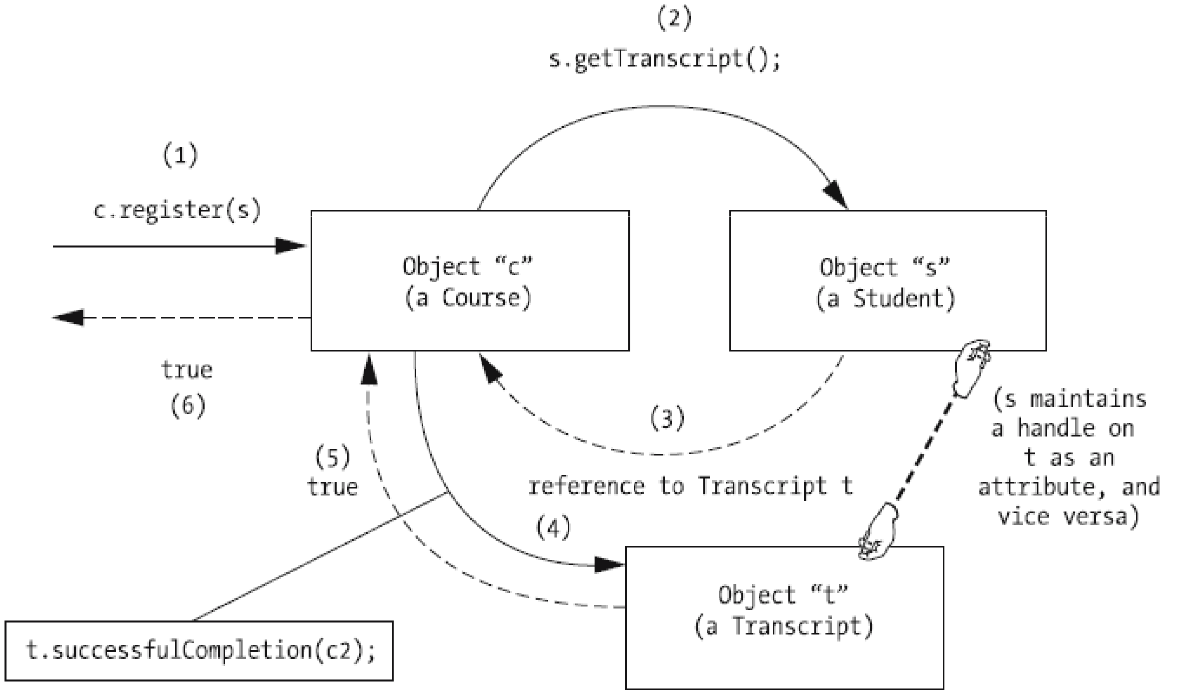
Obtaining Handles on Objects

The only way that an object A can pass a message to an object B is if A has access to a reference to/handle on B. This can happen in several different ways.

4. ***Object A may have to explicitly request a handle/reference to B by calling a method on some third object C.***

Obtaining Handles on Objects

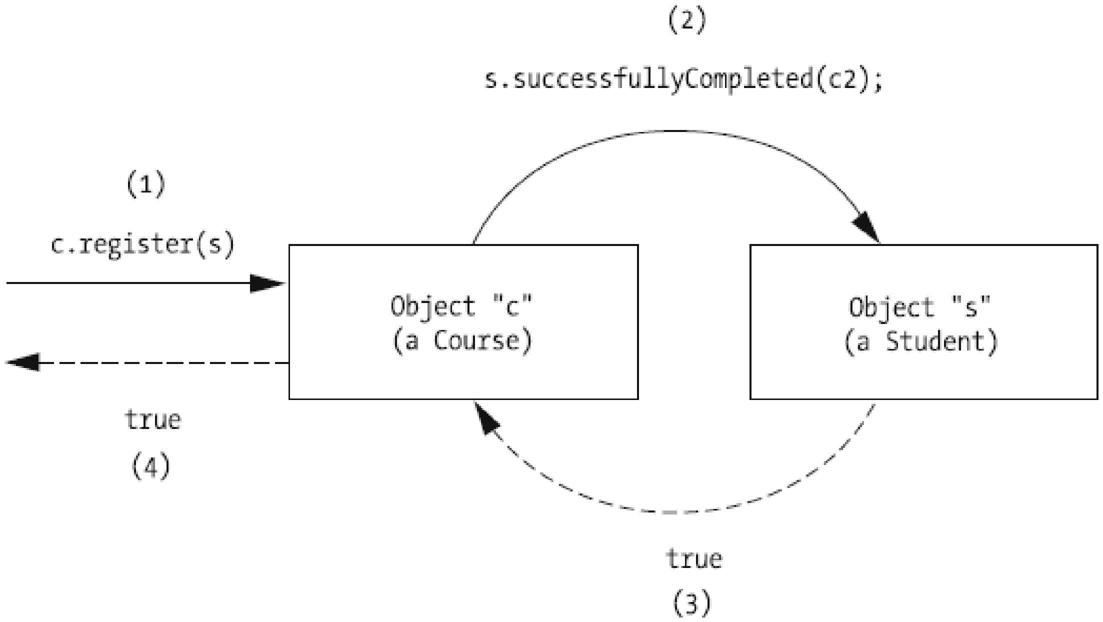
4. ***Object A may have to explicitly request a handle/reference to B by calling a method on some third object C.***



This is not unlike the real-world situation in which person A asks person C for person B's phone number, without telling C why they want to call B.

Objects as Clients and Suppliers

- we can consider Course object c to be a **client** of Student object s, because c is requesting that s perform one of its methods—namely, getTranscript—as a ***service*** to c.



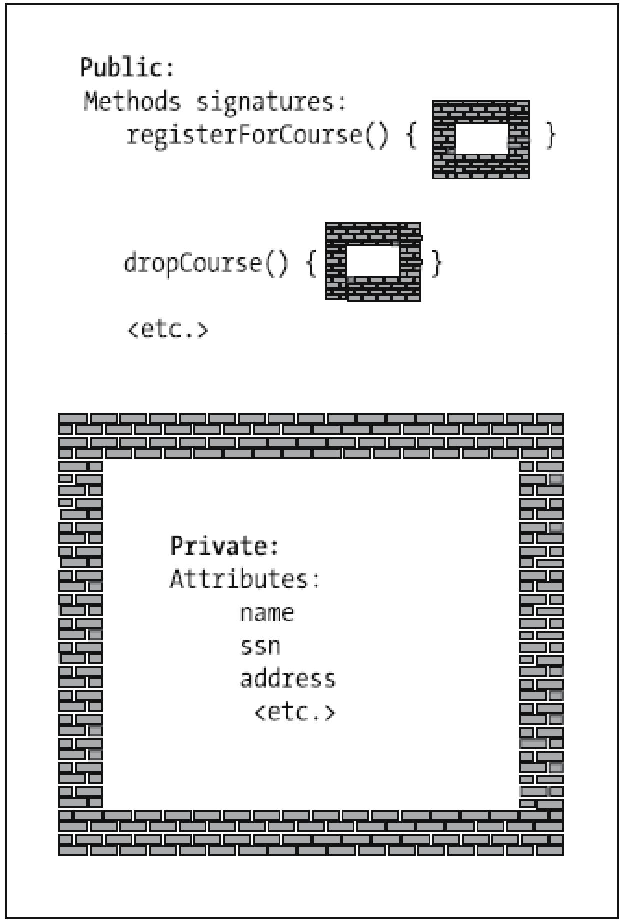
Objects as Clients and Suppliers

```
public class Course {
    //...
    public boolean register(Student s) {
        boolean outcome = false;
        // Request a handle on Student s's Transcript object.
        Transcript t = s.getTranscript();

        if (t.successfulCompletion(c2)) {
            outcome = true;
        }else {
            outcome = false;
        }
        return outcome;
    }
    // etc.
```

Course **class is considered to be client code relative to both** Student **object** s **and** Transcript **object** t

Information Hiding/Accesibility



Public:
Methods signatures:
registerForCourse() { }

dropCourse() { }

<etc.>

Private:
Attributes:
name
ssn
address
<etc.>

We are not permitted to access private methods/attributes directly via dot notation from client code.

```
public class Student {
    private String name;
    //…
}
```

```
public class MyProgram {
    public static void main(String[] args) {
        Student x = new Student();

        x.name = "123-45-6789";        ❌
        // etc.
    }
}
```

Information Hiding/Accesibility

- Accessing the Features of a Class from Within Its Own Methods
  - we can access all of a given class's features, ***regardless of their accessibility***, from within any of that class's ***own*** method bodies; that is, public/private designations only affect access to a feature ***from outside the class itself*** (i.e., ***from client code***).
  - dot notation is not required
  - Java keyword this

Information Hiding/Accesibility

- Accessing the Features of a Class from Within Its Own Methods

```
public class Student {
    private String name;
    private String ssn;

    public void printStudentInfo() {
        // Accessing attributes of the Student class.
        System.out.println("Name: " + name);
        System.out.println("Student ID: " + ssn);
    }

    public void updateName(Srting n) {
        name = n;
    }
}
```

Information Hiding/Accesibility

- Accessing the Features of a Class from Within Its Own Methods

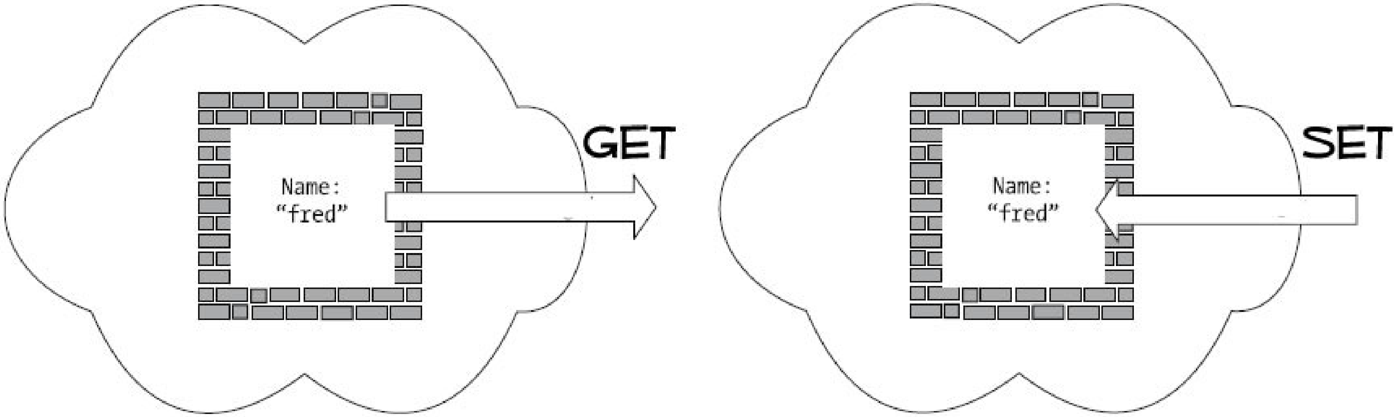
```
public class Student {
    private String name;
    private String ssn;

    public void printStudentInfo() {
        // Accessing attributes of the Student class.
        System.out.println("Name: " + this.name);
        System.out.println("Student ID: " + this.ssn);
    }

    public void updateName(Srting n) {
        name = n;
    }
}
```

Accessing Private Features from Client Code

- If private features can't be accessed outside of an object's own methods, how does client code ever manipulate them? Through ***public*** features, of course!
- we may empower an object to have the "final say" in whether or not what client code is trying to do to its attributes is valid
- Accessor Methods



Accessing Private Features from Client Code

- Get-Set Method Headers

```
public class Student {
    private String name;
    private boolean honorsStudent;
    //…
    public String getName(){
        return name;
    }
    public void setName(String nom){
         name = nom;
    }
    public boolean isHonorsStudent() {
        return honorsStudent;
    }
}
```

public type getVariableName()

public void setVariableName()

public boolean isBooleanVarName()

Accessing Private Features from Client Code

- Get-Set Method Bodies
  - one-liners, or…

```
public void setName(String newName) {
if (newName contains full first name) {
        newName = newName (first name converted to a single character
                                        followed by a period);
}

    newName = uppercase version of newName;

    name = newName;
```

First, reformat the newName, as necessary …

Accessing Private Features from Client Code

- Get-Set Method Bodies
  - one-liners, or…

```
public void setName(String newName) {
if (newName contains full first name) {
        newName = newName (first name converted to a single character
                                        followed by a period);
}

    newName = uppercase version of newName;

    name = newName;
```

Next, convert newName to all uppercase.

Accessing Private Features from Client Code

- Get-Set Method Bodies
  - one-liners, or…

```
public void setName(String newName) {
if (newName contains full first name) {
        newName = newName (first name converted to a single character
                                        followed by a period);
}

    newName = uppercase version of newName;

    name = newName;
```

Only then, we update the name attribute with the (modified) value.

The Power of Encapsulation Plus Information Hiding

- The object itself is responsible for the integrity of its own data

http://www.silhouettesclipart.com/wp-content/uploads/2008/06/couple-silhouette-clip-art.jpg

- Preventing Unauthorized Access to Encapsulated Data
- Helping to Ensure Data Integrity (Date formats,…)
- Limiting “Ripple Effects” When Private Features Change

Exceptions to the Private/Public Rule

- Exception #1: Internal Housekeeping Attributes

```
    public class Student {
    private int countOfDsAndFs;
    // ...
        public boolean onAcademicProbation() {
            boolean onProbation = false;

            if (countOfDsAndFs > 3) {
                onProbation = true;
            }
        return onProbation;
    }
// ...
}
```

Exceptions to the Private/Public Rule

- Exception #2: Internal Housekeeping Methods

```
public class Student {
private double gpa;
private int totalCoursesTaken;
private int countOfDsAndFs;

public void completeCourse(String courseName,
                           int creditHours, char grade) {
    if (grade == 'D' || grade == 'F') {
    countOfDsAndFs++;
    }

    totalCoursesTaken = totalCoursesTaken + 1;

    updateGpa(creditHours, grade);
}

private void updateGpa(int creditHours, char grade){…}
```

Exceptions to the Private/Public Rule

- Exception #3: “Read-Only” Attributes

```
    public class Student {
    private String StudientId;
    // ...

    public String getStudentId() {
        return studentId;
    }

    // The set method is intentionally omitted from the class.
    // ...
}
```

Exceptions to the Private/Public Rule

- Exception #4: Public Attributes

  On rare occasions, a class may declare selected attributes as public for ease of access; ***this is only done when there is no business logic governing the attributes per se.***

```
public class Point {
    // Both attributes are public:
    public double x;
    public double y;
    // etc.
}
```

so that, in client code, we may easily assign values as follows:

```
Point p = new Point();
p.x = 3.7;
```

Constructors

```
Student x = new Student();
```

- Invoking a constructor **objectClassName()** serves as a request to the JVM to construct (instantiate) a brand-new object at run time by allocating enough program memory to house the object's attributes.

"Inflate a new helium balloon of a particular type"

Constructors

- Default Constructors
  - Parameterless
  - Sets all attributes to their zero-equivalent default values

```
Student x = new Student();
```

```
public class Student {
    private String name;
    private String studentId;
    private double gpa;
    private boolean honorsStudent;
    //…
}
```

NAME?	NULL
ID?	NULL
GPA?	0.0
HONORS	FALSE

A John Doe…

Constructors

- Replace the default parameterless Constructor

public Student()

access modifier **NO return type!!** **Class name**

```
public class Student {
    private String name;
    private String studentId;
    private double gpa;
    private boolean honorsStudent;
    //attributes…

    public Student(){
        this.name = "John Doe"
    }
    //methods…
}
```

```
Student x = new Student();
```

NAME?	JOHN DOE
ID?	NULL
GPA?	0.0
HONORS	FALSE

Constructors

- Explicit Constructors

public Student(String name, String studentId)

access modifier **NO return type!!** **Class name** **List of formal parameters**

```
public class Student {
    private String name;
    private String studentId;
    private double gpa;
    private boolean honorsStudent;
    //attributes…

    public Student(String name){
        this.name = name
    }
    //methods…
}
```

```
Student x = new Student(“Juan Pérez”);
```

NAME?	JUAN PÉREZ
ID?	NULL
GPA?	0.0
HONORS	FALSE

Constructors

- Overloading Constructors

```
public class Student {
    private String name;
    private String studentId;
    private double gpa;
    //attributes…

    public Student(String name){
        this.name = name
    }
    public Student(String name, double gpa){
        this.name = name
        this.gpa = gpa
    }

    //methods…
}
```

Student x = new Student(“Juan”);

Student x = new Student(“JP”, 1.0);

Student x = new Student();

↓

It’s an error

The constructor Student() is not defined

Constructors

- Using the “this” Keyword to Facilitate Constructor Reuse

```
public class Student {
    public Student() {
        alert the registrar's office of this student's existence
        transcript = new Transcript();
    }
    public Student(String s) {
        this.setSsn(s);
        alert the registrar's office of this student's existence
        transcript = new Transcript();
    }
    public Student(String s, String n) {
        this.setSsn(s);
        this.setName(n);
        alert the registrar's office of this student's existence
        transcript = new Transcript();
    }
    // etc.
}
```

This code is duplicated from above!

DUPLICATION YET AGAIN!!!

Constructors

- Using the “this” Keyword to Facilitate Constructor Reuse

```
public class Student {
    public Student() {
        alert the registrar's office of this student's existence
        transcript = new Transcript();
    }
    public Student(String s) {
        this();
        this.setSsn(s);
    }
    public Student(String s, String n) {
        this();
        this.setSsn(s);
        this.setName(n);
    }
    // etc.
}
```

REUSE the code of the first constructor within the second! Then, do whatever else extra is necessary for constructor #2.

REUSE the code of the first constructor within the third! Then, do whatever else extra is necessary for constructor #3.

Constructors

- Using the “this” Keyword to Facilitate Constructor Reuse

```
public class Student {
    public Student() {
        alert the registrar's office of this student's existence
        transcript = new Transcript();
    }
    public Student(String s) {
        this();
        this.setSsn(s);
    }
    public Student(String s, String n) {
        this(s);
        this.setName(n);
    }
    // etc.
}
```

REUSE the code of the first constructor within the second! Then, do whatever else extra is necessary for constructor #2.

REUSE the code of the second constructor within the third! Then, do whatever else extra is necessary for constructor #3.

References

- J. Barker, *Beginning Java Objects: From Concepts To Code, Second Edition*, Apress, 2005.