CIS 110: Introduction to Computer Programming

Lecture 22 and 23

Objects, objects, objects

(§ 8.1-8.4)

Outline

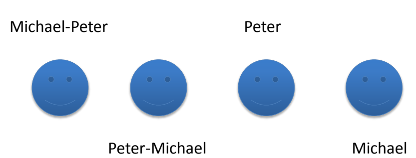
- Object-oriented programming.
- What is an object?
- Classes as blueprints for objects.
- Encapsulation

Any questions?

- Questions, questions, questions?

My life story

The awful truth



A horrible incident

Revenge

MHOA

11/28/2011 CIS 110 (11fa) - University of Pennsylvania 7

Object-oriented programming

11/28/2011 CIS 110 (11fa) - University of Pennsylvania 8

Procedural programming

Reasoning about programs as a set of interacting procedures/methods.

11/28/2011 CIS 110 (11fa) - University of Pennsylvania 9

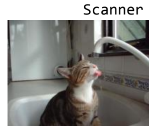
Object-oriented programming

Reasoning about programs as a set of interacting objects rather than actions.

11/28/2011 CIS 110 (11fa) - University of Pennsylvania 10

Review: what is an object?

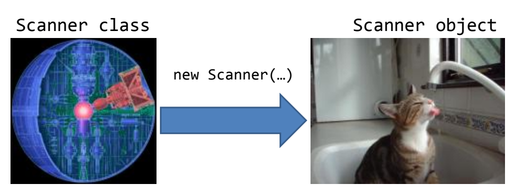
- An object is an entity with *state* and *behavior*.
  - State = values or internal data
  - Behavior = actions or methods
- Example: the Scanner object
  - State = position in text
  - Behavior = nextX(), hasNextX()



11/28/2011 CIS 110 (11fa) - University of Pennsylvania 11

Classes revisited

- Classes are *programs*, i.e., containers for methods.
- Classes are also *blueprints for objects*.

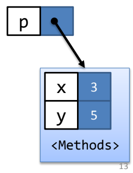


11/28/2011 CIS 110 (11fa) - University of Pennsylvania 12

Example: the Point class

- In package `java.awt`.
- Represents a coordinate pair in 2D-space.
  - State = (x, y) coordinates
  - Behavior = translate or shift coordinates

```
Point p = new Point(3, 5);
System.out.println("y-coordinate = " + p.y);
p.translate(1, 1);
System.out.println(p);
```



Step 1: declaring state

- State = (x, y) coordinates
  - Declared as instance variables or *fields*.

```
public class Point {
  public int x;
  public int y;
  // ... methods go here ...
}
```


Step 2: declaring behavior

- Behavior = translate or shift coordinates
  - Declared as *instance methods*.

```
public class Point {
  // ... fields goes here ...
  public void translate(int dx, int dy) {
    x += dx;
    y += dy;
  }
}
```


Step 3: declaring constructors

- Constructors allow us to make new Point objects from a class.
  - Constructors are *special methods* that are only invoked when new is used.

```
public class Point {
  // ... everything else goes here ...
  public Point(int initialX, int initialY) {
    x = initialX;
    y = initialY;
  }
}
```


Default constructors

- If we don't provide a constructor, Java inserts a default constructor automatically.

```
public Point() { }
```

- However, since Point has a constructor, the default constructor is not inserted!

```
Point p = new Point();  // fails to compile
```


Multiple constructors

- We can have multiple constructors to allow clients to create Points in different ways.

```
public class Point {
  // ... everything else goes here ...
  // Instantiate with, e.g., new Point(3, 5)
  public Point(int initialX, int initialY) {
    x = initialX;
    y = initialY;
  }

  // Instantiate with, e.g., new Point()
  public Point() {
    x = 0;
    y = 0;
  }
}
```

Revisited: accessing members of objects

- To access a member (field or instance method) of an object, we use *dot notation*.

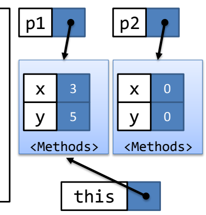
```
Point p1 = new Point(3, 5);
Point p2 = new Point(0, 0);
System.out.println("y-coordinate = " + p1.y);
p1.translate(1, 1);
```

- We access/modify p1's members rather than p2's.

The implicit this parameter

- In reality, when we reference members of an object inside a class, we go through the special `this` reference.

```
Point p1 = new Point(3, 5);
Point p2 = new Point(0, 0);
p1.translate(1, 1);
// ...
public class Point {
  // ... everything else goes here ...
  public void translate(int dx, int dy) {
    this.x += dx;
    this.y += dy;
  }
}
```



Static vs. non-static members

- Note that we don't have `static` anywhere!

```
public class Point {
  // ... fields goes here ...
  public static void translate(int dx, int dy) {
    x += dx;
    y += dy;
  }
}
```

- Error: "Cannot make a static reference to the non-static field x"

A tale of two worlds

- Non-static members = part of a particular object (i.e., instance of a class)
- Static members = part of the class itself
  - Have no `this` reference to play with!

```
public class Point {

  public static void main(String[] args) { }
  // Static stuff goes here ^^
  // THE STATIC WORLD AND THE INSTANCE WORLD
  // Non-static stuff goes here vv
  public int x;

}
```


Example: a Student class

```
public class Student {
  public String firstName;
  public String lastName;
  public String fullName;
  public Student(String firstName, String lastName, String fullName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.fullName = fullName;
  }
}
```

- See anything that can go wrong here?

Inconsistent state

```
Student s = new Student("Peter-Michael", "Osera",
                        "Peter-Michael Osera");
s.firstName = "Michael-Peter";
System.out.println(s.firstName + " " + s.lastName);
System.out.println(s.fullName);
```

- fullName can get out of sync pretty easily!
  - Seems like bad design: client shouldn't be able to set fullName differently from firstName and secondName.
  - Also, doesn't seem like fullName should be a field anyways…

Encapsulation

- *Hide away implementation details and only expose essential functionality*.
  1. I want to hide the fact that the names are fields that can be modified.
  2. I want to expose the names to the client.
- Encapsulation is a cornerstone of *abstraction*.

1. Private fields

```
public class Student {
  private String firstName;
  private String lastName;
  private String fullName;
  public Student(String firstName, String lastName, String fullName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.fullName = fullName;
  }
}
```

- Private fields aren't visible to code outside of the class.
  - e.g., `s.firstName` now gives an error, so we can't access anything!

2. getter methods

```
public class Student {
  // Rest of implementation here
  public String getFullName() {
    return fullName;
  }
}
```

- Getter methods are regular methods whose job is to "get" some value from the class.
  - e.g., a private field or some calculated value.

A side-benefit: implementation hiding

```
public class Student {
  // Rest of implementation here
  public String getFullName() {
    return firstName + lastName;
  }
}
```

- Observation: we don't need `fullName`!
  - Makes no difference to users since they couldn't access `fullName` anyways!
  - Users only care about what `getFullName` returns.

A properly encapsulated Student

```
public class Student {
  private String firstName;
  private String lastName;

  public Student(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public String getLastName() {
    return lastName;
  }

  public String getFullName() {
    return firstName +  " " + lastName;
  }
}
```


Another example: Student revisited

```
public class Student {
  private int age;
  public Student(int age) {
    this.age = age;
  }
  public int getAge() {
    return age;
  }
}
```

- See anything else that can go wrong?

More inconsistent state

```
Student s = new Student(-3175);
```

- Negative ages don't make any sense!
- How do we restrict this behavior?

Enforcing class invariants

```
public Student(int age) {
  if (age < 0) {
    throw new IllegalArgumentException();
  }
  this.age = age;
}
```

- If the user provides a bad age, throw an exception!
- `age >= 0` is now an invariant of our class.
  1. Ensure the user never gives us a bad age.
  2. Ensure that we never make age go bad.