Specification and implementation

- **specification**: an object’s features, as seen by its clients.
- **implementation**: the “internals” that make up the features.
Specification and implementation (cont.)

- Specifications isolate you from the details of the implementation.
  - “I don’t care how you do it, just get the job done” (to specifications).
  - How the features are actually implemented by the server, is of no concern to the client.
- Preserving the distinction between specification and implementation is absolutely essential.
Let’s start specifying

- First, enumerate the object’s responsibilities. Then determine its properties and commands.
- Java syntax does not allow us to separate a class specification from its implementation.
A simple counter

- A simple counter’s responsibilities:
  
  Know (one property):
  the value of the count
  
  Do (two commands):
  set the count to 0
  increment the count by 1
A simple counter (cont.)

- A simple counter

  **Class:** Counter

  **Properties:**
  - count (A non-negative integer)

  **Commands:**
  - reset (sets count value to 0)
  - step_count (increments the count by 1)
package counters;

/**
 * A simple integer counter.
 */
public class Counter {

    // Constructors:

    /**
     * Create a new Counter object.
     */
    public Counter () {
        ...
    }

    // Queries:

    /**
     * Current count; the number of items counted.
     */
    public int count () {
        ...
    }

    // Commands:

    /**
     * Increment the count by 1.
     */
    public void stepCount () {
        ...
    }

    /**
     * Reset the count to 0.
     */
    public void reset () {
        ...
    }

} // end of class Counter
Tools such as *javadoc* generate sets of HTML documents containing specifications extracted from program source files.
public class Counter
extends java.lang.Object

A simple integer counter.

### Constructor Summary

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>counter()</code></td>
<td>Create a new Counter object</td>
</tr>
</tbody>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td>Current count, the number of items counted.</td>
</tr>
<tr>
<td><code>reset()</code></td>
<td>Reset the count to 0</td>
</tr>
<tr>
<td><code>stepCount()</code></td>
<td>Increment the count by 1</td>
</tr>
</tbody>
</table>

### Constructor Detail

**Counter**

public **counter**()

Create a new Counter object.

### Method Detail

**count**

public int **count**()

Current count, the number of items counted.

**stepCount**

public void **stepCount**()

Increment the count by 1

**reset**

public void **reset**()

Reset the count to 0
Invoking a method

- **method**: a language construct that defines and implements a query or command.
- In order to invoke a method, you must have an *instance* of the class call the method.
- **Syntax**: `instance.method()`
- **Example**: `c.count()`
1. client invokes the method `count` of the object `c`.

2. object c performs actions as prescribed by the method definition.

3. the result of c executing the method is that the value of the property `count` is delivered to the client.

---

1. client invokes the method `reset` of the object `c`.

2. object c performs actions as prescribed by the method definition.

3. the result of c executing the method is that the value of the property `count` is changed.
Client

Counter

count

reset

count

the object is active

time
Maze game example

- An explorer (player) must navigate successfully through the rooms without being killed by denizen (monsters).
- There can be several rooms and denizen, but only one explorer.
- We need 3 classes: Explorer, Denizen, and Room.
Explorer responsibilities

Know:

- his name
- his location in the maze
- damage inflicted upon an opponent
- damage received from an opponent
- his stamina

Do:

- set or change name
- change location
- fight a maze Denizen
Invoking methods with parameters

- To be invoked properly, a method sometimes requires information from the client. Information is passed in as **parameters**.

- When a client invokes a method, the client must provide a value of the appropriate type for each parameter.

  **Syntax:** `instance.method(p0,p1,p2...)`

- The only way an object’s state can change is by invoking one of the object’s methods.
Parameters and arguments

- If we wish to change an explorer’s name, we must provide a new name.
- If we wish to change an explorer’s location, we must provide either a new location or a movement from which the location can be determined.
- The needed elements are referred to as the **parameters** of the command.
- The actual values we provide are referred to as **arguments**.
The diagram illustrates the interaction between a client and a server in a game environment. At time $t_1$, the client sends a command to the server to "take hit (strength 5)" to a character named "Explorer." The command is processed by the server, resulting in a state change: the character's strength decreases from 10 to 5, and stamina decreases from 100 to 95. At time $t_2$, the updated state of the Explorer is reflected on the client's screen.
 Constructors

- Can be used to set the initial values of an object's properties.

- Examples:
  - A counter's initial value of 0.
  - An explorer's initial name, location, strength, and stamina.
explorer
Class Explorer

    public class Explorer
    A maze game player.

Constructors:

    public Explorer (String name, rooms.Room location,
                   int hitStrength, int stamina)
    Create a new Explorer.

Queries:

    public String name ()
    Name of this Explorer.

    public Rooms.Room location ()
    Room in which this Explorer is currently located.

    public int strength ()
    Damage (hit points) this Explorer does when striking.

    public int stamina ()
    Damage (hit points) required to defeat this Explorer.

Commands:

    public void changeName (String newName)
    Change the name of this Explorer to the specified String.

    public void move (rooms.Room newRoom)
    Move to the specified Room.

    public void takeHit (int hitStrength)
    Receive a blow of the specified number of hit points.

    public void strike (denizens.Denizen monster)
    Strike the specified Denizen.
Class Denizen

public class Denizen
    A maze monster.

Constructors:

public Denizen (String kind, rooms.Room location)
    Create a new Denizen.

Queries:

public String kind ()
    Kind of Denizen.

public rooms.Room location ()
    Room in which this Denizen is currently located.

Commands:

public void move (rooms.Room newRoom)
    Move to the specified Room.

public void takeHit (int hitStrength)
    Receive a blow of the specified number of hit points.

public void strike (explorer.Explorer explorer)
    Strike the specified Explorer.
Consider a university registration system.

- Students register for courses.
  - Each student is represented by a distinct object.
  - All students are members of the Student class.

- The university
  - Assesses fees.
  - Prints student schedules.
  - Produces class rolls.
  - Etc.
Properties

public String name ()
public String address ()
public String ssn ()
public int creditHours ()
public int fees ()
public int feesPaid ()
public courses.CourseList schedule()
**Student**

- name = Arthur Brooke
- address = Tipton Grange
- ssn = 10
- creditHours = 95
- fees = $1200
- feesPaid = $600
- schedule =

**CourseList**

- 1st = CSCI 1583 Sec 1
- 2nd = MATH 0106 Sec 12
- ...

...
Commands

public void changeName ()
public void changeAddress ()
public void changeSsn ()
public void addCourse ()
public void dropCourse ()
public void payFees ()
Constructors

```java
public Student (String name, String address, String ssn)
```