Object Model

Overview

- Essential aspects of the Object Model:
  - Abstraction
  - Encapsulation
- Realization of the Object Model using:
  - Classes
  - Objects

What is an ABSRACTION?

- An abstraction is one of the fundamental ways in which we handle *complexity.*

  *An abstraction denotes the essential **properties** of an object,*
  which distinguish it from other objects, and
  *thereby establishes well defined conceptual **boundaries**,*
  *relative to an observer's **perspective**.*

- Main problem in object-oriented system development (OOSD):

  Choosing the right abstractions.

- Abstractions can be about *tangible* things (vehicle, map) and *conceptual* things (meeting, dates, different processes).

Example of an Abstraction

- Which essential details describe this "thingy"?
  - Abstraction name: `Light`
  - Light's wattage, i.e. energy usage.
  - Light can be on or off.

  Different lights (objects) will have the above properties defined.

  Other properties like shape, mat, color, socket size, etc. are less essential for this particular definition of the abstraction.

  Essential properties are dictated by the problem.

Modelling Abstraction using Classes

- A *class* defines
  - all attributes/properties/data, and
  - all methods/behaviors/operations

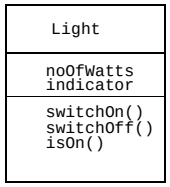
  of an abstraction.

Schematic specification of a class

Class `Light`

Attributes:	Methods:
`noOfWatts`	`switchOn`
`indicator`	`switchOff`
	`isOn`

Graphical Notation for a class



- UML: Unified Modelling Language (*http://www.rational.com/uml/*)

Classes

- A class denotes a category of *objects*.
  - defines the properties and behaviors of the objects
  - defines a "template" or a "blue print" for creating objects (also called *instances*)

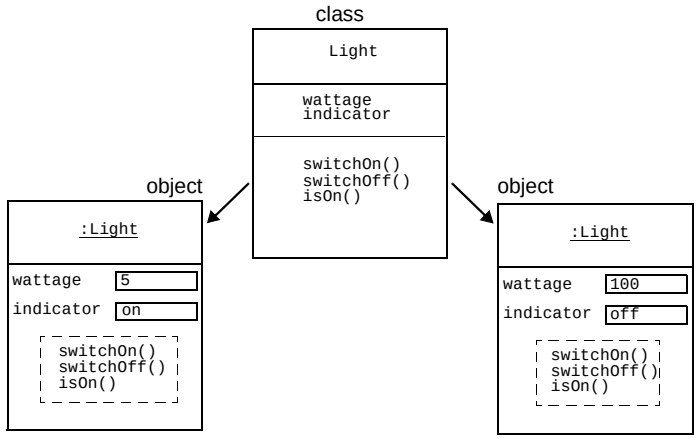
```
class Light {
    // Instance variables
    private int noOfWatts;          // wattage
    private boolean indicator;      // on or off

    // Instance methods
    public void switchOn()  { indicator = true; }
    public void switchOff() { indicator = false; }
    public boolean isOn()   { return indicator; }
}
```


Terminology

Term:	**Other synonyms:**
class	object-class
object	instance, object-instance, class-instance, occurrence
(instance-)methods	operations, behaviors, member functions, procedures, functions
(instance-)variables	attributes, properties, fields, data, data-member
(instance-)members	instance-variables *and* -methods

Objects as Instances of Classes



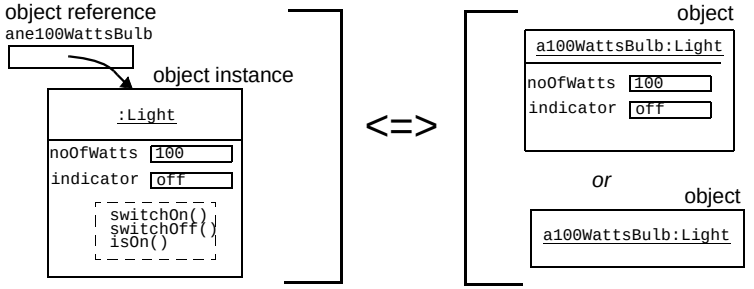
Objects

- *Objects* contain *instance variables* (which can be references to other references, leading to *aggregation*).
- Values of an object's instance variables at any given time constitute its *state*.
- Each object is unique (i.e. has a unique id) even if the objects have the same state.
- Behavior of an object is implemented by methods.
  - Objects of the same class share method implementations.
- In Java, the `new` operator is used to create objects.

```
Light one100WattBulb = new Light(); // declaration + instantiation
```

  - the `new` operator takes a *constructor call* as argument.
  - A constructor is primarily used to set the *initial state* of the object.
- In Java, objects can only be manipulated by *object references* (ex. `one100WattBulb`).
  - A reference *denotes* an object.

Object references

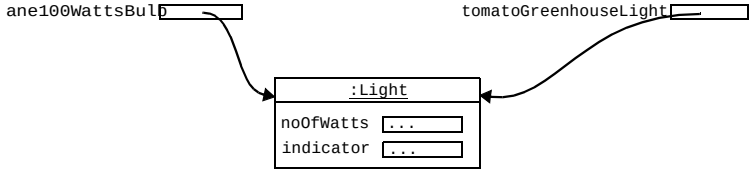


- When we *instantiate* a class to create an object,
  - we get a *reference* to the object, and
  - we must declare a (reference) variable to store the reference.
- A reference provides a handle for an object, and can be used to send *messages* to the object.
- *References are values which can be assigned, cast and passed as parameters.*

Aliases

- An object can have several references, called *aliases*.

```
Light one100WattBulb = new Light();
Light tomatoGreenhouseLight = one100WattBulb;  // results in aliases
```



- Any alias of an object can be used to send messages to the object.

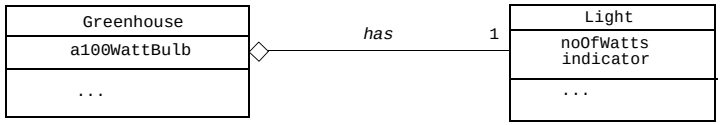
```
// ...
tomatoGreenhouseLight.switchOn();
if (one100WattBulb.isOn())
    System.out.println("Light in the tomato greenhouse is on.");
// ...
```


Aggregation

- In Java, objects cannot contain other objects, they can only have *references* to other objects.

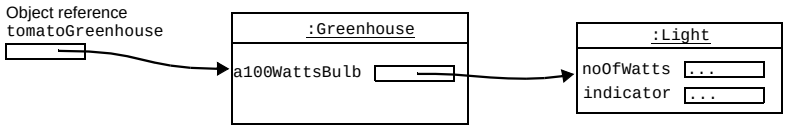
```
class Greenhouse {
    Light a100WattBulb;
    Greenhouse() {
        Light a100WattBulb = new Light();
        // ...
    }
    // ...
}
```

- *Class-diagram* can be used to depict aggregation (*static view*).



```
// Some client
Greenhouse tomatoGreenhouse = new Greenhouse();
```

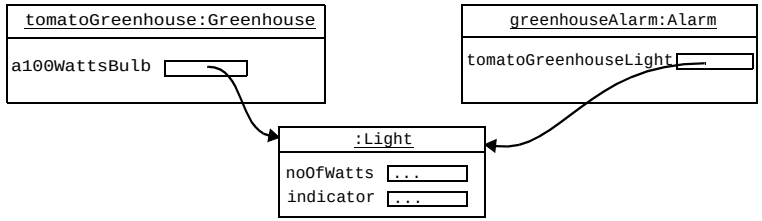
- *Dynamic view* at runtime:



  - Value of the instance variable `a100WattsBulb` is the reference to an object of class `Light`.

Sharing Constituent Objects

- Aggregate objects can share constituent objects by aliases.



Communication between Objects

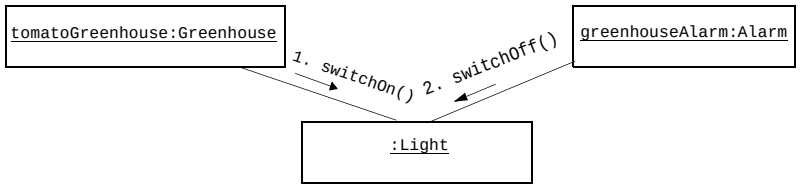
- Given a reference to an object, we can use the reference to invoke methods on the object to elicit particular behavior from the object.

```
// ...
Boolean onFlag = one100WattBulb.isOn(); // call returns a value
if (!onFlag)
    one100WattBulb.switchOn();          // call does not return a value
// ...
```

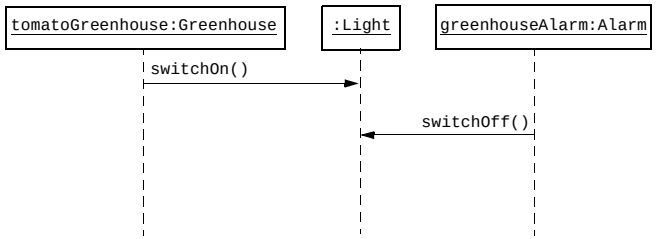
- Objects communicate by calling methods on each other, and return values when appropriate.
- *In OOSD, understanding the communication between objects is an essential part of designing the system.*

Interaction Diagrams

Collaboration Diagram



Sequence Diagram



Static Members

- There are cases where *members* should only belong to the class, and not be part of any object instantiated from the class.
- Clients can call *static methods* and access *static variables* by using the class name or object references of the class.

Terminology:

static methods	*class-methods* which only belong to the class
static variables	*class-variables* which only belong to the class
static members	static variables *and* methods

Example:

We wish to keep track of how many objects of class `Light` have been created.

- We need a "global" counter, but it should not be instantiated with any object of the class.
- We need a "global" method which can be called to find out how many instances have been created.

```
class Light {
    // Instance variables
    private int noOfWatts;          // wattage
    private boolean indicator;      // on or off
    private String location;        // placement
    // Static variable
    private static int counter;     // no. of Light objects created
    // Instance methods
    public void switchOn()  { indicator = true; }
    public void switchOff() { indicator = false; }
    public boolean isOn()   { return indicator; }
    // Static methods
    public static void writeCount() {
         System.out.println("Number of lights: " + counter);
    }
    //...
}
```


Inheritance

- Classes can be *extended.*
- A *subclass* inherits members (variables and methods) from its *superclass.*

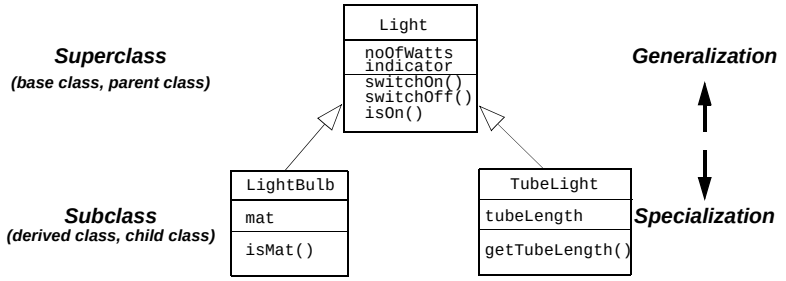
```
class LightBulb extends Light {
    private boolean mat;
    // ...
    public int isMat() { return mat; }
}

class TubeLight extends Light {
    private int tubeLength;
    // ...
    public int getTubeLength() { return tubeLength; }
}
```

This is called *implementation inheritance.*

Extensibility by inheritance

Inheritance Hierarchy



Encapsulation

- In OOSD, *abstraction* for an object comes before its *implementation.*
- *Abstraction* focuses on visible behavior of an object (called the *contract* or *interface*), whereas *encapsulation* focuses on *implementation* which give this behavior.
- Encapsulation leads to separation of contract and implementation.
  - Objects are regarded as "black boxes" whose internals are hidden.

A class has 2 *views*:

- *Contract* which corresponds to our abstraction of behaviors common to all instances of the class.
  - specification/documentation on how *clients* (i.e. other objects) can use instances of the class, in particular, how each method should be called.
  - WHAT the class offers.
- *Implementation* which corresponds to *representation* of the abstraction and the *mechanisms* which give provide the desired behaviors.
  - comprises of the instance variables and Java instructions which, when executed, give the desired behavior.
  - HOW the class implements its behaviors - *not a concern of the clients*.

This separation of concerns has major implications in system design.

Interfaces - *Programming by Contract*

- Java allows *contracts* to be defined by using *interfaces*.

```
interface ISwitch {
    void switchOn();
    void switchOff();
    boolean isOn();
}
```

- Classes can *implement* interfaces, thereby guaranteeing that they will honor the contract.

```
class Light implements ISwitch {
    // same as before
}
```

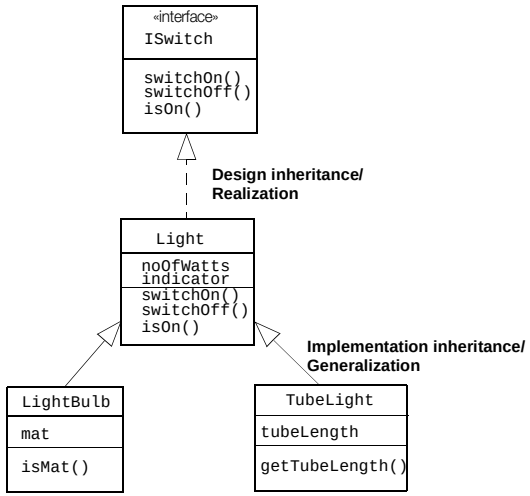
  This is also called *design inheritance*.

  Note that subclasses `LightBulb` *and* `TubeLight` *also implement the* `ISwitch` *interface.*

- *Programming by contract* uses interfaces, and not concrete implementations of them.

```
// ...
public void SwitchOnAllLights(ISwitch[] lights) {
    for (int i = 0; i < lights.length; i++)
        lights[i].switchOn();
}
// Elements of array lights can be objects of Light, TubeLight or LightBulb.
```


Inheritance Hierarchy



OOP (Object-oriented Programming)

OOP is an implementation methodology,

where a program is organized as a collection of co-operating ***objects****,*

where each object is ***an instance of a class****, and*

where all classes are members in a hierarchy of classes based on ***inheritance*** *relationships.*

- If the last criteria is not fulfilled, the implementation methodology is called *OBP* (*Object-based Programming*).
- One major consequence of OOP is that it promotes *code-reuse*, i.e. that classes can be "recycled".

Data + Methods = Objects

Programming Methodology: *Structured Programming*

- "Internals" of objects implemented using structured programming.
- Operations/actions in methods are described by using the following *control structures*:
  - sequence (block)
  - conditionals
  - repetition (loops)
- Structured programming leads to methods that are easier:
  - to understand,
  - to test and debug,
  - to modify and maintain.