Object-oriented Programming - I

Reference:
Chapter 4 & Chapter 6

Overview

• Object Model (OM)
• Values, Variables and Types
• Defining classes
• Instance and static members
• Method signatures
• Overloading methods
• The `this` reference
• Constructors: default and non-default
• Overloading constructors

• The inheritance relationship: is-a
• The aggregation relationship: has-a

• Overridden and overloaded methods
• The keyword `super`
• Variable shadowing

• Constructors and constructor chaining using `this()` and `super()`

Object Model

Important Concepts

• Essential aspects of the Object Model:
  – Abstraction
  – Encapsulation
• Realization of the Object Model using:
  – Classes
  – Objects
What is an ABSTRACTION?

- 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/clear, 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

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Attributes</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light</td>
<td>noOfWatts</td>
<td>switchOn()</td>
</tr>
<tr>
<td></td>
<td>indicator</td>
<td>switchOff()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>isOn()</td>
</tr>
</tbody>
</table>

Graphical Notation for a class

```
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; }
}
```

Class

- 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)

UML: Unified Modelling Language (http://www.rational.com/uml/)
# Terminology

**Term:**
- class
- object
- (instance-)methods
- (instance-)variables
- (instance-)members

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

---

# Objects as Instances of Classes

- **Light**
  - `noOfWatts`
  - `indicator`
  - `switchOn()`
  - `isOn()`
  - `switchOff()`

---

# Objects

- **Objects** contain **instance variables** (which can be references to other objects, 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 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.

```java
Light a100WattsBulb = 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** which are values that can be stored in variables (ex. `a100WattsBulb`).
  - A reference value (and thereby a reference variable which stores this value) **denotes** an object.

---

# Object references

- **object reference** `a100WattsBulb`

- **object instance**

  - `noOfWatts`
  - `indicator`
  - `switchOn()`
  - `isOn()`
  - `switchOff()`

- **object**

  - **object reference** `a100WattsBulb:Light`

- 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**.
- Assignment can result in aliases:
  ```java
  Light one100WattsBulb = new Light();
  Light tomatoGreenhouseLight = one100WattsBulb;
  ```
- Parameter-passing when a method is invoked can create aliases.
  ```java
  // Client
  tomatoGreenhouse.setLight(one100WattsBulb);
  // In class GreenHouse:
  void setLight(Light tomatoGreenhouseLight) { ... }
  ```

### Aggregation

- In Java, objects cannot contain other objects, they can only have **references** to other objects.
  ```java
class Greenhouse {
    Light a100WattsBulb;    
    Greenhouse(Light aLight) {
      a100WattsBulb = aLight;
    }
    // ...
  }
  ```
- Class-diagram can be used to depict aggregation (**static view**).

### Sharing Constituent Objects

- Aggregate objects can share constituent objects by aliases.
  ```java
class Alarm {
    Light tomatoGreenHouseLight;
    Alarm(Light aLight) {
      tomatoGreenHouseLight = aLight;
    }
    // ...
  }

  // Some client
  Alarm greenHouseAlarm = new Alarm(oneLight);
  ```
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.

```java
// ...
Boolean onFlag = one100WattsBulb.isOn(); // call returns a value
if (!onFlag)
one100WattsBulb.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

1. switchOn()
   - tomatoGreenhouse:Greenhouse
   - greenhouseAlarm:Alarm

2. switchOff()
   - tomatoGreenhouse:Greenhouse
   - greenhouseAlarm:Alarm

Collaboration Diagram

Sequence Diagram

Click "Crisis"

Click "Normal"

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.
- Static variables can be used to define "global" constants and variables used by clients.
- Static methods provide "global" utilities used by clients.

```java
class DotDotCom {
// Constants
    static final int MAX_JOLTS = 10;
    static final String logo = "Jump-start with Joe's Java Jolt";
// Variables
    private static Car deliveryCars[];
// Methods
    static Car getNextAvailableCar() { ... } // ...
    }
// Some client
    int jolts = DotDotCom.MAX_JOLTS;
    Car expressDispatch = DotDotCom.getNextAvailableCar();
```
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.

```java
class Light {
    // Static variable
    private static int counter;  // no. of Light objects created

    // Static methods
    public static void writeCount() {
        System.out.println("Number of lights: "+ counter);
    }  // ...
}

// Some client
Light.writeCount();  // Invocation thru class
Light myLight = new Light(); myLight.writeCount();  // Invocation thru reference
```

Values, Variables and Types

Values in Java

- There are three kinds of "values":
  - primitive data values
  - reference values
  - objects
- A reference value denotes an object.
- Only primitive data values and reference values can be stored in variables (names).
- A reference variable (sometimes just called reference) can store a reference value.

Types in Java

- Both variables and objects have types.
- The type of a variable is determined when it is declared.

```java
int i;               // i is a variable of (primitive) type int
Light spotlight;    // spotlight is a variable of type Light
TubeLight ceilingLight; // ceilinglight is a variable of type TubeLight
ISwitch toggle;     // toggle is a variable of type ISwitch
```
The type of an object is determined when it is created.

```java
new Light();                  // Creates an object of class Light and
// returns a reference value which denotes
// this object.
new TubeLight();              // creates an object of class TubeLight and
// returns a reference value which denotes
// this object.
```

Reference values can be stored in variables.

```java
Light spotlight = new Light();
TubeLight ceilingLight = new TubeLight();
```

- The type of a variable and the type of an object cannot be changed.
- Reference values (like primitive values) can be assigned, cast and passed as parameters.

implementing om: classes and objects

Defining Classes

- A class definition specifies a new type and its implementation.
  
  `<class header> { <class body> }`
  
  - In the class header, the name of the class is preceded by the keyword `class`.
  - In addition, the class header can specify the following information:
    - Scope or accessibility modifier.
    - Additional class modifiers.
    - Any class it extends.
    - Any interfaces it implements.
  - The class body can contain declarations of the following members:
    - Instance variables and methods (`instance members`)
    - Static variables and methods (`static members`)
    - Constructors
    - Nested classes (`inner classes`)
    - Static and instance initializers
Defining Methods

- The behavior of objects is specified by the instance methods of the class.
- The general syntax of a method definition is:
  
  ```java
  <method header> (<formal parameter list>) <throws clause> {<method body>}
  ```
- Like member variables, member methods can be characterized as:
  - **Instance methods** belonging to the object of the class.
  - **Static methods** belonging only to the class.
- The **method header** must specify:
  - the **name** of the method
  - the **type** of the return value
  and, in addition, can specify
  - Scope or accessibility modifier.
  - Additional method modifiers.

Statements

- A statement in Java is terminated by a semicolon (;).
- Variable declarations with explicit initialization of the variables are called **declaration statements**.
- An **expression statement** is an expression terminated by a semicolon:
  - Assignments
  - Increment and decrement operators
  - Method calls
  - Object creation with the `new` operator
- **Flow control statements** include the following: `if-else`, `for`, `while`, `do-while`, `switch`, `break`, `continue`, `return`, `throw`, `try-catch`, `finally`
- Statements can also be labelled. `label: <statement>`
- A solitary semicolon (`;`) denotes the **empty statement** that does nothing.
- A block, `{}`, is a **compound** statement which can be used to group zero or more local declarations and statements.
- It can be used in any context where a simple statement is permitted.

Instance Methods and Object Reference

- The **this** reference is considered a short-hand notation for `this.member`.
- Each method needs to pass the object on which it is being invoked to another method, it can do so using the `this` reference.
- Note that no implicit `this` reference is passed to static methods, as these are not invoked on behalf of any object.
class Light {
    // Instance variables
    private int noOfWatts;     // wattage
    private boolean indicator; // on or off
    // Instance methods
    public void switchOn()  {
        this.indicator = true;  }
    public void switchOff() {
        this.indicator = false; }
    public boolean isOn()   { return this.indicator; }
    // Static variable
    private static int counter; // no. of Light objects created
    // Static methods
    public static void writeCount() {
        // Not OK
        System.out.println("Number of lights: " + this.counter); // Not OK
    } // ...
}

Local Variables and Instance Variables

- A local variable can shadow a member variable that has the same name (also called hiding).
- The reference this can be used to distinguish the instance variables from the local variables inside the method.

Example 1 Using this Reference

class Light {
    // Instance variables
    int noOfWatts;     // wattage
    boolean indicator; // on or off
    String location;  // placement
    // Non-default Constructor
    public Light(int noOfWatts, boolean indicator, String site) {
        String location;
        this.noOfWatts = noOfWatts;   // (1) Assignment to instance variable.
        indicator = indicator;        // (2) Assignment to parameter.
        location = site;              // (3) Assignment to local variable.
        this.someAuxilliaryMethod();  // (4)
        someAuxilliaryMethod();       // equivalent to call at (4)
    }
    void someAuxilliaryMethod() { System.out.println(this); } // (5)
}

Method Overloading

- Each method has a signature, which is comprised of the name of the method and the types and order of the parameters in the parameter list.
- Method overloading allows a method with the same name but different parameters, thus with different signatures, to have different implementations and return values of different types.
- Rather than invent new method names all the time, method overloading can be used when the same operation has different implementations.
- The JDK APIs make heavy use of method overloading.

```java
public static double min(double a, double b)
public static float min(float a, float b)
public static int min(int a, int b)
public static long min(long a, long b)
```

- Parameter list in the method header must differ either:
  - in the no. of parameters, or
  - the type of at least one parameter must be different.
- At compile time, the right implementation is chosen based on the signature of the method call.
In the examples below, five implementations of the method methodA are shown:

- public void methodA(int a, double b) {/* ... */}            // (1)
- public int methodA(int a) { return a; }                   // (2)
- public int methodA() { return 1; }                       // (3)
- public long methodA(double a, int b) { return b; }        // (4)
- public long methodA(int x, double y) { return x; }        // (5) Not OK.

The corresponding signatures of the methods are as follows:

- methodA(int, double)                             // (1')
- methodA(int)                                     // (2') Number of parameters.
- methodA()                                        // (3') Number of parameters.
- methodA(double, int)                             // (4') Order of parameters.
- methodA(int, double)                             // (5') Same as (1').

Changing just the return type (as shown at (3) and (4) below), or the exceptions thrown, in the implementation is not enough to overload a method, and will be flagged as a compile time error.

- The parameter list in the definitions must be different.
- void bake(Cake k) { /* ... */ }                 // (1)
- void bake(Pizza p) { /* ... */ }                 // (2)
- int halfIt(int a) { return a/2; }                // (3)
- double halfIt(int a) { return a/2.0; }          // (4) Not OK. Same signature.

Constructors

- The main purpose of constructors is to set the initial state of an object when the object is created using the new operator.
- A constructor has the following general syntax:
  <constructor header> <parameter list> { 
  <constructor body> 
  }
- Constructors are like member methods, but the constructor header may only contain the following information:
  - Scope or accessibility modifier. Accessibility modifiers for methods also apply to constructors.
  - Constructor name, which must be the same as the class name.
- The following restrictions should be noted:
  - Modifiers other than accessibility modifiers are not permitted.
  - Constructors cannot return a value.
  - Constructors can only be called using the new operator.

- The following implicit default constructor is generated and employed when a Light object is created at (1):
  Light() { }
  The instance variables of the object are initialized to their default values.
- A class can choose to provide an implementation of the default constructor.
  class Light {
  // ...
  // Explicit Default Constructor
  Light() {                        // (1)
    noOfWatts = 50;
    indicator = true;
    location = new String("X");
  }
  //...
  }
  class Greenhouse {
  // ...
  Light extralight = new Light();  // (2) Call of explicit default constructor.
  }

What is different about the state of the objects created in the two examples above?
If a class defines one or more constructors, it cannot rely on the implicit default constructor being generated.

If the class then requires a default constructor, its implementation must be provided.

```java
class Light {
    // ...
    // Only non-default Constructor
    Light(int watts, boolean state, String place) {        // (1)
        noOfWatts = watts;
        indicator = state;
        location = place;
    }
    // ...
}
class Greenhouse {
    // ...
    Light moreLight = new Light(100,true,"Greenhouse");      // (2) OK.
    // Light firstLight = new Light();                          // (3) Error.
}
```

Overloaded Constructors

Like methods, constructors can also be overloaded.

Overloading of constructors allows appropriate initialization of objects on creation, depending on the constructor invoked.

```java
class Light {
    // ...
    // Explicit Default Constructor
    Light() {                                                // (1)
        noOfWatts = 50;
        indicator = true;
        location = new String("X");
    }
    // Non-default Constructor
    Light(int watts, boolean ind, String loc) {              // (2)
        noOfWatts = watts;
        indicator = ind;
        location = loc;
    }
    // ...
}
class Greenhouse {
    // ...
    Light moreLight = new Light(100,true,"Greenhouse");      // (3) OK.
    Light firstLight = new Light();                          // (4) OK.
}
```

Inheritance

Extensibility by Linear Implementation Inheritance

- One fundamental mechanism for code reuse.
- The new class inherits all the members of the old class - not to be confused with accessibility of superclass members.
- A class in Java can only extend one other class, i.e. it can only have one immediate superclass.
- The superclass is specified using the extends clause in the header of the subclass.
- The definition of the subclass only specifies the additional new and modified members in its class definition.
- All classes extend the java.lang.Object class.
Example 2  Extending Classes

class Light {  // (1)
   // 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); }
}

class LightBulb extends Light {  // (2)
   private boolean mat;
   // ...
   public int isMat() { return mat; }
}

class TubeLight extends Light {  // (2)
   // Instance variables
   private int tubeLength;
   private int color;
   // Instance method
   public int getTubeLength() { return tubeLength; }
   // ...
}

Implementation Inheritance Hierarchy

- Inheritance defines the relationship is-a (also called superclass–subclass relationship) between a superclass and its subclasses.
- Classes higher up in the hierarchy are more generalized, as they abstract the class behavior.
- Classes lower down in the hierarchy are more specialized, as they customize the inherited behavior by additional properties and behavior.
- The Object class is always the root of any inheritance hierarchy.

Figure 1 Inheritance Hierarchy
Implications of Inheritance

- An object of a subclass can be used wherever an object of the superclass can be used.
- An object of the TubeLight class can be used wherever an object of the superclass Light can be used.
- An object of the TubeLight class is-a object of the superclass Light.
- The inheritance relationship is transitive: if class B extends class A, then a class C, which extends class B, will also inherit from class A via class B.
- An object of the SpotLightBulb class is-a object of the class Light.
- The is-a relationship does not hold between peer classes: an object of the LightBulb class is not an object of the class TubeLight, and vice versa.

Litmus test for using inheritance: if B is an A, then only let B inherit from A.

A superclass reference can denote objects of its subclasses.

Objects and Inheritance I

TubeLight deskLight = new TubeLight();
deskLight.getTubeLength();
deskLight.switchOn();
deskLight.equals(someOtherLight);

- The object denoted by deskLight behaves:
  - as a TubeLight
  - as a Light
  - as an Object

Visibility bar for reference deskLight

Upcasting

TubeLight deskLight = new TubeLight();
deskLight.getTubeLength();
deskLight.switchOn();
deskLight.equals(someOtherLight);

Light lightRef = deskLight; // Upcasting
lightRef.getTubeLength(); // Compile error
lightRef.switchOn(); // Compile error
lightRef.equals(someOtherLight);

Object objRef = deskLight; // Upcasting
objRef.getTubeLength(); // Compile error
objRef.switchOn(); // Compile error
objRef.equals(someOtherLight);
**Inheritance and Program Design**

- **Implementation Inheritance**: superclass specifies behavior, subclasses inherit behavior.
- **Extensibility**: superclass too general behavior, so subclasses adds their own behavior.
- **Overriding**: superclass specifies behavior, subclasses override behavior.
- **Final classes**: superclass can specify behavior, subclasses have to live with it.
- **Abstract Classes**: superclass specifies abstract behavior (contract), subclasses must provide the implementation.

**Aggregation**

- A major mechanism for code reuse mechanism is aggregation.
- Aggregation defines the relationship *has-a* (a.k.a. whole–part relationship) between an instance of a class and its constituents (a.k.a. parts).
- In Java, an aggregate object cannot contain other objects.
  - It can only have references to its constituent objects.
  - The *has-a* relationship defines an aggregation hierarchy.
- In this simple form of aggregation, constituent objects can be shared between objects, and their lifetimes are independent of the lifetime of the aggregate object.

```java
class GreenHouse {
    TubeLight[] dayLights;
    LightBulb alarmBulb;
    Plant[] produce;
    ... 
}
```

**Method Overriding**

- A subclass may *override* non-static methods (non-private and non-final) inherited from the superclass.
- When the method is invoked on an object of the subclass, it is the new method definition in the subclass that is executed.
- The new method definition in the subclass must have the same *method signature* (i.e. method name and parameters) and the same *return type*.
  - The new method definition, in addition, cannot “narrow” the accessibility of the method, but it can “widen” it.
  - The new method definition in the subclass can only specify all or a subset of the exception classes (including their subclasses) specified in the *throws* clause of the overridden method in the superclass.
- A subclass can also use the keyword `super` to invoke the overridden method in the superclass.
- Any *final*, *static* and *private* methods in a class cannot be overridden but a subclass can redefine such methods -- not be a good idea.
Method Invocation and Variable Access

- When a method is invoked on an object using a reference, it is the class of the current object denoted by the reference, not the type of the reference, that determines which method implementation will be executed.
- When a variable of an object is accessed using a reference, it is the type of the reference, not the class of the current object denoted by the reference, that determines which variable will actually be accessed.

Example 3  Overriding and Overloading Methods and Shadowing Variables

```java
// Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

class Light {
    protected String billType = "Small bill";               // (1)
    protected double getBill(int noOfHours)                // (2)
            throws InvalidHoursException {
        double smallAmount = 10.0,
                        smallBill = smallAmount * noOfHours;
        System.out.println(billType + " : " + smallBill);
        return smallBill;
    }
}

class TubeLight extends Light {
    public String billType = "Large bill";                  // (3) Shadowing.
    public double getBill(final int noOfHours)              // (4) Overriding.
            throws ZeroHoursException {
        double largeAmount = 100.0,
                        largeBill = largeAmount * noOfHours;
        System.out.println(billType + " : " + largeBill);
        return largeBill;
    }
}

class Client {
    public static void main(String args[])                  // (6)
            throws InvalidHoursException {
        TubeLight tubeLightRef = new TubeLight();           // (7)
        Light lightRef1 = tubeLightRef;                      // (8)        Light lightRef2 = new Light();                      // (9)
        // Invoke overridden methods
        tubeLightRef.getBill(5);                              // (10)
        lightRef1.getBill(5);                                // (11)
        lightRef2.getBill(5);                                // (12)
        // Access shadowed variables
        System.out.println(tubeLightRef.billType);            // (13)
        System.out.println(lightRef1.billType);              // (14)
        System.out.println(lightRef2.billType);              // (15)
        // Invoke overloaded method
        tubeLightRef.getBill();                               // (16)
    }
}
```

Output from the program:
Large bill: 500.0
Large bill: 500.0
Small bill: 50.0
Large bill
Small bill
Small bill
No bill
Example: Method Overriding and Overloading

The object type determines the method executed.

Overriding:
- Large bill: 500.0
- Large bill: 500.0

Overloading:
- No bill

Overriding vs. Overloading
- Method overriding requires the same method signature (name and parameters) and the same return type, and that the original method is inherited from its superclass.
- Overloading requires different method signatures, but the method name should be the same.
- To overload methods, the parameters must differ in type or number.
- The return type is not a part of the signature, changing it is not enough to overload methods.
- A method can be overloaded in the class it is defined in, or in a subclass of its class.
- Invoking an overridden method in the superclass from a subclass requires special syntax (for example, the keyword `super`).

Variable Shadowing
- A subclass cannot override variable members of the superclass, but it can shadow them.
- A subclass method can use the keyword `super` to access inherited members, including shadowed variables.

Example: Variable Shadowing

The reference type determines the variable accessed.

Output:
- Large bill
- Small bill
Object Reference super

- The this reference is passed as an implicit parameter when an instance method is invoked.
- It denotes the object on which the method is called.
- The keyword super can be used in the body of an instance method in a subclass to access variables and invoke methods inherited from the superclass.
- The keyword super provides a reference to the current object as an instance of its superclass.
- The super.super.X construct is invalid.

Example 4 Using super Keyword

```java
// Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}
class Light {
    protected String billType = "Small bill";               // (1)
    protected double getBill(int noOfHours)                // (2)
        throws InvalidHoursException {
        double smallAmount = 10.0,
            smallBill = smallAmount * noOfHours;        System.out.println(billType + " \": " + smallBill);
        return smallBill;
    }
    public void banner() {                                  // (3)
        System.out.println("Let there be light!");    }
}

class TubeLight extends Light {                        // (4) Shadowing.
    public String billType = "Large bill";                  // (5) Overriding.
    public double getBill(int noOfHours)                  // (6)
        throws ZeroHoursException {
        double largeAmount = 100.0,
            largeBill = largeAmount * noOfHours;
        System.out.println(billType + " \": " + largeBill);
        return largeBill;
    }
    public double getBill() {                               // (7)
        System.out.println("No bill"); return 0.0;    }
}
class NeonLight extends TubeLight {                        // (8)
    // ...
    public void demonstrate()            throws InvalidHoursException {                  // (9)
        super.banner();                                     // (10)
        super.getBill(20);                                  // (11)
        super.getBill();                                    // (12)
        System.out.println(super.billType);                 // (13)
        ((Light) this).getBill(20);                         // (14)
        System.out.println(((Light) this).billType);        // (15)
    }
}

class Client {    public static void main(String args[]) throws InvalidHoursException {
    NeonLight neonRef = new NeonLight();
    neonRef.demonstrate();
    }
}

Output from the program:
Let there be light!
Large bill: 2000.0
No bill
Large bill
Large bill: 2000.0
Small bill
```
Example: super

```java
NeonLight neonRef
// neonRef.demonstrate();
...
class NeonLight extends TubeLight {
    public void demonstrate() throws InvalidHoursException {
        super.banner();
        super.getBill(20);
        super.getBill();
        System.out.println(super.billType);
        System.out.println(((Light) this).getBill(20));
        ((Light) this).getBill();
        System.out.println(((Light) this).billType);
    }
}
Program output:
Let there be light!
Large bill: 2000.0
No bill
Large bill
Large bill: 2000.0
Small bill
```
**this() Constructor Call**

- The `this()` construct can be regarded as being "locally overloaded".
- The `this()` call invokes the constructor with the corresponding parameter list.
- **Local chaining** of constructors in the class when an instance of the class is created.
- Java specifies that when using the `this()` call, it must occur as the first statement in a constructor, and it can only be used in a constructor definition.
- Note the order in which the constructors are invoked in the example.

**Example 5  this() Constructor Call**

class Light {
    // Instance Variables
    private int noOfWatts;
    private boolean indicator;
    private String location;
    // Constructors
    Light() {                              // (1) Explicit default constructor
        this(0, false);
        System.out.println("Returning from default constructor no. 1.");
    }
    Light(int watt, boolean ind) {         // (2) Non-default
        this(watt, ind, "X");
        System.out.println("Returning from non-default constructor no. 2.");
    }
    Light(int noOfWatts, boolean indicator, String location) {  // (3) Non-default
        this.noOfWatts = noOfWatts;
        this.indicator = indicator;
        this.location = new String(location);
        System.out.println("Returning from non-default constructor no. 3.");
    }
    }

`public class DemoThisCall {
    public static void main(String args[]) {                    // (4)
        System.out.println("Creating Light object no.1.");    // (5)
        Light light1 = new Light();                             // (6)
        System.out.println("Creating Light object no.2.");    // (6)
        Light light2 = new Light(250, true);                    // (6)
        System.out.println("Creating Light object no.3.");    // (6)
        Light light3 = new Light(250, true, "attic");           // (6)
    }
    }

Output from the program:
Creating Light object no.1.
Returning from non-default constructor no. 3.
Returning from non-default constructor no. 2.
Returning from default constructor no. 1.
Creating Light object no.2.
Returning from non-default constructor no. 3.
Returning from non-default constructor no. 2.
Creating Light object no.3.
Returning from non-default constructor no. 3.

**super() Constructor Call**

- The `super()` construct is used in a subclass constructor to invoke constructors in the immediate superclass.
- This allows the subclass to influence the initialization of its inherited state when an object of the subclass is created.
- A `super()` call in the constructor of a subclass will result in the execution of the relevant constructor from the superclass, based on the arguments passed.
- The `super()` call must occur as the first statement in a constructor, and it can only be used in a constructor definition.
- This implies that `this()` and `super()` calls cannot both occur in the same constructor.
Example 6  super() Constructor Call

class Light {
    // Instance Variables
    private int noOfWatts;
    private boolean indicator;
    private String location;

    // Constructors
    Light() {                              // (1) Explicit default constructor
        this(0, false);
        System.out.println("Returning from default constructor no. 1 in class Light");
    } 
    Light(int watt, boolean ind) {                              // (2) Non-default
        this(watt, ind, "X");
        System.out.println("Returning from non-default constructor no. 2 in class Light");
    }
    Light(int noOfWatts, boolean indicator, String location) {  // (3) Non-default
        super();                                                // (4)
        this.noOfWatts = noOfWatts;
        this.indicator = indicator;        this.location = new String(location);
        System.out.println("Returning from non-default constructor no. 3 in class Light");
    }
}

class TubeLight extends Light {
    // Instance variables
    private int tubeLength;
    private int colorNo;

    TubeLight(int tubeLength, int colorNo) {                    // (5) Non-default
        this(tubeLength, colorNo, 100, true, "Unknown");
        System.out.println("Returning from non-default constructor no. 1 in class TubeLight");
    }
    TubeLight(int tubeLength, int colorNo, int noOfWatts,
              boolean indicator, String location) {             // (6) Non-default
        super(noOfWatts, indicator, location);                  // (7)
        this.tubeLength = tubeLength;
        this.colorNo = colorNo;        System.out.println("Returning from non-default constructor no. 2 in class TubeLight");
    }
}

public class Chaining {
    public static void main(String args[]) {
        System.out.println("Creating a TubeLight object.");
        TubeLight tubeLightRef = new TubeLight(20, 5);          // (8)
    }
}

Output from the program:

Creating a TubeLight object.
Returning from non-default constructor no. 3 in class Light
Returning from non-default constructor no. 2 in class TubeLight
Returning from non-default constructor no. 1 in class TubeLight

(subclass–superclass) Constructor Chaining

- The this() construct is used to "chain" constructors in the same class, and the constructor at the end of such a chain can invoke a superclass constructor using the super() construct.
- The super() construct leads to chaining of subclass constructors to superclass constructors.
- This chaining behavior guarantees that all superclass constructors are called, starting with the constructor of the class being instantiated, all the way up to the root of the inheritance hierarchy, which is always the Object class.
- Note that the body of the constructors is executed in the reverse order to the call order, as super() can only occur as the first statement in a constructor.
**Default super() Call**

- If a constructor does not have either a this() or a super() call as its first statement, then a super() call to the default constructor in the superclass is inserted. The code

```java
class A {
    public A() {}
    // ...
}
class B extends A {
    // no constructors
    // ...
}
```

is equivalent to

```java
class A {
    public A() { super(); } // (1)
    // ...
}
class B extends A {
    public B() { super(); } // (2)
    // ...
}
```

**Summary of super() Call Usage**

<table>
<thead>
<tr>
<th>Superclass</th>
<th>No constructors or only the default constructor</th>
<th>Only non-default constructors</th>
<th>Both default and non-default constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subclass</td>
<td>Implicit default super() call or explicit default super() call</td>
<td>Explicit non-default super() call</td>
<td>Implicit default super() call or explicit default super() call or explicit non-default super() call</td>
</tr>
</tbody>
</table>

- If a class only defines non-default constructors (i.e. only constructors with parameters), then its subclasses cannot rely on the implicit behavior of a super() call being inserted.
- This will be flagged as a compile time error.
- The subclasses must then explicitly call a superclass constructor, using the super() construct with the right arguments.

```java
class NeonLight extends Tubelight {
    // Instance Variable
    String sign;

    NeonLight() { // (1)
        super(10, 2, 100, true, "Roof-top"); // (2) Cannot be commented out.
        sign = "All will be revealed!";
    }
    // ...
}
```

Subclasses without any declared constructors will fail to compile if the superclass does not have a default constructor and provides only non-default constructors.