IT602: Object-Oriented Programming

Lecture - 12

Inheritance

Arpit Rana

24th March 2022

Inheritance: What & Why

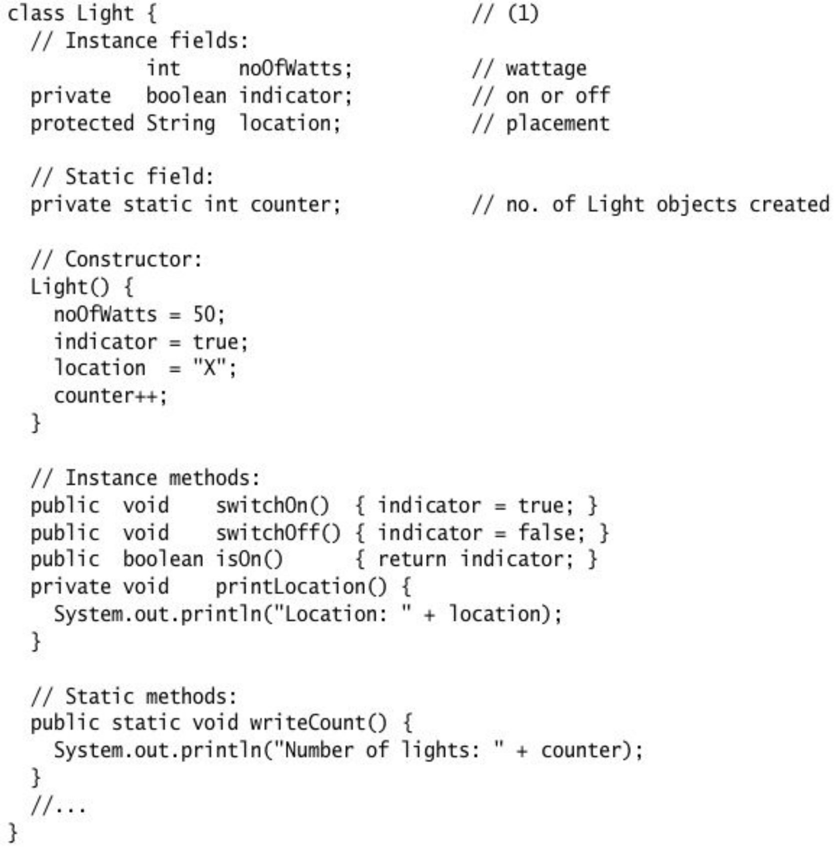
Inheritance is one of the fundamental mechanisms for ***code reuse*** in OOP.

- It allows new classes to be derived from an existing class.
- The new class (also called ***subclass, subtype, derived class, child class***) can inherit members from the old class (also called ***superclass, supertype, base class, parent class***).
- The ***subclass*** can add new behavior and properties (i.e., adding new fields and methods) and, modify its inherited behavior.

Inheritance: Facts

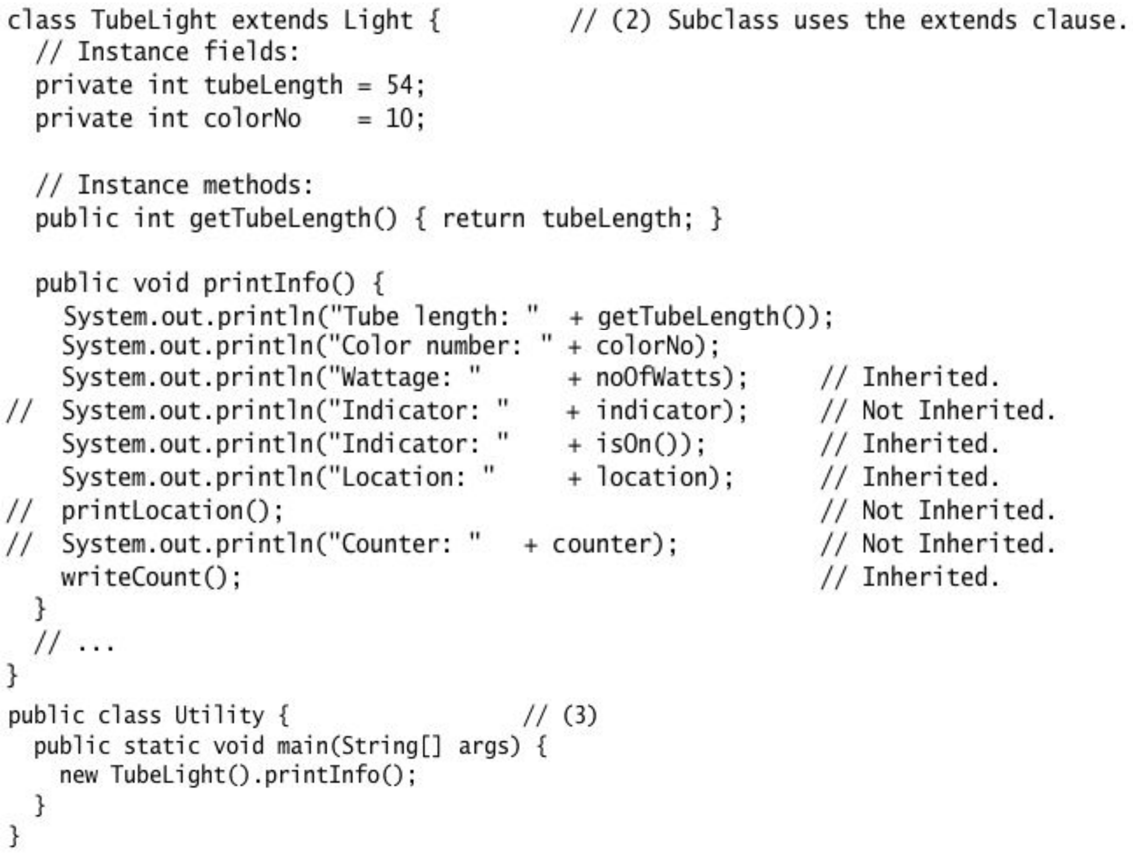
- The superclass is specified using the `extends` clause in the header of the subclass declaration.
- If no `extends` clause is specified in the header of a class declaration, the class implicitly inherits from the `java.lang.Object` class.
- `private` members of the superclass are not inherited by the subclass.
- Since constructors and initializer blocks are not members of a class, they are not inherited by a subclass.

Inheritance: Example



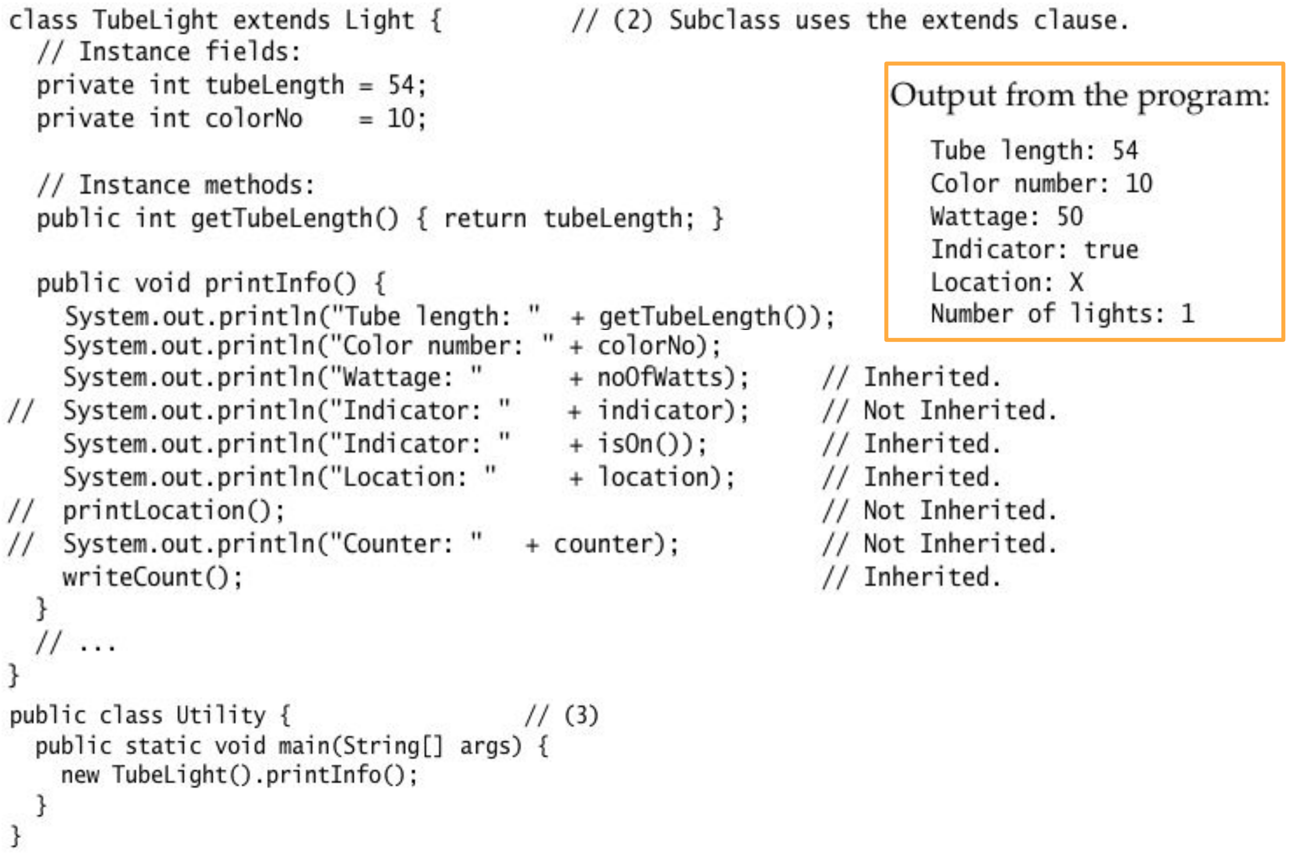
```
class Light {                                  // (1)
  // Instance fields:
            int     noOfWatts;                 // wattage
  private   boolean indicator;                 // on or off
  protected String  location;                  // placement

  // Static field:
  private static int counter;                  // no. of Light objects created

  // Constructor:
  Light() {
    noOfWatts = 50;
    indicator = true;
    location  = "X";
    counter++;
  }

  // Instance methods:
  public  void    switchOn()  { indicator = true; }
  public  void    switchOff() { indicator = false; }
  public  boolean isOn()      { return indicator; }
  private void    printLocation() {
    System.out.println("Location: " + location);
  }

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

Inheritance: Example



```
class TubeLight extends Light {        // (2) Subclass uses the extends clause.
  // Instance fields:
  private int tubeLength = 54;
  private int colorNo    = 10;

  // Instance methods:
  public int getTubeLength() { return tubeLength; }

  public void printInfo() {
    System.out.println("Tube length: "  + getTubeLength());
    System.out.println("Color number: " + colorNo);
    System.out.println("Wattage: "      + noOfWatts);     // Inherited.
//  System.out.println("Indicator: "    + indicator);     // Not Inherited.
    System.out.println("Indicator: "    + isOn());        // Inherited.
    System.out.println("Location: "     + location);      // Inherited.
//  printLocation();                                      // Not Inherited.
//  System.out.println("Counter: "   + counter);          // Not Inherited.
    writeCount();                                         // Inherited.
  }
  // ...
}
public class Utility {                 // (3)
  public static void main(String[] args) {
    new TubeLight().printInfo();
  }
}
```

Inheritance: Example



```
class TubeLight extends Light {        // (2) Subclass uses the extends clause.
  // Instance fields:
  private int tubeLength = 54;
  private int colorNo    = 10;

  // Instance methods:
  public int getTubeLength() { return tubeLength; }

  public void printInfo() {
    System.out.println("Tube length: "  + getTubeLength());
    System.out.println("Color number: " + colorNo);
    System.out.println("Wattage: "      + noOfWatts);     // Inherited.
//  System.out.println("Indicator: "    + indicator);     // Not Inherited.
    System.out.println("Indicator: "    + isOn());        // Inherited.
    System.out.println("Location: "     + location);      // Inherited.
//  printLocation();                                      // Not Inherited.
//  System.out.println("Counter: "    + counter);         // Not Inherited.
    writeCount();                                         // Inherited.
  }
  // ...
}

public class Utility {                  // (3)
  public static void main(String[] args) {
    new TubeLight().printInfo();
  }
}
```

Output from the program:

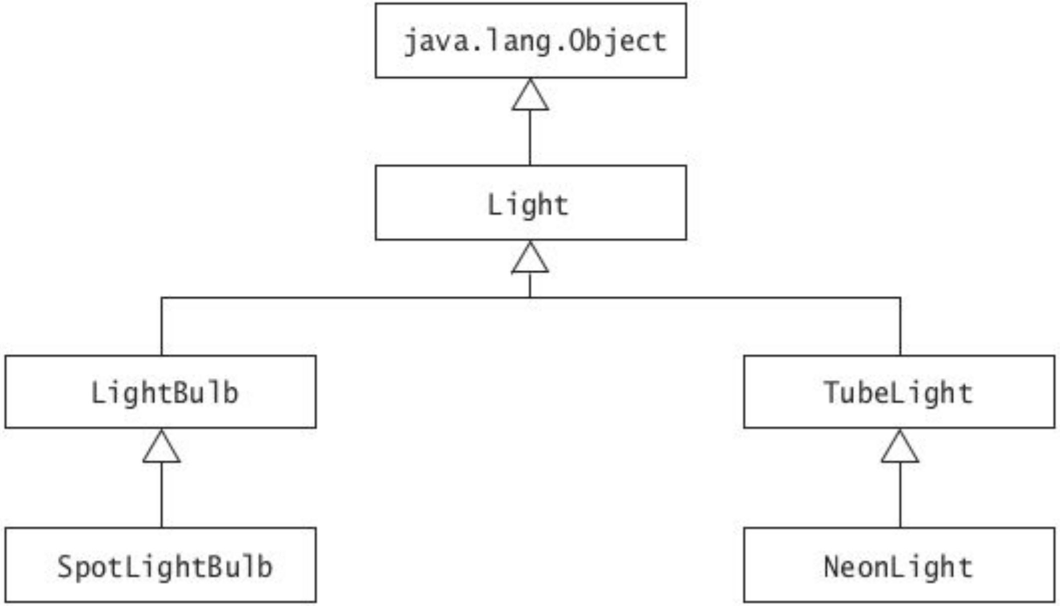
```
Tube length: 54
Color number: 10
Wattage: 50
Indicator: true
Location: X
Number of lights: 1
```

Inheritance Hierarchy

In Java, ***a class can only extend one other class***; i.e., it can only have one immediate superclass, a.k.a ***single*** or ***linear inheritance***.

- The inheritance relationship can be depicted as an inheritance hierarchy (a.k.a. class hierarchy).
- 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.

Inheritance Hierarchy



Relationships: ***is–a***

Inheritance defines the relationship ***is-a*** (also called the superclass–subclass relationship) between a superclass and its subclasses.

This means that an object of a subclass ***is-a*** superclass object, and can be used wherever an object of the superclass can be used.

- 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-an*** 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.

Relationships: ***supertype – subtype***

A class defines a reference type. Therefore, the inheritance hierarchy can be regarded as a ***type hierarchy***.

- Type hierarchy embodies the ***supertype-subtype*** relationship between reference types.
- The ***supertype-subtype*** relationship implies that the reference value of a subtype object can be assigned to a supertype reference, because a subtype object can be substituted for a supertype object.

```
Light light = new TubeLight();                  // (1) widening reference conversion
```

- The reference `light` can invoke those methods on the subtype object that are inherited from the supertype `Light`.

```
light.switchOn();                                     // (2)
light.getTubeLength();                              // (3) Not OK.
```

Relationships: ***supertype – subtype***

Note: The compiler only knows about the declared type of the reference `light`, which is `Light`, and ensures that only methods from this type can be called using `light`.

- However, at runtime, the reference `light` will refer to an object of the subtype `TubeLight` when the call to the method `switchOn()` is executed.
- It is the type of the object that the reference is referring to at runtime that determines which method is executed.
- The subtype object inherits the `switchOn()` method from its supertype `Light` , and it is this method that is executed.

The type of the object that the reference refers to at runtime is often called the ***dynamic type*** of the reference.

Overriding Methods

Instance Method Overriding

- A subclass may override instance methods that it inherits from a superclass.
- Overriding such a method allows the subclass to provide its own implementation of the method.
- When the method is invoked on an object of the subclass, it is the method implementation in the subclass that is executed.
- The overridden method in the superclass is not inherited by the subclass.

Overriding Methods

Instance Method Overriding

The new method in the subclass must abide by the following rules of method overriding:

- The new method definition must have the same method signature, i.e., the method name, and the types and the number of parameters, including their order, are the same as in the overridden method.
- Whether parameters in the overriding method should be final is at the discretion of the subclass. A method's signature does not comprise the final modifier of parameters, only their types and order.
- The return type of the overriding method can be a subtype of the return type of the overridden method (called ***covariant return***).

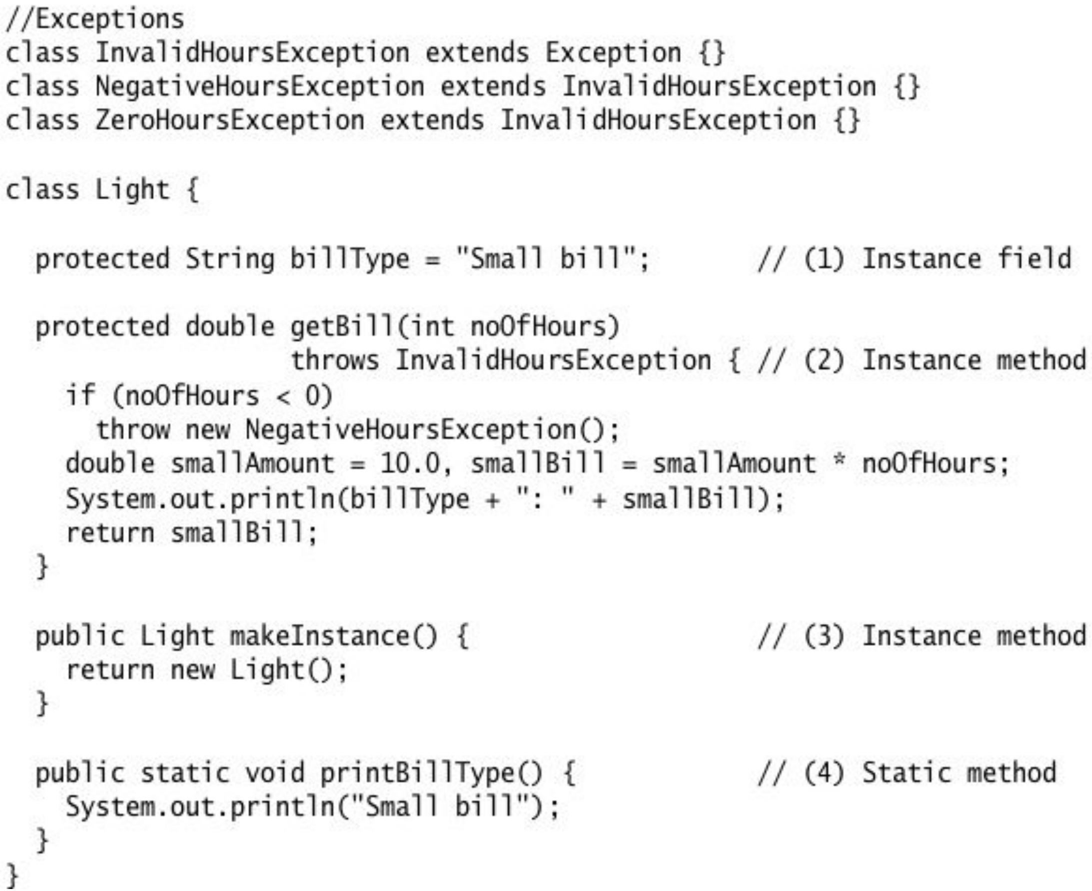
Overriding Methods

Instance Method Overriding

the new method in the subclass must abide by the following rules of method overriding:

- Note that *covariant return* only applies to reference types, not to primitive types. If we try to do so, it will result in a compile-time error. There is no subtype relationship between primitive types.
- The new method definition cannot narrow the accessibility of the method, but it can widen it.
- The new method definition can only throw all or none, or a subset of the checked exceptions (including their subclasses) that are specified in the throws clause of the overridden method in the superclass.

Overriding Methods: Example

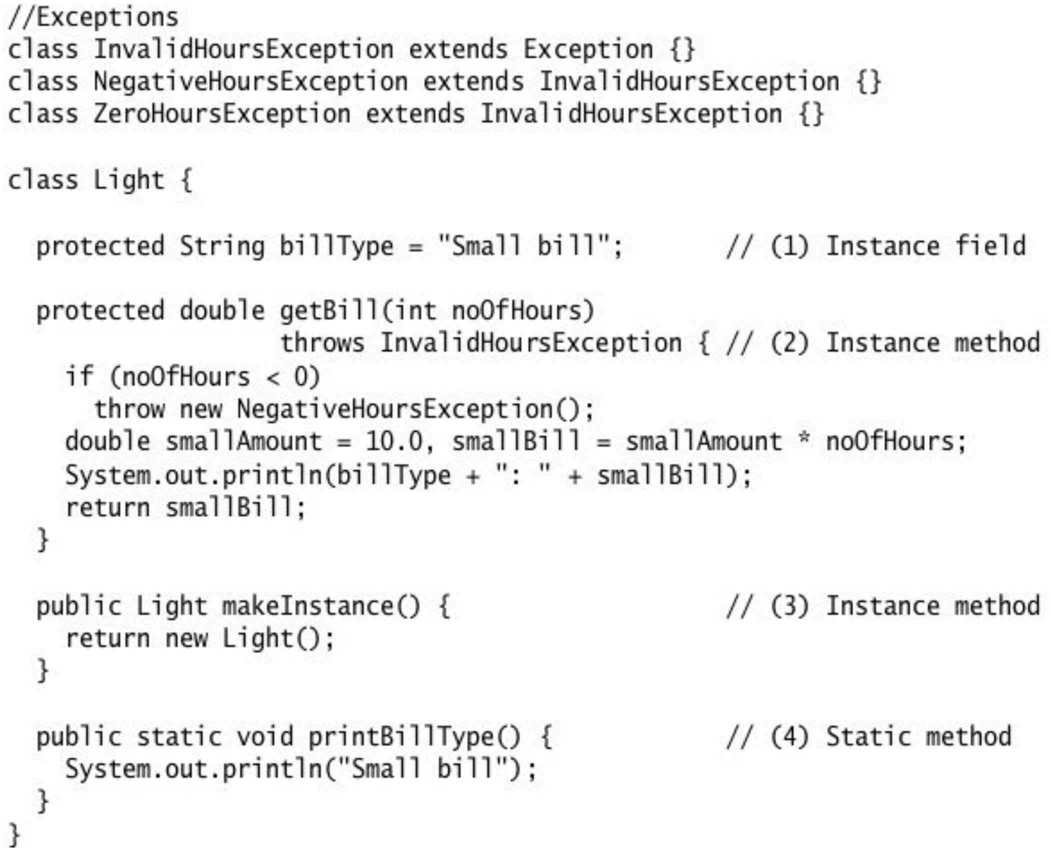


```
//Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

class Light {

  protected String billType = "Small bill";       // (1) Instance field

  protected double getBill(int noOfHours)
                   throws InvalidHoursException { // (2) Instance method
    if (noOfHours < 0)
      throw new NegativeHoursException();
    double smallAmount = 10.0, smallBill = smallAmount * noOfHours;
    System.out.println(billType + ": " + smallBill);
    return smallBill;
  }

  public Light makeInstance() {                   // (3) Instance method
    return new Light();
  }

  public static void printBillType() {            // (4) Static method
    System.out.println("Small bill");
  }
}
```

```
class TubeLight extends Light {

  public static String billType = "Large bill";  // (5) Hiding field at (1).

  @Override
  public double getBill(final int noOfHours)
        throws ZeroHoursException {   // (6) Overriding instance method at (2).
    if (noOfHours == 0)
      throw new ZeroHoursException();
    double largeAmount = 100.0, largeBill = largeAmount * noOfHours;
    System.out.println(billType + ": " + largeBill);
    return largeBill;
  }

  public double getBill() {           // (7) Overloading method at (6).
    System.out.println("No bill");
    return 0.0;
  }

  @Override
  public TubeLight makeInstance() {   // (8) Overriding instance method at (3).
    return new TubeLight();
  }

  public static void printBillType() { // (9) Hiding static method at (4).
    System.out.println(billType);
  }
}
```

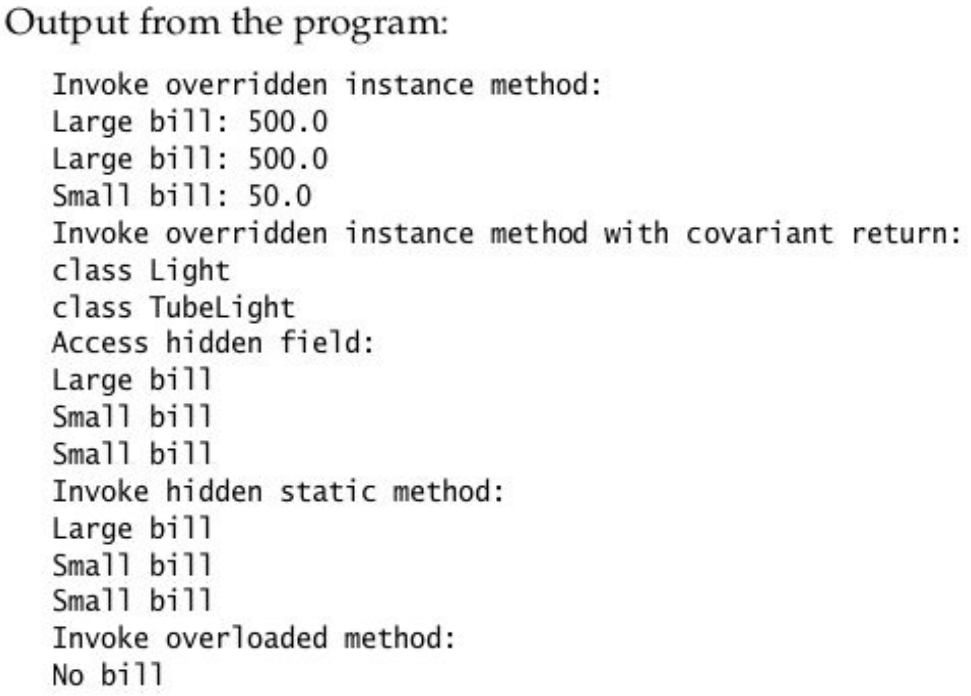
```
public class Client {
  public static void main(String[] args) throws InvalidHoursException { // (10)

    TubeLight tubeLight = new TubeLight();    // (11)
    Light     light1    = tubeLight;          // (12) Aliases.
    Light     light2    = new Light();        // (13)

    System.out.println("Invoke overridden instance method:");
    tubeLight.getBill(5);                     // (14) Invokes method at (6).
    light1.getBill(5);                        // (15) Invokes method at (6).
    light2.getBill(5);                        // (16) Invokes method at (2).

    System.out.println(
           "Invoke overridden instance method with covariant return:");
    System.out.println(
           light2.makeInstance().getClass()); // (17) Invokes method at (3).
    System.out.println(
        tubeLight.makeInstance().getClass()); // (18) Invokes method at (8).

    System.out.println("Access hidden field:");
    System.out.println(tubeLight.billType);   // (19) Accesses field at (5).
    System.out.println(light1.billType);      // (20) Accesses field at (1).
    System.out.println(light2.billType);      // (21) Accesses field at (1).

    System.out.println("Invoke hidden static method:");
    tubeLight.printBillType();                // (22) Invokes method at (9).
    light1.printBillType();                   // (23) Invokes method at (4).
    light2.printBillType();                   // (24) Invokes method at (4).

    System.out.println("Invoke overloaded method:");
    tubeLight.getBill();                      // (25) Invokes method at (7).
  }
}
```

Overriding Methods: Example



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

class Light {

  protected String billType = "Small bill";       // (1) Instance field

  protected double getBill(int noOfHours)
                   throws InvalidHoursException { // (2) Instance method
    if (noOfHours < 0)
      throw new NegativeHoursException();
    double smallAmount = 10.0, smallBill = smallAmount * noOfHours;
    System.out.println(billType + ": " + smallBill);
    return smallBill;
  }

  public Light makeInstance() {                   // (3) Instance method
    return new Light();
  }

  public static void printBillType() {            // (4) Static method
    System.out.println("Small bill");
  }
}
```

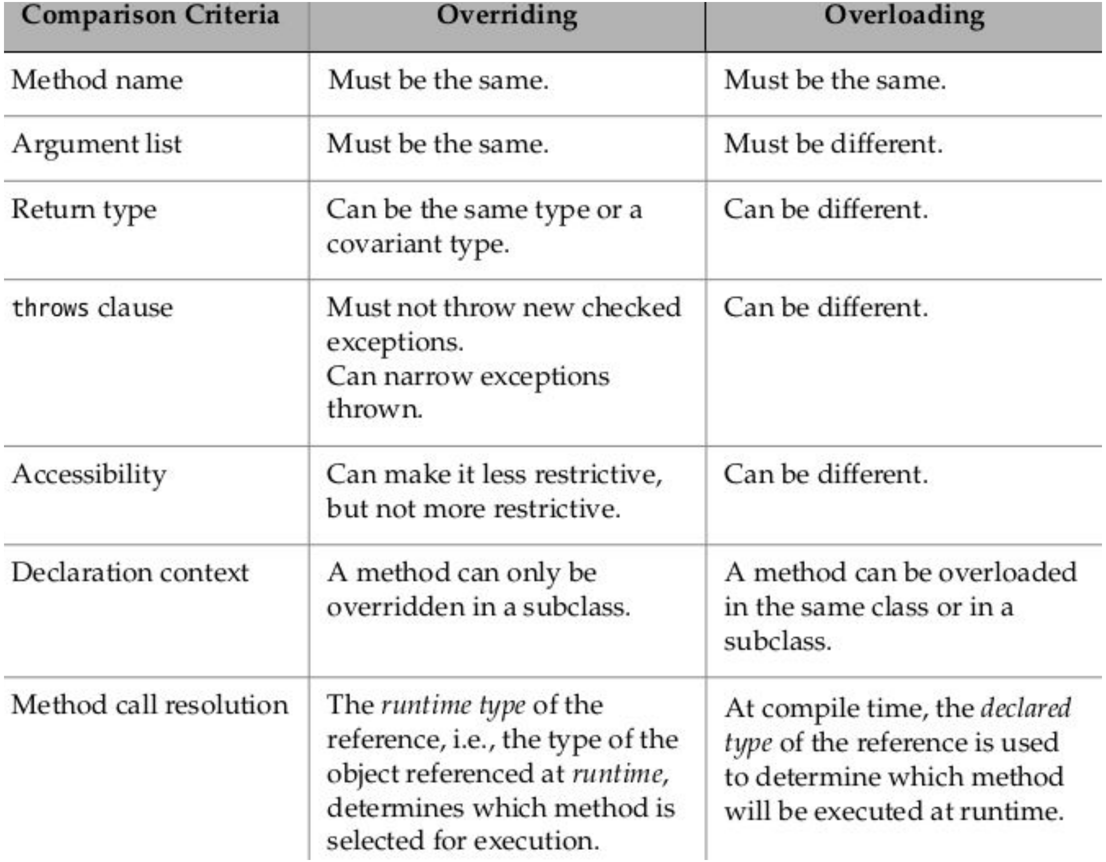
Overriding Methods: Example



Output from the program:

```
Invoke overridden instance method:
Large bill: 500.0
Large bill: 500.0
Small bill: 50.0
Invoke overridden instance method with covariant return:
class Light
class TubeLight
Access hidden field:
Large bill
Small bill
Small bill
Invoke hidden static method:
Large bill
Small bill
Small bill
Invoke overloaded method:
No bill
```

Overriding vs. Overloading



Comparison Criteria	Overriding	Overloading
Method name	Must be the same.	Must be the same.
Argument list	Must be the same.	Must be different.
Return type	Can be the same type or a covariant type.	Can be different.
`throws` clause	Must not throw new checked exceptions. Can narrow exceptions thrown.	Can be different.
Accessibility	Can make it less restrictive, but not more restrictive.	Can be different.
Declaration context	A method can only be overridden in a subclass.	A method can be overloaded in the same class or in a subclass.
Method call resolution	The *runtime type* of the reference, i.e., the type of the object referenced at *runtime*, determines which method is selected for execution.	At compile time, the *declared type* of the reference is used to determine which method will be executed at runtime.

Hiding Members

Field Hiding

A subclass cannot override fields of the superclass, but it can hide them.

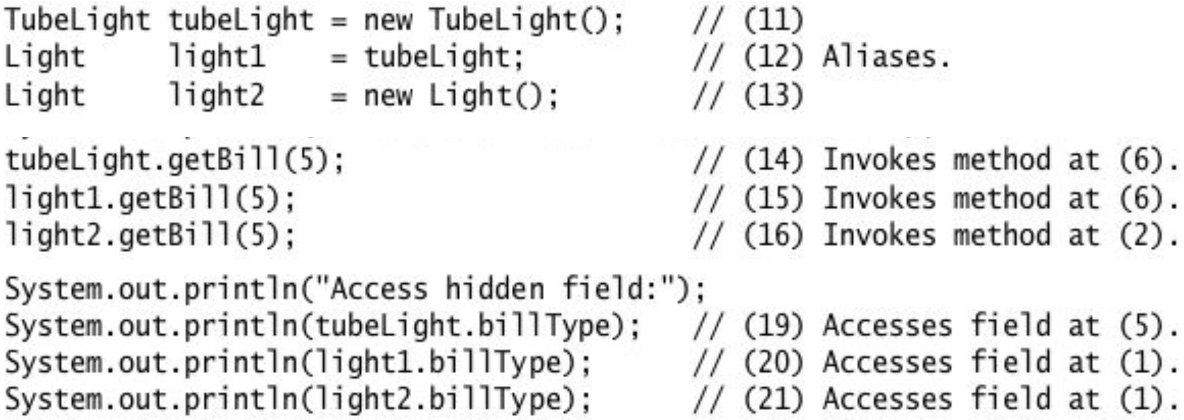
- The subclass can define fields with the same name as in the superclass. In this case, the fields in the superclass cannot be accessed in the subclass by their simple names; therefore, they are not inherited by the subclass.
- Code in the subclass can use the keyword ***super*** to access such members, including hidden fields.
- If the hidden field is static, it can also be accessed by the superclass name.

Hiding Members

Field Hiding

A client can use a reference of the superclass to access members that are hidden in the subclass.

- When a field of an object is accessed using a reference, (unlike the instance method invocation) it is the type of the reference, not the class of the current object denoted by the reference, that determines which field will actually be accessed.



```
TubeLight tubeLight = new TubeLight();    // (11)
Light     light1    = tubeLight;          // (12) Aliases.
Light     light2    = new Light();        // (13)

tubeLight.getBill(5);                     // (14) Invokes method at (6).
light1.getBill(5);                        // (15) Invokes method at (6).
light2.getBill(5);                        // (16) Invokes method at (2).

System.out.println("Access hidden field:");
System.out.println(tubeLight.billType);   // (19) Accesses field at (5).
System.out.println(light1.billType);      // (20) Accesses field at (1).
System.out.println(light2.billType);      // (21) Accesses field at (1).
```

Hiding Members

Static Method Hiding

A static method cannot override an inherited instance method, but it can hide a static method if the exact requirements for overriding instance methods are fulfilled.

- A hidden superclass static method is not inherited.
- The compiler will flag an error if the signatures are the same, but the other requirements regarding return type, throws clause, and accessibility are not met.
- If the signatures are different, the method name is overloaded, not hidden.

Hiding Members

Static Method Hiding

- A hidden static method can always be invoked by using the superclass name in the subclass declaration.
- The keyword ***super*** can be used in non-static code in the subclass declaration to invoke hidden static methods.
- Analogous to accessing fields, the method invoked is determined by the class of the reference.

```
tubeLight.printBillType();                    // (22) Invokes method at (9).
light1.printBillType();                       // (23) Invokes method at (4).
light2.printBillType();                       // (24) Invokes method at (4).
```

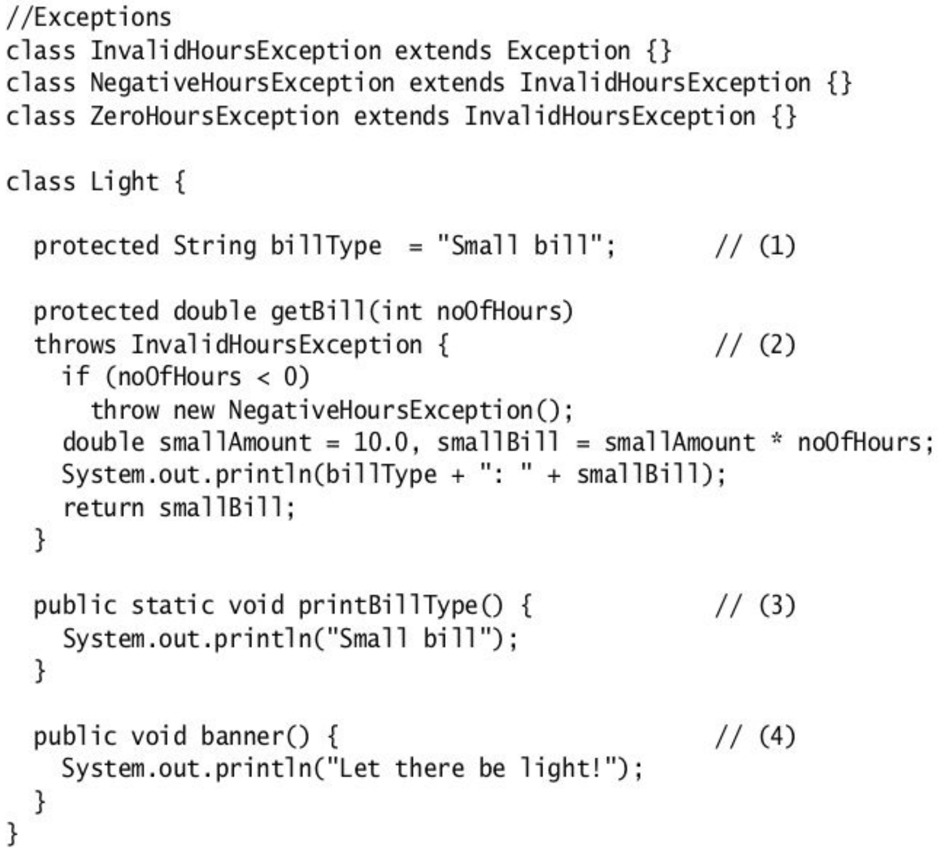
- In (22) the class type is TubeLight, therefore, the static method `printBillType()` of TubeLight class is invoked. In (23) and (24), the class type is Light and the hidden static method `printBillType()` in that class is invoked.

The Object Reference ***super***

The keyword ***super*** can be used in non-static code (e.g., in the body of an instance method), but only in a subclass, to access fields and invoke methods from the superclass.

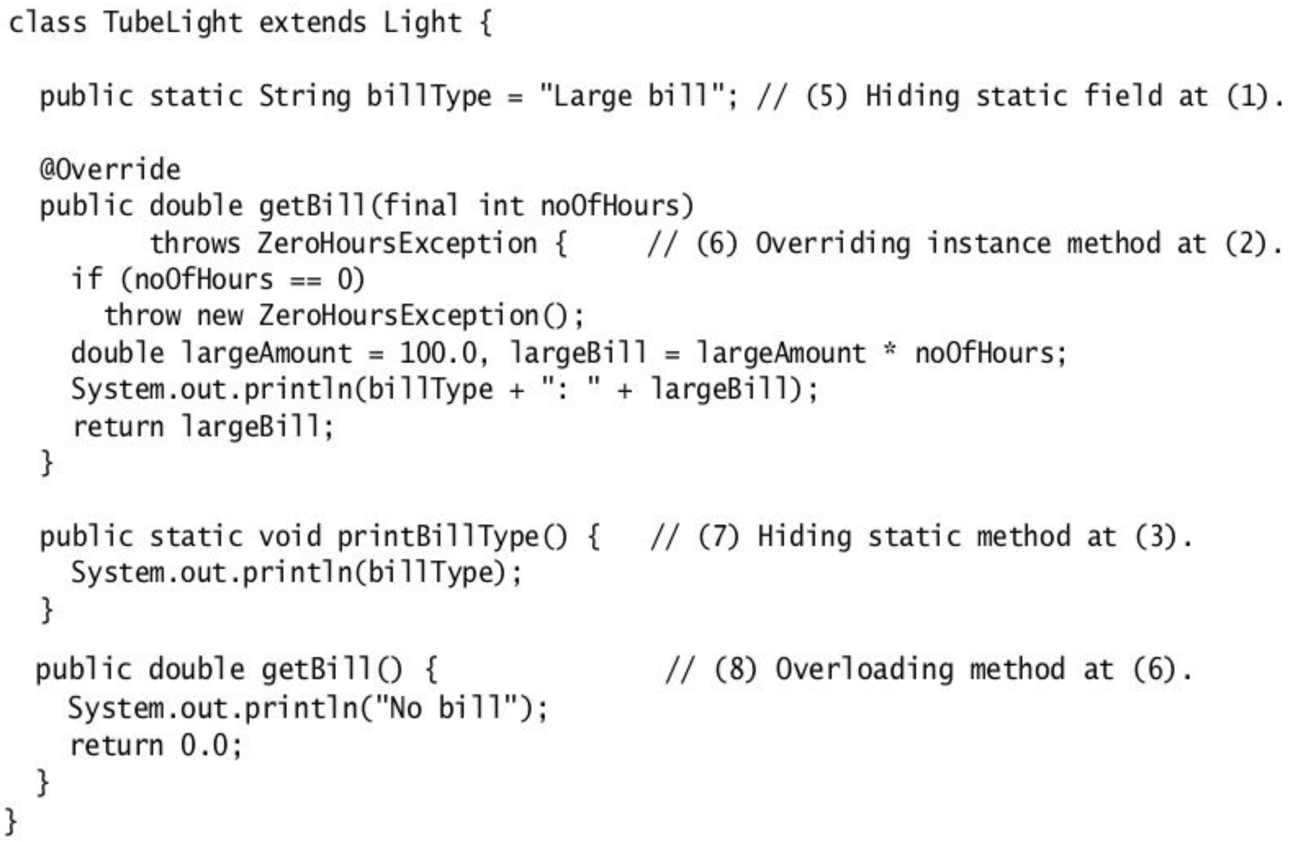
- The keyword ***super*** provides a reference to the current object as an instance of its superclass.
- In method invocations with ***super***, the method from the superclass is invoked regardless of the actual type of the object or whether the current class overrides the method.
- It is typically used to invoke methods that are overridden and to access members that are hidden in the subclass.
- Unlike the ***this*** keyword, the ***super*** keyword cannot be used as an ordinary reference. For example, it cannot be assigned to other references or cast to other reference types.

super : Example



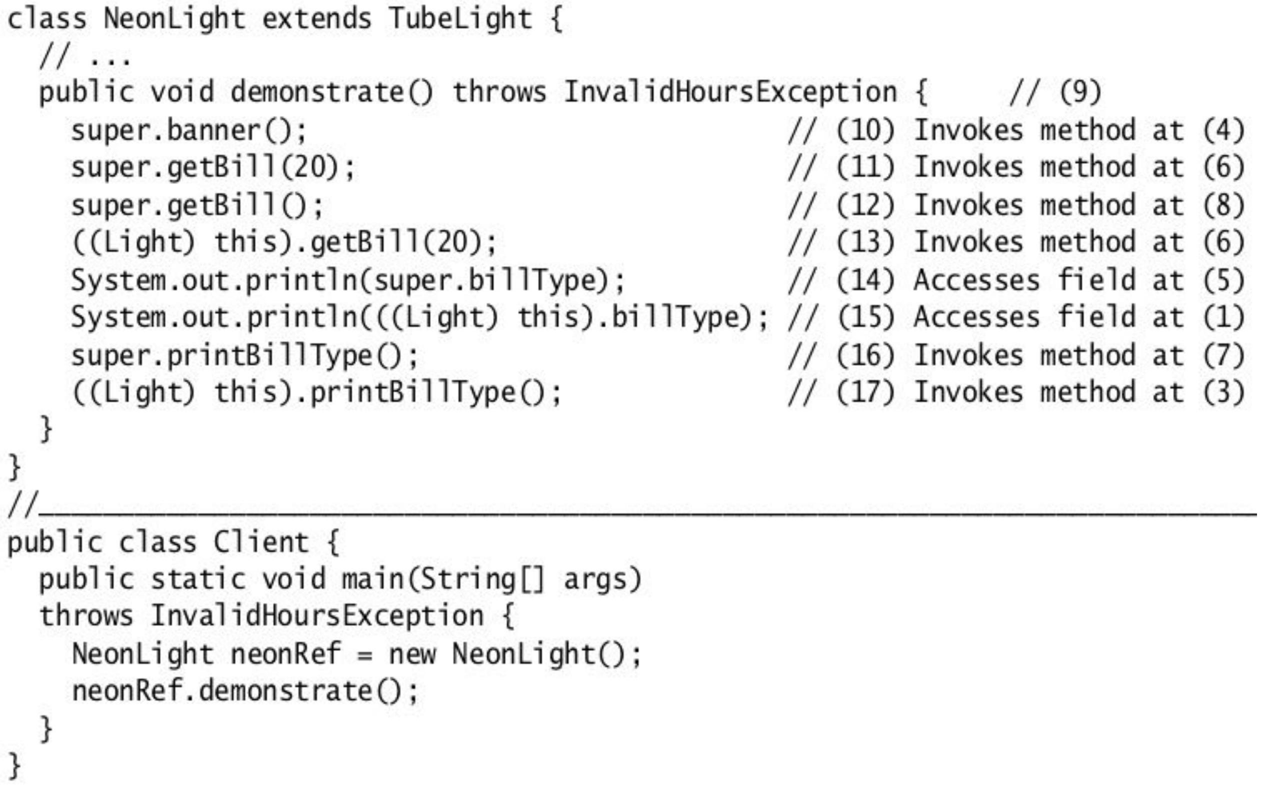
```
//Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

class Light {

  protected String billType  = "Small bill";       // (1)

  protected double getBill(int noOfHours)
  throws InvalidHoursException {                   // (2)
    if (noOfHours < 0)
      throw new NegativeHoursException();
    double smallAmount = 10.0, smallBill = smallAmount * noOfHours;
    System.out.println(billType + ": " + smallBill);
    return smallBill;
  }

  public static void printBillType() {             // (3)
    System.out.println("Small bill");
  }

  public void banner() {                           // (4)
    System.out.println("Let there be light!");
  }
}
```

super : Example



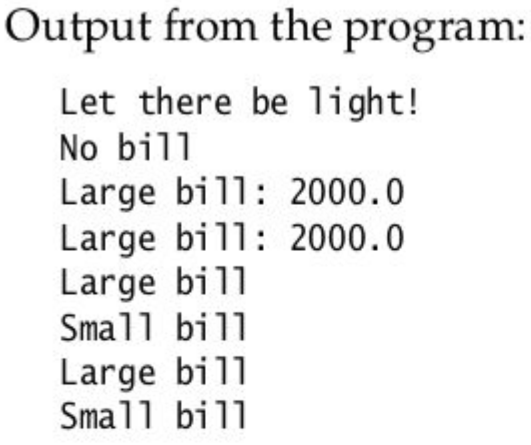
```java
class TubeLight extends Light {

  public static String billType = "Large bill"; // (5) Hiding static field at (1).

  @Override
  public double getBill(final int noOfHours)
         throws ZeroHoursException {     // (6) Overriding instance method at (2).
    if (noOfHours == 0)
      throw new ZeroHoursException();
    double largeAmount = 100.0, largeBill = largeAmount * noOfHours;
    System.out.println(billType + ": " + largeBill);
    return largeBill;
  }

  public static void printBillType() {   // (7) Hiding static method at (3).
    System.out.println(billType);
  }

  public double getBill() {               // (8) Overloading method at (6).
    System.out.println("No bill");
    return 0.0;
  }
}
```

super : Example



```
class NeonLight extends TubeLight {
  // ...
  public void demonstrate() throws InvalidHoursException {    // (9)
    super.banner();                              // (10) Invokes method at (4)
    super.getBill(20);                           // (11) Invokes method at (6)
    super.getBill();                             // (12) Invokes method at (8)
    ((Light) this).getBill(20);                  // (13) Invokes method at (6)
    System.out.println(super.billType);          // (14) Accesses field at (5)
    System.out.println(((Light) this).billType); // (15) Accesses field at (1)
    super.printBillType();                       // (16) Invokes method at (7)
    ((Light) this).printBillType();              // (17) Invokes method at (3)
  }
}
//____________________________________________________________________________
public class Client {
  public static void main(String[] args)
  throws InvalidHoursException {
    NeonLight neonRef = new NeonLight();
    neonRef.demonstrate();
  }
}
```

super : Example



Output from the program:

```
Let there be light!
No bill
Large bill: 2000.0
Large bill: 2000.0
Large bill
Small bill
Large bill
Small bill
```

IT602: Object-Oriented Programming

Next lecture -

Inheritance contd. . .