IT602: Object-Oriented Programming

Lecture - 13

Inheritance

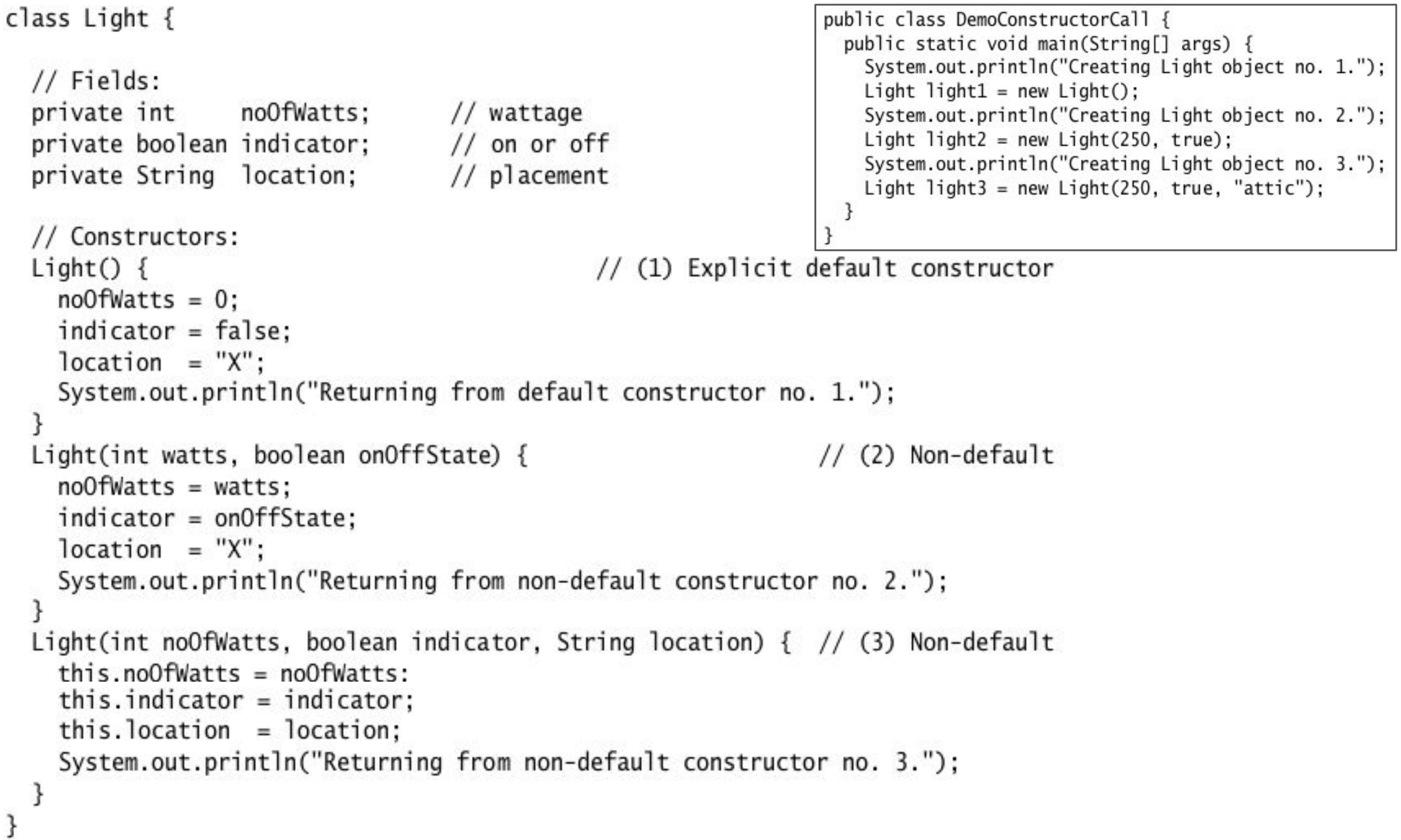
Arpit Rana

29th March 2022

Chaining Constructors using `this()` and `super()`

- Constructors cannot be inherited or overridden. They can be overloaded, but only in the same class.
- Since a constructor always has the same name as the class, each parameter list must be different when defining more than one constructor for a class.
    - The ***`this`*** reference is used to access the fields shadowed by the parameters (see example below).

Example: Use of *this*



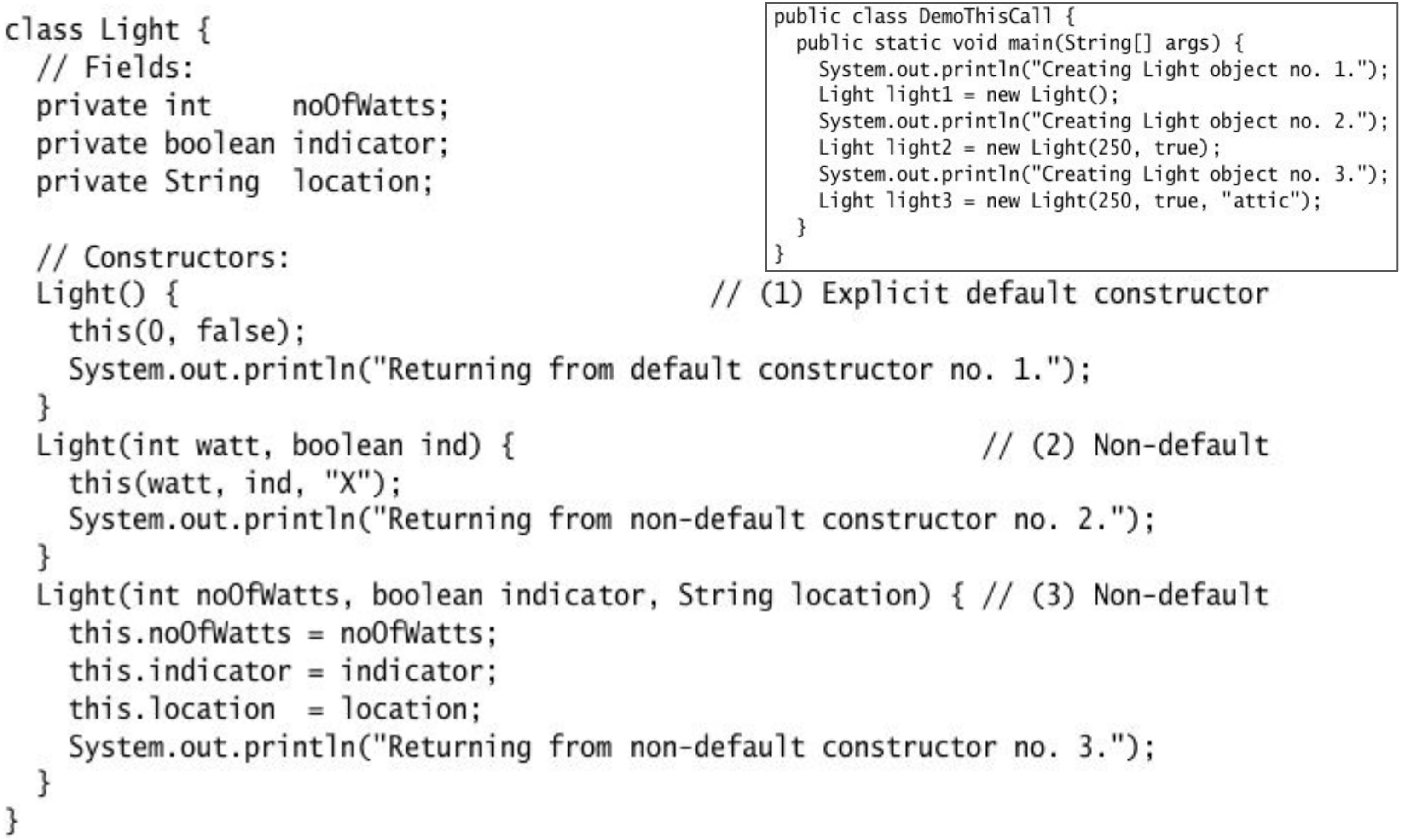
```
public class DemoConstructorCall {
  public static void main(String[] args) {
    System.out.println("Creating Light object no. 1.");
    Light light1 = new Light();
    System.out.println("Creating Light object no. 2.");
    Light light2 = new Light(250, true);
    System.out.println("Creating Light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

```
class Light {

  // Fields:
  private int     noOfWatts;      // wattage
  private boolean indicator;      // on or off
  private String  location;       // placement

  // Constructors:
  Light() {                                  // (1) Explicit default constructor
    noOfWatts = 0;
    indicator = false;
    location  = "X";
    System.out.println("Returning from default constructor no. 1.");
  }
  Light(int watts, boolean onOffState) {                    // (2) Non-default
    noOfWatts = watts;
    indicator = onOffState;
    location  = "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  = location;
    System.out.println("Returning from non-default constructor no. 3.");
  }
}
```

Use of the ***`this()`*** construct

- The ***`this()`*** call invokes the local constructor with the corresponding parameter list.
- Java requires that any ***`this()`*** call must occur as the first statement in a constructor.

Example: Use of the ***this()***



```
class Light {
  // Fields:
  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  = location;
    System.out.println("Returning from non-default constructor no. 3.");
  }
}
```

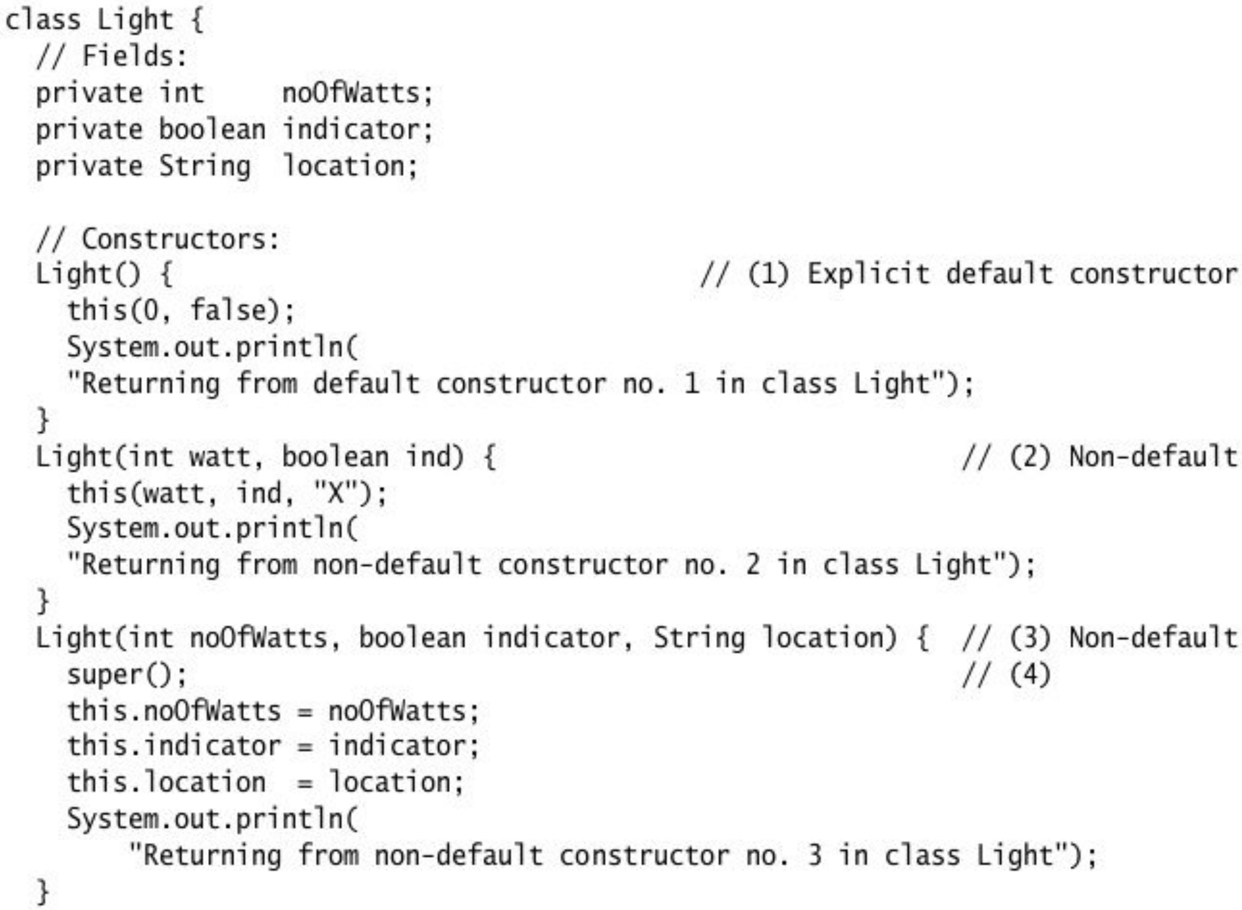
```
public class DemoThisCall {
  public static void main(String[] args) {
    System.out.println("Creating Light object no. 1.");
    Light light1 = new Light();
    System.out.println("Creating Light object no. 2.");
    Light light2 = new Light(250, true);
    System.out.println("Creating Light object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

Use of the ***super()*** construct

The ***super()*** construct is used in a subclass constructor to invoke a constructor in the immediate superclass.

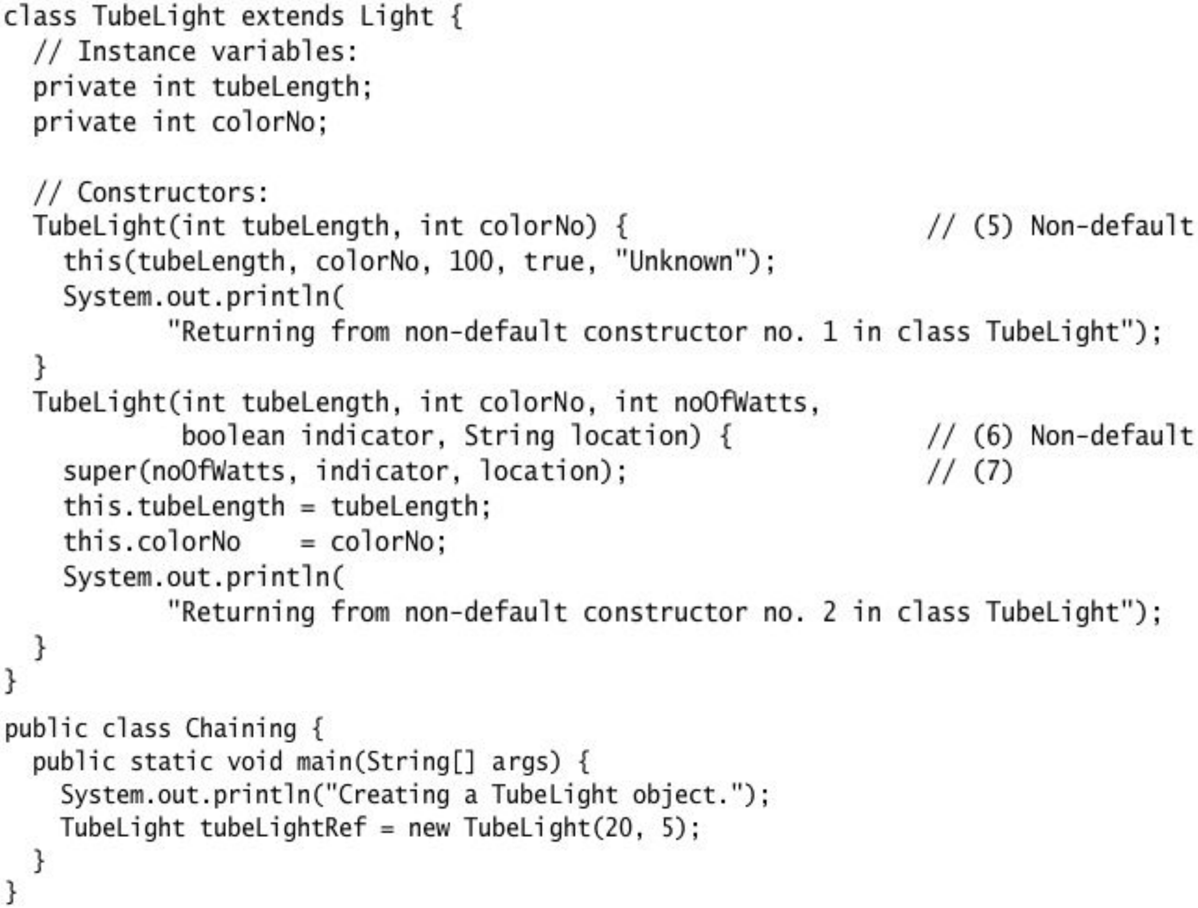
- A ***super()*** call in the constructor of a subclass will result in the execution of the relevant constructor from the superclass, based on the signature of the call.
- Since the superclass name is known in the subclass declaration, the compiler can determine the superclass constructor invoked from the signature of the parameter list.

Example: Use of the *super()*



```
class Light {
  // Fields:
  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  = location;
    System.out.println(
        "Returning from non-default constructor no. 3 in class Light");
  }
```

Example: Use of the ***super()***



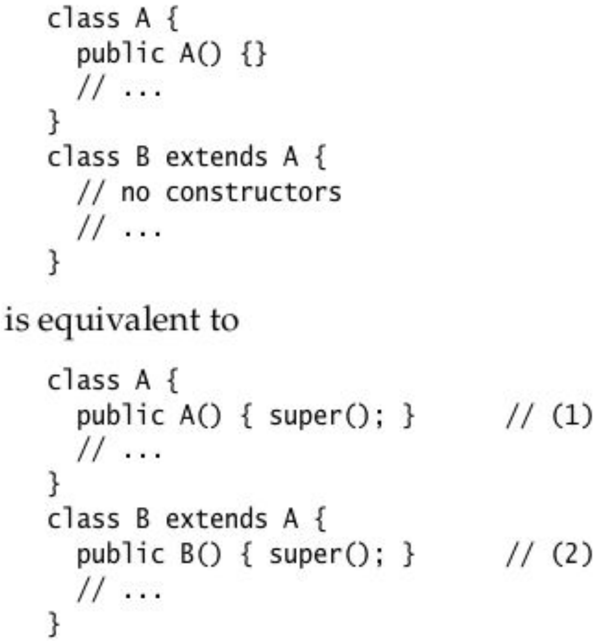
```
class TubeLight extends Light {
  // Instance variables:
  private int tubeLength;
  private int colorNo;

  // Constructors:
  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);
  }
}
```

Use of the ***`super()`*** construct

- The ***`super()`*** call must occur as the first statement in a constructor, and it can only be used in a constructor declaration.
- This implies that ***`this()`*** and ***`super()`*** calls cannot both occur in the same constructor.
- The ***`this()`*** construct leads to chaining of constructors in the same class, whereas the ***`super()`*** construct leads to chaining of subclass constructors to superclass constructors (up to the ***`Object`*** class).
- This is called (subclass–superclass) ***constructor chaining***.

Use of the ***super()*** construct

- If a constructor has neither a ***this()*** nor a ***super()*** call as its first statement, the compiler inserts a ***super()*** call to the default constructor in the superclass.



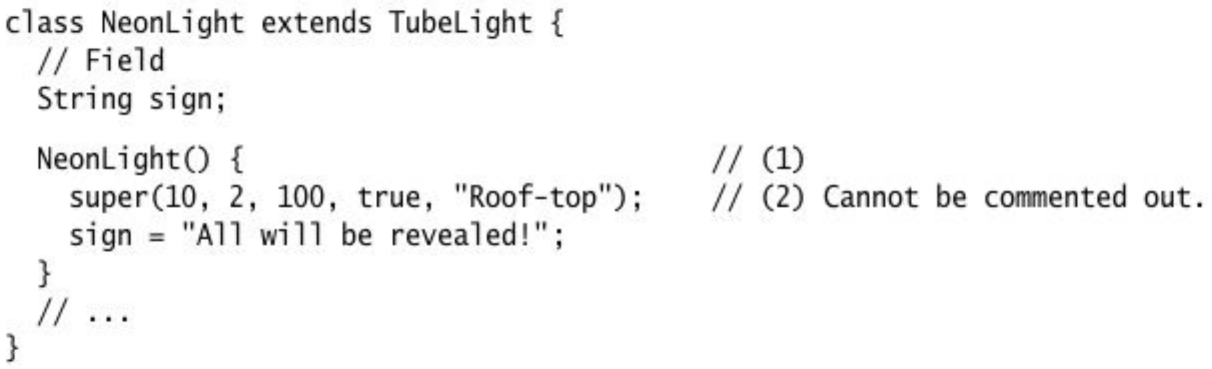
```
class A {
  public A() {}
  // ...
}
class B extends A {
  // no constructors
  // ...
}
```

is equivalent to

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

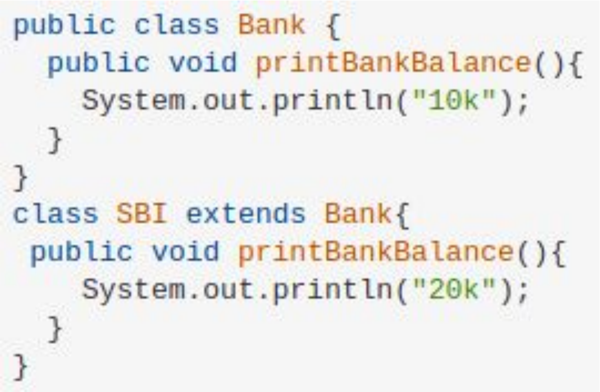
Use of the ***super()*** construct

- If a superclass only defines non-default constructors (i.e., only constructors with parameters):
    - Its subclasses cannot rely on the implicit ***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.



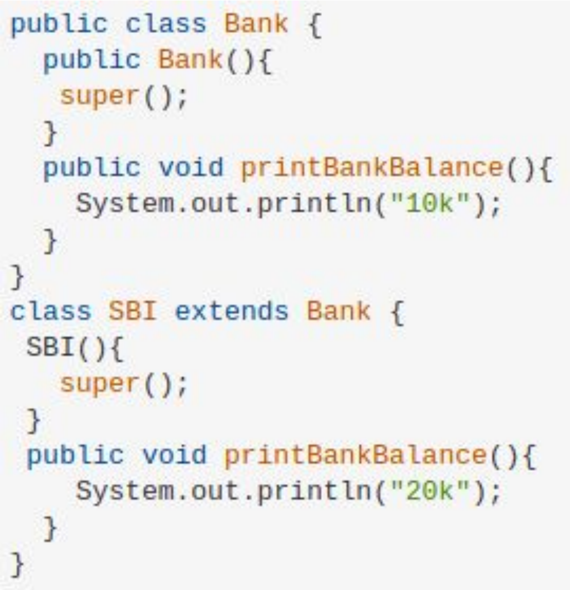
```
class NeonLight extends TubeLight {
  // Field
  String sign;

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

Java doesn’t Support Multiple Inheritance



```
public class Bank {
  public void printBankBalance(){
    System.out.println("10k");
  }
}
class SBI extends Bank{
 public void printBankBalance(){
    System.out.println("20k");
  }
}
```

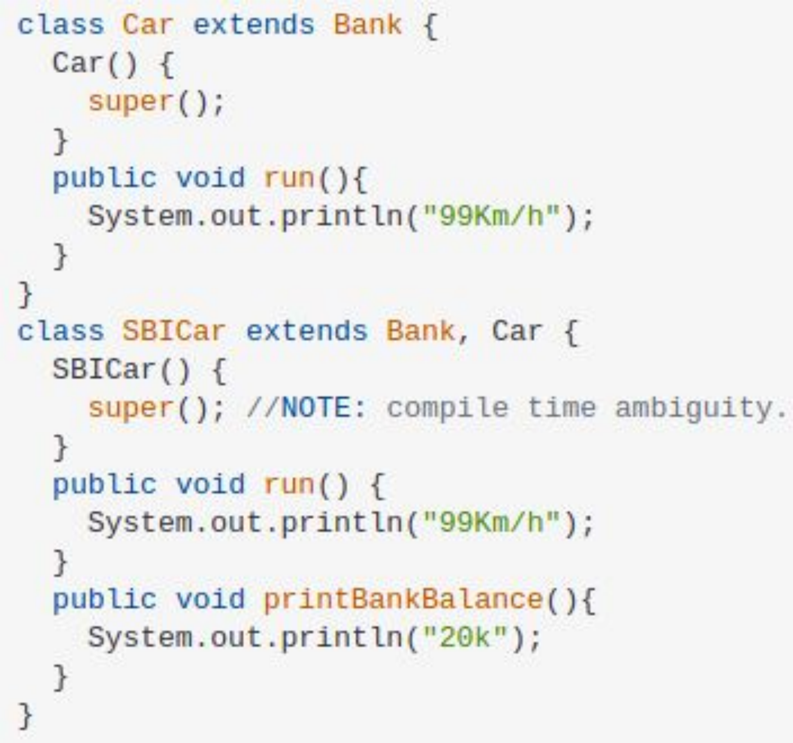
On compile



```
public class Bank {
  public Bank(){
   super();
  }
  public void printBankBalance(){
    System.out.println("10k");
  }
}
class SBI extends Bank {
 SBI(){
   super();
 }
 public void printBankBalance(){
    System.out.println("20k");
  }
}
```

Java doesn’t Support Multiple Inheritance

- In this case (`SBICar`) will fail to create constructor chain (compile time ambiguity).
- For interfaces this is allowed because we cannot create an object of it.

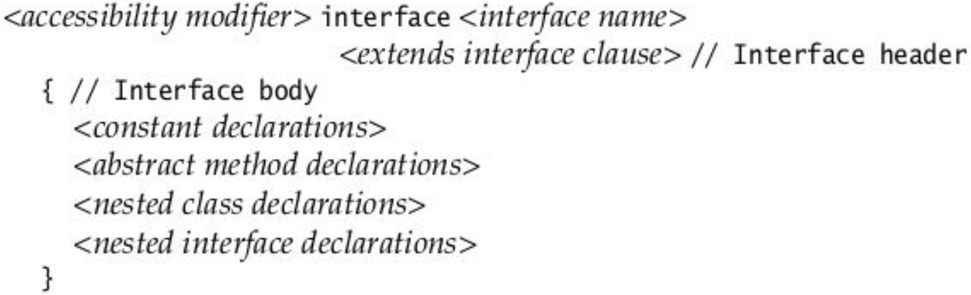


```
class Car extends Bank {
  Car() {
    super();
  }
  public void run(){
    System.out.println("99Km/h");
  }
}
class SBICar extends Bank, Car {
  SBICar() {
    super(); //NOTE: compile time ambiguity.
  }
  public void run() {
    System.out.println("99Km/h");
  }
  public void printBankBalance(){
    System.out.println("20k");
  }
}
```

Interfaces

Java provides interfaces, which allow new named reference types to be introduced, and also permit ***multiple interface inheritance***.

- A top-level interface has the following general syntax:



```
<accessibility modifier> interface <interface name>
                                   <extends interface clause> // Interface header
  { // Interface body
     <constant declarations>
     <abstract method declarations>
     <nested class declarations>
     <nested interface declarations>
  }
```

- The interface header can specify: the scope or accessibility modifier and any interfaces it extends.

- The interface body can contain member declarations which comprise: constant declarations, abstract method declarations, nested class and interface declarations.

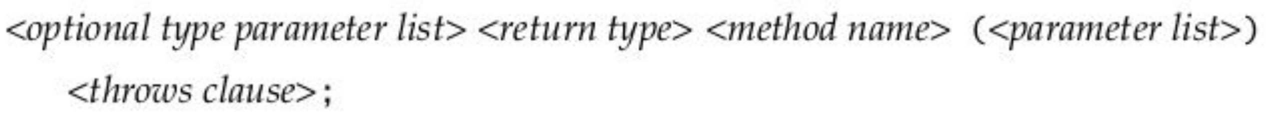
Interfaces

Java provides interfaces, which allow new named reference types to be introduced, and also permit ***multiple interface inheritance***.

- An interface does not provide any implementation and is, therefore, abstract by definition. This means that it cannot be instantiated.
- The member declarations can appear in any order in the interface body.
- Interface members implicitly have public accessibility (meant to be implemented by classes) and the public modifier can be omitted.
- Interfaces with empty bodies can be used as ***markers*** to tag classes as having a certain property or behavior.
    - Such interfaces are also called ***ability*** interfaces.
    - Java APIs provide several examples of such marker interfaces: `java.lang.Cloneable`,`java.io.Serializable`, `java.util.EventListener`.

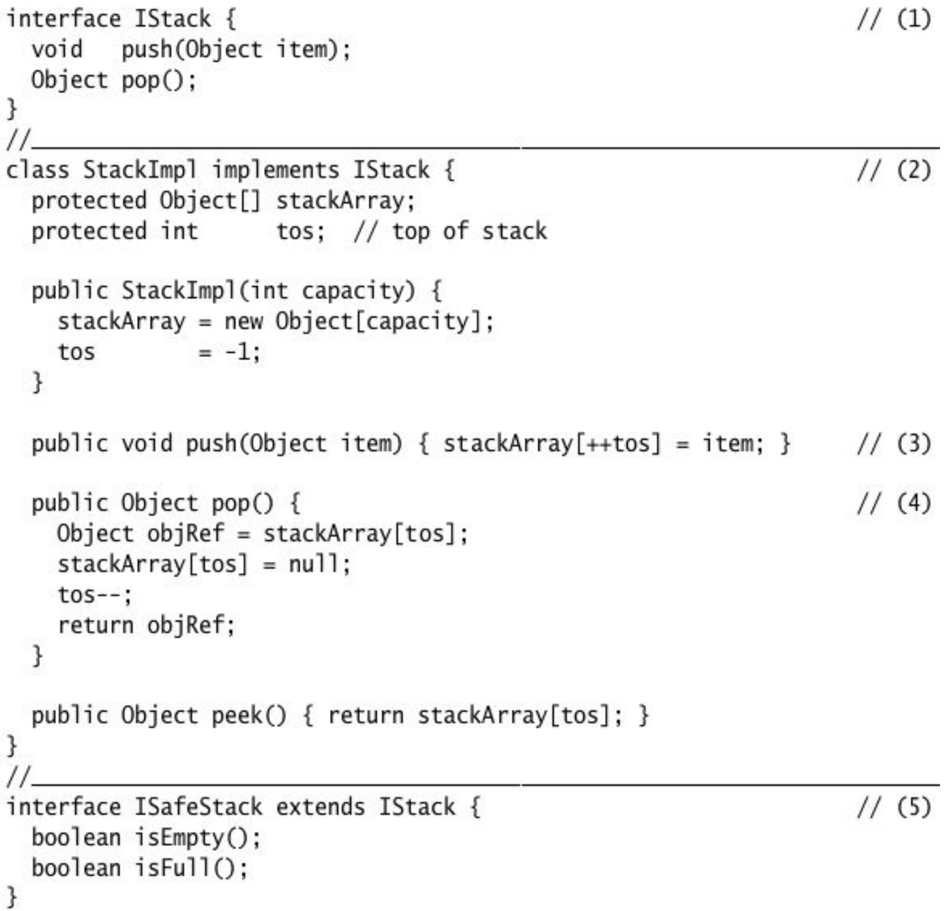
Abstract Method Declaration

- An interface defines a contract by specifying a set of abstract method declarations, but provides no implementations.
- The methods in an interface are all implicitly `abstract` and `public` by virtue of their definition.
- Only the modifiers `abstract` and `public` are allowed, but these are invariably omitted.



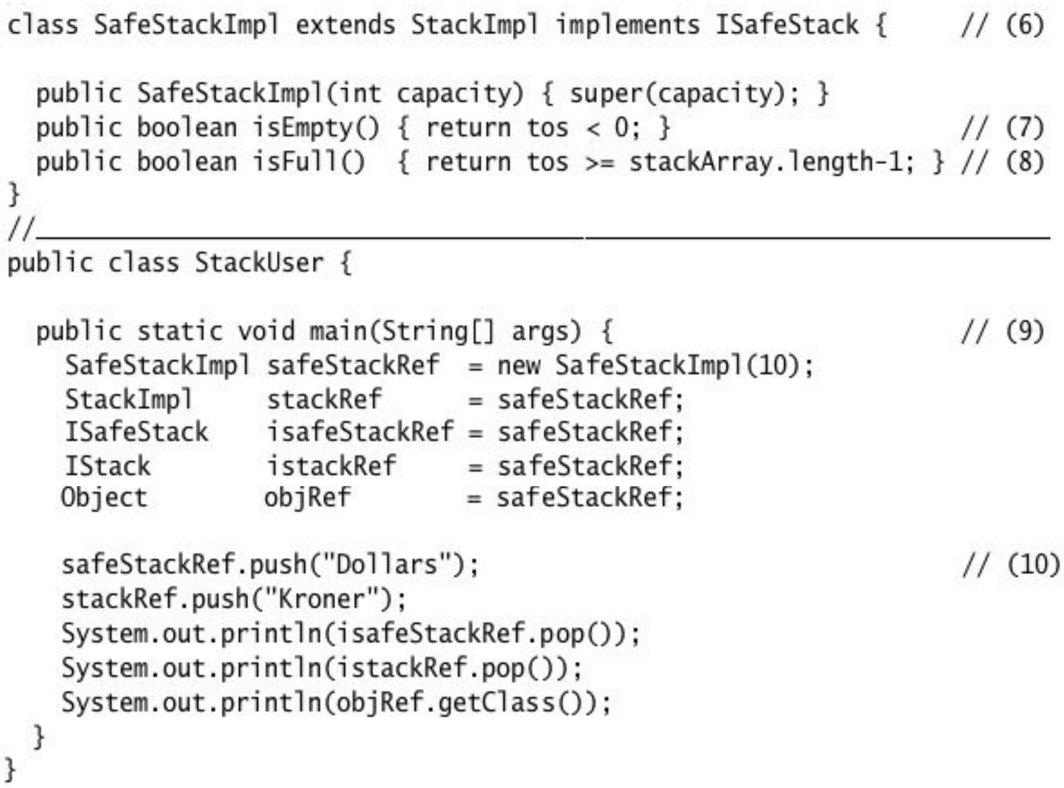
<optional type parameter list> <return type> <method name> (*<parameter list>*)
  <throws clause>;

Interface: Example



```
interface IStack {                                                  // (1)
  void   push(Object item);
  Object pop();
}
//________________________________________________________________________
class StackImpl implements IStack {                                 // (2)
  protected Object[] stackArray;
  protected int      tos;  // top of stack

  public StackImpl(int capacity) {
    stackArray = new Object[capacity];
    tos        = -1;
  }

  public void push(Object item) { stackArray[++tos] = item; }     // (3)

  public Object pop() {                                             // (4)
    Object objRef = stackArray[tos];
    stackArray[tos] = null;
    tos--;
    return objRef;
  }

  public Object peek() { return stackArray[tos]; }
}
//________________________________________________________________________
interface ISafeStack extends IStack {                               // (5)
  boolean isEmpty();
  boolean isFull();
}
```

Interface: Example



```
class SafeStackImpl extends StackImpl implements ISafeStack {     // (6)

  public SafeStackImpl(int capacity) { super(capacity); }
  public boolean isEmpty() { return tos < 0; }                     // (7)
  public boolean isFull()  { return tos >= stackArray.length-1; } // (8)
}
//____________________________________________________________________
public class StackUser {

  public static void main(String[] args) {                         // (9)
    SafeStackImpl safeStackRef  = new SafeStackImpl(10);
    StackImpl     stackRef      = safeStackRef;
    ISafeStack    isafeStackRef = safeStackRef;
    IStack        istackRef     = safeStackRef;
    Object        objRef        = safeStackRef;

    safeStackRef.push("Dollars");                                   // (10)
    stackRef.push("Kroner");
    System.out.println(isafeStackRef.pop());
    System.out.println(istackRef.pop());
    System.out.println(objRef.getClass());
  }
}
```

Implementing Interfaces

- Any class can elect to implement, wholly or partially, zero or more interfaces.
- A class specifies the interfaces it implements as a comma-separated list of unique interface names in an implements clause in the class header.
- A class can neither narrow the accessibility of an interface method nor specify new exceptions in the method’s throws clause (is illegal).
- The criteria for overriding methods also apply when implementing interface methods

Implementing Interfaces

- A class can choose to implement only some of the methods of its interfaces (i.e., give a partial implementation of its interfaces). The class must then be declared as abstract.

- Note that interface methods cannot be declared `static`, because they comprise the contract fulfilled by the objects of the class implementing the interface.

- Interface methods are always implemented as instance methods.

Extending Interfaces

- An interface can extend other interfaces, using the extends clause. Unlike extending classes, an interface can extend several interfaces.
- The interfaces extended by an interface (directly or indirectly) are called ***superinterfaces***. Conversely, the interface is a ***subinterface*** of its ***superinterfaces***.
- A subinterface inherits all methods from its superinterfaces, as their method declarations are all implicitly public .
- A subinterface can override abstract method declarations from its superinterfaces. Overridden methods are not inherited.
- Abstract method declarations can also be overloaded, analogous to method overloading in classes.

Extending Interfaces

Note that there are three different inheritance relations at work when defining inheritance among classes and interfaces:

- Single implementation inheritance hierarchy between classes: a class extends another class (subclasses–superclasses).
- Multiple inheritance hierarchy between interfaces: an interface extends other interfaces (subinterfaces–superinterfaces).
- Multiple interface inheritance hierarchy between interfaces and classes: a class implements interfaces.

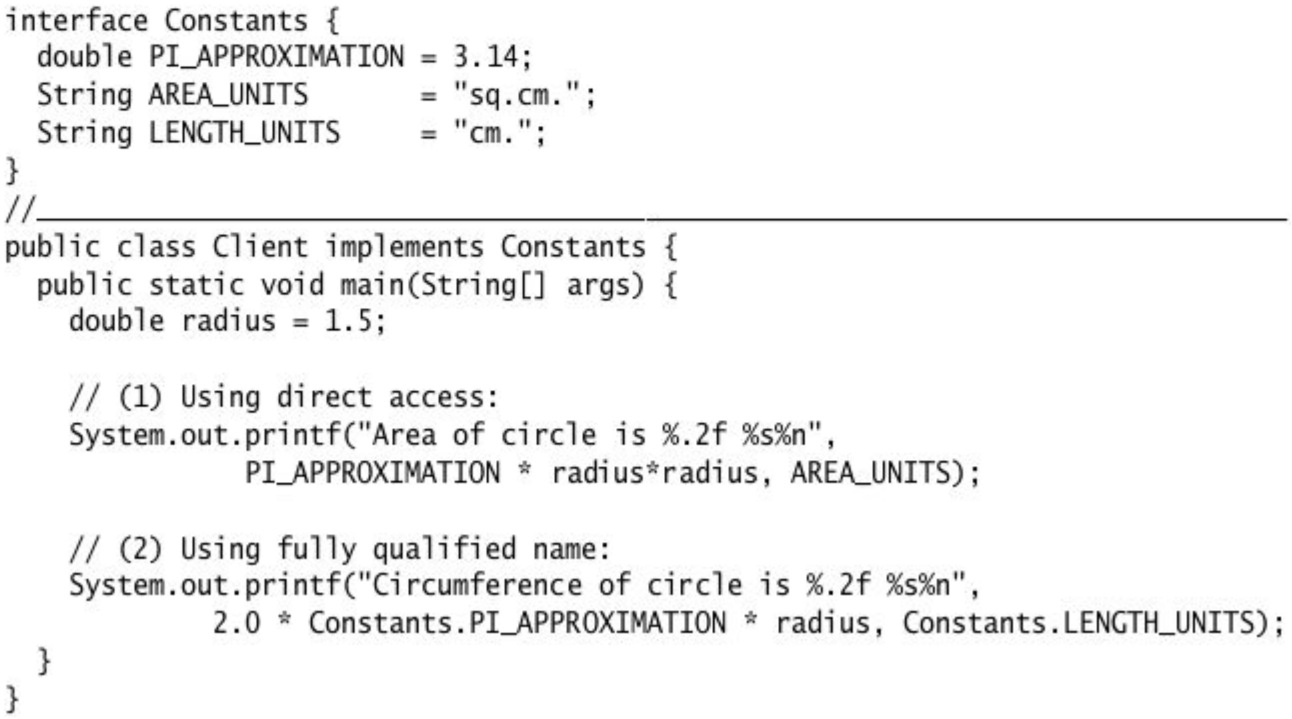
Interface References

- Although interfaces cannot be instantiated, references of an interface type can be declared.
- The reference value of an object can be assigned to references of the object’s supertypes.

Interface Constants

- An interface can also define named constants.
- Such constants are defined by field declarations and are considered to be `public`, `static`, and `final` (can be omitted from the declaration).
- An interface constant can be accessed by any client (a class or interface) using its fully qualified name, regardless of whether the client extends or implements its interface.
- if a client is a class that implements this interface or an interface that extends this interface, then the client can also access such constants directly by their simple names.
- Extending an interface that has constants is analogous to extending a class having `static` variables. In particular, these constants can be hidden by the subinterfaces.

Interface Constants



```
interface Constants {
  double PI_APPROXIMATION = 3.14;
  String AREA_UNITS       = "sq.cm.";
  String LENGTH_UNITS     = "cm.";
}
//________________________________________________________________________________
public class Client implements Constants {
  public static void main(String[] args) {
    double radius = 1.5;

    // (1) Using direct access:
    System.out.printf("Area of circle is %.2f %s%n",
               PI_APPROXIMATION * radius*radius, AREA_UNITS);

    // (2) Using fully qualified name:
    System.out.printf("Circumference of circle is %.2f %s%n",
             2.0 * Constants.PI_APPROXIMATION * radius, Constants.LENGTH_UNITS);
  }
}
```

IT602: Object-Oriented Programming

Next lecture -

Arrays and Subtyping