Chapter 7

Defining classes

Lecture slides for:

Java Actually: A Comprehensive Primer in Programming

Khalid Azim Mughal, Torill Hamre, Rolf W. Rasmussen

Cengage Learning, 2008.

ISBN: 978-1-844480-933-2

`http://www.ii.uib.no/~khalid/jac/`

Modified: 18/2/18

Overview

- Class declarations
- Field variable declarations
- Object creation and references
- Method declarations
- Parameter Passing:
  - Messages/method call
  - Actual parameters
  - Formal parameters
  - Passing of parameter by value
  - Arrays as parameters
- The reference `this`
- Static members
- Types
- Local blocks and variables
- Variables: Scope and Life time
- Program Arguments
- Constructors:
  - Constructor declarations
  - Constructor call
  - Use of the default constructor
- Enum types

Class Declaration

- In Java a *class declaration* comprises a list of *variable* and *method declarations*.
  - Declarations can be specified in any order.

```
<classHeader> {
    <variableDeclarations>
    ...
    <methodDeclarations>
    ...
}
```

- A class declaration is identified by a `<classHeader>` that contains the keyword `class` and the name of the class.

```
class Light {
    ...
}
```

Declaring the class `Light`

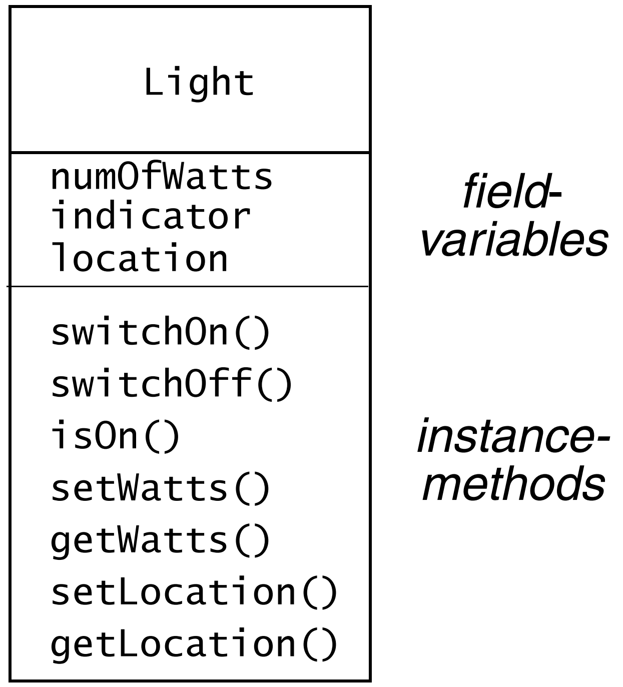
Specification of a class

Class name:

`Light`

Filed variables:	**Methods:**
`numOfWatts`	`switchOn()`
`indicator`	`switchOff()`
`location`	`isOn()`
	`setWatts()`
	`getWatts()`
	`setLocation()`
	`getLocation()`

Graphical notation for a class



Declaration of properties: field variables

```
/**
 * The class Light models the abstraction Light.
 */
class Light {
    int numOfWatts;    // power in watts
    boolean indicator; // off == false, on == true
    String location;   // where the light is located
    ...
}
```

- The class declaration specifies 3 *field declarations* that declare 3 *field variables*.
  - The fields can store information about the power (`numOfWatts`), whether the light is on or off (`indicator`), and where it is located (`location`).

Objects

- A class must be *instantiated* to create an object.
- Instantiation (i.e. creation of objects from classes) consists of:
  - creation of objects using the `new` operator and a *constructor call*, which returns the *reference value* of the newly created object.
  - initialization of the object *state* using a constructor.
- We can declare a *reference* to store the *reference value* of an object.
  - The reference can be used to manipulate the object.

Reference Declaration

Declaration:

```
<className> <objectReference>;
```

- The `<className>` is the name of a class, and the `<objectReference>` is a variable that can store reference values of objects of this class.
- The declaration only creates an *reference* for an object of this class.

  `Light denLight;` creates an reference for an object of the class Light:

  name: `denLight` $\boxed{\texttt{null}}$

  type: `ref(Light)`

- Compare with *arrays* and the `String` class.

Object Creation

Creation:

- The object itself is created using the `new` operator, together with a *constructor call*.
- We can combine the declaration of a reference to store the reference value of an object when the object is created:

  ```
  <className> <objectReference> = new <constructorCall>;
  ```

- The `new` operator creates an object of the specified class.
  - It allocates memory for the object.
  - All the field variables of the object are initialized to *default values.*
  - Before the operator returns the reference value of the object, the *constructor call* specified in the object creation expression is executed, which normally initializes the field variables of the object.
- An object is *alive* as long as the *automatic garbage collector* has not deleted it from the memory.
  - An object is not deleted as long as the program still has a reference to the object.

Constructor Call

- A constructor call has the following form:

  ```
  <className>(<parameterList>);
  ```

- Constructors are declared in the class declaration, and they resemble method declarations.
- The constructor without any parameters is called the *default constructor:*

  ```
  <className>() {...}
  ```

  Example:

  ```
  Light() {...}
  ```

- If a class does not have any constructors, the compiler generates an *implicit default constructor* for the class:

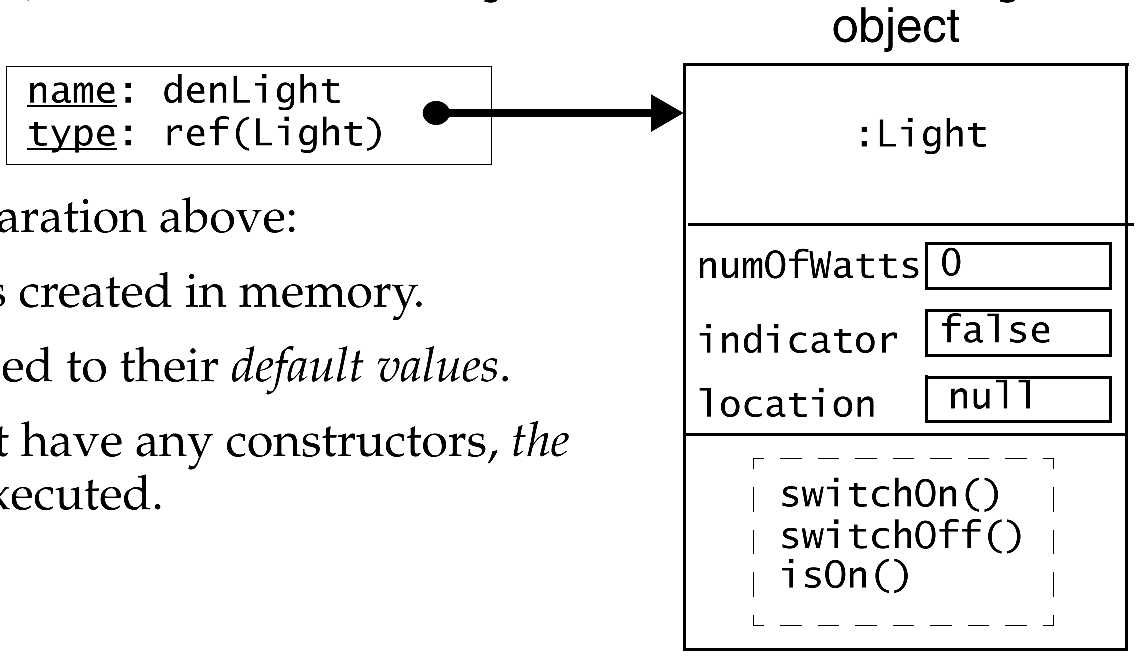
  ```
  <className>() {/*empty body*/}
  ```

  Example:

  ```
  Light() {}
  ```

The implicit default constructor

```
Light denLight = new Light(); // creates an object of the class Light.
```



Steps in the execution of the declaration above:

- An object of the class `Light` is created in memory.
- All field variables are initialized to their *default values*.
- Since the class `Light` does not have any constructors*, the implicit default constructor* is executed.

Default Values for Field Variables

- In Java the field variables of an object are initialized to their default values when the object is created.

Data type of field variables:	**Default value:**
`boolean`	**`false`**
`char`	`'\u0000'`
Integer (`byte`, `short`, `int`, `long`)	`0`
Floating-point (`float`, `double`)	`+0.0f` or `+0.0d`
Reference	`null`

Declaration of behaviour: methods

- Behaviour of objects is defined with the help of *methods*.

Syntax:

```
<accessModifier> <returnType> <methodName>(<formalParameterList>) {

  /* Method body: declarations and statements */


}
```

The class Light: methods

```
/**
 * The class Light models the abstraction Light.
 */
class Light {
    int numOfWatts;    // wattage
    boolean indicator; // off == false, on == true
    String location;   // where the light is located
    /** Default Constructor */
    Light() {
        numOfWatts = 0;
        indicator  = false;
        location   = "X";
    }

    /** Method to turn on the light */
    void switchOn() {
        indicator = true;
        System.out.println("Light in location " + location + " is on.");
    }
    /** Method to turn off the light */
    void switchOff() {
        indicator = false;
        System.out.println("Light in location " + location + " is off.");
    }
    /** Method to find out if the light is on or not. */
    boolean isOn() {
        return indicator;
    }
    /** Method to set the location of the light */
    void setLocation(String loc) {
        location = loc;
    }
    // Other methods ...
}
```

Access Modifiers

- `<accessModifier>` indicates the *visibility* of the method:

`public`	The method is accessible from all other classes: all methods in the class `Light` are `public`.
`private`	The method is only accessible in the class in which it is declared.
None (called *standard, default* or *package visibility*)	The method is only accessible from the classes in the same package.

Return value

- `<returnType>` is the *data type* of the value returned by the method if the method is executed.
  - `void` means that the method does not return any value.
  - The method `switchOff()` does not return any value, while the method `isOn()` returns a `boolean` value.

Signature: Method name and formal parameters

- <*methodName*> is the name of the method: `switchOff` and `isOn` are method names.
- <*formalParameterList*> is data that the method needs in order to do its job.
  - Neither `switchOff` nor `isOn` has any parameters, indicated by `()`.
  - The method `setLocation` has a formal parameter `loc` of type `String`, indicated by `(`**`String`**` loc)`.

- <*methodName*> and <*formalParameterList*> form the *signature* of the method.
  - The method `switchOn` has the signature `switchOn()`.
  - The method `setLocation` has the signature `setLocation(String)`.

Method body and the `return` statement

- The *method body* is a *block* that contains the declarations of *local variables* that the method uses and the *statements* that define actions performed by the method.
  - Neither method body of the `switchOff()` method nor the `isOn()` method contains any declarations. (*Example of local variable declarations will be given later.*)
  - Actions of the method `switchOff()` are to set the field variable `indicator` to `false`, and print the message that the light is on.
  - Action in the method `isOn()` is to return the value of the field variable `indicator` (using the `return` statement).
    - The `return` statement *returns* this value to the *calling* method.
    - If the method `isOn()` did not have a `return` statement, the compiler will issue an error.
    - The *return value* must match the *return type* declared in the *method declaration*.
    - The `return` statement can either return a *value of primitive data type* or a *reference to an object*.
    - The `return` statement alone (without the return value) can be used to terminate the execution of a `void` method.

Local variables and field variables

- Let us extend the class `Light` with a method that computes the cost of having the light on for a certain number of hours.

```
class Light {
    // Field variables
    int numOfWatts;        // wattage
    boolean indicator;     // off == false, on == true
    String location;       // where the light is located
    // ...
    double cost(int numOfHours) {
        // Local variables
        double kWh_price = 0.35;        // cents per kiloWatt hour
        double price = (numOfWatts * numOfHours/ 1000.0)* kWh_price;
        return price;
    }
    // ...
}
```

- The method `cost()` declares *3 local variables*: `kWh_price`, `price` and `numOfHours`.
  - Note! *Parameters are also local variables*.

Difference between field variables and local variables

- Field variables (for example `numOfWatts`) is declared in the class body, while local variables (for example `kWh_price`) is declared in the method body.
- Field variables can be declared with the keyword `private`, while local variables cannot.
- Local variables in a method are not accessible from other methods of the class, i.e local variables can only be used in the method they are declared in.
- Local variables exist as long as the method is executed, while field variables exist as long as the object exists.

```
class Light {
    double cost(int numOfHours) {
        double kWh_price = 0.35;    // cents pr. kiloWatt-hour
        return (numOfWatts * numOfHours/ 1000.0)* kWh_price;
    }
    void setLocation(String loc) {
       // ...
        kWh_price = 0.5;           // Compiler complains that kWh_price is not
                                   // declared.
    }
}
```

Local **block**

- Parameter variables *cannot* be re-declared in a method body.
- A local variable in a block can be redeclared if the blocks are *disjoint*.
- A local variable *already* declared in a block *cannot* be redeclared in a nested block.

```
public static void main(String[] args) {
    String args = ""; // cannot redeclare parameters.
    char digit;
    for (int counter = 0; counter < 10; counter++) {
        switch (digit) {
            case 'a': int i; // OK
            default:  int i; // already declared in this block
        } // switch
        if (true) {
            int i;       // OK
            int digit;   // already declared in the outer block
            int counter; // already declared in the outer block
        } //if
    } // for
    int counter; // OK
} // main
```

Method classification

- *Mutators:*
  - write-operations that change the state of the object.
  - maintain the integrity of the object state.
  - usually declared as `public`.
  - comprise the contract of the class
  - For example, the method `switchOn()` is a mutator.
- *Selectors:*
  - read-operations that have access to the object state, but they do not change the state.
  - usually declared as `public`.
  - comprise the contract of the class.
  - For example, the method `isOn()` is a selector.
- *Help methods:*
  - Operations used by other methods in the class to implement behaviour.
  - usually declared as `private`.
  - are not part of the contract of the class, but the implementation of the class.

Messages: Method Call

Example of a client of the class `Light`:

```
public class Client {
    public static void main(String[] args) {  // calling method
        // ...
        Light denLight = new Light();
        denLight.switchOn(); // message via method call
        // ...
    }
}
```

- `denLight.switchOn()` is a *method call*.
  - `denLight` is the reference that refers to the object that is to receive the message.
  - Dot (’`.`’) separates the reference from the method name.
  - `switchOn` is the name of the method that must be defined in the class of the object.
  - `()` indicates the *actual parameter list* with values that the method can use.
  - A method call on an object starts the execution of the corresponding method in the in the class of the object.
  - The method call `denLight.switchOn()` does not return any value, as the method `switchOn()` is a `void` method.

Example of a client of the class `Light` (cont.)

```
public class Client {
    public static void main(String[] args) {
        Light denLight = new Light();
        boolean lightFlagg = denLight.isOn();
        if (lightFlagg) { System.out.println("denLight is on."); }
        else { System.out.println("denLight is off."); }
        denLight.setLocation("the den");
    }
}
```

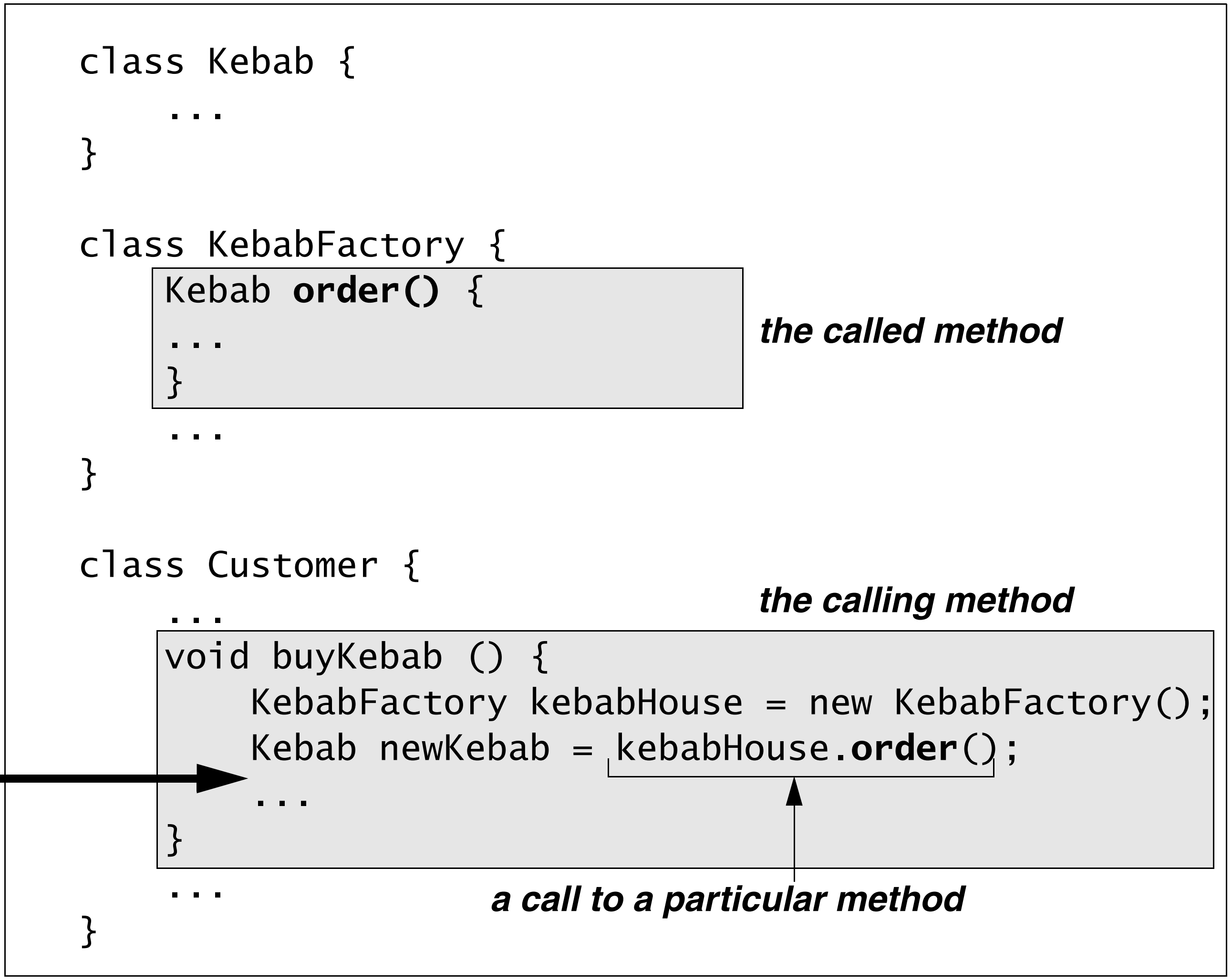
- `denLight.isOn()` is another method call.
  - The method call results in the method `isOn()` declared in the class `Light` to be executed.
  - The method call returns a value of type `boolean`, that can be used as other values/variables of this type.
- `denLight.setLocation("the den")` is a method call with *one actual parameter* `"the den"`.
  - The actual parameter `"the den"` is *matched* with the corresponding formal parameter `(String loc)` in the method declaration of `setLocation()`, and is used to initialize the field variable `location` for the object referred to by the reference `denLight`.

Methods with parameters

Example: Clients orders kebabs.

- `Customers` have no possibility of influencing what kebab they get.
- `KebabFactory` can offer different methods for choice of kebab variants, but this is not the best solution.
- Use of *parameters* allows more *flexible and reusable* methods: methods calls can be made more specific by proving additional information.

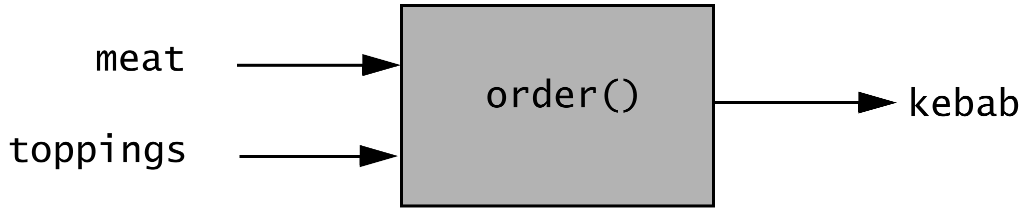
Program execution continues immediately after the call on return from the called method.



```
class Kebab {
    ...
}

class KebabFactory {
    Kebab order() {
    ...
    }
    ...
}

class Customer {
    ...
    void buyKebab () {
        KebabFactory kebabHouse = new KebabFactory();
        Kebab newKebab = kebabHouse.order();
        ...
    }
    ...
}
```

Parameter specification in the method call: Actual parameters

- Instead of writing methods for all possible situations, the calling method sends *specific information* to *generalized methods* via *actual parameters*.



```
class Customer {
    ...
    void buyKebab () {
        KebabFactory kebabHouse = new KebabFactory();

        // Create actual parameters
        Meat chicken = new Meat("chicken");
        Topping onion = new Topping("onion");
        // Order kebab
        Kebab chickenKebab = kebabHouse.order( chicken, onion );
        ...
    }
    ...
}
```

actual parameters

Methods as *black boxes*: client do not need to know about *inner workings* of the box, only how it can be *used*.

Actual parameters specify *actual objects* that can be used in this call.

How does the `KebabFactory` handles this order?

Parameter declaration: Formal parameters

- We must define the method `order()` so that it can accept parameters.

```
class KebabFactory {
    Kebab order ( Meat meatChoice, Topping toppingChoice ) {
        // source code to make a kebab
    }
    ...
}
```

formal parameters

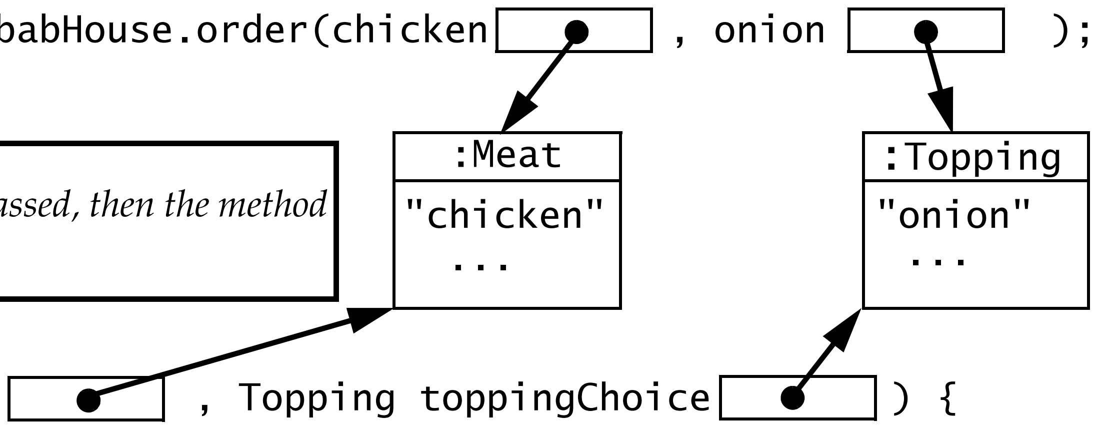
- Formal parameters act as *placeholders*, for example `x` and `y` in the equation $x^2 + y^2 = 1$.
- Formal parameters have no values before they are assigned the *values* of the *actual parameters* that are passed when the method is called – and the values can *vary* from call to call.
- Formelle parameter names need *not* be the same as actual parameter names.
  - They act as local variables of the method.
- Formal parameters specify a *data type* (for example primitive datatypes or classes) as any other variable declaration, but actual parameters do not specify any type.

Parameter Passing: *reference values of objekter*

```
class Customer {
    ...
    void buyKebab () {
        KebabFactory kebabHouse = new KebabFactory();

        // Create parameters
        Meat chicken = new Meat("chicken");
        Topping onion = new Topping("onion");
        // Order kebab
        Kebab chickenKebab = kebabHouse.order(chicken      , onion       );
        ...
    }
    ...
}

class KebabFactory {
    Kebab order(Meat meatChoice       , Topping toppingChoice       ) {
        // source code to make a kebab
    }
    ...
}
```

Formal parameters refer to the same objects as actual parameters, i.e. parameter passing copies reference values if we have objects as actual parameters.

First the reference values are passed, then the method is executed.



The references can be used to change the state of the objects.

Parameter Passing: *values of primitive datatypes* **(I)**

- Formal parameters are assigned values of *primitive datatypes*.

```
class Customer {
    ...
    void buyKebab () {
        KebabFactory kebabHouse = new KebabFactory();
        ...
        // Calculate the price for all the kebabs
        // ordered, and assume same unit price.
        double price = kebabHouse.calculatePrice( 4 , 30.50 );
        ...
    }
    ...
}

class KebabFactory {
    double calculatePrice(int numOfKebabs  4  , double unitPrice  30.5  ) {
        // source code to calculate the price
    }
    ...
}
```

Formal parameters are assigned values of the actual parameters, i.e. parameter passing copies the primitive values if we have formal parameters of primitive datatypes.

First formal parameters are initialized, then the method is executed.

Actual parameter can be an arbitrary expression.

Parameter Passing: *values of primitive datatypes* **(II)**

- An actual parameter can be an expression that evaluates to a *primitive value*.

```
class Customer {
    ...
    void buyKebab () {
        KebabFactory kebabHouse = new KebabFactory();
        ...
        int quantity = 4;
        double itemPrice = 30.5;
       double price = kebabHouse.calculatePrice(quantity  4  ,itemPrice  30.5  );
        ...
    }
    ...
}

class KebabFactory {
    double calculatePrice(int numOfKebabs  4  ,double unitPrice  30.5  ) {
       // source code to calculate the price.
    }
    ...
}
```

First formal parameters are initialized, then the method is executed.

Values of actual parameters are unchanged on return.

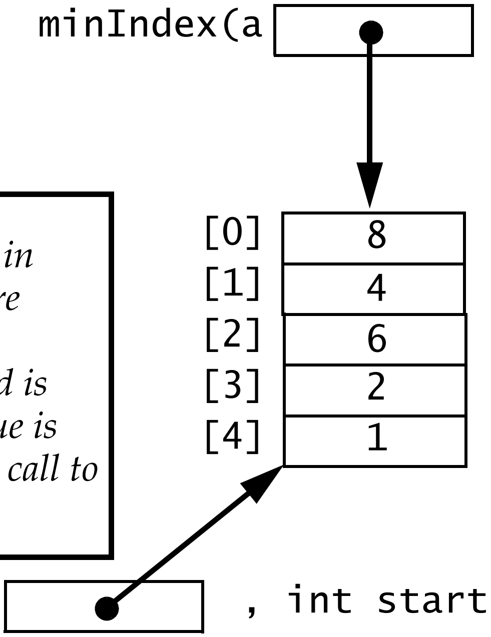
Formal parameters can never change the values of actual parameters.

Parameter Passing: *reference value of arrays*

```
class SelectionSort {
    int a[] = {8,4,6,2,1};
    ...
    void sort() {
        for (int index = 0; index < a.length - 1; ++index)
            swap(a, index,    minIndex(a          , index));
        ...
    }
    ...
```

First the reference value of the array is passed, then the method is executed.

Note that a method call is nested in another method call. Calls that are nested are executed first. In this example, the `minIndex()` *method is executed first and the return value is used as a actual parameter in the call to the* `swap()` *method.*

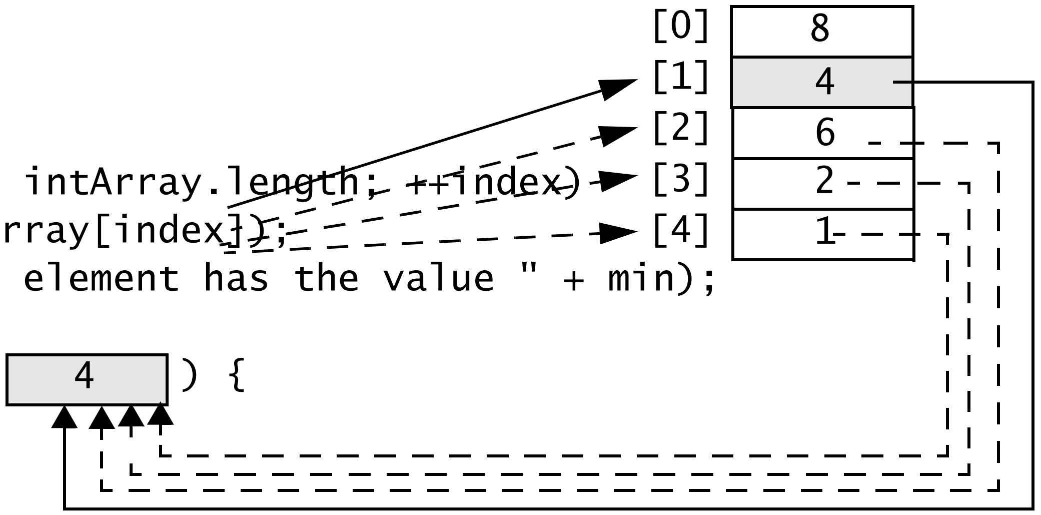


The array reference can be used to read and write element values.

```
    int minIndex(int[] array          , int startIndex) {
    ...
    }
    void swap(int[] array, int i, int j) {
    ...
    }
}
```

Parameter Passing: *array element of primitive types*

```
class FindMin {
    int intArray[] = {8,4,6,2,1};
    void findMin() {
        int min = intArray[0];
        for (int index = 1; index < intArray.length; ++index)
            min = minimum(min, intArray[index]);
        System.out.println("Minimum element has the value " + min);
    }
    int minimum(int i,       int j ) {
        int min = i;
        if (j < i) min = j;
        return min;
    }
}
```

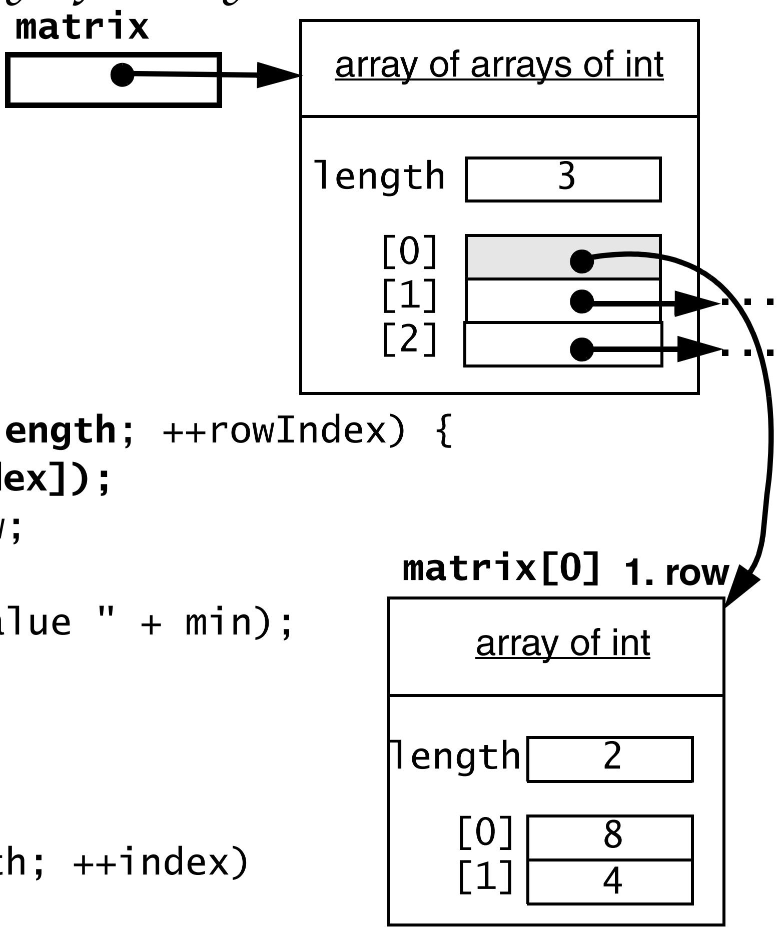


Each element of the `intArray` *is of the primitive data type* `int`*, and the value of each element is passed.*

```
public class MinClient {
    public static void main(String[] args) {
        new FindMin().findMin();
    }
}
```

Parameter Passing: *array of arrays*

```
public class FinnMinMxN {
    int matrix[][] = {{8,4},{6,2,2},{9,4,1,7,1}};
    public static void main(String[] args) {
        new FinnMinMxN().find();
    }

    void find() {
        int min = findMin(matrix[0]);
        for (int rowIndex = 1; rowIndex < matrix.length; ++rowIndex) {
            int minFromRow = findMin(matrix[rowIndex]);
            if (minFromRow < min) min = minFromRow;
        }
        System.out.println("Minimum element has value " + min);
    }

    int findMin(int intArray[]) {
        int min = intArray[0];
        for (int index = 1; index < intArray.length; ++index)
            min = Math.min(min, intArray[index]);
        return min;
    }
}
```



Each element in `matrix` is an array, such that reference value of this array is passed.

Correspondence between formal and actual parameters

- 1-1 correspondence between formal and actual parameters.
  - the order of and the number of actual parameters must match the order of and number of formal parameters in the method definition.
  - Actual parameters must be *type compatible* with formal parameters: *the value of the actual parameter can be assigned to the formal parameter*.
    - if the formal parameter is of a primitive data type, the actual parameter must be of the same primitive data type (or a data type that can explicitly be converted to the formal primitive data type).
    - if the formal parameter is of a reference type, the actual parameter must be an object of the same class (or an object of a subtype of the formal reference type).
  - Compile-time error if order, number and types do not correspond.

Examples with parameter passing

- What is wrong? Logical error in the program!

```
public class SwapClient {
    public static void main(String[] args) {
        int n = 10, m = 5;
        System.out.println("Before swapping: n = " + n + " and m = " + m);
        swap(n, m);
        System.out.println("After swapping: n = " + n + " and m = " + m);
    }
    static void swap(int i, int j) {
        int temp = i;
        i = j;
        j = temp;
    }
}
```

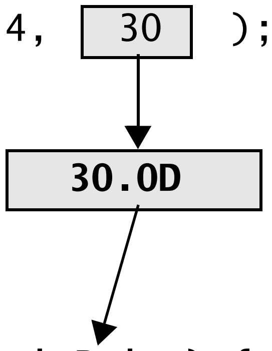
Output:

```
% java swap
Before swapping: n = 10 and m = 5
After swapping: n = 10 and m = 5
```

Implicit conversion during parameter passing

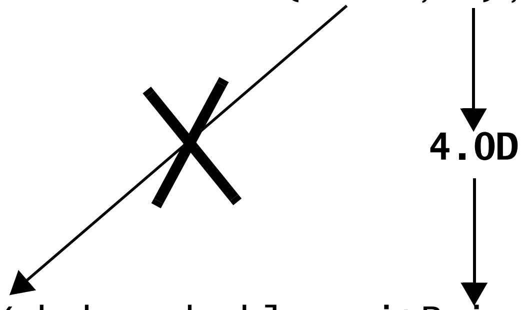
- This happens when an actual parameter has a value which is of a narrower type than that of the formal parameter.

```
class Customer {
    ...
    void buyKebab () {
        KebabFactory kebabHouse = new KebabFactory();
        ...
        // Calculate the price for all kebabs eaten.
        // Assume same price of a kebab.
        double price = kebabHouse.calculatePrice(  4,  30   );

        ...
    }
    ...
}
class KebabFactory {
    double calculatePrice(int numOfKebabs, double unitPrice) {
        // source code to calculate the price
    }
    ...
}
```



- Order/ type compatibility:

```
class Customer {
    ...
    void buyKebab () {
        KebabFactory kebabHouse = new KebabFactory();
        ...
        // Calculate the price for all kebabs eaten.
        // Assume same price of a kebab.
        double price = kebabHouse.calculatePrice(30.50, 4);// order: logical error?

        ...
    }
    ...
}
class KebabFactory {
    double calculatePrice(int numOfKebabs, double unitPrice) {
        // source code to calculate the price
    }
    ...
}
```



- What does this program do?

```
public class RefParameters {
    public static void main(String[] args) {
        String str = "Hold on";
        System.out.println("Before: " + str);   // Before: Hold on
        concat(str);
        System.out.println("After: " + str);    // After: Hold on
    }
    static void concat(String str) {
        System.out.println("In concat: " + str);  // In concat: Hold on
        str = str + " tight";
        System.out.println("Out concat: " + str); // Out concat: Hold on tight
    }
}
```

  – Formal parameters are local variables, and can be used as such variables.
    - They can be *assigned* values in the called method, but that does *not* change the values of the actual parameters.

Summary of Parameter Passing

Data type of the formal parameters:	**Parameter Passing:**
primitive data type	passing of primitive value
reference type	passing of reference value

Syntax of *formal parameters*:

`(<type`$_1$`> <formparam`$_1$`>, <type`$_2$`> <formparam`$_2$`>,…, <type`$_n$`> <formparam`$_n$`>)`

where `<type`$_i$`>` is either a *primitive data type* or a *reference type*, and `<formparam`$_i$`>` is an *identifier* (with or without the array operator `[]`).

Syntax of *actual parameters* (also called *arguments*):

`(<actparam`$_1$`>, <actparam`$_2$`>, ..., <actparam`$_n$`>)`

where `<actparam`$_i$`>` is an *expression* (arithmetic, boolean, strung) or a *reference*.

- The *empty* parameter list is specified as `()`.
- Formal and actual parameters must match with regard to *order*, *number*, and *type*.

The current object: `this`

- When a method is called in an object, how does the method refer to the object itself?

  Use the keyword `this`!
- The keyword `this` is a reference to the *current object* of the method that is being executed.
- The `this` reference is sent as an *implicit* parameter in calls to instance methods, and can be used to refer to all members in the class.
- The `this` reference can be used to refer to field variables that are *shadowed*.

```
class Light {
    // Field variables
    int numOfWatts;
    boolean indicator;
    String location;
    void setValues(int numOfWatts, boolean indicator, String location) {
        this.numOfWatts = numOfWatts;
        this.indicator = indicator;
        this.location = location;
    }
    // Other methods ...
}
```

Example: Use of `this`

```
class GUIWindow {
    GUIButton button;
    GUIWindow() { // A window has a button.
       button = new GUIButton(this);// The current object is sent as a parameter.
        // ...
    }
    void doAction() {
        // ...
    }
}
class GUIButton {
    GUIWindow mainWindow;
    GUIButton(GUIWindow window) { // A button has a reference to a window.
        mainWindow = window;
        // ...
    }
    void informMainWindow() {
         mainWindow.doAction(); // The button can inform the window.
    }
}
```

Static members in a class

- There are cases where *members* should only belong to the class, and are not a part of the objects that are created from the class.

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

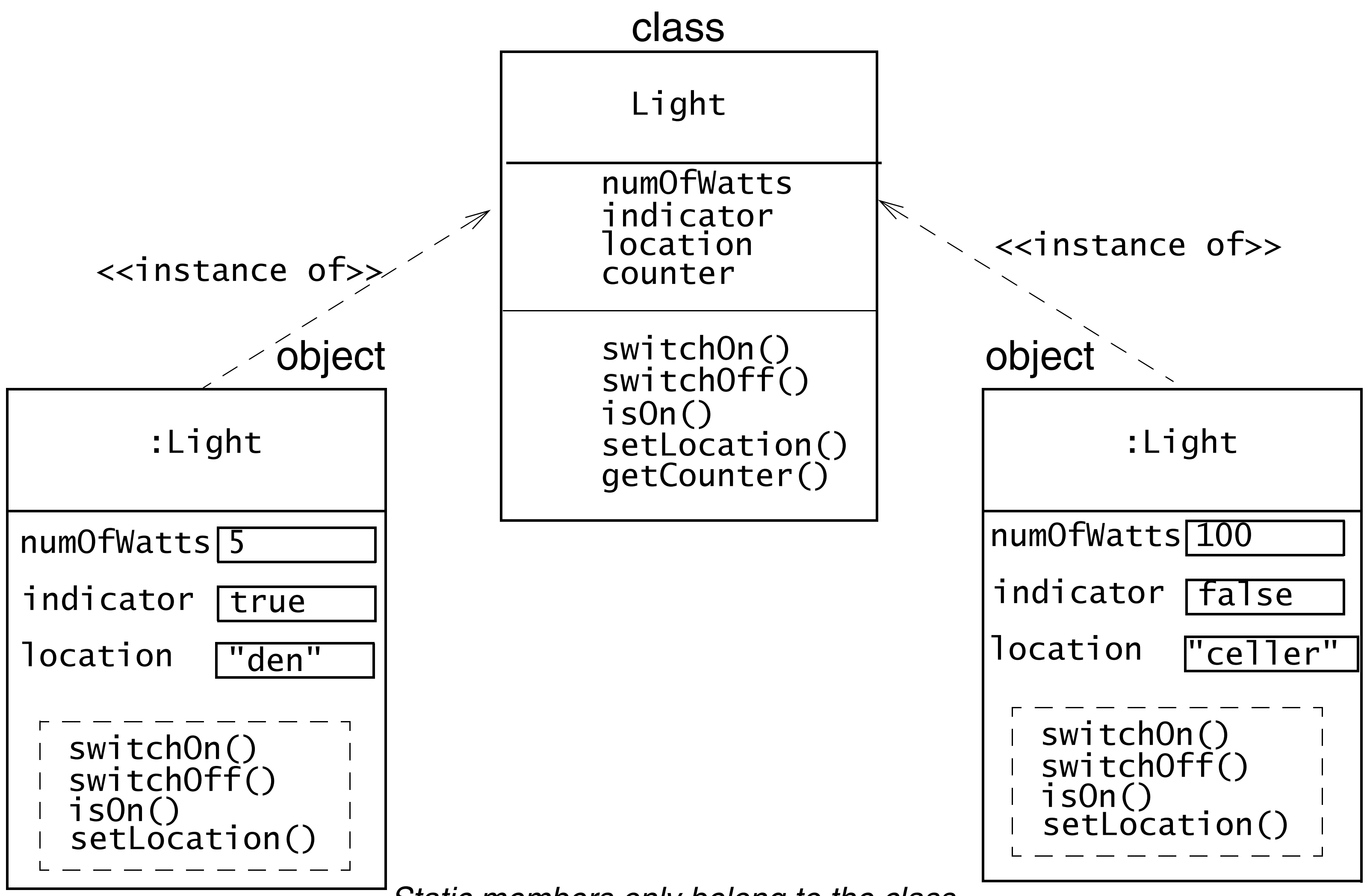
  - We need a *global* counter, but it cannot be in each object that is created.
  - We need a *global* method that can be called to find out how many objects have been created so far.

- Clients can call *static methods* and access *static variables* using the class name or via references of this class.

Terminology:

Static methods	*Methods* that only belong to the class
Static variables	*Variables* that only belong to the class
Static members	Static variables *and* methods

Static members in a class (cont.)



Static members only belong to the class

Remarks on Class Members

- A class declaration can contain following members:
  - *field variables*
  - *instance methods*
  - *static variables*
  - *static methods*
- Field variables are local to an object, i.e. each object has its own copy of all the field variables.
- Static variables are *global* for all objects of the class, and such variables only exist in the class.
- Only one implementation of a method exists in the class, and it used by all objects of the class.
- Static methods *cannot* refer to non-static (i.e. instance) members.
  - Static members of a class exist independent of the objects of the class.
  - The `this` reference is *not* passed when a static method is called.

Example: Static members

```
class Light {
    // Static variable
    static int counter;
    // Field variables
    int numOfWatts;    // wattage
    boolean indicator; // off == false, on == true
    String location;   // where the light is located
    // Static methods
    static void incrCounter() {
        ++counter;
    }
    static int getCounter() {
        return counter;
    }
    ...
}
```

- After we have created an object of the class `Light`, we call the static mutator method `incrCounter()` to increment the value of the static variable `counter`.
- The static selector method `getCounter()` returns the value of the static variable `counter`.

Example: Static members (cont.)

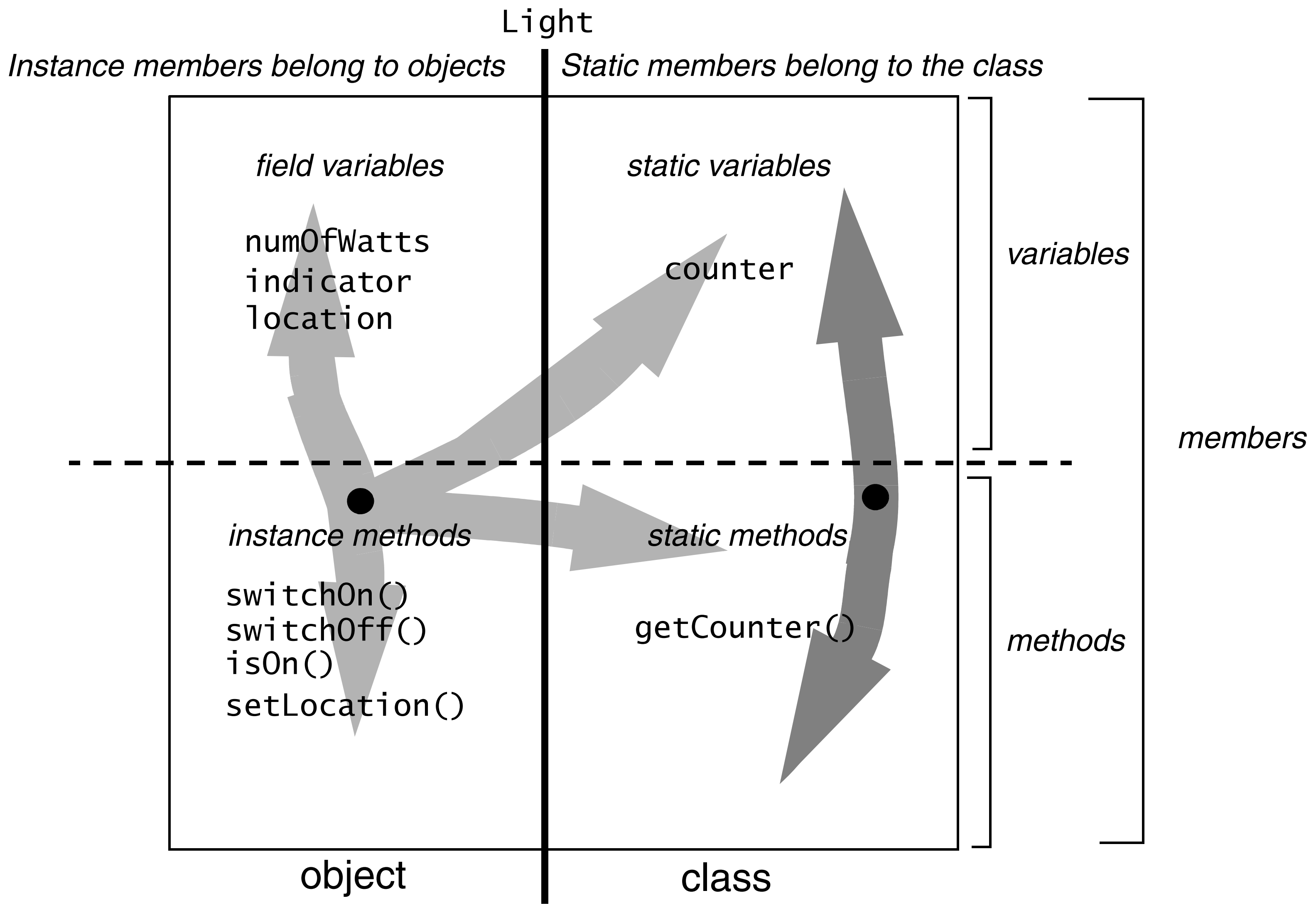
We can always use the class name.

If we have declared references, we can use any reference.

```
public class StaticDemo {
  public static void main(String[] args) {


    System.out.println("Number of Light objects created: " + Light.getCounter() );
    System.out.println("Create a Light object.");
    Light denLight = new Light();
    denLight.incrCounter();
    System.out.println("Number of Light objects created: " + denLight.getCounter() );

    System.out.println("Create an array of Light objects.");
    Light[] lightArray = new Light[10];
    for (int i = 0; i < 10; i++) {
      lightArray[i] = new Light();
      lightArray[i].incrCounter();
    }
    System.out.println("Number of Light objects created: " + Light.getCounter() );
  }
}
```

Accessing members inside the class

Arrows indicate which methods can be accessed inside the class.



Types

- A *type* defines a set of values that can be stores in a variable or that can result from evaluation of an expression.

Primitive datatypes	• primitive values
Reference types:	• reference values of objects
• Classes	
• Arrays	

- Primitive datatypes are already defined in Java.
- Arrays are also already defined in Java.
- Classes are *user-defined* types.

Scope

- *Scope* is where in the source code a variable can be *used directly* with its *simple name*, without indicating where it is declared.

Inside a class:

- An instance method in a class can use *all* members of the class *directly*.
- A static method in a class can use *static* members of the class *directly*.

Inside a block:

- Applies to *local variables* (parameters + variables in a *method body*).
- Scope of a local variable starts from where it is declared and ends where the block in which it is declared ends (*see the figure with local block*).

Life time

- *Life time of a variable declaration* is the period the variables exists in the memory during execution.

Life time for local variables:

- Applies to *all local variables* (parameters + variables in the *method body*).
- Local variables in a block are created as variable declarations are executed in the block.
- Local variables must be initialized explicitly *before* use.
- Local variables cease to exist when program control leaves the block.

Life time for field variables:

- Applies to all *field variables* in an object.
- The field variables are allocated and automatically initialized to default values when an object is created, if no explicit initialization is attempted by the program.
- Field variables exist as long as the object they belong to exists.

Life time for static variables:

- Applies only to *static variables* of a class.
- During *loading* of the class at runtime, the *static variables* are created and initialized only once.
- Static variables exist as long as the class exists.

Program Arguments

- Formal parameter `args` of the method `main()` is an *array of strings* (`String[]`) where each string corresponds to an argument given on the command line.

```
public class Args {

    public static void main(String[] args) {

        for (int i = 0; i < args.length; ++i)
            System.out.println(args[i]);

    } // main
}
```

- Program arguments can be used to tailor the program according to the user's wishes.

Compiling and running the program:

```
> javac Args.java
> java Args green 18pt 24x80
green
18pt
24x80
>
```

Constructors

- A *constructor* has the same name as the class:

  `<className>(<parameterList>) {...}`

- If a class does *not* define a constructor, *the implicit default constructor* is automatically applied when objects of the class are created.
  - A class can explicitly define the default constructor in order to perform any necessary actions.

  `<className>() {...}`

  - A constructor cannot return a value.
- A constructor is executed when an object is created with the `new` operator.
- A constructor is typically be used to:
  - initialize field variables.
  - execute operations that are necessary in order to initialize the object, for example create other objects if necessary.

Constructor Declarations

- The class can define constructors that can be used at object creation time to initialize the object state with values other then default values.

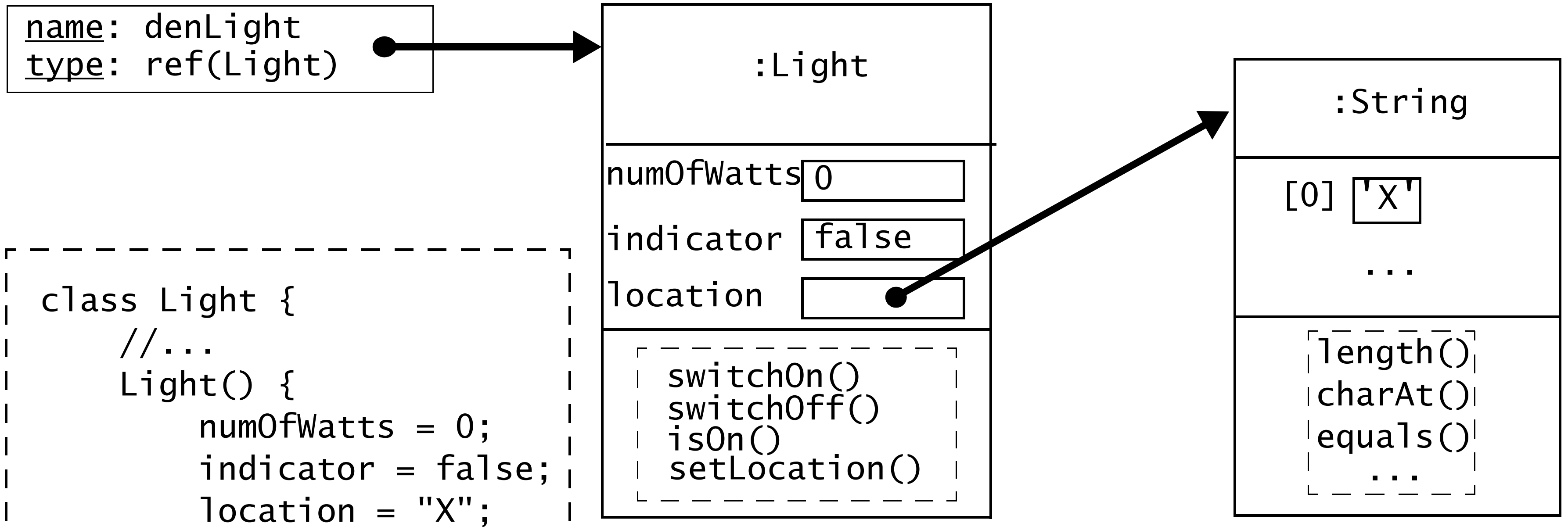
```
class Light {
    // Field variables can be initialized in field variable declarations:
    int     numOfWatts = 0;       // wattage
    boolean indicator  = false;   // off == false, on == true
    String  location   = "";      // where the light is located
    // or the explicit default constructor can be used:
    Light() {
        numOfWatts = 0;
        indicator  = false;
        location   = "X";
    }
    // and/or define constructors which clients can use to initialize
    // field variables with explicit values:
    Light(int watt, boolean ind, String loc) {
        numOfWatts = watt;
        indicator  = ind;
        location   = loc;
    }
}
```

Example of a client for the class `Light`:

```
public class LightClient {
    public static void main(String[] args) {

        Light denLight = new Light(); // Uses the explicit default constructor.

        // We can combine declaration, creation and initialization with
        // explicit values:
        Light cellerLight = new Light(30, true, "celler");
    }
}
```

The explicit default constructor

```
Light denLight = new Light(); // Uses the explicit default constructor.
```

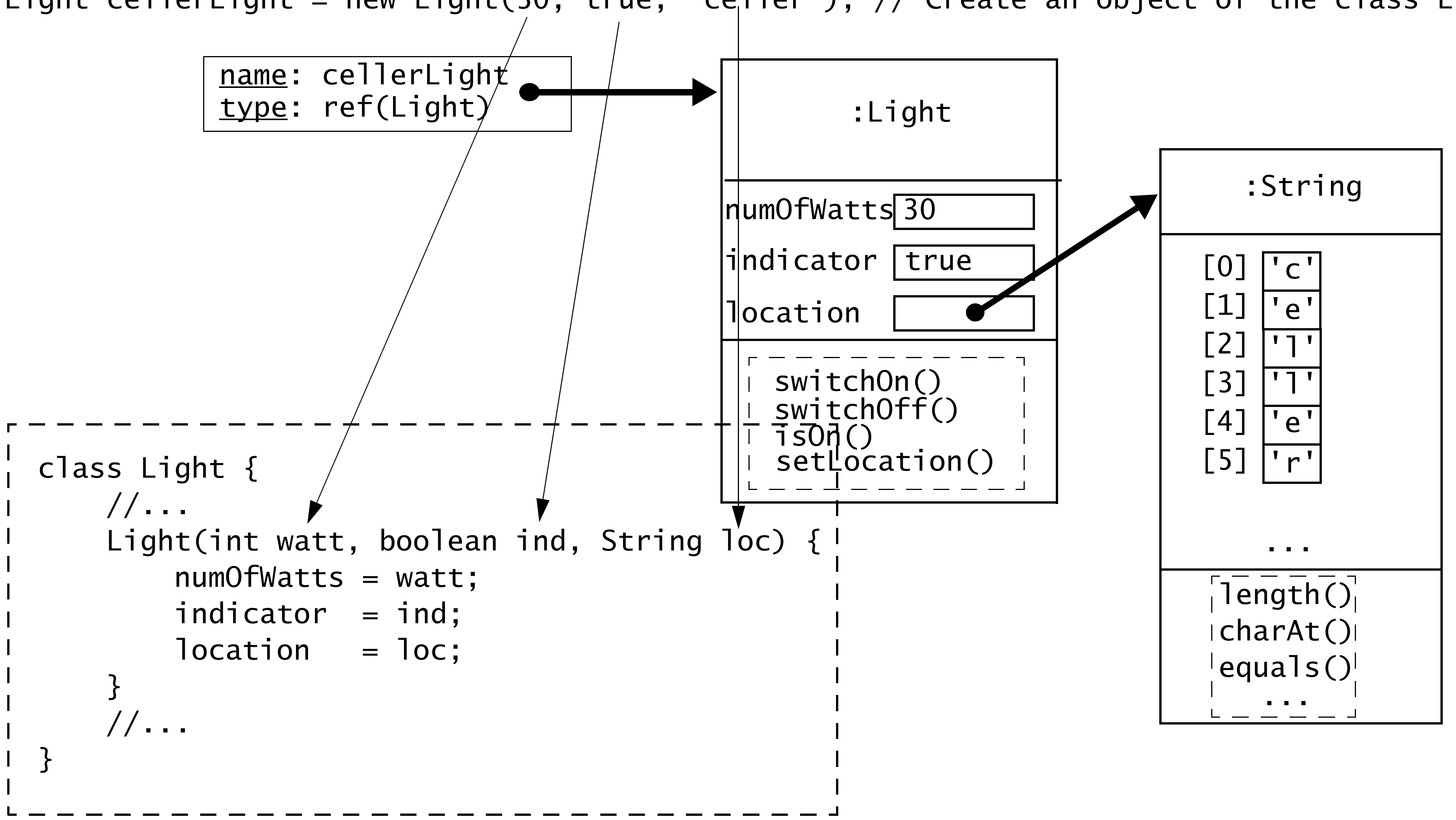


```
class Light {
    //...
    Light() {
        numOfWatts = 0;
        indicator = false;
        location = "X";
    }
    //...
}
```

Use of the default constructor leads to all objects of the class having the same state when they are created.

Non-default Constructors

```
Light cellerLight = new Light(30, true, "celler"); // Create an object of the class Light
```



```
class Light {
    //...
    Light(int watt, boolean ind, String loc) {
        numOfWatts = watt;
        indicator  = ind;
        location   = loc;
    }
    //...
}
```

Constructor Overloading

- We can declare several constructors for the class `Light` that have the same class name.

```
class Light {
    ...
    Light() {                   // 1: The explicit default constructor
        numOfWatts = 0;
        indicator  = false;
        location   = "X";
    }
    Light(int watt, boolean ind) {             // 2: A non-default constructor
        numOfWatts = watt;
        indicator  = ind;
    }
    Light(int watt, boolean ind, String loc) { // 3: A non-default constructor
        numOfWatts = watt;
        indicator  = ind;
        location   = loc;
    }
    ...
}
```

This is called *constructor overloading*. Example shows overloading in *number of parameters*.
The state of the object at creation time will be dependent on the constructor called.

Some remarks on constructors

- If no constructors are defined for the class, *the implicit default constructor* is executed when an object is created using the `new` operator.
- If constructor(s) are defined for the class, *the implicit default constructor* is *not* executed.
- If only *non-default constructor(s)* are defined for the class, it is an error to call *the implicit default constructor.*
- Constructors guarantee that an object will have a *consistent* state when an object is created.

Enumerated types

- An enumerated type defines a fixed number of *enum constants*.

```
enum CivilStatus {
  UNMARRIED, MARRIED, DIVORCED, LIVEIN_PARTNER, REGISTERT_PARTNER,
  SINGLE_PARENT
}
```

  – We must specify *all* enum constants of an enumerated type in the declaration.

- We can declare reference variables of enum types, as for any other reference type:

```
CivilStatus status = CivilStatus.LIVEIN_PARTNER;
```

- We can compare enum constants:

```
if (status == CivilStatus.LIVEIN_PARTNER) {
  System.out.println(status + ": taxed individually.");
} else {
  System.out.println(status + ": see tax regulation rules.");
}
```

- We can compare enum constants in a `switch`-statement:

```
switch(status) { // (1)
  case UNMARRIED:
    System.out.println(status + ": tax bracket 1.");
    break;
  case DIVORCED: case LIVEIN_PARTNER:
    System.out.println(status + ": taxed individually.");
    break;
  case SINGLE_PARENT:
    System.out.println(status + ": tax bracket 2.");
    break;
  default:
    System.out.println(status + ": taxed as a couple.");
}
```

  – When we call the `toString()` method on a enum constant, the name of the enum constant is returned, for example:

```
SINGLE_PARENT: tax bracket 2.
```

- We can create an array of an enumerated type by calling the static method `values()`:

```
CivilStatus[] statusArray = CivilStatus.values();
```

- We can iterate over *an array with enum constants* using the `for(:)` loop:

```
for (CivilStatus civilstatus : statusArray)
  System.out.println(civilstatus);
```

General form for enumerated types

- An enumerated type can declare constructors and other members, as for a class declaration.
  - Constructors cannot be called directly, as we cannot create new objects of an enum type with the `new` operator.
  - Instance members can only be accessed in the specified enum constants.
  - Example: see Figure 7.9 and Program 7.10.