SFWR ENG/COMP SCI 2S03 Principles of Programming

Dr. Ridha Khedri

Department of Computing and Software, McMaster University
Canada L8S 4L7, Hamilton, Ontario

Acknowledgments: Material based on *Java actually: A Comprehensive Primer in Programming* (Chapter 7)

Topics Covered

Object-oriented programming:

- Programs are composed of objects that collaborate to provide the required functionality
- An object belongs to a class that defines the common properties and behaviour of a particular type of objects
- When writing programs, defining and understanding problem-specific classes is essential
- We focus on
  - how to define and use our own classes
  - called user-defined classes
- We introduce a special kind of class (enumerated types)

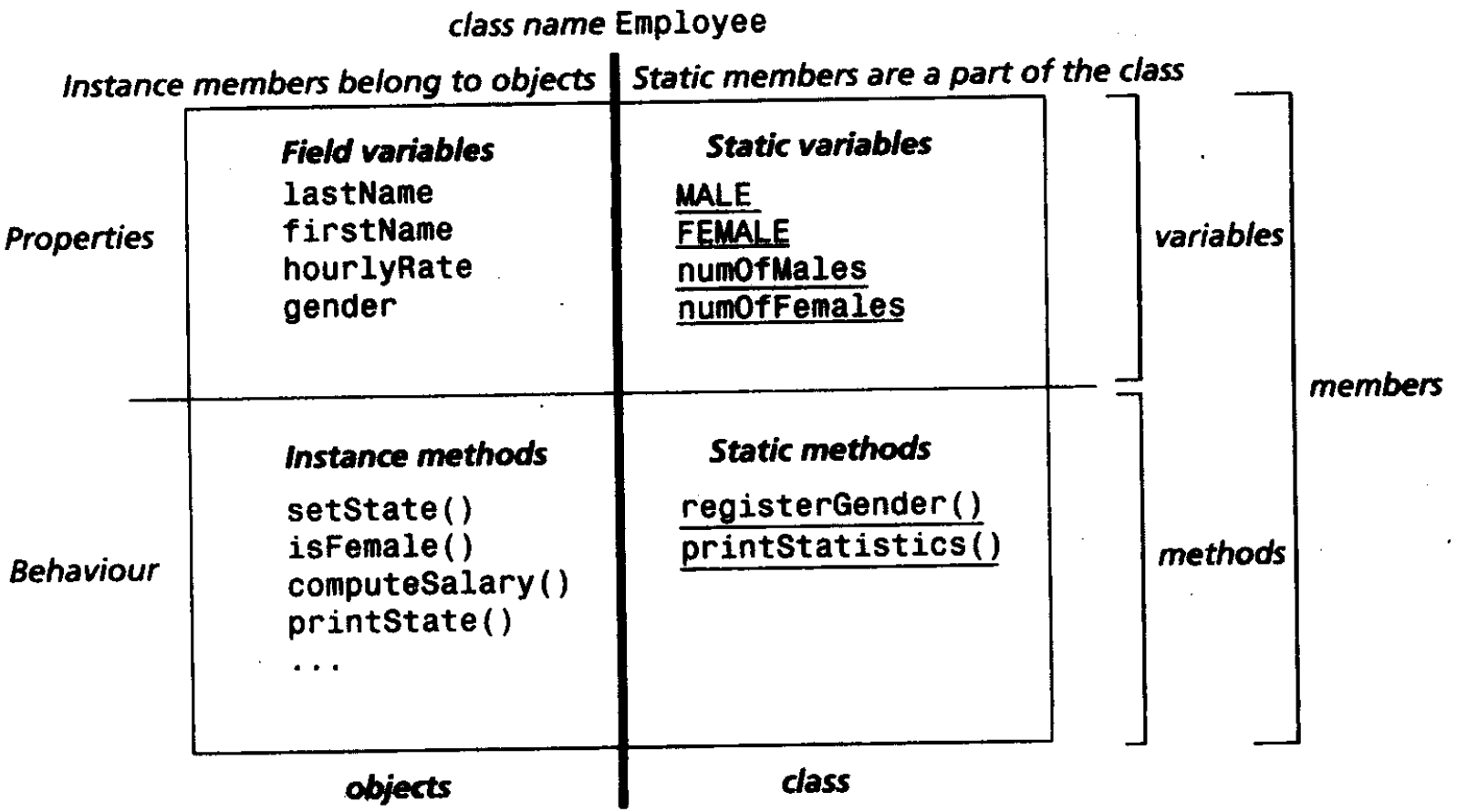
Learning Objectives

- Define your own classes that implement abstractions
- Declare and call methods
- Pass information to methods and methods return values
- Declaration and use of static members of a class
- Passing information to the program via arguments
- Initializing the state of newly-created objects using constructors
- Defining and using enumerated types

Class members

- A class declaration defines
  - the properties
  - the behaviour
- Member declarations:
  - Declarations that are used inside the class
  - Declarations that are used by other classes
- Field variables represent properties
- Instance methods define the behaviour of the objects of the class
- Field variables and instance methods are collectively called instance members

- Member declarations in a class can be declared in any order
- It is common practice to group instance and static members separately
- **Static variables** represent properties **Static methods** define the behaviour of the class
- **NOTE:**
  - Static variables and static methods are collectively called static members
  - They belong to the class, not to the objects of the class


class name Employee
Instance members belong to objects
Static members are a part of the class
Field variables
lastName
firstName
hourlyRate
gender
Static variables
MALE
FEMALE
numOfMales
numOfFemales
Properties
variables
members
Instance methods
setState()
isFemale()
computeSalary()
printState()
...
Static methods
registerGender()
printStatistics()
Behaviour
methods
objects
class

- Field variables define the properties of objects that can be created from the class
- Each object gets its own copy of the field variables
- A field declaration specifies both the field type and the field name

```
class EmployeeV1 {                  // Assume that no constructors are declared.
  // Field variables
  String  firstName;
  String  lastName;
  double  hourlyRate;
  boolean gender;                   // false means male, true means female
  // ...
}
```

- We create objects from a class using the **new** operator

  EmployeeV1 anEmployee = new EmployeeV1();

- "**new**" creates a new object of the EmployeeV1
- The object have room for the fields declared in the class
- All objects created this way will have their fields initialized to default values
- The state of an object comprises all values in the fields
- The state can change over time

anEmployee:EmployeeV1	
lastName	= null
firstName	= null
hourlyRate	= 0.0
gender	= false

```
class EmployeeV2 {                  // Assume that no constructors are declared
  // Field variables with initial values
  String  firstName  = "Joe";
  String  lastName   = "Jones";
  double  hourlyRate = 15.50;
  boolean gender     = false;  // false means male, true means female
  // ...
}
```

EmployeeV2 employeeA = new EmployeeV2( );

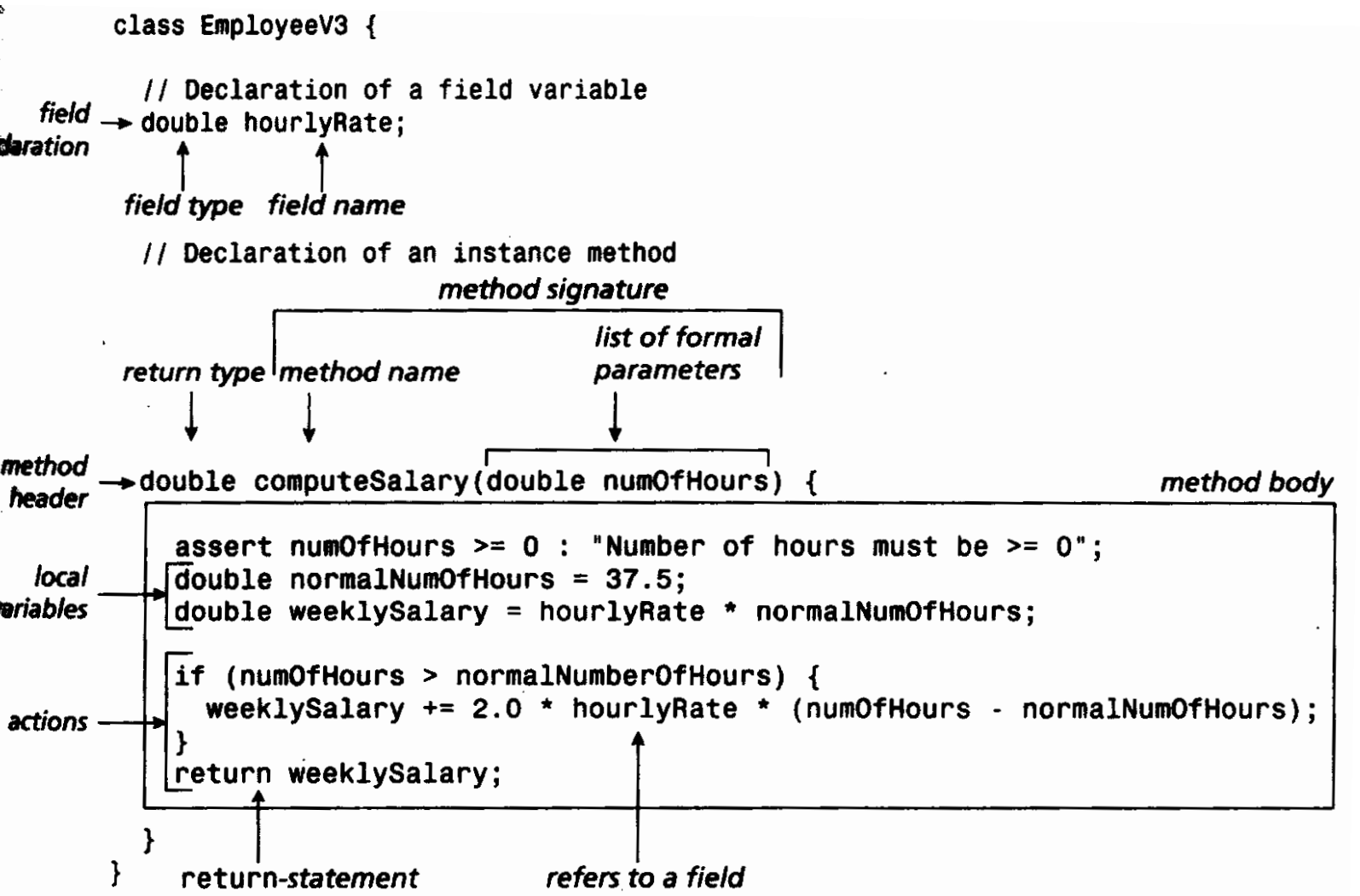
employeeA:EmployeeV2	
lastName	= "Joe"
firstName	= "Jones"
hourlyRate	= 15.50
gender	= false

- We saw that a method declaration comprises
  - a method header
  - method body
- The method header declares
  - the return type
  - the method name
  - the parameter list
- If a method is NOT supposed to return a value, the keyword void should be specified
- A non-void method must specify a return type

- The parameter list indicates what information the method needs to do its job
- The parameter list declares formal parameters for the method
- Each parameter is specified as a variable declaration (name and type)
- The type of a formal parameter can also be a reference type (e.g., a class or an array)
- The parameter list is enclosed in parentheses **( )** **(even if the method has no parameters)**

- The method and the parameter list constitute the signature of the method
- The signature determines which method declaration is chosen for execution by a method call
- Method body comprises variable declarations and actions
- Variable declarations in the method body define the local variables needed

- Local variables are accessible only in the method body
- Formal parameters are also local variables
- Several methods can have the same names for their local variables
- The method body implements the actions
- Local variables and statements can be defined in any order
- **Rule:** A local variable must be declared before it can be used in the method body



```
1  class EmployeeV3 {              // Assume that no constructors are
       declared.
     // Field variables
3    String firstName;
     String lastName;
5    double hourlyRate;
     boolean gender;               // false means male, true means female
7
     // Instance methods
9
     // Assign values to the field variables of an employee.
11   void setState(String fName, String lName,
                   double hRate, boolean genderValue) {
13     firstName = fName;
       lastName = lName;
15     hourlyRate = hRate;
       gender = genderValue;
17   }

19   // Determines whether an employee is female.
     boolean isFemale() { return gender; }
21
     // Computes the salary of an employee, based on the number of hours
23   // worked during the week.
     double computeSalary(double numOfHours) {
25     assert numOfHours >= 0 : "Number of hours must be >= 0";
       double normalNumOfHours = 37.5;
27     double weeklySalary = hourlyRate * normalNumOfHours;
       if (numOfHours > normalNumOfHours) {
29       weeklySalary += 2.0 * hourlyRate * (numOfHours - normalNumOfHours)
               ;
       }
31     return weeklySalary;
     }
33
     // Prints the values in the field variables of an employee.
35   void printState() {
       System.out.print("First name: " + firstName);
37     System.out.print("\tLast name: " + lastName);
       System.out.printf("\tHourly rate: %.2f", hourlyRate);
39     if (isFemale()) {
         System.out.println("\tGender: Female");
41     } else {
         System.out.println("\tGender: Male");
43     }
     }
45 }
```

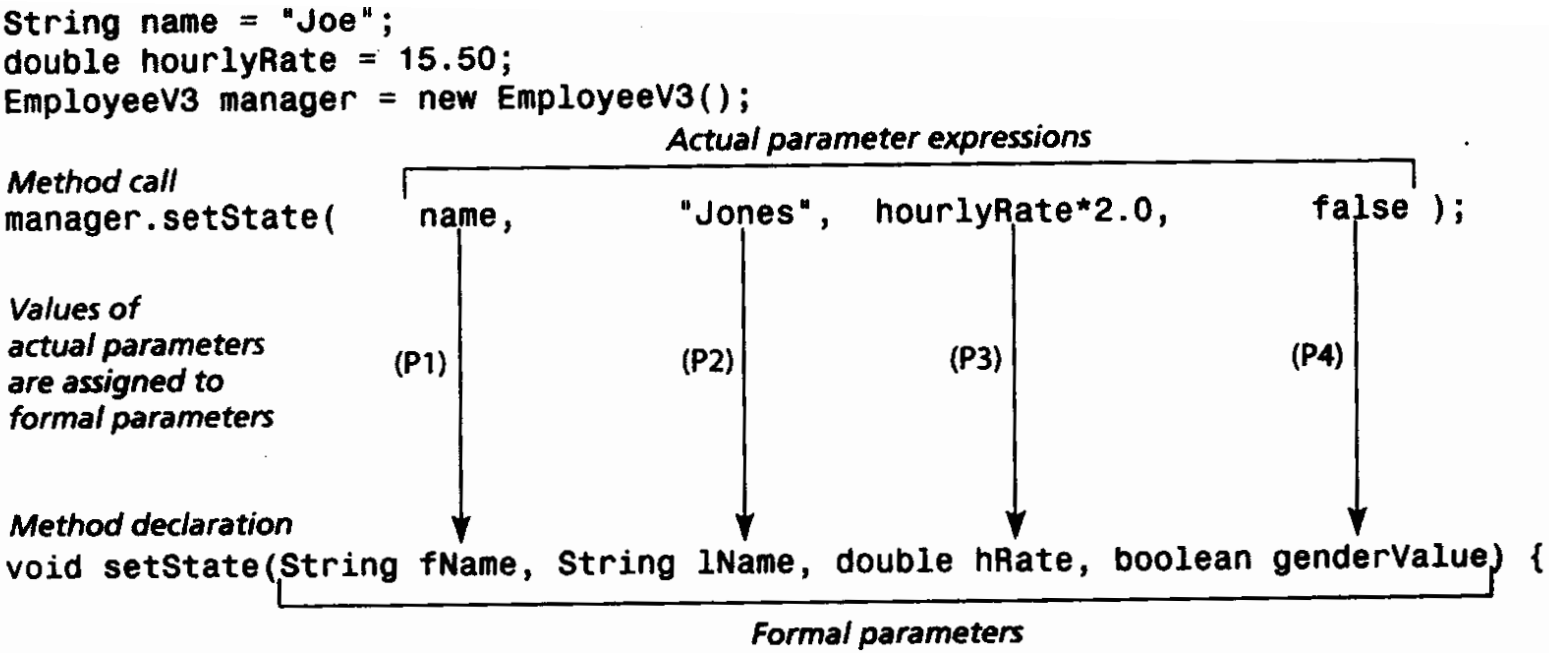
java EmployeeV3

Exception in thread "main" java.lang.NoSuchMethodError:

- A method call is used to execute a method body
- A call specifies
  - the object whose method is called
  - the name of the method
  - any information the method needs to execute its actions (actual parameters, or arguments)
- An actual parameter is an expression
- A formal parameter is specified in the method declaration

- The signature of a method call consists of the method name and the type of the actual parameter expressions **setState(String, String, double, boolean)** Call signature
- The compiler checks that a method exists that corresponds to the method call
- The signature of the method call has to be compatible with the signature of the method declaration
- Examples of method calls that result in compile-time errors:

```
manager.setState(name, hourlyRate*2.0, "Jones", false);// error
manager. setState(name, "Jones", hourlyRate*2.0);  // error
```



After parameter passing:
(P1) `fName` *and* `name` *are aliases of the* `String` *object with the value* `"Joe"`.
(P2) `lName` *refers to the* `String` *object with the value* `"Jones"`.
(P3) `hRate` *has the value* 31.0, *i.e.* `hourlyRate*2.0 = 15.50*2.0`.
(P4) `genderValue` *has the value* `false`.

- An actual parameter expression is evaluated
- Then, its value is assigned to the corresponding formal parameter variable
- Parameter passing in previous Figure is equivalent to the following assignments:

```
String fName = name;          // (P1) reference value of "Joe"
String lName = "Jones";       // (P2) reference value of "Jones"
double hRate = 31.0;          // (P3) primitive value 31.0
boolean genderValue = false;  // (P4) primitive value false
```

Defining Classes

Call-by-value

```
1  // Illustrating parameter passing
   public class Client3A {
3    public static void main(String[] args) {

5      String name = "Joe";
       double hourlyRate = 15.50;
7      EmployeeV3 manager = new EmployeeV3();                    // (1)
       System.out.println("Manager state before call to setState() method")
           ;
9      manager.printState();
       manager.setState(name, "Jones", hourlyRate*2.0, false);  // (2)
11     System.out.println("Manager state after call to setState() method");
       manager.printState();
13     System.out.println();

15     System.out.printf("Manager hourly rate before adjusting: %.2f%n",
                         manager.hourlyRate);                    // (3)
17     adjustHourlyRate(manager.hourlyRate);      // (4) LOGICAL ERROR!
       System.out.printf("Manager hourly rate after adjusting: %.2f%n",
19                       manager.hourlyRate);
       System.out.println();
21
       EmployeeV3 director = new EmployeeV3();                   // (5)
23     System.out.println("Director state before call to copyState() method
           ");
       director.printState();
25     copyState(manager, director);                             // (6)
       System.out.println("Director state after call to copyState() method:
           ");
27     director.printState();
       System.out.println("Manager state after call to copyState() method:"
           );
29     manager.printState();
       System.out.println();
31   }

33
   // CONTINIUED on next slide
```

Defining Classes
Call-by-value

```
//Continues the program on the previous slide ...

  // Method that tries to adjust the hourly rate.
  static void adjustHourlyRate(double hourlyRate) {          // (7)
    hourlyRate = 1.5 * hourlyRate;                           // (8)
    System.out.printf("Adjusted hourlyRate: %.2f%n", hourlyRate);
  }

  // Method that copies the state of one employee over to another
      employee.
  static void copyState(EmployeeV3 fromEmployee,
                        EmployeeV3 toEmployee) {             // (9)

    toEmployee.setState(fromEmployee.firstName,              // (10)
                        fromEmployee.lastName,
                        fromEmployee.hourlyRate,
                        fromEmployee.gender);

    toEmployee = fromEmployee = null;                        // (11)
  }
}
```

Program Output

```
Manager state before call to setState() method
First name: null Last name: null Hourly rate: 0.00
Gender: Male
Manager state after call to setState() method
First name: Joe Last name: Jones Hourly rate: 31.00
Gender: Male

Manager hourly rate before adjusting: 31.00
Adjusted hourlyRate: 46.50
Manager hourly rate after adjusting: 31.00

Director state before call to copyState() method
First name: null Last name: null Hourly rate: 0.00
Gender: Male
Director state after call to copyState() method:
First name: Joe Last name: Jones Hourly rate: 31.00
Gender: Male
Manager state after call to copyState() method:
First name: Joe Last name: Jones Hourly rate: 31.00
Gender: Male
```

Consequences of call-by-value

- Inside the method body, a formal parameter is used like any other local variable
- Changing its value in the method has no effect on the value of the corresponding actual parameter in the method call
- A simple solution for getting the adjusted value from the method is to have the method return the adjusted value

- The state of an object whose reference value is passed to a formal parameter variable can be changed in the method
- The changes will be apparent after return from the method call

```
2   // Method that copies the state of one employee over to another
        employee.
    static void copyState(EmployeeV3 fromEmployee,
4                         EmployeeV3 toEmployee) {            // (9)

6     toEmployee.setState(fromEmployee.firstName,             // (10)
                          fromEmployee.lastName,
8                         fromEmployee.hourlyRate,
                          fromEmployee.gender);
10
      toEmployee = fromEmployee = null;                       // (11)
12  }
```

copyState(manager, director);

- Passing arrays as parameter values does not differ from passing objects
- If the actual parameter evaluates to a reference value of an array, then this reference value is passed
- We use the [ ] notation to declare an array reference as a formal parameter

```
static void computeSalaries(EmployeeV3[ ] employeeArray,
                            double[ ] hoursArray) { ... }
```

```
// Passing arrays
public class Client3B {

  public static void main(String[] args) {
    // (1) Associated arrays with information about employees:
    String[] firstNameArray = { "Tom", "Dick", "Linda" };
    String[] lastNameArray = { "Tanner", "Dickens", "Larsen" };
    double[] hourlyRateArray = { 30.00, 25.50, 15.00 };
    boolean[] genderArray = { false, false, true };

    // (2) Array with employees:
    EmployeeV3[] employeeArray = new EmployeeV3[3];

    // (3) Create all employees:
    for (int i = 0; i < employeeArray.length; i++) {
      employeeArray[i] = new EmployeeV3();
      employeeArray[i].setState(firstNameArray[i], lastNameArray[i],
                                hourlyRateArray[i], genderArray[i]); //
                                    (4)
    }

    // (5) Array with hours worked by each employee:
    double[] hoursArray = { 50.5, 32.8, 66.0 };

    // (6) Compute the salary for all employees:
    computeSalaries(employeeArray, hoursArray);
  }

  // (7) Compute the salary for all employees:
  static void computeSalaries(EmployeeV3[] employees,
                              double[] hours) {
    for (int i = 0; i < employees.length; i++) {
      System.out.printf("Salary for %s %s: %.2f%n",
                        employees[i].firstName,
                        employees[i].lastName,
                        employees[i].computeSalary(hours[i]));
    }
  }
}
```

SFWR ENG/COMP SCI 2S03 Principles of Programming

Dr. R. Khedri

- When we call an instance method of an object, we say that the method is invoked on the object
- The object whose method is invoked becomes the current object
- Inside the method, the current object can be referred to by the keyword this

Example 1

manager. setState (name , "Jones', hourlyRate*2.0, false);

```
  // Assign values to the field variables of an employee.
2 void setState(String fName, String lName,
                double hRate, boolean genderValue) {
4   firstName = fName;
    lastName = lName;
6   hourlyRate = hRate;
    gender = genderValue;
8 }
```

Could be written as follows:

```
void setState(String fName, String lName,
              double hRate, boolean genderValue) {
  this.firstName  = fName;
  this.lastName   = lName;
  this.hourlyRate = hRate;
  this.gender     = genderValue;
}
```

```
void setState(String firstName, String lastName,
              double hRate, boolean genderValue) {
  firstName  = firstName;  // (1)
  lastName   = lastName;   // (2)
  hourlyRate = hRate;
  gender     = genderValue;
}
```

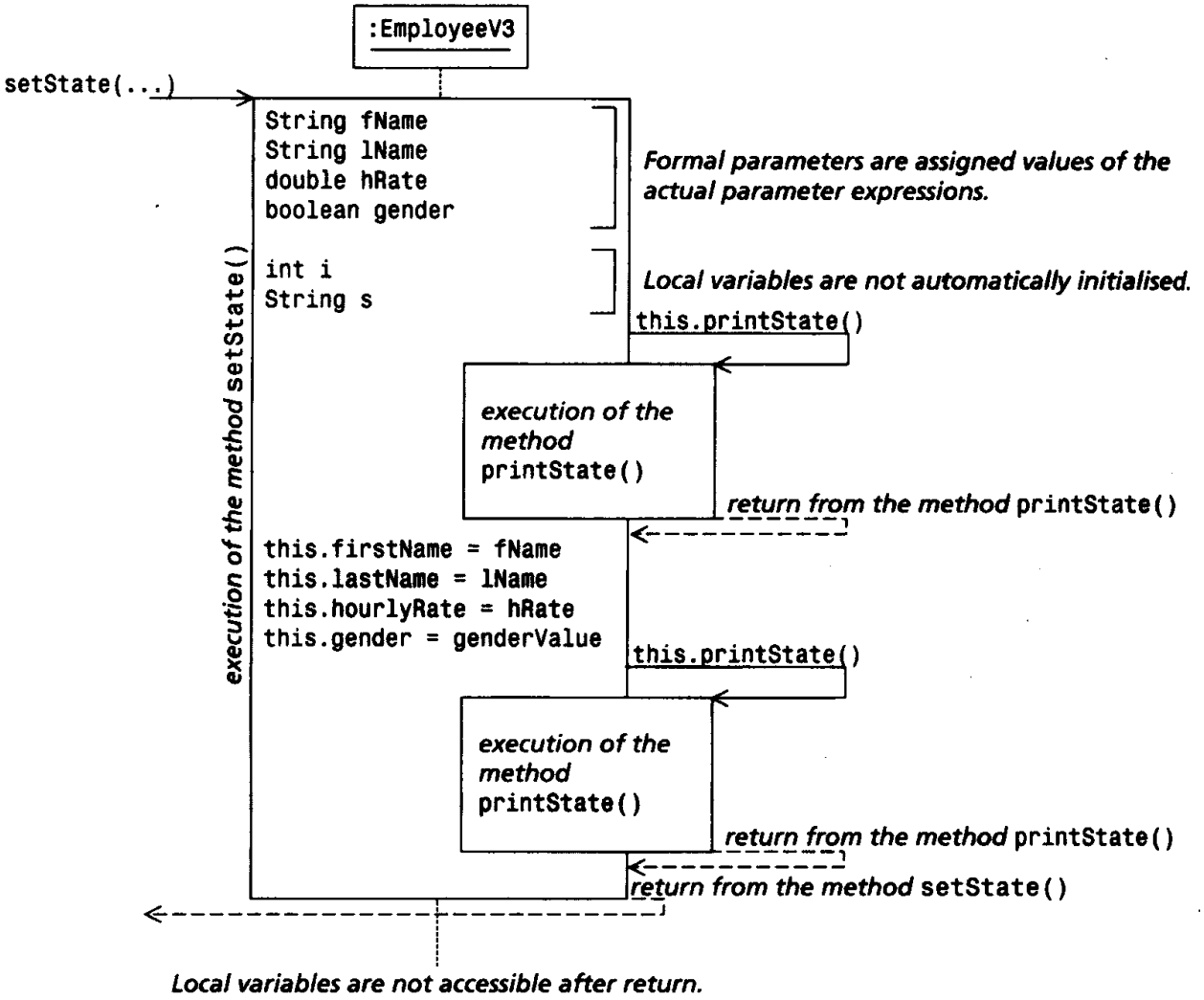
```
void setState(String firstName, String lastName,
              double hRate, boolean genderValue) {
  this.firstName  = firstName;  // (1')
  this.lastName   = lastName;   // (2')
  this.hourlyRate = hRate;
  this.gender     = genderValue;
}
```

We can use the this reference to distinguish the field variable from the local variable when they have the same (names shadowed by a local var.)

- When we invoke a method on an object, the method can in turn invoke other methods

```
void setState(String firstName, String lastName,
              double hRate, boolean genderValue) {
  // Some superfluous local variables
  int i;
  String s;

  this.printState();              // Print state before
  this.firstName  = firstName;
  this.lastName   = lastName;
  this.hourlyRate = hRate;
  this.gender     = genderValue;
  this.printState();              // Print state after
}
```

Method execution and the return statement



Method execution and the return statement

The return statement comes in two forms:

- **return;**
  - Method execution stops and control returns
  - This form can be used when the method does not return a value
  - In setState( ) method, no need to add return as the last statement (execution of the method will end anyway)
- **return <expression>;**
  - The return statement is required for methods that return a value
  - The value of this expression is returned (method stops)
  - The method must explicitly specify the return type
  - The return value must be of this return type

A method with two exits:

```
double computeSalary(double numOfHours) {
  assert numOfHours >= 0 : "Number of hours must be >= 0";
  double normalNumberOfHours = 37.5;
  double weeklySalary = hourlyRate * normalNumberOfHours;
  if (numOfHours <= normalNumberOfHours) {
    return weeklySalary;                                          // (1)
  }
  return weeklySalary +
         2.0 * hourlyRate * (numOfHours - normalNumberOfHours);   // (2)
}
```

We can rewrite as follows:

```
double computeSalary(double numOfHours) {
  assert numOfHours >= 0 : "Number of hours must be >= 0";
  double normalNumberOfHours = 37.5;
  double weeklySalary = hourlyRate * normalNumberOfHours;
  if (numOfHours > normalNumberOfHours) {
    weeklySalary += 2.0 * hourlyRate * (numOfHours - normalNumberOfHours);
  }
  return weeklySalary;
}
```

Defining Classes
Passing information using arrays


```
import java.util.Scanner;
public class ArrayMaker {

  public static void main(String[] args) {
    // (1) Read first names:
    String[] firstNameArray = new String[3];
    System.out.println("Read first names");
    fillStringArray(firstNameArray);                    // (1a)
    printStrArray(firstNameArray);

    // (2) Read last names:
    System.out.println("Read last names");
    String[] lastNameArray = createStringArray();       // (2a)
    printStrArray(lastNameArray);
  }

  // (3) Reference value of the array to be filled is passed
  //     as formal parameter:
  static void fillStringArray(String[] strArray) {      // (3a)
    Scanner keyboard = new Scanner(System.in);
    for (int i = 0; i < strArray.length; i++) {
      System.out.print("Next: ");
      strArray[i] = keyboard.nextLine();
    }
  }

// Continues on the next slide ...
```

Passing information using arrays


```
1  //....... Continues from previous slide
2
3
4    // (4) The method creates and fills an array of strings.
5    //     The reference value of this array is returned by the method:
6    static String[] createStringArray() {                          // (4a)
7      Scanner keyboard = new Scanner(System.in);
8      System.out.print("Enter the number of items to read: ");
9      int size = keyboard.nextInt();
10     String[] strArray = new String[size];
11     keyboard.nextLine();                          // Clear any input first
12     for (int i = 0; i < strArray.length; i++) {
13       System.out.print("Next: ");
14       strArray[i] = keyboard.nextLine();
15     }
16     return strArray;
17   }
18
19   // (5) Prints the strings in an array to the terminal window:
20   static void printStrArray(String[] strArray) {
21     for (String str : strArray) {
22       System.out.println(str);
23     }
24   }
25 }
```

Program Output

```
Read first names
Next: Tom
Next: Dick
Next: Linda
Tom
Dick
Linda
Read last names
Enter the number of items to read: 3
Next: Tanner
Next: Dickens
Next: Larsen
Tanner
Dickens
Larsen
```

- Objects that are no longer in use are removed by JVM
- It is always the JVM that decides when such a clean up should take place
- This clean-up process is called automatic garbage collection
- If a method creates an object and returns its reference value, the reference value of the object can be used after the method returns **(object kept)**
- If the reference value of an object is only stored in a local variable, the object will not be accessible after return from the method **(candidate for garbage)**

Static members of a class

- Static members specify properties and behaviour of the class

- They are NOT a part of any object of class
- **Example:**
  How can we keep track of the number of male and female employee objects that have been created?
  - We can define two counters that are incremented
  - These counters will exist in every employee object
  - Each object will have two counters
  - So, which counters should we use to do the book keeping?
  - **One solution is to maintain the counters as static Variables for the class only**
- We use the keyword static in the declaration of static members

Static members of a class


```
1  class EmployeeV4 {            // Assume that no constructors are declared
       .

3    // (1) Static variables:
     static final boolean MALE = false;
5    static final boolean FEMALE = true;
     static final double NORMAL_WORKWEEK = 37.5;
7    static int numOfFemales;
     static int numOfMales;
9
     // (2) Static methods:
11   // Register an employee's gender by updating the relevant counter.
     static void registerGender(boolean gender) {
13     if (gender == FEMALE) {
         ++numOfFemales;
15     } else {
         ++numOfMales;
17     }
     }
19   // Print statistics about the number of males and females registered.
     static void printStatistics() {
21     System.out.println("Number of females registered: " +
                          EmployeeV4.numOfFemales);                    //
                              (3)
23     System.out.println("Number of males registered: " +
                          EmployeeV4.numOfMales);                      //
                              (4)
25   }

27   // Rest of the specification is the same as in class EmployeeV3

29  // ...See EmployeeV3 program

31   }
}
```

- A static member can be accessed using the notation **className.memberName**
- We can use the member name to refer to static members in the same class inside any method (cond.: the name is not shadowed by a local variable)
- Inside an instance method, we can also use this.staticMember to uniquely identify the member in the class

Accessing static members

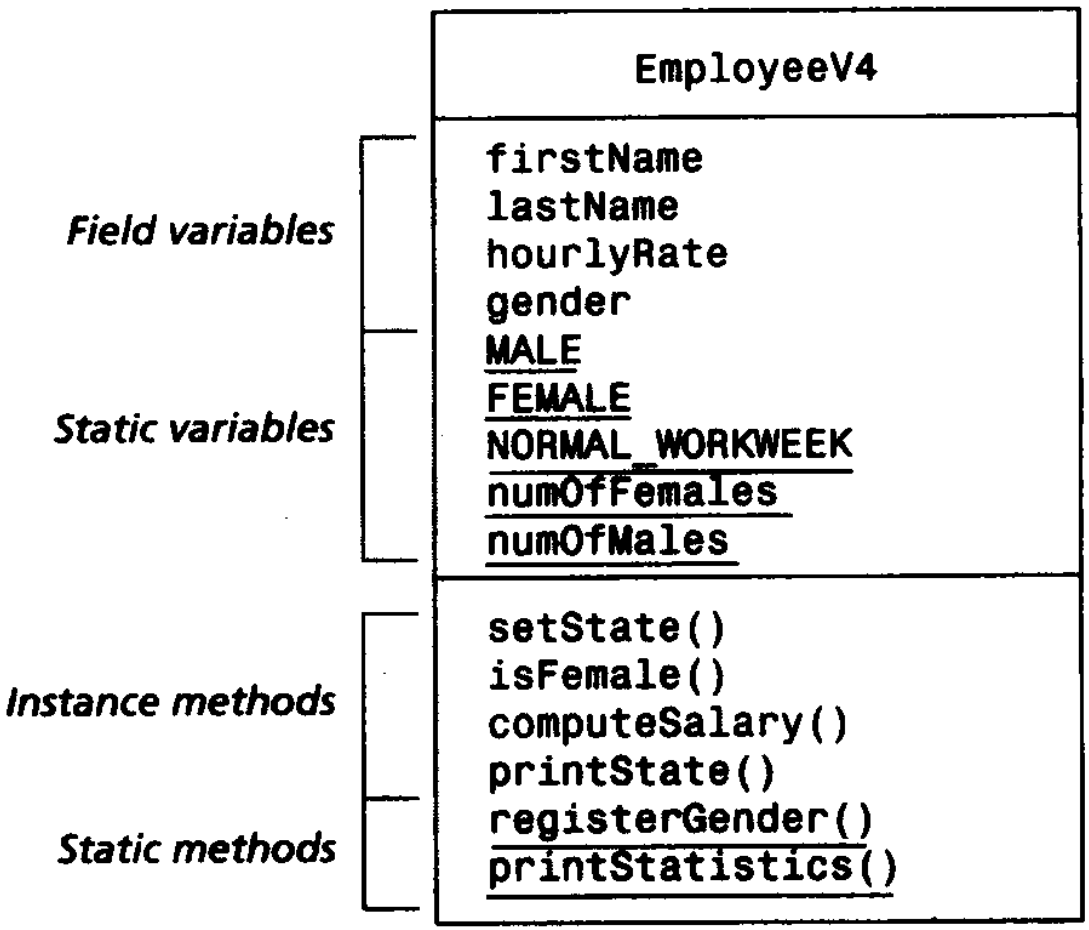
```
// Accessing static members
public class Client4A {

  public static void main(String[] args) {
    // (5) Print information in class EmployeeV4 before any objects
    //     are created:
    System.out.println("Print information in class EmployeeV4:");
    System.out.println("Females registered: " +
                       EmployeeV4.numOfFemales);          // (6) class
                           name
    System.out.println("Males registered: " +
                       EmployeeV4.numOfMales);            // (7) class
                           name

    // (8) Create a male employee.
    EmployeeV4 coffeeboy = new EmployeeV4();
    coffeeboy.setState("Tim", "Turner", 30.00, EmployeeV4.MALE);
    coffeeboy.registerGender(EmployeeV4.MALE);            // (9)
         referanse
    System.out.println("Print information in class EmployeeV4:");
    coffeeboy.printStatistics();                          // (10)
         referanse

    // (11)  Create a female employee.
    EmployeeV4 receptionist = new EmployeeV4();
    receptionist.setState("Amy", "Archer", 20.50, EmployeeV4.FEMALE);
    EmployeeV4.registerGender(EmployeeV4.FEMALE);
    System.out.println("Print information in class EmployeeV4:");
    EmployeeV4.printStatistics();
  }
}
```

Program Output

```
Print information in class EmployeeV4:
Females registered: 0
Males registered: 0
Print information in class EmployeeV4:
Number of females registered: 0
Number of males registered: 1
Print information in class EmployeeV4:
Number of females registered: 1
Number of males registered: 1
```

- **ATTENTION:** No **this** reference for static members
  - Static methods have no **this** reference
  - They are a part of the class, and not a part of the current object
- **Initializing static variables**
  - They are initialized automatically before the class is used
  - One can initialize them with initial values

UML diagram showing all members in a class



The main( ) method and program argmts

- A Java application must have a primary class that defines a method main

  public static void main(String[] args)

- Execution always starts in the main( ) method
- The program terminates when all the actions in the main( ) method have been executed
- The signature of the main( ) method shows that it has an array of strings as a formal parameter
- Program arguments specified on the command line are stored in an array of strings

The main( ) method and program argmts

- In the program, the main () method can obtain these program arguments from the array
- For that, it uses the formal parameter variable args
- If there are no program arguments, the parameter variable args will refer to an array of strings with zero elements
- **Example:**

> **java Client4B Mona Lisa 20.5 true**

The main( ) method and program argmts


```
1  // Using program arguments
   public class Client4B {
3
     public static void main(String[] args) {
5
       // (1) Check that all information about an employee is given
7      //     on the command line:
       if (args.length != 4) {
9        return;
       }
11
     // (2) Print the array args:
13     System.out.println("Program arguments:");
       for (String arg : args) {
15       System.out.println(arg);
       }
17
       // (3) Assign information from the array args to local variables:
19     String firstName = args[0];
       String lastName = args[1];
21     double hourlyRate = Double.parseDouble(args[2]); // (4) Floating-
           point
       boolean gender;
23     if (args[3].equals("true")) {                    // (5) Boolean
           value
         gender = EmployeeV4.FEMALE;
25     } else {
         gender = EmployeeV4.MALE;
27     }

29     // (6) Create an employee, and print its state:
       EmployeeV4 decorator = new EmployeeV4();
31     decorator.setState(firstName, lastName, hourlyRate, gender);
       System.out.println("Information about an employee:");
33     decorator.printState();
       System.out.printf("Salary: %.2f%n", decorator.computeSalary(40.0));
35   }
   }
```

Program Output

```
Program arguments:
Mona
Lisa
20.5
true
Information about an employee:
First name: Mona Last name: Lisa Hourly rate: 20.50
Gender: Female
Salary: 871.25
```

- Constructors have a special role in a class declaration
- The main purpose of a constructor is to set the initial state of the current
- **Explicit default Constructor**: A class can choose to declare an explicit default constructor

```
class EmployeeV5 {
  // Explicit default constructor
  EmployeeV5() {                                    // (3) No parameters
    firstName  = "Joe";
    lastName   = "Jones";
    hourlyRate = 15.50;
    gender     = MALE;
  }
  // Rest of the specification is the same as in class EmployeeV4
}
```

- The constructor call

  EmployeeV5 cook = new EmployeeV5( );

  results in executing the explicit default constructor

- A constructor always has the same name as the class
- A constructor cannot return a value
- The constructor body can contain declarations and actions, similar to an instance method body
- A constructor can use the this reference to refer to the current object

Implicit default Constructor

- If the class EmployeeV5 does not declare a constructor, the compiler will generate a constructor for the class

  EmployeeV5( ) { ... }

- Note the use of the class name and the absence of parameters in the constructor header
- The constructor body has no actions to initialize the state of the current object
- The constructor declaration resembles a method declaration, but it is not a method

EmployeeV5( ) { ... }

- It is called the implicit default constructor for the class EmployeeV5
- The implicit default constructor is not always adequate (field variables initialized to their default values)
- THEREFORE, a class should usually provide explicit constructors to set the initial state of an object

- A class can declare constructors with formal parameters
- Constructors with parameters are called non-default constructors
- The values of the actual parameter expressions in the constructor call are used to initialize the state of the object
- If a class declares any constructor, the implicit default constructor cannot be applied

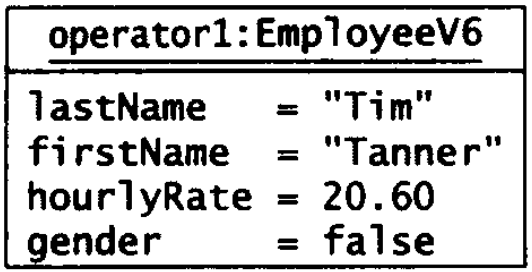
```
class EmployeeV6 {

  // Only non-default Constructor
  EmployeeV6(String fName, String lName, double hRate, boolean gender) {
    this.firstName = fName;            // field access via this reference
    this.lastName = lName;
    this.hourlyRate = hRate;
    this.gender = gender;
    this.registerGender(gender);       // call to static method
  }

  // Static variables
  final static boolean MALE = false;
  final static boolean FEMALE = true;
  final static double NORMAL_WORKWEEK = 37.5;
  static int numOfFemales;
  static int numOfMales;

  // Field variables
  String firstName;
  String lastName;
  double hourlyRate;
  boolean gender;

  // Instance methods

  // Determines whether an employee is a female.
  boolean isFemale() { return (gender == FEMALE); }

  // Computes the salary of an employee, based on the number of hours
  // worked during the week.
  double computeSalary(double numOfHours) {
    assert numOfHours >= 0 : "Number of hours must be >= 0";
    double weeklySalary = hourlyRate * this.NORMAL_WORKWEEK;
    if (numOfHours > EmployeeV6.NORMAL_WORKWEEK) {
      weeklySalary += 2.0 * hourlyRate * (numOfHours - NORMAL_WORKWEEK);
    }
    return weeklySalary;
  }

  // Prints the values in the field variables of an employee.
  void printState() {
    System.out.print("First name: " + firstName);
    System.out.print("\tLast name: " + lastName);
    System.out.printf("\tHourly rate: %.2f", hourlyRate);
    if (isFemale()) {
      System.out.println("\tGender: Female");
    } else {
      System.out.println("\tGender: Male");
    }
  }

  // Static methods

  // Register an employee's gender by updating the relevant counter.
  static void registerGender(boolean gender) {
    if (gender == FEMALE) {
      ++numOfFemales;
    } else {
      ++numOfMales;
    }
  }
  // Print the number of females and males registered.
  static void printStatistics() {
    System.out.println("Number of females registered: " + numOfFemales);
```

```
// Using constructors
public class Client6 {

  public static void main(String[] args) {

    // EmployeeV6 clerk = new EmployeeV6();      // (1) Compile time error!
                                                 // No default constructor.
    // Print information in class EmployeeV6
    System.out.println("Print information in class EmployeeV6:");
    EmployeeV6.printStatistics();
    System.out.println();

    // Create an employee, and print its information
    EmployeeV6 operator1 = new EmployeeV6("Tim", "Turner", 30.00,
                                          EmployeeV6.MALE);          // (2)
    printEmployeeInfo(operator1, 40.0);
    System.out.println();

    // Create a new employee, and print its information
    EmployeeV6 operator2 = new EmployeeV6("Amy", "Archer", 20.50,
                                          EmployeeV6.FEMALE);        // (3)
    printEmployeeInfo(operator2, 50.0);
  }

  static void printEmployeeInfo(EmployeeV6 employee,
                                double numOfHours) {
    System.out.println("Printing information about an employee:");
    employee.printState();
    System.out.printf("Salary: %.2f%n",
                      employee.computeSalary(numOfHours));
    System.out.println("Print information in class EmployeeV6:");
    EmployeeV6.printStatistics();
  }
}
```

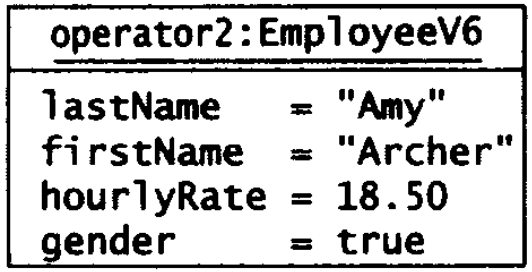
Program Output

```
Print information in class EmployeeV6:
Number of females registered: 0
Number of males registered: 0

Printing information about an employee:
First name: Tim Last name: Turner Hourly rate: 30.00
Gender: Male
Salary: 1275.00
Print information in class EmployeeV6:
Number of females registered: 0
Number of males registered: 1

Printing information about an employee:
First name: Amy Last name: Archer Hourly rate: 20.50
Gender: Female
Salary: 1281.25
Print information in class EmployeeV6:
Number of females registered: 1
Number of males registered: 1
```

```
EmployeeV6 operator1 = new EmployeeV6("Tim", "Tanner", 20.60, EmployeeV6.MALE);
EmployeeV6 operator2 = new EmployeeV6("Amy", "Archer", 18.50, EmployeeV6.FEMALE);
```



(a)



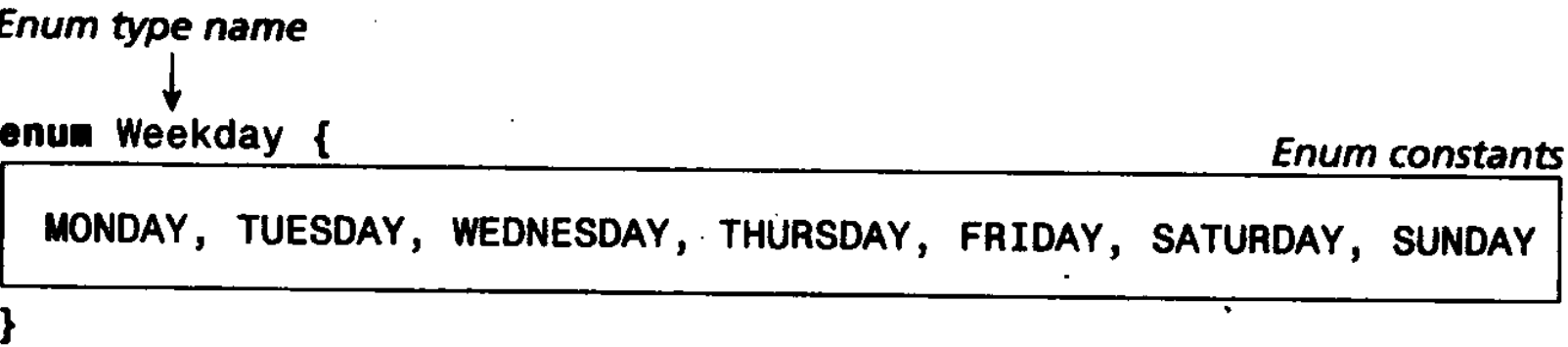
(b)

- A class can declare several constructors
- Each constructor call will result in the execution of the constructor that has a signature compatible with the constructor call
- The initial state of each object is dependent on which constructor was called when the object was created

```
class EmployeeV7 {

  static final double STANDARD_HOURLY_RATE = 15.50;

  // Constructors
  EmployeeV7() {                                                    // (1)
    firstName  = "Joe";
    lastName   = "Jones";
    hourlyRate = STANDARD_HOURLY_RATE;
    gender     = MALE;
    registerGender(MALE);
  }

  EmployeeV7(String fName, String lName, boolean gender) {          // (2)
    this.firstName  = fName;
    this.lastName   = lName;
    this.hourlyRate = STANDARD_HOURLY_RATE;
    this.gender     = gender;
    this.registerGender(gender);
  }

  EmployeeV7(String fName, String lName,
             double hRate, boolean gender) {                        // (3)
    this.firstName  = fName;
    this.lastName   = lName;
    this.hourlyRate = hRate;
    this.gender     = gender;
    this.registerGender(gender);
  }

  // Rest of the specification is the same as in class EmployeeV7

// .... to be continued on next Slide
```

```
1
   // ..............  Continues the previous slide
3
   // Static variables
5  final static boolean MALE = false;
   final static boolean FEMALE = true;
7  final static double NORMAL_WORKWEEK = 37.5;
   static int numOfFemales;
9  static int numOfMales;

11 // Field variables
   String firstName;
13 String lastName;
   double hourlyRate;
15 boolean gender;

17 // Instance methods

19 // Determines whether an employee is a female.
   boolean isFemale() { return (gender == FEMALE); }
21
   // Computes the salary of an employee, based on the number of hours
23 // worked during the week.
   double computeSalary(double numOfHours) {
25   assert numOfHours >= 0 : "Number of hours must be >= 0";
     double weeklySalary = hourlyRate * this.NORMAL_WORKWEEK;
27   if (numOfHours > EmployeeV7.NORMAL_WORKWEEK) {
       weeklySalary += 2.0 * hourlyRate * (numOfHours - NORMAL_WORKWEEK);
29   }
     return weeklySalary;
31 }

33 // Prints the values in the field variables of an employee.
   void printState() {
35   System.out.print("First name: " + firstName);
     System.out.print("\tLast name: " + lastName);
37   System.out.printf("\tHourly rate: %.2f", hourlyRate);
     if (isFemale()) {
39     System.out.println("\tGender: Female");
     } else {
41     System.out.println("\tGender: Male");
     }
43 }

45 // Static methods
   // Register an employee's gender by updating the relevant counter.
47 static void registerGender(boolean gender) {
     if (gender == FEMALE) {
49     ++numOfFemales;
     } else {
51     ++numOfMales;
     }
53 }

55 // Print the number of females and males registered.
   static void printStatistics() {
57   System.out.println("Number of females registered: " + numOfFemales);
     System.out.println("Number of males registered: " + numOfMales);
59 }
}
```

- An enumerated type (also called **enum** for short) defines a fixed number of enum constants
- An enumerated constant is a unique name that refers to a particular object
- We use the keyword **enum** to indicate that the declaration is an enumerated type
- The enum constants are specified in a list in the block that comprises the body of the enumerated type
- These objects are created automatically only once during program execution



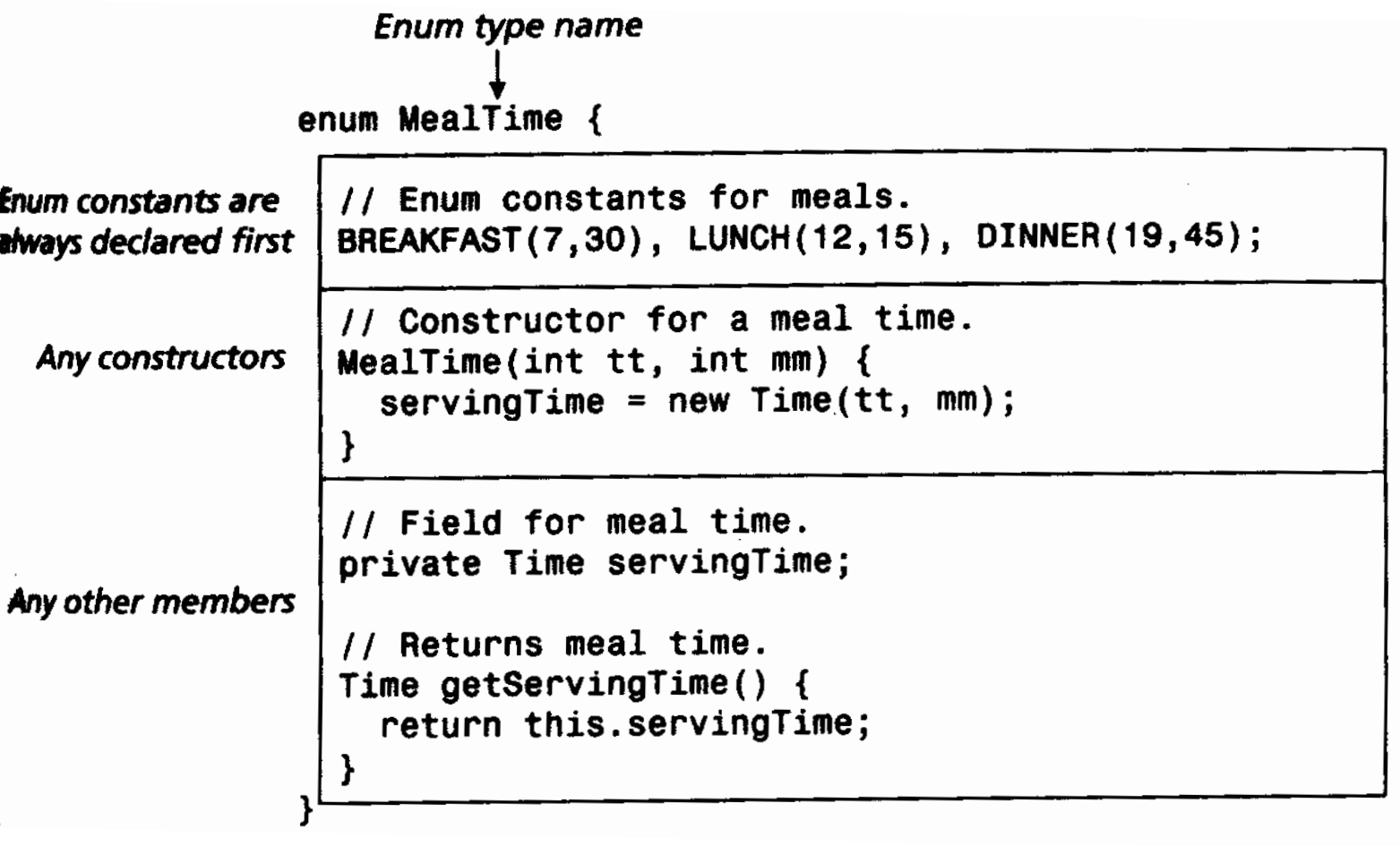
```
switch(day) {                                    // (1)
  case SATURDAY: case SUNDAY:
    System.out.println("The day is " + day + ", it must be weekend.");
    break;
  default:
    System.out.println(day + " is a working day.");
}
```

Method	Description
`String toString()`	Returns the string representation of the current enum constant, i.e. the name of the current constant.
`static enumTypeName[] values()`	Returns an array with the enum constants that are declared in the enum type that has the *enumTypeName*. The order of the constants in the array is the same as the order in the enum declaration.

```
// An enum client
public class Weekdays {
  public static void main(String[] args) {

    Weekday lateOpeningDay = Weekday.THURSDAY;  // Reference of enum
        type

    // Method toString() applied implicitly
    System.out.println(lateOpeningDay);         // Prints THURSDAY.
    System.out.println(Weekday.SUNDAY);         // Prints SUNDAY.

    // Testing for equality
    assert(lateOpeningDay != Weekday.SUNDAY);   // true
    assert(lateOpeningDay == Weekday.THURSDAY); // true

    // Iterate over days of the week:
    System.out.println("Days of the week:");
    Weekday[] daysArray = Weekday.values();
    for (Weekday day : daysArray) {
      switch(day) {                             // (1)
        case SATURDAY: case SUNDAY:
          System.out.println("The day is " + day +
                             ", it must be weekend.");
          break;
        default:
          System.out.println(day + " is a working day.");
      }
    }
  }
}
```

Program Output

```
THURSDAY
SUNDAY
Days of the week:
MONDAY is a working day.
TUESDAY is a working day.
WEDNESDAY is a working day.
THURSDAY is a working day.
FRIDAY is a working day.
The day is SATURDAY, it must be weekend.
The day is SUNDAY, it must be weekend.
```



- In this way, it is possible to define properties and behaviour of enum constants in an enumerated type
- Constructors cannot be called directly
- A constructor is called implicitly when an object representing an enum constant is created automatically

MealTime.java

```
1  enum MealTime {
2
3    // Enum constants for meals.
4    BREAKFAST(7,30), LUNCH(12,15), DINNER(19,45);
5
6    // Constructor for a meal time.
7    MealTime(int tt, int mm) {
8      servingTime = new Time(tt, mm);
9    }
10
11   // Field for meal time.
12   private Time servingTime;
13
14   // Returns meal time.
15   Time getServingTime() {
16     return this.servingTime;
17   }
18 }
```

Time.java

```
// Time is given as hours (0-23) and minutes (0-59).
class Time {

  // Fields for the time.
  int hours;
  int minutes;

  // Constructor
  Time(int hours, int minutes) {
    assert (0 <= hours && hours <= 23 &&
            0 <= minutes && minutes <= 59) :
            "Invalid hours and/or minutes";
    this.hours = hours;
    this.minutes = minutes;
  }

  // String representation of the time, TT:MM
  public String toString() {
    return String.format("%02d:%02d", hours, minutes);
  }
}
```

MealService.java

```
1  // Using enums
   public class MealService {
3    public static void main(String[] args) {

5      // (1) Create an array of meals:
       MealTime[] meals = MealTime.values();
7
       // (2) Print meal times:
9      for (MealTime meal : meals) {
         System.out.println(meal + " is served at " + meal.getServingTime()
               );
11     }
     }
13 }
```

Program Output

```
BREAKFAST is served at 07:30
LUNCH is served at 12:15
DINNER is served at 19:45
```

- We have declared an enumerated type as a top-level declaration in its own separate source file
- Other clients can access enum constants by using the class name and the constant name
- An enumerated type can also be declared as a member in a class declaration
- It makes sense to do this if the use of the enum constants is localized to a single class

```
// Enum type as member in a class
public class Weekdays {
  // Enum type as member in a class.
  enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
  }
  public static void main(String[] args) {
    // Same as before.
  }
}
```