JAVA™
PROGRAMMING

Chapter 3: Using Methods, Classes, and Objects

Objectives

- Learn about method calls and placement

- Identify the parts of a method

- Add parameters to methods

- Create methods that return values

- Learn about classes and objects

- Create a class

- Create instance methods in a class

Objectives (cont'd.)

- Declare objects and use their methods

- Create constructors

- Appreciate classes as data types

Understanding Method Calls and Placement

- **Method**
  - A program module
  - Contains a series of statements
  - Carries out a task

- Execute a method
  - **Invoke** or **call** from another method

- **Calling method** (**client method**)
  - Makes a **method call**

- **Called method**
  - Invoked by a calling method

Understanding Method Calls and Placement (cont'd.)

```java
public class First
{
    public static void main(String[] args)
    {
        displayAddress();
        System.out.println("First Java application");
    }
}
```

Figure 3-2   The First class with a call to the displayAddress() method

Understanding Method Calls and Placement (cont'd.)

- `main()` method executes automatically
- Other methods are called as needed

Understanding Method Calls and Placement (cont'd.)

```java
public class First
{
    // You can place additional methods here, before main()
    public static void main(String[] args)
    {
        displayAddress();
        System.out.println("First Java application");
    }
    // You can place additional methods here, after main()
}
```

Figure 3-3   Placement of methods within a class

Understanding Method Construction

- A method must include:
  - **Method header**
    - Also called a **declaration**
  - **Method body**
    - Between a pair of curly braces
    - Contains the statements that carry out the work
    - Also called **implementation**
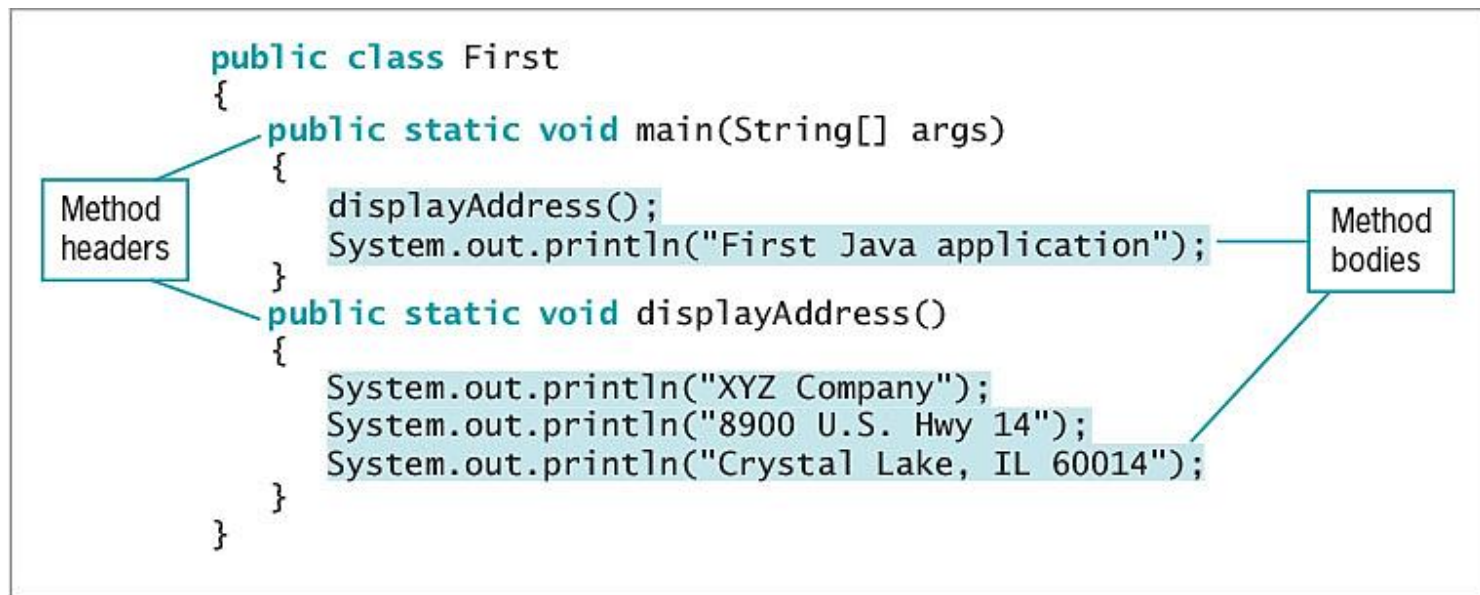
Understanding Method Construction (cont'd.)

```
public class First
{
    public static void main(String[] args)
    {
        displayAddress();
        System.out.println("First Java application");
    }
    public static void displayAddress()
    {
        System.out.println("XYZ Company");
        System.out.println("8900 U.S. Hwy 14");
        System.out.println("Crystal Lake, IL 60014");
    }
}
```

Method headers

Method bodies

Figure 3-6 The headers and bodies of the methods in the First class

Understanding Method Construction (cont'd.)

- The method header contains:
  - Optional access specifiers
  - A return type
  - An identifier
  - Parentheses
    - Might contain data to be sent to the method
- Place the entire method within the class that will use it
  - Not within any other method

Access Specifiers

- Can be `public`, `private`, `protected`, or `package`
- `public` access allows use by any other class
- Also called access modifiers
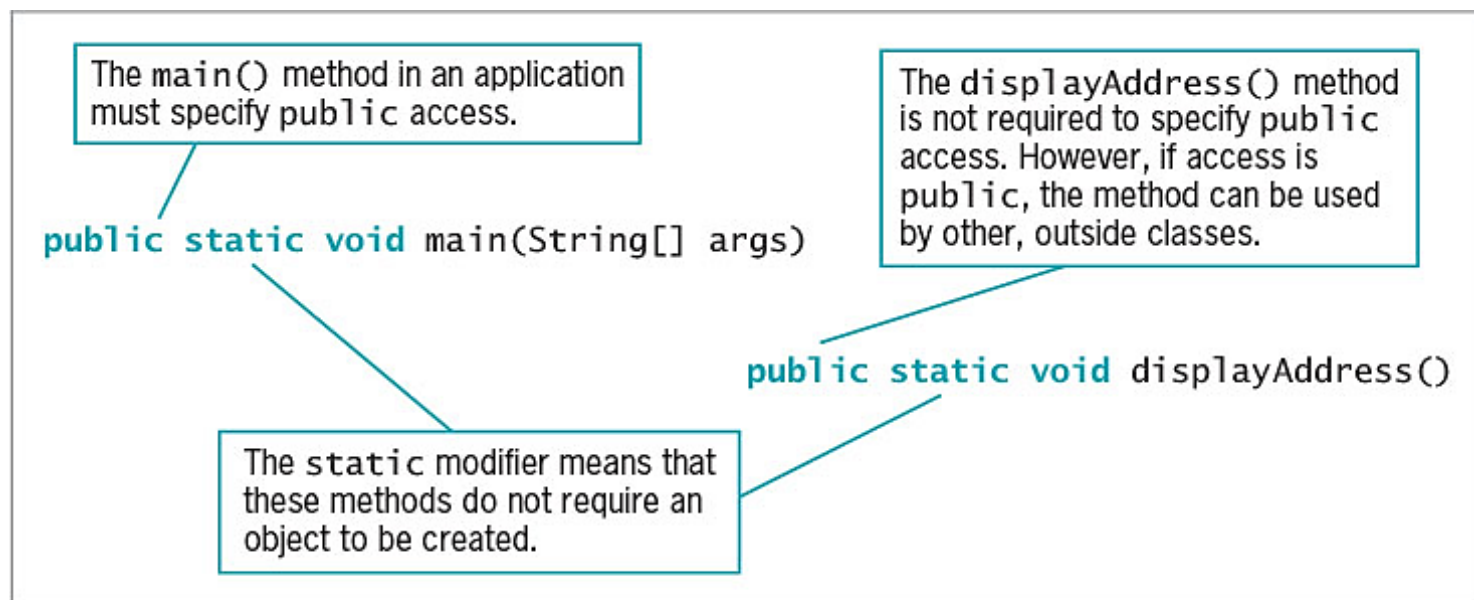- Methods most commonly use `public` access

Access Specifiers (cont'd.)

Figure 3-7 Access specifiers for two methods

Return Type

- Describes the type of data the method sends back to the calling method

- If no data is returned to the method, the return value is `void`

Return Type (cont'd.)

The main() method in an application must have a void return type.

public static void main(String[] args)

public static void displayAddress()

The displayAddress() method does not send any information back to the method that calls it, so its return type is void. Later in this chapter you will write methods with other return types.

Figure 3-8 Return types for two methods

Method Name

- Can be any legal identifier
  - Must be one word
  - No embedded spaces
  - Cannot be a Java keyword

Method Name (cont'd.)

```
public static void main(String[] args)
```

The method that executes first when you run an application must be named main().

Other methods you write in a class can have any legal identifier.

```
public static void displayAddress()
```
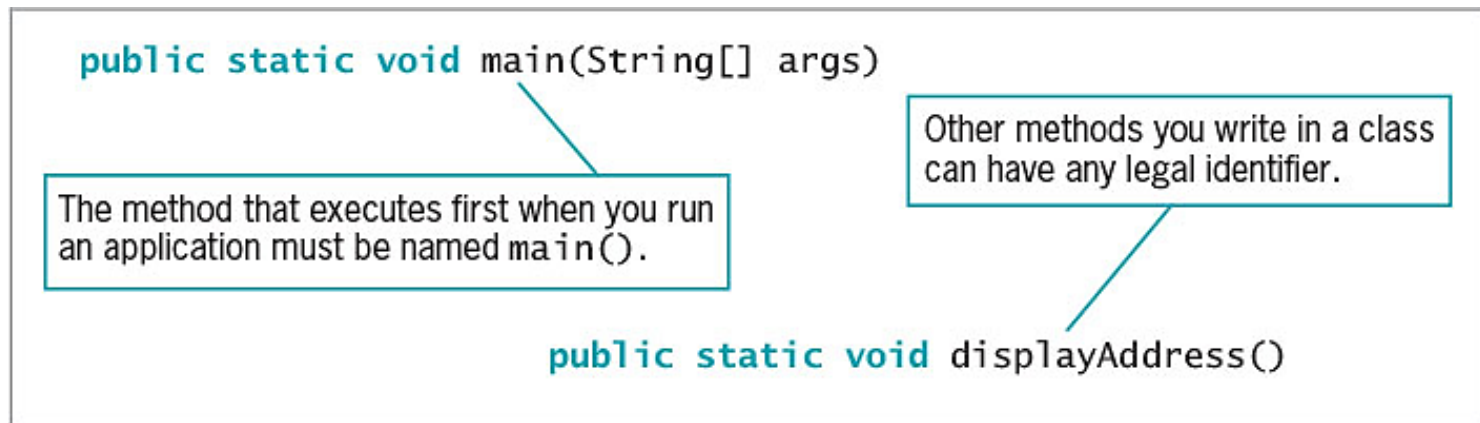
Figure 3-9   Identifiers for two methods

Parentheses

- Every method header contains a set of parentheses that follow the identifier

- May contain data to be sent to the method

- **Fully qualified identifier**
  - A complete name that includes the class
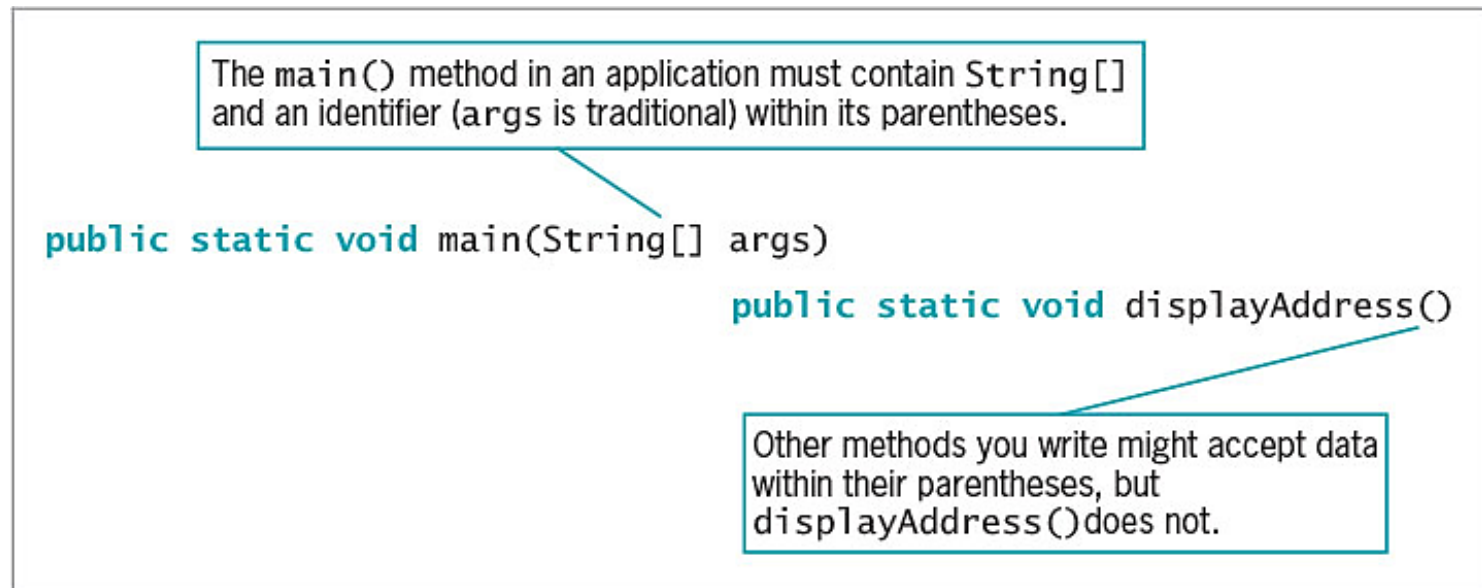
Parentheses (cont'd.)

The `main()` method in an application must contain `String[]` and an identifier (`args` is traditional) within its parentheses.

```
public static void main(String[] args)
```

```
public static void displayAddress()
```

Other methods you write might accept data within their parentheses, but `displayAddress()` does not.

Figure 3-10   Parentheses and their contents for two methods

Adding Parameters to Methods

- **Arguments**
  - Data items you use in a call to a method

- **Parameters**
  - Data items received by the method

- **Implementation hiding**
  - Encapsulation of method details within a class
  - The calling method needs to understand only the interface to the called method
  - **Interface**
    - The only part of a method that the client sees or with which it interacts

Creating a Method That Receives a Single Parameter

- Define the following:
  - Optional access specifiers
  - Return type for the method
  - Method name
  - Parameter type
  - Local name for the parameter

Creating a Method That Receives a Single Parameter (cont'd.)

Parameter data type

Parameter identifier

```java
public static void predictRaise(double salary)
{
    double newSalary;
    final double RAISE_RATE = 1.10;
    newSalary = salary * RAISE_RATE;
    System.out.println("Current salary: " +
        salary + "    After raise: " +
        newSalary);
}
```
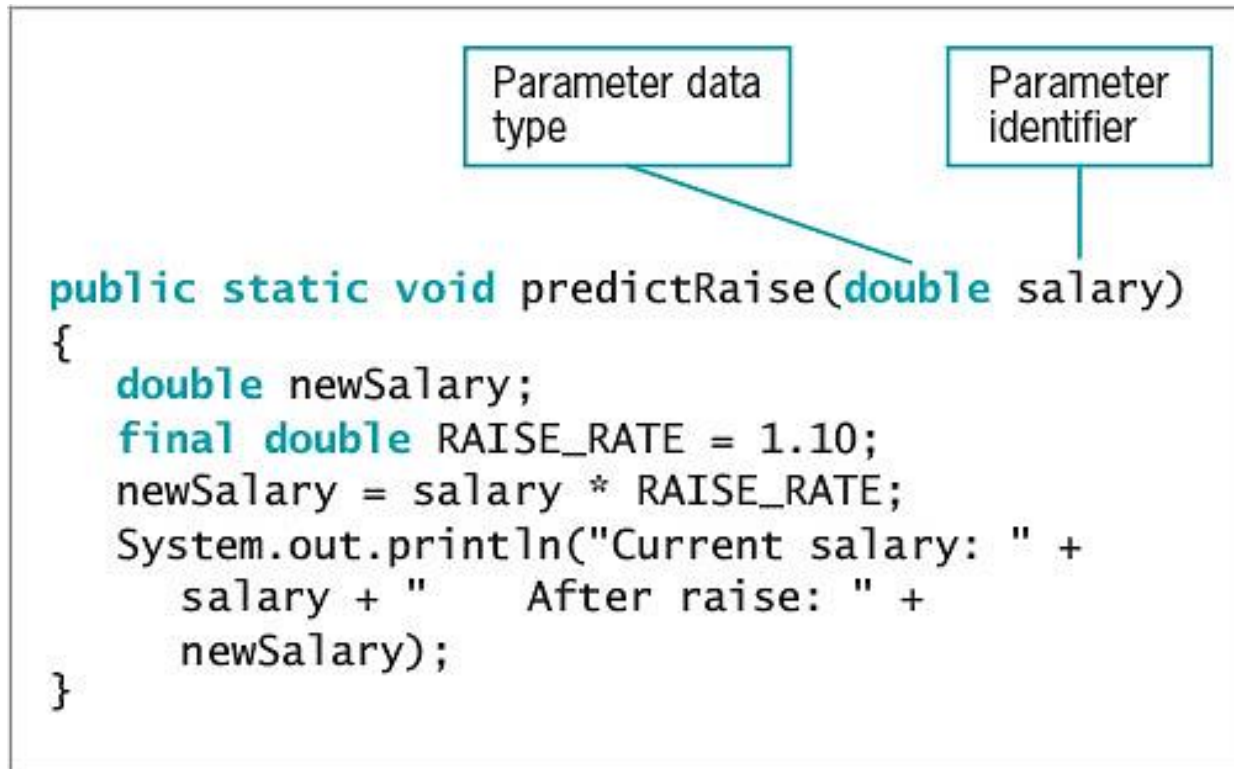
Figure 3-13   The predictRaise() method

Creating a Method That Receives a Single Parameter (cont'd.)

- **Local variable**
  - Known only within the boundaries of the method
  - Each time the method executes:
    - The variable is redeclared
    - A new memory location large enough to hold the type is set up and named

Creating a Method That Receives a Single Parameter (cont'd.)

```java
public class DemoRaise
{
    public static void main(String[] args)
    {
        double salary = 200.00;
        double startingWage = 800.00;
        System.out.println("Demonstrating some raises");
        predictRaise(400.00);
        predictRaise(salary);
        predictRaise(startingWage);
    }

    public static void predictRaise(double salary)
    {
        double newSalary;
        final double RAISE_RATE = 1.10;
        newSalary = salary * RAISE_RATE;
        System.out.println("Current salary: " +
            salary + "    After raise: " +
            newSalary);
    }
}
```

The predictRaise() method is called three times using three different arguments.

The parameter salary receives a copy of the value in each argument that is passed.

Figure 3-14 The DemoRaise class with a main() method that uses the predictRaise() method three times

Creating a Method That Requires Multiple Parameters

- A method can require more than one parameter

- List the arguments within the call to the method
  - Separate with commas

- Call a method
  - Arguments sent to the method must match the parameters listed in the method declaration by:
    - Number
    - Type

Creating a Method That Requires Multiple Parameters (cont'd.)

```java
public static void predictRaiseUsingRate(double salary, double rate)
{
    double newAmount;
    newAmount = salary * (1 + rate);
    System.out.println("With raise, new salary is " + newAmount);
}
```
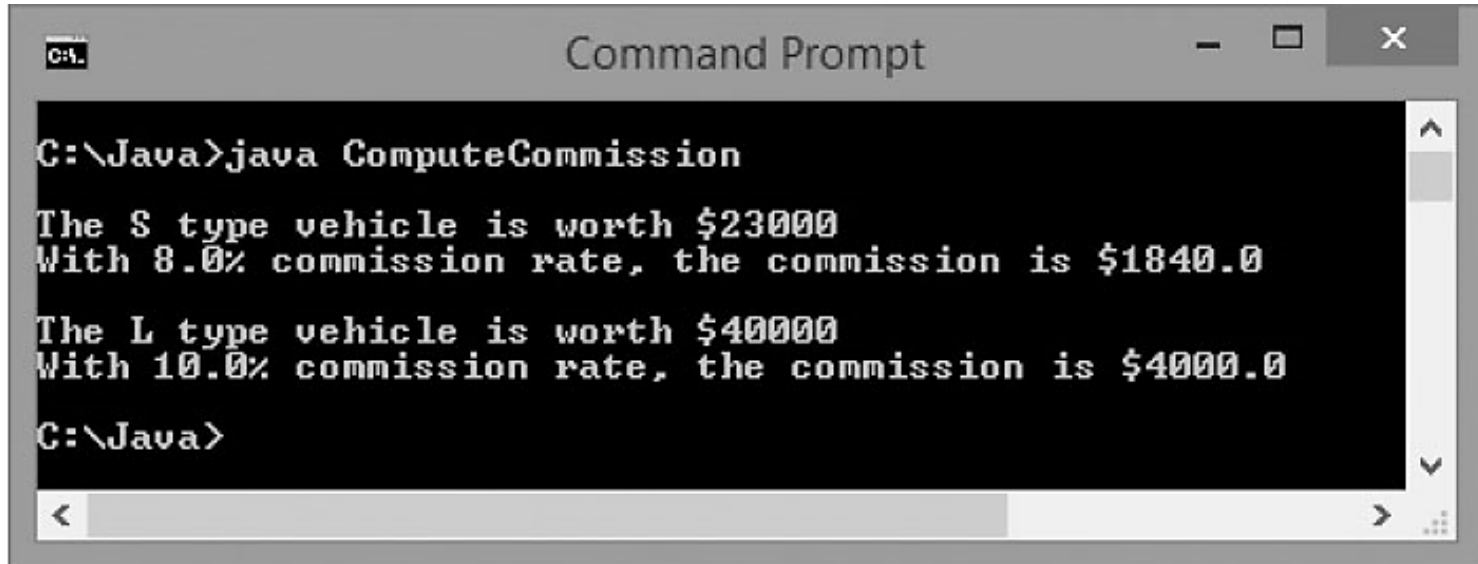
Figure 3-16 The predictRaiseUsingRate() method that accepts two parameters

Creating a Method That Requires Multiple Parameters (cont'd.)

```java
public class ComputeCommission
{
    public static void main(String[] args)
    {
        char vType = 'S';
        int value = 23000;
        double commRate = 0.08;
        computeCommission(value, commRate, vType);
        computeCommission(40000, 0.10, 'L');
    }
    public static void computeCommission(int value,
        double rate, char vehicle)
    {
        double commission;
        commission = value * rate;
        System.out.println("\nThe " + vehicle +
            " type vehicle is worth $" + value);
        System.out.println("With " + (rate * 100) +
            "% commission rate, the commission is $" +
            commission);
    }
}
```

Figure 3-17 The ComputeCommission class

Creating a Method That Requires Multiple Parameters (cont'd.)

Figure 3-18   Output of the `ComputeCommission` application

Creating Methods That Return Values

- **`return` statement**
  - Causes a value to be sent from the called method back to the calling method

- The return type can be any type used in Java
  - Primitive types
  - Class types
  - `void`
    - Returns nothing

- **Method's type**
  - A method's return type

Creating Methods That Return Values (cont'd.)

```java
public static double predictRaise(double salary)
{
    double newAmount;
    final double RAISE = 1.10;
    newAmount = salary * RAISE;
    return newAmount;
}
```

Figure 3-19   The predictRaise() method returning a double

Creating Methods That Return Values (cont'd.)

- **Unreachable statements (dead code)**
  - Logical flow leaves the method at the `return` statement
  - Can never execute
    - Causes a compiler error

Chaining Method Calls

- Any method might call any number of other methods

- Method acts as a **black box**
  - Do not need to know how it works
  - Just call and use the result

Chaining Method Calls (cont'd.)

```java
public static double predictRaise(double salary)
{
    double newAmount;
    double bonusAmount;
    final double RAISE = 1.10;
    newAmount = salary * RAISE;
    bonusAmount = calculateBonus(newAmount);
    newAmount = newAmount + bonusAmount;
    return newAmount;
}
```

Figure 3-20   The predictRaise() method calling the calculateBonus() method

Learning About Classes and Objects

- Every object is a member of a class

- **Is-a relationships**
  - An object "is a" concrete example of the class
  - The zoo's shark "is a" `Fish`

- **Instantiation**
  - Shark is an instantiation of the `Fish` class

- Reusability

Learning About Classes and Objects (cont'd.)

- Methods are often called upon to return a piece of information to the source of the request

- **Class client** or **class user**
  - An application or a class that instantiates objects of another prewritten class

Creating a Class

- Assign a name to the class

- Determine what data and methods will be part of the class

- Create a class header with three parts:
  - An optional access modifier
  - The keyword `class`
  - Any legal identifier for the name of the class

- `public` **class**
  - Accessible by all objects

Creating a Class (cont'd.)

```java
public class Employee
{
    private int empNum;
}
```

Figure 3-23 The Employee class with one field

Creating a Class (cont'd.)

- **Extended**
  - To be used as a basis for any other class

- **Data fields**
  - Variables declared within a class but outside of any method

- **Instance variables**
  - Nonstatic fields given to each object

Creating a Class (cont'd.)

- **Private access** for fields
  - No other classes can access the field's values
  - Only methods of the same class are allowed to use `private` variables

- **Information hiding**

- Most class methods are `public`

Creating Instance Methods in a Class

- Classes contain methods
  - **Mutator methods**
    - Set or change field values
  - **Accessor methods**
    - Retrieve values
  - **Nonstatic methods**
    - **Instance methods**
    - "Belong" to objects

- Typically declare nonstatic data fields
- `static` class variables are not instance variables

Static	Nonstatic
In Java, `static` is a keyword. It also can be used as an adjective.	There is no keyword for nonstatic items. When you do not explicitly declare a field or method to be static, then it is nonstatic by default.
Static fields in a class are called class fields.	Nonstatic fields in a class are called instance variables.
Static methods in a class are called class methods.	Nonstatic methods in a class are called instance methods.
When you use a static field or method, you do not use an object; for example: `JOptionPane.showDialog();`	When you use a nonstatic field or method, you must use an object; for example: `System.out.println();`
When you create a class with a static field and instantiate 100 objects, only one copy of that field exists in memory.	When you create a class with a nonstatic field and instantiate 100 objects, then 100 copies of that field exist in memory.
When you create a static method in a class and instantiate 100 objects, only one copy of the method exists in memory and the method does not receive a `this` reference.	When you create a nonstatic method in a class and instantiate 100 objects, only one copy of the method exists in memory, but the method receives a `this` reference that contains the address of the object currently using it.
Static class variables are not instance variables. The system allocates memory to hold class variables once per class, no matter how many instances of the class you instantiate. The system allocates memory for class variables the first time it encounters a class, and every instance of a class shares the same copy of any static class variables.	Instance fields and methods are nonstatic. The system allocates a separate memory location for each nonstatic field in each instance.

Table 3-1   Comparison of static and nonstatic

```
public class MyClass
{
    public static pubStatMethod()
    private static privStatMethod()
    public pubNonstatMethod()
    private privNonstatMethod()
}
```

The public static method can be used from TestClass with or without an object.

The public nonstatic method can be used from TestClass with a MyClass object.

```
public class TestClass
{
    MyClass object = new MyClass();

    object.pubNonstatMethod();

    object.pubStatMethod();

    MyClass.pubStatMethod();

// None of the following work

    MyClass.privStatMethod();

    MyClass.pubNonstatMethod();

    object.privStatMethod();

    object.privNonstatMethod();

    MyClass.privNonstatMethod();

}
```

TestClass doesn't have access to the private method.

The nonstatic method must be used with a MyClass object.

An object can use a static or nonstatic method, but these methods are private and cannot be used here.

This is wrong on two counts—the method is nonstatic, so it needs an object, and in any event, the method is private.
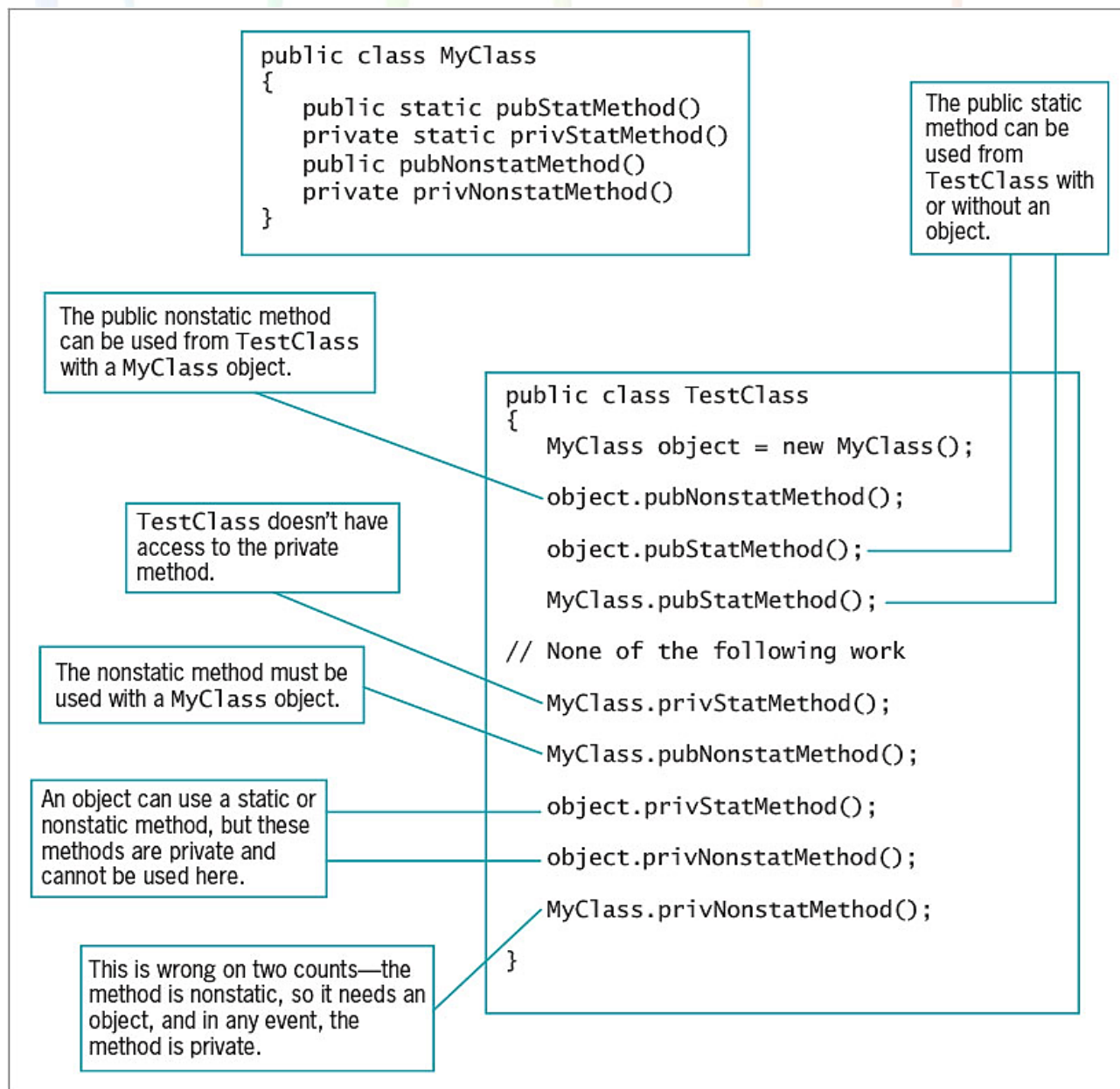
Figure 3-25 Summary of legal and illegal method calls based on combinations of method modifiers

Creating Instance Methods in a Class (cont'd.)

```java
public class Employee
{
    private int empNum;
    public int getEmpNum()
    {
        return empNum;
    }
    public void setEmpNum(int emp)
    {
        empNum = emp;
    }
}
```

Figure 3-26   The Employee class with one field and two methods

Organizing Classes

- Place data fields in logical order
  - At the beginning of a class
  - List the fields vertically

- Data fields and methods may be placed in any order within a class
  - It's common to list all data fields first
  - Names and data types can be seen before reading the methods that use the data fields

Organizing Classes (cont'd.)

```java
public class Employee
{
    private int empNum;
    private String empLastName;
    private String empFirstName;
    private double empSalary;
    public int getEmpNum()
    {
        return empNum;
    }
    public void setEmpNum(int emp)
    {
        empNum = emp;
    }
    public String getEmpLastName()
    {
        return empLastName;
    }
```

Figure 3-28   The Employee class with several data fields and corresponding methods *(continues)*

Organizing Classes (cont'd.)

(continued)

```java
    public void setEmpLastName(String name)
    {
        empLastName = name;
    }
    public String getEmpFirstName()
    {
        return empFirstName;
    }
    public void setEmpFirstName(String name)
    {
        empFirstName = name;
    }
    public double getEmpSalary()
    {
        return empSalary;
    }
    public void setEmpSalary(double sal)
    {
        empSalary = sal;
    }
}
```

Figure 3-28 The `Employee` class with several data fields and corresponding methods

Declaring Objects and Using Their Methods

- Declaring a class does not create any actual objects

- To create an instance of a class:
  - Supply a type and an identifier
  - Allocate computer memory for the object
  - Use the **new operator**

    ```
    Employee someEmployee;
    someEmployee = new Employee();
    ```
    or
    ```
    Employee someEmployee = new Employee();
    ```

Declaring Objects and Using Their Methods (cont'd.)

- **Reference to the object**
  - The name for a memory address where the object is held
- **Constructor** method
  - A method that creates and initializes class objects
  - You can write your own constructor methods
  - Java writes a constructor when you don't write one
  - The name of the constructor is always the same as the name of the class whose objects it constructs

Declaring Objects and Using Their Methods (cont'd.)

- After an object is instantiated, its methods can be accessed using:
  - The object's identifier
  - A dot
  - A method call

Declaring Objects and Using Their Methods (cont'd.)

```java
public class DeclareTwoEmployees
{
    public static void main(String[] args)
    {
        Employee clerk = new Employee();
        Employee driver = new Employee();
        clerk.setEmpNum(345);
        driver.setEmpNum(567);
        System.out.println("The clerk's number is " +
            clerk.getEmpNum() + " and the driver's number is " +
            driver.getEmpNum());
    }
}
```

Figure 3-29   The DeclareTwoEmployees class

Understanding Data Hiding

- Data hiding using encapsulation
  - Data fields are usually `private`
  - The client application accesses them only through `public` interfaces
- `set` method
  - Controls the data values used to set a variable
- `get` method
  - Controls how a value is retrieved

An Introduction to Using Constructors

`Employee chauffeur = new Employee();`

– Actually a calling method named `Employee()`

- **Default constructors**

  – Require no arguments

  – Created automatically by a Java compiler

    - For any class
    - Whenever you do not write a constructor

An Introduction to Using Constructors (cont'd.)

- The default constructor provides specific initial values to an object's data fields
  - Numeric fields
    - Set to 0 (zero)
  - Character fields
    - Set to Unicode '\u0000'
  - Boolean fields
    - Set to `false`
  - Nonprimitive object fields
    - Set to `null`

An Introduction to Using Constructors (cont'd.)

- A constructor method:
  - Must have the same name as the class it constructs
  - Cannot have a return type
  - `public` access modifier

An Introduction to Using Constructors (cont'd.)

```java
public Employee()
{
    empSalary = 300.00;
}
```

Figure 3-32 The Employee class constructor that assigns a salary

Understanding That Classes Are Data Types

- Classes you create become data types
  - Often referred to as **abstract data types** (**ADT**s)
    - Implementation is hidden and accessed through public methods
  - **Programmer-defined data type**
    - Not built into the language

- Declare an object from one of your classes
  - Provide the type and identifier

You Do It

- Creating a `static` Method that Requires No Arguments and Returns No Values

- Creating `static` Methods that Accept Arguments and Return a Value

- Creating a Class that Contains Instance Fields and Methods

- Declaring and Using Objects

- Adding a Constructor to a Class

- Understanding that Classes are Data Types

Don't Do It

- Don't place a semicolon at the end of a method header

- Don't think "default constructor" means only the automatically supplied constructor

- Don't think that a class's methods must:
  - Accept its own fields' values as parameters
  - Return values to its own fields

- Don't create a class method that has a parameter with the same identifier as a class field

Summary

- Method
  - A series of statements that carry out a task
    - A declaration includes the parameter type and local name for a parameter
    - You can pass multiple arguments to methods
  - Has a return type

- Class objects
  - Have attributes and methods associated with them

- Instantiate objects that are members of a class

Summary (cont'd.)

- Constructor
  - A method establishes an object and provides specific initial values for an object's data fields

- Everything is an object
  - Every object is a member of a more general class

- Implementation hiding, or encapsulation
  - `private` data fields
  - `public` access methods