Java Classes

Appendix B

Objects and Classes

- An object belongs to a class, which defines its data type
- A class specifies …
  - Kind of data objects of that class have
  - What actions the objects can take
  - How they accomplish these actions
- Object Oriented Programming
  - A world consisting of objects that interact with one another by means of actions

Objects and Classes

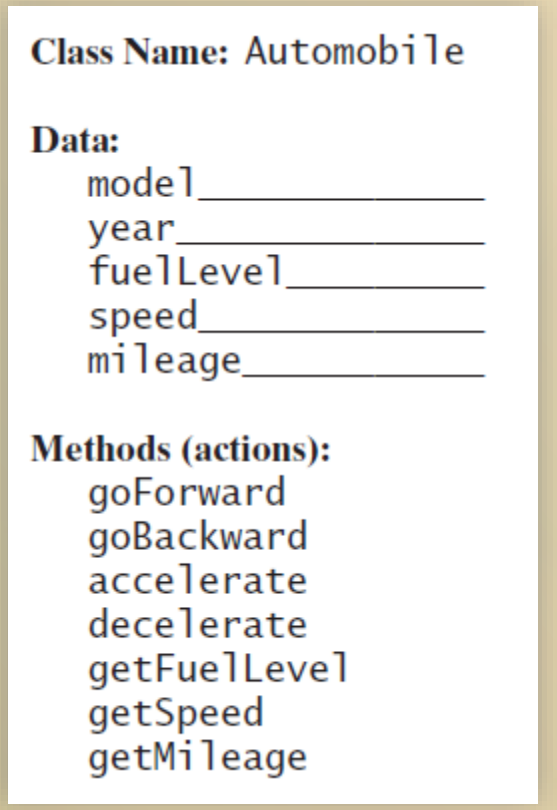
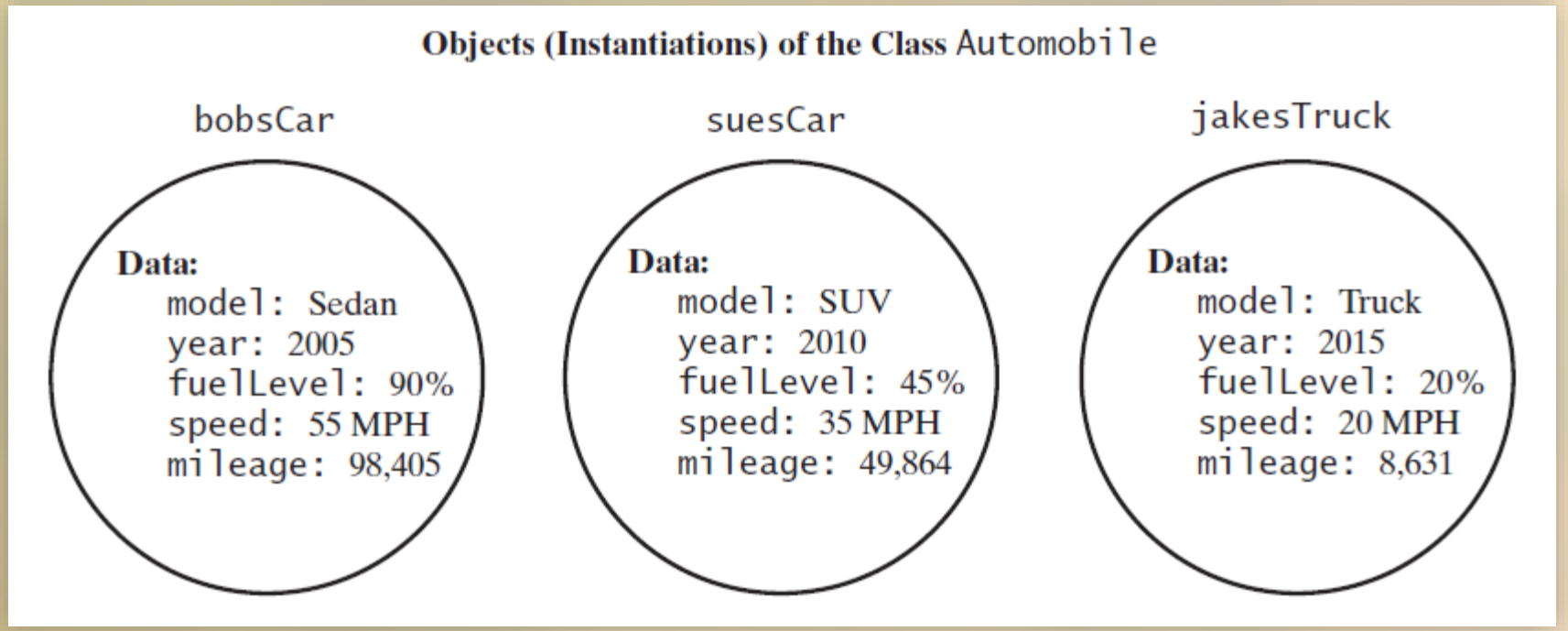


Figure C-1 An outline of a class and …

Objects and Classes



… three of its instances

Using the Methods in a Java Class

- A program component that uses a class is called a client of the class
- The **new** operator creates an instance of a class
  - By invoking a special method within the class
  - Known as a constructor
- Variable **joe** references memory location where object is stored

```
Name joe = new Name();
```

Using the Methods in a Java Class

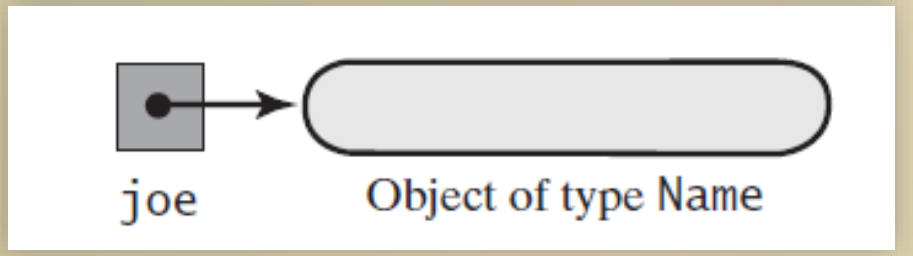


Figure C-2 A variable that references an object

Using the Methods in a Java Class

- Class should have methods that give capability to set data values

```
joe.setFirst("Joseph");
joe.setLast("Brown");
```

  - Void methods, they do not return a value.
- Class needs methods to retrieve values

```
String hisName = joe.getFirst();
```

  - Valued methods, return a value

References and Aliases

- A reference variable contains the address in memory of an actual object.
- Consider:

```
Name jamie = new Name();
jamie.setFirst("Jamie");
jamie.setLast("Jones");
Name friend = jamie;
```

- Variables **jamie** and **friend** reference the same instance of **Name**

References and Aliases

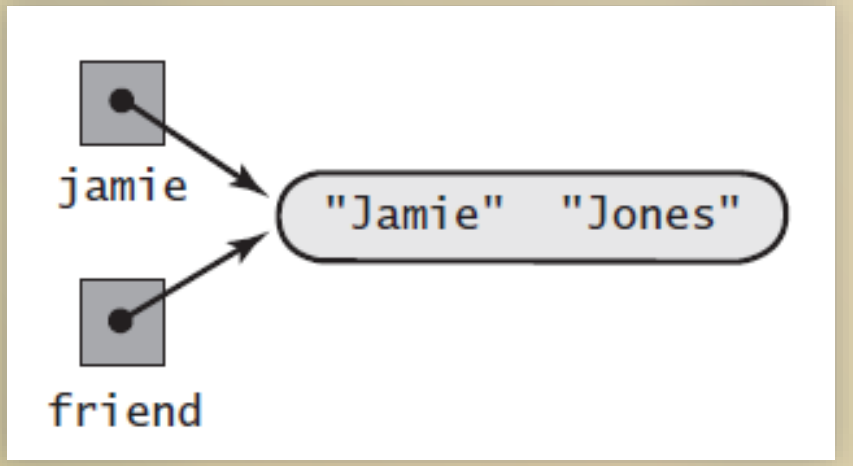


Figure C-3 Aliases of an object

Defining a Java Class

- The Java class Name that represents a person's name.
  - Store a class definition in a file
  - File name is the name of the class followed by .java.

```
public class Name
{
   private String first; // First name
   private String last;  // Last name
   < Definitions of methods are here >
   . . .
} // end Name
```

Defining a Java Class

```
public class Name
{
   private String first; // First name
   private String last;  // Last name
   < Definitions of methods are here >
   . . .
} // end Name
```

- **public** means no restrictions on where class is used
- Strings **first** and **last** are class's data fields
- **private** means only methods within class can refer to the data fields

Defining a Java Class

```
public class Name
{
   private String first; // First name
   private String last;  // Last name

   < Definitions of methods are here >

   . . .
} // end Name
```

- **private** fields accessed by
  - Accessor methods (get)
  - Mutator methods (set)

Method Definitions

- General form:

```
access-modifier  use-modifier  return-type  method-name(parameter-list)
{
   method-body
}
```

  - *use modifier* is optional and in most cases omitted
  - *return type*, (for a valued method), data type of the value method returns
  - *parameters* specify values, objects that are inputs

Method Definitions

- Examples of **get** and **set** methods

```
public String getFirst()          <-- Header
{
   return first;                  } Body
} // end getFirst
```

```
public void setFirst(String firstName)
{
   first = firstName;
} // end setFirst
```

- Possible to reference class data field **first** with **this.first**
  - **this** references "this" instance of the **Name** object

Arguments and Parameters

```
Name joe = new Name();
joe.setFirst("Joseph");
joe.setLast("Brown");
```

- Consider:
- Strings "Joseph" and "Brown" are the arguments.
  - Correspond to the parameters of the method definition
- Method invocation must provide exactly as many arguments as parameters as method definition

Passing Arguments

- Method cannot change the value of an argument that has a primitive data type
  - Mechanism is described as *call-by-value.*
- When parameter has class type, corresponding argument in method invocation must be object of that class type
  - Parameter is initialized to the memory address of that object
  - Method can change the data in the object

Passing Arguments

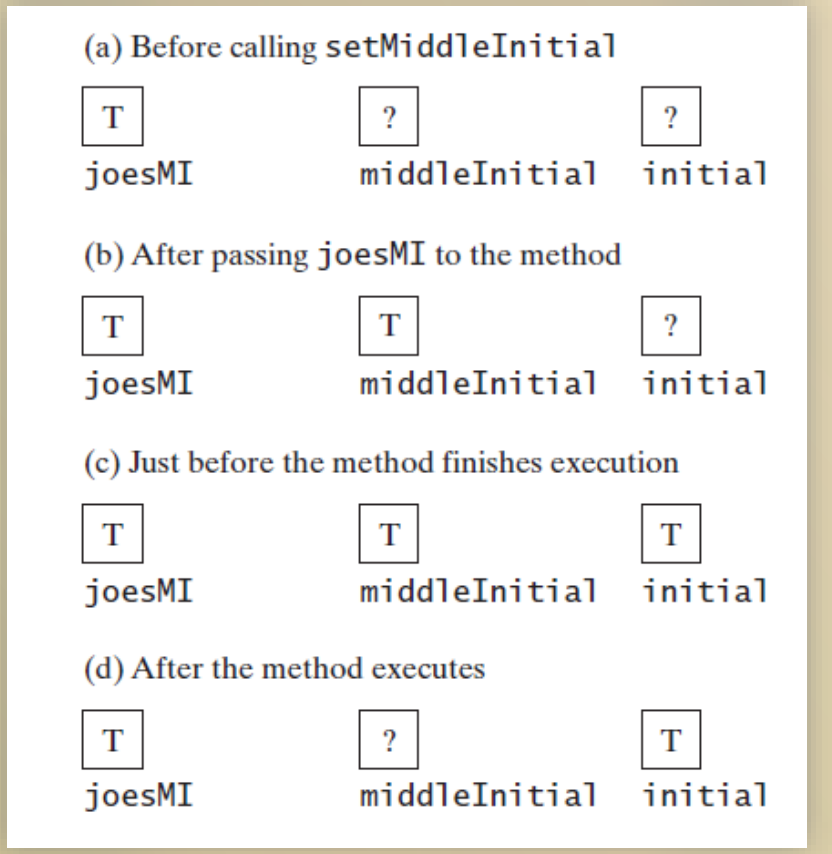


Figure C-4 The effect of executing the method **setMiddleInitial** on its argument **joesMI**, its parameter **middleInitial**, and the data field initial

Passing Arguments

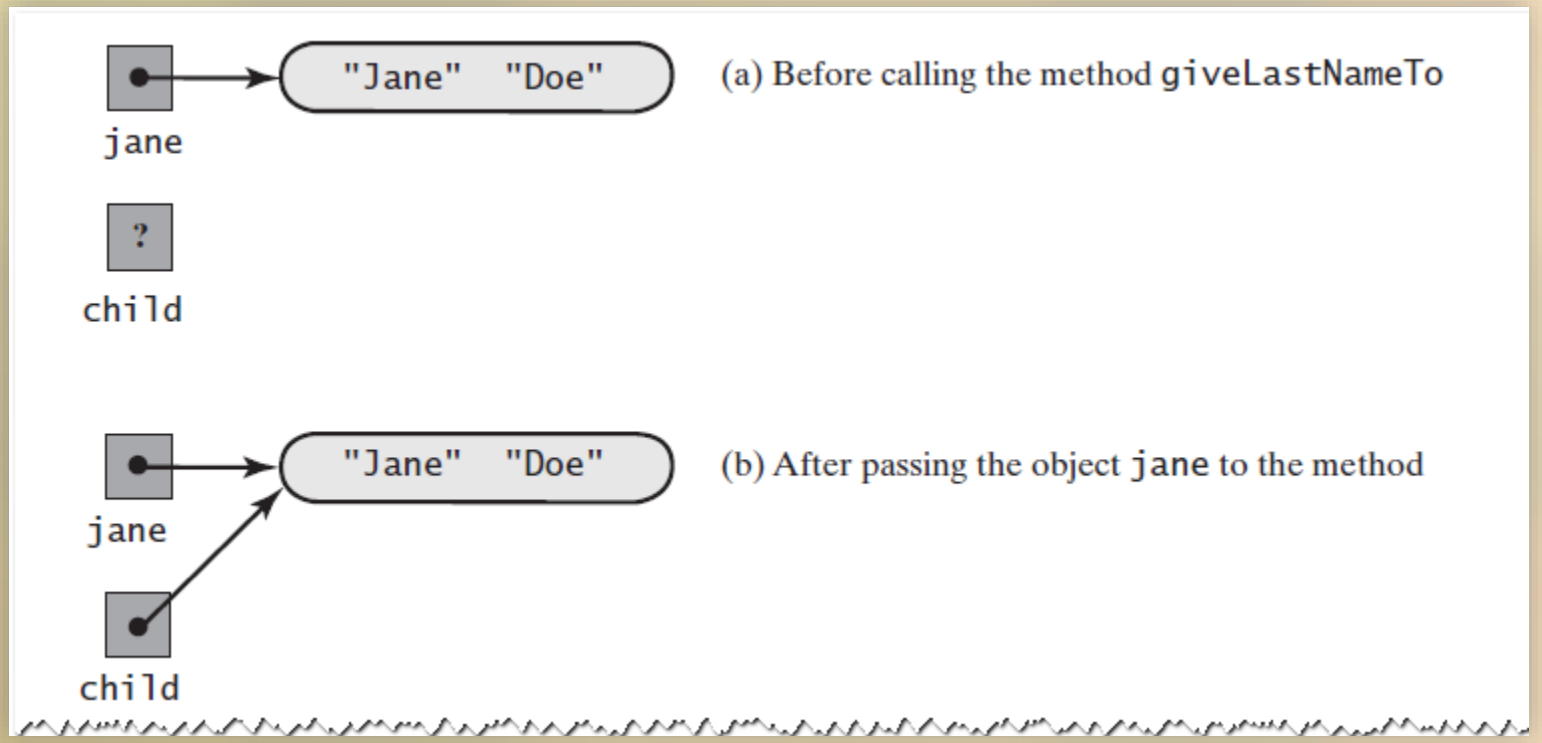


Figure C-5 The method **giveLastNameTo** modifies the object passed to it as an argument

Passing Arguments

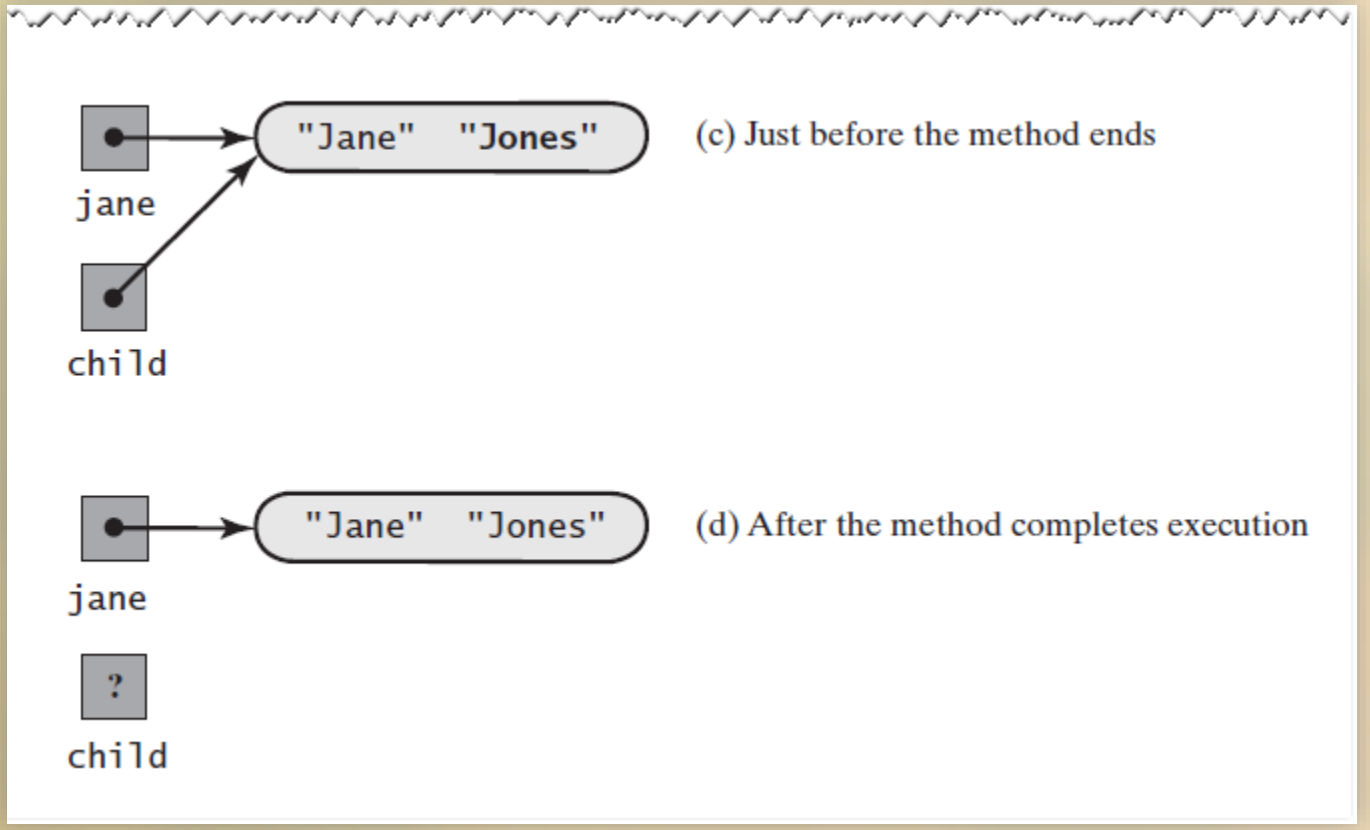


Figure C-5 The method **giveLastNameTo** modifies the object passed to it as an argument

Passing Arguments

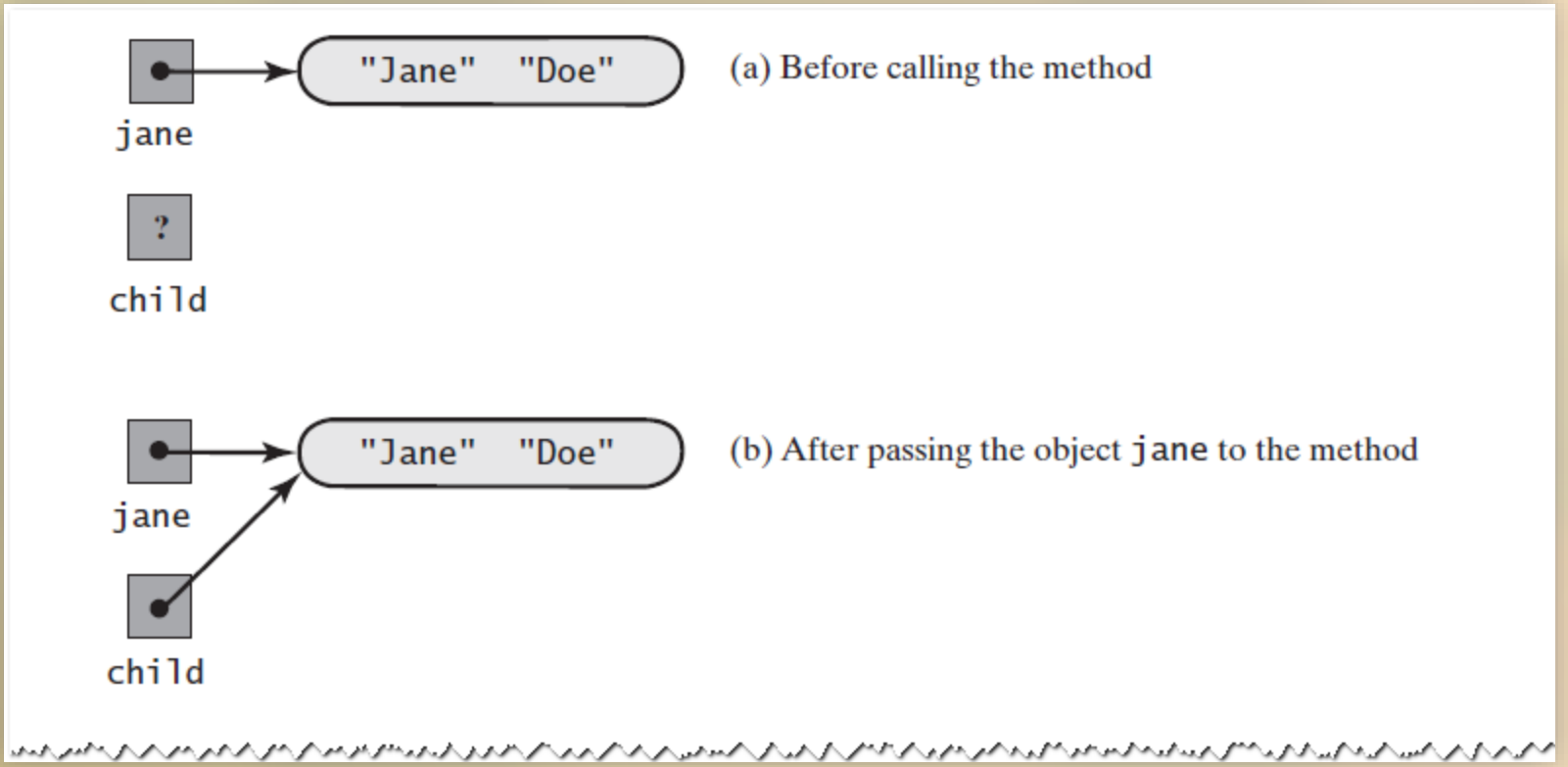


Figure C-6 A method cannot replace an object passed to it as an argument

Passing Arguments

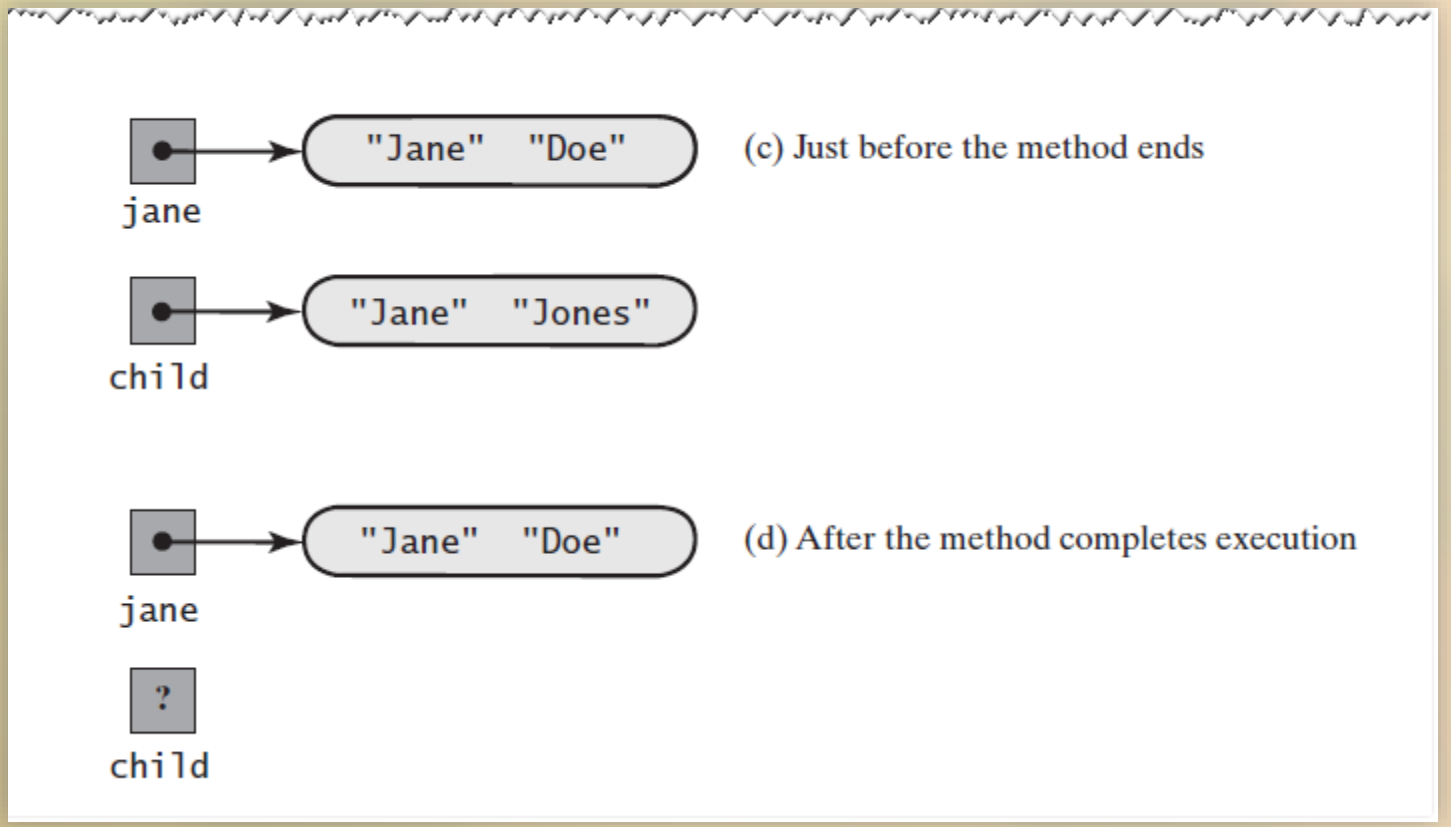


Figure C-6 A method cannot replace an object passed to it as an argument

A Definition of the Class Name

```
1  public class Name
2  {
3     private String first; // First name
4     private String last;  // Last name
5
6     public Name()
7     {
8     } // end default constructor
9
10    public Name(String firstName, String lastName)
11    {
12       first = firstName;
13       last = lastName;
14    } // end constructor
15
16    public void setName(String firstName, String lastName)
17    {
18       setFirst(firstName);
19       setLast(lastName);
20    } // end setName
21
22    public String getName()
```

LISTING C-1 The class **Name**

A Definition of the Class Name

```
21
22      public String getName()
23      {
24         return toString();
25      } // end getName
26
27      public void setFirst(String firstName)
28      {
29         first = firstName;
30      } // end setFirst
31
32      public String getFirst()
33      {
34         return first;
35      } // end getFirst
36
37      public void setLast(String lastName)
38      {
39         last = lastName;
40      } // end setLast
41
```

LISTING C-1 The class **Name**

A Definition of the Class Name

```
40    } // end setLast
41
42    public String getLast()
43    {
44       return last;
45    } // end getLast
46
47    public void giveLastNameTo(Name aName)
48    {
49       aName.setLast(last);
50    } // end giveLastNameTo
51
52    public String toString()
53    {
54       return first + " " + last;
55    } // end toString
56 } // end Name
```

LISTING C-1 The class **Name**

Constructors

- Constructor allocates memory for object, initializes the data fields
- Constructor has certain special properties
  - Same name as the class
  - No return type, not even **void**
  - Any number of parameters, including no parameters
- Constructor without parameters called the default constructor

Constructors

- Consider these two statements:

```
Name jill = new Name("Jill", "Jones");
jill = new Name("Jill", "Smith");
```

  - Second statement allocates new memory, with **jill** pointing to it
  - Previous memory location "lost"
  - System periodically deallocates, returns to O.S.

Constructors

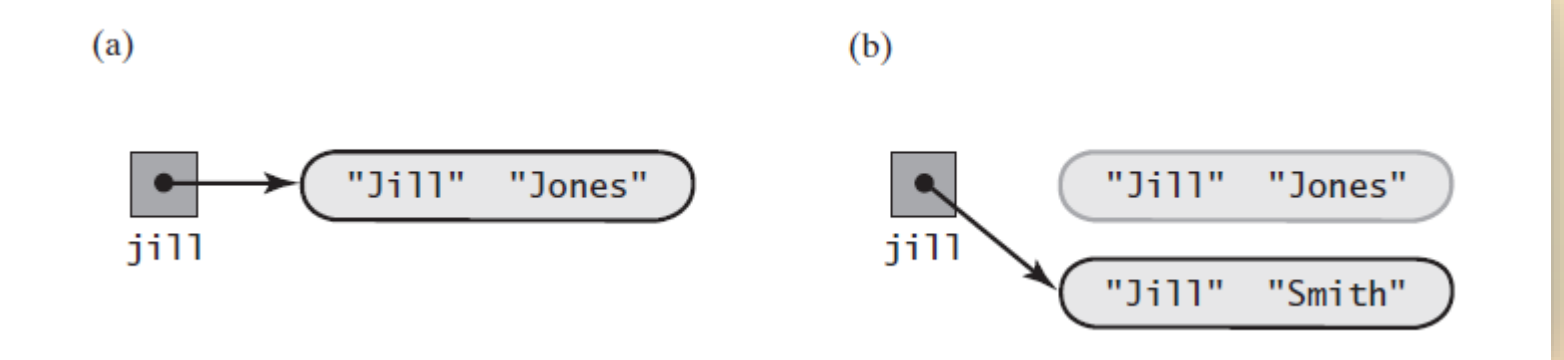


Figure C-7 An object (a) after its initial creation;
(b) after its reference is lost

The Method **toString**

- Method **toString** in class **Name** returns a string that is person's full name
  - Java will invoke it automatically when you write

```
System.out.println(jill);
```

- Providing a class with a method **toString** is a good idea in general

Methods That Call Other Methods

- **setName** could use assignment statements to initialize **first** and **last**
  - Instead invokes the methods **setFirst** and **setLast**
- Method **getName** in the class **Name** also invokes another of **Name**'s methods
  - Uses **toString**
  - Rather than writing same statements in both methods, we have one method call the other

Methods That Call Other Methods

- Can use the reserved word **this** to call a constructor
  - From within the body of another constructor.

```
public Name()
{
   this("", "");
} // end default constructor
```

  - Revision of default constructor to initialize **first** and **last**, by calling the second constructor

Methods That Return an Instance of Their Class

- Could have **setName** return reference to revised instance of **Name**, as follows:

```
public Name setName(String firstName, String lastName)
{
   setFirst(firstName);
   setLast(lastName);

   return this;
} // end setName
```

  - Can call this definition of **setName**

```
Name jill = new Name();
Name myFriend = jill.setName("Jill", "Greene");
```

Static Fields and Methods

- Sometimes you need a data field that does not belong to any one object
  - Such a data field is called a *static field*

```
private static int numberOfInvocations = 0;
```

- Objects can use static field to communicate with each other
  - Or to perform some joint action.

Static Fields and Methods

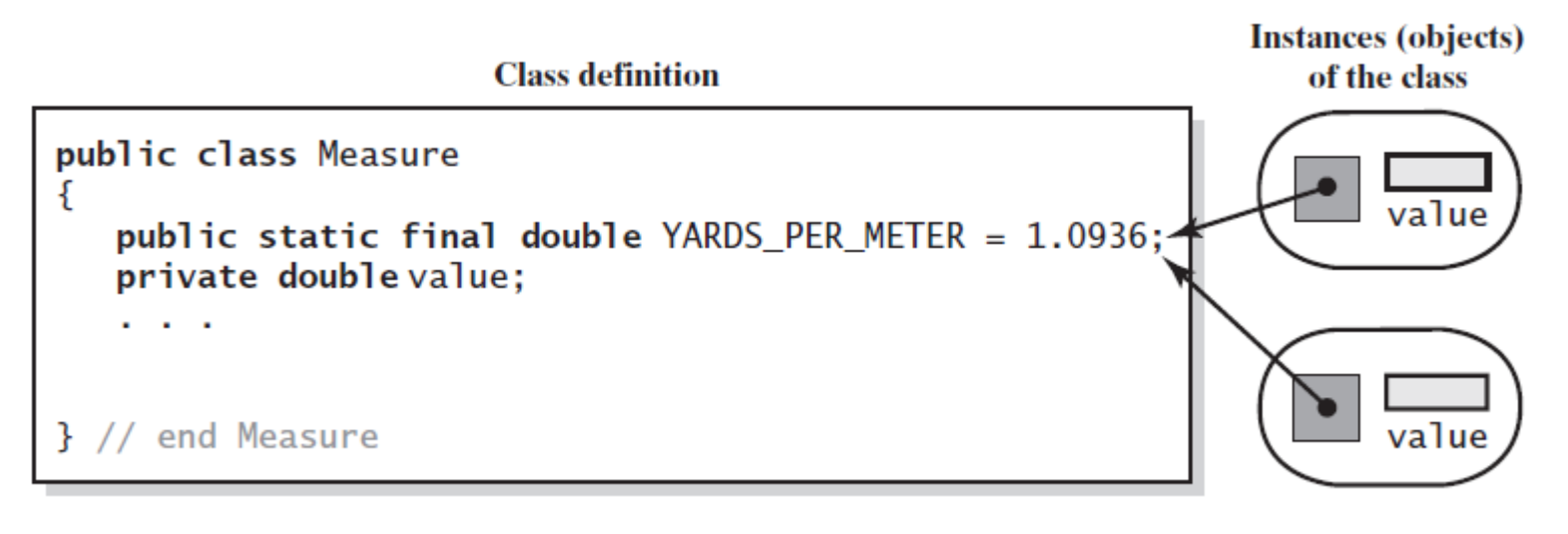


Figure C-8 A static field **YARDS_PER_METER** versus a nonstatic field value. Objects of the class **Measure** all reference the same static field but have their own copy of value

Static Fields and Methods

- Static method: a method that does not belong to an object of any kind.
  - Still a member of a class
  - Use the class name instead of an object name to invoke the method
- Methods from class **Math**

```
int maximum = Math.max(2, 3);
double root = Math.sqrt(4.2);
```

Overloading Methods

- Methods within same class can have same name,

```
public void setName(String firstName, String lastName)
public void setName(Name otherName)
```

  - As long as they do not have identical parameters

Enumeration as a Class

- When you define an enumeration, the class created has methods such as
  - **toString**, **equals**, **ordinal**, and **valueOf**
- You can define additional methods for any enumeration
  - Including constructors

Enumeration as a Class

```
1  /** An enumeration of card suits. */
2  enum Suit
3  {
4     CLUBS("black"), DIAMONDS("red"), HEARTS("red"), SPADES("black");
5
6     private final String color;
7
8     private Suit(String suitColor)
9     {
10       color = suitColor;
11    } // end constructor
12
13    public String getColor()
14    {
15       return color;
16    } // end getColor
17 } // end Suit
```

LISTING C-2 The enumeration **Suit**

Enumeration as a Class

```
1  /** A demonstration of the enumeration Suit. */
2  public class SuitDemo
3  {
4     private enum Suit
5     {
6        . . . <See Listing C-2>
7     } // end Suit
8
9     public static void main(String[] args)
10    {
11       for (Suit nextSuit : Suit.values())
12       {
13          System.out.println(nextSuit + " are " + nextSuit.getColor() +
14                             " and have an ordinal value of " +
15                             nextSuit.ordinal());
16       } // end for
17    } // end main
18 } // end SuitDemo
```

Output

```
CLUBS are black and have an ordinal value of 0
DIAMONDS are red and have an ordinal value of 1
HEARTS are red and have an ordinal value of 2
SPADES are black and have an ordinal value of 3
```

LISTING C-2 The enumeration **Suit**

Enumeration as a Class

```
1  public enum LetterGrade
2  {
3     A("A", 4.0), A_MINUS("A-", 3.7), B_PLUS("B+", 3.3), B("B", 3.0),
4     B_MINUS("B-", 2.7), C_PLUS("C+", 2.3), C("C", 2.0), C_MINUS("C-", 1.7),
5     D_PLUS("D+", 1.3), D("D", 1.0), F("F", 0.0);
6
7     private final String grade;
8     private final double points;
9
10    private LetterGrade(String letterGrade, double qualityPoints)
11    {
12       grade = letterGrade;
13       points = qualityPoints;
14    } // end constructor
15
16    public String getGrade()
17    {
18       return grade;
19    } // end getGrade
20
```

LISTING C-4 The enumeration **LetterGrade**

Enumeration as a Class

```
21    public double getQualityPoints()
22    {
23       return points;
24    } // end getQualityPoints
25
26    public String toString()
27    {
28       return getGrade();
29    } // end toString
30 } // end LetterGrade
```

LISTING C-4 The enumeration **LetterGrade**

Given LetterGrade myGrade = LetterGrade.B_PLUS;
Then …

`myGrade.toString()` returns the string *B+*.
`System.out.println(myGrade)` displays *B+*, since it calls `toString` implicitly.
`myGrade.getGrade()` returns the string *B+*.
`myGrade.getQualityPoints()` returns 3.3.

Packages

- Using several related classes is more convenient if …
  - You group them together within a Java package
- To identify a class as part of a particular package
  - Begin the file that contains the class with a statement like **package myStuff;**
  - Then place all of the files within one directory or folder, give it same name as the package.

Packages

- To use a package in your program …
  - Begin the program with a statement such as **import myStuff.*;**
- Asterisk makes all public classes within package available to the program

The Java Class Library

- Java comes with a collection of many classes you can use
  - This collection of classes is known as **the Java Class Library**
  - Sometimes as the **Java Application Programming Interface**

Java Classes

End