CS2400 - Data Structures and Advanced Programming

Module 2: Review of Java Programming Basics, Interface, and Generic Data Types

Hao Ji

Computer Science Department

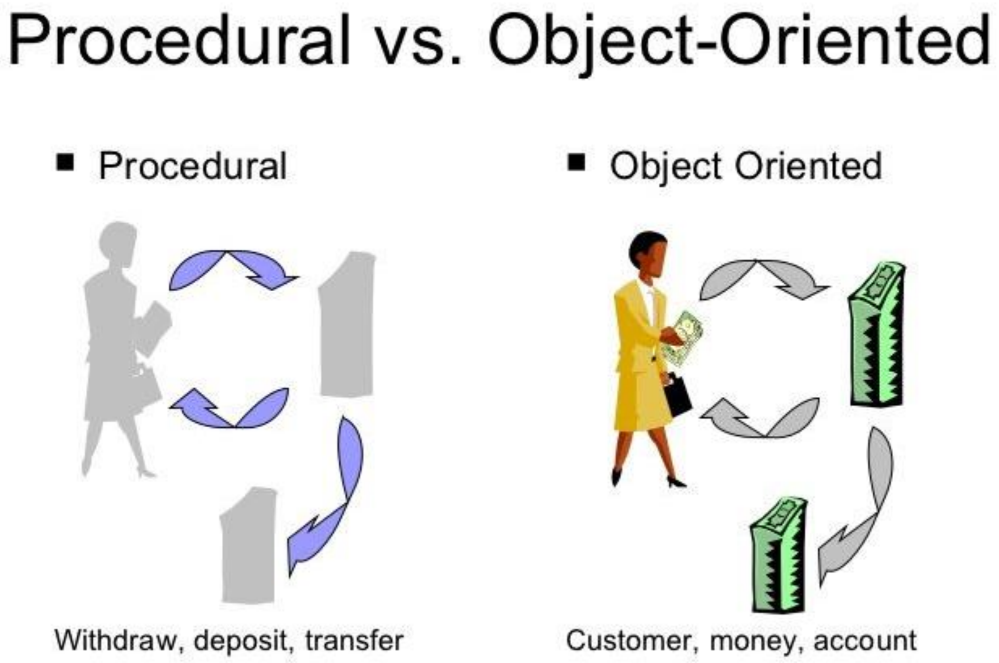
Cal Poly Pomona

Data Structures and **Advanced Programming**

- Objectives
  - Comprehend relationships between data structures and algorithms, and performance issues involved.
  - Use linear data structures such as array, list, stacks, queues, and hash tables.
  - Use non-linear data structures such as trees and graphs.
  - Perform analysis of algorithms.
  - Have proficiency in recursion.
  - Know advanced file access.
  - **Gain experience in Java generic programming.**
  - **Practice library facilities of a high-level programming language.**

Review: Java Programming

- **Object-oriented programming**, or **OOP**,
  - Views a program as a sort of world consisting of objects that interact with one another by means of actions.



Procedural vs. Object-Oriented

- Procedural
- Object Oriented

Withdraw, deposit, transfer

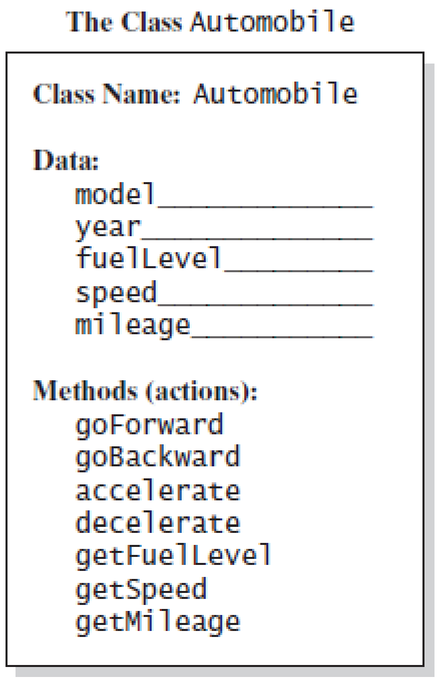
Customer, money, account

https://www.alphansotech.com/procedural-vs-object-oriented-programming/

Review: Java Programming

- **Object-oriented programming**, or **OOP**,
  - Views a program as a sort of world consisting of objects that interact with one another by means of actions.

- A **class** specifies the kind of data the objects of that class have and what actions the objects can take and how they accomplish those actions.
  - For example, in a program that simulates automobiles, each automobile is an object.



The Class Automobile

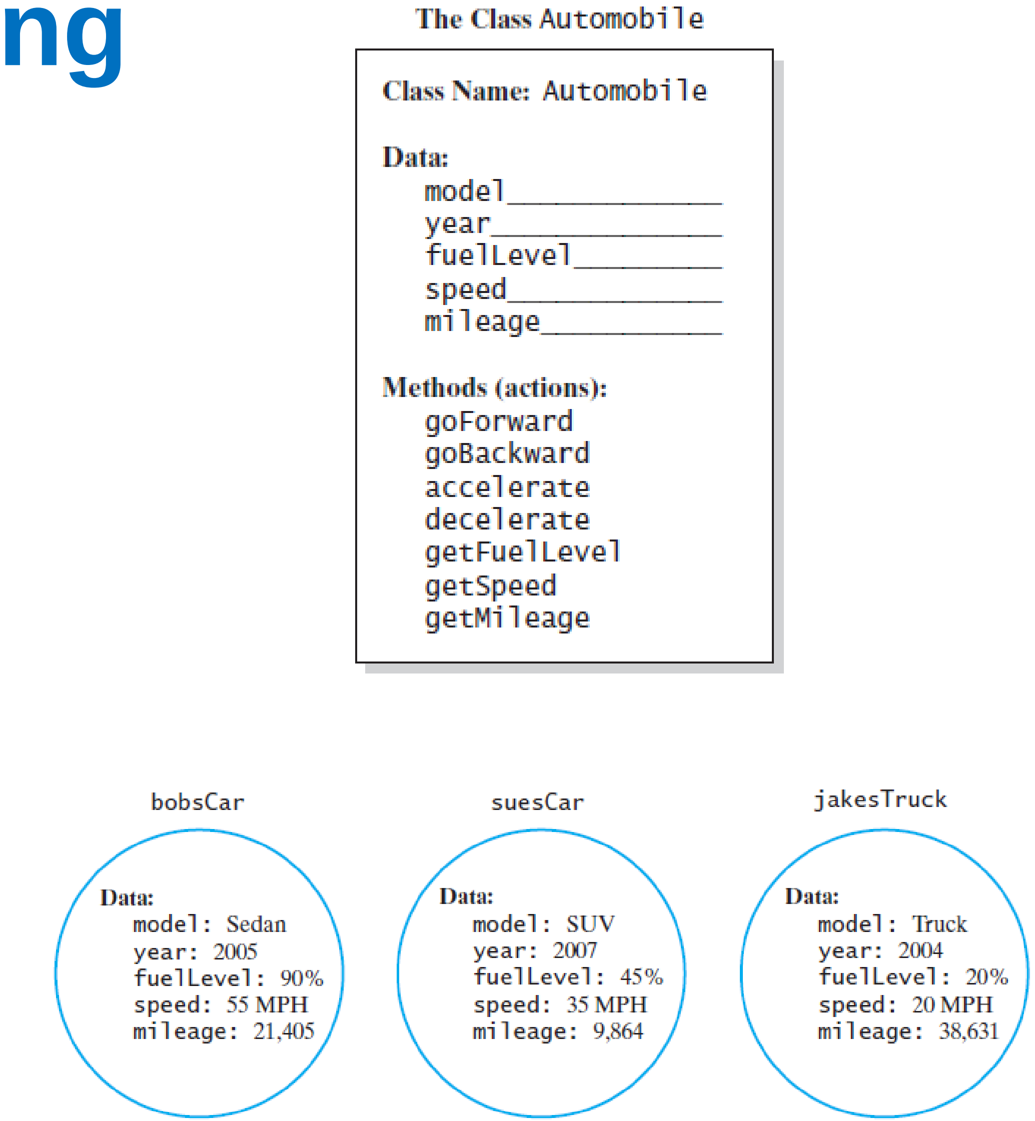
Class Name: Automobile

Data:
    model______________
    year_______________
    fuelLevel__________
    speed______________
    mileage____________

Methods (actions):
    goForward
    goBackward
    accelerate
    decelerate
    getFuelLevel
    getSpeed
    getMileage

Review: Java Programming

- **Object-oriented programming**, or **OOP**,
  - Views a program as a sort of world consisting of objects that interact with one another by means of actions.
- A **class** specifies the kind of data the objects of that class have and what actions the objects can take and how they accomplish those actions.
  - For example, in a program that simulates automobiles, each automobile is an object.
- The **objects** in a Java program interact, and this interaction forms the solution to a given problem.



The Class Automobile

Class Name: Automobile

Data:
model_____________
year_____________
fuelLevel__________
speed_____________
mileage___________

Methods (actions):
goForward
goBackward
accelerate
decelerate
getFuelLevel
getSpeed
getMileage

bobsCar

Data:
model: Sedan
year: 2005
fuelLevel: 90%
speed: 55 MPH
mileage: 21,405

suesCar

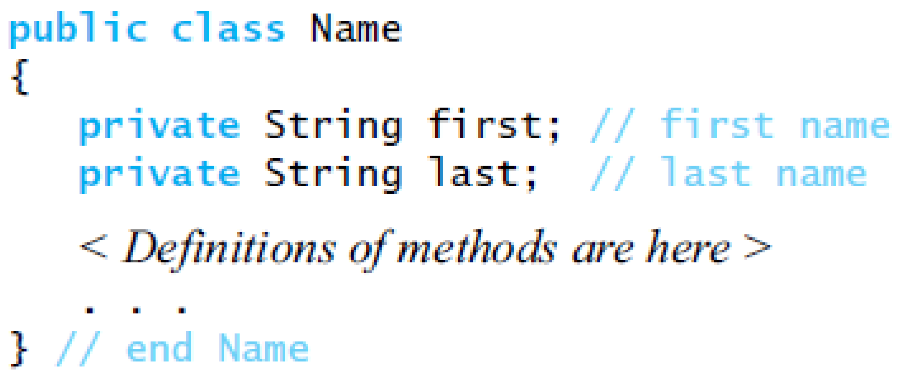
Data:
model: SUV
year: 2007
fuelLevel: 45%
speed: 35 MPH
mileage: 9,864

jakesTruck

Data:
model: Truck
year: 2004
fuelLevel: 20%
speed: 20 MPH
mileage: 38,631

Review: Java Programming

- **Defining a Java Class**



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

   < Definitions of methods are here >

   . . .
} // end Name
```

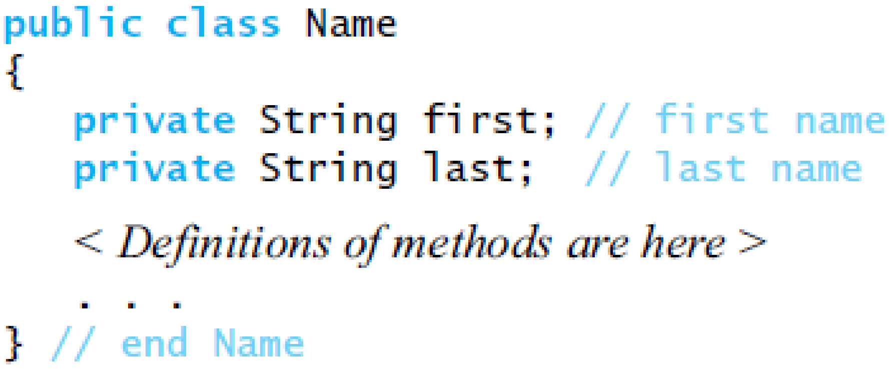
- “Name.java” stores a class definition in a file whose name is the name of the class followed by .java. Typically, you store only one class per file

Review: Java Programming

- **Defining a Java Class**

The word public simply means that there are no restrictions on where the class is used.

That is, the class *Name* is available for use in any other Java class



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

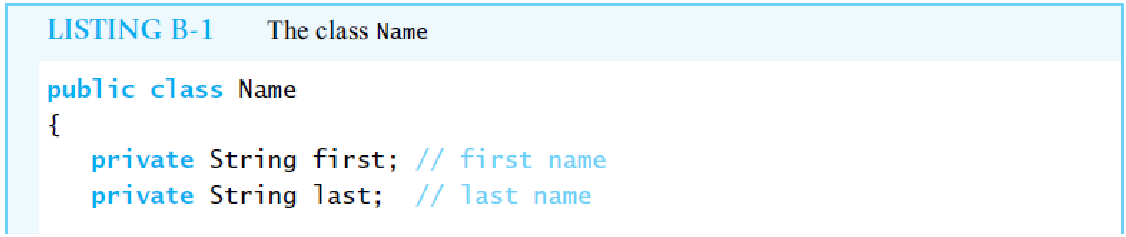
Two strings first and last are called the class’s **data fields.**

The word private means that only the methods within the class can refer to the data fields by their names first and last.

- “Name.java” stores a class definition in a file whose name is the name of the class followed by .java. Typically, you store only one class per file

Review: Java Programming

- Since the data fields are private, we need to define methods in a class that look at or change the values of its data fields.



LISTING B-1 The class Name

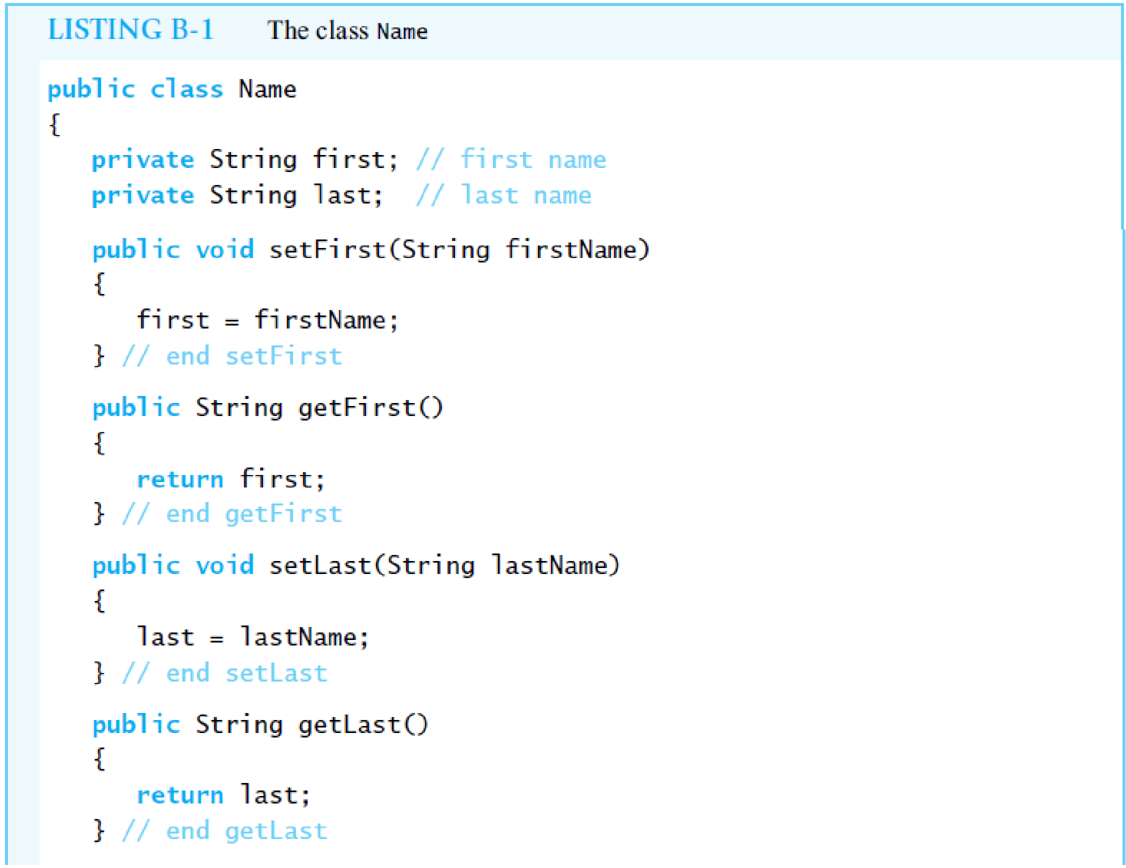
```
public class Name
{
   private String first; // first name
   private String last;  // last name
```

Review: Java Programming

- Since the data fields are private, we need to define methods in a class that look at or change the values of its data fields.

- Get methods enable you to look at the value of a data field
  - getFirst()
  - getLast()

- Set methods change the value of a data field
  - setFirst()
  - setLast()

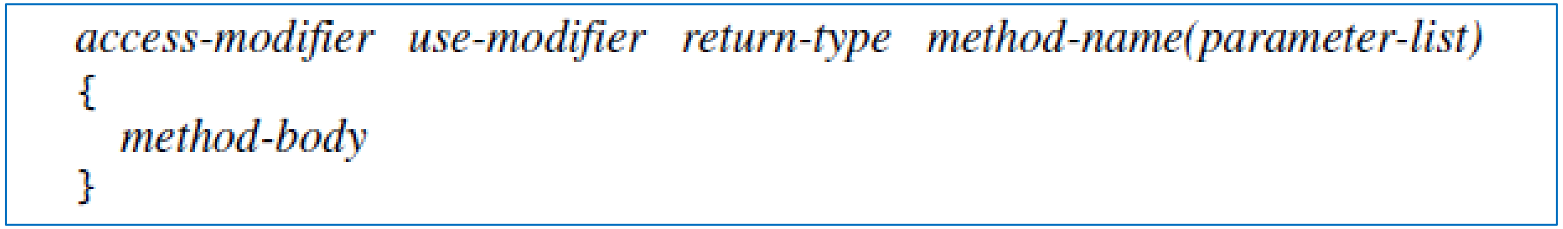


LISTING B-1 The class Name

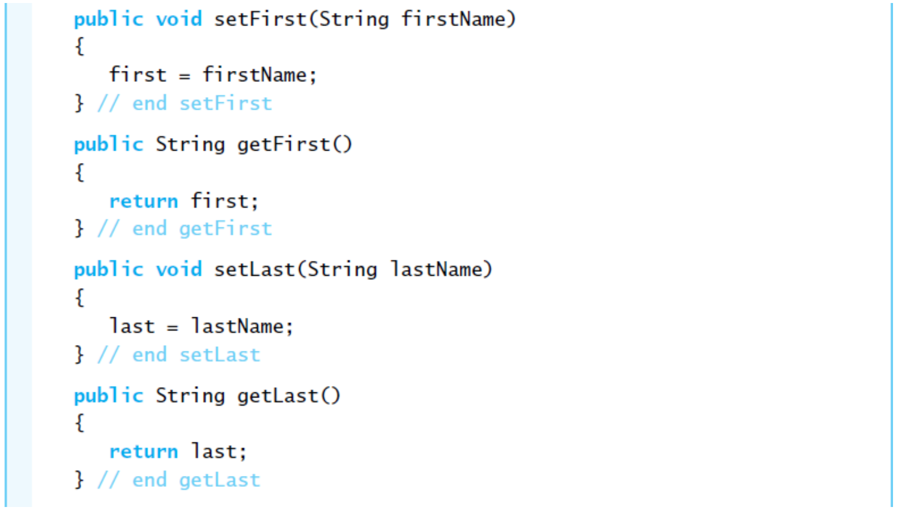
```java
public class Name
{
   private String first; // first name
   private String last;  // last name

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

   public String getFirst()
   {
      return first;
   } // end getFirst

   public void setLast(String lastName)
   {
      last = lastName;
   } // end setLast

   public String getLast()
   {
      return last;
   } // end getLast
```

Review: Java Programming

- The **definition of a method** has the following general form:



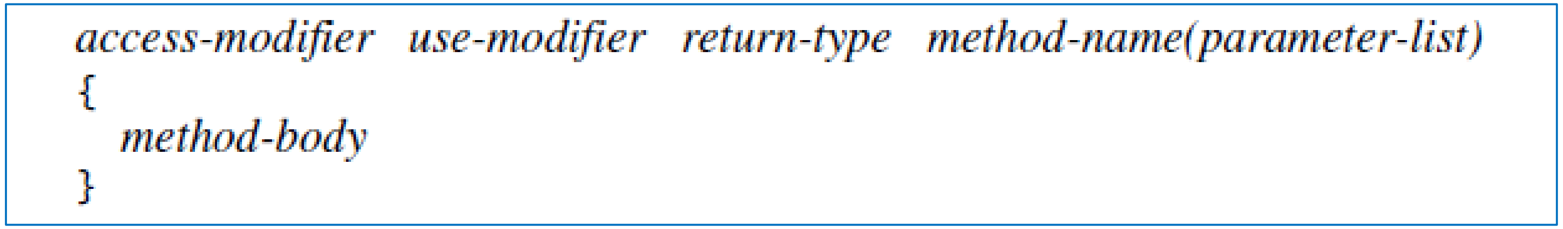
access-modifier   use-modifier   return-type   method-name(parameter-list)
{
  method-body
}



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

public String getFirst()
{
   return first;
} // end getFirst

public void setLast(String lastName)
{
   last = lastName;
} // end setLast

public String getLast()
{
   return last;
} // end getLast
```

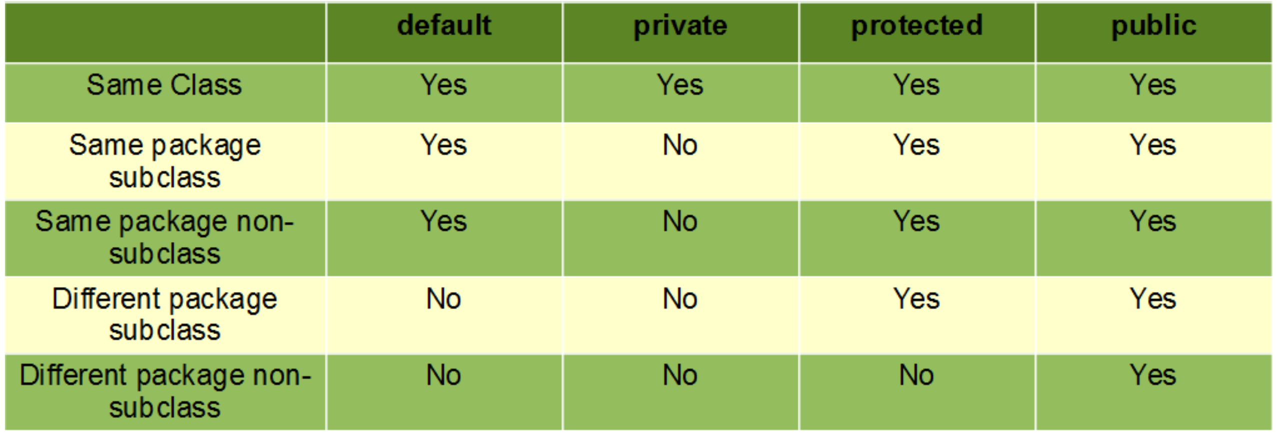
Review: Java Programming

- The **definition of a method** has the following general form:

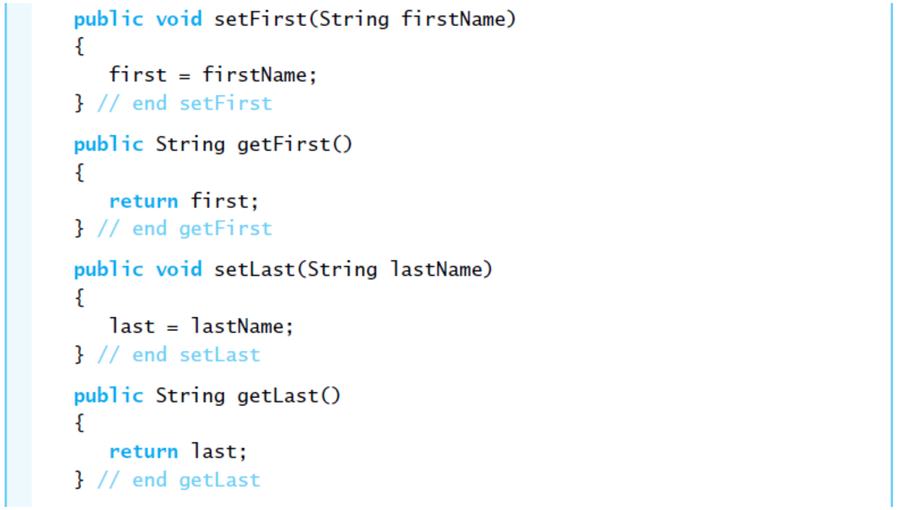


```
access-modifier   use-modifier   return-type   method-name(parameter-list)
{
   method-body
}
```

- **Access modifier** restricts the scope of method



	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

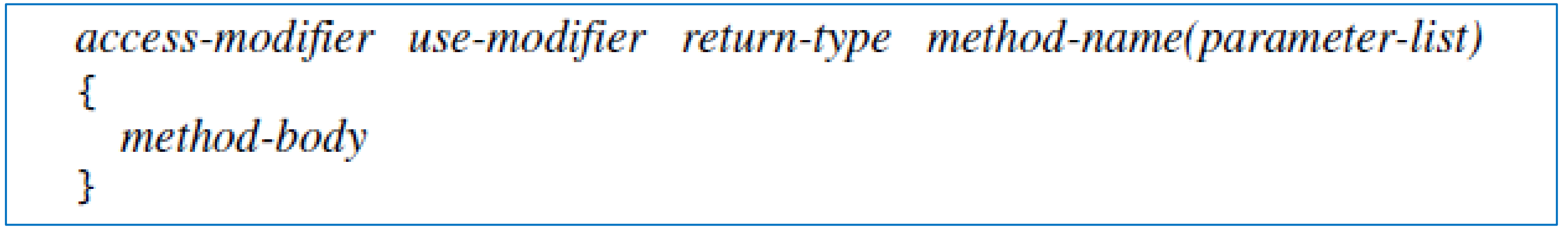


```
public void setFirst(String firstName)
{
   first = firstName;
} // end setFirst

public String getFirst()
{
   return first;
} // end getFirst

public void setLast(String lastName)
{
   last = lastName;
} // end setLast

public String getLast()
{
   return last;
} // end getLast
```

https://www.geeksforgeeks.org/access-modifiers-java/

Review: Java Programming

- The **definition of a method** has the following general form:

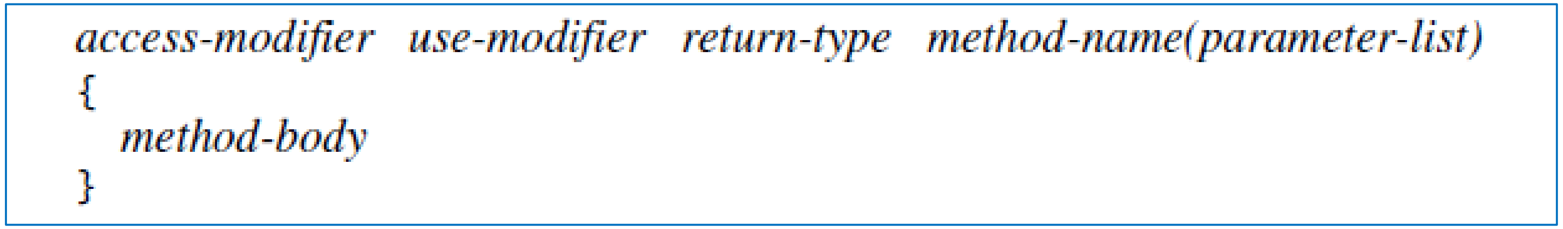


```
access-modifier  use-modifier  return-type  method-name(parameter-list)
{
   method-body
}
```

- **Use modifier** is optional. When used, it can be either abstract, final, or static.
  - An *abstract* method has no definition and must be overridden in a derived class.
  - A final method cannot be overridden in a derived class.
  - A static method is shared by all instances of the class.

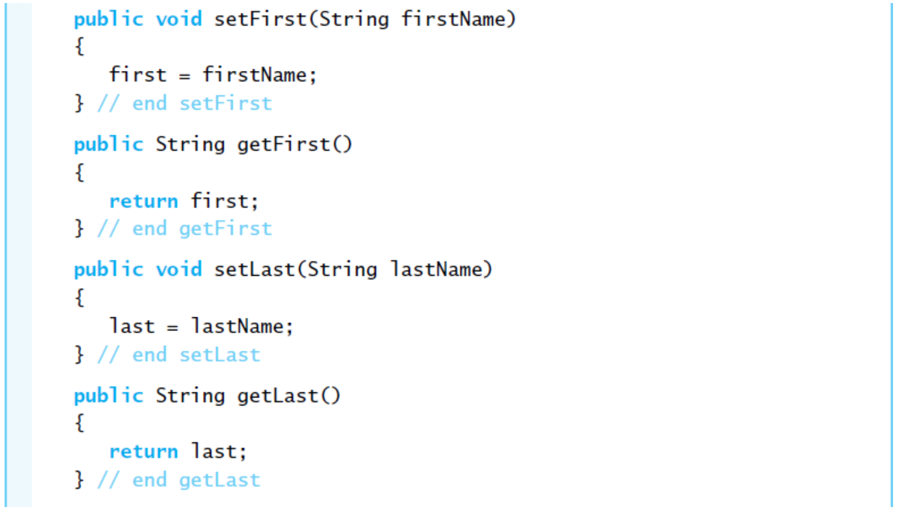
Review: Java Programming

- The **definition of a method** has the following general form:



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

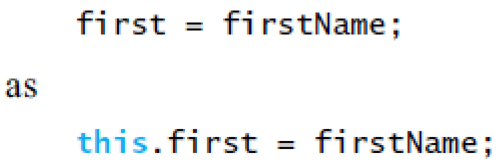
- **Return type**, which for a valued method is the data type of the value that the method returns. For a void method, the return type is void.
- **Parameters**, specify values or objects that are inputs to the method.
- **Body**—which is simply a sequence of Java statements—enclosed in curly braces.



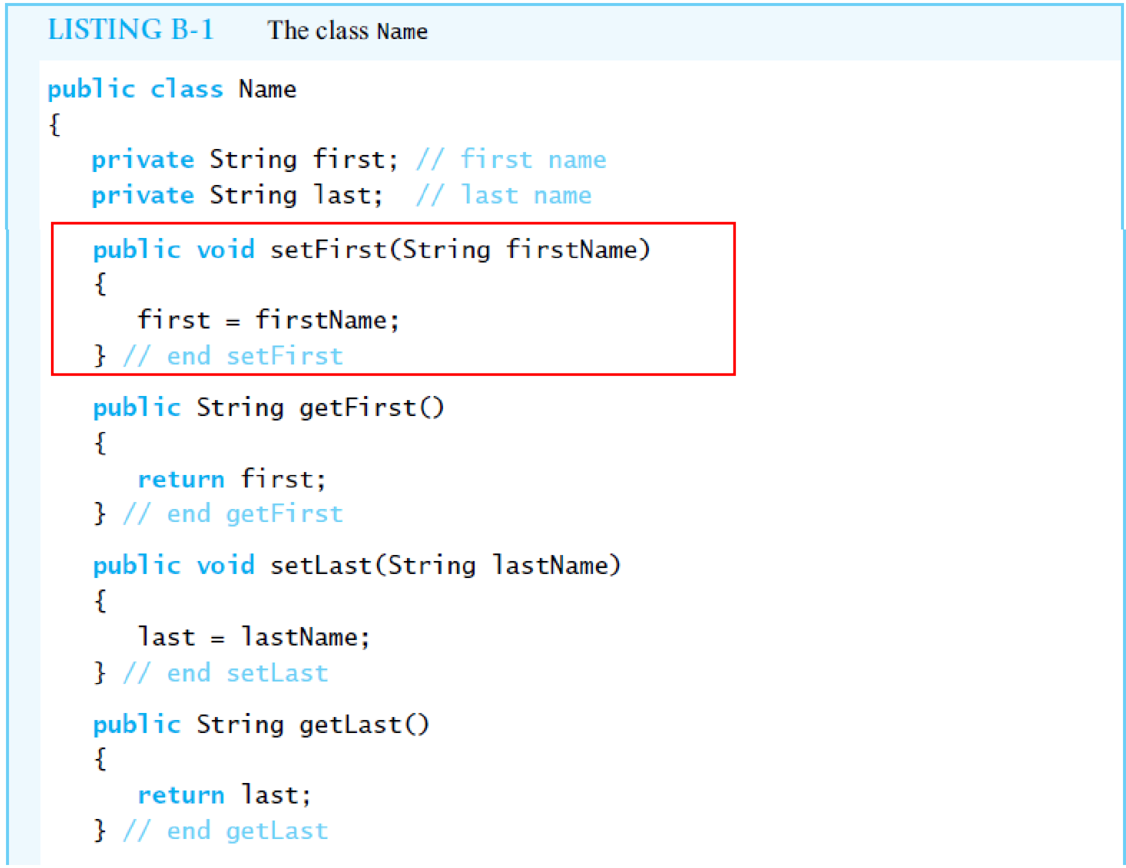
```
public void setFirst(String firstName)
{
   first = firstName;
} // end setFirst

public String getFirst()
{
   return first;
} // end getFirst

public void setLast(String lastName)
{
   last = lastName;
} // end setLast

public String getLast()
{
   return last;
} // end getLast
```

Review: Java Programming

- The object ***this***
  - Given an object, Java has a name for it when you want to refer to the object within the body of a method definition.



```
    first = firstName;
as
    this.first = firstName;
```

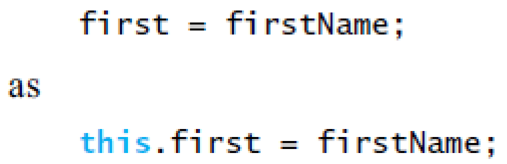


LISTING B-1 The class Name

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

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

   public String getFirst()
   {
      return first;
   } // end getFirst

   public void setLast(String lastName)
   {
      last = lastName;
   } // end setLast

   public String getLast()
   {
      return last;
   } // end getLast
```

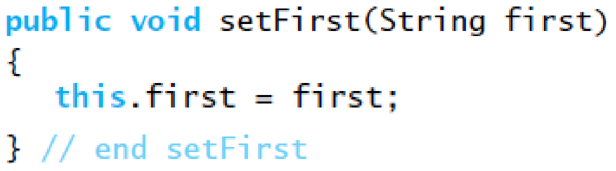
Review: Java Programming

- The object ***this***
  - Given an object, Java has a name for it when you want to refer to the object within the body of a method definition.

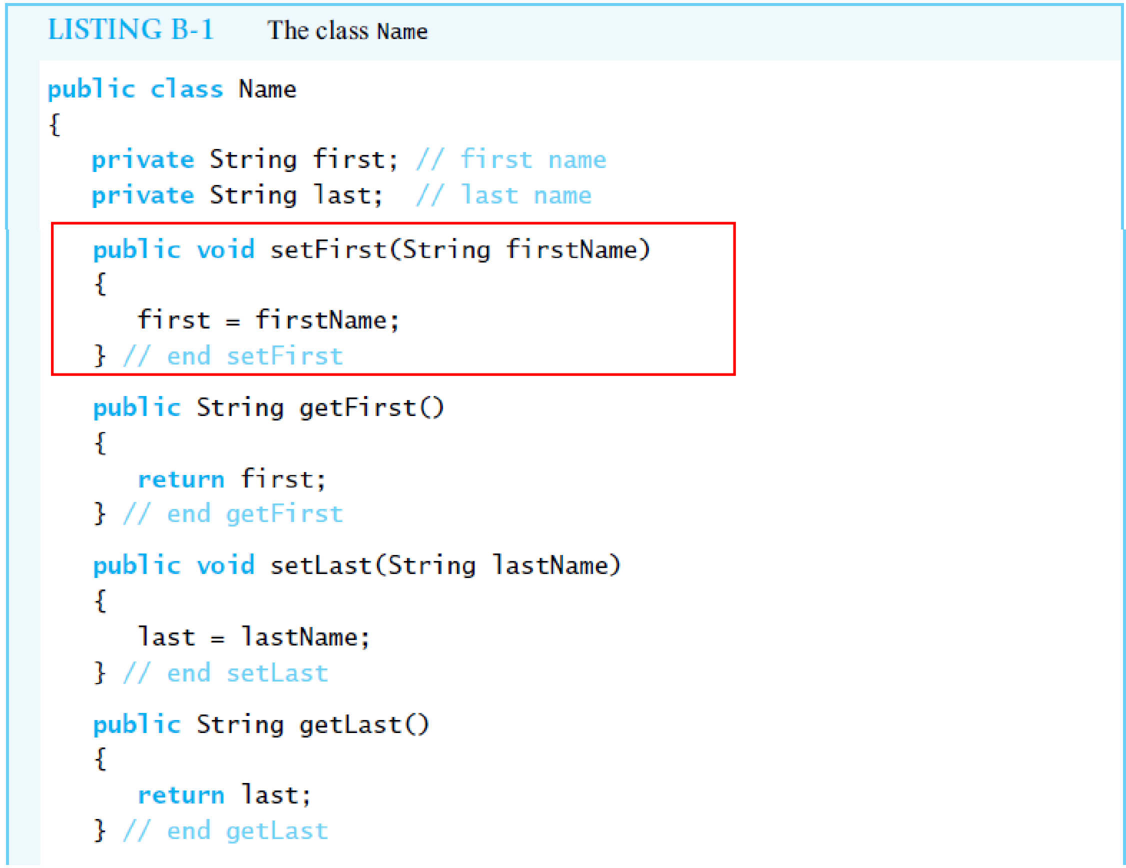


```
        first = firstName;
as
        this.first = firstName;
```

- Note: if the parameter is set to *first*



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



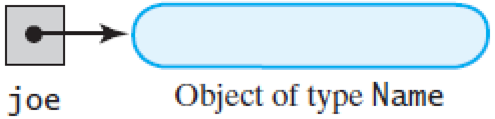
LISTING B-1   The class Name

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

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

   public String getFirst()
   {
      return first;
   } // end getFirst

   public void setLast(String lastName)
   {
      last = lastName;
   } // end setLast

   public String getLast()
   {
      return last;
   } // end getLast
```

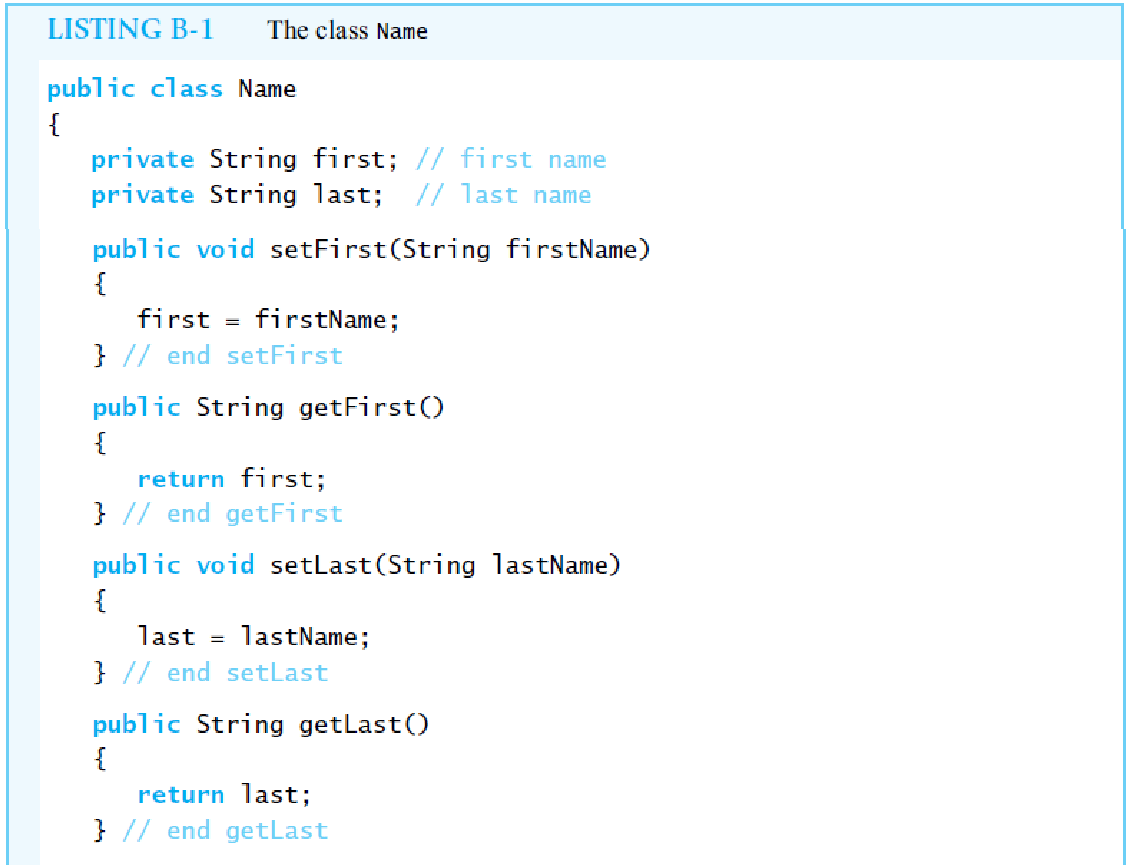
Review: Java Programming

- Create an object and use a variable that references the object

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



- The new operator creates an instance of Name by invoking a special method within the class, known as a **constructor**.
- The memory address of the new object is assigned to joe.



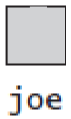
LISTING B-1 The class Name

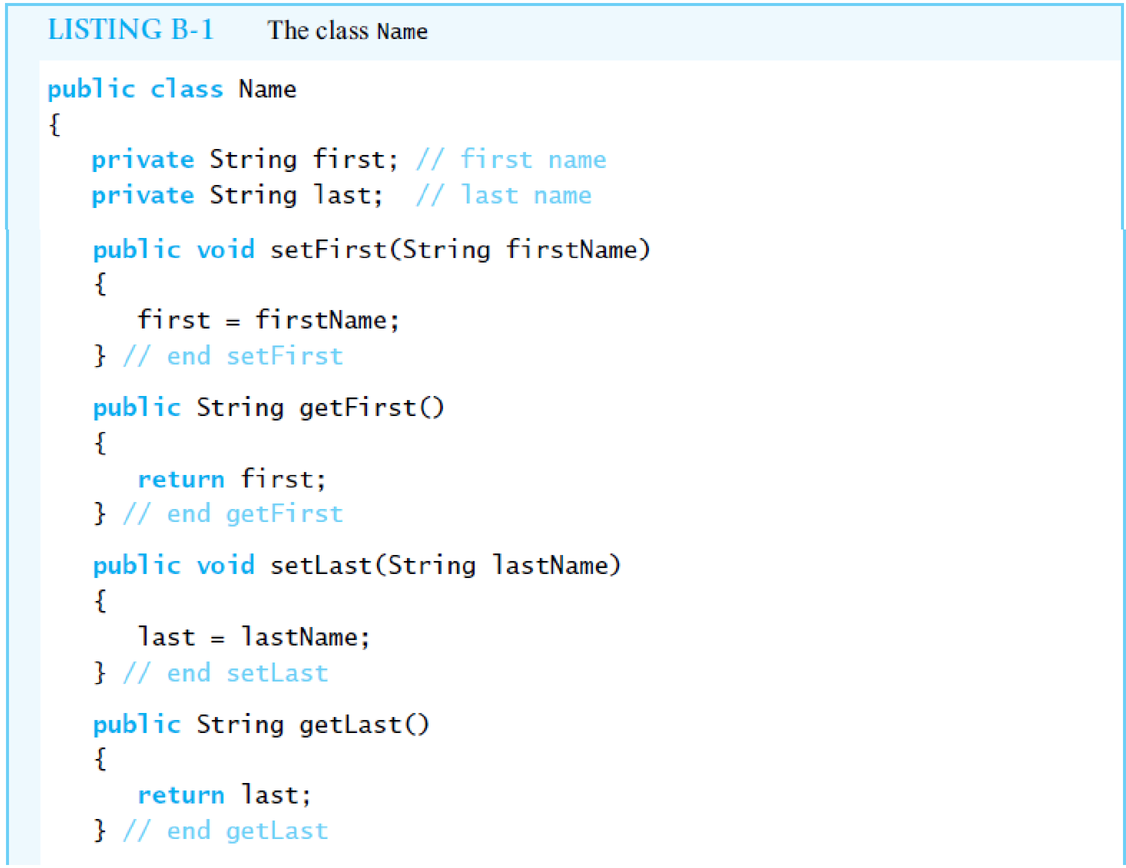
```
public class Name
{
   private String first; // first name
   private String last;  // last name

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

   public String getFirst()
   {
      return first;
   } // end getFirst

   public void setLast(String lastName)
   {
      last = lastName;
   } // end setLast

   public String getLast()
   {
      return last;
   } // end getLast
```

Review: Java Programming

- Note that in the following case, the variable joe contains nothing in particular; it is uninitialized.

```
Name joe;
```



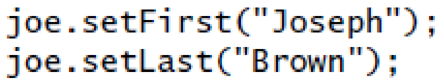


LISTING B-1 The class Name

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

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

   public String getFirst()
   {
      return first;
   } // end getFirst

   public void setLast(String lastName)
   {
      last = lastName;
   } // end setLast

   public String getLast()
   {
      return last;
   } // end getLast
```

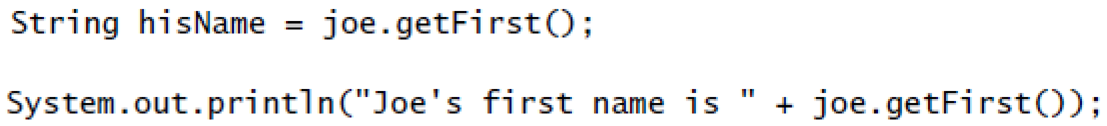
Review: Java Programming

- **Calling Set methods**

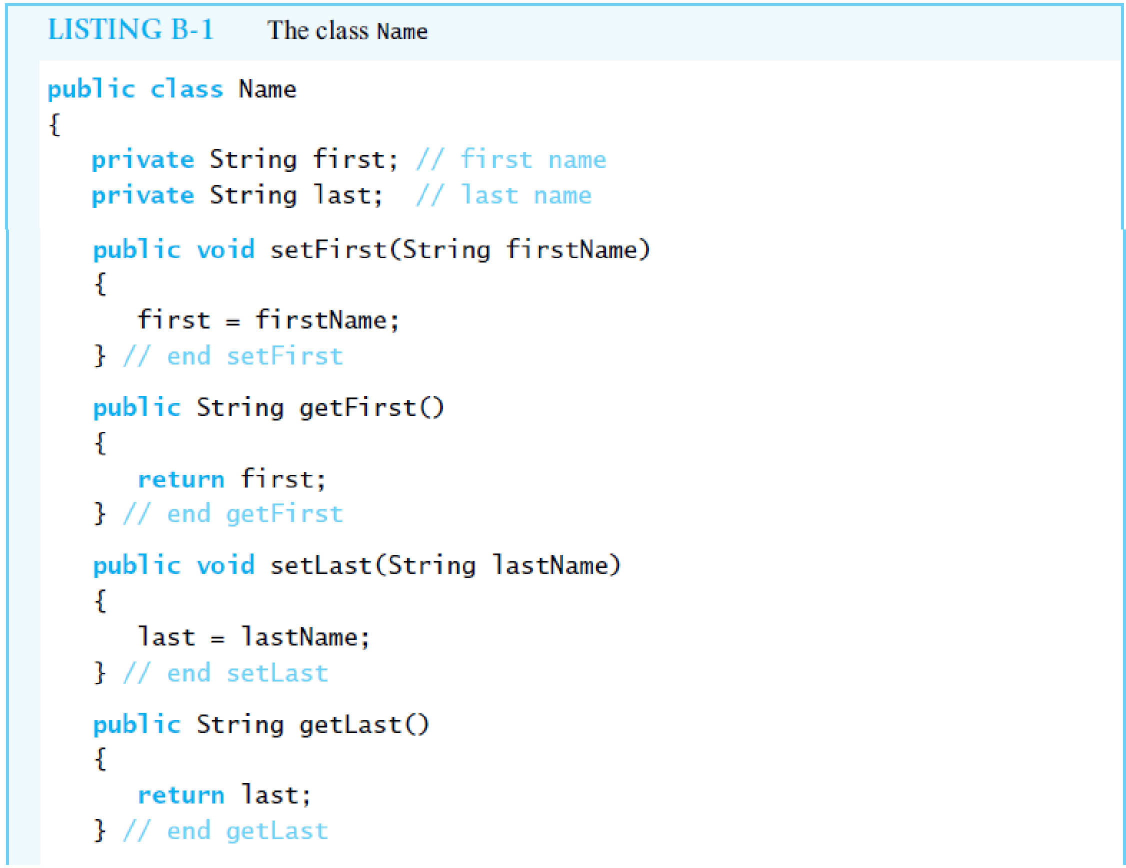


```
joe.setFirst("Joseph");
joe.setLast("Brown");
```

- **Calling Get methods**



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

System.out.println("Joe's first name is " + joe.getFirst());
```

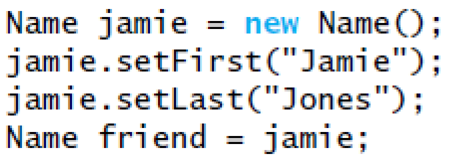


LISTING B-1 The class Name

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

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

   public String getFirst()
   {
      return first;
   } // end getFirst

   public void setLast(String lastName)
   {
      last = lastName;
   } // end setLast

   public String getLast()
   {
      return last;
   } // end getLast
```

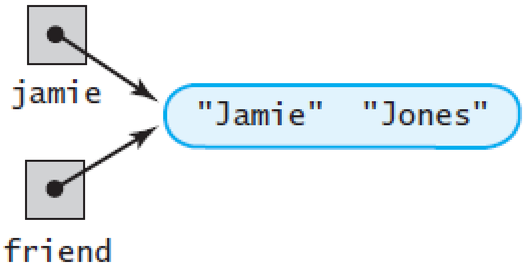
Review: Java Programming

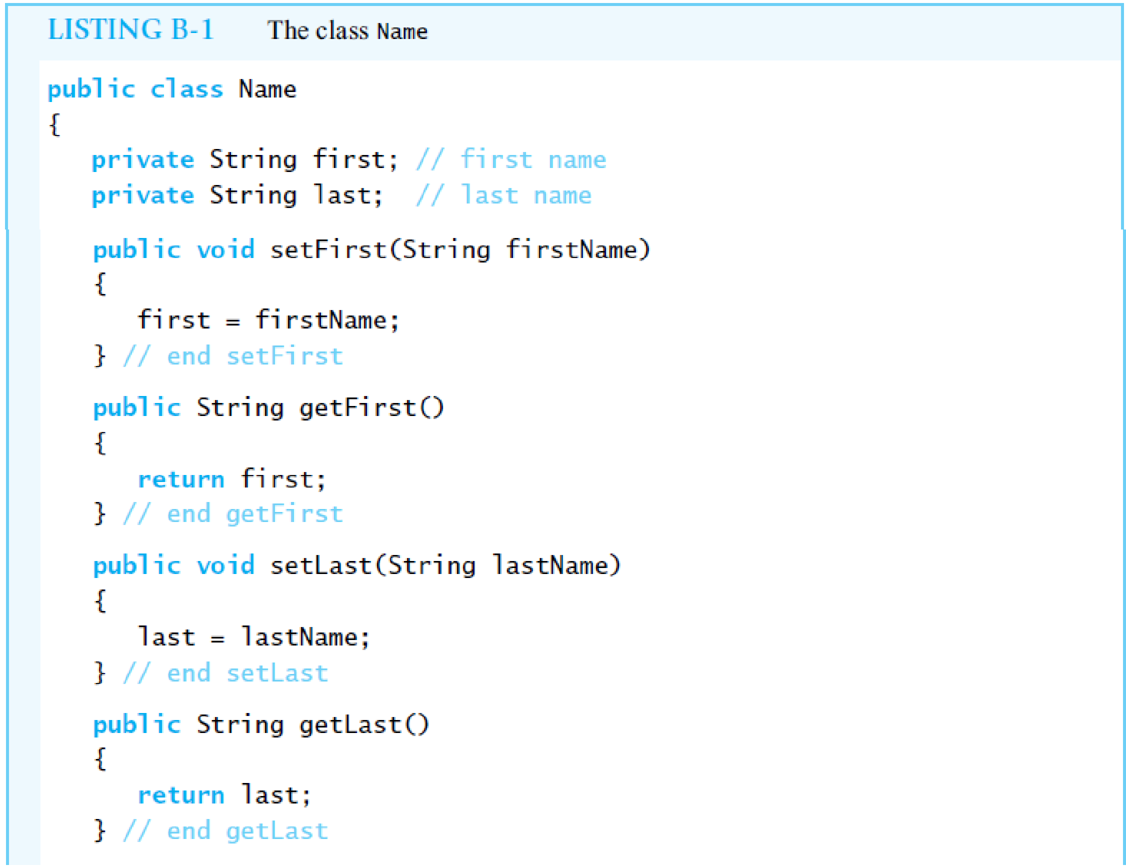
- **References and Aliases**



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

- The two variables *jamie* and *friend* reference the same instance of Name





LISTING B-1 The class Name

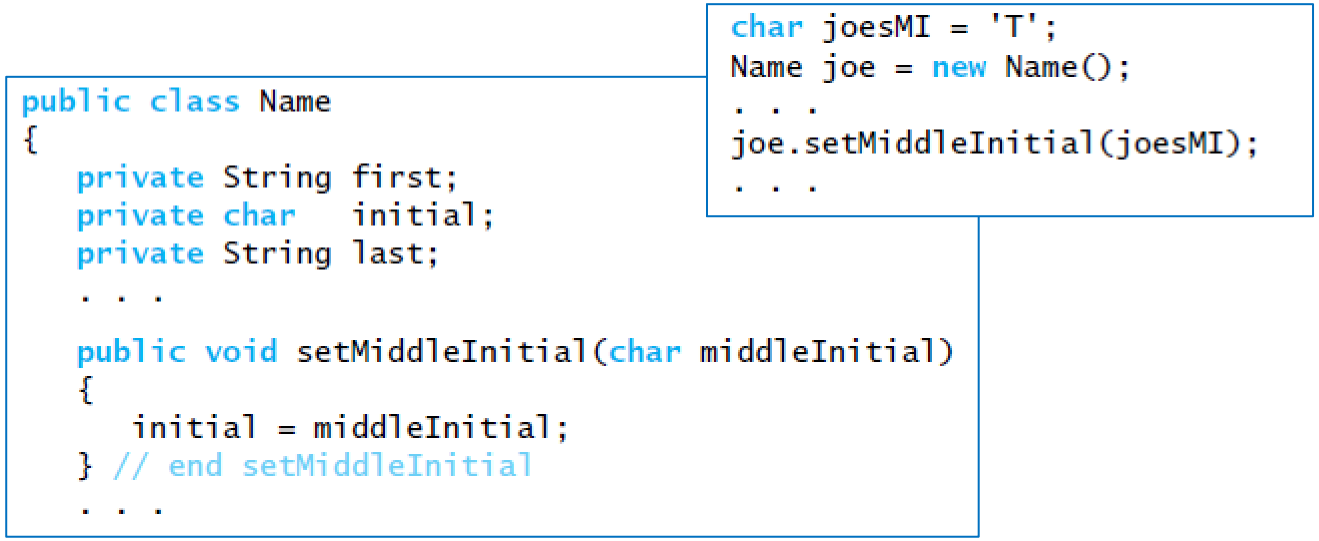
```
public class Name
{
   private String first; // first name
   private String last;  // last name

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

   public String getFirst()
   {
      return first;
   } // end getFirst

   public void setLast(String lastName)
   {
      last = lastName;
   } // end setLast

   public String getLast()
   {
      return last;
   } // end getLast
```

Review: Java Programming

- **Passing Arguments**
  - Call by Value
  - Call by Reference

Review: Java Programming

- **Call by Value**
  - When a formal parameter has a primitive type, such as int or char, the parameter is initialized to the value of the corresponding argument in the method invocation.

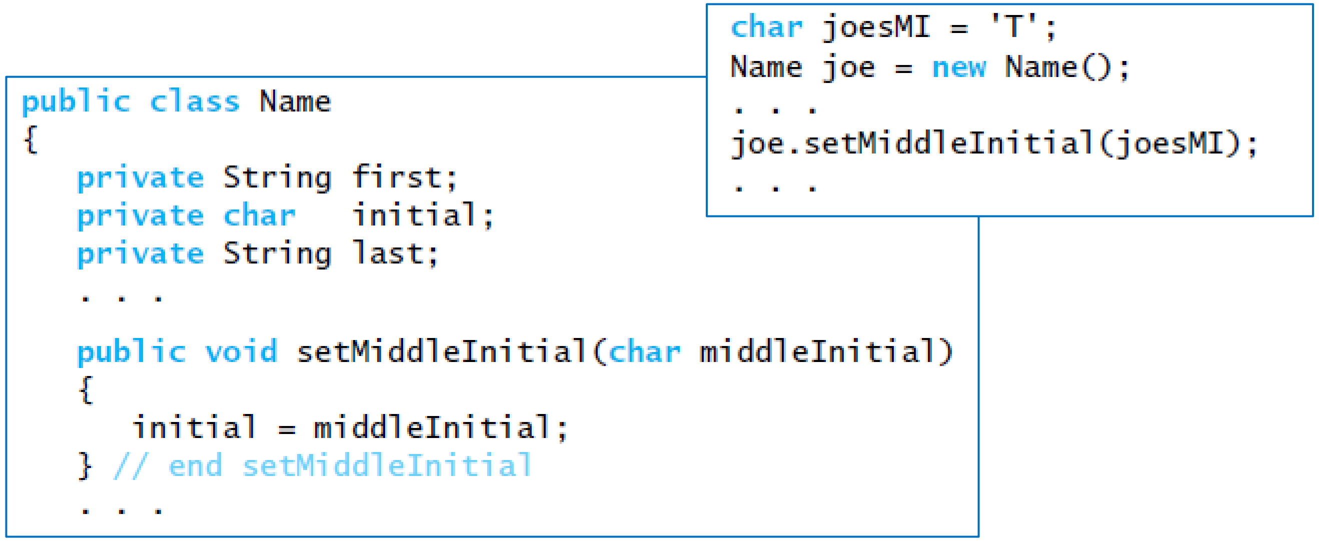


```
char joesMI = 'T';
Name joe = new Name();
. . .
joe.setMiddleInitial(joesMI);
. . .
```

```
public class Name
{
   private String first;
   private char   initial;
   private String last;
   . . .

   public void setMiddleInitial(char middleInitial)
   {
      initial = middleInitial;
   } // end setMiddleInitial
   . . .
```

Review: Java Programming

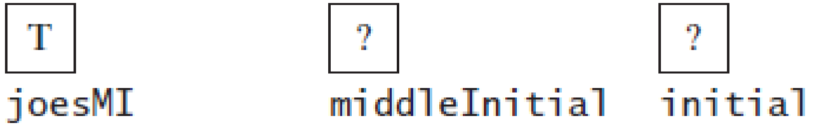
- **Call by Value**
  - When a formal parameter has a primitive type, such as int or char, the parameter is initialized to the value of the corresponding argument in the method invocation.



```
char joesMI = 'T';
Name joe = new Name();
. . .
joe.setMiddleInitial(joesMI);
. . .
```

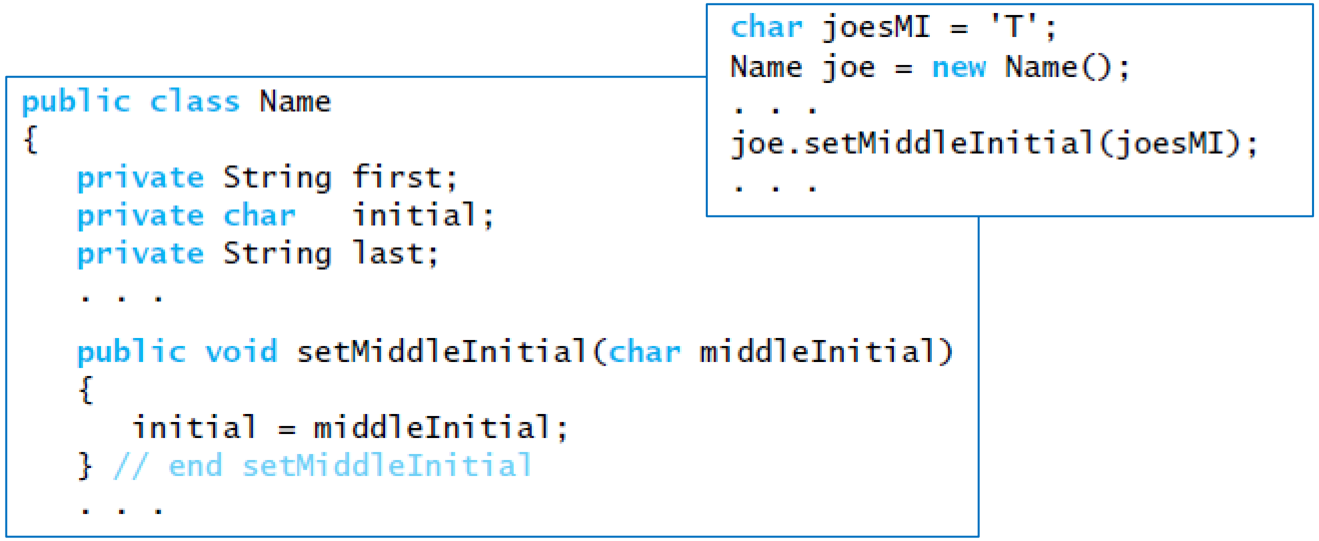
```
public class Name
{
   private String first;
   private char   initial;
   private String last;
   . . .
   public void setMiddleInitial(char middleInitial)
   {
      initial = middleInitial;
   } // end setMiddleInitial
   . . .
```

(a) Before calling setMiddleInitial



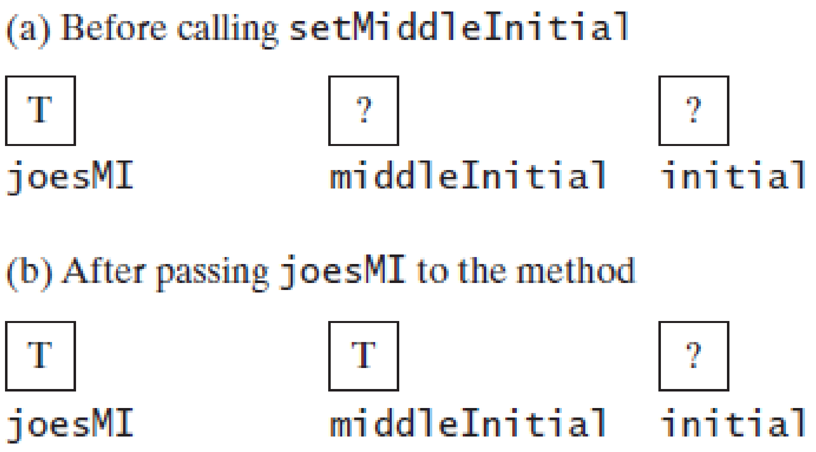
Review: Java Programming

- **Call by Value**
  - When a formal parameter has a primitive type, such as int or char, the parameter is initialized to the value of the corresponding argument in the method invocation.



```
char joesMI = 'T';
Name joe = new Name();
. . .
joe.setMiddleInitial(joesMI);
. . .
```

```
public class Name
{
   private String first;
   private char   initial;
   private String last;
   . . .

   public void setMiddleInitial(char middleInitial)
   {
      initial = middleInitial;
   } // end setMiddleInitial
   . . .
```

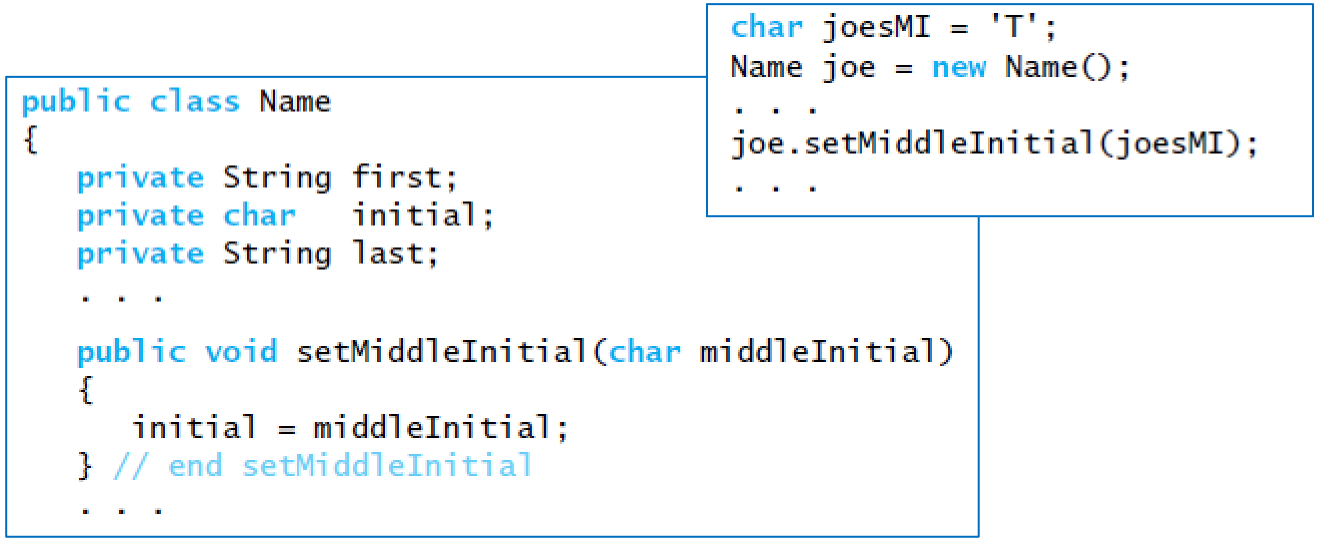


(a) Before calling setMiddleInitial

(b) After passing joesMI to the method

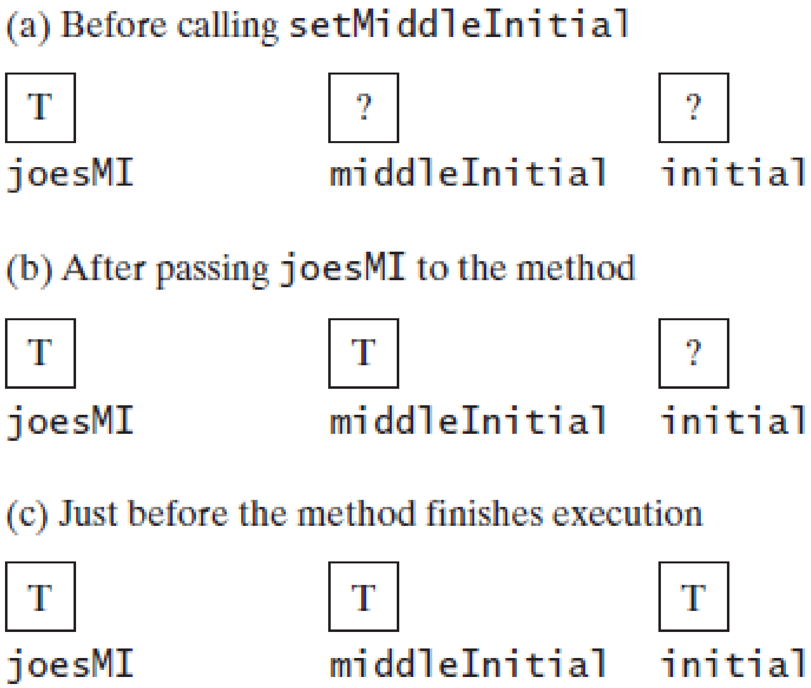
Review: Java Programming

- **Call by Value**
  - When a formal parameter has a primitive type, such as int or char, the parameter is initialized to the value of the corresponding argument in the method invocation.



```
public class Name
{
   private String first;
   private char   initial;
   private String last;
   . . .
   public void setMiddleInitial(char middleInitial)
   {
      initial = middleInitial;
   } // end setMiddleInitial
   . . .
```

```
char joesMI = 'T';
Name joe = new Name();
. . .
joe.setMiddleInitial(joesMI);
. . .
```



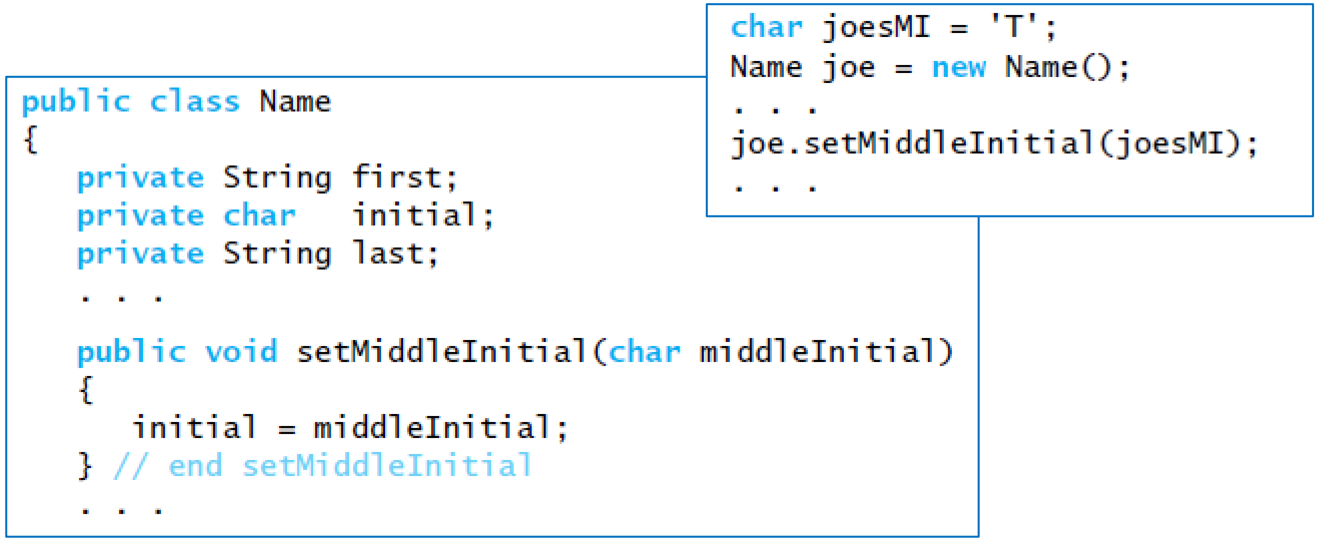
(a) Before calling setMiddleInitial

(b) After passing joesMI to the method

(c) Just before the method finishes execution

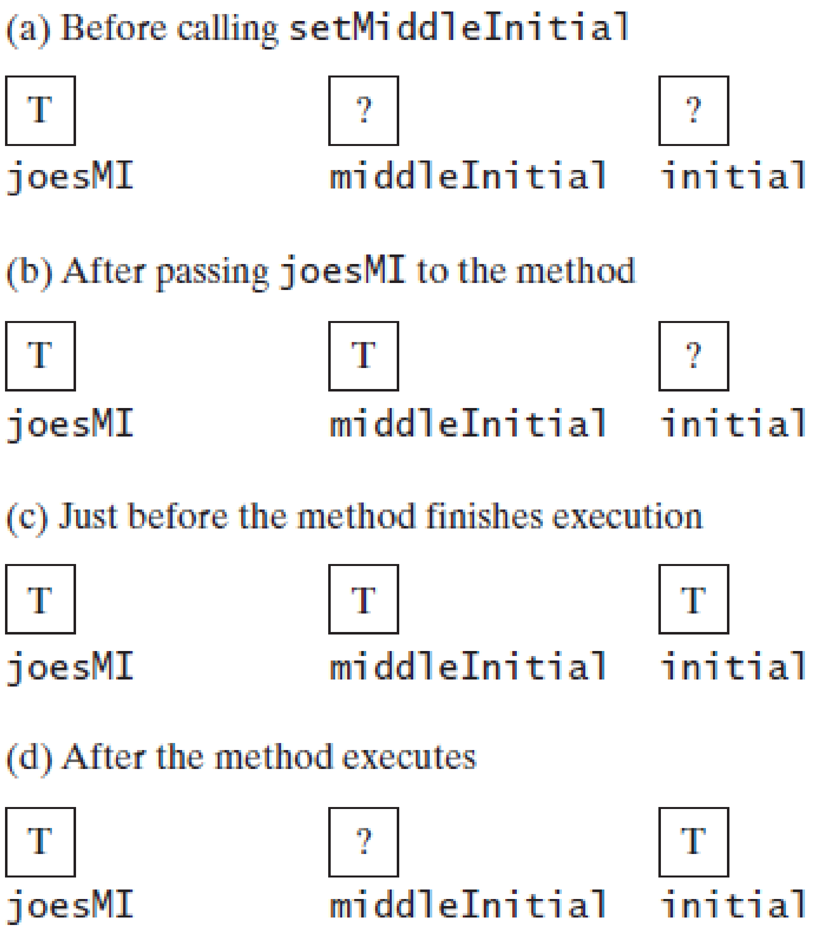
Review: Java Programming

- **Call by Value**
    - When a formal parameter has a primitive type, such as int or char, the parameter is initialized to the value of the corresponding argument in the method invocation.



```
char joesMI = 'T';
Name joe = new Name();
. . .
joe.setMiddleInitial(joesMI);
. . .
```

```
public class Name
{
   private String first;
   private char   initial;
   private String last;
   . . .
   public void setMiddleInitial(char middleInitial)
   {
      initial = middleInitial;
   } // end setMiddleInitial
   . . .
```



(a) Before calling setMiddleInitial

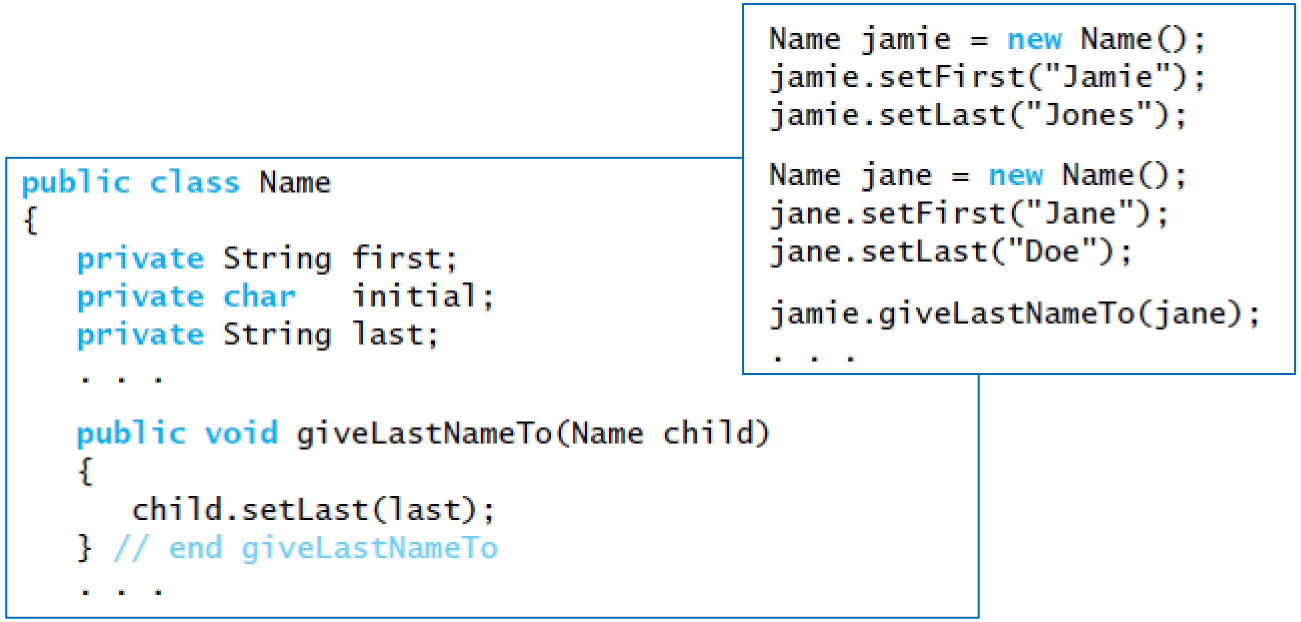
(b) After passing joesMI to the method

(c) Just before the method finishes execution

(d) After the method executes

Review: Java Programming

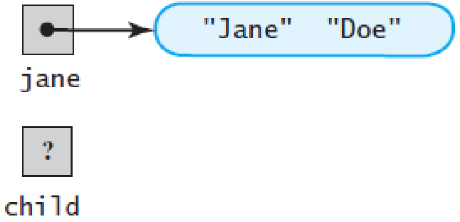
- **Call by Reference**
  - When a formal parameter has a class type, the parameter is initialized to the memory address of that object



```
public class Name
{
   private String first;
   private char   initial;
   private String last;
   . . .

   public void giveLastNameTo(Name child)
   {
      child.setLast(last);
   } // end giveLastNameTo
   . . .
```

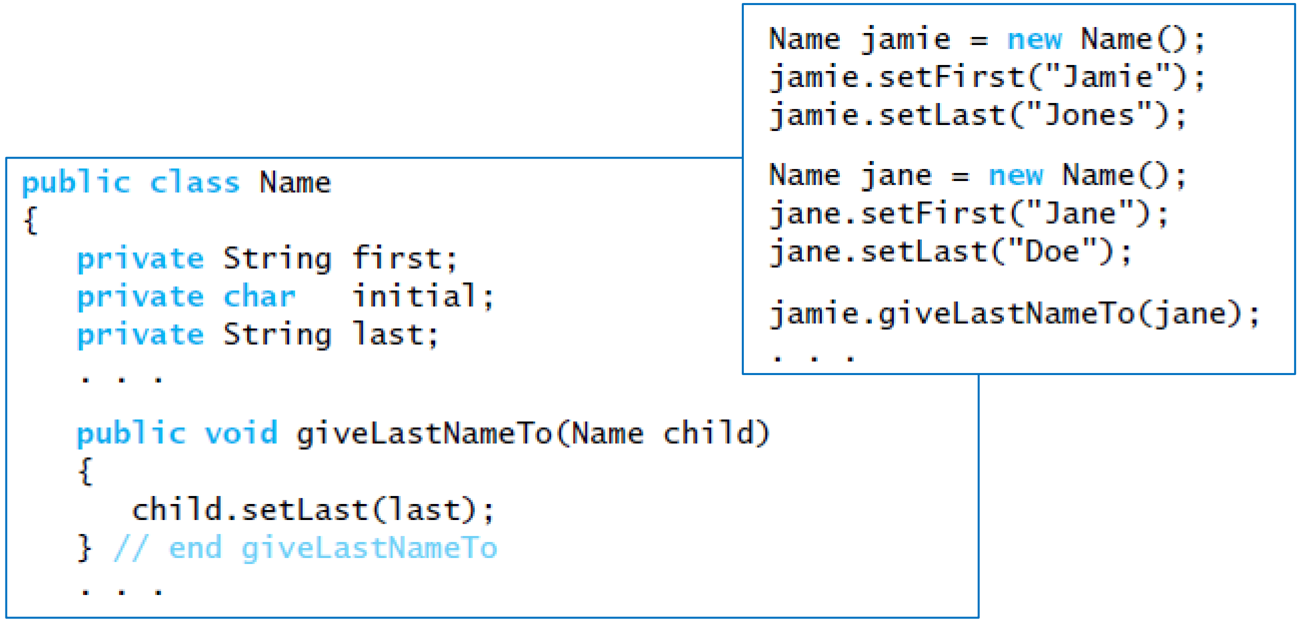
```
Name jamie = new Name();
jamie.setFirst("Jamie");
jamie.setLast("Jones");

Name jane = new Name();
jane.setFirst("Jane");
jane.setLast("Doe");

jamie.giveLastNameTo(jane);
. . .
```

Review: Java Programming



(a) Before calling the method giveLastNameTo

- **Call by Reference**
  - When a formal parameter has a class type, the parameter is initialized to the memory address of that object

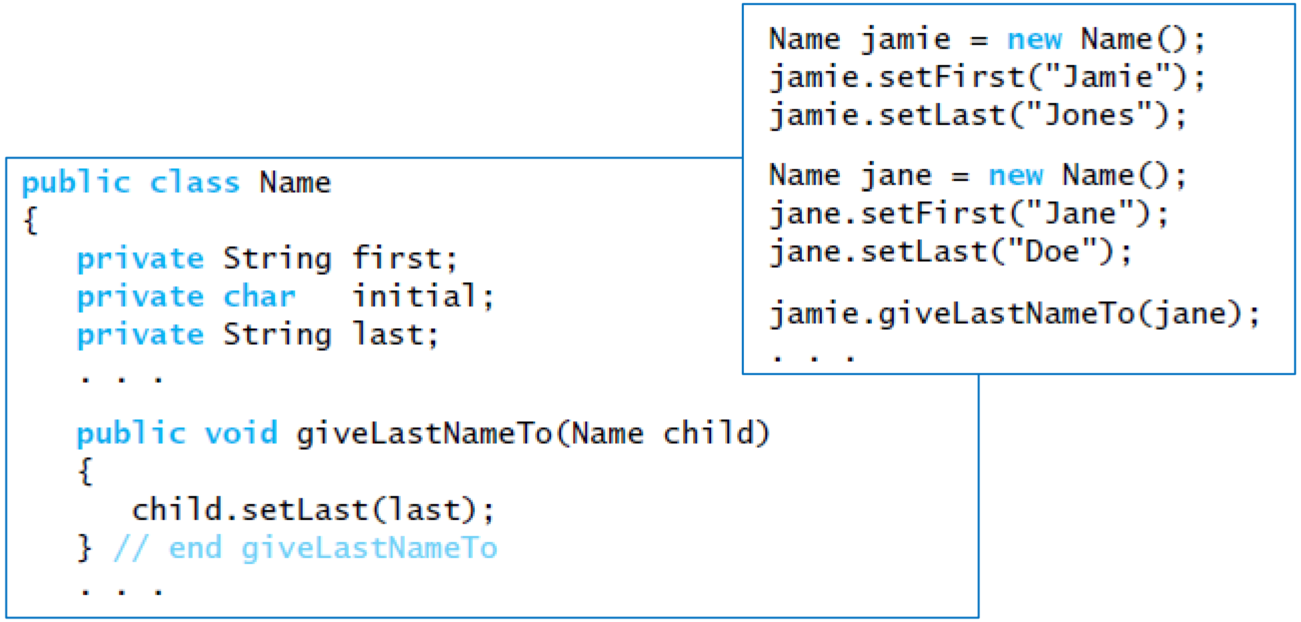


```java
Name jamie = new Name();
jamie.setFirst("Jamie");
jamie.setLast("Jones");

Name jane = new Name();
jane.setFirst("Jane");
jane.setLast("Doe");

jamie.giveLastNameTo(jane);
. . .
```

```java
public class Name
{
   private String first;
   private char   initial;
   private String last;
   . . .

   public void giveLastNameTo(Name child)
   {
      child.setLast(last);
   } // end giveLastNameTo
   . . .
```

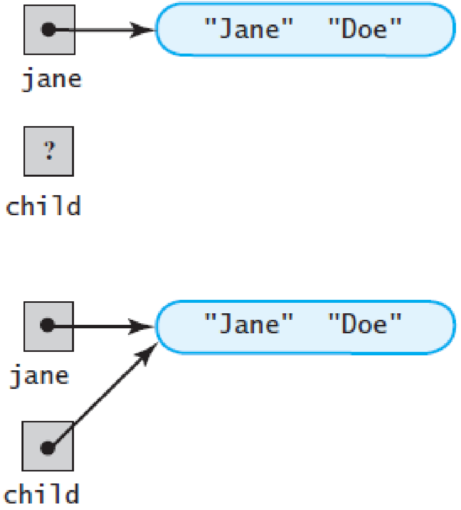
Review: Java Programming

- **Call by Reference**
  - When a formal parameter has a class type, the parameter is initialized to the memory address of that object



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

Name jane = new Name();
jane.setFirst("Jane");
jane.setLast("Doe");

jamie.giveLastNameTo(jane);
. . .
```

```
public class Name
{
   private String first;
   private char   initial;
   private String last;
   . . .

   public void giveLastNameTo(Name child)
   {
      child.setLast(last);
   } // end giveLastNameTo
   . . .
```

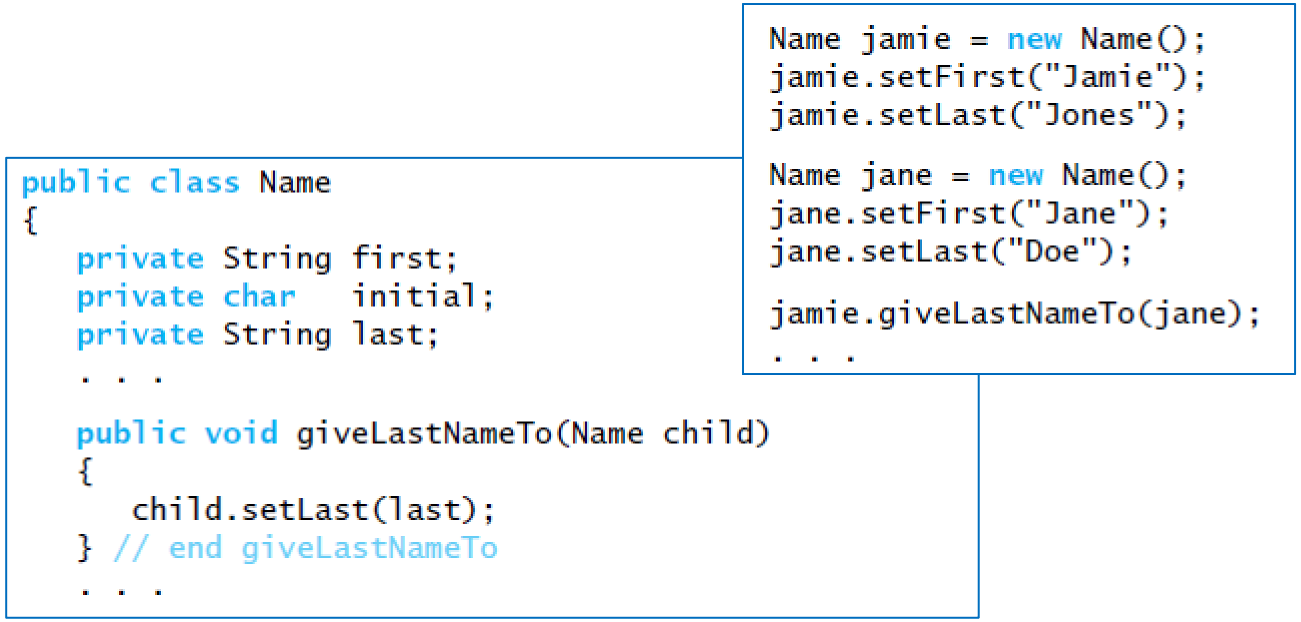


(a) Before calling the method giveLastNameTo

(b) After passing the object jane to the method

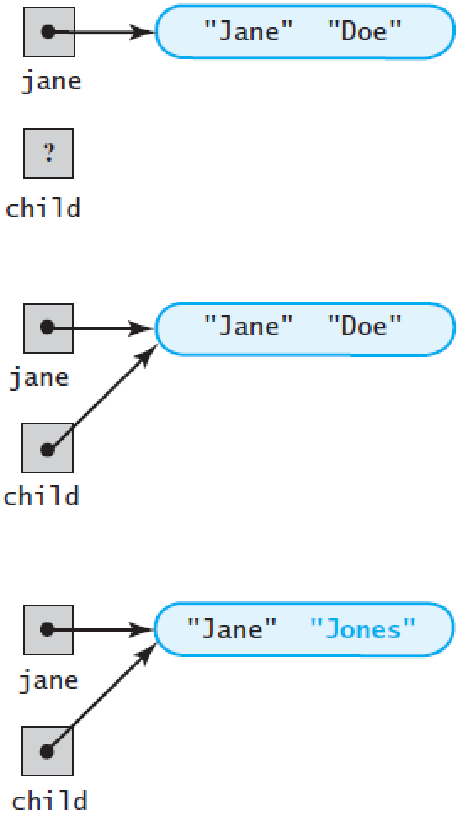
Review: Java Programming

- **Call by Reference**
  - When a formal parameter has a class type, the parameter is initialized to the memory address of that object



```
public class Name
{
   private String first;
   private char   initial;
   private String last;
   . . .
   public void giveLastNameTo(Name child)
   {
      child.setLast(last);
   } // end giveLastNameTo
   . . .
```

```
Name jamie = new Name();
jamie.setFirst("Jamie");
jamie.setLast("Jones");
Name jane = new Name();
jane.setFirst("Jane");
jane.setLast("Doe");
jamie.giveLastNameTo(jane);
. . .
```



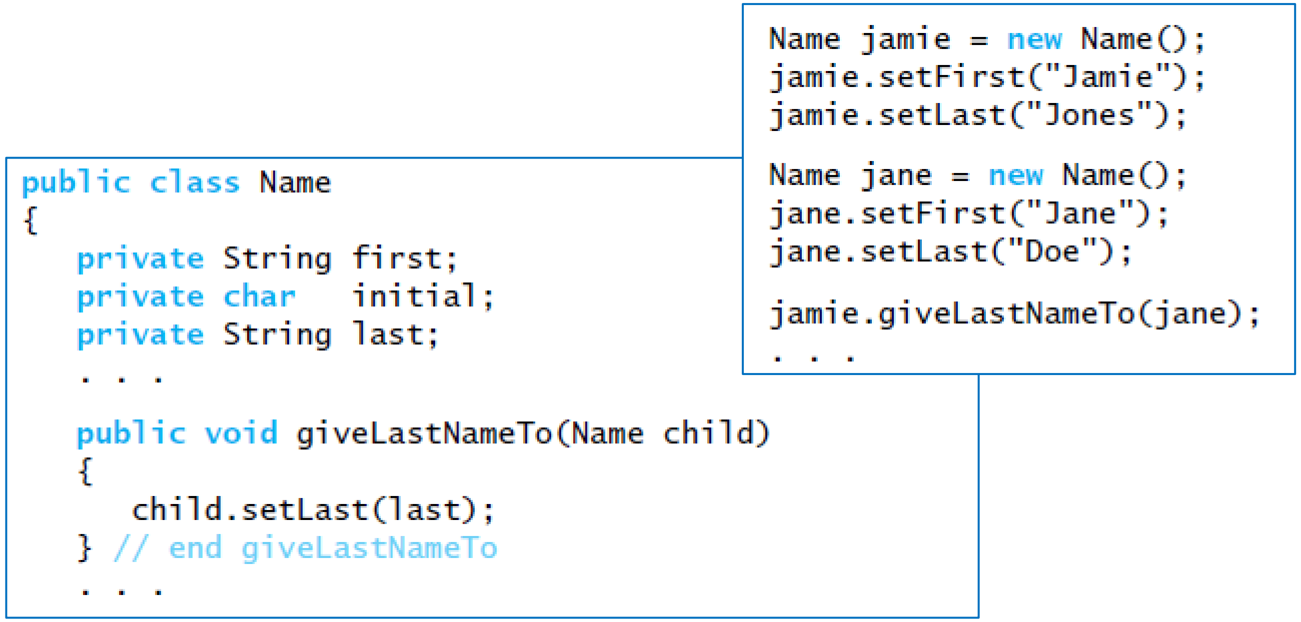
(a) Before calling the method giveLastNameTo

(b) After passing the object jane to the method

(c) Just before the method ends

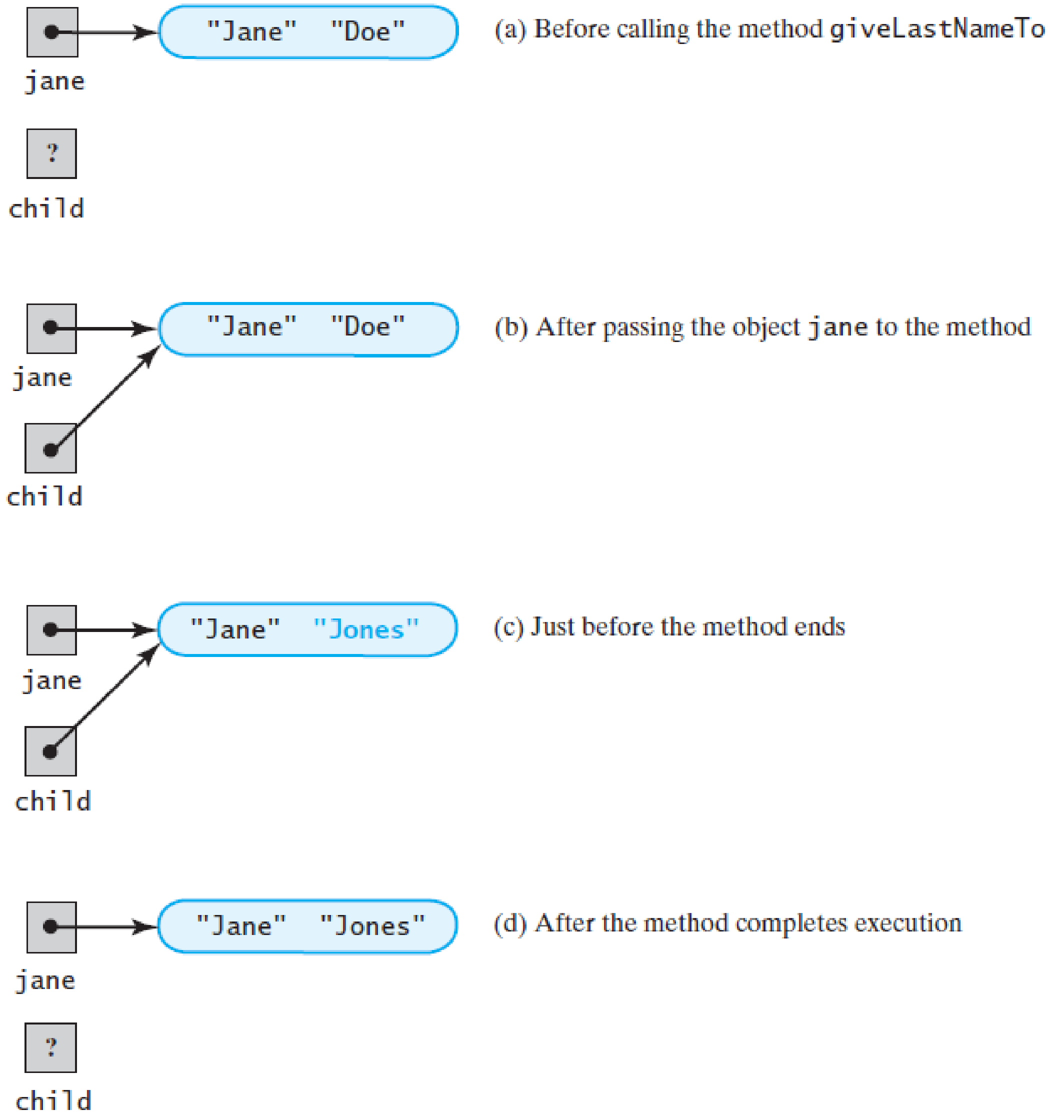
Review: Java Programming

- **Call by Reference**
  - When a formal parameter has a class type, the parameter is initialized to the memory address of that object



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

Name jane = new Name();
jane.setFirst("Jane");
jane.setLast("Doe");

jamie.giveLastNameTo(jane);
. . .
```

```
public class Name
{
   private String first;
   private char   initial;
   private String last;
   . . .

   public void giveLastNameTo(Name child)
   {
      child.setLast(last);
   } // end giveLastNameTo
   . . .
```



(a) Before calling the method giveLastNameTo

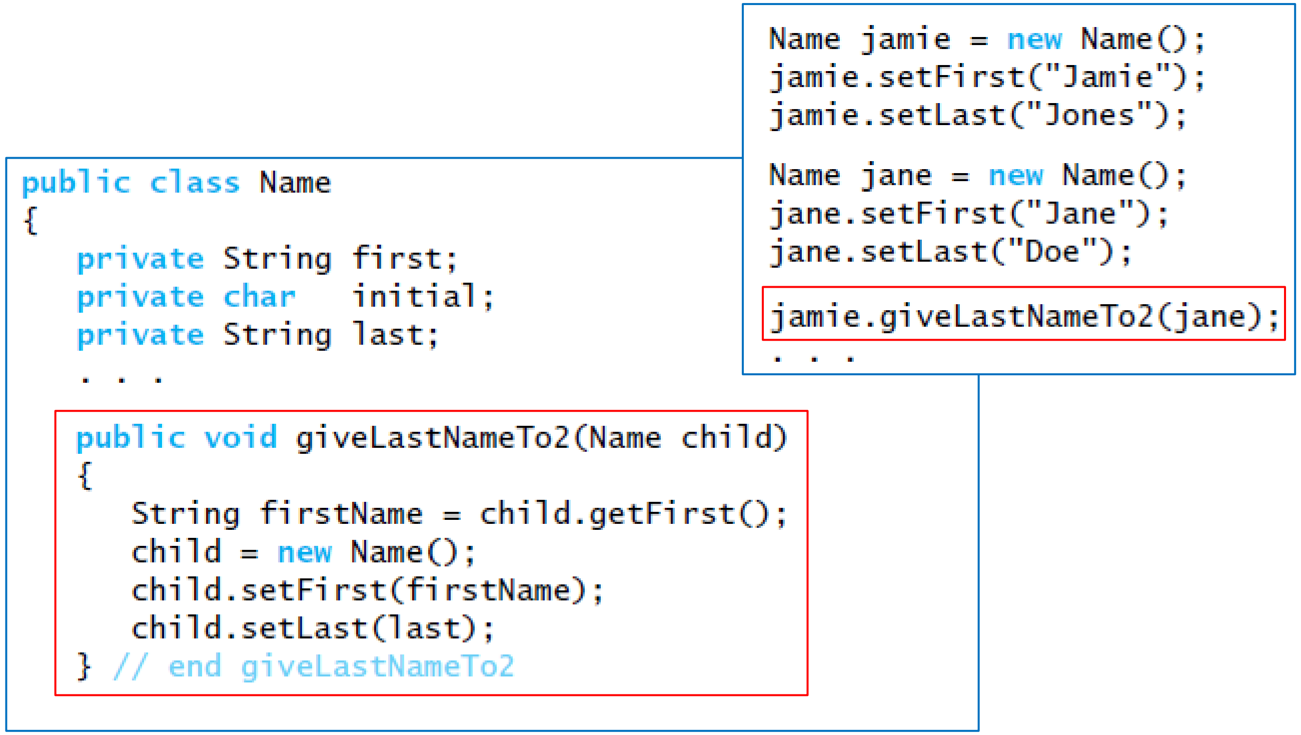
(b) After passing the object jane to the method

(c) Just before the method ends

(d) After the method completes execution

Questions?

- What happens if you change the method definition to allocate a new name, as follows?

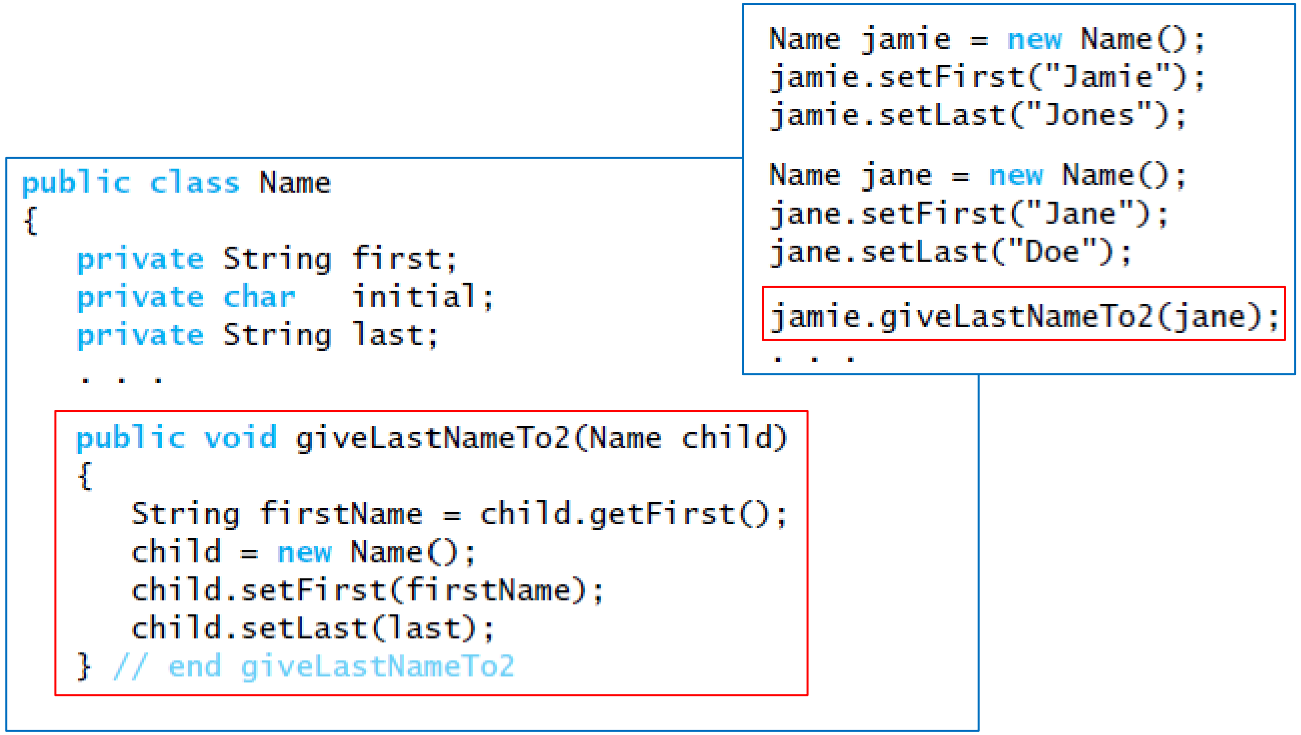


```
public class Name
{
   private String first;
   private char   initial;
   private String last;
   . . .

   public void giveLastNameTo2(Name child)
   {
      String firstName = child.getFirst();
      child = new Name();
      child.setFirst(firstName);
      child.setLast(last);
   } // end giveLastNameTo2
```

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

Name jane = new Name();
jane.setFirst("Jane");
jane.setLast("Doe");

jamie.giveLastNameTo2(jane);
. . .
```

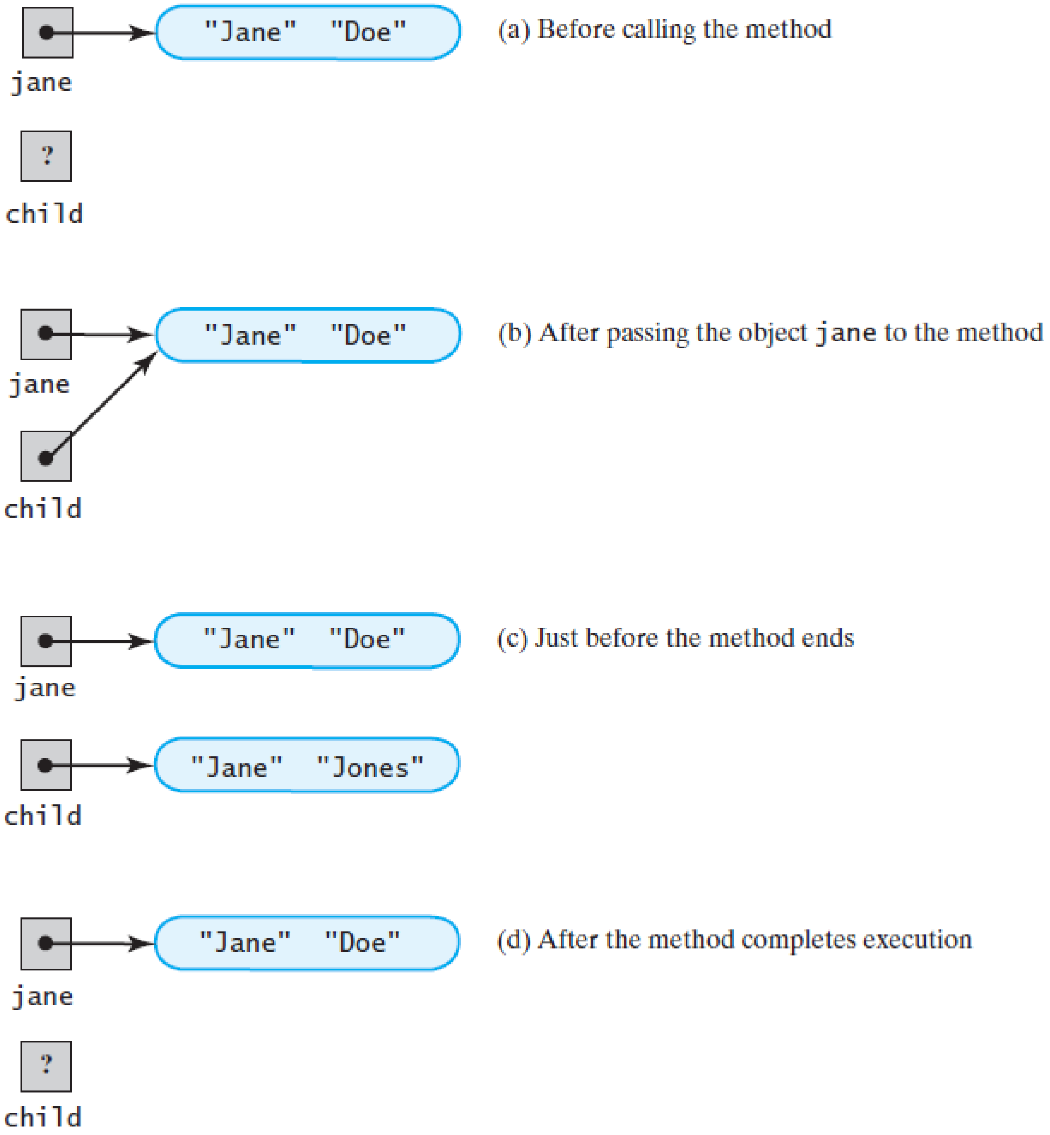
Questions?

- What happens if you change the method definition to allocate a new name, as follows?



```
public class Name
{
   private String first;
   private char   initial;
   private String last;
   . . .

public void giveLastNameTo2(Name child)
{
   String firstName = child.getFirst();
   child = new Name();
   child.setFirst(firstName);
   child.setLast(last);
} // end giveLastNameTo2
```

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

Name jane = new Name();
jane.setFirst("Jane");
jane.setLast("Doe");

jamie.giveLastNameTo2(jane);
. . .
```



(a) Before calling the method

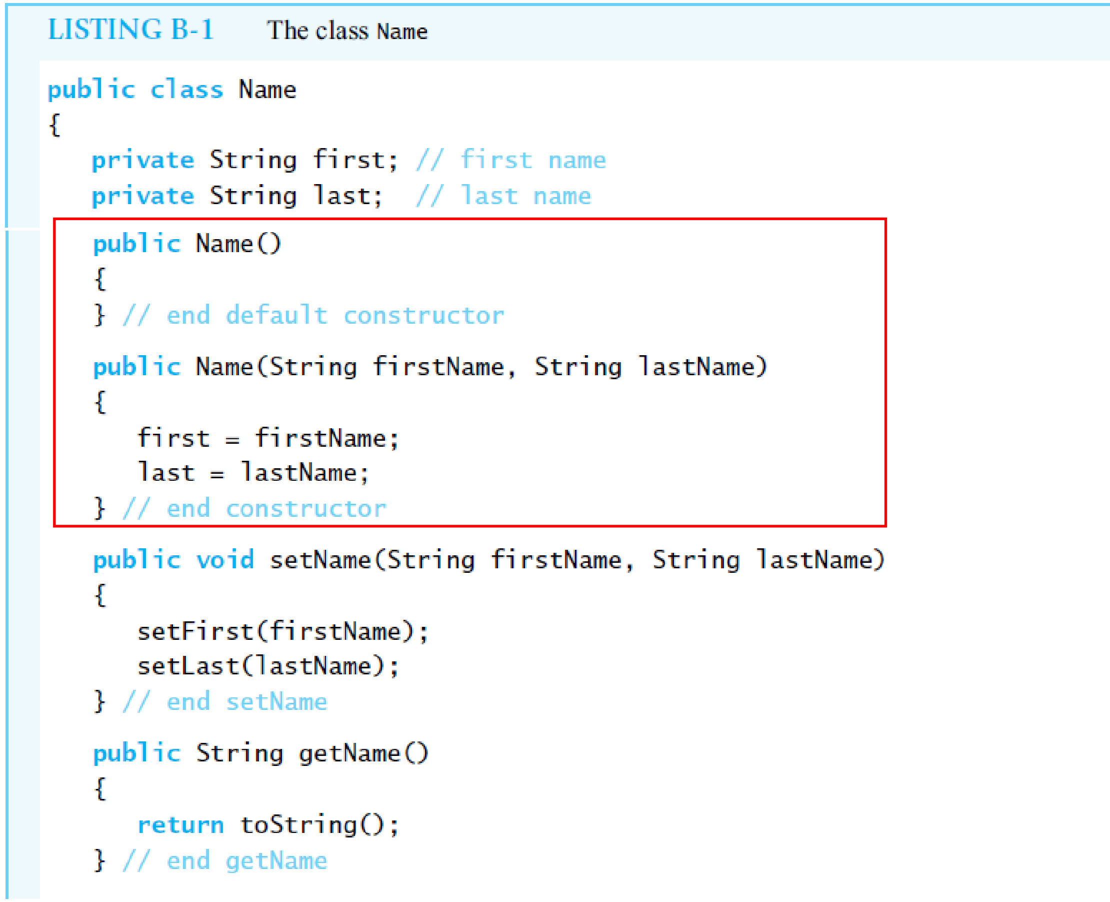
(b) After passing the object jane to the method

(c) Just before the method ends

(d) After the method completes execution

Review: Java Programming

- **Constructors**
  - Creating an object by using the *new* operator to invoke a special method called a constructor, which allocates memory for the object and initializes the data fields.



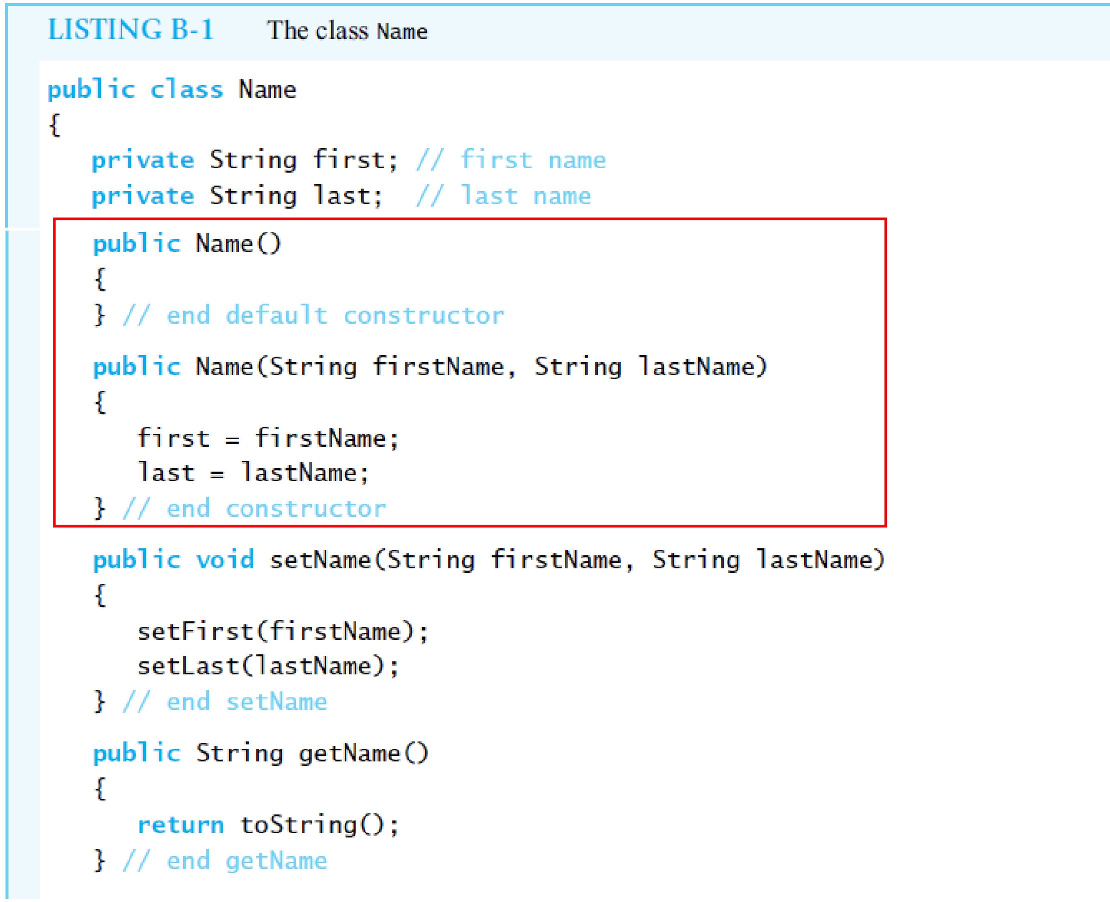
LISTING B-1 The class Name

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

   public Name()
   {
   } // end default constructor

   public Name(String firstName, String lastName)
   {
      first = firstName;
      last = lastName;
   } // end constructor

   public void setName(String firstName, String lastName)
   {
      setFirst(firstName);
      setLast(lastName);
   } // end setName

   public String getName()
   {
      return toString();
   } // end getName
```

Review: Java Programming

- **Constructors**
  - Creating an object by using the *new* operator to invoke a special method called a constructor, which allocates memory for the object and initializes the data fields.

- A **constructor**
  - Has the same name as the class
  - Has no return type, not even void
  - Has any number of formal parameters, including no parameters



LISTING B-1 The class Name

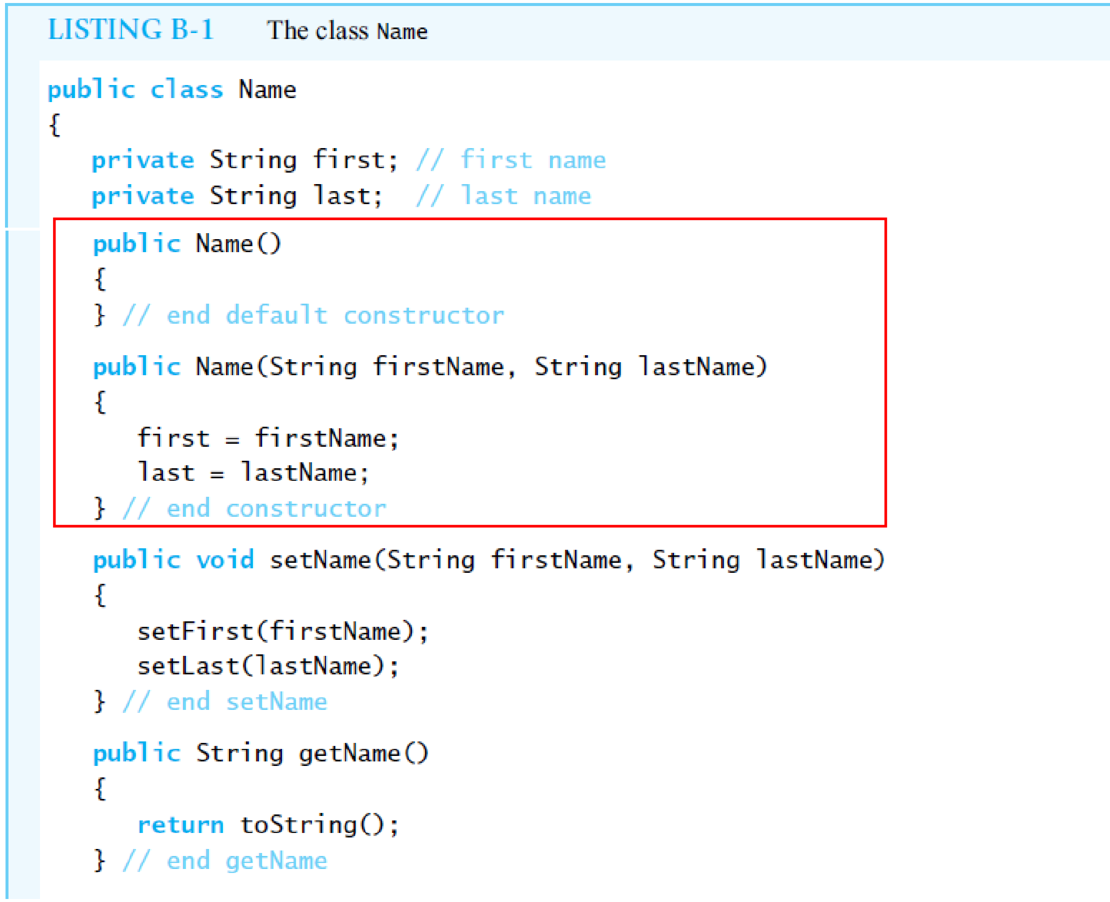
```
public class Name
{
   private String first; // first name
   private String last;  // last name

   public Name()
   {
   } // end default constructor

   public Name(String firstName, String lastName)
   {
      first = firstName;
      last = lastName;
   } // end constructor

   public void setName(String firstName, String lastName)
   {
      setFirst(firstName);
      setLast(lastName);
   } // end setName

   public String getName()
   {
      return toString();
   } // end getName
```

Review: Java Programming

- **Constructors**
  - Creating an object by using the *new* operator to invoke a special method called a constructor, which allocates memory for the object and initializes the data fields.

- A **constructor**
  - Has the same name as the class
  - Has no return type, not even void
  - Has any number of formal parameters, including no parameters

- A class can have several constructors that differ in the number or type of parameters.



LISTING B-1 The class Name

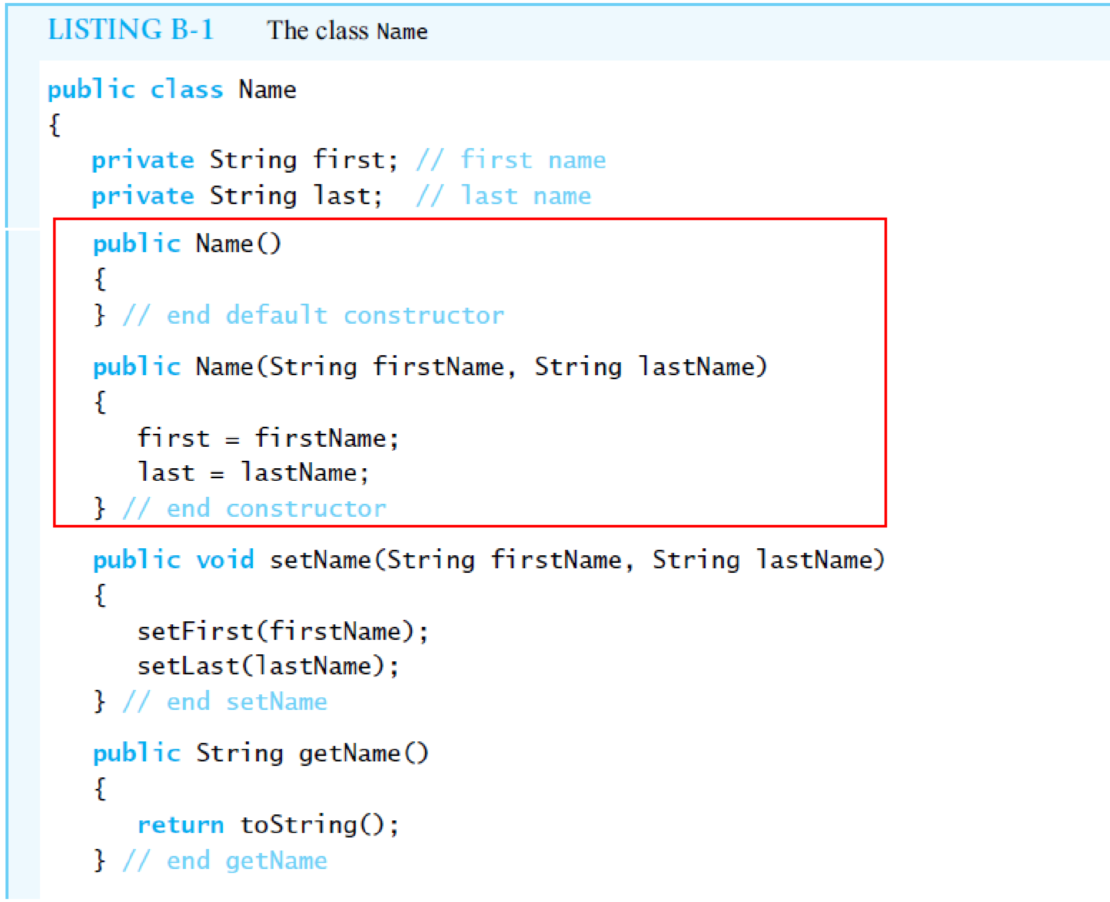
```
public class Name
{
   private String first; // first name
   private String last;  // last name

   public Name()
   {
   } // end default constructor

   public Name(String firstName, String lastName)
   {
      first = firstName;
      last = lastName;
   } // end constructor

   public void setName(String firstName, String lastName)
   {
      setFirst(firstName);
      setLast(lastName);
   } // end setName

   public String getName()
   {
      return toString();
   } // end getName
```

Review: Java Programming

- **Default Constructor**
  - A constructor without parameters

- If you do not define any constructors for a class, Java will automatically provide a default constructor.

- Once you start defining constructors, Java will not define any other constructors for you.

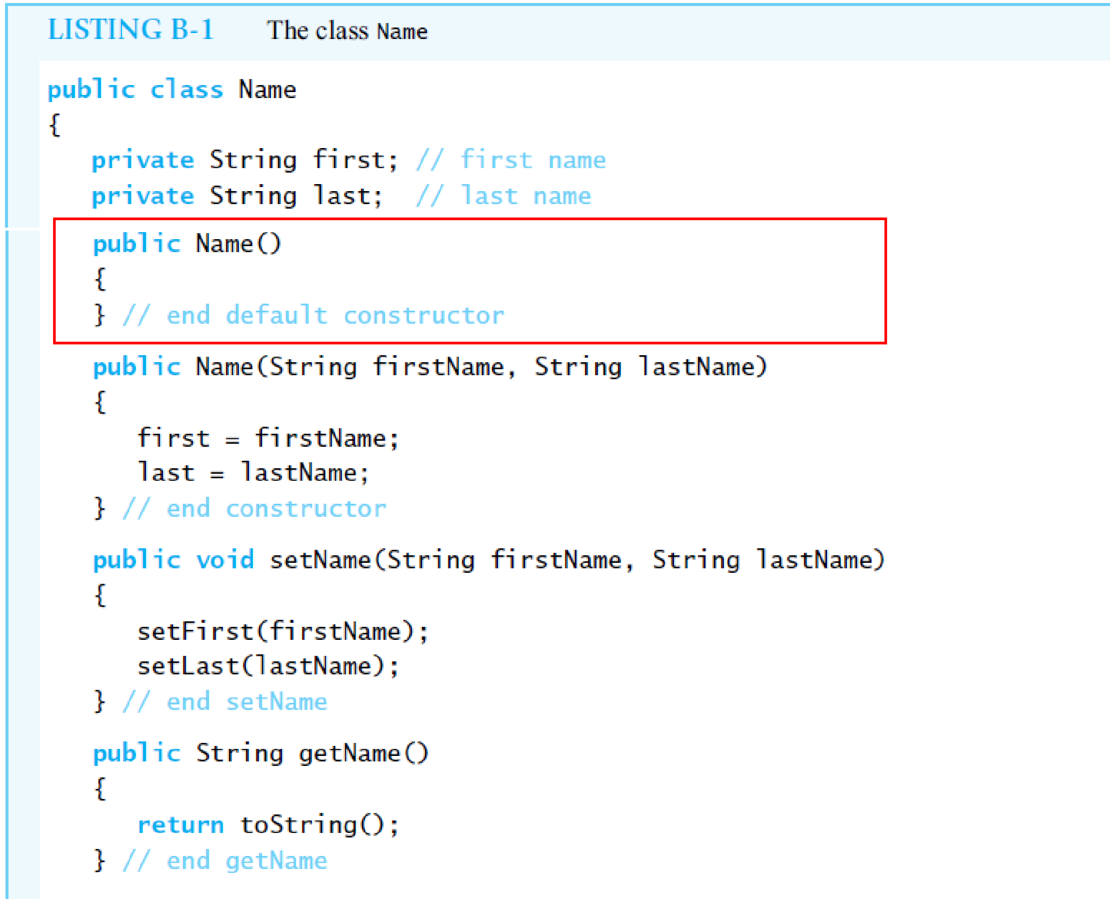


LISTING B-1 The class Name

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

   public Name()
   {
   } // end default constructor

   public Name(String firstName, String lastName)
   {
      first = firstName;
      last = lastName;
   } // end constructor

   public void setName(String firstName, String lastName)
   {
      setFirst(firstName);
      setLast(lastName);
   } // end setName

   public String getName()
   {
      return toString();
   } // end getName
```

Review: Java Programming

- **Two jobs of a Constructor**
  - allocates memory for the object
  - **initializes the data fields.**



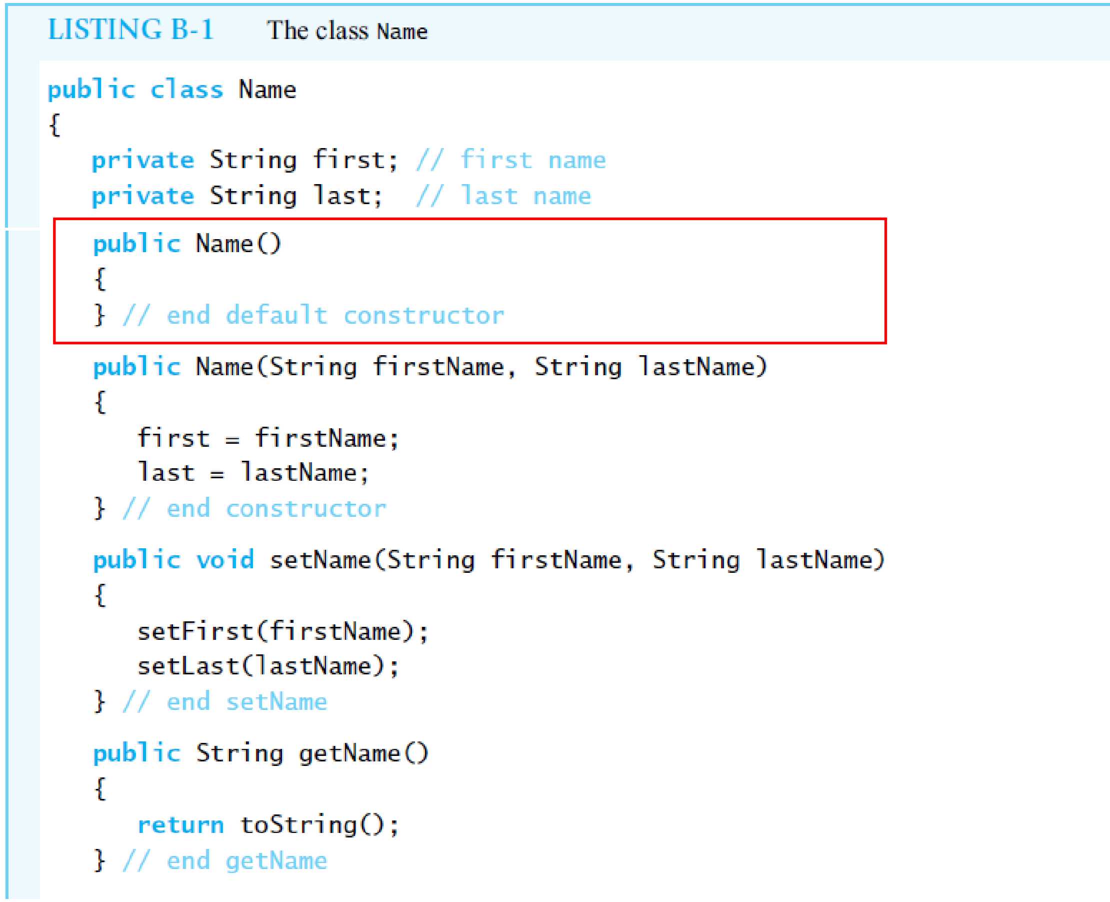
LISTING B-1 The class Name

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

   public Name()
   {
   } // end default constructor

   public Name(String firstName, String lastName)
   {
      first = firstName;
      last = lastName;
   } // end constructor

   public void setName(String firstName, String lastName)
   {
      setFirst(firstName);
      setLast(lastName);
   } // end setName

   public String getName()
   {
      return toString();
   } // end getName
```

Review: Java Programming

- **Two jobs of a Constructor**
  - allocates memory for the object
  - **initializes the data fields.**

- In the absence of any explicit initialization within a constructor, data fields are set to default values:
  - Reference types are null
  - primitive numeric types are zero
  - boolean types are false.

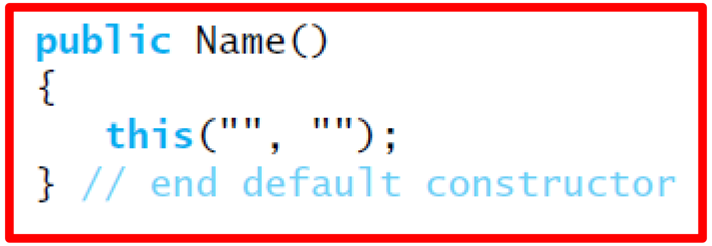


LISTING B-1 The class Name

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

   public Name()
   {
   } // end default constructor

   public Name(String firstName, String lastName)
   {
      first = firstName;
      last = lastName;
   } // end constructor

   public void setName(String firstName, String lastName)
   {
      setFirst(firstName);
      setLast(lastName);
   } // end setName

   public String getName()
   {
      return toString();
   } // end getName
```

Review: Java Programming

Or

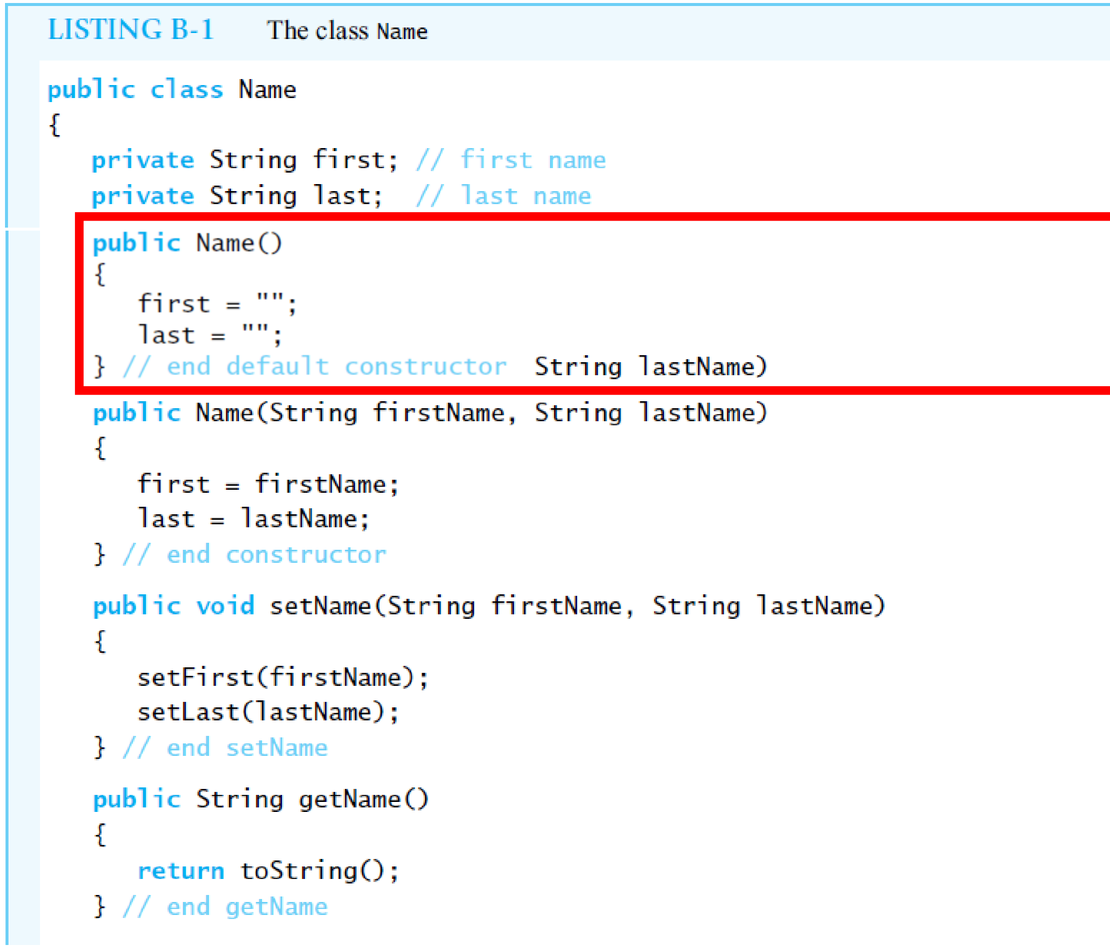


```
public Name()
{
   this("", "");
} // end default constructor
```

- **Two jobs of a Constructor**
  - allocates memory for the object
  - **initializes the data fields**.

- In the absence of any explicit initialization within a constructor, data fields are set to default values:
  - Reference types are null
  - primitive numeric types are zero
  - boolean types are false.

- Most of the classes you define should include a default constructor and initialize the data fields explicitly.



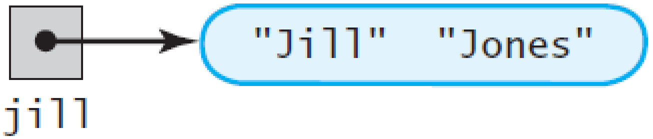
LISTING B-1 The class Name

```
public class Name
{
   private String first; // first name
   private String last;  // last name
   public Name()
   {
      first = "";
      last = "";
   } // end default constructor  String lastName)
   public Name(String firstName, String lastName)
   {
      first = firstName;
      last = lastName;
   } // end constructor
   public void setName(String firstName, String lastName)
   {
      setFirst(firstName);
      setLast(lastName);
   } // end setName
   public String getName()
   {
      return toString();
   } // end getName
```

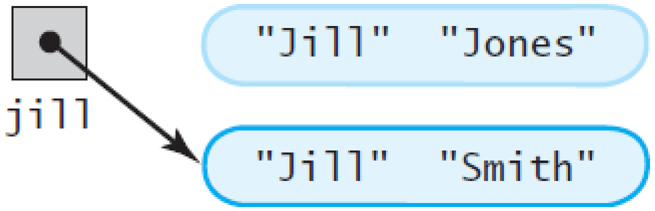
Review: Java Programming

- **Garbage Collection in Java**
  - Suppose there is a memory location which the variables in your program no longer reference.
  - The Java run-time environment **deallocates** such memory locations by returning them to the operating system so that they can be used again.

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

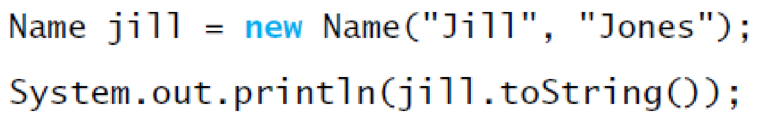


```
jill = new Name("Jill", "Smith");
```



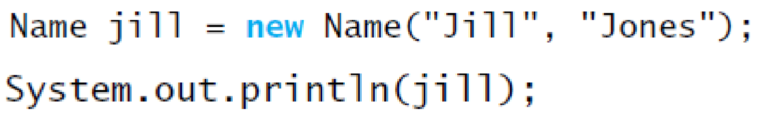
Review: Java Programming

- Method ***toString***
  - The method toString in the class Name returns a string that is the person’s full name.

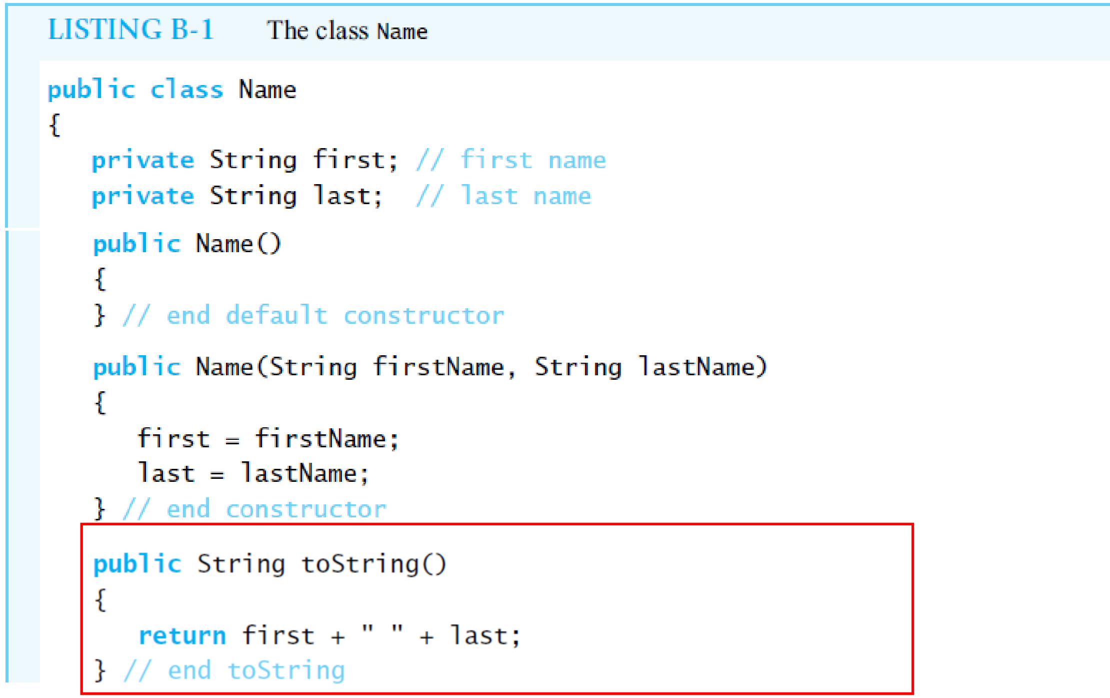


```
Name jill = new Name("Jill", "Jones");
System.out.println(jill.toString());
```

- In Java, the program invokes it automatically when you write



```
Name jill = new Name("Jill", "Jones");
System.out.println(jill);
```

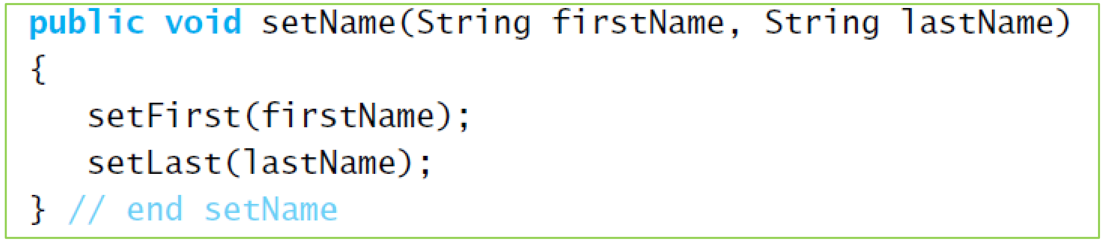


LISTING B-1 The class Name

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

   public Name()
   {
   } // end default constructor

   public Name(String firstName, String lastName)
   {
      first = firstName;
      last = lastName;
   } // end constructor

   public String toString()
   {
      return first + " " + last;
   } // end toString
```

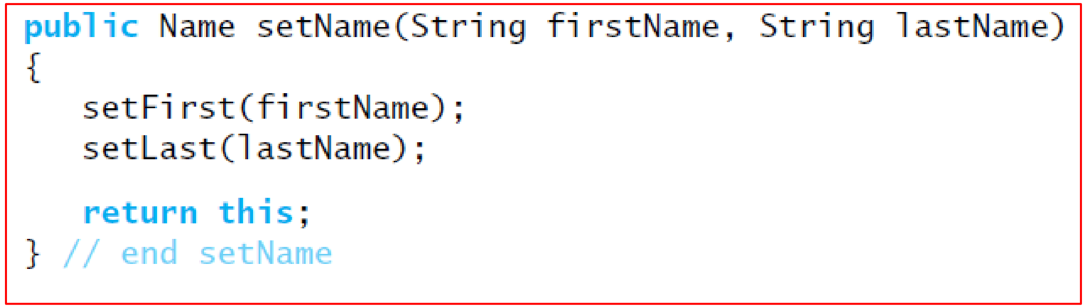
Review: Java Programming

- **Methods That Return an Instance of Their Class**

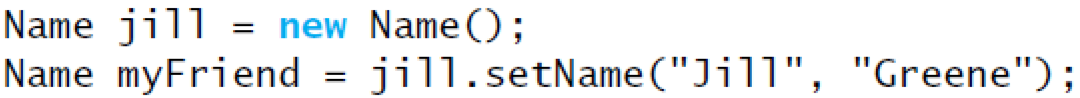


```
public void setName(String firstName, String lastName)
{
   setFirst(firstName);
   setLast(lastName);
} // end setName
```

VS



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

   return this;
} // end setName
```

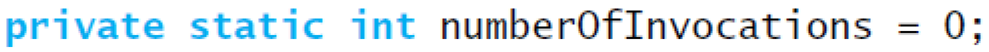


```
Name jill = new Name();
Name myFriend = jill.setName("Jill", "Greene");
```

The invocation of setName could appear as an argument to another method

Review: Java Programming

- A **Static data field** that does not belong to any one object



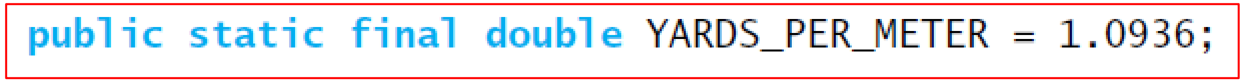
```
private static int numberOfInvocations = 0;
```

- Every object of the class can access the static data field, so objects can use a static field to communicate with each other or to perform some joint action.

- For example, tracking how many invocations of the class’s methods are made by all objects of the class.

Review: Java Programming

- Question: what happens if we declare a data field as follows?

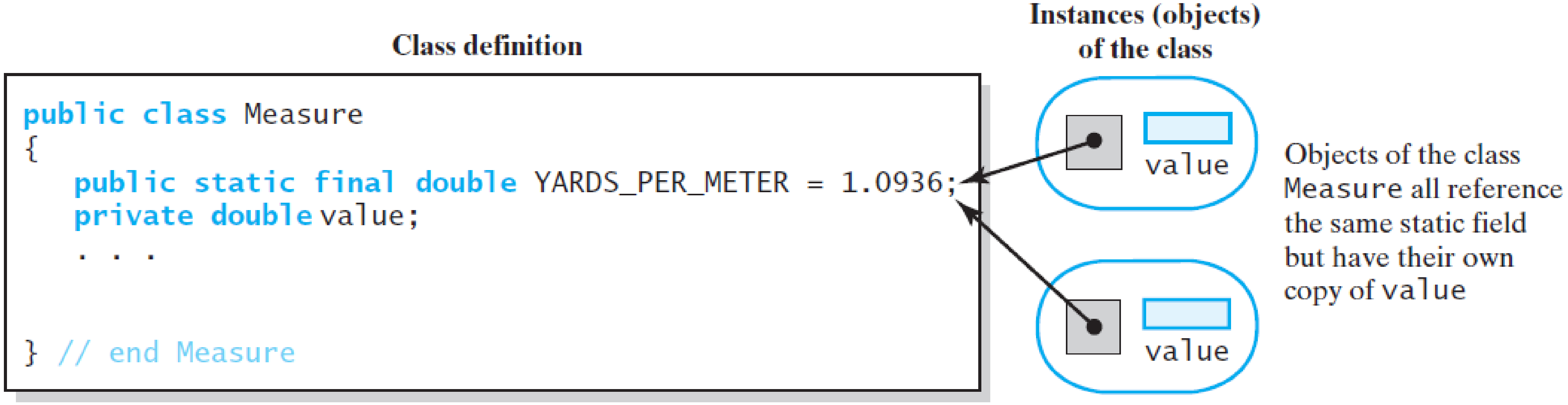


```
public static final double YARDS_PER_METER = 1.0936;
```

Review: Java Programming

- Question: what happens if we declare a data field as follows?

```
public static final double YARDS_PER_METER = 1.0936;
```



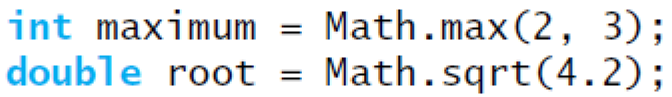
Class definition

```
public class Measure
{
   public static final double YARDS_PER_METER = 1.0936;
   private double value;
   . . .


} // end Measure
```

Objects of the class Measure all reference the same static field but have their own copy of value

Review: Java Programming

- A **Static <u>method</u>** that does not belong to an object of any kind
  - We need to use the class name instead of an object name to invoke the method.



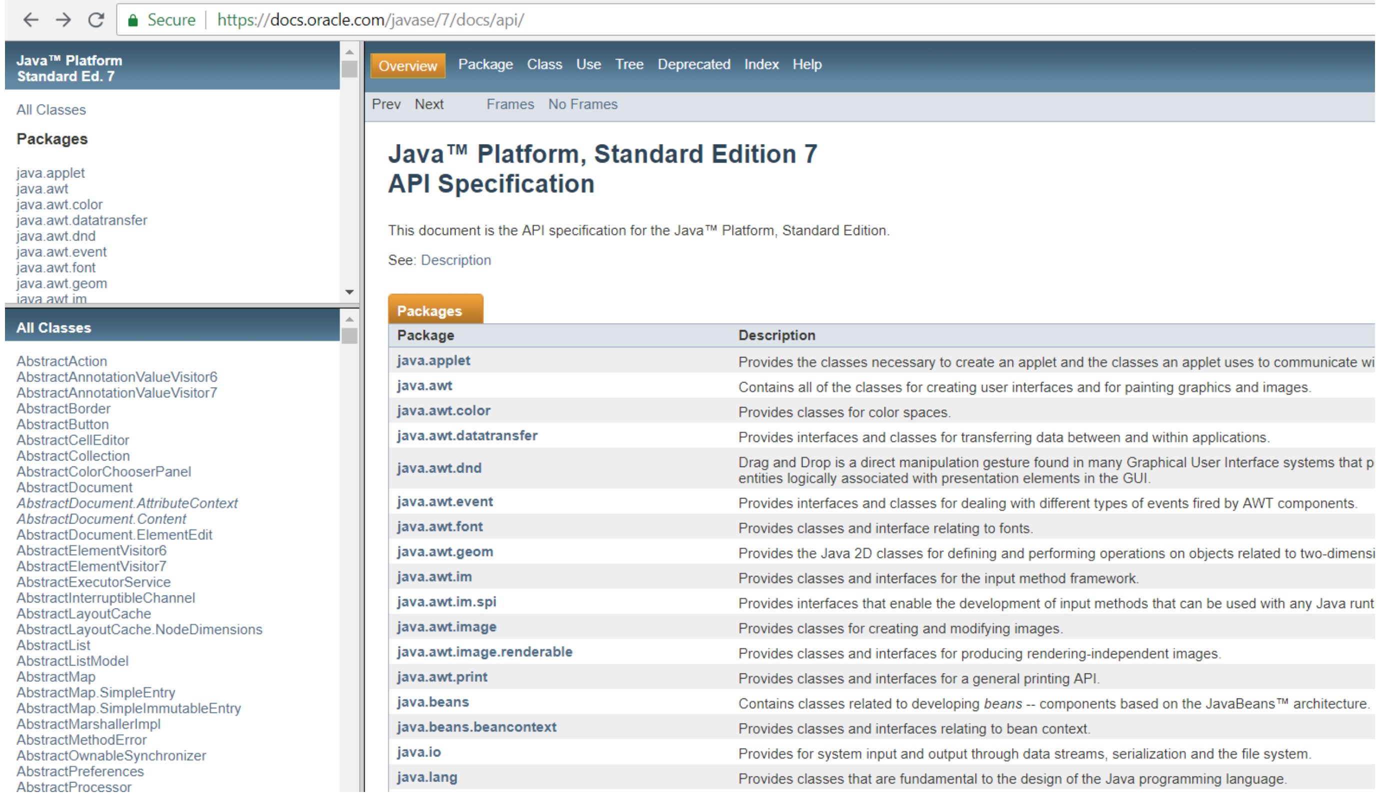
```
int maximum = Math.max(2, 3);
double root = Math.sqrt(4.2);
```

- You can include a test of a class as a main method in the class's definition. Every application program's main method is <u>static</u>.

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

Review: Java Programming

- **Java Class Library**
  - Java comes with a collection of many classes that you can use in your programs



Java™ Platform
Standard Ed. 7

All Classes

Packages

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font
java.awt.geom
java.awt.im

All Classes

AbstractAction
AbstractAnnotationValueVisitor6
AbstractAnnotationValueVisitor7
AbstractBorder
AbstractButton
AbstractCellEditor
AbstractCollection
AbstractColorChooserPanel
AbstractDocument
AbstractDocument.AttributeContext
AbstractDocument.Content
AbstractDocument.ElementEdit
AbstractElementVisitor6
AbstractElementVisitor7
AbstractExecutorService
AbstractInterruptibleChannel
AbstractLayoutCache
AbstractLayoutCache.NodeDimensions
AbstractList
AbstractListModel
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutableEntry
AbstractMarshallerImpl
AbstractMethodError
AbstractOwnableSynchronizer
AbstractPreferences
AbstractProcessor

Java™ Platform, Standard Edition 7
API Specification

This document is the API specification for the Java™ Platform, Standard Edition.

See: Description

Packages

Package	Description
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate wi
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that p entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensi
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runt
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.
java.awt.print	Provides classes and interfaces for a general printing API.
java.beans	Contains classes related to developing *beans* -- components based on the JavaBeans™ architecture.
java.beans.beancontext	Provides classes and interfaces relating to bean context.
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.

Review: Java Programming

- Object-oriented programming embodies three design concepts:
  - Encapsulation
  - Inheritance
  - Polymorphism

Review: Java Programming

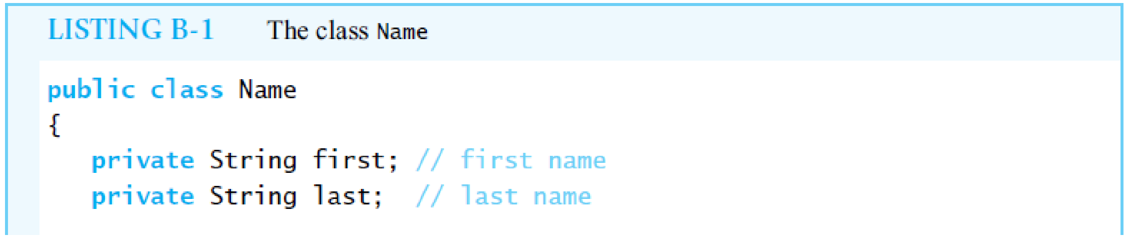
- Object-oriented programming embodies three design concepts:
  - Encapsulation
  - **Inheritance**
  - Polymorphism

Review: Java Programming

- Creating Classes from Other Classes
  - **Composition**
  - **Inheritance**

Review: Java Programming

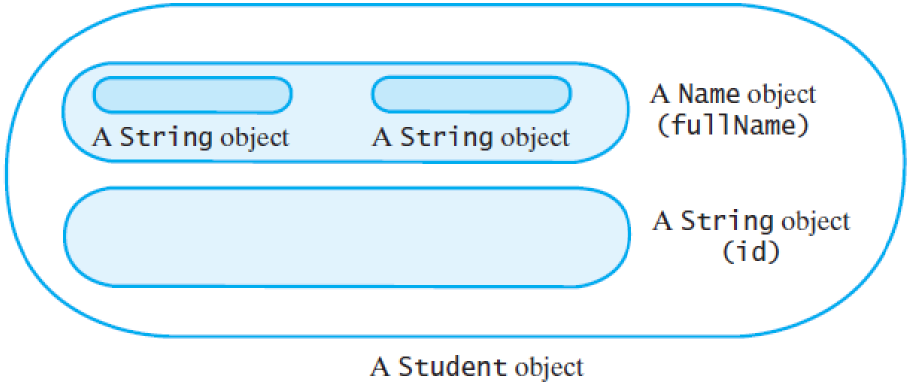
- Creating Classes from Other Classes
    - Composition
    - Inheritance

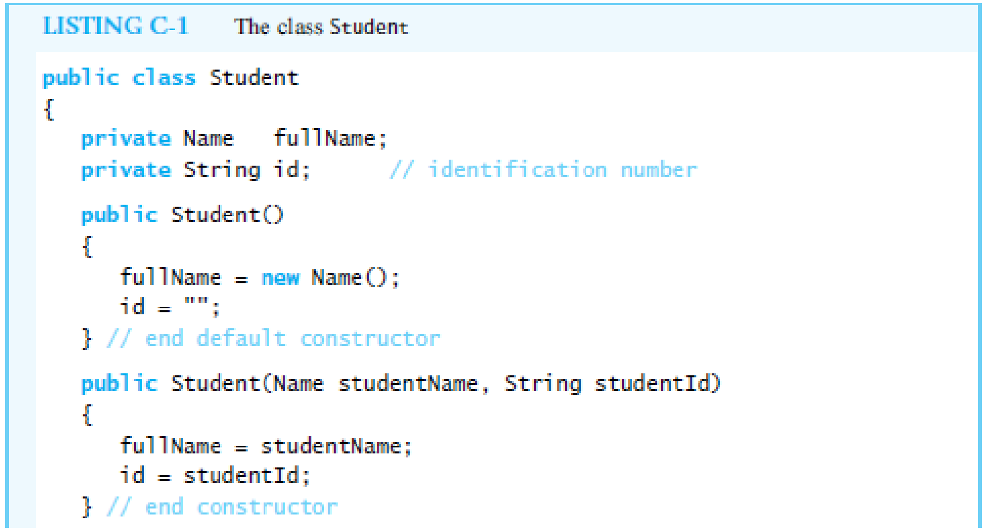


LISTING B-1 The class Name

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

- **Composition:** A class uses composition when it has objects as data fields.





LISTING C-1 The class Student

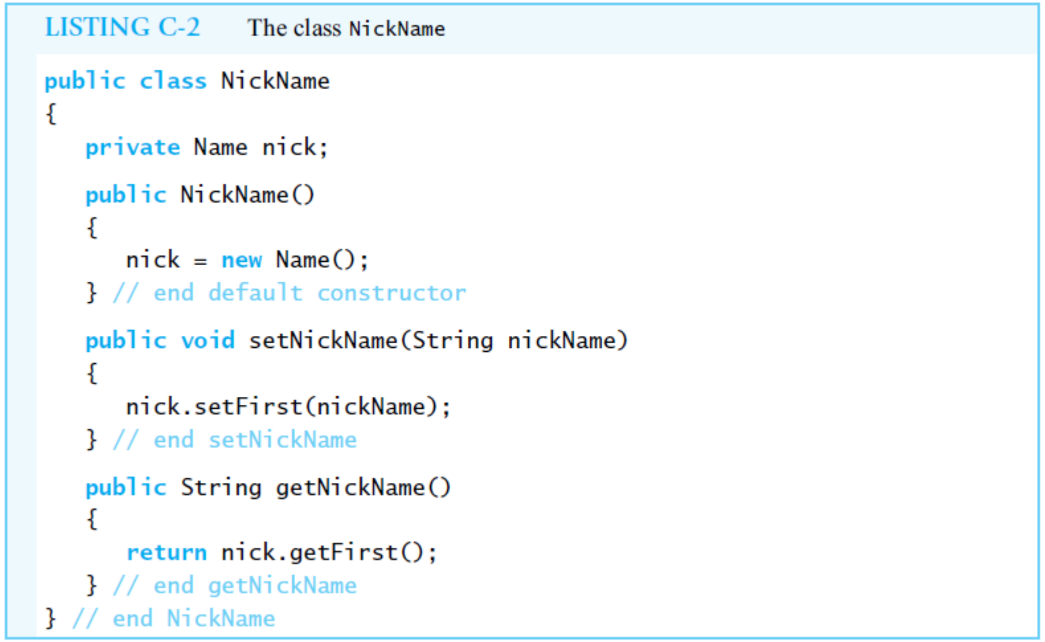
```java
public class Student
{
   private Name   fullName;
   private String id;      // identification number

   public Student()
   {
      fullName = new Name();
      id = "";
   } // end default constructor

   public Student(Name studentName, String studentId)
   {
      fullName = studentName;
      id = studentId;
   } // end constructor
```

Review: Java Programming

- **Adapter class**
  - Suppose that you have a class, but the names of its methods do not suit your application. Or maybe you want to simplify some methods or eliminate others.
  - You can use composition to write a new class that has an instance of your existing class as a data field and defines the methods that you want. Such a new class is called an ***adapter class***.

Review: Java Programming

- **Adapter class**
  - Suppose that you have a class, but the names of its methods do not suit your application. Or maybe you want to simplify some methods or eliminate others.
  - You can use composition to write a new class that has an instance of your existing class as a data field and defines the methods that you want. Such a new class is called an ***adapter class***.



LISTING C-2 The class NickName

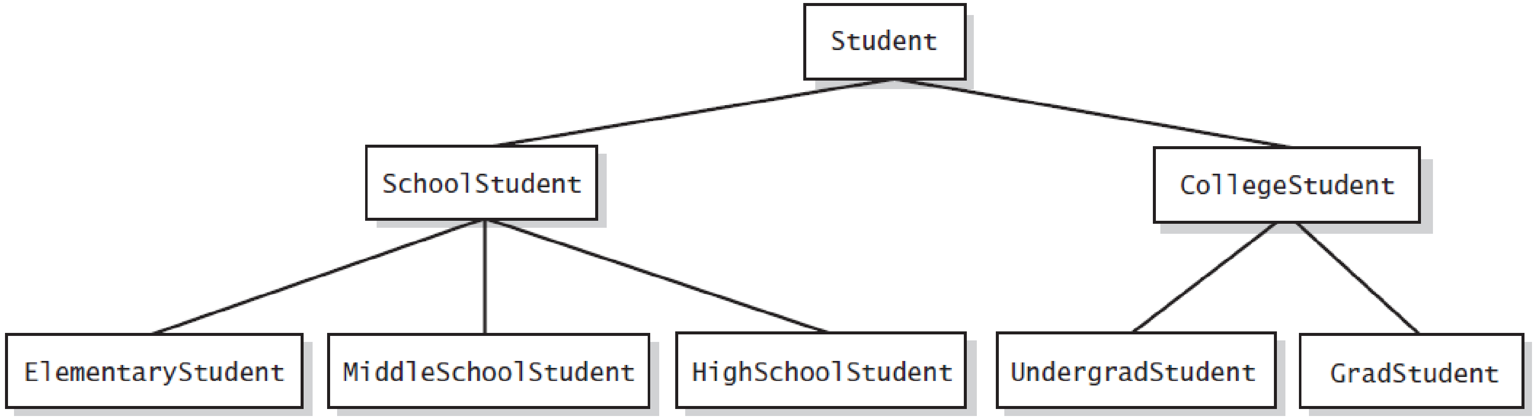
```
public class NickName
{
   private Name nick;

   public NickName()
   {
      nick = new Name();
   } // end default constructor

   public void setNickName(String nickName)
   {
      nick.setFirst(nickName);
   } // end setNickName

   public String getNickName()
   {
      return nick.getFirst();
   } // end getNickName
} // end NickName
```

A ***NickName*** object now has only *NickName*'s methods, and not the methods of ***Name***

Review: Java Programming

- **Inheritance** allows you to define a general class and then later to define more specialized classes that add to or revise the details of the older, more general class definition.

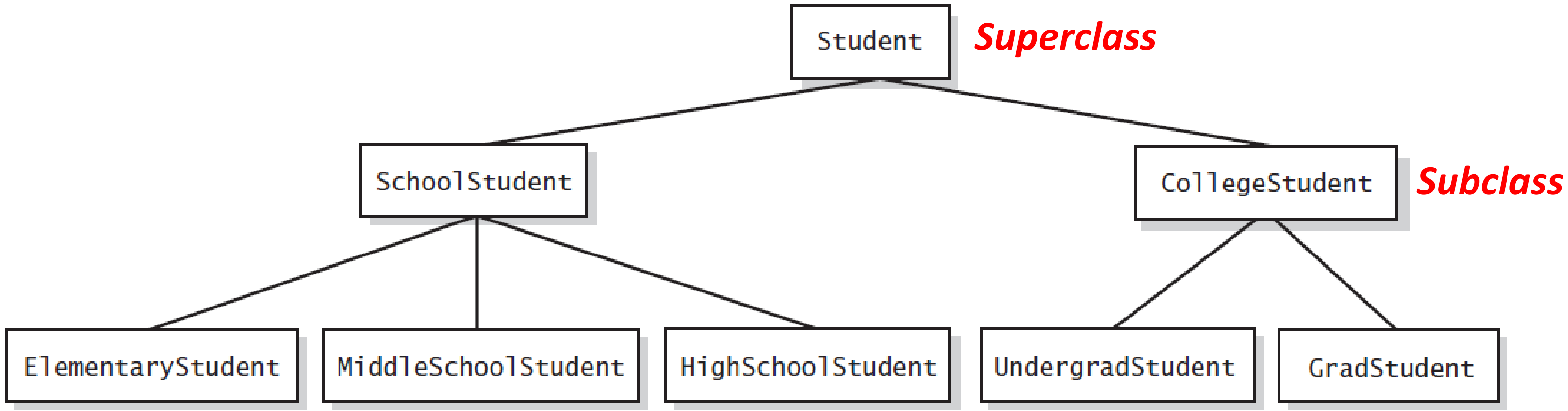
FIGURE C-3 A hierarchy of student classes



Review: Java Programming

- **Inheritance** allows you to define a general class and then later to define more specialized classes that add to or revise the details of the older, more general class definition.

FIGURE C-3 A hierarchy of student classes



Superclass

Subclass

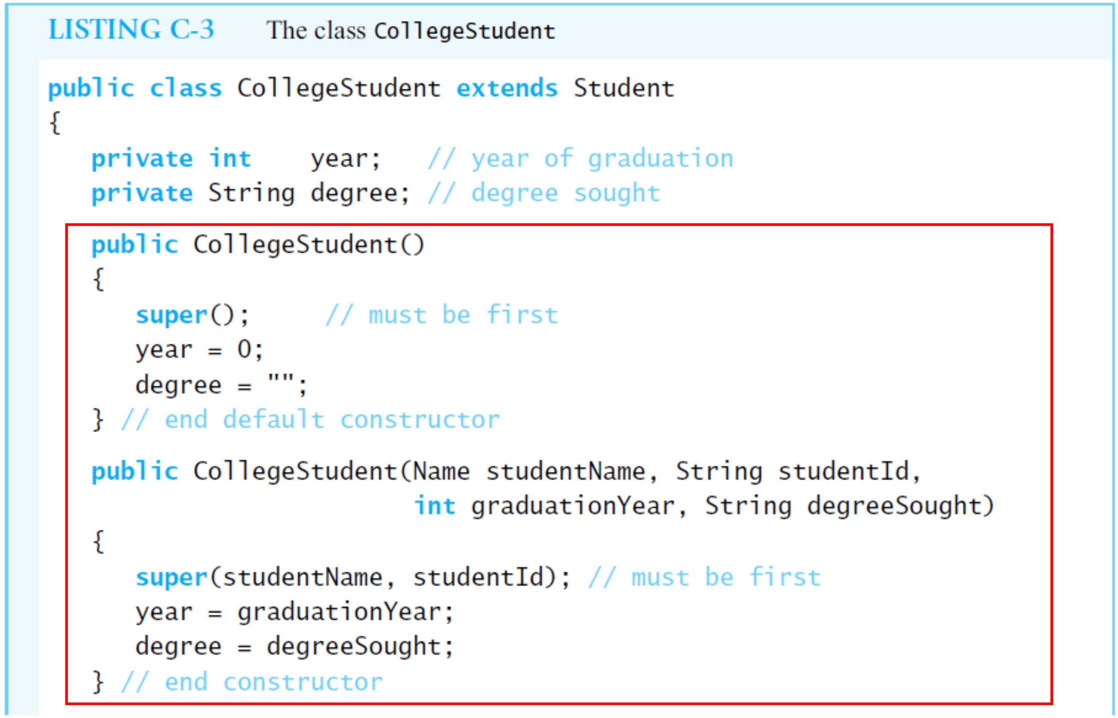
```
public class CollegeStudent extends Student
```

Inheritance

- **Invoking Constructors from Within Constructors**
  - The constructor needs to initialize data fields inherited from the superclass

```
super();

super(studentName, studentId);
```

- Note that:
  - If you do not invoke super, Java will do it for you.
  - The call to super must occur first in the constructor.
  - You can use super to invoke a constructor only from within another constructor.



LISTING C-3 The class `CollegeStudent`

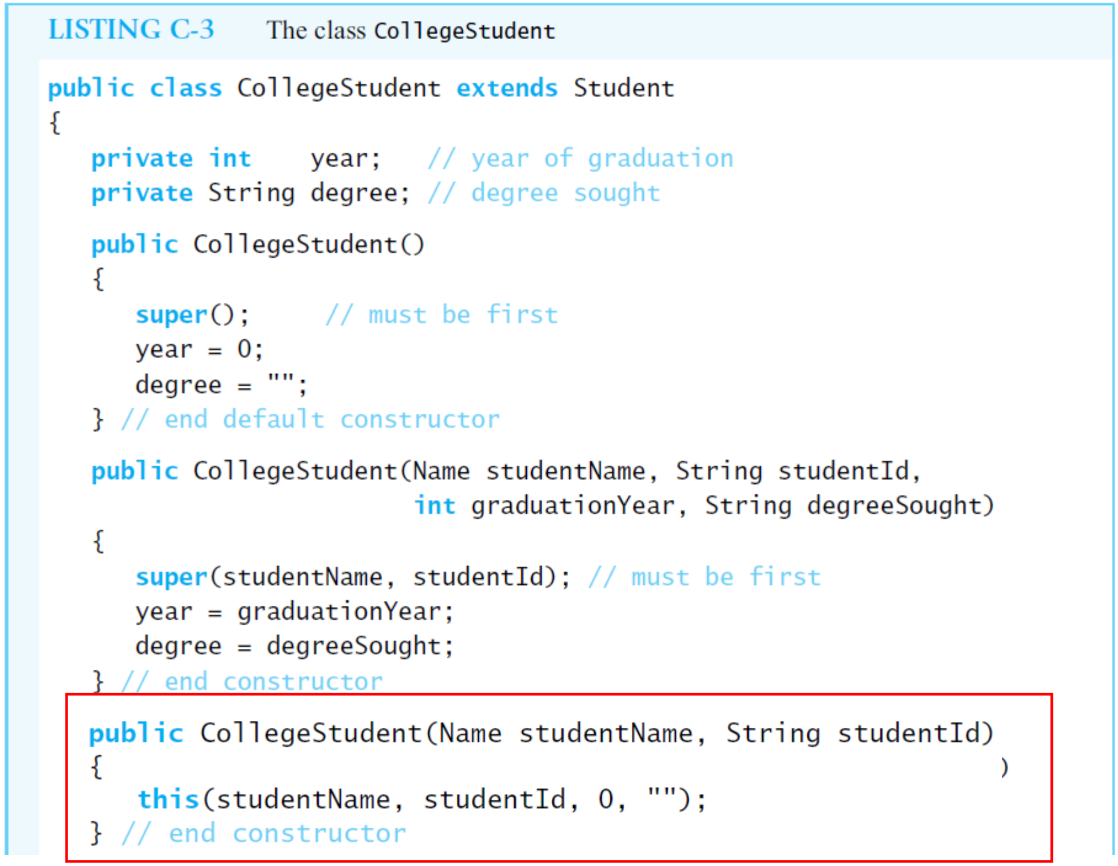
```
public class CollegeStudent extends Student
{
   private int    year;    // year of graduation
   private String degree; // degree sought

   public CollegeStudent()
   {
      super();      // must be first
      year = 0;
      degree = "";
   } // end default constructor

   public CollegeStudent(Name studentName, String studentId,
                         int graduationYear, String degreeSought)
   {
      super(studentName, studentId); // must be first
      year = graduationYear;
      degree = degreeSought;
   } // end constructor
```

Inheritance

- **Invoking Constructors from Within Constructors**
  - The constructor needs to initialize data fields inherited from the superclass

```
super();
super(studentName, studentId);
```

- **Using *this* to invoke a constructor**
  - *this* calls a constructor of the same class instead of a constructor of the superclass



LISTING C-3 The class CollegeStudent

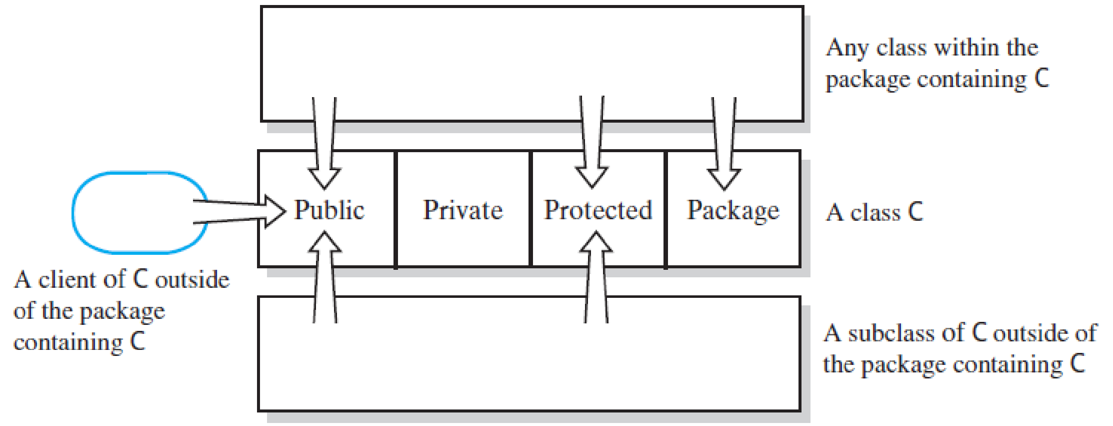
```
public class CollegeStudent extends Student
{
   private int    year;    // year of graduation
   private String degree; // degree sought

   public CollegeStudent()
   {
      super();      // must be first
      year = 0;
      degree = "";
   } // end default constructor

   public CollegeStudent(Name studentName, String studentId,
                         int graduationYear, String degreeSought)
   {
      super(studentName, studentId); // must be first
      year = graduationYear;
      degree = degreeSought;
   } // end constructor

   public CollegeStudent(Name studentName, String studentId)
   {                                                            )
      this(studentName, studentId, 0, "");
   } // end constructor
```

Inheritance

- **Protected** access modifier

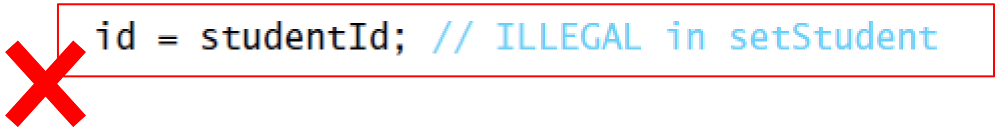
- A method or data field that is modified by **protected** can be accessed by name only within
  - Its own class definition C
  - Any class derived from C
  - Any class within the same package as C

FIGURE C-4 Public, private, protected, and package access of the data fields and methods of class C


Any class within the package containing C
Public
Private
Protected
Package
A class C
A client of C outside of the package containing C
A subclass of C outside of the package containing C

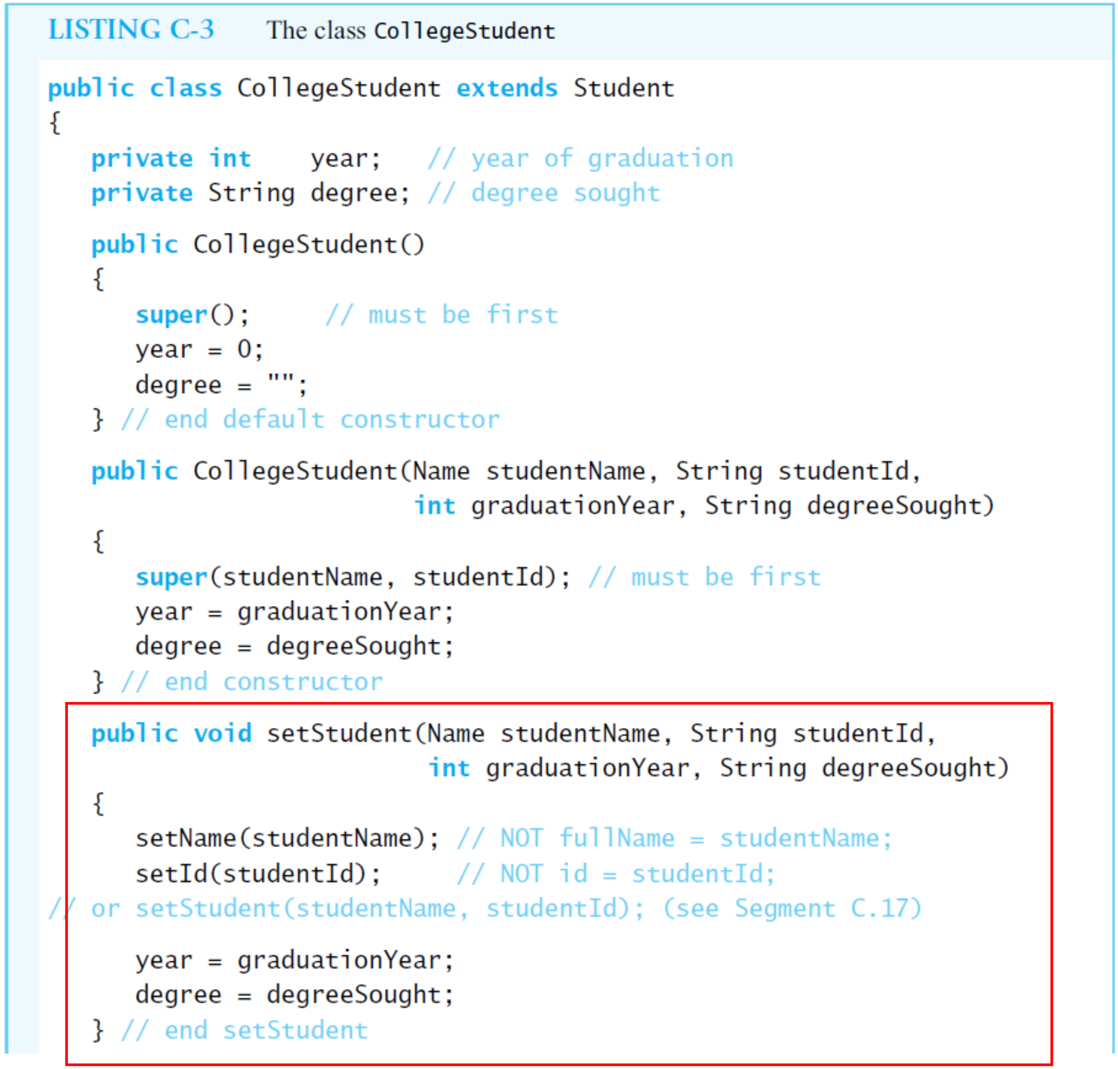
Inheritance

- **Private Fields and Methods of the Superclass**
  - A data field that is private in a superclass is not accessible by name within the definition of a method for any other class, including a subclass.



```
id = studentId; // ILLEGAL in setStudent
```

  - A subclass cannot invoke a superclass's private methods directly

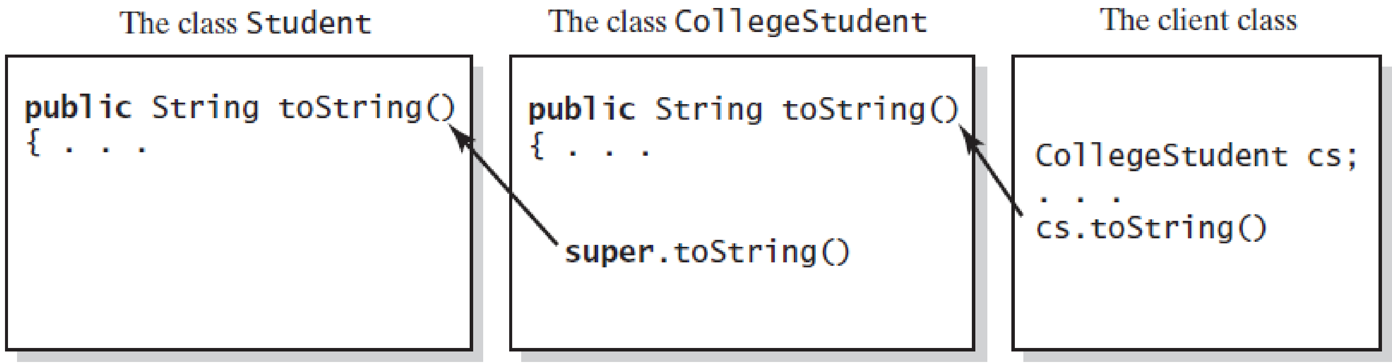


LISTING C-3 The class CollegeStudent

```
public class CollegeStudent extends Student
{
   private int    year;   // year of graduation
   private String degree; // degree sought

   public CollegeStudent()
   {
      super();      // must be first
      year = 0;
      degree = "";
   } // end default constructor

   public CollegeStudent(Name studentName, String studentId,
                         int graduationYear, String degreeSought)
   {
      super(studentName, studentId); // must be first
      year = graduationYear;
      degree = degreeSought;
   } // end constructor

   public void setStudent(Name studentName, String studentId,
                          int graduationYear, String degreeSought)
   {
      setName(studentName); // NOT fullName = studentName;
      setId(studentId);     // NOT id = studentId;
// or setStudent(studentName, studentId); (see Segment C.17)

      year = graduationYear;
      degree = degreeSought;
   } // end setStudent
```

Inheritance

- **Overriding a method**
  - A method in a subclass overrides a method in the superclass when both methods have the same name, the same number and types of parameters, and the same return type.



The class Student

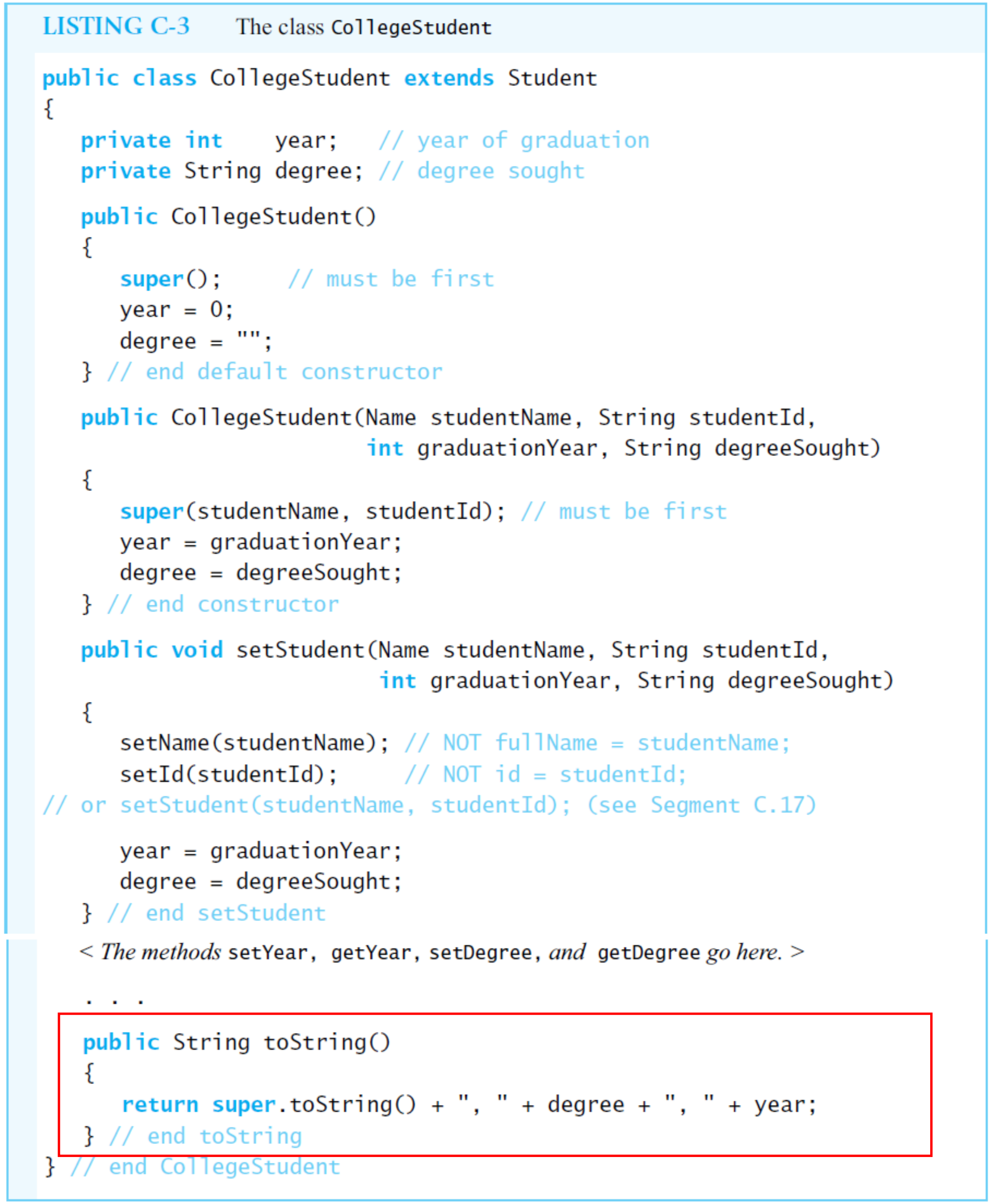
```
public String toString()
{ . . .
```

The class CollegeStudent

```
public String toString()
{ . . .

  super.toString()
```

The client class

```
CollegeStudent cs;
. . .
cs.toString()
```

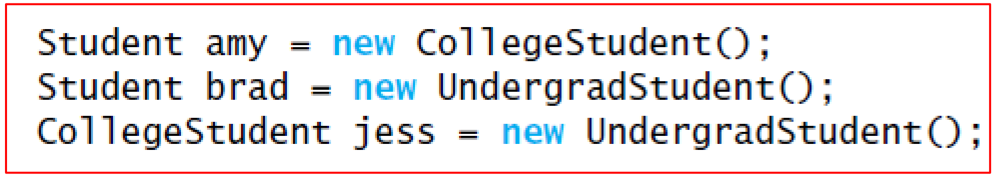


LISTING C-3 The class CollegeStudent

```
public class CollegeStudent extends Student
{
   private int    year;   // year of graduation
   private String degree; // degree sought

   public CollegeStudent()
   {
      super();      // must be first
      year = 0;
      degree = "";
   } // end default constructor

   public CollegeStudent(Name studentName, String studentId,
                         int graduationYear, String degreeSought)
   {
      super(studentName, studentId); // must be first
      year = graduationYear;
      degree = degreeSought;
   } // end constructor

   public void setStudent(Name studentName, String studentId,
                          int graduationYear, String degreeSought)
   {
      setName(studentName); // NOT fullName = studentName;
      setId(studentId);     // NOT id = studentId;
// or setStudent(studentName, studentId); (see Segment C.17)

      year = graduationYear;
      degree = degreeSought;
   } // end setStudent

   < The methods setYear, getYear, setDegree, and getDegree go here. >

   . . .

   public String toString()
   {
      return super.toString() + ", " + degree + ", " + year;
   } // end toString
} // end CollegeStudent
```

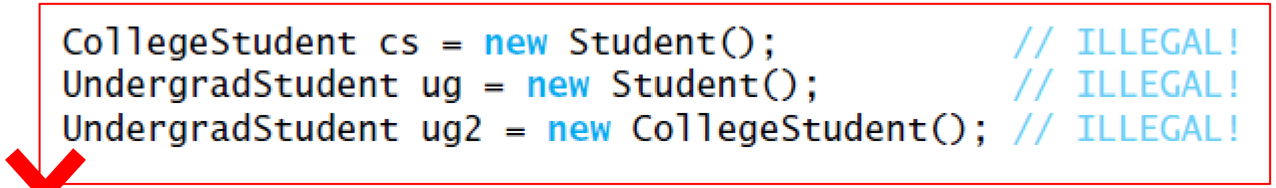
Inheritance

- **Object types of a subclass**
  - Because an object of a subclass also has the types of all of its ancestor classes, you can assign an object of a class to a variable of any ancestor type, but not the other way around.



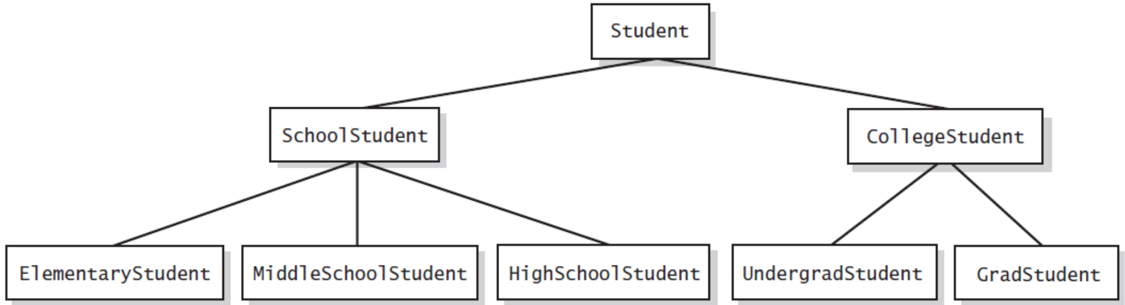
```
Student amy = new CollegeStudent();
Student brad = new UndergradStudent();
CollegeStudent jess = new UndergradStudent();
```

✗



```
CollegeStudent cs = new Student();              // ILLEGAL!
UndergradStudent ug = new Student();            // ILLEGAL!
UndergradStudent ug2 = new CollegeStudent();    // ILLEGAL!
```

FIGURE C-3 A hierarchy of student classes

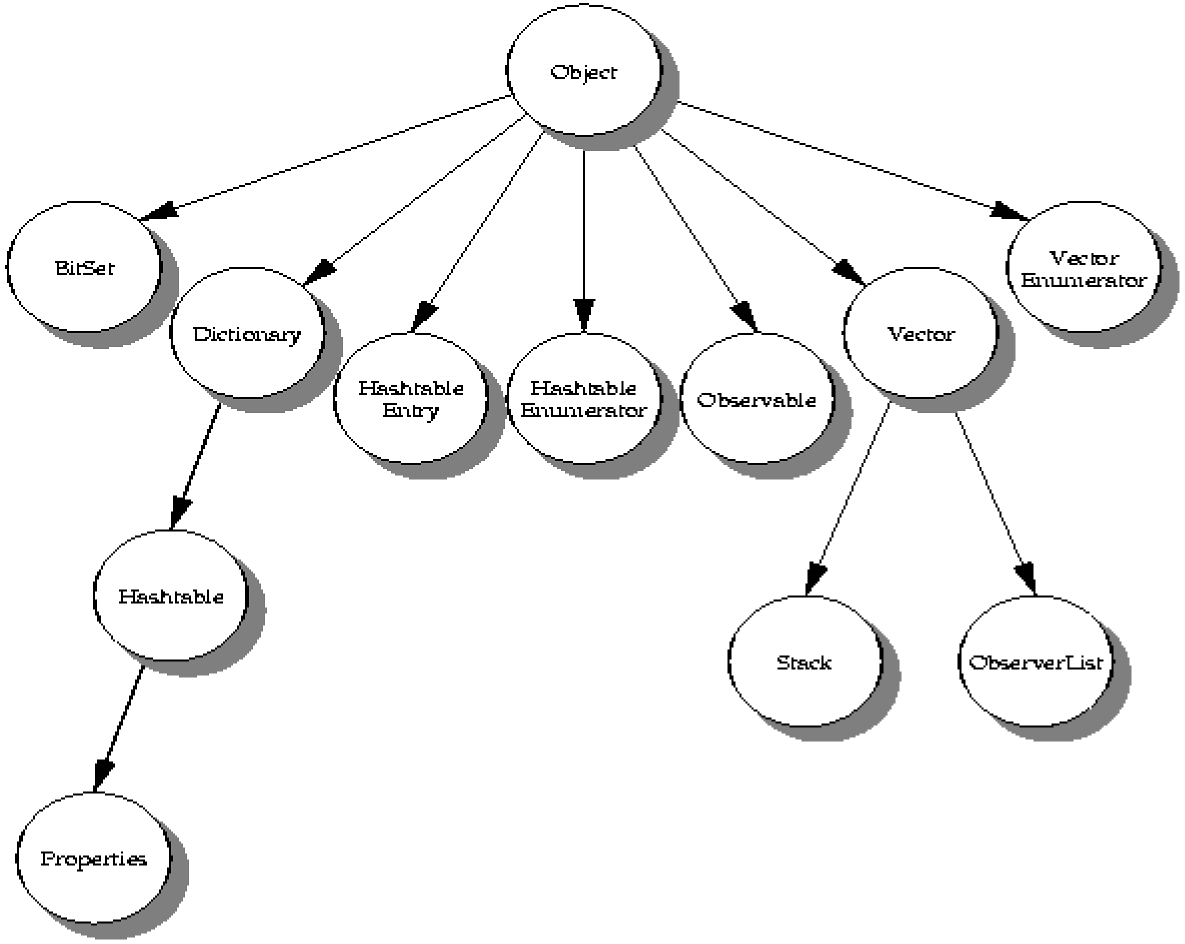


Review: Java Programming

- The Class **Object**
  - Every object of every class is of type Object

- The class **Object** contains certain methods, among which are
  - toString,
  - equals
  - clone

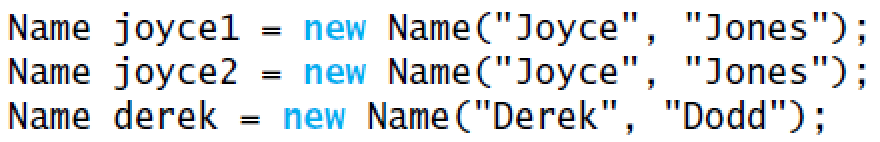
- Typically, you need to override the inherited method definitions with new, more appropriate definitions.



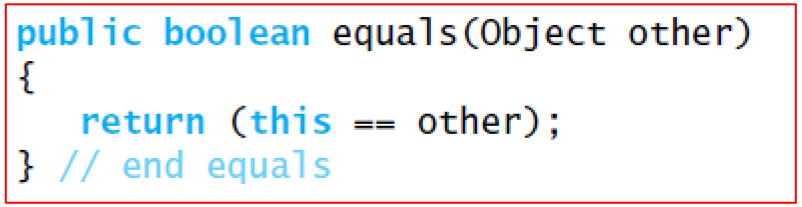
http://www.oracle.com/technetwork/java/object-142075.html

Review: Java Programming

- The **equals** method



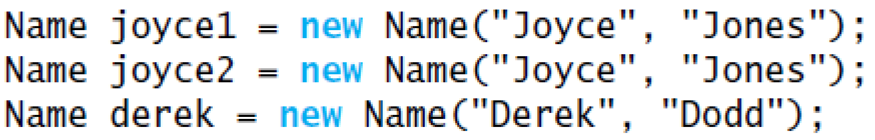
```
Name joyce1 = new Name("Joyce", "Jones");
Name joyce2 = new Name("Joyce", "Jones");
Name derek = new Name("Derek", "Dodd");
```



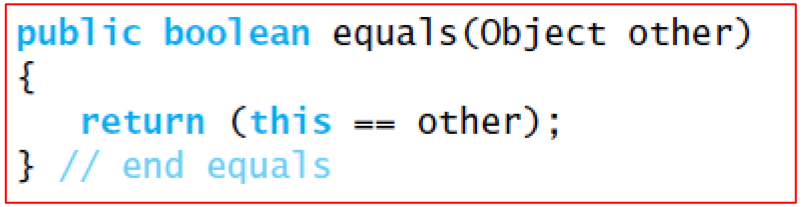
```
public boolean equals(Object other)
{
   return (this == other);
} // end equals
```

Review: Java Programming

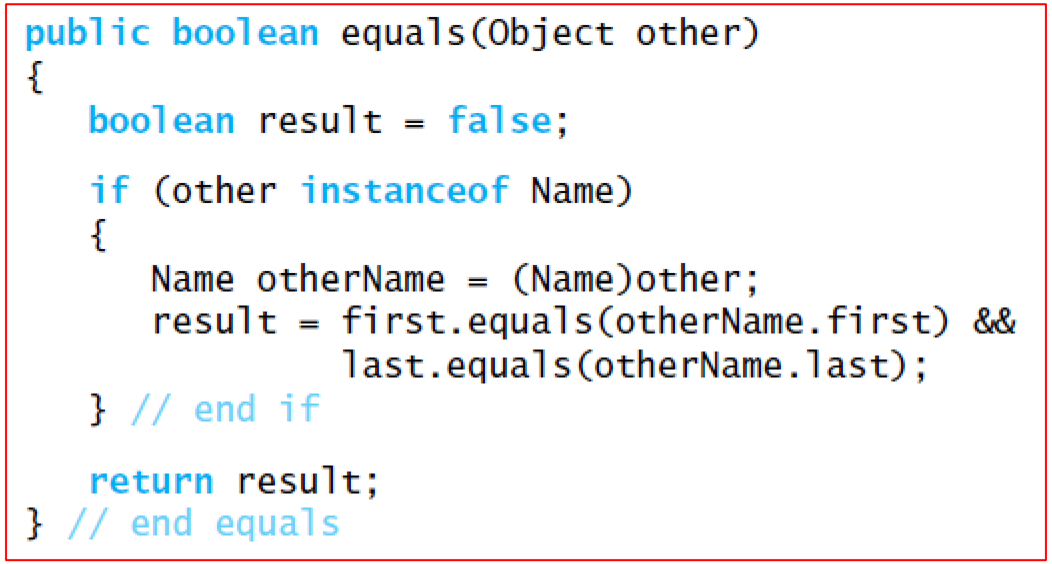
- The **equals** method



```java
Name joyce1 = new Name("Joyce", "Jones");
Name joyce2 = new Name("Joyce", "Jones");
Name derek = new Name("Derek", "Dodd");
```



```java
public boolean equals(Object other)
{
   return (this == other);
} // end equals
```



```java
public boolean equals(Object other)
{
   boolean result = false;

   if (other instanceof Name)
   {
      Name otherName = (Name)other;
      result = first.equals(otherName.first) &&
               last.equals(otherName.last);
   } // end if

   return result;
} // end equals
```

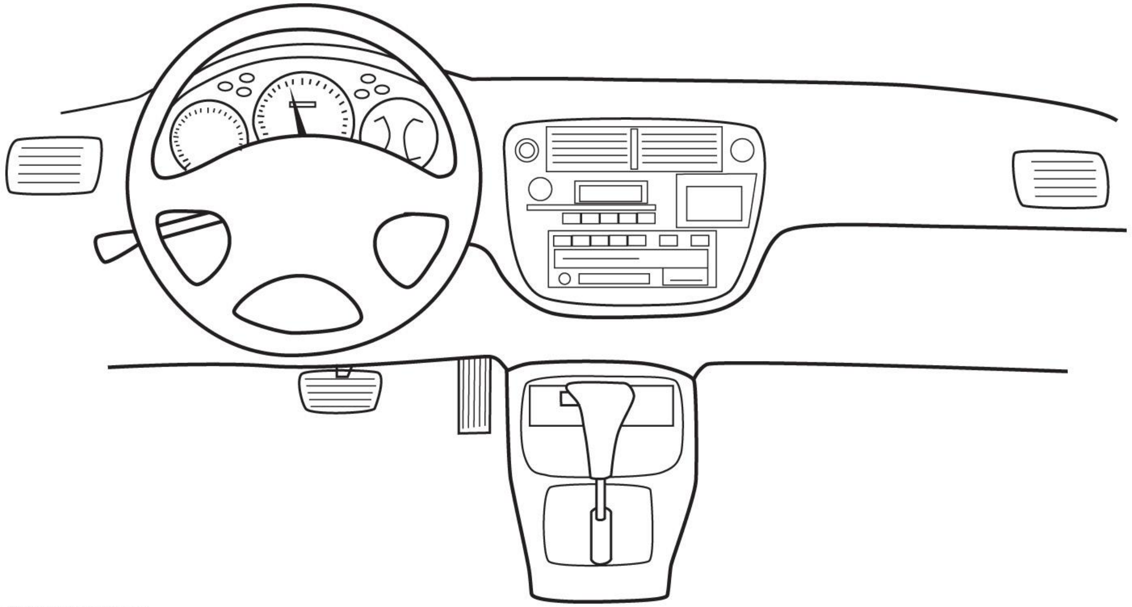
Review: Java Programming

- Object-oriented programming embodies three design concepts:
    - Encapsulation
    - Inheritance
    - Polymorphism

Review: Java Programming

- Encapsulation (Information Hiding)
  - Enclose data and methods within a class
  - Hide implementation details

- Programmer receives only enough information to be able to use the class

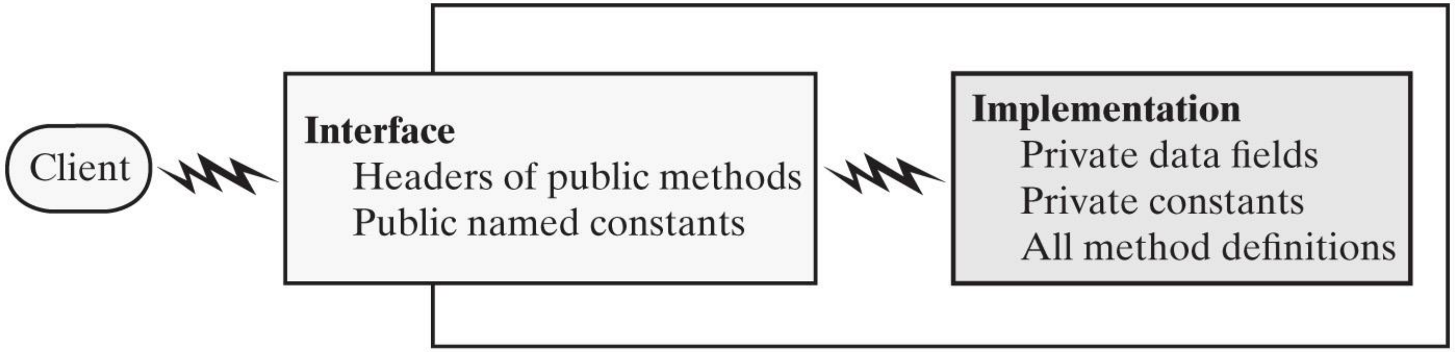


An automobile's controls are visible to the driver, but its inner workings are hidden

Review: Java Programming

- **Abstraction** (Focus on what instead of how)
  - What needs to be done?
  - For the moment ignore how it will be done.

- Divide class into two parts
  - Java interface
  - Implementation

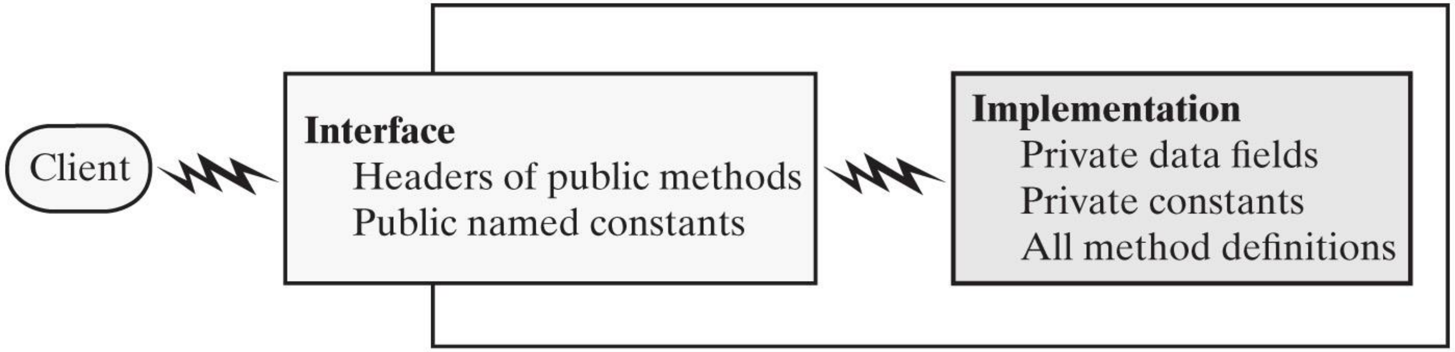


© 2019 Pearson Education, Inc.

An interface provides well-regulated communication between a hidden implementation and a client

Java Interfaces

- **Java Interfaces**
  - Declare a number of public methods
  - Should include comments to inform programmer
  - Any data fields here should be public, final, static



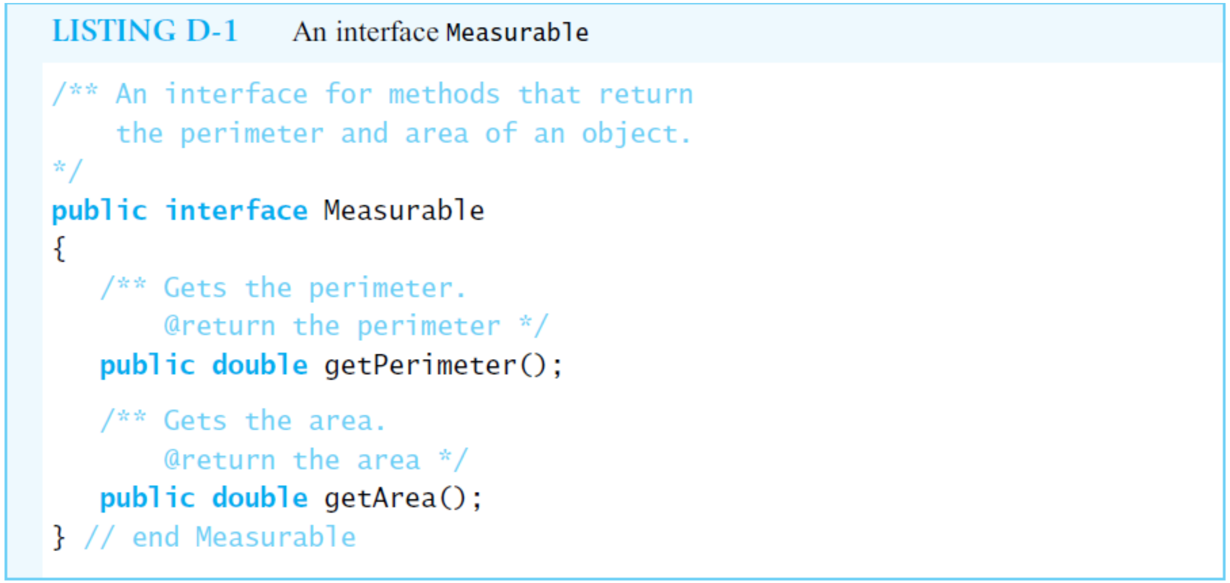
An interface provides well-regulated communication between a hidden implementation and a client

Java Interfaces

- **Writing a Java Interface**

```
public interface interface-name
```

- You store an interface definition in a file with the same name as the interface, followed by .java. For example, the interface is in the file *Measureable.java*

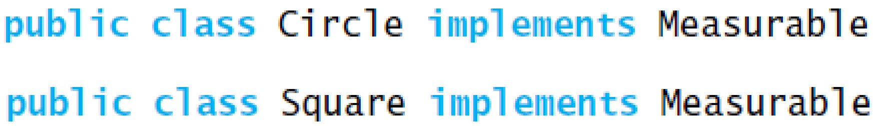


LISTING D-1 An interface `Measurable`

```
/** An interface for methods that return
    the perimeter and area of an object.
*/
public interface Measurable
{
   /** Gets the perimeter.
       @return the perimeter */
   public double getPerimeter();

   /** Gets the area.
       @return the area */
   public double getArea();
} // end Measurable
```

Java Interfaces

- **Implementing a Java Interface**

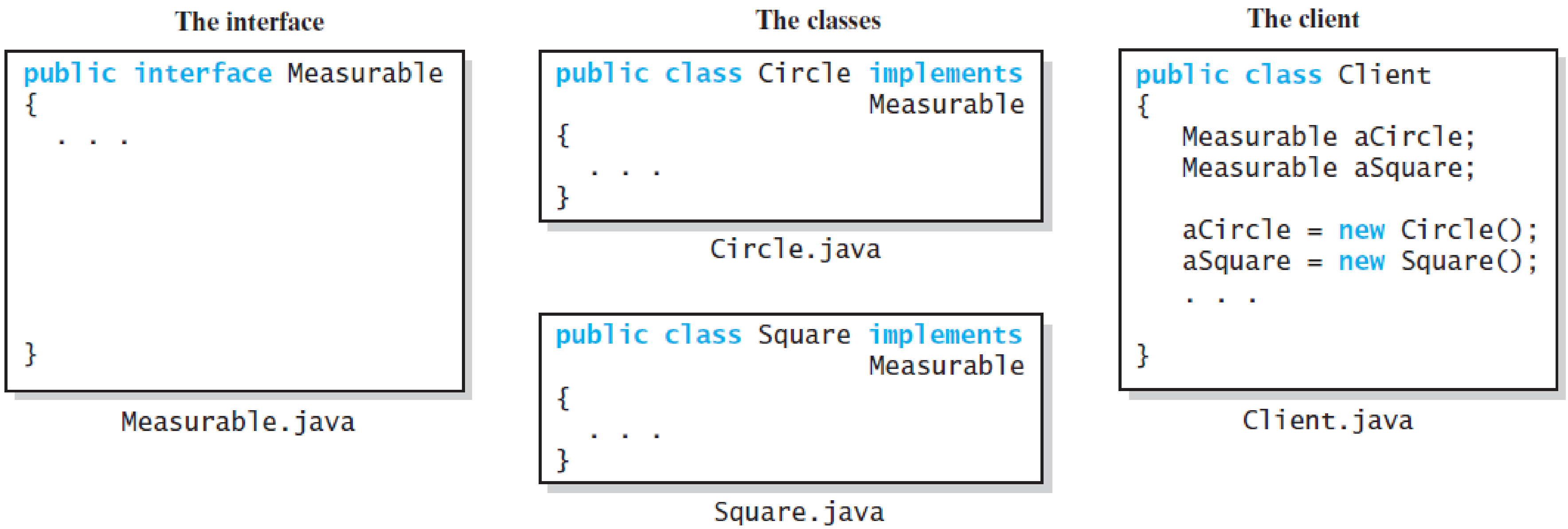


```
public class Circle implements Measurable

public class Square implements Measurable
```

Java Interfaces

- **Implementing a Java Interface**

```
public class Circle implements Measurable

public class Square implements Measurable
```



```
public interface Measurable
{
   . . .



}
```

```
public class Circle implements
                    Measurable
{
   . . .
}
```

```
public class Square implements
                    Measurable
{
   . . .
}
```

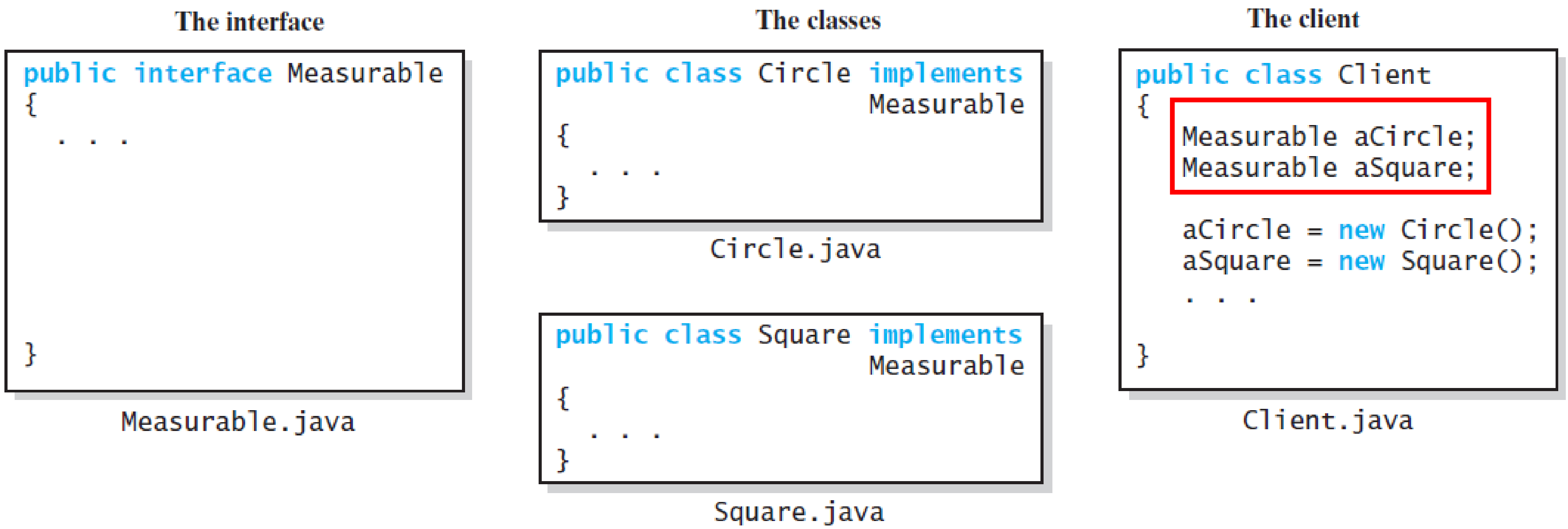
```
public class Client
{
   Measurable aCircle;
   Measurable aSquare;

   aCircle = new Circle();
   aSquare = new Square();
   . . .
}
```

Java Interfaces

- **Implementing a Java Interface**

```
public class Circle implements Measurable

public class Square implements Measurable
```

An interface type is a reference type. You can use a Java interface as you would a data type



The interface

```
public interface Measurable
{
   . . .



}
```

Measurable.java

The classes

```
public class Circle implements
                    Measurable
{
   . . .
}
```

Circle.java

```
public class Square implements
                    Measurable
{
   . . .
}
```

Square.java

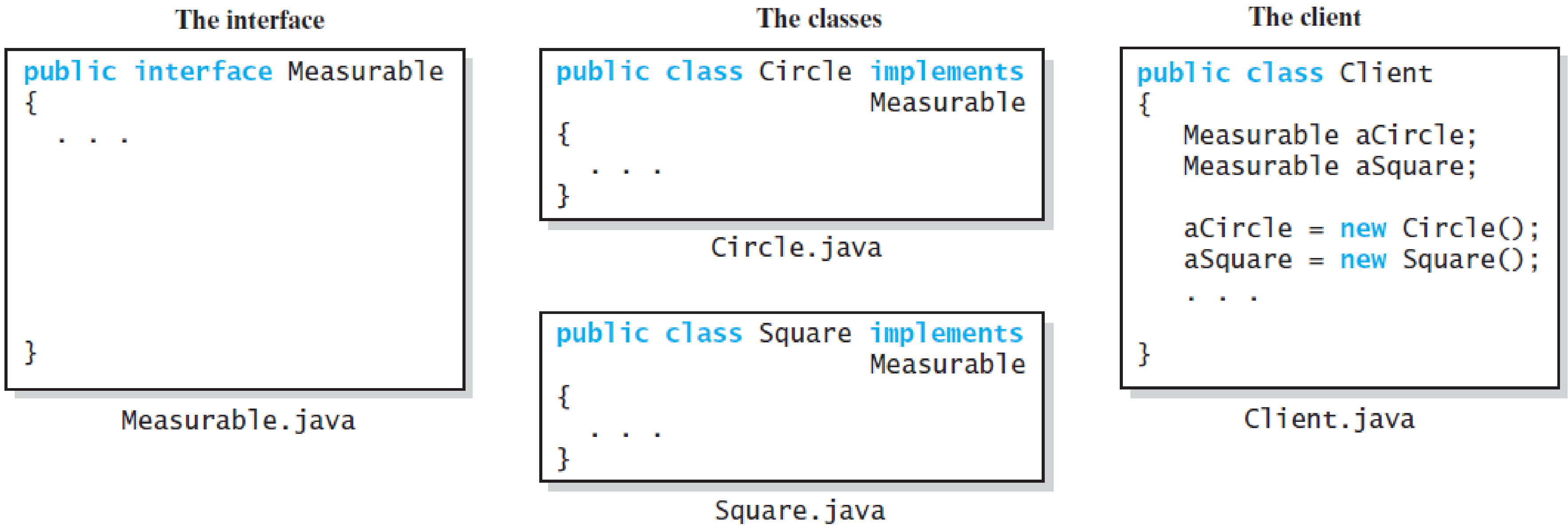
The client

```
public class Client
{
   Measurable aCircle;
   Measurable aSquare;

   aCircle = new Circle();
   aSquare = new Square();
   . . .

}
```

Client.java

Java Interfaces

- A way for programmer to guarantee a class has certain methods
  - Several classes can implement the same interface
  - A class can implement more than one interface
  - An interface can be used to derive another interface by using inheritance



The interface

```java
public interface Measurable
{
  . . .



}
```

Measurable.java

The classes

```java
public class Circle implements
                        Measurable
{
   . . .
}
```

Circle.java

```java
public class Square implements
                        Measurable
{
   . . .
}
```

Square.java

The client

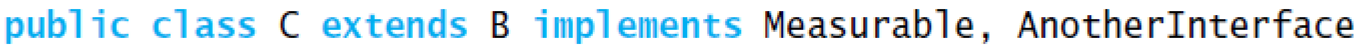
```java
public class Client
{
   Measurable aCircle;
   Measurable aSquare;

   aCircle = new Circle();
   aSquare = new Square();
   . . .
}
```

Client.java

Java Interfaces

- A way for programmer to guarantee a class has certain methods
  - Several classes can implement the same interface
  - A class can implement more than one interface
  - An interface can be used to derive another interface by using inheritance

- **Multiple interfaces**
  - If it does, you simply list all the interface names, separated by commas.
  - If the class is derived from another class, the implements clause always follows the extends clause



```
public class C extends B implements Measurable, AnotherInterface
```

Note: Interface vs. Abstract Class

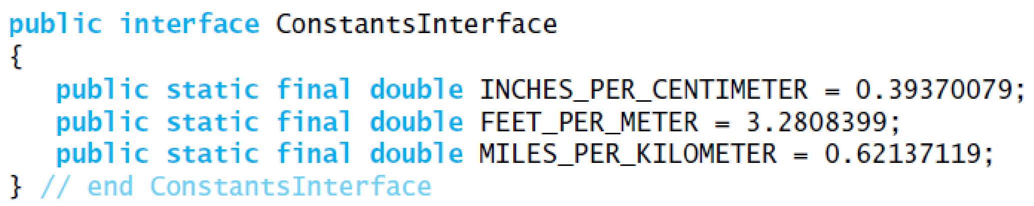
- Purpose of interface similar to that of abstract class
  - But an interface is not a class

- Use an abstract class …
  - If you want to provide a method definition
  - Or declare a private data field that your classes will have in common

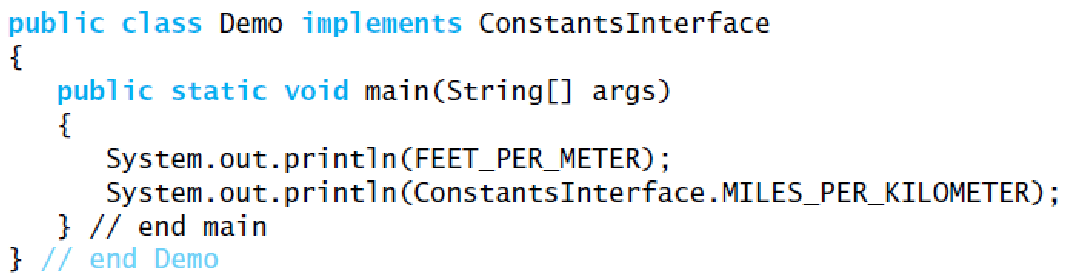
- A class can implement several interfaces but can extend only one abstract class.

Java Interfaces

- **Named Constants Within an Interface**
  - Public data fields that you initialize and declare as ***final***.



```
public interface ConstantsInterface
{
   public static final double INCHES_PER_CENTIMETER = 0.39370079;
   public static final double FEET_PER_METER = 3.2808399;
   public static final double MILES_PER_KILOMETER = 0.62137119;
} // end ConstantsInterface
```

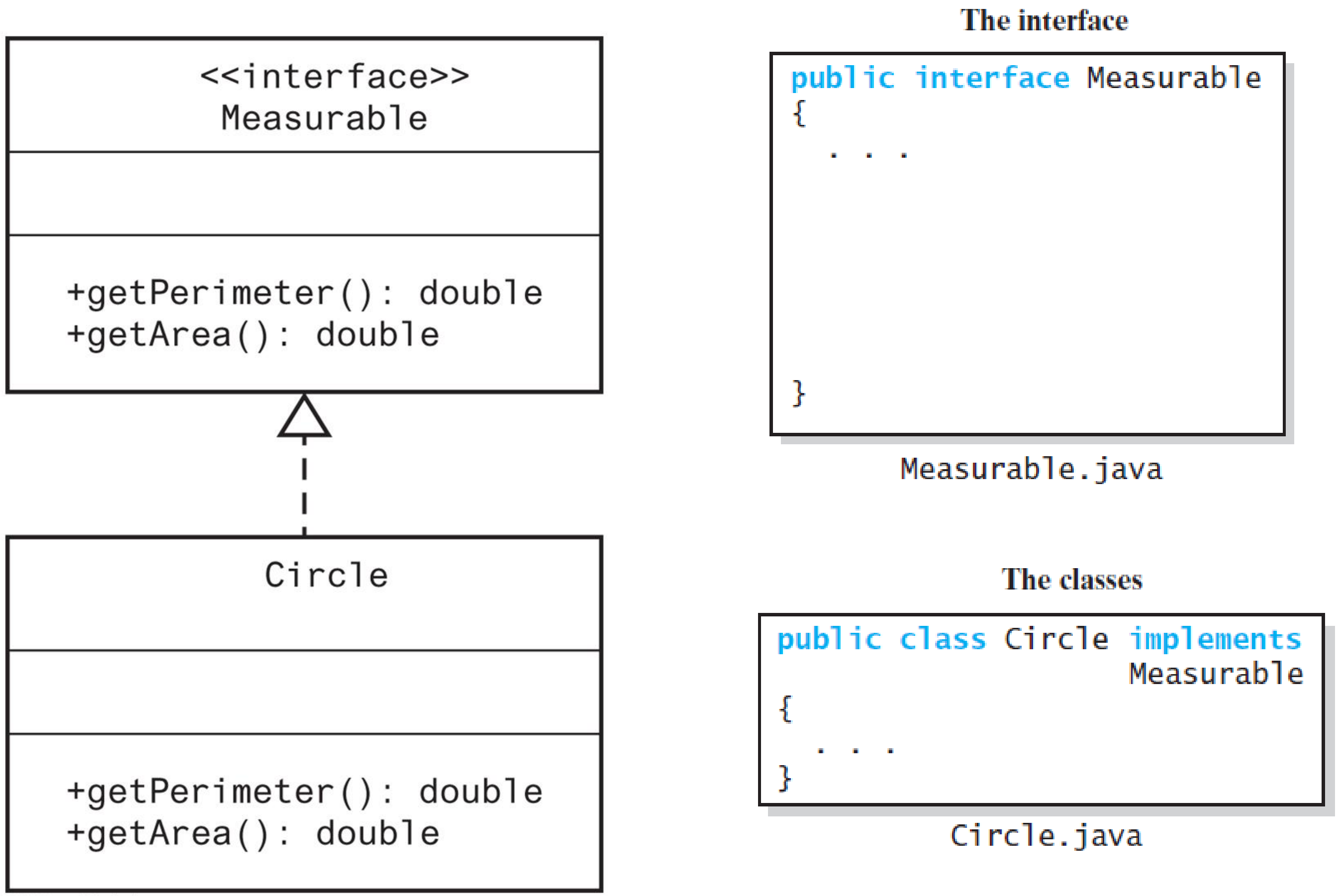


```
public class Demo implements ConstantsInterface
{
   public static void main(String[] args)
   {
      System.out.println(FEET_PER_METER);
      System.out.println(ConstantsInterface.MILES_PER_KILOMETER);
   } // end main
} // end Demo
```

Review: Java Programming

- **Unified Modeling Language (UML)**

+ for public
- for private
# for protected



The interface

```
public interface Measurable
{
   . . .




}
```

Measurable.java

The classes

```
public class Circle implements
                         Measurable
{
   . . .
}
```

Circle.java

Generic Data Types

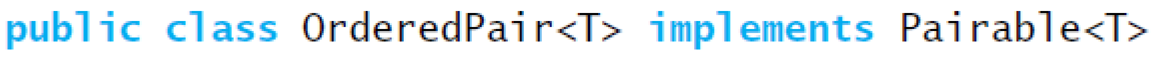
- Generics enable you to write a placeholder (a generic data type) instead of an actual class type.

- Using generics, you can define a class of objects whose data type is determined later by the client of your class
  - You define a generic class
  - Client chooses data type of the objects in collection.

Generic Data Types

- **Generic Types Within an Interface**
  - Write an identifier **T**, enclosed in **angle brackets after** the **name of an interface**.

- Implementing the generic interface in a **generic class**
  - Write an identifier **T**, enclosed in **angle brackets after** the **name of the a class**



```
public class OrderedPair<T> implements Pairable<T>
```

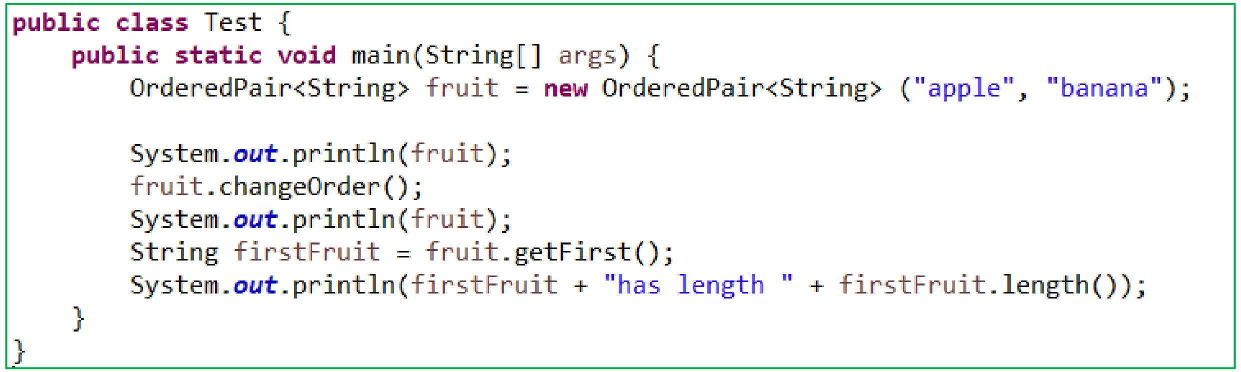
```
/**
   An interface for pairs of objects.
*/
public interface Pairable<T>
{
  public T getFirst();
  public T getSecond();
  public void changeOrder();
} // end Pairable
```

Programming Exercises

```
/**
   An interface for pairs of objects.
*/
public interface Pairable<T>
{
   public T getFirst();
   public T getSecond();
   public void changeOrder();
} // end Pairable
```

Pairable.java

Test.java



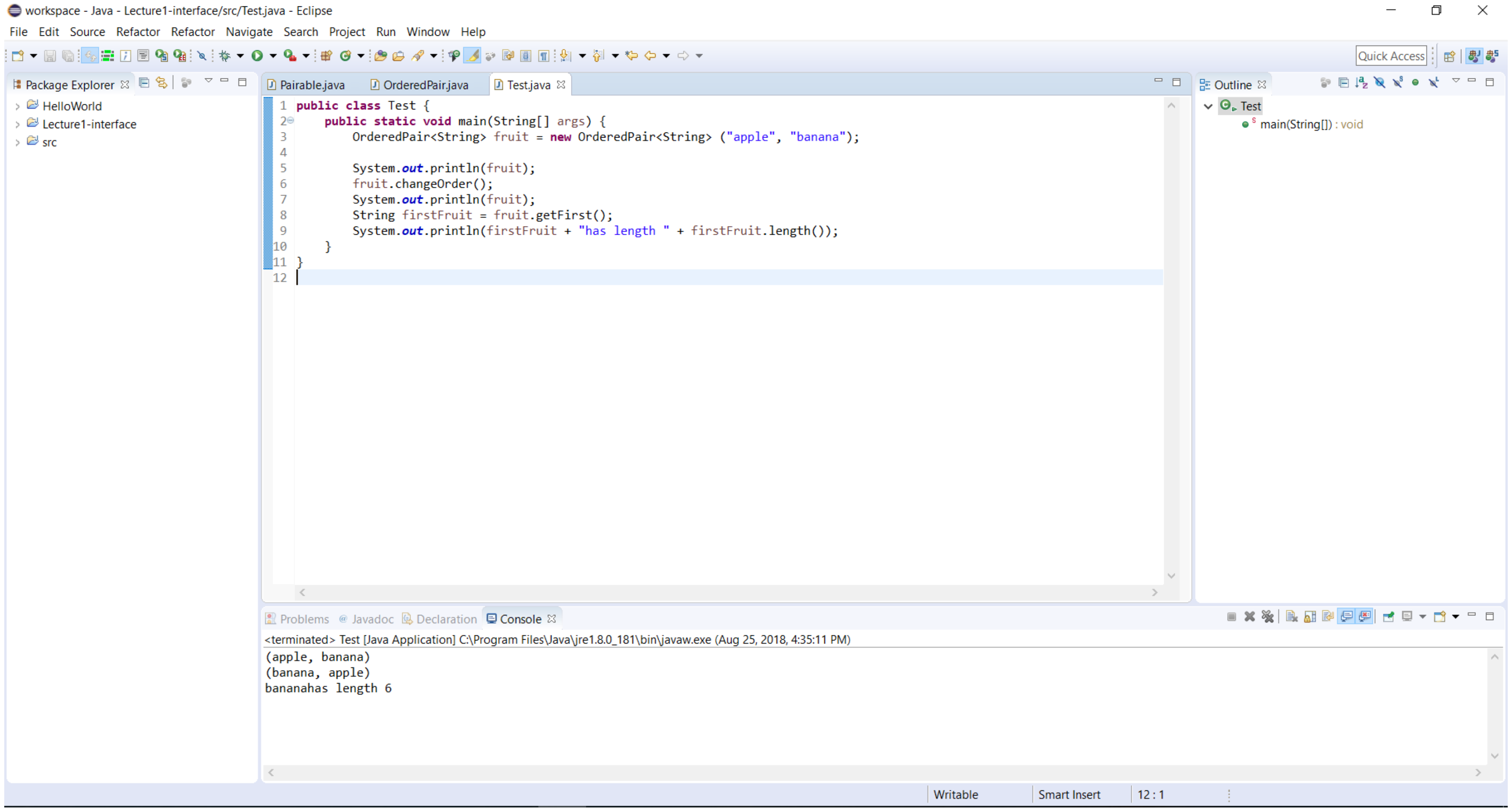
```
public class Test {
    public static void main(String[] args) {
        OrderedPair<String> fruit = new OrderedPair<String> ("apple", "banana");

        System.out.println(fruit);
        fruit.changeOrder();
        System.out.println(fruit);
        String firstFruit = fruit.getFirst();
        System.out.println(firstFruit + "has length " + firstFruit.length());
    }
}
```

(apple, banana)
(banana, apple)
Banana has length 6

```
/** A class of ordered pairs of objects having the same data type. */
public class OrderedPair<T> implements Pairable<T>
{
  private T first, second;

  public OrderedPair(T firstItem, T secondItem)
        // NOTE: no <T> after constructor name
  {
    first = firstItem;
    second = secondItem;
  } // end constructor

  /** Returns the first object in this pair. */
  public T getFirst()
  {
    return first;
  } // end getFirst

  /** Returns the second object in this pair. */
  public T getSecond()
  {
    return second;
  } // end getSecond

  /** Returns a string representation of this pair. */
  public String toString()
  {
    return "(" + first + ", " + second + ")";
  } // end toString

  /** Interchanges the objects in this pair. */
  public void changeOrder()
  {
    T temp = first;
    first = second;
    second = temp;
  } // changeOrder
} // end OrderedPair
```

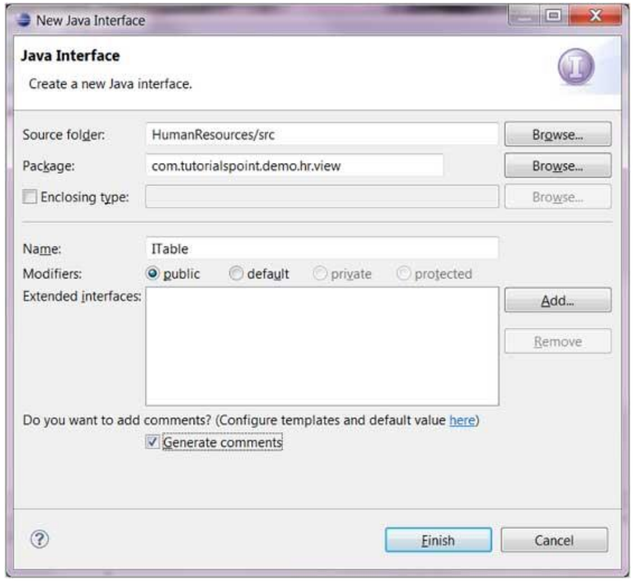
OrderedPair.java

Readings: https://www.tutorialspoint.com/eclipse/index.htm



```
public class Test {
    public static void main(String[] args) {
        OrderedPair<String> fruit = new OrderedPair<String> ("apple", "banana");

        System.out.println(fruit);
        fruit.changeOrder();
        System.out.println(fruit);
        String firstFruit = fruit.getFirst();
        System.out.println(firstFruit + "has length " + firstFruit.length());
    }
}
```

```
<terminated> Test [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (Aug 25, 2018, 4:35:11 PM)
(apple, banana)
(banana, apple)
bananahas length 6
```



Java Interface

Create a new Java interface.

Source folder: HumanResources/src

Package: com.tutorialspoint.demo.hr.view

□ Enclosing type:

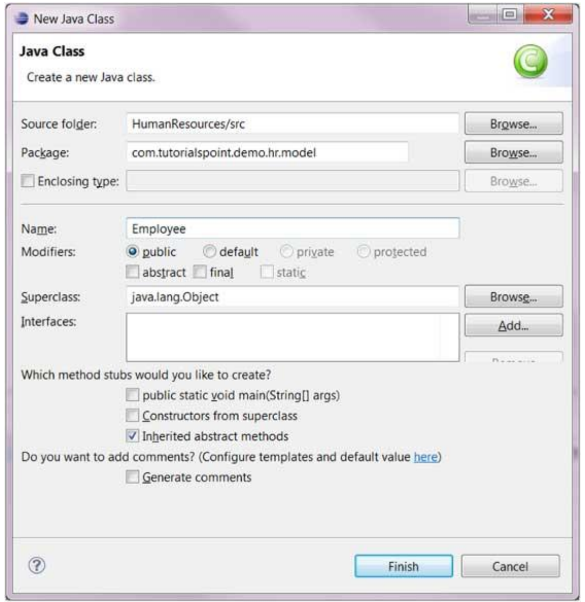
Name: ITable

Modifiers: ● public ○ default ○ private ○ protected

Extended interfaces:

Do you want to add comments? (Configure templates and default value here)

☑ Generate comments



Java Class

Create a new Java class.

Source folder: HumanResources/src

Package: com.tutorialspoint.demo.hr.model

□ Enclosing type:

Name: Employee

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

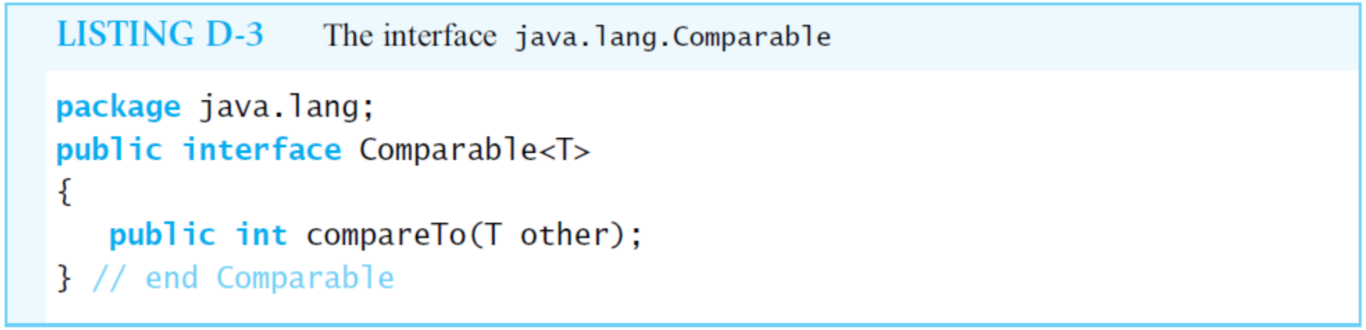
☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

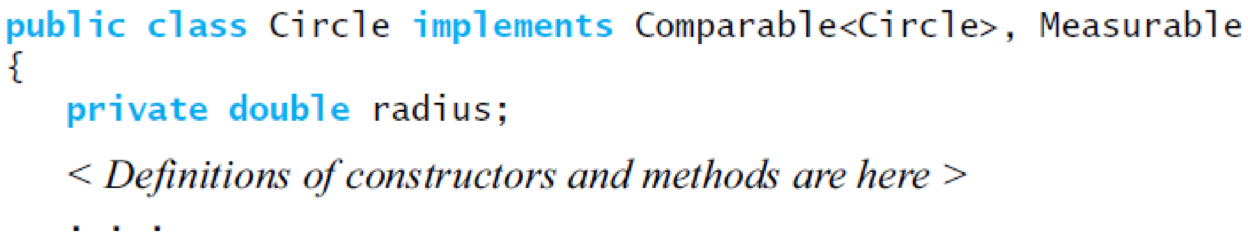
More on Generics

- All classes that define the method **compareTo** implement the standard interface Comparable, which is in the Java Class Library in the package java.lang

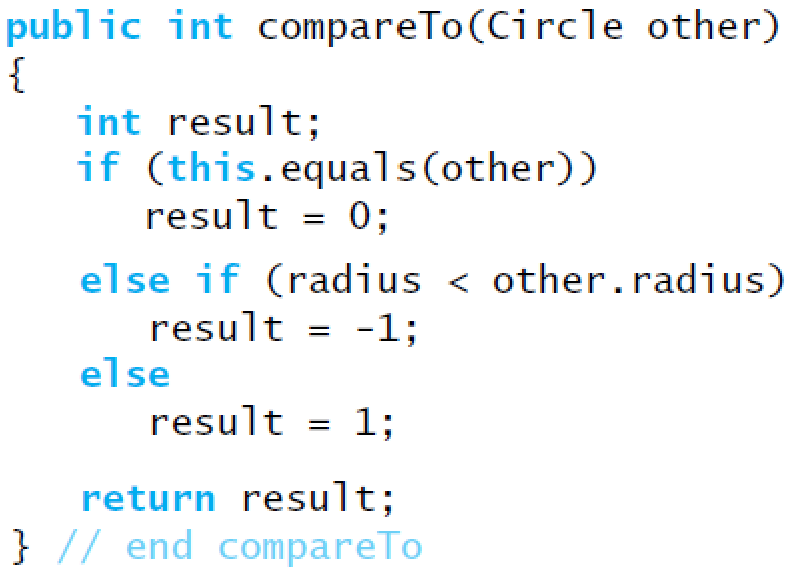


LISTING D-3 The interface `java.lang.Comparable`

```
package java.lang;
public interface Comparable<T>
{
   public int compareTo(T other);
} // end Comparable
```



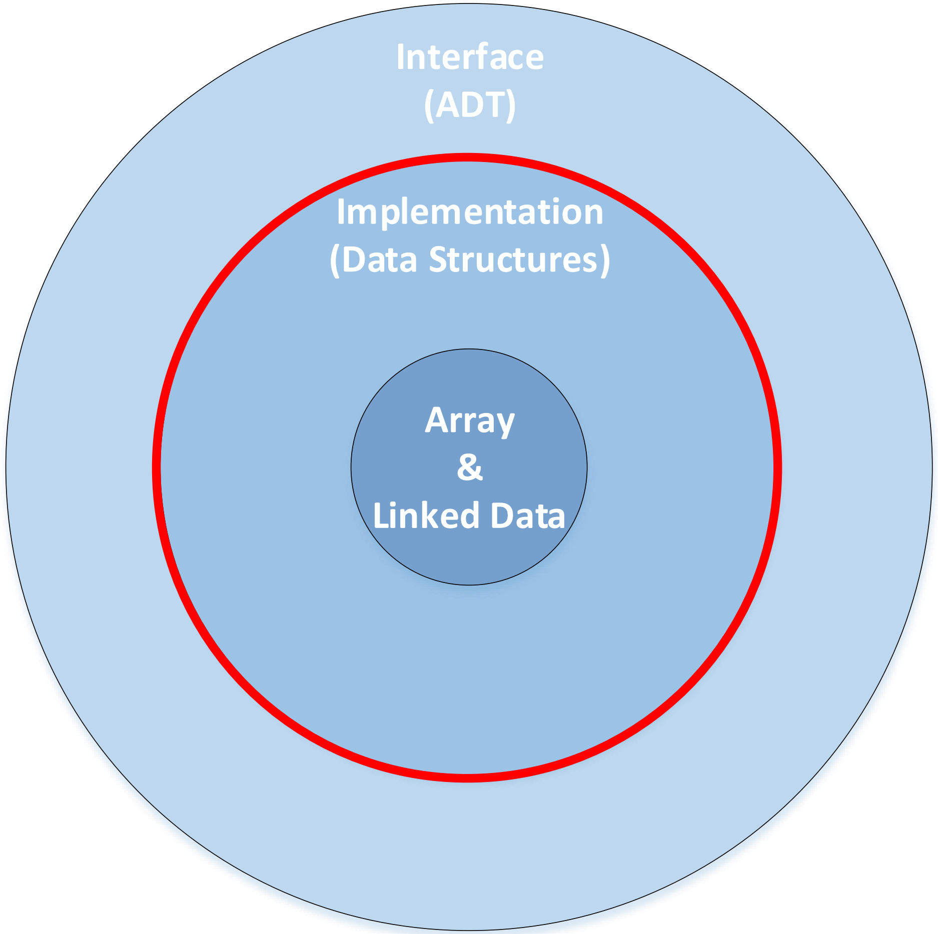
```
public class Circle implements Comparable<Circle>, Measurable
{
   private double radius;

   < Definitions of constructors and methods are here >
   . . .
```



```
public int compareTo(Circle other)
{
   int result;
   if (this.equals(other))
      result = 0;
   else if (radius < other.radius)
      result = -1;
   else
      result = 1;

   return result;
} // end compareTo
```

Data Structures (Cont'd)

- An ADT makes a clean separation between interface and implementation
  - The user only sees the interface and therefore does not need to tamper with the implementation.
  - The abstraction makes the code more robust and easier to maintain.

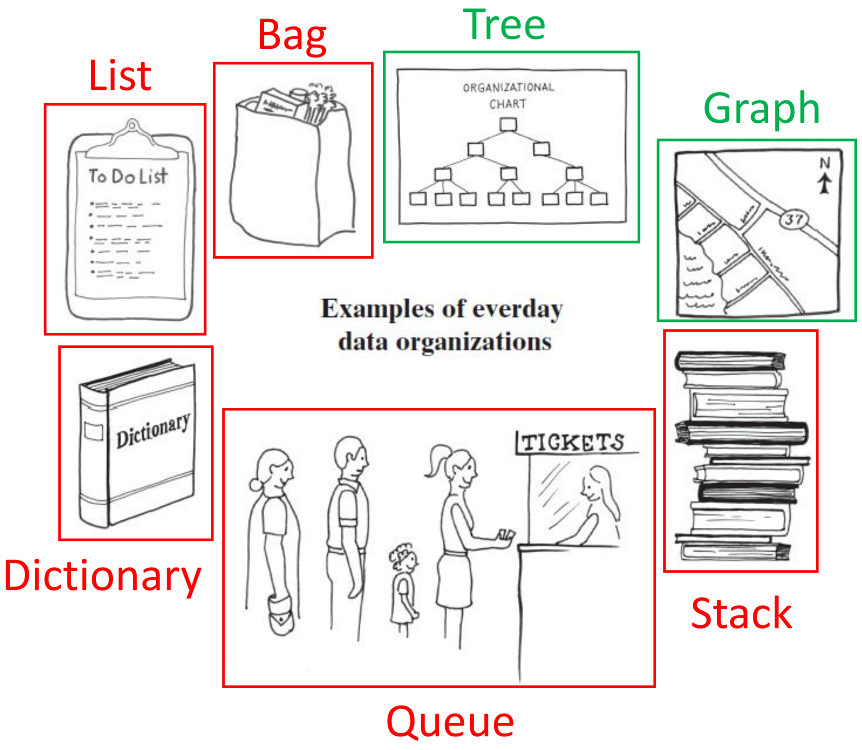


Data Structures (Cont'd)

- **Collection:** a general term of an ADT that contains a group of objects

- Bag, List, Queue, Stack, and Dictionary
  - Each of them is a collection that stores its entries in a linear sequence, and in which entries may be added or removed at will.
  - They differ in the restrictions they place on how these entries may be added, removed, or accessed.

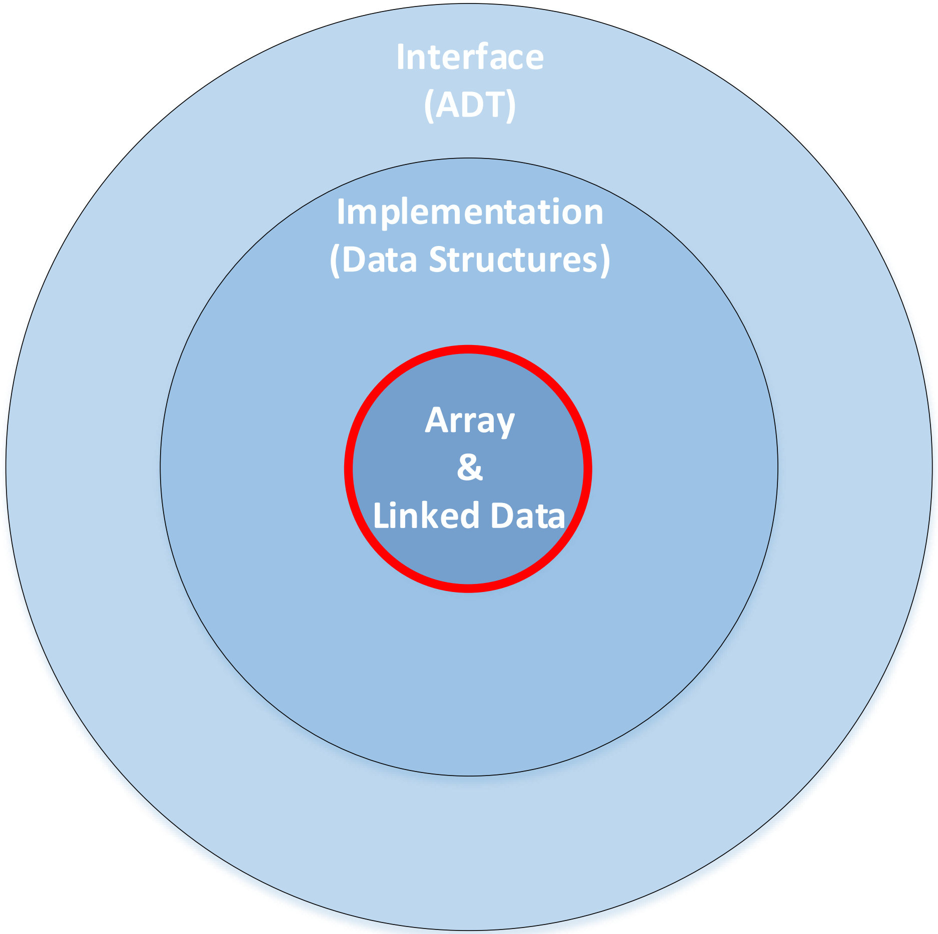
- Tree and Graph
  - Data entries are not arranged in a sequence, but with different rules



Examples of everday data organizations

Data Structures (Cont'd)

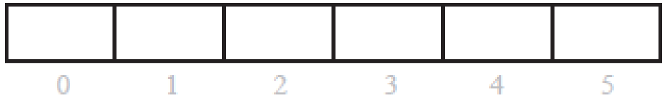
- An ADT makes a clean separation between interface and implementation
  - The user only sees the interface and therefore does not need to tamper with the implementation.
  - The abstraction makes the code more robust and easier to maintain.
- We will use Array or Linked Data to implement different data structures in Java
- We will perform efficiency analysis.



Data Structures (Cont’d)

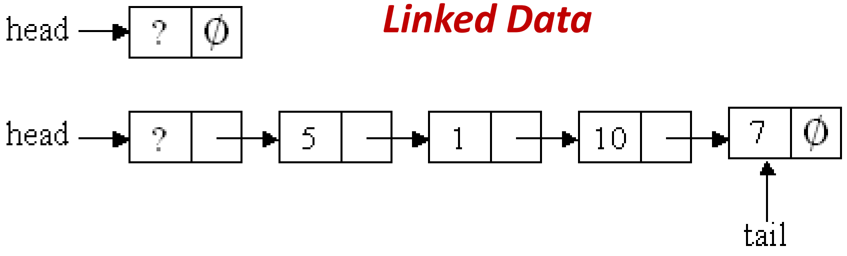
- Array
  - Entries stored at contiguous memory locations.
  - All entries are the same data type.

Array



- Linked Data
  - Entries are not stored in contiguous locations.
  - Every entry is a separate node with a data field and a reference field. The entries are linked using reference field.

Linked Data



Review: Java Programming

- An array is a collection of items stored at contiguous memory locations.

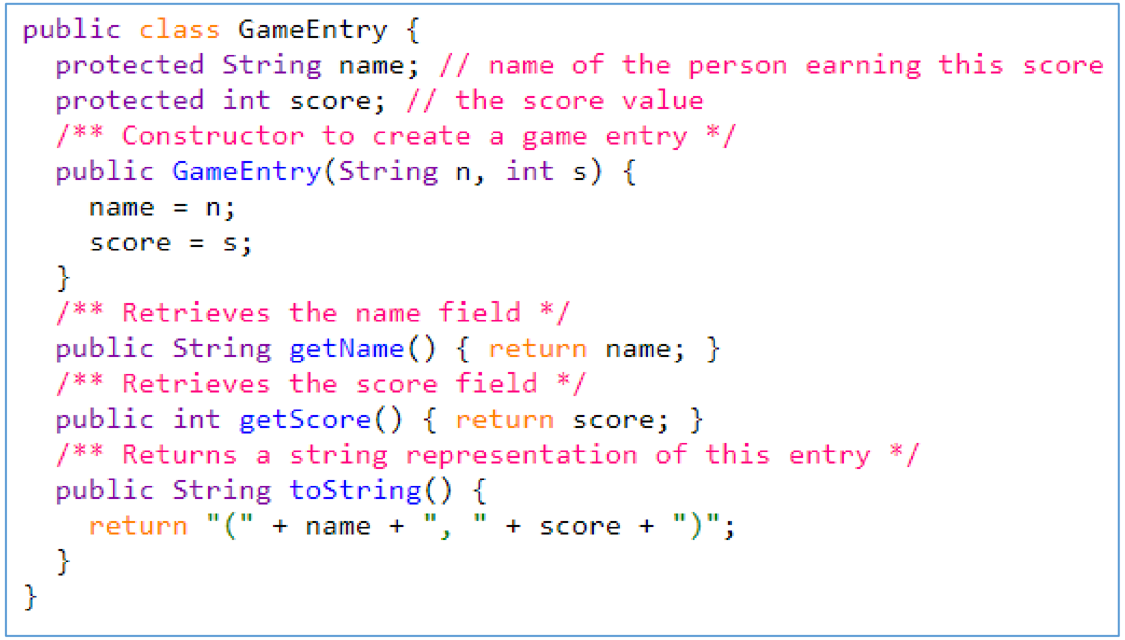
- Arrays in Java work differently than they do in C/C++:
  - In Java all arrays are dynamically allocated.
  - Since arrays are objects in Java, we can find their length using member *length*.
  - Once an array is created, its size cannot be changed.
  - A Java array variable can also be declared like other variables with [] after the data type.
  - Each item in an array has an index beginning from 0.
  - The direct superclass of an array type is Object.

```
int[] numbers;
numbers = new int[6];
          or
int[] numbers = new int[6];
           or
int[] numbers = {1, 2, 3, 4, 5, 6};
```

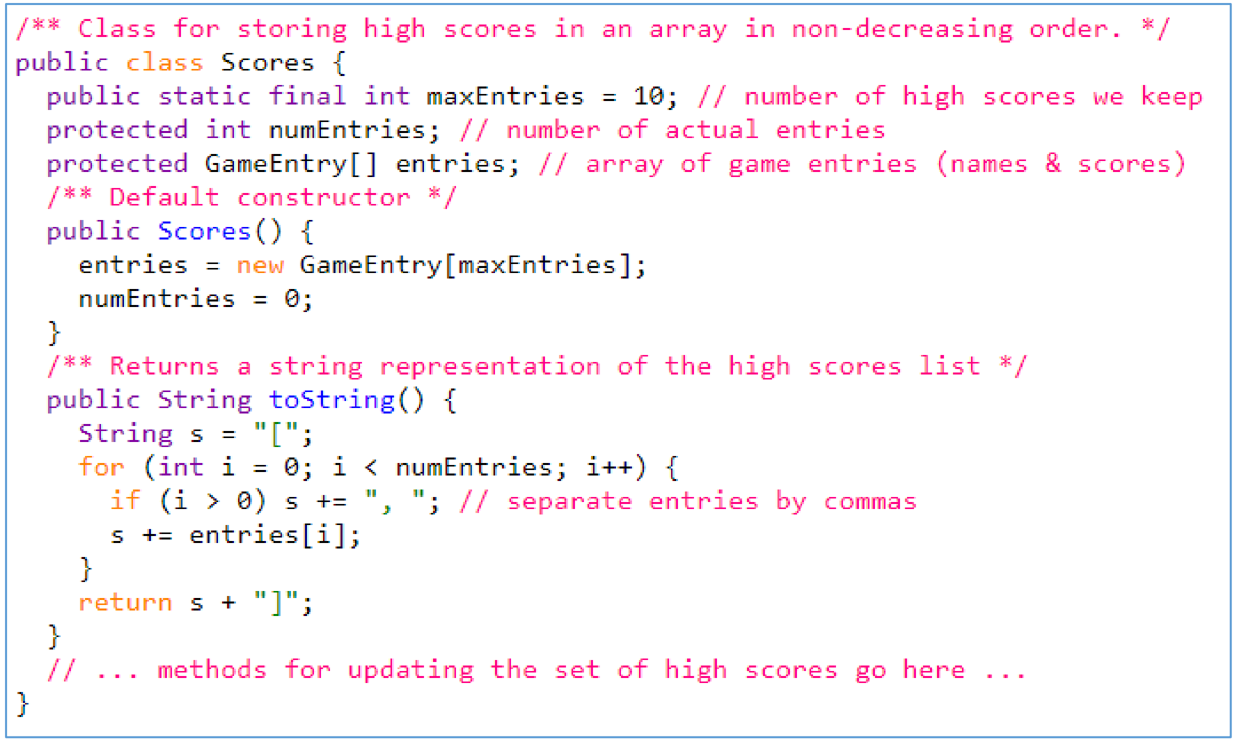
https://www.cpp.edu/~ftang/courses/CS240/lectures/array.htm
https://www.geeksforgeeks.org/arrays-in-java/

Review: Java Programming

- Code Sample: Storing Game Entries in an Array



```java
public class GameEntry {
  protected String name; // name of the person earning this score
  protected int score; // the score value
  /** Constructor to create a game entry */
  public GameEntry(String n, int s) {
    name = n;
    score = s;
  }
  /** Retrieves the name field */
  public String getName() { return name; }
  /** Retrieves the score field */
  public int getScore() { return score; }
  /** Returns a string representation of this entry */
  public String toString() {
    return "(" + name + ", " + score + ")";
  }
}
```



```java
/** Class for storing high scores in an array in non-decreasing order. */
public class Scores {
  public static final int maxEntries = 10; // number of high scores we keep
  protected int numEntries; // number of actual entries
  protected GameEntry[] entries; // array of game entries (names & scores)
  /** Default constructor */
  public Scores() {
    entries = new GameEntry[maxEntries];
    numEntries = 0;
  }
  /** Returns a string representation of the high scores list */
  public String toString() {
    String s = "[";
    for (int i = 0; i < numEntries; i++) {
      if (i > 0) s += ", "; // separate entries by commas
      s += entries[i];
    }
    return s + "]";
  }
  // ... methods for updating the set of high scores go here ...
}
```

https://www.cpp.edu/~ftang/courses/CS240/lectures/array.htm
https://www.geeksforgeeks.org/arrays-in-java/

Review: Java Programming

- Java provides a number of built-in methods for performing common tasks on arrays. These methods are static methods in the **java.util.Arrays** class.

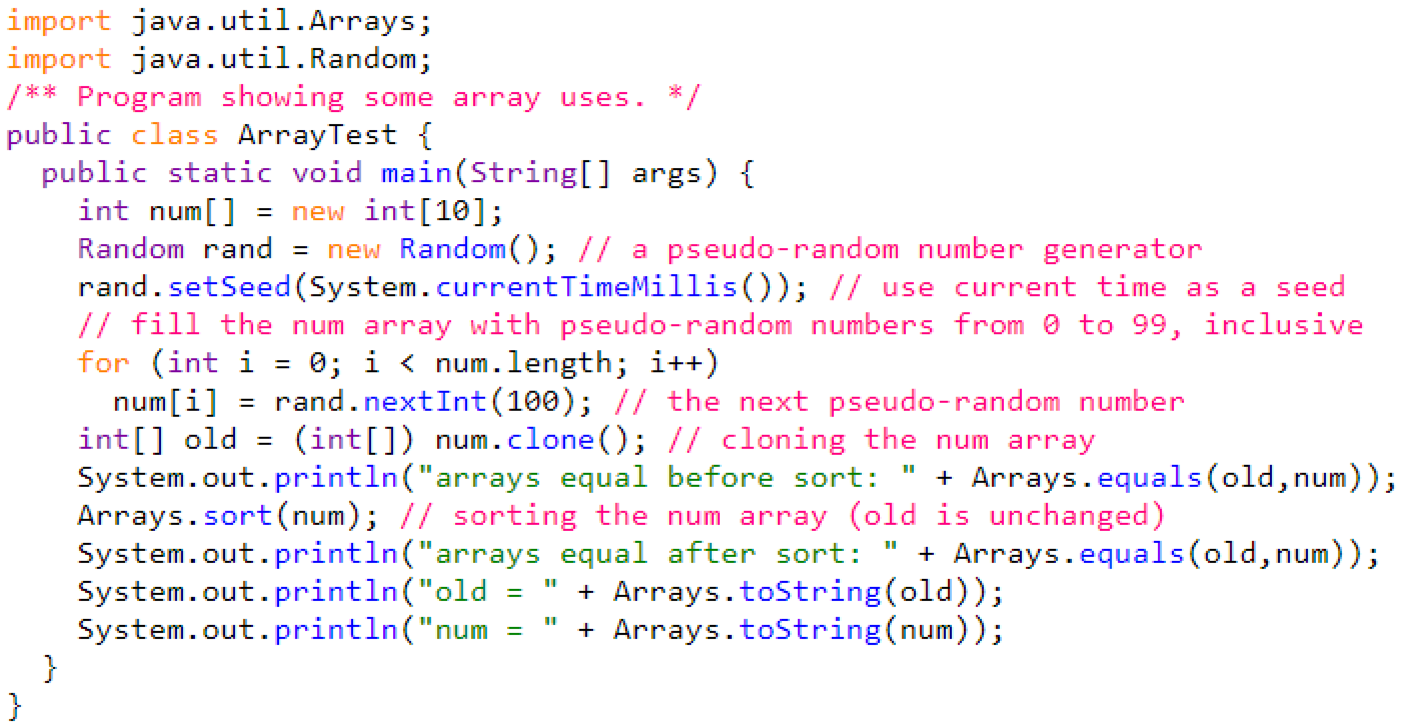
- Some simple methods are:
  - **equals(A, B):** returns true if the array A and the array B are equal, which means they have the same number of elements and every corresponding pair of elements in the two arrays are equal.
  - **fill(A, x)**: stores element x into every cell of array A.
  - **sort(A)**: sorts the array A using the natural ordering of its elements. For example, ascending numerical order. A tuned quicksort.
  - **toString(A)**: returns a String representation of the array A.
  - **A.clones():** creates a complete copy of an array A.

https://www.cpp.edu/~ftang/courses/CS240/lectures/array.htm

Review: Java Programming

https://www.cpp.edu/~ftang/courses/CS240/lectures/array.htm

- An example that uses various built-in methods of the Arrays class.



```
import java.util.Arrays;
import java.util.Random;
/** Program showing some array uses. */
public class ArrayTest {
  public static void main(String[] args) {
    int num[] = new int[10];
    Random rand = new Random(); // a pseudo-random number generator
    rand.setSeed(System.currentTimeMillis()); // use current time as a seed
    // fill the num array with pseudo-random numbers from 0 to 99, inclusive
    for (int i = 0; i < num.length; i++)
      num[i] = rand.nextInt(100); // the next pseudo-random number
    int[] old = (int[]) num.clone(); // cloning the num array
    System.out.println("arrays equal before sort: " + Arrays.equals(old,num));
    Arrays.sort(num); // sorting the num array (old is unchanged)
    System.out.println("arrays equal after sort: " + Arrays.equals(old,num));
    System.out.println("old = " + Arrays.toString(old));
    System.out.println("num = " + Arrays.toString(num));
  }
}
```

Readings:

- https://www.geeksforgeeks.org/array-class-in-java/
- https://www.geeksforgeeks.org/arraylist-in-java/

Summary

- Course Information
- Course outline
- Introduction (including a quick review on Java programming)

What I Want You to Do

- Review class slides
- Review Appendix B, Prelude (Designing Classes), Appendix C, and Interlude 1

- Next Topic
    - Bags