http://www.tutorialspoint.com/java/java_overriding.htm

JAVA- PROGRAMMING

CH2-EX1

```
package javaapplication1;

public class ch1_ex1 {

   //main method
      public static void main(String[] args) {
       System.out.print("hello world!_");  // System.out meaning object, println is
command
       System.out.println("welcome to java");
       System.out.println("welcome \n to \n java");
      System.out.printf("%s\n%s\n", "welcome to" , "java program");
      /**System.out.printf method
       f means “formatted”
       displays formatted data  , fixed text using print or println
       and format specifiers  using percent sign (%). */
         } //end method main
   }   // end class ch1_ex1
```

---

run:
hello world!_welcome to java
welcome
to
java
welcome to
java program
BUILD SUCCESSFUL (total time: 1 second)

CH2-EX2 (SUM two numbers and IF statement)

```
package javaapplication2;
// Fig. 2.7: Addition.java
// Addition program that displays the sum of two numbers.
import java.util.Scanner; // program uses class Scanner
public class CH2_EX2
{
  // main method begins execution of Java application
  public static void main( String[] args )
  {
    // create a Scanner to obtain input from the command window
    Scanner input = new Scanner( System.in );

    int number1; // first number to add
    int number2; // second number to add
    int sum; // sum of number1 and number2

    System.out.print( "Enter first integer: " ); // prompt
    number1 = input.nextInt(); // read first number from user

    System.out.print( "Enter second integer: " ); // prompt
    number2 = input.nextInt(); // read second number from user

    sum = number1 + number2; // add numbers, then store total in sum

    System.out.printf( "Sum is %d\n", sum ); // display sum as decimal number
if ( number1 == number2 )
      System.out.printf( "%d == %d\n", number1, number2 );
    if ( number1 != number2 )
      System.out.printf( "%d != %d\n", number1, number2 );
    if ( number1 < number2 )
      System.out.printf( "%d < %d\n", number1, number2 );
    if ( number1 > number2 )
      System.out.printf( "%d > %d\n", number1, number2 );
    if ( number1 <= number2 )
      System.out.printf( "%d <= %d\n", number1, number2 );
    if ( number1 >= number2 )
      System.out.printf( "%d >= %d\n", number1, number2 );
  } // end method main
} // end class Addition
```

```
----------------------------------------------------------
run:
Enter first integer: 3
Enter second integer: 5
Sum is 8
3 != 5
3 < 5
3 <= 5
BUILD SUCCESSFUL (total time: 11 seconds)
```

CH3-EX1 : 3- Introduction to Classes, Objects, Methods and Strings
Class Gradebook and GradeBookTest

```
public class GradeBookTest
{
  // main method begins program execution
  public static void main( String[] args )
  {
    // create a GradeBook object and assign it to myGradeBook
    GradeBook myGradeBook = new GradeBook();

    // call myGradeBook's displayMessage method
    myGradeBook.displayMessage();
  } // end main
} // end class GradeBookTest
```

A **static** method (such as main) is special

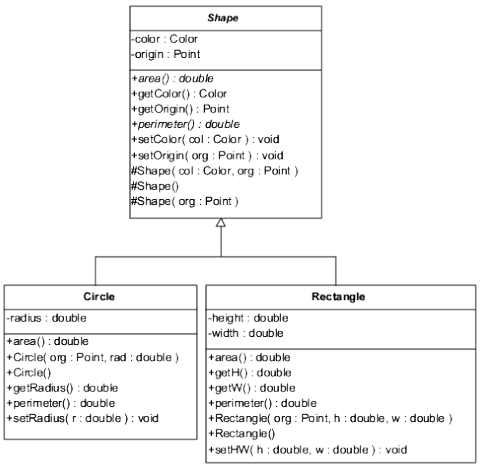
It can be called **without first creating an object** of the class in which the method is declared.

Typically, you **cannot call** a method that belongs to another class until you **create an object of that class**.

```
// -------------------------------------------------------------------------------
public class GradeBook
{
  // display a welcome message to the GradeBook user
  public void displayMessage()
  {
    System.out.println( "Welcome to the Grade Book!" );
  } // end method displayMessage
} // end class GradeBook
```

Parentheses in combination with a class name represent a call to a ***constructor***, **which is similar to a method but is used only at the time an object is created to initialize the object's data.**

```
run:
Welcome to the Grade Book!
BUILD SUCCESSFUL (total time: 0 seconds)
```



- **UML class diagram**
- Three compartments.

Top: contains the class name centered horizontally in boldface type.

Middle: contains the class's attributes, which correspond to instance variables

Bottom: contains the class's operations, which correspond to methods.

The plus sign (+) corresponds to the keyword **public**

CH3_EX2 (using parameters with class and methods)

```
import java.util.Scanner; // program uses Scanner

public class GradeBookTest
{
  // main method begins program execution
  public static void main( String[] args )
  {
    // create Scanner to obtain input from command window
    Scanner input = new Scanner( System.in );

    // create a GradeBook object and assign it to myGradeBook
    GradeBook myGradeBook = new GradeBook();

    // prompt for and input course name
    System.out.println( "Please enter the course name:" );
    String courseName = input.nextLine(); // read a line of text (string text)
    System.out.println(); // outputs a blank line

    // call myGradeBook's displayMessage method
    // and pass courseName as an argument
    myGradeBook.displayMessage( courseName );
  } // end main
} // end class GradeBookTest

--------------------------------------------------------------------------
public class GradeBook
{
  // display a welcome message to the GradeBook user
  public void displayMessage( String courseName )
  {
    System.out.printf( "Welcome to the grade book for\n%s!\n",
      courseName );
  } // end method displayMessage
} // end class GradeBook

---------------------------------
run:
Please enter the course name:
java how to program

Welcome to the grade book for
java how to program!
BUILD SUCCESSFUL (total time: 24 seconds)
```

Parameter: Additional information a method needs to perform its task.

A method call supplies values—called **arguments**—for each of the method's parameters.

More on Arguments and Parameters

The number of arguments in a method call **must match the number** of parameters in the parameter list of the method's declaration.

The **argument types** in the method call must be "**consistent with**" the types of the **corresponding parameters** in the method's declaration.

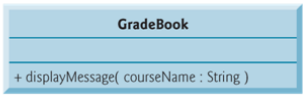


Fig. 3.6 | UML class diagram indicating that class `GradeBook` has a `displayMessage` operation with a `courseName` parameter of UML type `String`.

Notes on **import** Declarations

Classes that are compiled in the same directory on disk are in the same **package**—known as the **default package.**

Classes System and String are in package **java.lang**

Package name followed by a dot **(.)** and the **class name.**

CH3_EX3

Instance Variables, *set Methods and get Methods*

A class normally consists of one or more methods that manipulate the attributes that belong to a particular object of the class.

Attributes are represented as variables in a class declaration. Called **fields**.

Declared **inside a class** declaration but **outside the bodies** of the **class's method** declarations.

Instance variable

When each object of a class maintains its own copy of an attribute, the field is an **instance variable**

Each object (**instance**) of the class has a separate instance of the variable in memory.

```
import java.util.Scanner; // program uses Scanner

public class GradeBookTest
{
  // main method begins program execution
  public static void main( String[] args )
  {
    // create Scanner to obtain input from command window
    Scanner input = new Scanner( System.in );

    // create a GradeBook object and assign it to myGradeBook
    GradeBook  myGradeBook = new GradeBook();

    // display initial value of courseName
    System.out.printf( "Initial course name is: %s\n\n",
      myGradeBook.getCourseName() );

    // prompt for and read course name
    System.out.println( "Please enter the course name:" );
    String theName = input.nextLine(); // read a line of text
    myGradeBook.setCourseName( theName ); // set the course name
    System.out.println(); // outputs a blank line

    // display welcome message after specifying course name
    myGradeBook.displayMessage();
  } // end main
} // end class GradeBookTest
```

```
// Fig. 3.7: GradeBook.java
// GradeBook class that contains a courseName instance variable
// and methods to set and get its value.

public class GradeBook
{
  private String courseName; // course name for this GradeBook
  // method to set the course name
  public void setCourseName( String name )
  {
    courseName = name; // store the course name
  } // end method setCourseName

  // method to retrieve the course name
  public String getCourseName()
  {
    return courseName;
  } // end method getCourseName
  // display a welcome message to the GradeBook user
```

```
 public void displayMessage()
  {
    // calls getCourseName to get the name of
    // the course this GradeBook represents
    System.out.printf( "Welcome to the grade book for\n%s!\n",
      getCourseName() );
  } // end method displayMessage
} // end class GradeBook
```

```
run:
Initial course name is: null

Please enter the course name:
JAVA PROGRAMMING

Welcome to the grade book for
JAVA PROGRAMMING!
BUILD SUCCESSFUL (total time: 12 seconds)
```

Every **instance (i.e., object)** of a class contains one copy of each instance variable.

Instance variables typically declared **private**.

private is an access modifier.

private variables and methods are accessible only to methods of the class **in which they are declared**.

Declaring **instance private** is known as **data hiding** or **information hiding**.

Set and ***get*** methods used **to access instance variables**

A minus sign **(–)** access modifier corresponds to access modifier **private.**

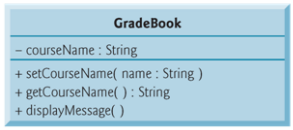


Fig. 3.9 | UML class diagram indicating that class `GradeBook` has a private `courseName` attribute of UML type `String` and three public operations—`setCourseName` (with a `name` parameter of UML type `String`), `getCourseName` (which returns UML type `String`) and `displayMessage`.

One method of a class can **call another method** of the same class by **using just the method name.**

EX: **getCourseName()**

every field has a **default initial value**—a value provided by **Java**

- The **default value** for a field of type **String** is **null**

Classes often provide **public** methods to allow clients to ***set*** **(i.e., assign values to) or *get* (i.e., obtain the values of) private** instance variables.

The names of these methods **need not begin** with *set* or *get*, but this naming **convention is recommended**.

3.5 Primitive Types vs. Reference Types

- Types are divided into **primitive types and reference types.**
- The **primitive types** are **boolean, byte, char, short, int, long, float and double**.
- All nonprimitive types are reference types.
- A primitive-type variable can store exactly one value of its declared type at a time.
- **Primitive-type instance variables are initialized by default—variables of types byte, char, short, int, long, float and double are initialized to 0**, and variables of type **boolean are initialized to false**.

You can specify your own initial value for a primitive-type variable by assigning the variable a value in its declaration

3.6 Initializing Objects with Constructors

- When an object of a class is created, its instance variables are initialized by default.
- Each class can provide a constructor that initializes an object of a class when the object is created.
- Java requires a constructor call for *every* object that is created.
- Keyword new requests memory from the system to store an object, then calls the corresponding class's constructor to initialize the object.

A constructor *must* have the same name as the class.

Initializing Objects with Constructors (Cont.)

- **By default, the compiler provides a default constructor with no parameters in any class that does not explicitly include a constructor**.
  - **Instance variables are initialized to their default values**.
- Can provide your own constructor to specify custom initialization for objects of your class.
- A constructor's parameter list specifies the data it requires to perform its task.
- Constructors cannot return values, so they cannot specify a return type.
- Normally, constructors are declared public.
- *If you declare any constructors for a class, the Java compiler will not create a default constructor for that class.*

```
public class GradeBookTest
{
  // main method begins program execution
  public static void main( String[] args )
  {
    // create GradeBook object
    GradeBook gradeBook1 = new GradeBook(          //gradeboke1 and gradebook2 are
                                                   //    reference variable
      "CS101 Introduction to Java Programming" );
    GradeBook gradeBook2 = new GradeBook(
      "CS102 Data Structures in Java" );

    // display initial value of courseName for each GradeBook
    System.out.printf( "gradeBook1 course name is: %s\n",
      gradeBook1.getCourseName() );
    System.out.printf( "gradeBook2 course name is: %s\n",
      gradeBook2.getCourseName() );
  } // end main
} // end class GradeBookTest
```

```
public class GradeBook
{
   private String courseName; // course name for this GradeBook

   // constructor initializes courseName with String argument
   public GradeBook( String name ) // constructor name is class name
   {
      courseName = name; // initializes courseName
   } // end constructor

     // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

} // end class GradeBook
```

Output

```
run:
gradeBook1 course name is: CS101 Introduction to Java Programming
gradeBook2 course name is: CS102 Data Structures in Java
BUILD SUCCESSFUL (total time: 0 seconds)
```

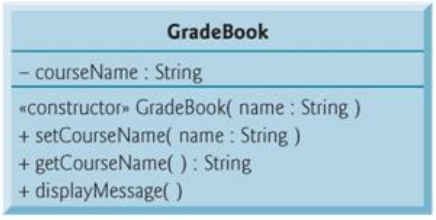


Fig. 3.12 | UML class diagram indicating that class `GradeBook` has a constructor that has a `name` parameter of UML type `String`.

Q1:Write a program in java language to design the following UML class with its object and variables

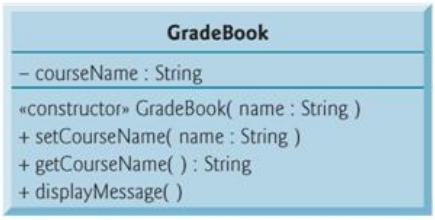


Fig. 3.12 | UML class diagram indicating that class `GradeBook` has a constructor that has a `name` parameter of UML type `String`.

The output should be as follow

run:
gradeBook1 course name is: CS101 Introduction to Java Programming
gradeBook2 course name is: CS102 Data Structures in Java
Initial course name is: java how to program

Please enter the course name:
java

Welcome to the grade book for
java!

ANSWER -CH3_EX5

```
import java.util.Scanner; // program uses Scanner

public class GradeBookTest
{
  // main method begins program execution
  public static void main( String[] args )
  {
    // create Scanner to obtain input from command window
    Scanner input = new Scanner( System.in );

    // create a GradeBook object and assign it to myGradeBook
    GradeBook myGradeBook = new GradeBook("java how to program");
// create GradeBook object
    GradeBook gradeBook1 = new GradeBook(
      "CS101 Introduction to Java Programming" );
    GradeBook gradeBook2 = new GradeBook(
      "CS102 Data Structures in Java" );

    // display initial value of courseName for each GradeBook
    System.out.printf( "gradeBook1 course name is: %s\n",
      gradeBook1.getCourseName() );
    System.out.printf( "gradeBook2 course name is: %s\n",
      gradeBook2.getCourseName() );
    // display initial value of courseName
    System.out.printf( "Initial course name is: %s\n\n",
```

```
      myGradeBook.getCourseName() );

    // prompt for and read course name
    System.out.println( "Please enter the course name:" );
    String theName = input.nextLine(); // read a line of text
    myGradeBook.setCourseName( theName ); // set the course name
    System.out.println(); // outputs a blank line

    // display welcome message after specifying course name
    myGradeBook.displayMessage();
  } // end main
} // end class GradeBookTest


----------------------------------
public class GradeBook
{
  private String courseName; // course name for this GradeBook
public GradeBook( String name ) // constructor name is class name
  {
    courseName = name; // initializes courseName
  } // end constructor

  // method to set the course name
  // method to set the course name
  public void setCourseName( String name )
  {
    courseName = name; // store the course name
  } // end method setCourseName

  // method to retrieve the course name
  public String getCourseName()
  {
    return courseName;
  } // end method getCourseName

  // display a welcome message to the GradeBook user
  public void displayMessage()
  {
    // calls getCourseName to get the name of
    // the course this GradeBook represents
    System.out.printf( "Welcome to the grade book for\n%s!\n",
      getCourseName() );
  } // end method displayMessage
} // end class GradeBook
```

3.7 Floating-Point Numbers and Type double

Q2: Q1:

Write a program in java language to design the following UML class with its object and variables

THE OUTPUT as follow using float and double variables

```
account1 balance: $50.00
account2 balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

account1 balance: $75.53
account2 balance: $0.00

Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

account1 balance: $75.53
account2 balance: $123.45
```

Fig. 3.14 | Inputting and outputting floating-point numbers with `Account` objects. (Part 3 of 3.)

Answer

```
import java.util.Scanner;
public class AccountTest
{
  // main method begins execution of Java application
  public static void main( String[] args )
  {
    Account account1 = new Account( 50.00 ); // create Account object
    Account account2 = new Account( -7.53 ); // create Account object

    // display initial balance of each object
    System.out.printf( "account1 balance: $%.2f\n",
      account1.getBalance() );
    System.out.printf( "account2 balance: $%.2f\n\n",
      account2.getBalance() );

    // create Scanner to obtain input from command window
    Scanner input = new Scanner( System.in );
    double depositAmount; // deposit amount read from user

    System.out.println( "Enter deposit amount for account1: " ); // prompt
    depositAmount = input.nextDouble(); // obtain user input
    System.out.printf( "\nadding %.2f to account1 balance\n\n",  depositAmount );
    account1.credit( depositAmount ); // add to account1 balance
```

%f is used to output values of type **float or double.**

.2 represents the number of decimal places (2) to output to the right of the decimal point

Ex: **%.2f**

```
    // display balances
    System.out.printf( "account1 balance: $%.2f\n",
      account1.getBalance() );
    System.out.printf( "account2 balance: $%.2f\n\n",
      account2.getBalance() );

    System.out.print( "Enter deposit amount for account2: " ); // prompt
    depositAmount = input.nextDouble(); // obtain user input
    System.out.printf( "\nadding %.2f to account2 balance\n\n",
      depositAmount );
    account2.credit( depositAmount ); // add to account2 balance

    // display balances
    System.out.printf( "account1 balance: $%.2f\n",
      account1.getBalance() );
    System.out.printf( "account2 balance: $%.2f\n",
      account2.getBalance() );
  } // end main
} // end class AccountTest



public class Account
{
  private double balance; // instance variable that stores the balance

  // constructor
  public Account( double initialBalance )
  {
    // validate that initialBalance is greater than 0.0;
    // if it is not, balance is initialized to the default value 0.0
    if ( initialBalance > 0.0 )
      balance = initialBalance;
  } // end Account constructor

  // credit (add) an amount to the account
  public void credit( double amount )
  {
    balance = balance + amount; // add amount to balance
  } // end method credit

  // return the account balance
  public double getBalance()
  {
    return balance; // gives the value of balance to the calling method
  } // end method getBalance
} // end class Account
```

Ch4.control statement 1

- **Three types of selection statements.**
  - **if statement:**

```
if ( studentGrade >= 60 )
   System.out.println( "Passed" );
```

  - **if…else** statement:

```
if ( grade >= 60 )
   System.out.println( "Passed" );
else
   System.out.println( "Failed" );
```

```
public class CH_EX6 {

  public static void main(String args[]){
    int x = 30;

    if( x == 10 ){
      System.out.print("Value of X is 10");
    }else if( x == 20 ){
      System.out.print("Value of X is 20");
    }else if( x == 30 ){
      System.out.println("Value of X is 30");
      System.out.println("Value of X is 30");
    }else{
      System.out.print("This is else statement");
    }
  }
}
```

```
run:
Value of X is 30
Value of X is 30
```

  - **switch statement**

```
import java.util.Scanner;
 public class ch4_ex7 {

  public static void main(String args[]){
{
   Scanner input= new Scanner(System.in);


int  grade;

   System.out.print("enter your grade (1,2,3,4,5):\n");
grade= input.nextInt();

  switch (grade)
  {
  case 1 :
    System.out.print("Excellent!" );
```

```
   case 2 :
   case 3 :
      System.out.print( "Well done" );
      break;
case 5 :
      System.out.print("You passed" );

   case 6 :
      System.out.print("Better try again" );
      break;

   default :
      System.out.print("Invalid grade");
   }
   System.out.printf("Your grade is %d\n", grade);
}}}
```

```
run:
enter your grade (1,2,3,4,5):
3
Well doneYour grade is 3


------
run:
enter your grade (1,2,3,4,5):
5
You passedBetter try againYour grade is
5
--------------------------
run:
enter your grade (1,2,3,4,5):
7
Invalid gradeYour grade is 7
```

- **5.Three repetition** statements (also called **looping statements)**

- **while and for** statements
  int product =3;
  while (product <= 100)
  product = 3 * product;

- **The do…while** ***statement performs the action(s) in its body one or more times.***
- **if, else, switch, while, do and for** are keywords

2. control statement part2
CH4_EX1.

```
public class CH4_EX1 {


  public static void main( String[] args )
  {
    int sum;
    int x;

    x = 1;
    sum = 0;

    while ( x <= 10 )
    {
      sum += x;
      ++x;
    }
    System.out.printf( "The sum is: %d\n", sum );
  }
}
```

run:
The sum is: 55

```
public class CH4_EX2
{
  public static void main( String[] args )
  {
    int y;
    int x = 1;
    int total = 0;

    while ( x <= 10 )
    {
      y = x * x;
      System.out.println( y );
      total += y;
      ++x;
    } // end while

    System.out.printf( "Total is %d\n", total );
  } // end main
} // end class Mystery
```

```
run:
1
4
9
16
25
36
49
64
81
100
Total is 385
```

Task1

```
public class CH4_EX3
{
  public static void main( String[] args )
  {
    int count = 1;

    while ( count <= 10 )
    {
      System.out.println( count % 2 == 1 ? "****" : "++++++++" );
      ++count;
    } // end while
  } // end main
} // end class Mystery2
```

```
run:
****
++++++++
****
++++++++
****
++++++++
****
++++++++
****
++++++++
```

```
public class CH4_EX4
{
  public static void main( String[] args )
  {
    int row = 10;
    int column;

    while ( row >= 1 )
    {
      column = 1;

      while ( column <= 10 )
      {
        System.out.print( row % 2 == 1 ? "<" : ">" );
        ++column;
      } // end while
      --row;
      System.out.println();
    } // end while
  } // end main
} // end class Mystery3
```

```
run:
>>>>>>>>>>
<<<<<<<<<<
>>>>>>>>>>
<<<<<<<<<<
>>>>>>>>>>
<<<<<<<<<<
>>>>>>>>>>
<<<<<<<<<<
>>>>>>>>>>
<<<<<<<<<<
```

---

ch4_ex6

```
import java.util.Scanner;
public class ch4_ex6 {


   public static void main(String[] args) {
System.out.print("enter grade or -1 to quit:");
Scanner input = new Scanner(System.in);
      int grade;
      grade= input.nextInt();
      double total =0.0;
      int gradecounter = 0;
      while(grade!=-1)
      {
         total= total + grade;
         gradecounter++;
         System.out.print("enter grade or -1 to quit:");
         grade= input.nextInt();

      }
      if(gradecounter!=0)
      {

       double average =(double)total/gradecounter;

      System.out.printf("total=%.2f\n",total);
      System.out.printf("average=%.2f\n",average);
       }
      else
         System.out.print("no grade was entered");
  }}
```

```
run:

enter grade or -1 to quit:30

enter grade or -1 to quit:70

enter grade or -1 to quit:80

enter grade or -1 to quit:-1

total=180.00

average=60.00
```

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

Fig. 4.13 | Arithmetic compound assignment operators.

Operator	Operator name	Sample expression	Explanation
++	prefix increment	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	postfix increment	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	prefix decrement	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	postfix decrement	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

Fig. 4.14 | Increment and decrement operators.

```
public static void main( String[] args )
  {
    int c;

    // demonstrate postfix increment operator
    c = 5; // assign 5 to c
    System.out.println( c );
    System.out.println( c++ );
    System.out.println( c );

    System.out.println(); // skip a line

    // demonstrate prefix increment operator
    c = 5; // assign 5 to c
    System.out.println( c );
    System.out.println( ++c );
    System.out.println( c

    c = 5; // assign 5 to c
    System.out.println( c );
    System.out.println( c-- );
    System.out.println( c );

    System.out.println(); // skip a line

    // demonstrate prefix increment operator
    c = 5; // assign 5 to c
    System.out.println( c );
    System.out.println( --c );
    System.out.println( c );
  }
```

```
run:
5
5
6

5
6
6

5
5
4

5
4
4
```

int x = 5, y = 8; **switch (i)** **{** **case 0:** **x+=(++x)+(x++);** **System.out.printf ("x =%d\n", x);** **break;**	x=x+(++x)+(x++); 1)++x (6) 2)x+x+x (6+6+6) = 18 3)x++ (18+1**=19)**
case 1: x+=++y; **System.out.printf ("x =%d\n", x);** break;	X=x+(++y) 1)++y =9 X+y = 5+9=**14**
case 2: x+=2*x++; **System.out.printf ("x =%d\n", x);** break;	x=x+2*x++; 1) x+2*x=5+2*5=15 2) X++= 15+1= **16**
case 3: x=--y+x--+x; **System.out.printf ("x =%d\n", x);** break;	x=x-(--y)+(x--)+x; 1) –y = 7 2) y+x+x=(7+5+5)=17 3)x-- = 17-1=**16**

case 4: **x-=(-y)%3;** **System.out.printf ("x =%d\n", x);** **break;**	**x=x- (-y)%3;** 1) **-8%3** 2) **5- -2=7**
case 5: y+=--y+x-y%x--; **System.out.printf ("y =%d\n", y);** break;	y=y+ (--y)+x-y%(x--); 1) **–y =7** 2) **Y+y+x-y%x = 7+7+5-7%5 = 17** 3) **X - - لا تنفذ لان جملة الطباعة لا تشملها**
case 6: x= ++y - --x + y++; **System.out.printf ("x =%d\n", x);** break;	x= ++y - --x + y++; 1) ++y 2) - - x 3) Y-x+y= 9-4+9 = 14 4) Y++ لا تنفذ لان جملة الطباعة لا تشملها

CH5-Control statement 2

For Statement

- Format:

```
for (Initialization; Test; Update counter)
{
    //the block of code which is executed multiple times
}
```

  - **Initialization** : Variable declaration and Initialization.
  - **Test** : A boolean expression.
  - **Update**: Update the Variable.

- **Example 1:**

```
public static void main( String[] args )
  {
    int total = 0;

    // total even integers from 2 through 20
 for ( int number = 2; number <= 20; number += 2 )
      total += number;
    System.out.printf( "Sum is %d\n", total );
  }
}
```

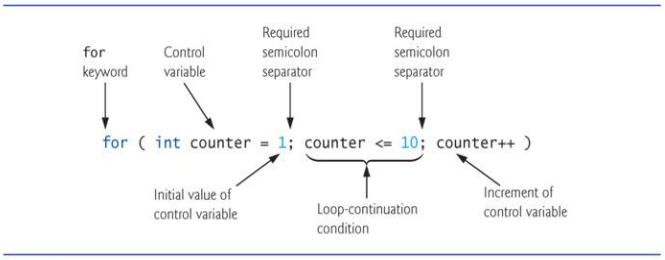


Fig. 5.3 | for statement header components.

Ch5-ex1 output

Sum is 110

- The increment expression in a for acts as if it were a standalone statement at the end of the for's body, so

```
counter = counter + 1
counter += 1
++counter
counter++
```

are **equivalent increment** expressions in a **for** statement.

Example:

- For example, assume that **x = 2** and **y = 10**. If x and y are not modified in the body of the loop, the statement

  for (int j = x; j <= 4 * x * y; j += y / x)

- is **equivalent** to the statement

  for (int j = 2; j <= 80; j += 5)

```
ch5_ex2 :     // total even integers from 2 through 20

public static void main( String[] args )

 {

  int total = 0;
   for (  int number = 2;
          number <= 20;
          total += number,
          System.out.printf( "number is %d Sum is %d\n",number,total ),
          number += 2);   }}
```

```
run:
number is 2 Sum is 2
number is 4 Sum is 6
number is 6 Sum is 12
number is 8 Sum is 20
number is 10 Sum is 30
number is 12 Sum is 42
number is 14 Sum is 56
number is 16 Sum is 72
number is 18 Sum is 90
number is 20 Sum is 110
```

Ch5_ex3 : nested for statement

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

     int num1, num2;
     for(num1=0;num1<=3;num1++)
     {
   for(num2=0;num2<=2;num2++)
   {
      System.out.printf("num1=%d num2=%d",num1,num2);
      System.out.println();
       }    }
  }
```

```
run:
num1=0 num2=0
num1=0 num2=1
num1=0 num2=2
num1=1 num2=0
num1=1 num2=1
num1=1 num2=2
num1=2 num2=0
num1=2 num2=1
num1=2 num2=2
num1=3 num2=0
num1=3 num2=1
num1=3 num2=2
```

Ch5_ex4: do-while statement

■ **Format**

```
 //condition variable is initiated
    do
 {
   Statement
//condition variable is updated
  } while ( Condition );
```

```
public static void main( String[] args )
  {
    int counter = 1; // initialize counter

    do
    {
      System.out.printf( "%d  ", counter );
      ++counter;
    } while ( counter <= 10 ); // end
do...while
 System.out.println(); // outputs a
newline
  }
```

```
run:

1 2 3 4 5 6 7 8 9 10
```

Ch5: break and continue Statements

Ch5:ex5 break statement

```
public static void main( String[] args )
  {
    int count;
    for ( count = 1; count <= 10; count++ )     {
      if ( count == 5 ) // if count is 5,
        break;
      System.out.printf( "%d ", count );
    }
    System.out.printf( "\n Broke out of loop at count = %d\n", count );
  }
```

```
run:

1 2 3 4

Broke out of loop at count = 5
```

- The **break** statement causes program control to proceed with the first statement after the switch. Or **terminates the loop**

CH5_EX6: **continue** statement

```
public static void main( String[] args )
  {
    int count;
    for ( count = 1; count <= 10; count++ )     {
      if ( count == 5 ) // if count is 5,
continue;       // terminate loop
      System.out.printf( "%d ", count );
    }
    System.out.printf( "\n Broke out of loop at count = %d\n", count );
  }
```

- The **continue** statement causes program control to **terminates the loop immediately and program control continues**

```
run:

1 2 3 4 6 7 8 9 10

Broke out of loop at count = 11
```

The Logical Operators are

- **&&** (conditional AND)
- **||** (conditional OR)
- **!** (logical NOT).

Example1:

```
if ( gender == 1 && age >= 65 )
  seniorFemales++;
```

Example2:

```
if ( 3<x && x<7 )
{
  System.out.print("you are welcome")
}
```

Local and non local variable

- **Local Variable :**Variables declared in the loop structure only exist within the block
- Variables declared outside the loop structure are **non-local variables**. The values of these variables are kept after finishing each iteration of the loop structure.

Ch5_ex6b: **Local and non local variable**

```
public static void main(String[] args) {
 int x=1; //non local variable
for (int counter = 1; counter<=4; counter++) // counter is local variable
 {
   System.out.printf("couter=%d x= %d\n",counter,x);

   } }}
```

```
run:

couter=1    x= 1

couter=2    x= 1

couter=3    x= 1

couter=4    x= 1
```

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


int x=1; //non local variable
for (int counter = 1; counter<=4; counter++) // counter local variable
 {
   System.out.printf("couter=%d    x= %d\n",counter,x);

   }
   System.out.printf("couter=%d    x= %d\n",counter,x);
   }}
```

Run

Error , because counter is undefined outside for statement

Ch5_ex7: nested for with break statement

```
public static void main(String[] args) {
   int i, j;
for(i=2; i<=7; i++)
{
for(j=2; j <= (i/j); j++)
if(!(i%j==1)) break;
 if(j > (i/j))
 System.out.printf("%d is prime\n",i);
} } }
```

```
run:

2 is prime

3 is prime

5 is prime

7 is prime
```

For-Cond(T/F)	I	J	COUT
F	2	2	**2 is prime**
F	3	2	**3 is prime**
T (break) !(0==1)	4	2	No print
T F (break) !(1==1) F	5	2	No print
F	5	3	**5 is prime**
F F	6	2	No print
T F !(1==1)	7	2	No print
T (break) !(1==1) T	7	3	**7 is prime**

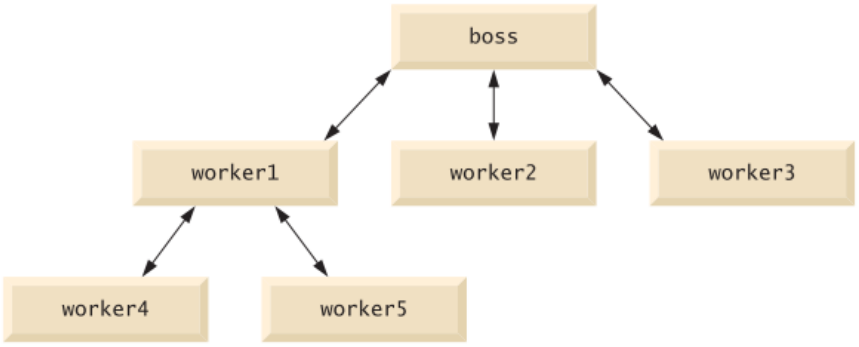


Fig. 6.1 | Hierarchical boss-method/worker-method relationship.

Method	Description	Example

`abs( x )`	absolute value of *x*	`abs( 23.7 )` is `23.7`
		`abs( 0.0 )` is `0.0`
		`abs( -23.7 )` is `23.7`

`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`exp( x )`	exponential method e^x	`exp( 1.0 )` is `2.71828` `exp( 2.0 )` is `7.38906`
`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`log( x )`	natural logarithm of *x* (base *e*)	`log( Math.E )` is `1.0` `log( Math.E * Math.E )` is `2.0`
`max( x, y )`	larger value of *x* and *y*	`max( 2.3, 12.7 )` is `12.7` `max( -2.3, -12.7 )` is `-2.3`
`min( x, y )`	smaller value of *x* and *y*	`min( 2.3, 12.7 )` is `2.3` `min( -2.3, -12.7 )` is `-12.7`

Fig. 6.2 | `Math` class methods. (Part 1 of 2.)

Method	Description	Example
`pow( x, y )`	*x* raised to the power *y* (i.e., x^y)	`pow( 2.0, 7.0 )` is `128.0` `pow( 9.0, 0.5 )` is `3.0`
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0.0`
`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is `30.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0.0`

Fig. 6.2 | `Math` class methods. (Part 2 of 2.)

6.3 static Methods, static Fields and Class Math (Cont.)

Ch6_ex1: static Methods or class method
Perform common tasks

```
public static void main( String[] args )
  {
    System.out.printf( "Math.abs( -23.7 ) = %f\n", Math.abs( -23.7 ) );
    System.out.printf( "Math.floor( 9.2 ) = %f\n", Math.floor( 9.2 ) );
    System.out.printf( "Math.floor( -9.8 ) = %f\n",Math.floor( -9.8 ) );
    System.out.printf( "Math.max( 2.3, 12.7 ) = %f\n",Math.max( 2.3, 12.7 ));
    System.out.printf( "Math.min( 2.3, 12.7 ) = %f\n", Math.min( 2.3, 12.7 ) );
    System.out.printf( "Math.pow( 2.0, 7.0 ) = %f\n", Math.pow( 2.0, 7.0 ) );
    System.out.printf( "Math.sqrt( 900.0 ) = %f\n", Math.sqrt( 900.0 ) );
    System.out.printf( "Math.PI = %f\n", Math.PI );
    System.out.printf( "Math.E = %f\n", Math.E );
  } // end main
} // end class
```

```
run:
Math.abs( -23.7 ) = 23.700000
Math.floor( 9.2 ) = 9.000000
Math.floor( -9.8 ) = -10.000000
Math.max( 2.3, 12.7 ) = 12.700000
Math.min( 2.3, 12.7 ) = 2.300000
Math.pow( 2.0, 7.0 ) = 128.000000
Math.sqrt( 900.0 ) = 30.000000
Math.PI = 3.141593
Math.E = 2.718282
```

- **Math fields** for common mathematical **constants**
  Math class constants **PI** and **E**
  - **Math.PI** = (3.141593)
  - **Math.E** = (2.718282)

PI and **E** are declared **final** because their values **never change**.

Why Is Method main Declared static ?
public **static** void main(String[] args)

Answer: When you execute the Java Virtual Machine (JVM) with the java command, the JVM Attempts to invoke the main method of the class you specify—when no objects of the classhave been created. Declaring main as static allows the JVM to invoke main without creating an instance of the class.

Type	Valid promotions
double	None
float	double
long	float or double
int	long, float or double
char	int, long, float or double
short	int, long, float or double (but not char)
byte	short, int, long, float or double (but not char)
boolean	None (boolean values are not considered to be numbers in Java)

System.out.println(Math.sqrt(4)); correctly evaluates **Math.sqrt(4)** and prints the value **2.0**. The method declaration's parameter list causes Java **to convert the int value 4 to the double value 4.0** before passing the value to method sqrt. Such conversions may lead to compilation errors if Java's promotion rules
are **not satisfied**

Ch6_ex2: // obtain three floating-point values and determine maximum value

```
import java.util.Scanner;

public class MaximumFinder
{
  public static void main( String[] args )
  {
    // create Scanner for input from command window
    Scanner input = new Scanner( System.in );

 System.out.print( "Enter three floating-point values separated by spaces: " );
    double number1 = input.nextDouble(); // read first double
    double number2 = input.nextDouble(); // read second double
    double number3 = input.nextDouble(); // read third double

   double result = maximum( number1, number2, number3 );
  System.out.println( "Maximum is: " + result );
  }

  public static double maximum( double x, double y, double z )
  {
    double maximumValue = x;
    if ( y > maximumValue )
      maximumValue = y;

    if ( z > maximumValue )
      maximumValue = z;

    return maximumValue;
  } // end method maximum
} // end class MaximumFinder
```

```
run:

Enter three floating-point values separated by spaces: 3.2  5.7  1.1

Maximum is: 5.7
```

Implementing Method maximum by Reusing Method Math.max

```
return Math.max( x, Math.max( y, z ) );
```

6.5 Notes on Declaring and Using Methods

- **Three ways to call a method:**
  - **1-** Using a method name by itself to call another method of the same class : like **ch6_ex2 page 28**
    maximum(number1, number2, number3)
  - **2-** Using a variable that contains a reference to an object, followed by a dot (.) and the method name to call a method of the referenced object such as **maximumFinder.determineMaximum()**
  - **3-** Using the class name and a dot (.) to call a static method of a class such as **Math.sqrt(900.0)**

Note: **A static method can call only other static methods of the same class directly** (i.e., using the method name by itself) and can manipulate only static variables in the same class directly.

Such as

```
return Math.max( x, Math.max( y, z ) );
```

There are **three ways to return control** to the statement that calls a method.

1- If the method does not return a result, use **return;**
2- is executed. If the method returns a result, use the statement **return expression;**
3- evaluates the expression, then returns the result to the caller. Such as
return 20; or return x;

6.8 Java API Packages

Package	Description
java.applet	The Java Applet Package contains a class and several interfaces required to create Java applets—**programs that execute in web browsers**.
java.awt	The Java Abstract Window Toolkit Package contains the classes and interfaces required to create and manipulate **GUIs**
java.io	The Java Input/Output Package contains classes and interfaces that enable programs to **input and output data.**
java.lang	The Java Language Package contains classes and interfaces, This package is imported by the **compiler into all programs**
java.net	The Java Networking Package contains classes and interfaces that enable programs to communicate via computer **networks like the Internet.**
java.util	The Java Utilities Package contains utility classes and interfaces that enable such actions as **date and time manipulations, random-number**

6.9 Case Study: Random-Number Generation

- Simulation and game playing
- element of chance
- Class **Random** (package **java.util**)
- static method **random of class Math.**
- Objects of class Random can produce random **boolean, byte, float, double, int, long and Gaussian** values
- Math method random can produce only double values in the range **0.0 <= x < 1.0.**

Ch6_ex3 : Shifted and scaled random integers.

```
import java.util.Random; // program uses class Random
public class ch6_ex3 {

   // Fig. 6.6: RandomIntegers.java
// Shifted and scaled random integers.
   public static void main( String[] args )
   {
      Random randomNumbers = new Random(); // random number generator
      int face; // stores each random integer generated

      // loop 20 times
      for ( int counter = 1; counter <= 20; counter++ )
      {
         // pick random integer from 1 to 6
         face = 1 + randomNumbers.nextInt( 6 );

         System.out.printf( "%d  ", face ); // display generated value

         // if counter is divisible by 5, start a new line of output
         if ( counter % 5 == 0 )
            System.out.println();
      } // end for
   } // end main
} // end class RandomIntegers
```

```
run:
4  3  2  2  1
6  1  2  5  1
6  3  4  2  5
2  5  6  1  5
```

number = shiftingValue + randomNumbers.nextInt(scalingFactor);
ex: face = 1 + randomNumbers.nextInt(6);
range from 1-6

6.11 Scope of Declarations

Basic scope rules:

- **The scope of a parameter declaration** is the body of the method in which the declaration appears.
- **The scope of a local-variable declaration** is from the point at which the declaration appears to the end of that block.
- **The scope of a local-variable declaration that appears in the initialization section of a for statement's** header is the body of the for statement and the other expressions in the header.
- A method or **field's scope** is the entire **body of the class**.
- Any block may contain variable declarations.
- **If a local variable or parameter in a method has the same name** as a field of the class, the field is "hidden" until the block terminates execution—this is called shadowing.

```
public class scope {

  // Scope class demonstrates field and local variable scopes.
  // field that is accessible to all methods of this class
  private static int x = 1;

  // method main creates and initializes local variable x
  // and calls methods useLocalVariable and useField
  public static void main( String[] args )
  {
    int x = 5; // method's local variable x shadows field x

    System.out.printf( "local x in main is %d\n", x );

    useLocalVariable(); // useLocalVariable has local x
    useField(); // useField uses class Scope's field x
    useLocalVariable(); // useLocalVariable reinitializes local x
    useField(); // class Scope's field x retains its value

    System.out.printf( "\nlocal x in main is %d\n", x );
  } // end main
  // create and initialize local variable x during each call
  public static void useLocalVariable()
  {
    int x = 25; // initialized each time useLocalVariable is called
    System.out.printf(
      "\nlocal x on entering method useLocalVariable is %d\n", x );
    ++x; // modifies this method's local variable x
    System.out.printf(
      "local x before exiting method useLocalVariable is %d\n", x );
  } // end method useLocalVariable
```

```
  // modify class Scope's field x during each call
public static void useField()
  {
    System.out.printf(
      "\nfield x on entering method useField is %d\n", x );
    x *= 10; // modifies class Scope's field x
    System.out.printf(
      "field x before exiting method useField is %d\n", x );
  } // end method useField
} // end class Scope
```

```
run:
local x in main is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in main is 5
```

```
public class MethodOverload
{
  // test overloaded square methods
  public static void main( String[] args )
  {
    System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
    System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
  }

  // square method with int argument
  public static int square( int intValue )
  {
    System.out.printf( "\nCalled square with int argument: %d\n",
      intValue );
    return intValue * intValue;
  } // end method square with int argument

  // square method with double argument
  public static double square( double doubleValue )
  {
    System.out.printf( "\nCalled square with double argument: %f\n",
      doubleValue );
    return doubleValue * doubleValue;
  } // end method square with double argument
} // end class MethodOverload
```

```
run:

Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.5
Square of double 7.5 is 56.25
```

Ch7-Arrays and Arraylists

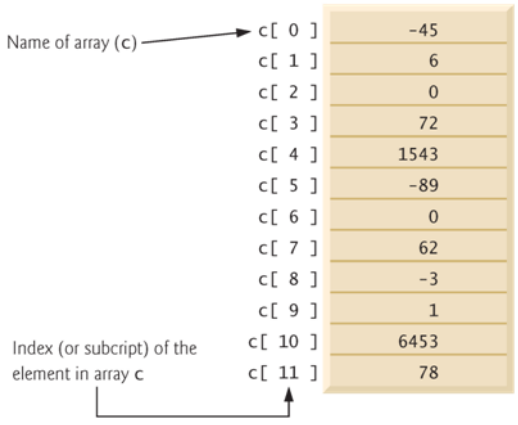
Arrays

- **Data structures** consisting of related data items of the **same type**.
- Make it convenient to process related groups of values.
- Remain the **same length** once they are created.

Common array manipulations with **static methods** of class Arrays from the **java.util** package.

- Array is **Group of variables** (called **elements**) containing values of the **same type.**
- Arrays are **objects** so they are **reference types.**
- **Elements** can be either **primitive** or **reference types.**

ArrayList collection



A 12-element array.

Note:

- An index must be a **nonnegative** integer.
- Every array object **knows its own length** and stores it in a **length** instance variable.
- length **cannot be changed** because it's **a final variable.**

7.3 Declaring and Creating Arrays

Declaration and array-creation expression for an array of 12 int elements

```
int[] c = new int[ 12 ];
```

Can be performed in two steps as follows:

```
int[] c; // declare the array variable
c = new int[ 12 ]; // creates the array
```

- When an array is created, each element of the array receives **a default value**
  - **Zero** for the **numeric primitive-type** elements, **false** for **boolean** elements and **null** for **references.**

Array initializer

A comma-separated list of expressions (called an **initializer list**) enclosed in **braces**.
int[] n = { 10, 20, 30, 40, 50 };
Creates a five-element array with index values **0–4**.

Ch7_ex1:

Fig. 7.2 uses keyword new to create an array of 10 int elements which are initially zero (the default for int variables)., included initialize 10 element of array .

```
public class ch7_ex1 {
  public static void main( String[] args )
  {
    int[] array; // declare array named array

    array = new int[ 10 ]; // create the array object

    // initializer list specifies the value for each element
    int[] array2 = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
    System.out.printf( "%s%8s\n", "Index", "Value" );
   // column headings

    // output each array element's value
    for ( int counter = 0; counter < array.length; counter++ )
      System.out.printf( "%5d%8d\n", counter, array[ counter ] );

    for ( int counter = 0; counter < array2.length; counter++ )
      System.out.printf( "%5d%8d\n", counter, array2[ counter ] );
  } // end main
} // end class InitArray
```

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
    5       0
    6       0
    7       0
    8       0
    9       0
    0      32
    1      27
    2      64
    3      18
    4      95
    5      14
    6      90
    7      70
    8      60
    9      37
```

Ch7_ex2:

creates a 10-element array and assigns to each element one of the even integers from 2 to 20 (2, 4, 6, …, 20).?

```
public class ch7_ex2 {

   public static void main( String[] args )
   {
      final int ARRAY_LENGTH = 10; // constant
      int[] array = new int[ ARRAY_LENGTH ]; // create array

      // calculate value for each array element
      for ( int counter = 0; counter < array.length; counter++ )
         array[ counter ] = 2 + 2 * counter;

      System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings

      // output each array element's value
      for ( int counter = 0; counter < array.length; counter++ )
         System.out.printf( "%5d%8d\n", counter, array[ counter ] );
   }
}
```

Index	Value
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20

Ch7_ex3:

```
// Computing the sum of the elements of an array.

public class SumArray
{
   public static void main( String[] args )
   {
      int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
      int total = 0;

      // add each element's value to total
      for ( int counter = 0; counter < array.length; counter++ )
         total += array[ counter ];

      System.out.printf( "Total of array elements: %d\n", total );
   }
}
```

Total of array elements: 849

// **Using enhanced for statement**
Iterates through the elements of an array **without using a counter**

// The enhanced for statement can be used only to obtain array elements It cannot be used to modify elements.

```
for ( int number : array )
      total += number;
```

Syntax:

```
for ( parameter : arrayName )
  statement;
```

Note in array:

Format specifier **%02d** indicates that an int value should be formatted as a field **of two digits.**

The **0** flag displays a leading **0** for values with fewer digits than the field width (**2**).

Ch7_ex4: // Bar chart printing program.

```
public class BarChart
{
  public static void main( String[] args )
  {
    int[] array = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };

    System.out.println( "Grade distribution:" );

    // for each array element, output a bar of the chart
    for ( int counter = 0; counter < array.length; counter++ )
    {
      // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
      if ( counter == 10 )
        System.out.printf( "%5d: ", 100 );
      else
        System.out.printf( "%02d-%02d: ",
          counter * 10, counter * 10 + 9  );

      // print bar of asterisks
      for ( int stars = 0; stars < array[ counter ]; stars++ )
        System.out.print( "*" );

      System.out.println(); // start a new line of output
    }
  }
}
```

```
Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

7.7 Passing Arrays to Methods

- To pass an array argument to a method, specify the name of the array without any brackets.
- To receive an array, the method's parameter list must specify an array parameter.
- When an argument to a method is an entire array or an individual array element of a reference type, the called method receives a copy of the **reference. (Pass-by-reference (also called call-by-reference) send full array**

- When an argument to a method is an individual array element of a **primitive type,** the called method receives a copy of the element's value. **Pass-by-value (also called call-by-value)** send some elements of array

```
public class PassArray
{
  // main creates array and calls modifyArray and modifyElement
  public static void main( String[] args )
  {
    int[] array = { 1, 2, 3, 4, 5 };

    System.out.println(
      "Effects of passing reference to entire array:\n" +
      "The values of the original array are:" );

    // output original array elements
    for ( int value : array )
      System.out.printf( "   %d", value );

    modifyArray( array ); // pass array reference
    System.out.println( "\n\nThe values of the modified array are:" );

    // output modified array elements
    for ( int value : array )
      System.out.printf( "   %d", value );

    System.out.printf(
      "\n\nEffects of passing array element value:\n" +
      "array[3] before modifyElement: %d\n", array[ 3 ] );

    modifyElement( array[ 3 ] ); // attempt to modify array[ 3 ]
    System.out.printf(
      "array[3] after modifyElement: %d\n", array[ 3 ] );
  }
```

```
  public static void modifyArray( int array2[] )
  {
    for ( int counter = 0; counter < array2.length; counter++ )
      array2[ counter ] *= 2;
  }

  public static void modifyElement( int element )
  {
    element *= 2;
  System.out.printf( "Value of element in modifyElement: %d\n", element );
  }
}
```

```
Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5

The values of the modified array are:
   2   4   6   8   10

Effects of passing array element value:
array[3] before modifyElement: 8
Value of element in modifyElement: 16
array[3] after modifyElement: 8
```

Fig. 7.13 | Passing arrays and individual array elements to methods. (Part 3 of 3.)

7.9 Multidimensional Arrays

- Java does not support multidimensional arrays directly
- Allows you to specify one-dimensional arrays whose elements are also one-dimensional arrays, thus achieving the same effect.
- In general, an array with m rows and n columns is called an **m-by-n array**.

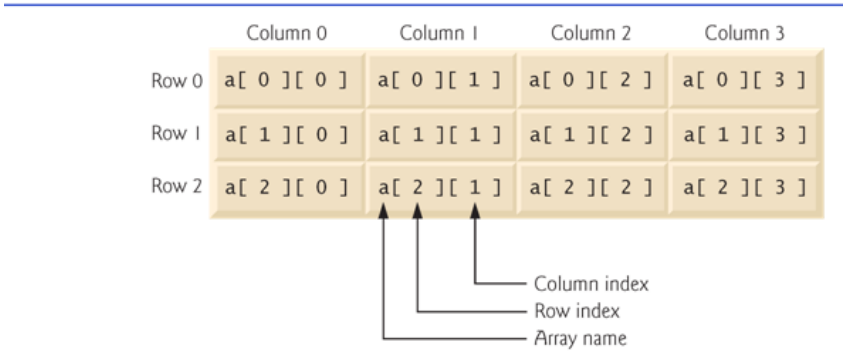


Fig. 7.16 | Two-dimensional array with three rows and four columns.

A two-dimensional array b with two rows and two columns could be declared and initialized with nested array initializers as follows:

```
int[][] b = { { 1, 2 },
              { 3, 4 } };
```

The lengths of the rows in a two-dimensional array are not required to be the same:

```
int[][] b = { { 1, 2 },
              { 3, 4, 5 } };
```

A multidimensional array with the same number of columns in every row can be created with an array-creation expression.

```
int[][] b = new int[ 3 ][ 4 ];
```

3 rows and 4 columns.

A multidimensional array in which each row has a different number of columns can be created as follows:

```
int[][] b = new int[ 2 ][ ];   // create 2 rows
b[ 0 ] = new int[ 5 ]; // create 5 columns for row 0
b[ 1 ] = new int[ 3 ]; // create 3 columns for row 1
```

Creates a two-dimensional array with two rows.
Row 0 has five columns, and row 1 has three columns.

Ch7_ex5: // create and output two-dimensional arrays

```
public class CH7_EX5 {

   public static void main( String[] args )
  {
   int[][] array1 = { { 1, 2, 3 }, { 4, 5, 6 } };
   int[][] array2 = { { 1, 2 }, { 3 }, { 4, 5, 6 } };

   System.out.println( "Values in array1 by row are" );
   outputArray( array1 ); // displays array1 by row

   System.out.println( "\nValues in array2 by row are" );
   outputArray( array2 ); // displays array2 by row
  }

  // output rows and columns of a two-dimensional array
  public static void outputArray( int[][] array )
  {
    // loop through array's rows
    for ( int row = 0; row < array.length; row++ )
    {
      // loop through columns of current row
      for ( int column = 0; column < array[ row ].length; column++ )
        System.out.printf( "%d  ", array[ row ][ column ] );

      System.out.println(); // start new line of output
    }   } }
```

```
Values in array1 by row are
1  2  3
4  5  6

Values in array2 by row are
1  2
3
4  5  6
```

CH7_EX6:TASK1: WE NEED A Function to find the sum of two dimensional arrays A and B

```
int [4][3] A={{12, 29, 11},
              {25, 25, 13},
              {24, 64, 67},
              {11, 18, 14}};

 int [4][3] B={{12, 29, 11},
              {25, 25, 13},
              {24, 64, 67},
              {11, 18, 14}};
```

CH8: Classes and Objects: A Deeper Look

8.2 Time Class Case Study

Ch8_ex1:

```
public class Time1Test
{
  public static void main( String[] args )
  {
    // create and initialize a Time1 object
    Time1 time = new Time1(); // invokes Time1 constructor

    // output string representations of the time
    System.out.print( "The initial universal time is: " );
    System.out.println( time.toUniversalString() );
    System.out.print( "The initial standard time is: " );
    System.out.println( time.toString() );
    System.out.println();

    // change time and output updated time
    time.setTime( 13, 27, 6 );
    System.out.print( "Universal time after setTime is: " );
    System.out.println( time.toUniversalString() );
    System.out.print( "Standard time after setTime is: " );
    System.out.println( time.toString() );
    System.out.println(); // output a blank line
    System.out.print( "Universal time: " );

    // attempt to set time with invalid values
    try
    {
      time.setTime( 99, 99, 99 ); // all values out of range
    }
     catch ( IllegalArgumentException e )
    {

   System.out.printf( "Exception: %s\n\n", e.getMessage() );     } //
end catch

       // display time after attempt to set invalid values

    System.out.println( "After attempting invalid settings:" );

    System.out.println( time.toUniversalString() );

    System.out.print( "Standard time: " );

    System.out.println( time.toString() );

  } // end main    } // end class Time1Test
```

Class Time1 does not declare a constructor, so the class has a default constructor that is supplied by the compiler.

Each instance variable implicitly receives the default value 0 for an int.

Access modifiers **public** and **private** control access to a class's variables and methods.

public methods present to the class's clients a view of the services the class provides (the class's public interface).

Clients need **not be concerned** with how the class accomplishes its tasks.

private class members are not accessible outside the class.

Method setTime and Throwing Exceptions (cont.)

For incorrect values, setTime throws an exception of type **IllegalArgumentException**

Notifies the client code that an invalid argument was passed to the method.

Can use **try...catch** to catch exceptions and attempt to recover from them.

The **throw** statement creates a new object of type **IllegalArgumentException**. In this case, we call the constructor that allows us to specify a custom error message.

After the exception object is created, the **throw** statement immediately terminates method setTime and the exception is returned to the code that attempted to set the time.

```
public class Time1
{
  private int hour; // 0 - 23
  private int minute; // 0 - 59
  private int second; // 0 - 59

    public void setTime( int h, int m, int s )
  {
       if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
     ( s >= 0 && s < 60 ) )
    {
      hour = h;
      minute = m;
      second = s;
    }
    else
          throw new IllegalArgumentException(
       "hour, minute and/or second was out of range" );
  }
  // convert to String in universal-time format (HH:MM:SS)
  public String toUniversalString()
  {
    return String.format( "%02d:%02d:%02d", hour, minute, second );
  }

  // convert to String in standard-time format (H:MM:SS AM or PM)
  public String toString()
  {
    return String.format( "%d:%02d:%02d %s",
      ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
      minute, second, ( hour < 12 ? "AM" : "PM" ) );
  } }
```

```
run:
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

Universal time: After attempting invalid settings:
13:27:06
Standard time: 1:27:06 PM
```

8.3 Controlling Access to Members

- Access modifiers **public** and **private** control access to a class's variables and methods.
- **public** methods present to the class's clients a view of the services the class provides (the class's public interface).
- Clients need not be concerned with how the class accomplishes its tasks.

For this reason, the class's **private** variables and **private** methods (i.e., its implementation details) are not accessible to its clients.

- **private class** members are not accessible outside the class.

8.4 Referring to the Current Object's Members with the this Reference
Ch8:ex2

```
public class ThisTest
{
   public static void main( String[] args )
   {
      SimpleTime time = new SimpleTime( 15, 30, 19 );
      System.out.println( time.buildString() );
   } // end main
} // end class ThisTest

// class SimpleTime demonstrates the "this" reference
class SimpleTime
{
  private int hour; // 0-23
   private int minute; // 0-59
   private int second; // 0-59
   // if the constructor uses parameter names identical to
   // instance variable names, the "this" reference is
   // required to distinguish between the names
   public SimpleTime( int hour, int minute, int second )
   {
      this.hour = hour; // set "this" object's hour
      this.minute = minute; // set "this" object's minute
      this.second = second; // set "this" object's second
   } // end SimpleTime constructor

   // use explicit and implicit "this" to call toUniversalString
   public String buildString()
   {
      return String.format( "%24s: %s\n%24s: %s",
         "this.toUniversalString()", this.toUniversalString(),
         "toUniversalString()",toUniversalString() );
   } // end method buildString

   // convert to String in universal-time format (HH:MM:SS)
   public String toUniversalString()
   {
      // "this" is not required here to access instance variables,
      // because method does not have local variables with same
  // names as instance variables
      return String.format( "%02d:%02d:%02d",
         hour, minute, second );
   } // end method toUniversalString
} // end class SimpleTime
```

Referring to the Current Object's Members with the this Reference

Every object can access a reference to itself with keyword **this**.

When a **non-static method** is called for a particular object, the method's body implicitly uses keyword **this** to refer to the object's **instance variables** and other methods.

Enables the class's code to know which object should be manipulated.

Can also use keyword **this** explicitly in a **non-static method's** body.

- Can use the **this** reference implicitly and explicitly.

- If a method contains a **local variable** with the same name as **a field**, that method uses the local variable rather than the field.
- The local variable ***shadows*** the field in the method's scope.
- A method can use the **this** reference to refer to the shadowed field explicitly.
- **this** explicitly in **a non-static method's** body
- **this** cannot be used in **a static method**

run:

this.toUniversalString(): 15:30:19

toUniversalString(): 15:30:19

8.5 Time Class Case Study: Overloaded Constructors

```
// Overloaded constructors used to initialize Time2 objects.

public class Time2Test
{
  public static void main( String[] args )
  {
    Time2 t1 = new Time2(); // 00:00:00
    Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
    Time2 t5 = new Time2( t4 ); // 12:25:42

    System.out.println( "Constructed with:" );
    System.out.println( "t1: all arguments defaulted" );
    System.out.printf( "   %s\n", t1.toUniversalString() );
    System.out.println( "t4: hour, minute and second specified" );
    System.out.printf( "   %s\n", t4.toUniversalString() );
     System.out.println( "t5: Time2 object t4 specified" );
    System.out.printf( "   %s\n", t5.toUniversalString() );

      } // end main
} // end class Time2Test
```

- **Overloaded constructors** enable objects of a class to be initialized in different ways.
- To overload constructors, simply provide **multiple constructor** declarations with **different signatures.**

Recall that the compiler **differentiates signatures** by the *number* of parameters, the *types* of the parameters and the *order* of the parameter types in each signature

Uses **this** in method-call syntax **to invoke the Time2 constructor**

Using the this reference as shown here is a popular way to **reuse initialization code** provided by another of the class's constructors rather than defining similar code in the no-argument constructor's body.

We use this syntax in four of the five Time2 constructors **to make the class easier to maintain and modify.**

this reference that is allowed only as the **first statement** in **constructor's body**

```
public class Time2
{
  private int hour; // 0 - 23
  private int minute; // 0 - 59
  private int second; // 0 - 59

  // Time2 no-argument constructor:
  // initializes each instance variable to zero
  public Time2()
  {
    this( 0, 0, 0 ); // invoke Time2 constructor with three
arguments
  } // end Time2 no-argument constructor


   // Time2 constructor: hour, minute and second supplied
  public Time2( int h, int m, int s )
  {
    setTime( h, m, s ); // invoke setTime to validate time
  } // end Time2 three-argument constructor

  // Time2 constructor: another Time2 object supplied
  public Time2( Time2 time )
  {
    // invoke Time2 three-argument constructor
    this( time.getHour(), time.getMinute(), time.getSecond() );
  } // end Time2 constructor with a Time2 object argument

    public void setTime( int h, int m, int s )
  {
    setHour( h );
    setMinute( m );
    setSecond( s );
  }

   public void setHour( int h )
  {
    hour=((h >= 0 && h<24)?h:0);
  }

    public void setMinute( int m )
  {
    minute=((m >= 0 && m<60)?m:0);
  }

   public void setSecond( int s )
  {
     second =((s >= 0 && s<60)?s:0);
  } // end method setSecond
```

```
  public int getHour()
  {
    return hour;
  }

public int getMinute()
  {
    return minute;
  }



  public int getSecond()
  {
    return second;
  }

  // convert to String in universal-time format
(HH:MM:SS)
  public String toUniversalString()
  {
    return String.format(
      "%02d:%02d:%02d", getHour(),
getMinute(), getSecond() );
  }
    }
```

```
run:
Constructed with:
t1: all arguments defaulted
  00:00:00
t4: hour, minute and second specified
  12:25:42
t5: Time2 object t4 specified
  12:25:42
```

8.6 Default and No-Argument Constructors

Every class must have **at least** one **constructor**. If you do not provide any in a class's declaration, the compiler creates a **default constructor** that takes **no arguments** when it's invoked.(**zero for primitive numeric types, false for boolean values and null for references**)

- If your class declares constructors, the compiler **will not** create a **default constructor.**

8.7 Notes on Set and Get Methods

- Classes often provide public methods to allow clients of the class **to *set*** (i.e., **assign values to)** or ***get* (i.e., obtain the values of) private instance variables**.
- ***Set* methods** are also commonly called **mutator methods**, because they typically change an object's state—i.e., **modify** the values of instance variables.

***Get* methods** are also commonly called **accessor methods** or **query methods**.

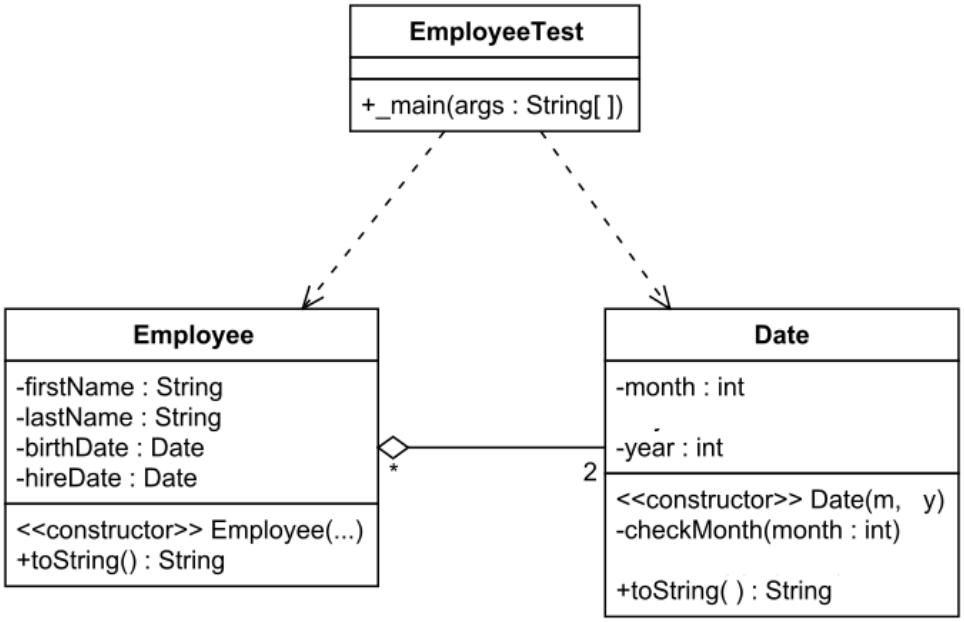
- Classes often provide public methods to allow clients of the class to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) private instance variables.
- *Set* methods are also commonly called mutator methods, because they typically change an object's state—i.e., modify the values of instance variables.

Get methods are also commonly called accessor methods or query methods.

8.8 Composition:

- A class can have references to objects of other classes as members.
- This is called **composition** and is sometimes referred to as a **has-a relationship.**
- Example: **An AlarmClock** object needs to know the current time and the time when it's supposed to sound its alarm, so it's reasonable to include two references to **Time** objects in an object.

The example used in this web page is **Employee** objects having references to **Date** objects. **Employee.java** has instance variables for the **employee's first name, last name, birth date, and date of hire:** two String objects and two Date objects. A **Date** object, as defined by **Date.java**, has instance variables for **month**, and **year**: all ints. **EmployeeTest.java** creates two **Date** objects, which are then used to create an **Employee** object.




```
public class EmployeeTest
{
  public static void main( String[] args )
  {
    Date birth = new Date( 7, 1949 );
    Date hire = new Date( 3,  1988 );
    Employee employee = new Employee( "Bob", "Blue", birth, hire );
    System.out.println( employee ); // implicitly invokes the Employee's
toString method to display the values of its instance variables
  } }
```

```
public class Date
{
  private int month; // 1-12
  private int year; // any year

    public Date( int theMonth,int theYear )
  {
    month = checkMonth( theMonth ); // validate month
    year = theYear; // could validate year

    System.out.printf(
      "Date object constructor for date %s\n", this );
  } // end Date constructor

  // utility method to confirm proper month value
  private int checkMonth( int testMonth )
  {
    if ( testMonth > 0 && testMonth <= 12 ) // validate month
      return testMonth;
    else // month is invalid
      throw new IllegalArgumentException( "month must be 1-12" );
  } // end method checkMonth

  // return a String of the form month/day/year
  public String toString()
  {
    return String.format( "%d/%d", month, year );
  } // end method toString
} // end class Date
```

Constructor **Date** output the **this** reference as **a String**. Since **this** is a reference to the current Date object, the object's **to-String method**

public String toString()

is called *implicitly* to obtain the object's String representation

```
public class Employee
{
  private String firstName;
  private String lastName;
  private Date birthDate;
  private Date hireDate;

  // constructor to initialize name, birth date and hire date
  public Employee( String first, String last, Date dateOfBirth,
    Date dateOfHire )
  {
    firstName = first;
    lastName = last;
    birthDate = dateOfBirth;
    hireDate = dateOfHire;
  } // end Employee constructor

  // convert Employee to String format
```

Class Employee

Members **birthDate** and **hireDate** are references to **Date** *objects*. This demonstrates that **a class can have as instance variables references to objects of other classes.**

```
run:

Date object constructor for date 7/1949

Date object constructor for date 3/1988

Blue, Bob  Hired: 3/1988  Birthday: 7/1949
```

8.11 static Class Members

- In certain cases, only one copy of a particular variable should be **shared by all objects of a class.**
  - **A static field**—*called a class variable*—is used in such cases.
- **A static variable** represents class wide information—all objects of the class share the same piece of data.
  - The declaration of a static variable begins with the keyword **static**.
- **Static** variables have **class scope.**
- Can access a class's **public static** members through a reference to any object of the class, or by qualifying the member name with the class name and **a dot (.),** as in **Math.random().**
- **private static** class members can be accessed by client code only through methods of the class.
- **static** class members are available as soon as the class is **loaded** into memory at execution time.
- To access a **public static** member when no objects of the class exist (and even when they do), prefix the class name and a dot (.) to the static member, as in **Math.PI.**
- A **static** method cannot access **non-static** class members, because a static method can be called even when **no objects** of the class have been **instantiated**.
- For the same reason, the **this** reference **cannot** be used in a **static method.**
- The **this** reference must refer to a **specific object of the class**, and when a static method is called, **there might not be any objects of its class in memory**.
- If a **static** variable is not initialized, the compiler assigns it a **default value**—in this **case 0**, the **default value for type int**.
- The following syntax imports a particular **static member**:
- **import static *packageName.ClassName.staticMemberName;***

ex:

```
import static java.lang.System.out;

import static java.lang.Math.PI;
import static java.lang.Math.cos;
```

- **The following syntax imports all static members of a class:**
- **import static *packageName.ClassName.*;***
- ex: **import static java.lang.Math.*;**
- **String** objects in Java are immutable—they **cannot be modified** after they are created.
- Therefore, it's safe to have many references to one String object.
- If String objects are immutable, you might wonder why are we able to use operators **+** and **+= to concatenate String objects**
- Keyword **final** specifies that a variable is **not modifiable** (i.e., it's a **constant**) and any attempt to modify it is **an error**.
- **private final int INCREMENT;**
- Declares a final (constant) instance variable INCREMENT of type int.

ch8_ex5

```
import java.util.Scanner; // program uses Scanner
public class EmployeeTest
{
  public static void main( String[] args )
  {
        Scanner input = new Scanner( System.in );
    // show that count is 0 before creating Employees
    System.out.printf( "Employees before instantiation: %d\n",
      Employee.getCount() );

    // create two Employees; count should be 2
    Employee e1 = new Employee( "Susan", "Baker" );
    Employee e2 = new Employee( "Bob", "Blue" );

    // show that count is 2 after creating two Employees
    System.out.println( "\nEmployees after instantiation: " );
    System.out.printf( "via e1.getCount(): %d\n", e1.getCount() );
    System.out.printf( "via e2.getCount(): %d\n", e2.getCount() );
    System.out.printf( "via Employee.getCount(): %d\n",
      Employee.getCount() );

    // get names of Employees
    System.out.printf( "\nEmployee 1: %s %s\nEmployee 2: %s %s\n",
      e1.getFirstName(), e1.getLastName(),
      e2.getFirstName(), e2.getLastName() );

System.out.println( "Enter 1 to clear creen " ); // prompt
    int cls; // deposit amount read from user
    cls = input.nextInt(); // obtain user input
if (cls==1)
   // for (int i = 0; i < 50; ++i) System.out.println();// for clean screen
   System.out.print("\u001b[2J");
System.out.flush(); //clear sceen

  } // end main
} // end class EmployeeTest
```

```
// Fig. 8.12: Employee.java
// Static variable used to maintain a count of the number of
// Employee objects in memory.

public class Employee
{
   final private String firstName;
   final  private String lastName;
  private static int count = 0; // number of Employees created

  // initialize Employee, add 1 to static count and
  // output String indicating that constructor was called
  public Employee( String first, String last )
  {
    firstName = first;
    lastName = last;

    ++count;  // increment static count of employees
    System.out.printf( "Employee constructor: %s %s; count = %d\n",
      firstName, lastName, count );
  } // end Employee constructor

  // get first name
  public String getFirstName()
  {
    return firstName;
  } // end method getFirstName

  // get last name
  public String getLastName()
  {
    return lastName;
  } // end method getLastName

  // static method to get static count value
  public static int getCount()
  {
    return count;
  } // end method getCount
} // end class Employee
```

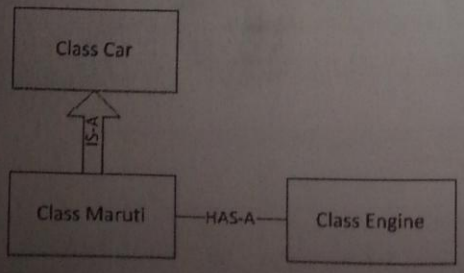
```
run:
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2

Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Susan Baker
Employee 2: Bob Blue
Enter 1 to clear screen
1
```

ch9: Inheritance

- **Inheritance**
  - A form of software reuse in which a new class is created by absorbing an existing class's members and enhance them with new or modified capabilities.
- inherit the members of an existing class.
- Existing class is the **superclass**
- New class is the **subclass**
- Each **subclass** can be a **superclass** of future subclasses.
- **A subclass** can **add** its own **fields** and **methods**.
- **A subclass** is more specific than its **superclass** and represents a more specialized group of objects.
- This is why **inheritance** is sometimes referred to as **specialization.**
- The **direct superclass** is the superclass from which the **subclass explicitly inherits**.
- An **indirect superclass** is any class **above** the **direct superclass** in the class hierarchy.
- The **Java class hierarchy begins** with class Object (in package **java.lang**)
- ***Every* class** in Java **directly or indirectly** extends (or "inherits **from") Object**.
- Java supports only **single inheritance**, in which each class is **derived from exactly one direct superclass.**

- **every subclass** object *is an* object of its **superclass**, and one **superclass** can have **many subclasses**, the **set of objects represented** by a **superclass** is typically **larger than the set of objects** represented by any of its **subclasses.**

- We **distinguish** between the **is-a** relationship and the **has-a** relationship
- ***Is-a*** represents **inheritance**
  - In an *is-a* relationship, an object of a subclass can also be treated as an object of its superclass
- ***Has-a*** represents **composition**



Maruti Suzuki Cars

an Employee ***has a*** BirthDate, and an Employee ***has a*** TelephoneNumber.

55

Superclass	Subclasses
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount

Fig. 9.1 | Inheritance examples.

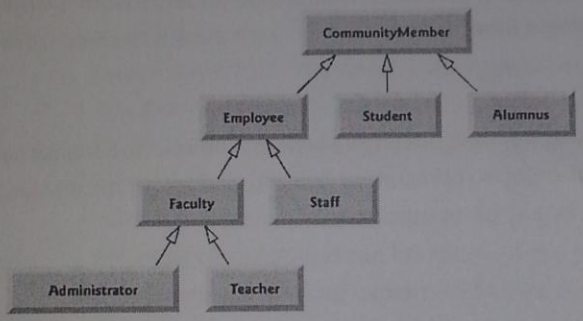


Fig. 9.2 | Inheritance hierarchy for university CommunityMembers.

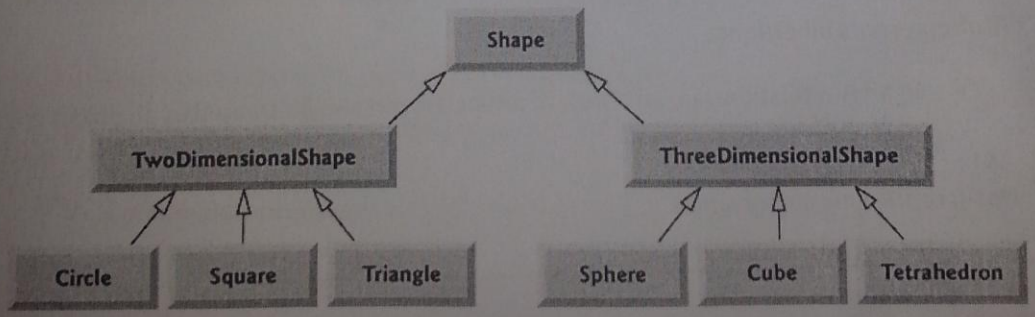
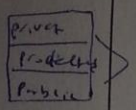


Fig. 9.3 | Inheritance hierarchy for Shapes.

- ASphere *is a* ***ThreeDimensionalShape*** and *is a* ***Shape***.

- **Inheritance issue**
- **A subclass** can **inherit methods** that it **does not need** or **should not have**.
- **subclass** often needs a **customized** version of the **method**
- The **subclass** can override **(redefine)** the **superclass method** with an appropriate implementation

9.3 protected Members



- **A class's private members** are accessible only within the **class itself.**
- **protected access** is an **intermediate level** of access **between public and private.**
  - **A superclass's protected members** can be **accessed by** members of that superclass, by members of its **subclasses** and by members of other **classes in the same package**
    - **protected members** also have **package access**.
    - **A superclass's private members** are **hidden** in its **subclasses**
    - They can be **accessed only through** the **public or protected** methods inherited from the **superclass**
    - When **a subclass** method overrides an **inherited superclass method**, the superclass method can be accessed from the subclass by **preceding the superclass method** name with keyword **super** and a **dot (.)** separator.

ch9_ex1: **9.4 Relationship between Superclasses and Subclasses**

- **Inheritance hierarchy** containing types of **employees** in a company's payroll application
- **Commission employees** are paid a **percentage** of their **sales**
- **Base-salaried commission** employees receive **a base salary** plus **a percentage of their sales.**

9.4.1 Creating and Using a CommissionEmployee Class

- Class **CommissionEmployee** extends(inherts from) class **Object** (from package **java.lang**).
  - CommissionEmployee inherits Object's methods.
  - every class in java (except **Object**) extends an existing class
  - **Object** class is the **root of the java class hierarchy**

If you don't explicitly specify which class a **new class extends**, the class extends **Object** implicitly

- **Constructors** are **not inherited.**
- The **first task** of a **subclass constructor** is to call its **direct superclass's constructor explicitly or implicitly**
-
- **toString** is one of the methods that every class inherits directly or indirectly from class Object.
  - Returns a String representing an object.
  - Called implicitly whenever an object must be converted to a String representation.

- **To override a superclass** method, a subclass must declare a method with the **same signature as the superclass method**
- **@Override** annotation
  - Indicates that a method should override a superclass method with the same signature.
  - If it does not, a compilation error occurs.

ch9-inheretance

ch9-ex1

```
package javaapplication38;

public class car {
  private int width=20;
  public int y;
  public int getdata()
  {
     return width;
  }
  public void setdata (int w)
  {
     width=w;
  }
  }
```

```
package javaapplication38;


public class bus extends car
{

}
```

```
package javaapplication38;

public class firstjava {

      public static void main(String[] args)
      {
         bus call=new bus();
         call.setdata(10);
         System.out.println( call.getdata());
         call.y=10;
          System.out.println( call.y);
      } }
```

```
run:
10
10
```

ch9_ex2

```
package javaapplication39;

public class food {
  public void eat()
  {
     System.out.println("i love the food");
  } }
```

```
package javaapplication39;

public class falafel extends food {

}
```

```
package javaapplication39;

public class shawerma extends food {

}
```

```
package javaapplication39;

public class hello {
 public static void main(String[] args) {
      falafel falobject= new falafel();
      shawerma shawobject= new shawerma();
      falobject.eat();
      shawobject.eat();
   }}
```

```
run:
i love the food
i love the food
```

ch9_ex3

```
package javaapplication44;

public class mainlist {
 public static void main(String args[]) {
                    MatchBox mb1 = new MatchBox(10, 10, 10, 10);
                    mb1.getVolume();
                    System.out.println("width of MatchBox 1 is " + mb1.width);
                    System.out.println("height of MatchBox 1 is " +
mb1.height);
                    System.out.println("depth of MatchBox 1 is " + mb1.depth);
                    System.out.println("weight of MatchBox 1 is " +
mb1.weight);
          }
}
```

```
package javaapplication44;

public class MatchBox extends Box {

          double weight;

          public MatchBox(double w, double h, double d, double m) {
                    super(w, h, d);
                    weight = m;
          }
}
```

```
package javaapplication44;

class Box {
          double width;
          double height;
          double depth;
          public Box(double w, double h, double d) {
                    width = w;
                    height = h;
                    depth = d;
          }

          void getVolume() {
          System.out.println("Volume is : " + width * height * depth);
          }}
```

Volume is : 1000.0
width of MatchBox 1 is 10.0
height of MatchBox 1 is 10.0
depth of MatchBox 1 is 10.0
weight of MatchBox 1 is 10.0

ch9_ex4

```
package person;

public class Person {
private  String firstname;
private String lastname;
    public Person(String firstname,String lastname)
    {
       this.firstname=firstname;
       this.lastname=lastname;
    }

    public void print()
    {
       System.out.println("\t" + firstname + "  " + lastname);
    }
}
```

```
package person;


public class professor extends Person{

   public professor(String firstname,String lastname)
   {
      super(firstname,lastname);
   }

   @Override
  public void print(){
      System.out.println("professor details");
      super.print();
   }
   }
```

```
package person;

public class student extends Person{
  private int graduationyear;
 private double gpa;
 public student (String firstname,String lastname ,int graduationyear,double gpa)
 {
    super (firstname,lastname);
 this.graduationyear=graduationyear;
 this.gpa=gpa;
}
public void print(){
System.out.println("student Details");
super.print();
System.out.println("\t" + graduationyear);
System.out.println("\t" + gpa);

}}
```

```
package person;

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


    professor knowitall= new professor ("peter","noetal");
    knowitall.print();

    student sam = new student("sam", "walton", 2012 ,3.8);
    sam.print();

  }
}
```

```
run:

professor details
        peter  noetal
student Details
        sam  walton
        2012
        3.8
```

```
Modifier    | Class | Package | Subclass | World
————————————+———————+—————————+——————————+———————
public      |  y    |    y    |    y     |   y
————————————+———————+—————————+——————————+———————
protected   |  y    |    y    |    y     |   n
————————————+———————+—————————+——————————+———————
no modifier |  y    |    y    |    n     |   n
————————————+———————+—————————+——————————+———————
private     |  y    |    n    |    n     |   n


y: accessible
n: not accessible
```

Private

Like you'd think, only the **class** in which it is declared can see it.

Package Private

Can only be seen and used by the **package** in which it was declared. This is the default in Java (which some see as a mistake).

Protected

Package Private + can be seen by subclasses or package member.

Public

Everyone can see it.

- **`Default`** are accessible by the **classes of the same package.**

In object-oriented terms, **overriding** means to override the functionality of an existing method.

The benefit of overriding is: ability to define a behavior that's specific to the subclass type which means a subclass can implement a parent class method based on its requirement

`instanceof` keyword is a *binary operator* used to test if an *object* (instance) is a subtype of a given Type.

ch9_ex5 - Quiz : 1

```
package testdog;

class Animal{

  public void move(){
    System.out.println("Animals can move");
  }
}
```

```
package testdog;

class Dog extends Animal{

  public void move(){
    System.out.println("Dogs can walk and run");
  }
}
```

[Type a quote from the document or the summary of an interesting point. You can position the text box anywhere in the document. Use the Text Box Tools tab to change the formatting of the pull quote text box.]

```
package testdog;

public class TestDog{

  public static void main(String args[]){
    Animal a = new Animal(); // Animal reference and object
    Animal b = new Dog(); // Animal reference but Dog object

    a.move();// runs the method in Animal class

    b.move();//Runs the method in Dog class
  }
}
```

```
run:

Animals can move

Dogs can walk and run
```

ch9_ex5 - Quiz2

```
class Animal{

   public void move(){
      System.out.println("Animals can move");
   }
}
```

```
class Dog extends Animal{

   public void move(){
      super.move(); // invokes the super class method
      System.out.println("Dogs can walk and run");
   }
```

```
public class TestDog{

   public static void main(String args[]){

      Animal b = new Dog();// Animal reference but Dog object
      b.move(); //Runs the method in Dog class

   }
}
```

```
Animals can move
Dogs can walk and run
```