Programming Languages: Java

Lecture 3

Methods

Arrays

Exception Handling

Instructor: Omer Boyaci

Methods

We will cover:

- How `static` methods and fields are associated with an entire class rather than specific instances of the class.
- To use common `Math` methods available in the Java API.
- To understand the mechanisms for passing information between methods.
- How the method call/return mechanism is supported by the method call stack and activation records.
- How packages group related classes.
- How to use random-number generation to implement game-playing applications.
- How the visibility of declarations is limited to specific regions of programs.
- What method overloading is and how to create overloaded methods.

Introduction

- Divide and conquer technique
  - Construct a large program from smaller pieces (or modules)
  - Can be accomplished using methods
- `static` methods can be called without the need for an object of the class
- Random number generation
- Constants

Program Modules in Java

- **Java Application Programming Interface (API)**
  - **Also known as the Java Class Library**
  - **Contains predefined methods and classes**
    - **Related classes are organized into packages**
    - **Includes methods for**
      - **Mathematics**
      - **string/character manipulations**
      - **input/output**
      - **databases**
      - **Networking**
      - **file processing**
      - **error checking**
      - **and more**

Program Modules in Java (Cont.)

- Methods
  - Called functions or procedures in some other languages
  - Modularize programs by separating its tasks into self-contained units
  - Enable a divide-and-conquer approach
  - Are reusable in later programs
  - Prevent repeating code

`static` Methods, `static` Fields and Class Math

- `static` method (or class method)
  - Applies to the class as a whole instead of a specific object of the class
  - Call a `static` method by using the method call: *ClassName* **.** *methodName* **(** *arguments* **)**
  - All methods of the `Math` class are `static`
    - example: `Math.sqrt( 900.0 )`

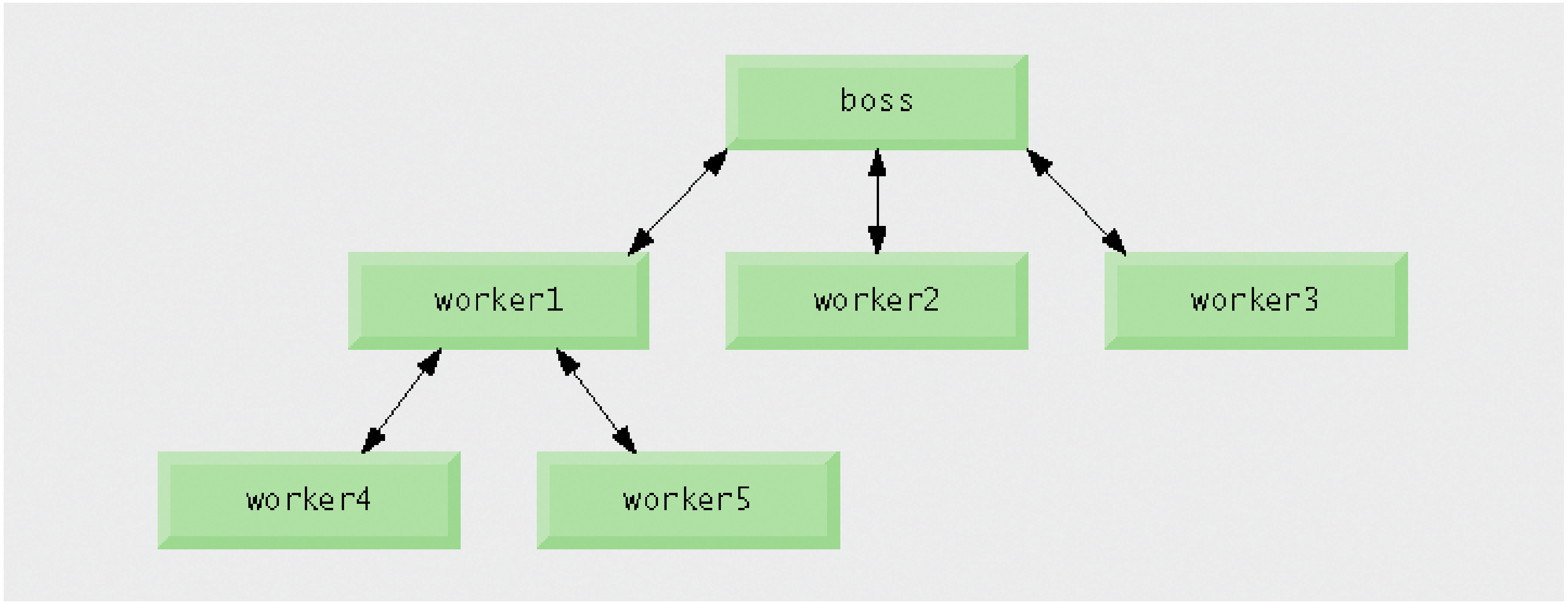

Fig. 6.1 | **Hierarchical boss-method/worker-method relationship.**

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

- **Constants**
  - **Keyword `final`**
  - **Cannot be changed after initialization**
- **`static` fields (or class variables)**
  - **Are fields where one copy of the variable is shared among all objects of the class**
- **`Math.PI` and `Math.E` are `final static` fields of the `Math` class**

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 *ex*	`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`

`pow( x, y )`	*x* raised to the power *y* (i.e., *xy*)	`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.

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

- Method `main`
  - `main` is declared `static` so it can be invoked without creating an object of the class containing `main`
  - Any class can contain a `main` method
    - The JVM invokes the `main` method belonging to the class specified by the first command-line argument to the `java` command

Declaring Methods with Multiple Parameters

- **Multiple parameters can be declared by specifying a comma-separated list.**
  - **Arguments passed in a method call must be consistent with the number, types and order of the parameters**
    - **Sometimes called formal parameters**

```
// Fig. 6.3: MaximumFinder.java
// Programmer-declared method maximum.
import java.util.Scanner;

public class MaximumFinder
{
   // obtain three floating-point values and locate the maximum value
   public void determineMaximum()
   {
      // create Scanner for input from command window
      Scanner input = new Scanner( System.in );

      // obtain user input
      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

      // determine the maximum value
      double result = maximum( number1, number2, number3 );

      // display maximum value
      System.out.println( "Maximum is: " + result );
   } // end method determineMaximum

```

Call method **`maximum`**

Display maximum value

`MaximumFinder.java`

(1 of 2)

```
   // returns the maximum of its three double parameters
   public double maximum( double x, double y, double z )
   {
      double maximumValue = x; // assume x is the largest to start

      // determine whether y is greater than maximumValue
      if ( y > maximumValue )
         maximumValue = y;

      // determine whether z is greater than maximumValue
      if ( z > maximumValue )
         maximumValue = z;

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

Declare the **maximum** method

MaximumFinder.java

2)

Compare **y** and **maximumValue**

Compare **z** and **maximumValue**

Return the maximum value

```
// Fig. 6.4: MaximumFinderTest.java
// Application to test class MaximumFinder.

public class MaximumFinderTest
{
   // application starting point
   public static void main( String args[] )
   {
      MaximumFinder maximumFinder = new MaximumFinder();
      maximumFinder.determineMaximum();
   } // end main
} // end class MaximumFinderTest
```

Create a **MaximumFinder** object

Call the **determineMaximum** method

MaximumFinderTest .java

```
Enter three floating-point values separated by spaces: 9.35 2.74 5.1
Maximum is: 9.35
```

```
Enter three floating-point values separated by spaces: 5.8 12.45 8.32
Maximum is: 12.45
```

```
Enter three floating-point values separated by spaces: 6.46 4.12 10.54
Maximum is: 10.54
```

Common Programming Error 6.1

Declaring method parameters of the same type as

```
float x, y
```

instead of

```
float x, float y
```

is a syntax error-a type is required for each parameter in the parameter list.

Declaring Methods with Multiple Parameters (Cont.)

- Reusing method `Math.max`
  - The expression `Math.max( x, Math.max( y, z ) )` determines the maximum of `y` and `z`, and then determines the maximum of `x` and that value
- String concatenation
  - Using the + operator with two `String`s concatenates them into a new `String`
  - Using the + operator with a `String` and a value of another data type concatenates the `String` with a `String` representation of the other value
    - When the other value is an object, its `toString` method is called to generate its `String` representation

Notes on Declaring and Using Methods

- **Three ways to call a method:**
  - **Use a method name by itself to call another method of the same class**
  - **Use a variable containing a reference to an object, followed by a dot (`.`) and the method name to call a method of the referenced object**
  - **Use the class name and a dot (`.`) to call a `static` method of a class**
- **`static` methods cannot call non-`static` methods of the same class directly**

Notes on Declaring and Using Methods (Cont.)

- **Three ways to return control to the calling statement:**
  - **If method does not return a result:**
    - **Program flow reaches the method-ending right brace or**
    - **Program executes the statement `return;`**
  - **If method does return a result:**
    - **Program executes the statement `return` *expression* `;`**
      - ***expression* is first evaluated and then its value is returned to the caller**

Common Programming Error

Declaring a method

outside the body of a class declaration

or

inside the body of another method

is a syntax error.

Method Call Stack and Activation Records

- Stacks
  - Last-in, first-out (LIFO) data structures
    - Items are pushed (inserted) onto the top
    - Items are popped (removed) from the top
- Program execution stack
  - Also known as the method call stack
  - Return addresses of calling methods are pushed onto this stack when they call other methods and popped off when control returns to them

Method Call Stack and Activation Records (Cont.)

- A method's local variables are stored in a portion of this stack known as the method's activation record or stack frame
  - When the last variable referencing a certain object is popped off this stack, that object is no longer accessible by the program
    - Will eventually be deleted from memory during "garbage collection"
  - Stack overflow occurs when the stack cannot allocate enough space for a method's activation record

Argument Promotion and Casting

- **Argument promotion**
  - **Java will promote a method call argument to match its corresponding method parameter according to the promotion rules**
  - **Values in an expression are promoted to the “highest” type in the expression (a temporary copy of the value is made)**
  - **Converting values to lower types results in a compilation error, unless the programmer explicitly forces the conversion to occur**
    - **Place the desired data type in parentheses before the value**
      - **example: `( int ) 4.5`**

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)

Fig. 6.5 | **Promotions allowed for primitive types.**

Java API Packages

- **Including the declaration**
  `import java.util.Scanner;`
  allows the programmer to use `Scanner` instead of `java.util.Scanner`
- **Java API documentation**
  - **java.sun.com/javase/6/docs/api/**
- **Overview of packages in Java SE 6**
  - **java.sun.com/javase/6/docs/api/overview-summary.html**

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. (Applets are discussed in Chapter 20, Introduction to Java Applets; interfaces are discussed in Chapter 10, Object_-Oriented Programming: Polymorphism.)
`java.awt`	The Java Abstract Window Toolkit Package contains the classes and interfaces required to create and manipulate GUIs in Java 1.0 and 1.1. In current versions of Java, the Swing GUI components of the `javax.swing` packages are often used instead. (Some elements of the `java.awt` package are discussed in Chapter 11, GUI Components: Part 1, Chapter 12, Graphics and Java2D, and Chapter 22, GUI Components: Part 2.)
`java.awt.event`	The Java Abstract Window Toolkit Event Package contains classes and interfaces that enable event handling for GUI components in both the `java.awt` and `javax.swing` packages. (You will learn more about this package in Chapter 11, GUI Components: Part 1 and Chapter 22, GUI Components: Part 2.)
`java.io`	The Java Input/Output Package contains classes and interfaces that enable programs to input and output data. (You will learn more about this package in Chapter 14, Files and Streams.)
`java.lang`	The Java Language Package contains classes and interfaces (discussed throughout this text) that are required by many Java programs. This package is imported by the compiler into all programs, so the programmer does not need to do so.

Fig. 6.6 | Java API packages (a subset). (Part 1 of 2)

Package	Description
`java.net`	The Java Networking Package contains classes and interfaces that enable programs to communicate via computer networks like the Internet. (You will learn more about this in Chapter 24, Networking.)
`java.text`	The Java Text Package contains classes and interfaces that enable programs to manipulate numbers, dates, characters and strings. The package provides internationalization capabilities that enable a program to be customized to a specific locale (e.g., a program may display strings in different languages, based on the user's country).
`java.util`	The Java Utilities Package contains utility classes and interfaces that enable such actions as date and time manipulations, random-number processing (class `Random`), the storing and processing of large amounts of data and the breaking of strings into smaller pieces called tokens (class `StringTokenizer`). (You will learn more about the features of this package in Chapter 19, Collections.)
`javax.swing`	The Java Swing GUI Components Package contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs. (You will learn more about this package in Chapter 11, GUI Components: Part 1 and Chapter 22, GUI Components: Part 2.)
`javax.swing.event`	The Java Swing Event Package contains classes and interfaces that enable event handling (e.g., responding to button clicks) for GUI components in package `javax.swing`. (You will learn more about this package in Chapter 11, GUI Components: Part 1 and Chapter 22, GUI Components: Part 2.)

Fig. 6.6 | Java API packages (a subset). (Part 2 of 2)

Case Study: Random-Number Generation

- Random-number generation
  - `static` method `random` from class `Math`
    - Returns `doubles` in the range `0.0` <= x < `1.0`
  - class `Random` from package `java.util`
    - Can produce pseudorandom `boolean`, `byte`, `float`, `double`, `int`, `long` and Gaussian values
    - Is seeded with the current time of day to generate different sequences of numbers each time the program executes

```
// Fig. 6.7: RandomIntegers.java
// Shifted and scaled random integers.
import java.util.Random; // program uses class Random

public class RandomIntegers
{
   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
```

Import class **`Random`** from the **`java.util`** package

Create a **`Random`** object

Generate a random die roll

`RandomIntegers.java`

(1 of 2)

```
1   5   3   6   2
5   2   6   5   2
4   4   4   2   6
3   1   6   2   2
```

```
6   5   4   2   6
1   2   5   1   3
6   3   2   2   1
6   4   2   6   4
```

Two different sets of results containing integers in the range 1-6

`RandomIntegers.java`

(2 of 2)

Outline

```
// Fig. 6.8: RollDie.java
// Roll a six-sided die 6000 times.
import java.util.Random;

public class RollDie
{
   public static void main( String args[] )
   {
      Random randomNumbers = new Random(); // random number generator

      int frequency1 = 0; // maintains count of 1s rolled
      int frequency2 = 0; // count of 2s rolled
      int frequency3 = 0; // count of 3s rolled
      int frequency4 = 0; // count of 4s rolled
      int frequency5 = 0; // count of 5s rolled
      int frequency6 = 0; // count of 6s rolled

```

Import class **Random** from the **java.util** package

Create a **Random** object

Declare frequency counters

RollDie.java

(1 of 2)

```
      int face; // stores most recently rolled value

      // summarize results of 6000 rolls of a die
      for ( int roll = 1; roll <= 6000; roll++ )
      {
         face = 1 + randomNumbers.nextInt( 6 ); // number from 1 to 6

         // determine roll value 1-6 and increment appropriate counter
         switch ( face )
         {
            case 1:
               ++frequency1; // increment the 1s counter
               break;
            case 2:
               ++frequency2; // increment the 2s counter
               break;
            case 3:
               ++frequency3; // increment the 3s counter
               break;
            case 4:
               ++frequency4; // increment the 4s counter
               break;
            case 5:
               ++frequency5; // increment the 5s counter
               break;
            case 6:
               ++frequency6; // increment the 6s counter
               break; // optional at end of switch
         } // end switch
      } // end for

```

Iterate `6000` times

RollDie.java

Generate a random die roll

`switch` based on the die roll

```
      System.out.println( "Face\tFrequency" ); // output headers
      System.out.printf( "1\t%d\n2\t%d\n3\t%d\n4\t%d\n5\t%d\n6\t%d\n",
         frequency1, frequency2, frequency3, frequency4,
         frequency5, frequency6 );
   } // end main
} // end class RollDie
```

Display die roll frequencies

```
Face	Frequency
1	982
2	1001
3	1015
4	1005
5	1009
6	988
```

```
Face	Frequency
1	1029
2	994
3	1017
4	1007
5	972
6	981
```

RollDie.java

(3 of 3)

Generalized Scaling and Shifting of Random Numbers

- To generate a random number in certain sequence or range
  - Use the expression
    shiftingValue + *differenceBetweenValues* * `randomNumbers.nextInt(` *scalingFactor* `)`
    where:
    - *shiftingValue* is the first number in the desired range of values
    - *differenceBetweenValues* represents the difference between consecutive numbers in the sequence
    - *scalingFactor* specifies how many numbers are in the range

Random-Number Repeatability for Testing and Debugging

- To get a `Random` object to generate the same sequence of random numbers every time the program executes, seed it with a certain value
  - When creating the `Random` object:

    ```
    Random randomNumbers =
      new Random( seedValue );
    ```
  - Use the `setSeed` method:

    ```
    randomNumbers.setSeed( seedValue );
    ```
  - `seedValue` should be an argument of type `long`

```
// Fig. 6.9: Craps.java
// Craps class simulates the dice game craps.
import java.util.Random;

public class Craps
{
   // create random number generator for use in method rollDice
   private Random randomNumbers = new Random();

   // enumeration with constants that represent the game status
   private enum Status { CONTINUE, WON, LOST };

   // constants that represent common rolls of the dice
   private final static int SNAKE_EYES = 2;
   private final static int TREY = 3;
   private final static int SEVEN = 7;
   private final static int YO_LEVEN = 11;
   private final static int BOX_CARS = 12;

```

Import class **Random** from the **java.util** package

Create a **Random** object

Declare an enumeration

Declare constants

Craps.java

(1 of 4)

```
   // plays one game of craps
   public void play()
   {
      int myPoint = 0; // point if no win or loss on first roll
      Status gameStatus; // can contain CONTINUE, WON or LOST

      int sumOfDice = rollDice(); // first roll of the dice

      // determine game status and point based on first roll
      switch ( sumOfDice )
      {
         case SEVEN: // win with 7 on first roll
         case YO_LEVEN: // win with 11 on first roll
            gameStatus = Status.WON;
            break;
         case SNAKE_EYES: // lose with 2 on first roll
         case TREY: // lose with 3 on first roll
         case BOX_CARS: // lose with 12 on first roll
            gameStatus = Status.LOST;
            break;
         default: // did not win or lose, so remember point
            gameStatus = Status.CONTINUE; // game is not over
            myPoint = sumOfDice; // remember the point
            System.out.printf( "Point is %d\n", myPoint );
            break; // optional at end of switch
      } // end switch

```

Call **rollDice** method

Craps.java

(2 of 4)

Player wins with a roll of 7 or 11

Player loses with a roll of 2, 3 or 12

Set and display the point

```
      // while game is not complete
      while ( gameStatus == Status.CONTINUE ) // not WON or LOST
      {
         sumOfDice = rollDice(); // roll dice again

         // determine game status
         if ( sumOfDice == myPoint ) // win by making point
            gameStatus = Status.WON;
         else
            if ( sumOfDice == SEVEN ) // lose by rolling 7 before point
               gameStatus = Status.LOST;
      } // end while

      // display won or lost message
      if ( gameStatus == Status.WON )
         System.out.println( "Player wins" );
      else
         System.out.println( "Player loses" );
   } // end method play

```

Call **`rollDice`** method

Craps.java

(2 of 4)

Player wins by making the point

Player loses by rolling 7

Display outcome

```
    // roll dice, calculate sum and display results
    public int rollDice()
    {
       // pick random die values
       int die1 = 1 + randomNumbers.nextInt( 6 ); // first die roll
       int die2 = 1 + randomNumbers.nextInt( 6 ); // second die roll

       int sum = die1 + die2; // sum of die values

       // display results of this roll
       System.out.printf( "Player rolled %d + %d = %d\n",
          die1, die2, sum );

       return sum; // return sum of dice
    } // end method rollDice
 } // end class Craps
```

Declare **rollDice** method

Craps.java

(4 of 4)

Generate two dice rolls

Display dice rolls and their sum

```java
// Fig. 6.10: CrapsTest.java
// Application to test class Craps.

public class CrapsTest
{
   public static void main( String args[] )
   {
      Craps game = new Craps();
      game.play(); // play one game of craps
   } // end main
} // end class CrapsTest
```

```
Player rolled 5 + 6 = 11
Player wins
```

```
Player rolled 1 + 2 = 3
Player loses
```

```
Player rolled 5 + 4 = 9
Point is 9
Player rolled 2 + 2 = 4
Player rolled 2 + 6 = 8
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
Player wins
```

```
Player rolled 2 + 6 = 8
Point is 8
Player rolled 5 + 1 = 6
Player rolled 2 + 1 = 3
Player rolled 1 + 6 = 7
Player loses
```

CrapsTest.java

(1 of 2)

Case Study: A Game of Chance (Introducing Enumerations)

- Enumerations
  - Programmer-declared types consisting of sets of constants
  - `enum` keyword
  - A type name (e.g. `Status`)
  - Enumeration constants (e.g. `WON`, `LOST` and `CONTINUE`)
    - cannot be compared against `int`s

Scope of Declarations

- Basic scope rules
  - Scope of a parameter declaration is the body of the method in which appears
  - Scope of a local-variable declaration is from the point of declaration to the end of that block
  - Scope of a local-variable declaration in the initialization section of a `for` header is the rest of the `for` header and the body of the `for` statement
  - Scope of a method or field of a class is the entire body of the class

Scope of Declarations (Cont.)

- **Shadowing**
  - **A field is shadowed (or hidden) if a local variable or parameter has the same name as the field**
    - **This lasts until the local variable or parameter goes out of scope**

```
// Fig. 6.11: Scope.java
// Scope class demonstrates field and local variable scopes.

public class Scope
{
   // field that is accessible to all methods of this class
   private int x = 1;

   // method begin creates and initializes local variable x
   // and calls methods useLocalVariable and useField
   public void begin()
   {
      int x = 5; // method's local variable x shadows field x

      System.out.printf( "local x in method begin 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

```

Shadows field **x**

Display value of local variable **x**

Scope.java

(1 of 2)

```
      System.out.printf( "\nlocal x in method begin is %d\n", x );
   } // end method begin

   // create and initialize local variable x during each call
   public 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 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
```

Shadows field **x**

Display value of local variable **x**

Display value of field **x**

Scope.java

(2 of 2)

```java
// Fig. 6.12: ScopeTest.java
// Application to test class Scope.

public class ScopeTest
{
   // application starting point
   public static void main( String args[] )
   {
      Scope testScope = new Scope();
      testScope.begin();
   } // end main
} // end class ScopeTest
```

```
local x in method begin 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 method begin is 5
```

ScopeTest.java

Method Overloading

- Method overloading
  - Multiple methods with the same name, but different types, number or order of parameters in their parameter lists
  - Compiler decides which method is being called by matching the method call's argument list to one of the overloaded methods' parameter lists
    - A method's name and number, type and order of its parameters form its signature
  - Differences in return type are irrelevant in method overloading
    - Overloaded methods can have different return types
    - Methods with different return types but the same signature cause a compilation error

```
// Fig. 6.13: MethodOverload.java
// Overloaded method declarations.

public class MethodOverload
{
   // test overloaded square methods
   public void testOverloadedMethods()
   {
      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 ) );
   } // end method testOverloadedMethods

   // square method with int argument
   public 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 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
```

MethodOverload.java

Correctly calls the "**square** of **int**" method

Correctly calls the "**square** of **double**" method

Declaring the "**square** of **int**" method

Declaring the "**square** of **double**" method

```java
// Fig. 6.14: MethodOverloadTest.java
// Application to test class MethodOverload.

public class MethodOverloadTest
{
   public static void main( String args[] )
   {
      MethodOverload methodOverload = new MethodOverload();
      methodOverload.testOverloadedMethods();
   } // end main
} // end class MethodOverloadTest
```

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

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Outline

MethodOverloadTest.java

```java
// Fig. 6.15: MethodOverloadError.java
// Overloaded methods with identical signatures
// cause compilation errors, even if return types are different.

public class MethodOverloadError
{
   // declaration of method square with int argument
   public int square( int x )
   {
      return x * x;
   }

   // second declaration of method square with int argument
   // causes compilation error even though return types are different
   public double square( int y )
   {
      return y * y;
   }
} // end class MethodOverloadError
```

MethodOverload

Error.java

Same method signature

```
MethodOverloadError.java:15: square(int) is already defined in
MethodOverloadError
   public double square( int y )
                 ^
1 error
```

Compilation error

Arrays

OBJECTIVES

We will cover:

- What arrays are.
- To use arrays to store data in and retrieve data from lists and tables of values.
- To declare an array, initialize an array and refer to individual elements of an array.
- To use the enhanced **`for`** statement to iterate through arrays.
- To pass arrays to methods.
- To declare and manipulate multidimensional arrays.
- To write methods that use variable-length argument lists.
- To read command-line arguments into a program.

Outline

Introduction

- Arrays
  - Data structures
  - Related data items of same type
  - Remain same size once created
    - Fixed-length entries

Arrays

- Array
  - Group of variables
    - Have same type
  - Reference type

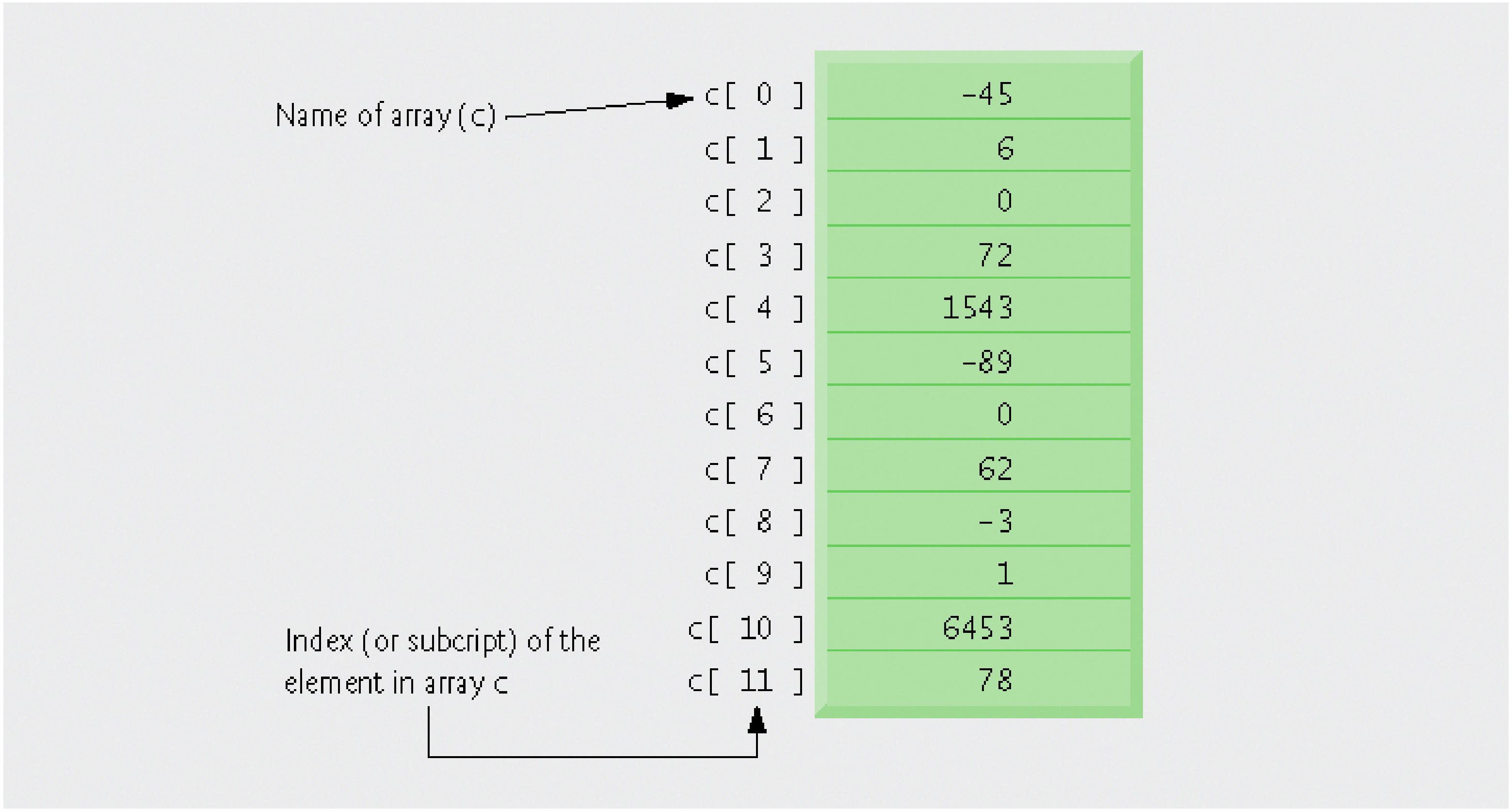

Fig. 7.1 | A 12-element array.

Arrays (Cont.)

- Index
  - Also called subscript
  - Position number in square brackets
  - Must be positive integer or integer expression
  - First element has index zero

```
a = 5;
b = 6;
c[ a + b ] += 2;
```

- Adds `2` to `c[ 11 ]`

Arrays (Cont.)

- Examine array `c`
  - `c` is the array *name*
  - `c.length` accesses array `c`'s *length*
  - `c` has 12 *elements* (`c[0]`, `c[1]`, … `c[11]`)
    - The *value* of `c[0]` is `–45`

Declaring and Creating Arrays

- Declaring and Creating arrays
  - Arrays are objects that occupy memory
  - Created dynamically with keyword `new`

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

  - Equivalent to

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

- We can create arrays of objects too

```
String b[] = new
  String[ 100 ];
```

Examples Using Arrays

- Declaring arrays
- Creating arrays
- Initializing arrays
- Manipulating array elements

Outline

```
 1 // Fig. 7.2: InitArray.java
 2 // Creating an array.
 3
 4 public class InitArray
 5 {
 6    public static void main( Stri
 7    {
 8       int array[]; // declare array named array
 9
10       array = new int[ 10 ]; // create the sp
11
12       System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings
13
14       // output each array element's value
15       for ( int counter = 0; counter < array.length; counter++ )
16          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
17    } // end main
18 } // 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
```

Declare `array` as an array of `int`s

Create `10` `int`s for `array`; each `int` is initialized to `0` by default

`array.length` returns length of `array`

Each `int` is initialized to `0` by default

`array[counter]` returns `int` associated with index in `array`

`InitArray.java`

Line 8
Declare array as an array of **ints**

Line 10
Create 10 **ints** for array; each **int** is initialized to **0** by default

Line 15
array.length returns length of array

Line 16
array[counter] returns **int** associated with index in array

Program output

Examples Using Arrays (Cont.)

- Using an array initializer
  - Use *initializer list*
    - Items enclosed in braces (`{}`)
    - Items in list separated by commas

```
int n[] = { 10, 20, 30, 40,
50 };
```

  - Creates a five-element array
  - Index values of `0`, `1`, `2`, `3`, `4`
- Do not need keyword `new`

Outline

```
// Fig. 7.3: InitArray.java
// Initializing the elements of an array with an array initializer.

public class InitArray
{
   public static void main( String args[] )
   {
      // initializer list specifies the value for each element
      int array[] = { 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 ] );
   } // end main
} // end class InitArray
```

Declare `array` as an array of `int`s

Compiler uses initializer list to allocate array

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

itArray2.java

Line 9
Declare array as an array of ints

Line 9
Compiler uses initializer list to allocate array

Program output

Examples Using Arrays (Cont.)

- Calculating a value to store in each array element
  - Initialize elements of 10-element array to even integers

Outline

```
 1 // Fig. 7.4: InitArray.java
 2 // Calculating values to be placed into elements of an array.
 3
 4 public class InitArray
 5 {
 6    public static void main( String args[] )
 7    {
 8       final int ARRAY_LENGTH = 10; // declare constant
 9       int array[] = new int[ ARRAY_LENGTH ]; // create ar
10
11       // calculate value for each array element
12       for ( int counter = 0; counter < array.length; counter++ )
13          array[ counter ] = 2 + 2 * counter;
14
15       System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings
16
17       // output each array element's value
18       for ( int counter = 0; counter < array.length; counter++ )
19          System.out.printf( "%5d%8d\n", counter
20    } // end main
21 } // end class InitArray
```

Declare constant variable ARRAY_LENGTH using the `final` modifier

Declare and create `array` that contains 10 `int`s

Use `array` index to assign array value

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

InitArray.java

8
are constant
able

Line 9
Declare and create array that contains 10 **int**s

Line 13
Use array index to assign array

Program output

Examples Using Arrays (Cont.)

- **Summing the elements of an array**
  - **Array elements can represent a series of values**
    - **We can sum these values**

Outline

SumArray.java

Line 8
Declare array with initializer list

Lines 12-13
Sum all array values

Program output

```java
1  // Fig. 7.5: SumArray.java
2  // Computing the sum of the elements of
3
4  public class SumArray
5  {
6     public static void main( String args[] )
7     {
8        int array[] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // add each element's value to total
12       for ( int counter = 0; counter < array.length; counter++ )
13          total += array[ counter ];
14
15       System.out.printf( "Total of array elements: %d\n",   total );
16    } // end main
17 } // end class SumArray
```

Declare array with initializer list

Sum all array values

```
Total of array elements: 849
```

Examples Using Arrays (Cont.)

- Using bar charts to display array data graphically
  - Present data in graphical manner
    - E.g., bar chart
  - Examine the distribution of grades

Outline

```
 1 // Fig. 7.6: BarChart.java
 2 // Bar chart printing program.
 3
 4 public class BarChart
 5 {
 6    public static void main( String args[] )
 7    {
 8       int array[] = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
 9
10       System.out.println( "Grade distribution:" );
11
12       // for each array element, output a bar of the chart
13       for ( int counter = 0; counter < array.length; counter++ )
14       {
15          // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
16          if ( counter == 10 )
17             System.out.printf( "%5d: ", 100 );
18          else
19             System.out.printf( "%02d-%02d: ",
20                counter * 10, counter * 10 + 9 );
21
22          // print bar of asterisks
23          for ( int stars = 0; stars < array[ counter ]; stars++ )
24             System.out.print( "*" );
25
26          System.out.println(); // start a new line of output
27       } // end outer for
28    } // end main
29 } // end class BarChart
```

Declare `array` with initializer list

Use the `0` flag to display one-digit grade with a leading 0

For each `array` element, print associated number of asterisks

`BarChart.java`

(1 of 2)

`Line 8`
`Declare array with initializer list`

`Line 19`
`Use the 0 flag to display one-digit grade with a leading 0`

`Lines 23-24`
`For each array element, print`

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

Outline

BarChart.java

(2 of 2)

Program output

Examples Using Arrays (Cont.)

- **Using the elements of an array as counters**
  - **Use a series of counter variables to summarize data**

Outline

```
1  // Fig. 7.7: RollDie.java
2  // Roll a six-sided die 6000 times.
3  import java.util.Random;
4
5  public class RollDie
6  {
7     public static void main( String args[] )
8     {
9        Random randomNumbers = new Random(); // random number generator
10       int frequency[] = new int[ 7 ]; // array of frequency counters
11
12       // roll die 6000 times; use die value as frequency ind
13       for ( int roll = 1; roll <= 6000; roll++ )
14          ++frequency[ 1 + randomNumbers.nextInt( 6 ) ];
15
16       System.out.printf( "%s%10s\n"
17
18       // output each array element's value
19       for ( int face = 1; face < frequency.length; face++ )
20          System.out.printf( "%4d%10d\n", face, frequency[ face ] );
21    } // end main
22 } // end class RollDie
```

Declare `frequency` as array of 7 `ints`

Generate 6000 random integers in range 1-6

Increment `frequency` values at index associated with random number

```
Face Frequency
   1       988
   2       963
   3      1018
   4      1041
   5       978
   6      1012
```

Die.java

Line 10
Declare frequency as array of 7 ints

3-14
e 6000 random integers in range 1-6

Line 14
Increment frequency values at index associated with random number

Program output

Examples Using Arrays (Cont.)

- Using arrays to analyze survey results
  - 40 students rate the quality of food
    - 1-10 Rating scale: 1 means awful, 10 means excellent
  - Place 40 responses in array of integers
  - Summarize results

Outline

```
1  // Fig. 7.8: StudentPoll.java
2  // Poll analysis program.
3
4  public class StudentPoll
5  {
6     public static void main( String args[] )
7     {
8        // array of survey responses
9        int responses[] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
10          10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
11          4, 8, 6, 8, 10 };
12       int frequency[] = new int[ 11 ]; // array of frequency counters
13
14       // for each answer, select responses element and use that value
15       // as frequency index to determine element to increment
16       for ( int answer = 0; answer < responses.length; answer++ )
17          ++frequency[ responses[ answer ] ];
18
19       System.out.printf( "%s%10s", "Rating", "Frequency" )
20
21       // output each array element's value
22       for ( int rating = 1; rating < frequency.length; rating++ )
23          System.out.printf( "%d%10d", rating, frequency[ rating ] );
24    } // end main
25 } // end class StudentPoll
```

Declare `responses` as array to store 40 responses

Declare `frequency` as array of 11 `int` and ignore the first element

For each response, increment `frequency` values at index associated with that response

`tudentPoll.java`

(1 of 2)

`ponses`
`as array to store`
`40 responses`

`Line 12`
`Declare frequency`
`as array of 11 int`
`and ignore the`
`first element`

`17`
`response,`
`frequency`
`index`
`d with`
`that response`

Outline

```
Rating Frequency
     1         2
     2         2
     3         2
     4         2
     5         5
     6        11
     7         5
     8         7
     9         1
    10         3
```

`StudentPoll.java`

(2 of 2)

Program output

Enhanced `for` Statement

- Enhanced `for` statement
  - Iterates through elements of an array or a collection without using a counter
  - Syntax

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

Outline

EnhancedForTest.java

```
// Fig. 7.12: EnhancedForTest.java
// Using enhanced for statement to total integers in an array.

public class EnhancedForTest
{
   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 number : array )
         total += number;

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

For each iteration, assign the next element of `array` to `int` variable `number`, then add it to `total`

```
Total of array elements: 849
```

Enhanced for Statement (Cont.)

- Lines 12-13 are equivalent to

```
for ( int counter = 0; counter < array.length; counter++ )
   total += array[ counter ];
```

- Usage
  - Can access array elements
  - Cannot modify array elements
  - Cannot access the counter indicating the index

Passing Arrays to Methods

- To pass array argument to a method
  - Specify array name without brackets
    - Array `hourlyTemperatures` is declared as

      ```
      int hourlyTemperatures = new int[24];
      ```

    - The method call

      ```
      modifyArray( hourlyTemperatures );
      ```

    - Passes array `hourlyTemperatures` to method `modifyArray`

```
1  // Fig. 7.13: PassArray.java
2  // Passing arrays and individual array elements to methods.
3
4  public class PassArray
5  {
6     // main creates array and calls modifyArray and modifyElement
7     public static void main( String args[] )
8     {
9        int array[] = { 1, 2, 3, 4, 5 };
10
11       System.out.println(
12          "Effects of passing reference to entire array:\n" +
13          "The values of the original array are:" );
14
15       // output original array elements
16       for ( int value : array )
17          System.out.printf( "   %d", value );
18
19       modifyArray( array ); // pass array reference
20       System.out.println( "\n\nThe values of the modified array are:" );
21
22       // output modified array elements
23       for ( int value : array )
24          System.out.printf( "   %d", value );
25
26       System.out.printf(
27          "\n\nEffects of passing array element value:\n" +
28          "array[3] before modifyElement: %d\n", array[ 3 ] );
```

Declare 5-`int` `array` with initializer list

Pass entire array to method `modifyArray`

<u>Outline</u>

`PassArray.java`

(1 of 2)

ine 9

Line 19

Outline

ava

(2 of 2)

```
29
30       modifyElement( array[ 3 ] ); // attempt to modify array[ 3 ]
31       System.out.printf(
32          "array[3] after modifyElement
33    } // end main
34
35    // multiply each element of an array by 2
36    public static void modifyArray( int array2[] )
37    {
38       for ( int counter = 0; counter < array2.length; counter++ )
39          array2[ counter ] *= 2;
40    } // end method modifyArray
41
42    // multiply argument by 2
43    public static void modifyElement( int element )
44    {
45       element *= 2;
46       System.out.printf(
47          "Value of element in modifyElement: %d\n", element );
48    } // end method modifyElement
49 } // end class PassArray
```

Pass array element `array[3]` to method `modifyElement`

Method `modifyArray` manipulates the array directly

Method `modifyElement` manipulates a primitive's copy

Line 30

Lines 36-40

Lines 43-48

```
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
```

Program output

Passing Arrays to Methods (Cont.)

- Notes on passing arguments to methods
  - Two ways to pass arguments to methods
    - Pass-by-value
      - Copy of argument's value is passed to called method
      - Every primitive type is passed-by-value
    - Pass-by-reference
      - Caller gives called method direct access to caller's data
      - Called method can manipulate this data
      - Improved performance over pass-by-value
      - Every object is passed-by-reference
        - Arrays are objects
        - Therefore, arrays are passed by reference

Case Study: Class GradeBook Using an Array to Store Grades

- **Further evolve class GradeBook**
- **Class GradeBook**
  - **Represents a grade book that stores and analyzes grades**
  - **Does not maintain individual grade values**
  - **Repeat calculations require reentering the same grades**
    - **Can be solved by storing grades in an array**

Outline

```
1  // Fig. 7.14: GradeBook.java
2  // Grade book using an array to store test grades.
3
4  public class GradeBook
5  {
6     private String courseName; // name of course this GradeBook represents
7     private int grades[]; // array of student grades
8
9     // two-argument constructor initializes courseName
10    public GradeBook( String name, int gradesArray[] )
11    {
12       courseName = name; // initialize courseName
13       grades = gradesArray; // store grades
14    } // end two-argument GradeBook constructor
15
16    // method to set the course name
17    public void setCourseName( String name )
18    {
19       courseName = name; // store the course name
20    } // end method setCourseName
21
22    // method to retrieve the course name
23    public String getCourseName()
24    {
25       return courseName;
26    } // end method getCourseName
27
```

Declare array `grades` to store individual grades

Assign the array's reference to instance variable `grades`

`GradeBook.java`

(1 of 5)

ine 7

Line 13

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

   // perform various operations on the data
   public void processGrades()
   {
      // output grades array
      outputGrades();

      // call method getAverage to calculate the average grade
      System.out.printf( "\nClass average is %.2f\n", getAverage() );

      // call methods getMinimum and getMaximum
      System.out.printf( "Lowest grade is %d\nHighest grade is %d\n\n",
         getMinimum(), getMaximum() );

      // call outputBarChart to print grade distribution chart
      outputBarChart();
   } // end method processGrades

   // find minimum grade
   public int getMinimum()
   {
      int lowGrade = grades[ 0 ]; // assume grades[ 0 ] is smallest

```

Outline

GradeBook.java

(2 of 5)

Outline

radeBook.java

(3 of 5)

Lines 59-64

Lines 75-80

```
58        // loop through grades array
59        for ( int grade : grades )
60        {
61           // if grade lower than lowGrade, assign it to lowGrade
62           if ( grade < lowGrade )
63              lowGrade = grade; // new lowest grade
64        } // end for
65
66        return lowGrade; // return lowest grade
67     } // end method getMinimum
68
69     // find maximum grade
70     public int getMaximum()
71     {
72        int highGrade = grades[ 0 ]; // assume grades[ 0 ] is largest
73
74        // loop through grades array
75        for ( int grade : grades )
76        {
77           // if grade greater than highGrade, assign it to highGrade
78           if ( grade > highGrade )
79              highGrade = grade; // new highest grade
80        } // end for
81
82        return highGrade; // return highest grade
83     } // end method getMaximum
84
```

Loop through `grades` to find the lowest grade

Loop through `grades` to find the highest grade

```
85    // determine average grade for test
86    public double getAverage()
87    {
88       int total = 0; // initialize total
89
90       // sum grades for one student
91       for ( int grade : grades )
92          total += grade;
93
94       // return average of grades
95       return (double) total / grades.length;
96    } // end method getAverage
97
98    // output bar chart displaying grade distribution
99    public void outputBarChart()
100     {
101        System.out.println( "Grade distribution:" );
102
103        // stores frequency of grades in each range of 10 grades
104        int frequency[] = new int[ 11 ];
105
106        // for each grade, increment the appropriate frequency
107        for ( int grade : grades )
108           ++frequency[ grade / 10 ];
109
```

Loop through `grades` to sum grades for one student

Loop through `grades` to calculate frequency

Outline

`GradeBook.java`

(4 of 5)

Lines 91-92

Lines 107-108

Outline

`GradeBook.java`

(5 of 5)

Lines 134-136

```
110       // for each grade frequency, print bar in chart
111       for ( int count = 0; count < frequency.length; count++ )
112       {
113          // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
114          if ( count == 10 )
115             System.out.printf( "%5d: ", 100 );
116          else
117             System.out.printf( "%02d-%02d: ",
118                count * 10, count * 10 + 9  );
119
120          // print bar of asterisks
121          for ( int stars = 0; stars < frequency[ count ]; stars++ )
122             System.out.print( "*" );
123
124          System.out.println(); // start a new line of output
125       } // end outer for
126    } // end method outputBarChart
127
128    // output the contents of the grades array
129    public void outputGrades()
130    {
131       System.out.println( "The grades are:\n" );
132
133       // output each student's grade
134       for ( int student = 0; student < grades.length; student++ )
135          System.out.printf( "Student %2d: %3d\n",
136             student + 1, grades[ student ] );
137    } // end method outputGrades
138 } // end class GradeBook
```

Loop through `grades` to display each grade

```
1  // Fig. 7.15: GradeBookTest.java
2  // Creates GradeBook object using an array of grades.
3
4  public class GradeBookTest
5  {
6     // main method begins program execution
7     public static void main( String args[] )
8     {
9        // array of student grades
10       int gradesArray[] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11
12       GradeBook myGradeBook = new GradeBook(
13          "CS101 Introduction to Java Programming", gradesArray );
14       myGradeBook.displayMessage();
15       myGradeBook.processGrades();
16    } // end main
17 } // end class GradeBookTest
```

Declare and initialize `gradesArray` with 10 elements

Pass `gradesArray` to `GradeBook` constructor

Outline

eBookTest

.java

(1 of 2)

Line 10

Line 13

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

The grades are:

Student  1:  87
Student  2:  68
Student  3:  94
Student  4: 100
Student  5:  83
Student  6:  78
Student  7:  85
Student  8:  91
Student  9:  76
Student 10:  87

Class average is 84.90
Lowest grade is 68
Highest grade is 100

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

Outline

GradeBookTest

.java

(2 of 2)

Program output

Multidimensional Arrays

- **Multidimensional arrays**
  - **Tables with rows and columns**
    - **Two-dimensional array**
    - **m-by-n array**

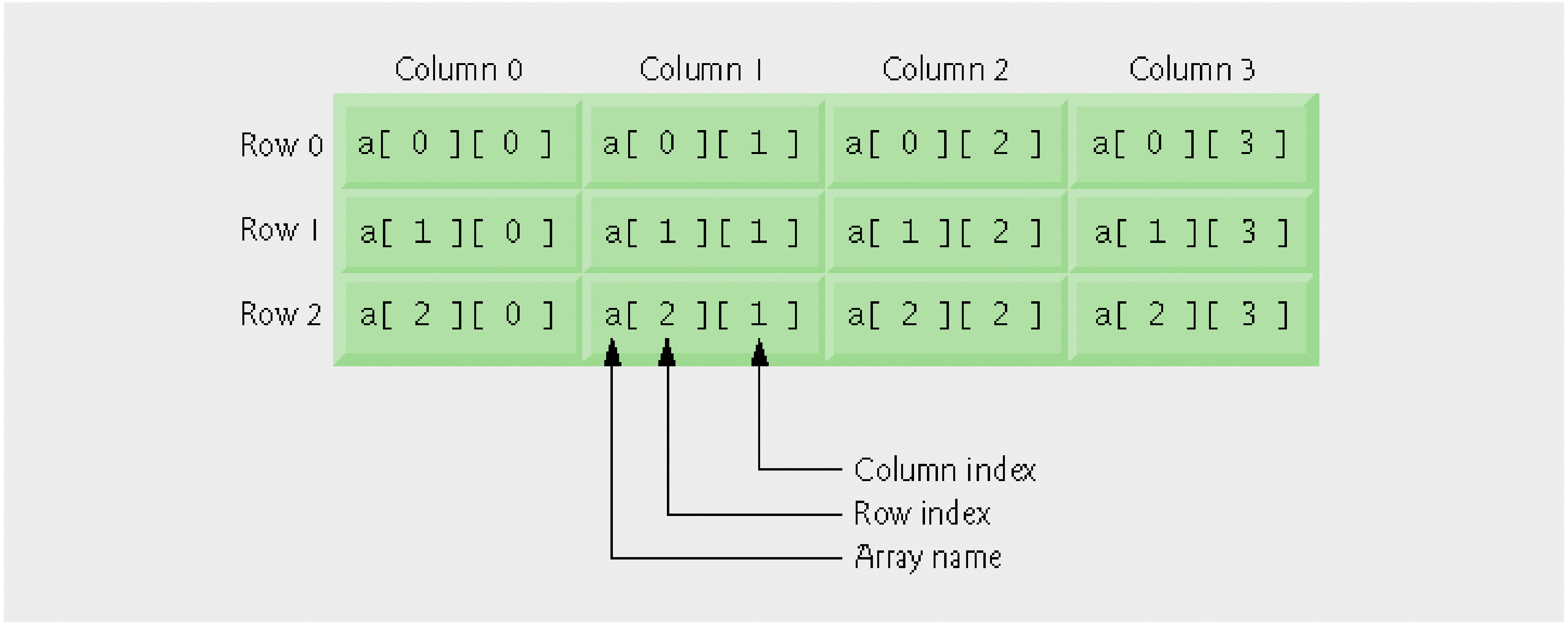

Fig. 7.16 | Two-dimensional array with three rows and four columns.

Multidimensional Arrays (Cont.)

- Arrays of one-dimensional array
  - Declaring two-dimensional array `b[2][2]`

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

- 1 and 2 initialize `b[0][0]` and `b[0][1]`
- 3 and 4 initialize `b[1][0]` and `b[1][1]`

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

- row 0 contains elements 1 and 2
- row 1 contains elements 3, 4 and 5

Multidimensional Arrays (Cont.)

- **Two-dimensional arrays with rows of different lengths**
  - **Lengths of rows in array are not required to be the same**
    - **E.g.,** `int b[][] = { { 1, 2 }, { 3, 4, 5 } };`

Multidimensional Arrays (Cont.)

- Creating two-dimensional arrays with array-creation expressions
  - 3-by-4 array

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

  - Rows can have different number of columns

```
int b[][];

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
```

```
1  // Fig. 7.17: InitArray.java
2  // Initializing two-dimensional arrays.
3
4  public class InitArray
5  {
6     // create and output two-dimensional arrays
7     public static void main( String args[] )
8     {
9        int array1[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
10       int array2[][] = { { 1, 2 }, { 3 }, { 4, 5, 6
11
12       System.out.println( "Values in array1 by row are" );
13       outputArray( array1 ); // displays array1 by row
14
15       System.out.println( "\nValues in array2 by row are" );
16       outputArray( array2 ); // displays array2 by row
17    } // end main
18
```

Use nested array initializers to initialize `array1`

Use nested array initializers of different lengths to initialize `array2`

<u>Outline</u>

`InitArray4.java`

1 of 2)

Line 9

Line 10

Outline

InitArray4.java

(2 of 2)

Line 26

Line 27

Program output

```
19    // output rows and columns of a two-dimensional array
20    public static void outputArray( int array[][] )
21    {
22       // loop through array's rows
23       for ( int row = 0; row < array.length; row++ )
24       {
25          // loop through columns of current row
26          for ( int column = 0; column < array[ row ].length; column++ )
27             System.out.printf( "%d  ", array[ row ][ column ] );
28
29          System.out.println(); // start new line of output
30       } // end outer for
31    } // end method outputArray
32 } // end class InitArray
```

`array[row].length` returns number of columns associated with `row` subscript

Use double-bracket notation to access two-dimensional array values

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

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

Multidimensional Arrays (Cont.)

- **Common multidimensional-array manipulations performed with `for` statements**
  - **Many common array manipulations use `for` statements**

E.g.,

```
for ( int column = 0; column < a[ 2 ].length; column++ )
    a[ 2 ][ column ] = 0;
```

Case Study: Class GradeBook Using a Two-Dimensional Array

- Class `GradeBook`
  - One-dimensional array
    - Store student grades on a single exam
  - Two-dimensional array
    - Store grades for a single student and for the class as a whole

```
 1 // Fig. 7.18: GradeBook.java
 2 // Grade book using a two-dimensional array to store grades.
 3
 4 public class GradeBook
 5 {
 6    private String courseName; // name of course this grade book represents
 7    private int grades[][]; // two-dimensional array of student grades
 8
 9    // two-argument constructor initializes courseName and grades array
10    public GradeBook( String name, int gradesArray[][] )
11    {
12       courseName = name; // initialize courseName
13       grades = gradesArray; // store grades
14    } // end two-argument GradeBook constructor
15
16    // method to set the course name
17    public void setCourseName( String name )
18    {
19       courseName = name; // store the course name
20    } // end method setCourseName
21
22    // method to retrieve the course name
23    public String getCourseName()
24    {
25       return courseName;
26    } // end method getCourseName
27
```

Declare two-dimensional array grades

`GradeBook` constructor accepts a `String` and a two-dimensional array

Outline

`GradeBook.java`

(1 of 7)

Line 7

Line 10

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

   // perform various operations on the data
   public void processGrades()
   {
      // output grades array
      outputGrades();

      // call methods getMinimum and getMaximum
      System.out.printf( "\n%s %d\n%s %d\n\n",
         "Lowest grade in the grade book is", getMinimum(),
         "Highest grade in the grade book is", getMaximum() );

      // output grade distribution chart of all grades on all tests
      outputBarChart();
   } // end method processGrades

   // find minimum grade
   public int getMinimum()
   {
      // assume first element of grades array is smallest
      int lowGrade = grades[ 0 ][ 0 ];

```

Outline

GradeBook.java

(2 of 7)

Outline

```
57       // loop through rows of grades array
58       for ( int studentGrades[] : grades )
59       {
60          // loop through columns of current row
61          for ( int grade : studentGrades
62          {
63             // if grade less than lowGra
64             if ( grade < lowGrade )
65                lowGrade = grade;
66          } // end inner for
67       } // end outer for
68
69       return lowGrade; // return lowest grade
70    } // end method getMinimum
71
72    // find maximum grade
73    public int getMaximum()
74    {
75       // assume first element of grades array is largest
76       int highGrade = grades[ 0 ][ 0 ];
77
```

Loop through rows of `grades` to find the lowest grade of any student

`GradeBook.java`

(3 of 7)

Lines 58-67

Outline

GradeBook.java

(4 of 7)

Lines 79-88

Lines 94-104

```
78       // loop through rows of grades array
79       for ( int studentGrades[] : grades )
80       {
81          // loop through columns of current row
82          for ( int grade : studentGrad
83          {
84             // if grade greater than h
85             if ( grade > highGrade )
86                highGrade = grade;
87          } // end inner for
88       } // end outer for
89
90       return highGrade; // return highest grade
91    } // end method getMaximum
92
93    // determine average grade for particular set of grades
94    public double getAverage( int setOfGrades[] )
95    {
96       int total = 0; // initialize total
97
98       // sum grades for one student
99       for ( int grade : setOfGrades )
100         total += grade;
101
102      // return average of grades
103      return (double) total / setOfGrades.length;
104   } // end method getAverage
105
```

Loop through rows of `grades` to find the highest grade of any student

Calculate a particular student's semester average

Outline

```
106     // output bar chart displaying overall grade distribution
107     public void outputBarChart()
108     {
109        System.out.println( "Overall grade distribution:" );
110
111        // stores frequency of grades in each range of 10 grades
112        int frequency[] = new int[ 11 ];
113
114        // for each grade in GradeBook, increment the appropriate frequency
115        for ( int studentGrades[] : grades )
116        {
117           for ( int grade : studentGrades )
118              ++frequency[ grade / 10 ];
119        } // end outer for
120
121        // for each grade frequency, print bar in chart
122        for ( int count = 0; count < frequency.length; count++ )
123        {
124           // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
125           if ( count == 10 )
126              System.out.printf( "%5d: ", 100 );
127           else
128              System.out.printf( "%02d-%02d: ",
129                 count * 10, count * 10 + 9  );
130
131           // print bar of asterisks
132           for ( int stars = 0; stars < frequency[ count ]; stars++ )
133              System.out.print( "*" );
```

Calculate the distribution of all student grades

GradeBook.java

(5 of 7)

Lines 115-119

Outline

```java
         System.out.println(); // start a new line of output
      } // end outer for
   } // end method outputBarChart

   // output the contents of the grades array
   public void outputGrades()
   {
      System.out.println( "The grades are:\n" );
      System.out.print( "            " ); // align column heads

      // create a column heading for each of the tests
      for ( int test = 0; test < grades[ 0 ].length; test++ )
         System.out.printf( "Test %d  ", test + 1 );

      System.out.println( "Average" ); // student average column heading

      // create rows/columns of text representing array grades
      for ( int student = 0; student < grades.length; student++ )
      {
         System.out.printf( "Student %2d", student + 1 );

         for ( int test : grades[ student ] ) // output student's grades
            System.out.printf( "%8d", test );

```

GradeBook.java

(6 of 7)

```
            // call method getAverage to calculate student's average grade;
            // pass row of grades as the argument to getAverage
            double average = getAverage( grades[ student ] );
            System.out.printf( "%9.2f\n", average );
         } // end outer for
      } // end method outputGrades
   } // end class GradeBook
```

Outline

GradeBook.java

(7 of 7)

Outline

.java

(1 of 2)

Lines 10-19

```
 1 // Fig. 7.19: GradeBookTest.java
 2 // Creates GradeBook object using a two-dimensional array of grades.
 3
 4 public class GradeBookTest
 5 {
 6    // main method begins program execution
 7    public static void main( String args[] )
 8    {
 9       // two-dimensional array of student grades
10       int gradesArray[][] = { { 87, 96, 70 },
11                               { 68, 87, 90 },
12                               { 94, 100, 90 },
13                               { 100, 81, 82 },
14                               { 83, 65, 85 },
15                               { 78, 87, 65 },
16                               { 85, 75, 83 },
17                               { 91, 94, 100 },
18                               { 76, 72, 84 },
19                               { 87, 93, 73 } };
20
21       GradeBook myGradeBook = new GradeB
22          "CS101 Introduction to Java Pro
23       myGradeBook.displayMessage();
24       myGradeBook.processGrades();
25    } // end main
26 } // end class GradeBookTest
```

Declare `gradesArray` as 10-by-3 array

Each row represents a student; each column represents an exam grade

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

The grades are:

            Test 1  Test 2  Test 3  Average
Student  1     87      96      70    84.33
Student  2     68      87      90    81.67
Student  3     94     100      90    94.67
Student  4    100      81      82    87.67
Student  5     83      65      85    77.67
Student  6     78      87      65    76.67
Student  7     85      75      83    81.00
Student  8     91      94     100    95.00
Student  9     76      72      84    77.33
Student 10     87      93      73    84.33

Lowest grade in the grade book is 65
Highest grade in the grade book is 100

Overall grade distribution:
00-09:
10-19:
20-29:
```

```
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

Outline

GradeBookTest

.java

(2 of 2)

Program output

Variable-Length Argument Lists

- Variable-length argument lists
  - Unspecified number of arguments
  - Use ellipsis (...) in method’s parameter list
    - Can occur only once in parameter list
    - Must be placed at the end of parameter list
  - Array whose elements are all of the same type

```
1  // Fig. 7.20: VarargsTest.java
2  // Using variable-length argument lists.
3
4  public class VarargsTest
5  {
6     // calculate average
7     public static double average( double... numbers )
8     {
9        double total = 0.0; // initialize total
10
11       // calculate total using the enha
12       for ( double d : numbers )
13          total += d;
14
15       return total / numbers.length;
16    } // end method average
17
18    public static void main( String args[] )
19    {
20       double d1 = 10.0;
21       double d2 = 20.0;
22       double d3 = 30.0;
23       double d4 = 40.0;
24
```

Method `average` receives a variable length sequence of `double`s

Calculate the total of the `double`s in the array

Access `numbers.length` to obtain the size of the `numbers` array

Outline

VarargsTest

.java

(1 of 2)

Line 7

Lines 12-13

Line 15

```
25       System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4 = %.1f\n\n",
26          d1, d2, d3, d4 );
27
28       System.out.printf( "Average of d1 and d2 is %.1f\n",
29          average( d1, d2 ) );
30       System.out.printf( "Average of d1, d2 and d
31          average( d1, d2, d3 ) );
32       System.out.printf( "Average of d1, d2, d3 and d4 is %.1f\n",
33          average( d1, d2, d3, d4 ) );
34    } // end main
35 } // end class VarargsTest
```

Invoke method average with two arguments

Invoke method average with three arguments

Invoke method average with four arguments

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

Outline

VarargsTest

.java

2)

Line 29

Line 31

Line 33

Program output

Using Command-Line Arguments

- Command-line arguments
  - Pass arguments from the command line
    - `String args[]`
  - Appear after the class name in the `java` command
    - `java MyClass a b`
  - Number of arguments passed in from command line
    - `args.length`
  - First command-line argument
    - `args[ 0 ]`

```
 1 // Fig. 7.21: InitArray.java
 2 // Using command-line arguments to initialize an array.
 3
 4 public class InitArray
 5 {
 6    public static void main( String args[] )
 7    {
 8       // check number of command-line argument
 9       if ( args.length != 3 )
10          System.out.println(
11             "Error: Please re-enter the entire command, including\n" +
12             "an array size, initial value and increment." );
13       else
14       {
15          // get array size from first command-line argument
16          int arrayLength = Integer.parseInt( args[ 0 ] );
17          int array[] = new int[ arrayLength ]; // create array
18
19          // get initial value and increment from co
20          int initialValue = Integer.parseInt( args[ 1 ] );
21          int increment = Integer.parseInt( args[ 2 ] );
22
23          // calculate value for each array element
24          for ( int counter = 0; counter < array.length; counter++ )
25             array[ counter ] = initialValue + increment * counter;
26
27          System.out.printf( "%s%8s\n", "Index", "Value" );
28
```

Array `args` stores command-line arguments

Check number of arguments passed in from the command line

Obtain first command-line argument

Obtain second and third command-line arguments

Calculate the value for each array element based on command-line arguments

Outline

InitArray5.java

(1 of 2)

Line 6

Line 9

Line 16

Lines 20-21

Lines 24-25

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

```
java InitArray
Error: Please re-enter the entire command, including
an array size, initial value and increment.
```

Missing command-line arguments

```
java InitArray 5 0 4
Index   Value
    0       0
    1       4
    2       8
    3      12
    4      16
```

Three command-line arguments are 5, 0 and 4

```
java InitArray 10 1 2
Index   Value
    0       1
    1       3
    2       5
    3       7
    4       9
    5      11
    6      13
    7      15
    8      17
    9      19
```

Three command-line arguments are 10, 1 and 2

Outline

InitArray5.java

(2 of 2)

Program output

Exception Handling

Outline

Introduction

- Exception – an indication of a problem that occurs during a program's execution
- Exception handling – resolving exceptions that may occur so program can continue or terminate gracefully
- Exception handling enables programmers to create programs that are more robust and fault-tolerant

Introduction

- Examples
  - `ArrayIndexOutOfBoundsException` – an attempt is made to access an element past the end of an array
  - `NullPointerException` – when a `null` reference is used where an object is expected

Exception-Handling Overview

- Intermixing program logic with error-handling logic can make programs difficult to read, modify, maintain and debug
- Exception handling enables programmers to remove error-handling code from the “main line” of the program’s execution
- Improves clarity
- Enhances modifiability

Example: Divide By Zero Without Exception Handling

- Thrown exception – an exception that has occurred
- Stack trace
  - Name of the exception in a descriptive message that indicates the problem
  - Complete method-call stack
- `ArithmeticException` – can arise from a number of different problems in arithmetic
- Throw point – initial point at which the exception occurs, top row of call chain
- `InputMismatchException` – occurs when `Scanner` method `nextInt` receives a string that does not represent a valid integer

```java
// Fig. 13.1: DivideByZeroNoExceptionHandling.java
// An application that attempts to divide by zero.
import java.util.Scanner;

public class DivideByZeroNoExceptionHandling
{
   // demonstrates throwing an exception
   public static int quotient( int numer
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient

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

      System.out.print( "Please enter an integer numerator: " );
      int numerator = scanner.nextInt();
      System.out.print( "Please enter an integer
      int denominator = scanner.nextInt();

      int result = quotient( numerator, denominator );
      System.out.printf(
         "\nResult: %d / %d = %d\n", numerator, denominator, result );
   } // end main
} // end class DivideByZeroNoExceptionHandling
```

Attempt to divide; `denominator` may be zero

Read input; exception occurs if input is not a valid integer

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at
DivideByZeroNoExceptionHandling.quotient(DivideByZeroNoExceptionHandling.java:10)
        at
DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:22)
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at
DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:20)
```

Example: Handling ArithmeticExceptions and InputMismatchExceptions

- **With exception handling, the program catches and handles (i.e., deals with) the exception**
- **Next example allows user to try again if invalid input is entered (zero for denominator, or non-integer input)**

Enclosing Code in a `try` Block

- `try` block – encloses code that might throw an exception and the code that should not execute if an exception occurs
- Consists of keyword `try` followed by a block of code enclosed in curly braces

Catching Exceptions

- `catch` block – catches (i.e., receives) and handles an exception, contains:
  - Begins with keyword `catch`
  - Exception parameter in parentheses – exception parameter identifies the exception type and enables `catch` block to interact with caught exception object
  - Block of code in curly braces that executes when exception of proper type occurs
- Matching `catch` block – the type of the exception parameter matches the thrown exception type exactly or is a superclass of it
- Uncaught exception – an exception that occurs for which there are no matching `catch` blocks
  - Cause program to terminate if program has only one thread; Otherwise only current thread is terminated and there may be adverse effects to the rest of the program

Termination Model of Exception Handling

- When an exception occurs:
  - `try` block terminates immediately
  - Program control transfers to first matching `catch` block
- After exception is handled:
  - Termination model of exception handling – program control does not return to the throw point because the `try` block has expired; Flow of control proceeds to the first statement after the last `catch` block
  - ~~Resumption model of exception handling – program control resumes just after throw point~~
- `try` statement – consists of `try` block and corresponding `catch` and/or `finally` blocks

Using the `throws` Clause

- `throws` clause – specifies the exceptions a method may throws
  - Appears after method’s parameter list and before the method’s body
  - Contains a comma-separated list of exceptions
  - Exceptions can be thrown by statements in method’s body of by methods called in method’s body
  - Exceptions can be of types listed in `throws` clause or subclasses

```java
// Fig. 13.2: DivideByZeroWithExceptionHandling.java
// An exception-handling example that checks for divide-by-zero.
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
      throws ArithmeticException
   {
      return numerator / denominator; //
   } // end method quotient

   public static void main( String args[] )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input
      boolean continueLoop = true; // determines if more input is needed

      do
      {
         try // read two numbers and calculate quotient
         {
            System.out.print( "Please enter an integer numerator: " );
            int numerator = scanner.nextInt();
            System.out.print( "Please enter an integer denominato
            int denominator = scanner.nextInt();

```

`throws` clause specifies that method `quotient` may throw an `ArithmeticException`

Repetition statement loops until `try` block completes successfully

`try` block attempts to read input and perform division

Retrieve input; `InputMismatchException` thrown if input not valid integers

```
            int result = quotient( numerator, denominator );
            System.out.printf( "\nResult: %d / %d = %d\n", numerator,
               denominator, result );
            continueLoop = false; // input successful; end looping
         } // end try
         catch ( InputMismatchException inputMismatchException )
         {
            System.err.printf( "\nException: %s\n",
               inputMismatchException );
            scanner.nextLine(); // discard input so user can try again
            System.out.println(
               "You must enter integers. Please try again.\n" );
         } // end catch
         catch ( ArithmeticException arithmeticException )
         {
            System.err.printf( "\nException: %s\n", arithmeticException );
            System.out.println(
               "Zero is an invalid denominator. Please try again.\n" );
         } // end catch
      } while ( continueLoop ); // end do...while
   } // end main
} // end class DivideByZeroWithExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

When to Use Exception Handling

- **Exception handling designed to process synchronous errors**
- **Synchronous errors – occur when a statement executes**
- **Asynchronous errors – occur in parallel with and independent of the program's flow of control**

Java Exception Hierarchy

- All exceptions inherit either directly or indirectly from class `Exception`
- Exception classes form an inheritance hierarchy that can be extended
- Class `Throwable`, superclass of `Exception`
  - Only `Throwable` objects can be used with the exception-handling mechanism
  - Has two subclasses: `Exception` and `Error`
    - Class `Exception` and its subclasses represent exception situations that can occur in a Java program and that can be caught by the application
    - Class `Error` and its subclasses represent abnormal situations that could happen in the JVM – it is usually not possible for a program to recover from `Error`s

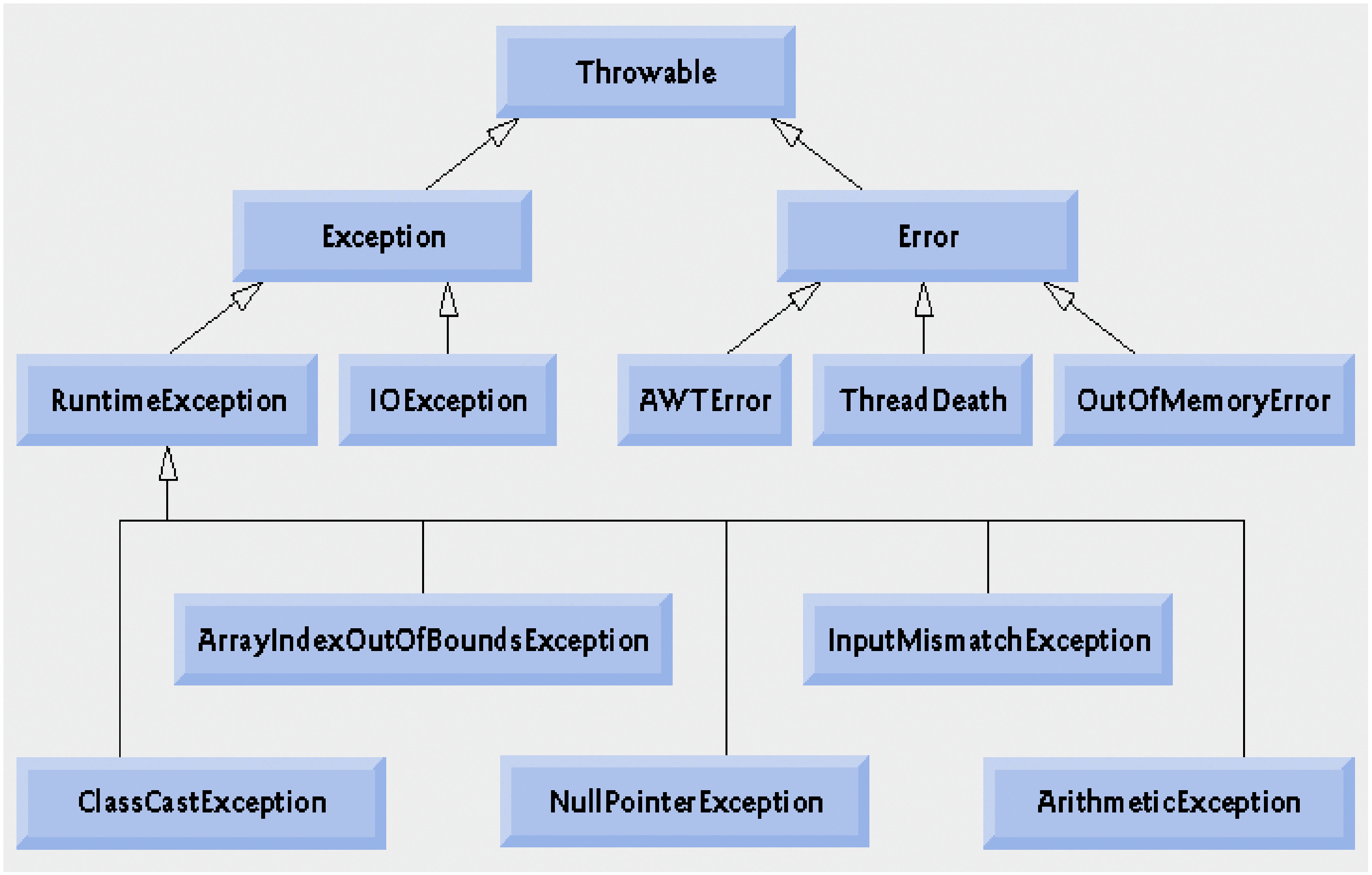

Fig. 13.3 | Portion of class `Throwable`'s inheritance hierarchy.

Java Exception Hierarchy

- Two categories of exceptions: checked and unchecked
- Checked exceptions
  - Exceptions that inherit from class `Exception` but not from `RuntimeException`
  - Compiler enforces a catch-or-declare requirement
  - Compiler checks each method call and method declaration to determine whether the method `throws` checked exceptions. If so, the compiler ensures that the checked exception is caught or is declared in a `throws` clause. If not caught or declared, compiler error occurs.
- Unchecked exceptions
  - Inherit from class `RuntimeException` or class `Error`
  - Compiler does not check code to see if exception is caught or declared
  - If an unchecked exception occurs and is not caught, the program terminates or runs with unexpected results
  - Can typically be prevented by proper coding

Java Exception Hierarchy

- `catch` block catches all exceptions of its type and subclasses of its type
- If there are multiple `catch` blocks that match a particular exception type, only the first matching `catch` block executes
- It makes sense to use a `catch` block of a superclass when all the `catch` blocks for that class's subclasses will perform the same functionality

`finally` block

- **Programs that obtain certain resources must return them explicitly to avoid resource leaks**
- **`finally` block**
  - **Consists of `finally` keyword followed by a block of code enclosed in curly braces**
  - **Optional in a `try` statement**
  - **If present, is placed after the last `catch` block**
  - **Executes whether or not an exception is thrown in the corresponding `try` block or any of its corresponding `catch` blocks**
  - **Will not execute if the application exits early from a `try` block via method `System.exit`**
  - **Typically contains resource-release code**

```
try
{
    statements
    resource-acquisition statements
} // end try
catch ( AKindOfException  exception1 )
{
    exception-handling statements
} // end catch
.
.
.
catch ( AnotherKindOfException  exception2 )
{
    exception-handling statements
} // end catch
finally
{
    statements
    resource-release statements
} // end finally
```

Fig. 13.4 | **Position of the finally block after the last catch block in a try statement.**

`finally` block

- If no exception occurs, `catch` blocks are skipped and control proceeds to `finally` block.
- After the `finally` block executes control proceeds to first statement after the `finally` block.
- If exception occurs in the `try` block, program skips rest of the `try` block. First matching the `catch` block executes and control proceeds to the `finally` block. If exception occurs and there are no matching `catch` blocks, control proceeds to the `finally` block. After the `finally` block executes, the program passes the exception to the next outer the `try` block.
- If `catch` block throws an exception, the `finally` block still executes.

```
// Fig. 13.5: UsingExceptions.java
// Demonstration of the try...catch...finally exception handling
// mechanism.

public class UsingExceptions
{
   public static void main( String args[] )
   {
      try
      {
         throwException(); // call method throwException
      } // end try
      catch ( Exception exception )
      {
         System.err.println( "Exception handled in main" );
      } // end catch

      doesNotThrowException();
   } // end main

```

Call method that throws an exception

```
   // demonstrate try...catch...finally
   public static void throwException() throws Exception
   {
      try // throw an exception and immediately catch it
      {
         System.out.println( "Method throwException" );
         throw new Exception(); // generate exception
      } // end try
      catch ( Exception exception ) // ca
      {
         System.err.println(
            "Exception handled in method throwException" );
         throw exception; // rethrow for further processing

         // any code here would not be

      } // end catch
      finally // executes regardless of what occurs in try...catch
      {
         System.err.println
      } // end finally

      // any code here would not be reached, exception rethrown in catch

```

Create new Exception and throw it

Throw previously created Exception

finally block executes even though exception is rethrown in catch block

```
   } // end method throwException

   // demonstrate finally when no exception occurs
   public static void doesNotThrowException()
   {
      try // try block does not throw an exception
      {
         System.out.println( "Method doesNotThrowException" );
      } // end try
      catch ( Exception exception ) // does not execute
      {
         System.err.println( exception );
      } // end catch
      finally // executes regardless of what occurs in try...catch
      {
         System.err.println(
            "Finally execute
      } // end finally

      System.out.println( "End of method doesNotThrowException" );
   } // end method doesNotThrowException
} // end class UsingExceptions
```

`finally` block executes even though no exception is thrown

```
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException
```

Throwing Exceptions Using the `throw` Statement

- `throw` statement – used to throw exceptions
- Programmers can thrown exceptions themselves from a method if something has gone wrong
- `throw` statement consists of keyword `throw` followed by the exception object

Stack Unwinding

- Stack unwinding – When an exception is thrown but not caught in a particular scope, the method-call stack is "unwound," and an attempt is made to catch the exception in the next outer `try` block.
- When unwinding occurs:
  - The method in which the exception was not caught terminates
  - All local variables in that method go out of scope
  - Control returns to the statement that originally invoked the method – if a try block encloses the method call, an attempt is made to catch the exception.

```
// Fig. 13.6: UsingExceptions.java
// Demonstration of stack unwinding.

public class UsingExceptions
{
   public static void main( String args[] )
   {
      try // call throwExce
      {
         throwException();
      } // end try
      catch ( Exception exception ) // exception thrown in throwException
      {
         System.err.println( "Except
      } // end catch
   } // end main

```

Call method that throws an exception

Catch exception that may occur in the above `try` block, including the call to method `throwException`

```java
   // throwException throws exception that is not caught in this method
   public static void throwException() throws Exception
   {
      try // throw an exception and catch it in main
      {
         System.out.println( "Method throwException" );
         throw new Exception(); // generate exception
      } // end try
      catch ( RuntimeException runtimeEx
      {
         System.err.println(
            "Exception handled in method throwException" );
      } // end catch
      finally // fi
      {
         System.err.println( "Finally is always executed" );
      } // end finally
   } // end method throwException
} // end class UsingExceptions
```

Method throws exception

Throw new exception; Exception not caught in current `try` block, so handled in outer `try` block

`finally` block executes before control returns to outer `try` block

```
Method throwException
Finally is always executed
Exception handled in main
```

printStackTrace, getStackTrace and getMessage

- Methods in class `Throwable` retrieve more information about an exception
  - `printStackTrace` – outputs stack trace to standard error stream
  - `getStackTrace` – retrieves stack trace information as an array of `StackTraceElement` objects; enables custom processing of the exception information
  - `getMessage` – returns the descriptive string stored in an exception

```java
// Fig. 13.7: UsingExceptions.java
// Demonstrating getMessage and printStackTrace from class Exception.

public class UsingExceptions
{
   public static void main(
   {
      try
      {
         method1(); // call method1
      } // end try
      catch ( Exception exception ) // catch exception thro
      {
         System.err.printf( "%s\n\n", exception.getMessage() );
         exception.printStackTrace(); // print exception stack t

         // obtain the stack-trace information
         StackTraceElement[] traceElements = exception.getStackTrace();

```

Call to `method1`, `method1` calls `method2`, `method2` calls `method3` and `method3` throws a new `Exception`

Display descriptive string of exception thrown in `method3`

Retrieve stack information as an array of `StackTraceElement` objects

Display stack trace for exception thrown in `method3`

```
      System.out.println( "\nStack trace from getStackTrace:" );
      System.out.println( "Class\t\tFile\t\t\tLine\tMethod" );

      // loop through traceElements to get exception description
      for ( StackTraceElement element : traceElements )
      {
         System.out.printf( "%s\t", element.getClassName() );
         System.out.printf( "%s\t", element.getFileName() );
         System.out.printf( "%s\t", element.getLineNumber() );
         System.out.printf( "%s\n", element.getMethodName() );
      } // end for
   } // end catch
} // end main

// call method2; throw exceptions back to main
public static void method1() throws Exception
{
   method2();
} // end method method1

```

Retrieve class name for current `StackTraceElement`

Retrieve file name for current `StackTraceElement`

Retrieve line number for current `StackTraceElement`

Retrieve method name for current `StackTraceElement`

`method1` calls `method2`, `method2` calls `method3` and `method3` throws an `Exception`

```java
    // call method3; throw exceptions back to method1
    public static void method2() throws Exception
    {
       method3();
    } // end method method2

    // throw Exception back to method2
    public static void method3() throws Exception
    {
       throw new Exception( "Exception thrown in method3" );
    } // end method method3
 } // end class UsingExceptions
```

method2 calls method3, which throws an Exception

Exception created and thrown

```
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
        at UsingExceptions.method3(UsingExceptions.java:49)
        at UsingExceptions.method2(UsingExceptions.java:43)
        at UsingExceptions.method1(UsingExceptions.java:37)
        at UsingExceptions.main(UsingExceptions.java:10)

Stack trace from getStackTrace:
Class           File                    Line    Method
UsingExceptions UsingExceptions.java    49      method3
UsingExceptions UsingExceptions.java    43      method2
UsingExceptions UsingExceptions.java    37      method1
UsingExceptions UsingExceptions.java    10      main
```

Software Engineering Observation 13.12

Never ignore an exception you catch. At least use `printStackTrace` to output an error message. This will inform users that a problem exists, so that they can take appropriate actions.

Declaring New Exception Types

- You can declare your own exception classes that are specific to the problems that can occur when another program uses your reusable classes
- New exception class must extend an existing exception class
- Typically contains only two constructors
  - One takes no arguments, passes a default exception messages to the superclass constructor
  - One that receives a customized exception message as a string and passes it to the superclass constructor

Assertions

- **Assertions are conditions that should be true at a particular point in a method**
- **Help ensure a program's validity by catching potential bugs**
- **Preconditions and Postconditions are two kinds of assertions**
- **Assertions can be stated as comments or assertions can be validated programmatically using the `assert` statement**

Assertions

- `assert` statement
  - Evaluates a `boolean` expression and determines whether it is `true` or `false`
  - Two forms
    - `assert` *expression*`;` -- `AssertionError` is thrown if *expression* is `false`
    - `assert` *expression1* `:` *expression2*`;` -- `AssertionError` is thrown if *expression1* is `false`, *expression2* is error message
  - Used to verify intermediate states to ensure code is working correctly
  - Used to implement preconditions and postconditions programmatically
- By default, assertions are disabled
- Assertions can be enabled with the `–ea` command-line option

```
 1  // Fig. 13.9: AssertTest.java
 2  // Demonstrates the assert statement
 3  import java.util.Scanner;
 4
 5  public class AssertTest
 6  {
 7     public static void main( String args[] )
 8     {
 9        Scanner input = new Scanner( System.in );
10
11        System.out.print( "Enter a number between 0 and 10: " );
12        int number =
13
14        // assert that the absolute value is >= 0
15        assert ( number >= 0 && number <= 10 ) : "bad number: " + number;
16
17        System.out.printf( "You e
18     } // end main
19  } // end class AssertTest
```

`assert` statement

Message to be displayed with `AssertionError`

If `number` is less than 0 or greater than 10, `AssertionError` occurs

```
Enter a number between 0 and 10: 5
You entered 5
```

```
Enter a number between 0 and 10: 50
Exception in thread "main" java.lang.AssertionError: bad number: 50
        at AssertTest.main(AssertTest.java:15)
```