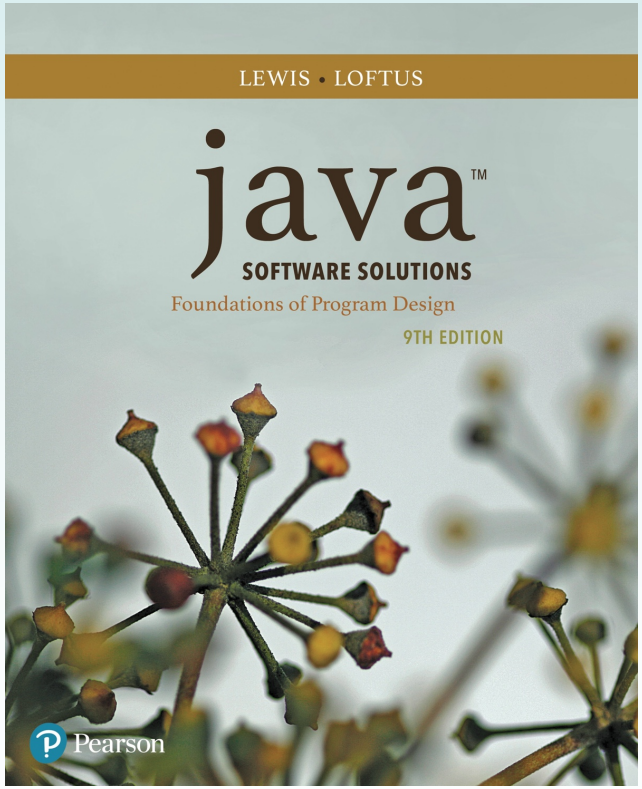


Java Software Solutions

Foundations of Program Design

9th Edition

John Lewis
William Loftus

Focus of the Course

- Object-Oriented Software Development
  - problem solving
  - program design, implementation, and testing
  - object-oriented concepts
    - classes
    - objects
    - encapsulation
    - inheritance
    - polymorphism
  - graphical user interfaces
  - the Java programming language

Java Program Structure

- In the Java programming language:
  - A program is made up of one or more *classes*
  - A class contains one or more *methods*
  - A method contains program *statements*
- These terms will be explored in detail throughout the course
- A Java application always contains a method called `main`
- See `Lincoln.java`

Java Program Structure

```
//  comments about the class
public class MyProgram
{
   //  comments about the method
   public static void main (String[] args)
   {

   }

}
```

method body

method header

Comments

- Comments should be included to explain the purpose of the program and describe processing steps
- They do not affect how a program works
- Java comments can take three forms:

```
// this comment runs to the end of the line

/*  this comment runs to the terminating
    symbol, even across line breaks        */

/** this is a javadoc comment   */
```

Identifiers

- *Identifiers* are the "words" in a program
- A Java identifier can be made up of letters, digits, the underscore character (_), and the dollar sign
- Identifiers cannot begin with a digit
- Java is *case sensitive*: `Total`, `total`, and `TOTAL` are different identifiers
- By convention, programmers use different case styles for different types of identifiers, such as
  - *title case* for class names - `Lincoln`
  - *upper case* for constants - `MAXIMUM`

Errors

- A program can have three types of errors
- The compiler will find syntax errors and other basic problems (*compile-time errors*)
  - If compile-time errors exist, an executable version of the program is not created
- A problem can occur during program execution, such as trying to divide by zero, which causes a program to terminate abnormally (*run-time errors*)
- A program may run, but produce incorrect results, perhaps using an incorrect formula (*logical errors*)

Problem Solving

- The purpose of writing a program is to solve a problem
- Solving a problem consists of multiple activities:
  - Understand the problem
  - Design a solution
  - Consider alternatives and refine the solution
  - Implement the solution
  - Test the solution
- These activities are not purely linear – they overlap and interact

Objects

- An object has:
  - *state* - descriptive characteristics
  - *behaviors* - what it can do (or what can be done to it)
- The state of a bank account includes its account number and its current balance
- The behaviors associated with a bank account include the ability to make deposits and withdrawals
- Note that the behavior of an object might change its state

Classes

- An object is defined by *a class*
- A class is the blueprint of an object
- The class uses methods to define the behaviors of the object
- The class that contains the main method of a Java program represents the entire program
- A class represents a concept, and an object represents the embodiment of that concept
- Multiple objects can be created from the same class

Summary

- Chapter 1 focused on:
  - components of a computer
  - how those components interact
  - how computers store and manipulate information
  - computer networks
  - the Internet and the World Wide Web
  - programming and programming languages
  - an introduction to Java
  - an overview of object-oriented concepts

Data and Expressions

- Let's explore some other fundamental programming concepts
- Chapter 2 focuses on:
  - character strings
  - primitive data
  - the declaration and use of variables
  - expressions and operator precedence
  - data conversions
  - accepting input from the user

Character Strings

- A *string literal* is represented by putting double quotes around the text
- Examples:

```
"This is a string literal."
"123 Main Street"
"X"
```

- Every character string is an object in Java, defined by the `String` class
- Every string literal represents a `String` object

Variables

- A *variable* is a name for a location in memory that holds a value
- A *variable declaration* specifies the variable's name and the type of information that it will hold


```
int total;
int count, temp, result;
```

Multiple variables can be created in one declaration

Assignment

- An *assignment statement* changes the value of a variable
- The assignment operator is the = sign

```
total = 55;
```

- The value that was in `total` is overwritten
- You can only assign a value to a variable that is consistent with the variable's declared type
- See `Geometry.java`

Primitive Data

- There are eight primitive data types in Java
- Four of them represent integers:
  - `byte`, `short`, `int`, `long`
- Two of them represent floating point numbers:
  - `float`, `double`
- One of them represents characters:
  - `char`
- And one of them represents boolean values:
  - `boolean`

Expressions

- An *expression* is a combination of one or more operators and operands
- *Arithmetic expressions* compute numeric results and make use of the arithmetic operators:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Remainder	%

- If either or both operands are floating point values, then the result is a floating point value

Data Conversion

- *Widening conversions* are safest because they tend to go from a small data type to a larger one (such as a `short` to an `int`)
- *Narrowing conversions* can lose information because they tend to go from a large data type to a smaller one (such as an `int` to a `short`)
- In Java, data conversions can occur in three ways:
  - assignment conversion
  - promotion
  - casting

Reading Input

- The `Scanner` class is part of the `java.util` class library, and must be imported into a program to be used
- The `nextLine` method reads all of the input until the end of the line is found
- See `Echo.java`
- The details of object creation and class libraries are discussed further in Chapter 3

Using Classes and Objects

- We can create more interesting programs using predefined classes and related objects
- Chapter 3 focuses on:
  - object creation and object references
  - the `String` class and its methods
  - the Java API class library
  - the `Random` and `Math` classes
  - formatting output
  - enumerated types
  - wrapper classes
  - JavaFX graphics API
  - shape classes

Creating Objects

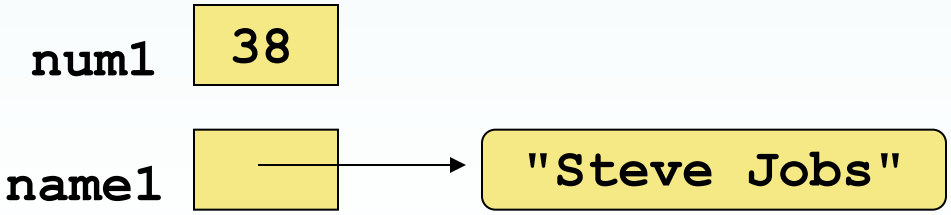
- A variable holds either a primitive value or a *reference* to an object
- A class name can be used as a type to declare an *object reference variable*

```
String title;
```

- No object is created with this declaration
- An object reference variable holds the address of an object
- The object itself must be created separately

References

- Note that a primitive variable contains the value itself, but an object variable contains the address of the object
- An object reference can be thought of as a pointer to the location of the object
- Rather than dealing with arbitrary addresses, we often depict a reference graphically



The String Class

- Because strings are so common, we don't have to use the `new` operator to create a `String` object

```
title = "Java Software Solutions";
```

- This is special syntax that works only for strings
- Each string literal (enclosed in double quotes) represents a `String` object

String Methods

- Once a `String` object has been created, neither its value nor its length can be changed
- Therefore we say that an object of the `String` class is *immutable*
- However, several methods of the `String` class return new `String` objects that are modified versions of the original

The Random Class

- The `Random` class is part of the `java.util` package
- It provides methods that generate pseudorandom numbers
- A `Random` object performs complicated calculations based on a *seed value* to produce a stream of seemingly random values
- See `RandomNumbers.java`

The Math Class

- The `Math` class is part of the `java.lang` package
- The `Math` class contains methods that perform various mathematical functions
- These include:
  - absolute value
  - square root
  - exponentiation
  - trigonometric functions

Enumerated Types

- Java allows you to define an *enumerated type*, which can then be used to declare variables
- An enumerated type declaration lists all possible values for a variable of that type
- The values are identifiers of your own choosing
- The following declaration creates an enumerated type called `Season`

```
enum Season {winter, spring, summer, fall};
```

- Any number of values can be listed

Enumerated Types

- The declaration of an enumerated type is a special type of class, and each variable of that type is an object
- The `ordinal` method returns the ordinal value of the object
- The `name` method returns the name of the identifier corresponding to the object's value
- See `IceCream.java`

Wrapper Classes

- The `java.lang` package contains *wrapper classes* that correspond to each primitive type:

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Autoboxing

- *Autoboxing* is the automatic conversion of a primitive value to a corresponding wrapper object:

```
Integer obj;
int num = 42;
obj = num;
```

- The assignment creates the appropriate `Integer` object
- The reverse conversion (called *unboxing*) also occurs automatically as needed

Intro to JavaFX

- JavaFX programs extend the `Application` class, inheriting core graphical functionality
- A JavaFX program has a `start` method
- The `main` method is only needed to launch the JavaFX application
- The `start` method accepts the primary stage (window) used by the program as a parameter
- JavaFX embraces a theatre analogy
- See `HelloJavaFX.java`

Writing Classes

- We've been using predefined classes from the Java API. Now we will learn to write our own classes.
- Chapter 4 focuses on:
  - class definitions
  - instance data
  - encapsulation and Java modifiers
  - method declaration and parameter passing
  - constructors
  - arcs and images
  - events and event handlers
  - buttons and text fields

The Die Class

- The `Die` class contains two data values
  - a constant `MAX` that represents the maximum face value
  - an integer `faceValue` that represents the current face value
- The `roll` method uses the `random` method of the `Math` class to determine a new face value
- There are also methods to explicitly set and retrieve the current face value at any time

Constructors

- As mentioned previously, a *constructor* is used to set up an object when it is initially created
- A constructor has the same name as the class
- The `Die` constructor is used to set the initial face value of each new die object to one
- We examine constructors in more detail later in this chapter

Instance Data

- A variable declared at the class level (such as `faceValue`) is called *instance data*
- Each instance (object) has its own instance variable
- A class declares the type of the data, but it does not reserve memory space for it
- Each time a `Die` object is created, a new `faceValue` variable is created as well
- The objects of a class share the method definitions, but each object has its own data space
- That's the only way two objects can have different states

Encapsulation

- There are two views of an object:
  - internal - the details of the variables and methods of the class that defines it
  - external - the services that an object provides and how the object interacts with the rest of the system
- From the external view, an object is an *encapsulated* entity, providing a set of specific services
- These services define the *interface* to the object

Visibility Modifiers

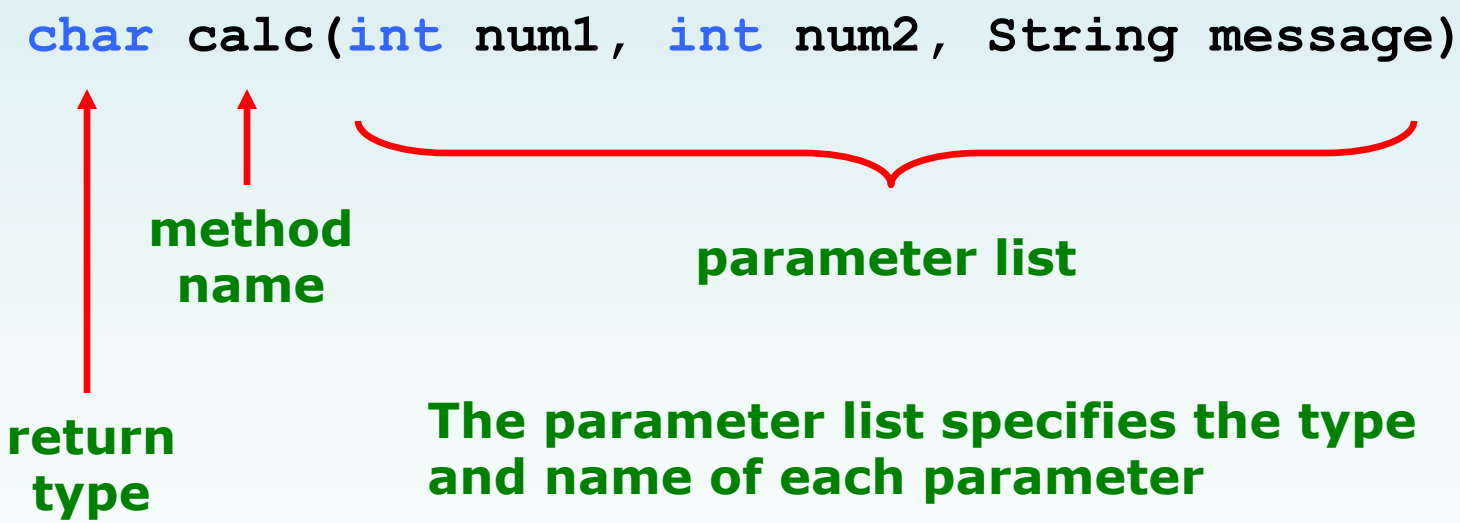
	`public`	`private`
Variables	**Violate encapsulation**	**Enforce encapsulation**
Methods	**Provide services to clients**	**Support other methods in the class**

Accessors and Mutators

- Because instance data is private, a class usually provides services to access and modify data values
- An *accessor method* returns the current value of a variable
- A *mutator method* changes the value of a variable
- The names of accessor and mutator methods take the form `getX` and `setX`, respectively, where `X` is the name of the value
- They are sometimes called “getters” and “setters”

Method Header

- A method declaration begins with a *method header*



The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal parameter*

Method Body

- The method header is followed by the *method body*

```
char calc(int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt(sum);

    return result;
}
```

The return expression must be consistent with the return type

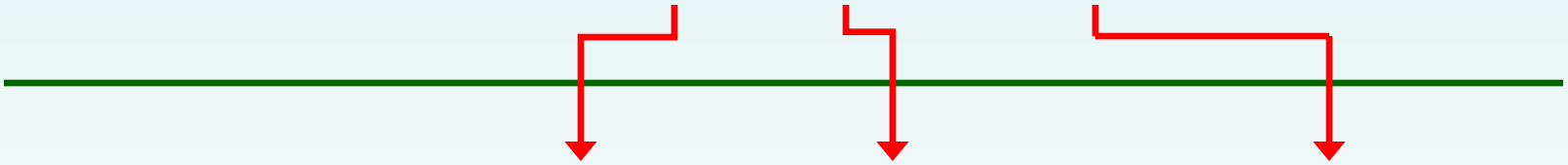
`sum` and `result` are local data

They are created each time the method is called, and are destroyed when it finishes executing

Parameters

- When a method is called, the *actual parameters* in the invocation are copied into the *formal parameters* in the method header

```
ch = obj.calc(25, count, "Hello");
```



```
char calc(int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt(sum);

    return result;
}
```

Conditionals and Loops

- Now we will examine programming statements that allow us to:
  - make decisions
  - repeat processing steps in a loop
- Chapter 5 focuses on:
  - boolean expressions
  - the if and if-else statements
  - comparing data
  - while loops
  - iterators
  - the `ArrayList` class
  - more GUI controls

Conditional Statements

- A *conditional statement* lets us choose which statement will be executed next
- They are sometimes called *selection statements*
- Conditional statements give us the power to make basic decisions
- The Java conditional statements are the:
  - `if` and `if-else` statement
  - `switch` statement
- We'll explore the `switch` statement in Chapter 6

Boolean Expressions

- A condition often uses one of Java's *equality operators* or *relational operators*, which all return boolean results:

`==`	equal to
`!=`	not equal to
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

- Note the difference between the equality operator (==) and the assignment operator (=)

Logical Operators

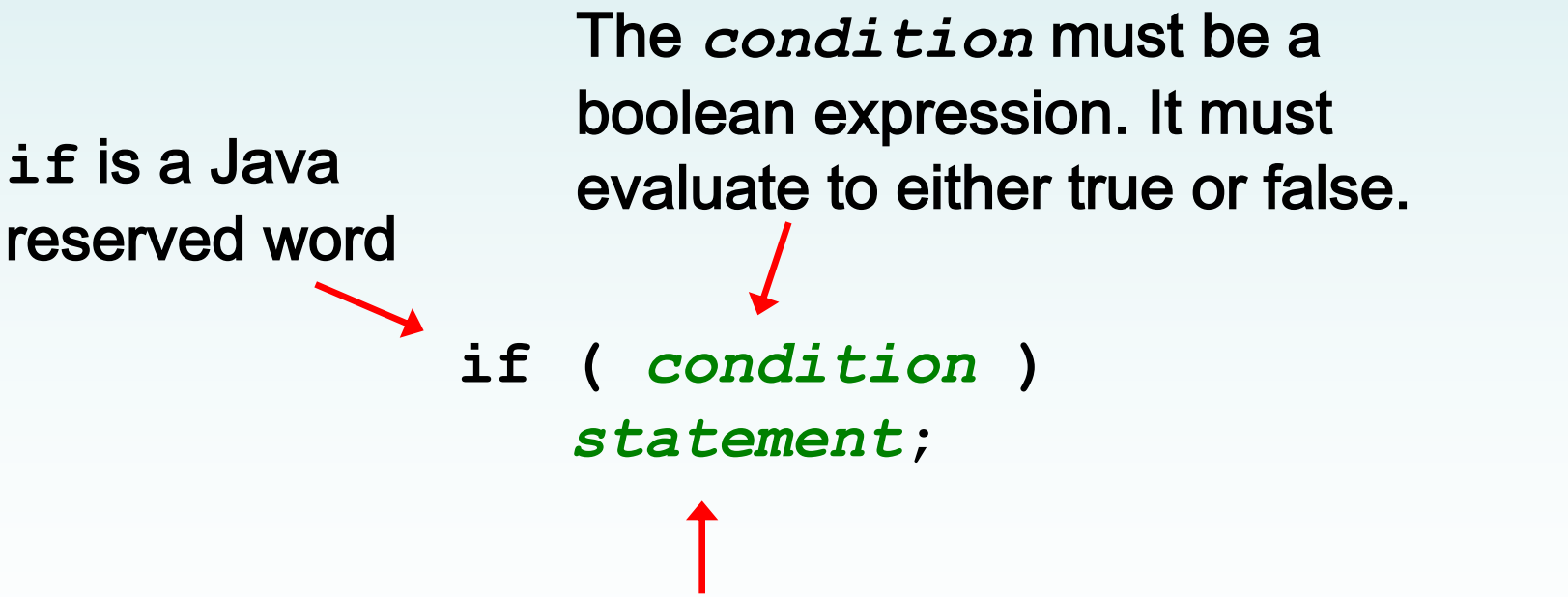
- Boolean expressions can also use the following *logical operators*:

`!`	Logical NOT
`&&`	Logical AND
`||`	Logical OR

- They all take boolean operands and produce boolean results
- Logical NOT is a unary operator (it operates on one operand)
- Logical AND and logical OR are binary operators (each operates on two operands)

The if Statement

- Let's now look at the `if` statement in more detail
- The *if statement* has the following syntax:



The if-else Statement

- An *else clause* can be added to an `if` statement to make an *if-else statement*

```
if ( condition )
   statement1;
else
   statement2;
```

- If the *condition* is true, *statement1* is executed; if the condition is false, *statement2* is executed
- One or the other will be executed, but not both
- See `Wages.java`

Block Statements

- Several statements can be grouped together into a *block statement* delimited by braces
- A block statement can be used wherever a statement is called for in the Java syntax rules

```
if (total > MAX)
{
   System.out.println("Error!!");
   errorCount++;
}
```

Comparing Data

- When comparing data using boolean expressions, it's important to understand the nuances of certain data types
- Let's examine some key situations:
  - Comparing floating point values for equality
  - Comparing characters
  - Comparing strings (alphabetical order)
  - Comparing object vs. comparing object references

Comparing Objects

- The == operator can be applied to objects – it returns true if the two references are aliases of each other
- The `equals` method is defined for all objects, but unless we redefine it when we write a class, it has the same semantics as the == operator
- It has been redefined in the `String` class to compare the characters in the two strings
- When you write a class, you can redefine the `equals` method to return true under whatever conditions are appropriate

Repetition Statements

- *Repetition statements* allow us to execute a statement multiple times
- Often they are referred to as *loops*
- Like conditional statements, they are controlled by boolean expressions
- Java has three kinds of repetition statements: `while`, `do`, and `for` loops
- The `do` and `for` loops are discussed in Chapter 6

The while Statement

- A *while statement* has the following syntax:

```
while ( condition )
    statement;
```

- If the **condition** is true, the **statement** is executed
- Then the condition is evaluated again, and if it is still true, the statement is executed again
- The statement is executed repeatedly until the condition becomes false

Iterators

- An *iterator* is an object that allows you to process a collection of items one at a time
- It lets you step through each item in turn and process it as needed
- An iterator has a `hasNext` method that returns true if there is at least one more item to process
- The `next` method returns the next item
- Iterator objects are defined using the `Iterator` interface, which is discussed further in Chapter 7

The ArrayList Class

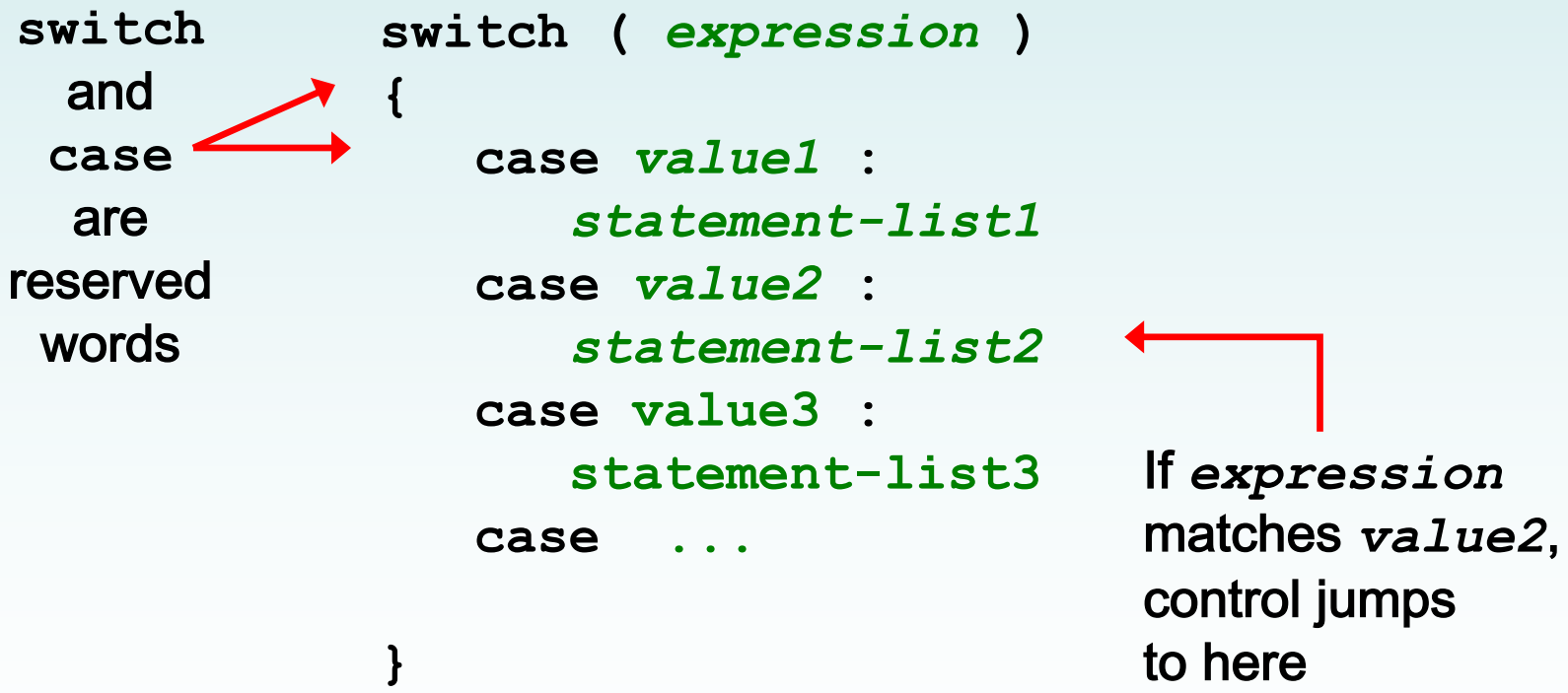
- An `ArrayList` object stores a list of objects, and is often processed using a loop
- The `ArrayList` class is part of the `java.util` package
- You can reference each object in the list using a numeric index
- An `ArrayList` object grows and shrinks as needed, adjusting its capacity as necessary

More Conditionals and Loops

- Now we can fill in some additional details regarding Java conditional and repetition statements
- Chapter 6 focuses on:
  - the `switch` statement
  - the conditional operator
  - the `do` loop
  - the `for` loop
  - using conditionals and loops with graphics
  - graphic transformations

The switch Statement

- The general syntax of a `switch` statement is:



The Conditional Operator

- The *conditional operator* evaluates to one of two expressions based on a boolean condition
- Its syntax is:

```
condition ? expression1 : expression2
```

- If the ***condition*** is true, ***expression1*** is evaluated; if it is false, ***expression2*** is evaluated
- The value of the entire conditional operator is the value of the selected expression

The do Statement

- A *do statement* has the following syntax:

```
do
{
    statement-list;
}
while (condition);
```

- The **`statement-list`** is executed once initially, and then the **`condition`** is evaluated
- The statement is executed repeatedly until the condition becomes false

The for Statement

- A *for statement* has the following syntax:

The *initialization* is executed once before the loop begins

The *statement* is executed until the *condition* becomes false

```
for ( initialization ; condition ; increment )
   statement;
```

The *increment* portion is executed at the end of each iteration

For-each Loops

- A variant of the `for` loop simplifies the repetitive processing of items in an iterator
- For example, suppose `bookList` is an `ArrayList<Book>` object
- The following loop will print each book:

```
for (Book myBook : bookList)
   System.out.println(myBook);
```

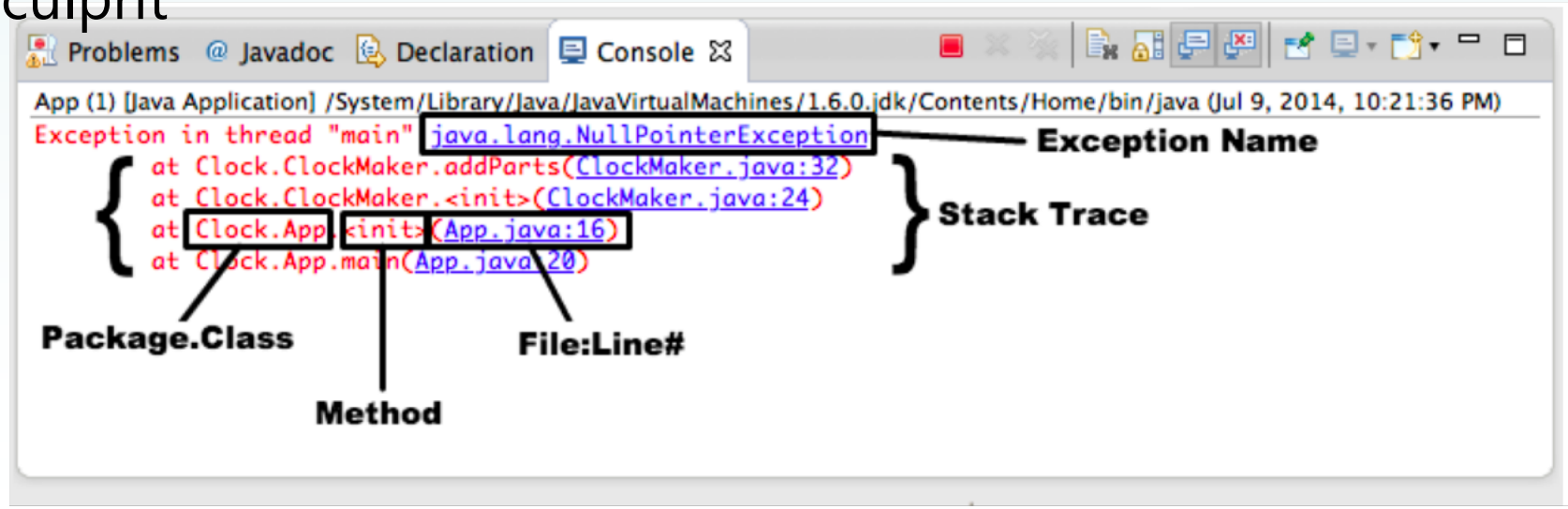
- This version of a for loop is often called a *for-each loop*

Debugging

- **Benefits** of using a source code debugger
- **Basic services** of a source code debugger
- **Terminology** used with source code debuggers
- How to **step** **through** a program
- When and how to use a **breakpoint**
- **Look at variables** as a program is executing
- **Look at the call stack** for a program

Traceback AKA Stack Trace

- In Java, a runtime error results in a traceback
- Traceback tells where the exception occurred
- Each line gives the statement that called the former line
- Look for code that you wrote to find the error
- Often the first line that references your code is the culprit



Stepping through code

- After you stop (at a breakpoint), how do you continue?
- Some common definitions:

Run	Start executing from current point. • run until program ends or a breakpoint is reached.
Step Into	If next instruction is a function call, step into it. If next instruction is not a function, just execute it.
Step Over	If next instruction is a function call, step over it. If next instruction is not a function, just execute it.
Step out of	Step out of the current function. Execute until the end of the current function, stop after caller line.

More on the Call Stack

- A *stack* is a data structure that has 2 operations
  - Push – put something on top
  - Pop – take something off of the top
- Last data stored is first data out (called LIFO)
- When a program calls a method the current position data is pushed onto a call stack
  - Allows program flow to return to the caller when call is done
- Debug mode lets us see the call stack.
- Call stack is typically displayed as separate window/tab.

Testing

- Code is not done until it is tested
- Test units first, then integrate with other units
  - Units ≈ Classes in Java, or possibly methods
- Testing looks for problems
  - Debugging looks for problem causes
- How has it been done?
  - Write test driver class
    - What we have so far
  - Use testing framework like JUnit, TestNG
    - Less work, more features

Levels of Testing

- From low to high
  - Unit test – usually done by developer
  - Integration tests
  - System tests
  - Acceptance tests
- Lower levels both structural and functional
  - Focus on finding bugs
- Higher levels primarily functional
  - Focus on end-to-end issues

Unit Testing Focus

- The structure of the unit should be tested
  - Every line of code been executed by some test
  - Every branch is taken in some test
    - Every Boolean condition has a test where it is true and another where it is false
    - if, if-else, while, do, for, switch statements and conditionals
- Unit's externally observable behaviour should be tested
  - does it give proper output for all inputs?
  - does it handle error cases correctly?
  - Goes in addition to code coverage (structure)

Core JUnit Concepts

- Test class
  - Public with zero-argument constructor
  - Usually have a test class for each production class
  - Each test method runs on a new instance
    - Cannot share instance variables across methods
- Test suite
  - Groups test classes and test suites
- Test runner
  - Controls how tests are run

Test Methods

- Annotated with org.junit.Test
- Public, no arguments, void return
- Uses assert methods from org.junit.Assert class
  - `assertArrayEquals("message", A, B)`
  - `assertEquals("message", A, B)`
  - `assertSame("message", A, B) – same object`
  - `assertTrue("message", A)`
  - `assertNotNull("message", A)`
    - Parameter A is expected value
    - Parameter B is actual value

Code Coverage Example

```
26    public Rational(int numer, int denom) {
27        if (denom == 0) {
28            denominator = 1;
29            numerator = numer;
30        } else if (denom < 0) {
31            // Make the numerator "store" the sign
32            numerator = numer * -1;
33            denominator = denom * -1;
34        } else {
35            numerator = numer;
36            denominator = denom;
37        }
38        reduce();
39    }
```

Object-Oriented Design

- Now we can extend our discussion of the design of classes and objects
- Chapter 7 focuses on:
  - software development activities
  - the relationships that can exist among classes
  - the static modifier
  - writing interfaces
  - the design of enumerated type classes
  - method design and method overloading
  - GUI design
  - mouse and keyboard events

Program Development

- The creation of software involves four basic activities:
  - establishing the requirements
  - creating a design
  - implementing the code
  - testing the implementation
- These activities are not strictly linear – they overlap and interact

Identifying Classes and Objects

- Remember that a class represents a group (classification) of objects with the same behaviors
- Generally, classes that represent objects should be given names that are singular nouns
- Examples: `Coin`, `Student`, `Message`
- A class represents the concept of one such object
- We are free to instantiate as many of each object as needed

Static Class Members

- Recall that a static method is one that can be invoked through its class name
- For example, the methods of the `Math` class are static:

```
result = Math.sqrt(25)
```

- Variables can be static as well
- Determining if a method or variable should be static is an important design decision

Class Relationships

- Classes in a software system can have various types of relationships to each other
- Three of the most common relationships:
  - Dependency: A *uses* B
  - Aggregation: A *has-a* B
  - Inheritance: A *is-a* B
- Let's discuss dependency and aggregation further
- Inheritance is discussed in detail in Chapter 9

Validating Parameters

- Often methods have restrictions on parameter values
  - Integer.parseInt(String arg) requires the argument to have the form of an integer
  - Math.sqrt(double x) returns the positive square root of a double x which should be positive
- If the parameter is not valid, handle it properly
  - Integer.parseInt throws NumberFormatException if the argument is not in the form of an integer
  - Math.sqrt returns NaN if the argument < zero

Interfaces

None of the methods in an interface are given a definition (body)

`interface` is a reserved word

```
public interface Doable
{
   public void doThis();
   public int doThat();
   public void doThis2(double value, char ch);
   public boolean doTheOther(int num);
}
```

A semicolon immediately follows each method header

The Comparable Interface

- Any class can implement `Comparable` to provide a mechanism for comparing objects of that type

```
if (obj1.compareTo(obj2) < 0)
   System.out.println ("obj1 is less than obj2");
```

- The value returned from `compareTo` should be negative is `obj1` is less that `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`

- It's up to the programmer to determine what makes one object less than another

Objects as Parameters

- Another important issue related to method design involves parameter passing
- Parameters in a Java method are *passed by value*
- A copy of the *actual parameter* (the value passed in) is stored into the *formal parameter* (in the method header)
- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other

Method Overloading

- Let's look at one more important method design issue: method overloading
- *Method overloading* is the process of giving a single method name multiple definitions in a class
- If a method is overloaded, the method name is not sufficient to determine which method is being called
- The *signature* of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters

Testing

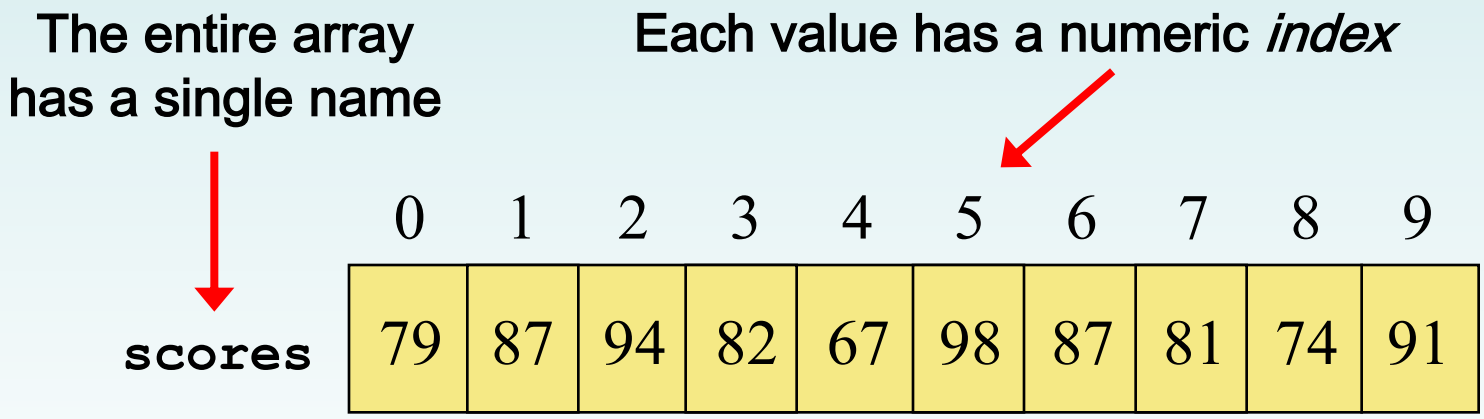
- The goal of testing is to find errors
- As we find and fix errors, we raise our confidence that a program will perform as intended
- We can never really be sure that all errors have been eliminated
- So when do we stop testing?
  - Conceptual answer: Never
  - Cynical answer: When we run out of time
  - Better answer: When we are willing to risk that an undiscovered error still exists

Arrays

- Arrays are objects that help us organize large amounts of information
- Chapter 8 focuses on:
  - array declaration and use
  - bounds checking and capacity
  - arrays that store object references
  - variable length parameter lists
  - multidimensional arrays
  - polygons and polylines
  - choice boxes

Arrays

- An array is an ordered list of values:



An array of size N is indexed from zero to N-1

This array holds 10 values that are indexed from 0 to 9

Arrays

- A particular value in an array is referenced using the array name followed by the index in brackets
- For example, the expression

```
scores[2]
```

  refers to the value `94` (the 3rd value in the array)

- That expression represents a place to store a single integer and can be used wherever an integer variable can be used

Declaring Arrays

- The `scores` array could be declared as follows:

```
int[] scores = new int[10];
```

- The type of the variable `scores` is `int[]` (an array of integers)
- Note that the array type does not specify its size, but each object of that type has a specific size
- The reference variable `scores` is set to a new array object that can hold 10 integers

Bounds Checking

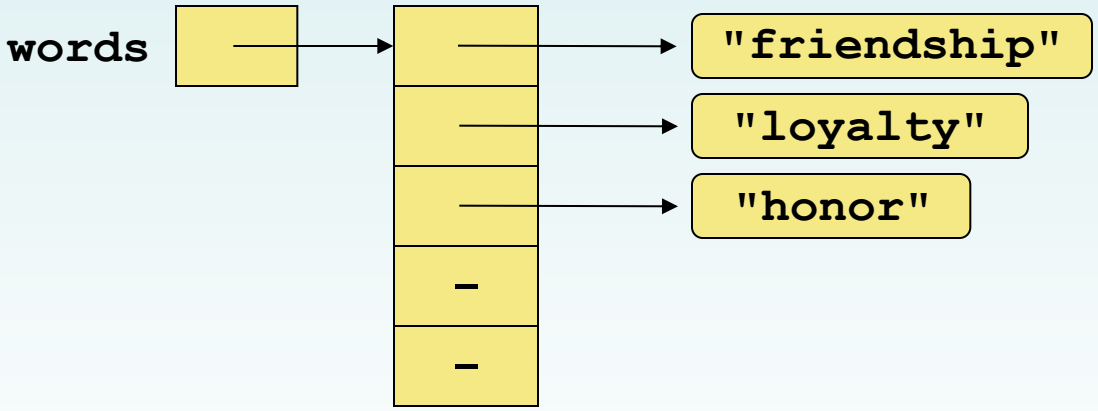
- Once an array is created, it has a fixed size
- An index used in an array reference must specify a valid element
- That is, the index value must be in range 0 to N-1
- The Java interpreter throws an `ArrayIndexOutOfBoundsException` if an array index is out of bounds
- This is called automatic *bounds checking*

Arrays as Parameters

- An entire array can be passed as a parameter to a method
- Like any other object, the reference to the array is passed, making the formal and actual parameters aliases of each other
- Therefore, changing an array element within the method changes the original
- An individual array element can be passed to a method as well, in which case the type of the formal parameter is the same as the element type

Arrays of Objects

- After some `String` objects are created and stored in the array:



Command-Line Arguments

- The signature of the `main` method indicates that it takes an array of `String` objects as a parameter
- These values come from *command-line arguments* that are provided when the interpreter is invoked
- For example, the following invocation of the interpreter passes three `String` objects into the `main` method of the `StateEval` program:

```
java StateEval pennsylvania texas arizona
```

- See `NameTag.java`

Variable Length Parameter Lists

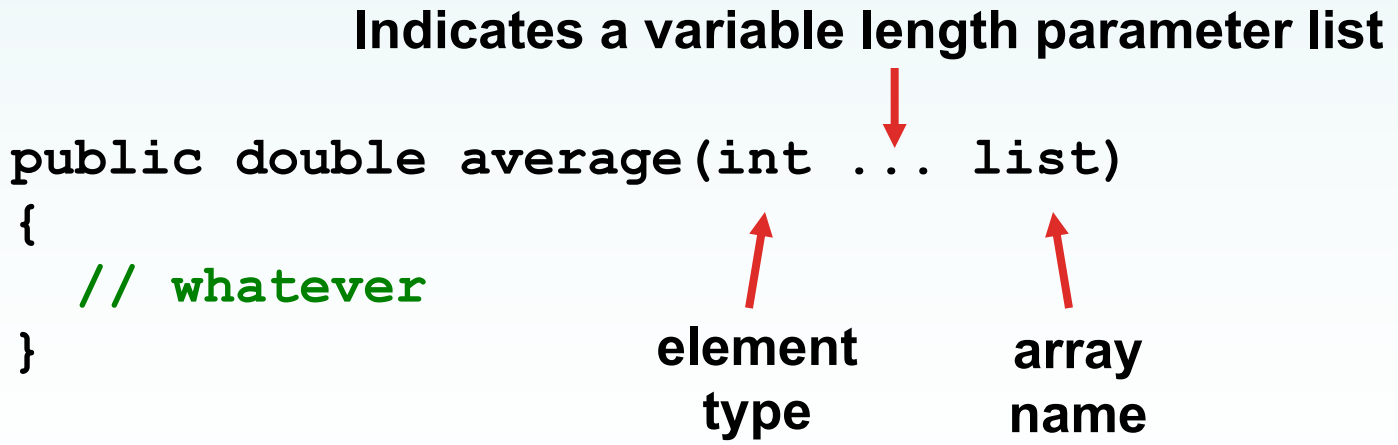
- Suppose we wanted to create a method that processed a different amount of data from one invocation to the next
- For example, let's define a method called `average` that returns the average of a set of integer parameters

```
// one call to average three values
mean1 = average(42, 69, 37);

// another call to average seven values
mean2 = average(35, 43, 93, 23, 40, 21, 75);
```

Variable Length Parameter Lists

- Using special syntax in the formal parameter list, we can define a method to accept any number of parameters of the same type
- For each call, the parameters are automatically put into an array for easy processing in the method



Variable Length Parameter Lists

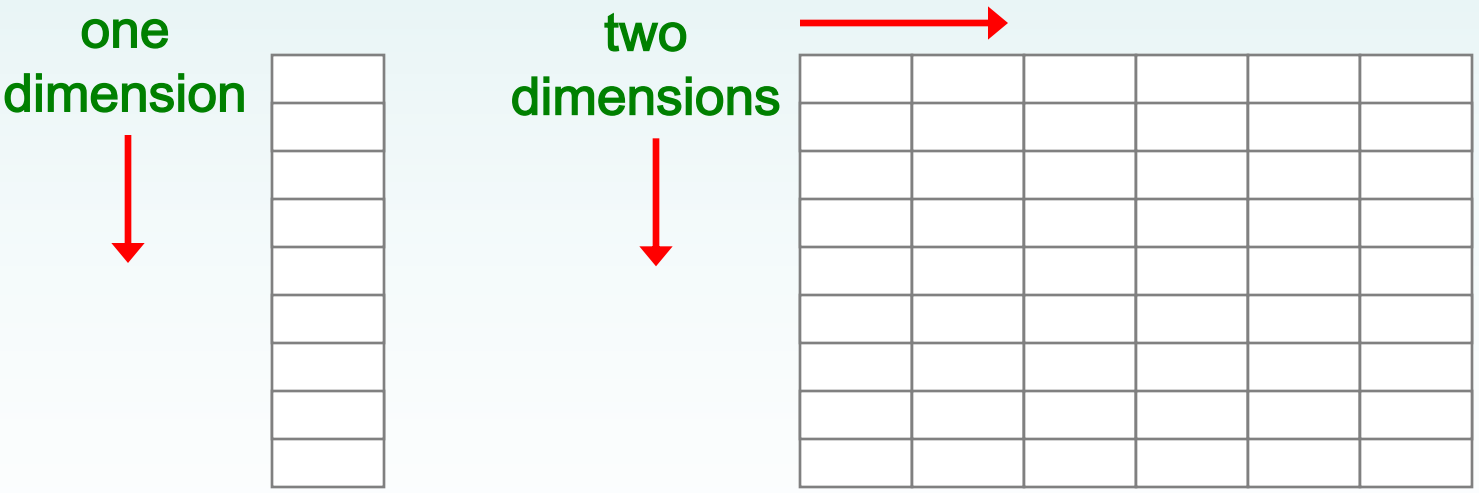
```
public double average(int ... list)
{
   double result = 0.0;

   if (list.length != 0)
   {
      int sum = 0;
      for (int num : list)
         sum += num;
      result = (double)num / list.length;
   }

   return result;
}
```

Two-Dimensional Arrays

- A *one-dimensional array* stores a list of elements
- A *two-dimensional array* can be thought of as a table of elements, with rows and columns



Two-Dimensional Arrays

- To be precise, in Java a two-dimensional array is an array of arrays
- A two-dimensional array is declared by specifying the size of each dimension separately:

```
int[][] table = new int[12][50];
```

- A array element is referenced using two index values:

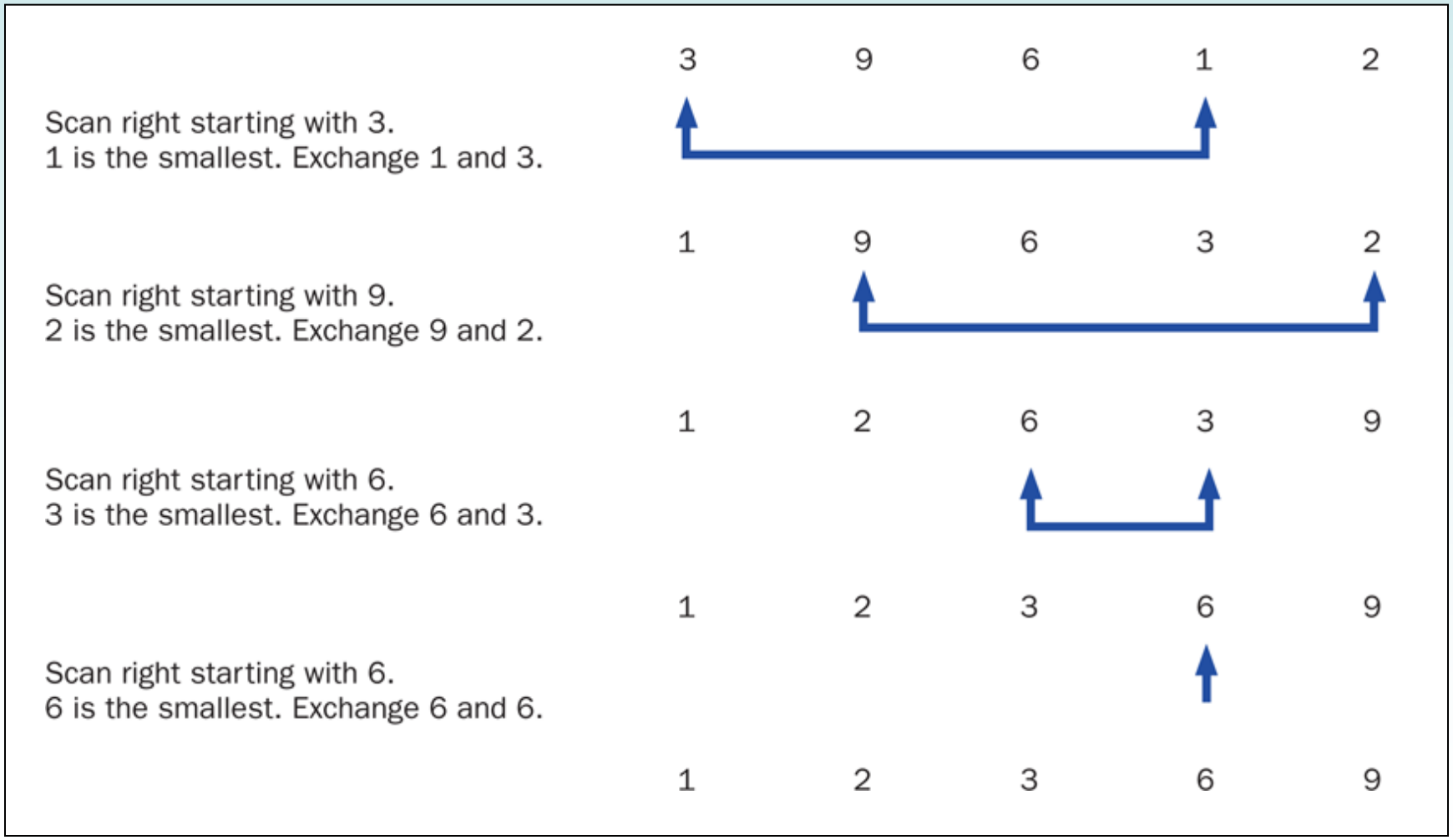
```
value = table[3][6]
```

- The array stored in one row can be specified using one index

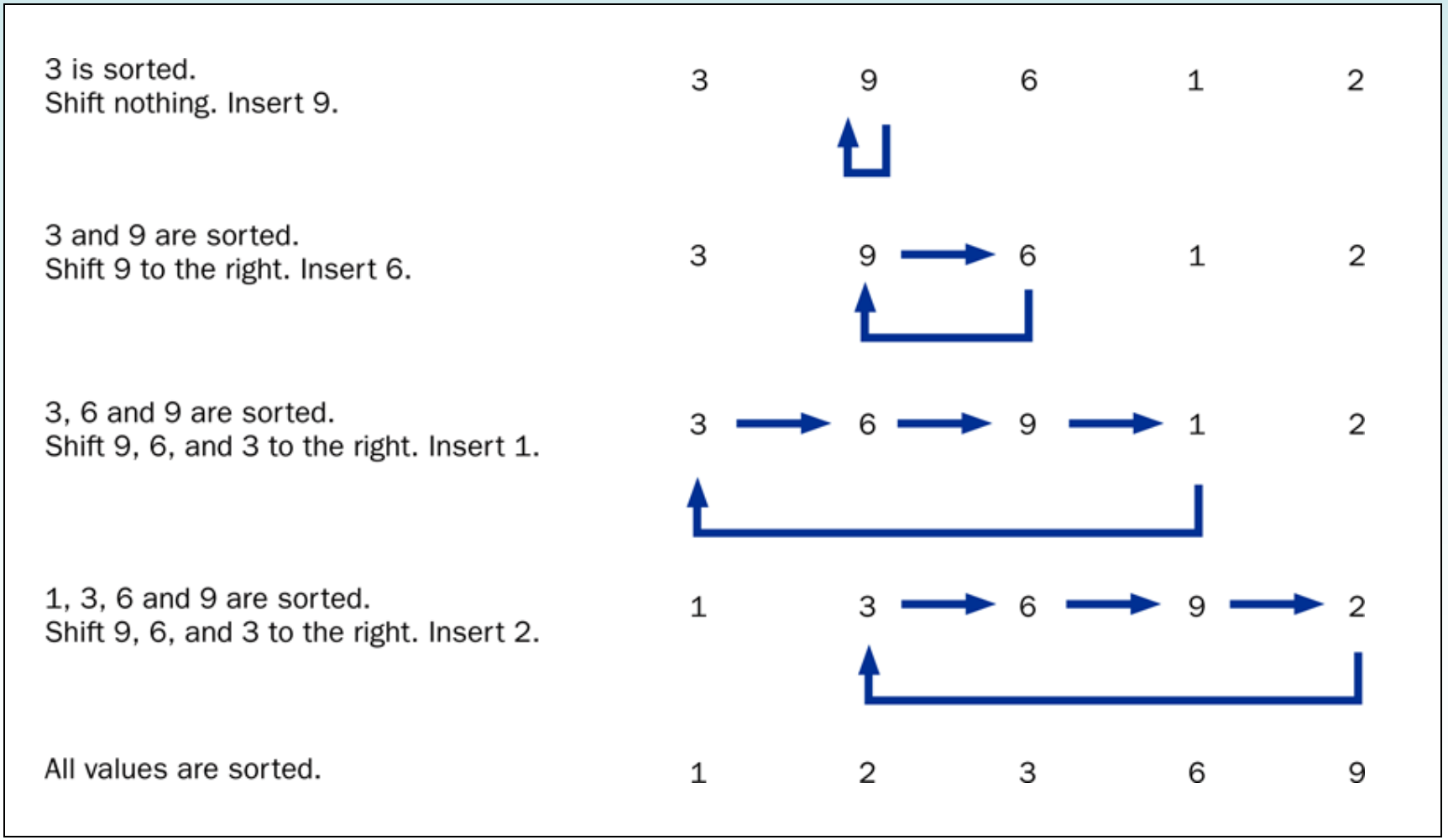
Sorting

- *Sorting* is the process of arranging a list of items in a particular order
- The sorting process is based on specific criteria:
  - sort test scores in ascending numeric order
  - sort a list of people alphabetically by last name
- There are many algorithms, which vary in efficiency, for sorting a list of items
- We will examine two specific algorithms:
  - Selection Sort
  - Insertion Sort

Selection Sort



Insertion Sort



Comparing Sorts

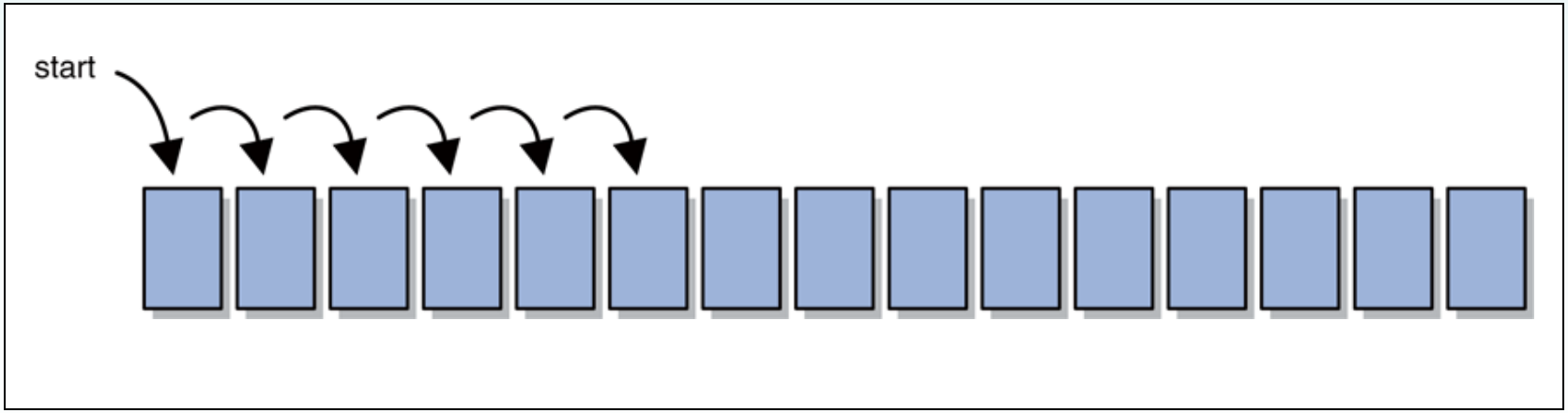
- The Selection and Insertion sort algorithms are similar in efficiency
- They both have outer loops that scan all elements, and inner loops that compare the value of the outer loop with almost all values in the list
- Approximately n^2 number of comparisons are made to sort a list of size n
- We therefore say that these sorts are of *order* n^2
- Other sorts are more efficient: *order* $n \log_2 n$

Searching

- *Searching* is the process of finding a *target element* within a group of items called the *search pool*
- The target may or may not be in the search pool
- We want to perform the search efficiently, minimizing the number of comparisons
- Let's look at two classic searching approaches: linear search and binary search
- As we did with sorting, we'll implement the searches with polymorphic `Comparable` parameters

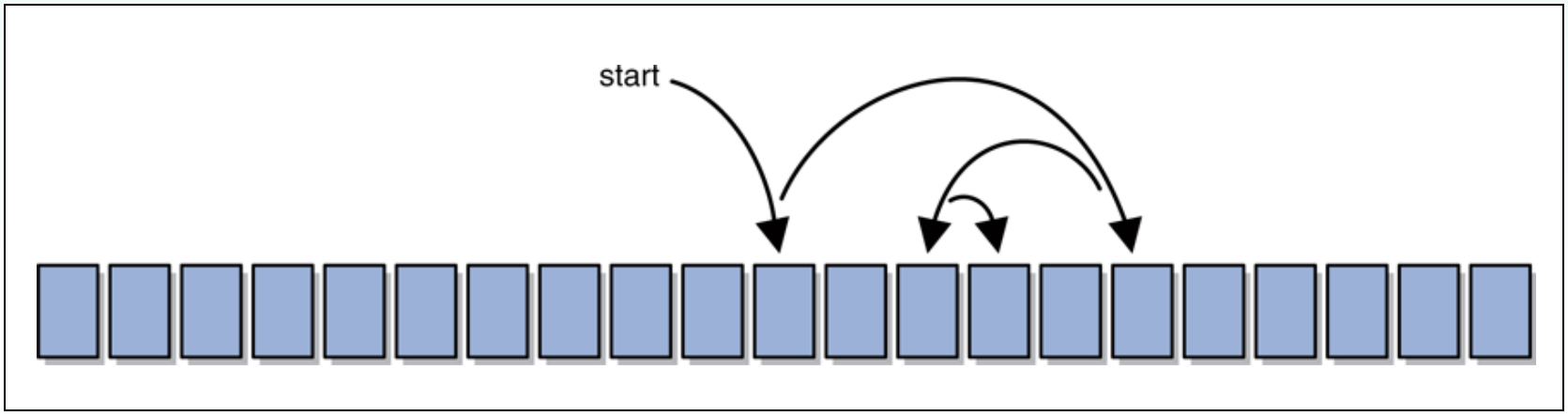
Linear Search

- A linear search begins at one end of a list and examines each element in turn
- Eventually, either the item is found or the end of the list is encountered



Binary Search

- The process continues by comparing the middle element of the remaining *viable candidates*
- Each comparison eliminates approximately half of the remaining data
- Eventually, the target is found or the data is exhausted



Summary

- Chapter 10 has focused on:
  - defining polymorphism and its benefits
  - using inheritance to create polymorphic references
  - using interfaces to create polymorphic references
  - using polymorphism to implement sorting and searching algorithms
  - property binding
  - additional GUI components