Java Basics

Supplement 1

Applications and Applets

- An application : a program that runs on your computer like any other program.
  - It is a stand-alone program
- An applet : a program that cannot run without the support of a browser or a viewer
- This book uses applications rather than applets.

Objects and Classes

- An object : a program construct that contains data and can perform certain actions
  - Objects interact with one another to accomplish a particular task
- Actions performed by objects are defined by methods in the program
  - Valued methods return a value
  - Void methods do not

First Java Application Program

```
import java.util.Scanner;
public class FirstProgram
{
   public static void main(String[] args)
   {
      Scanner keyboard = new Scanner(System.in);
      System.out.println("Hello out there.");
      System.out.println("Want to talk some more?");
      System.out.println("Answer yes or no.");

      String answer = keyboard.next();
      if (answer.equals("yes"))
         System.out.println("Nice weather we are having.");
      System.out.println("Good-bye.");
   } // end main
} // end FirstProgram
```

First Java Application Program

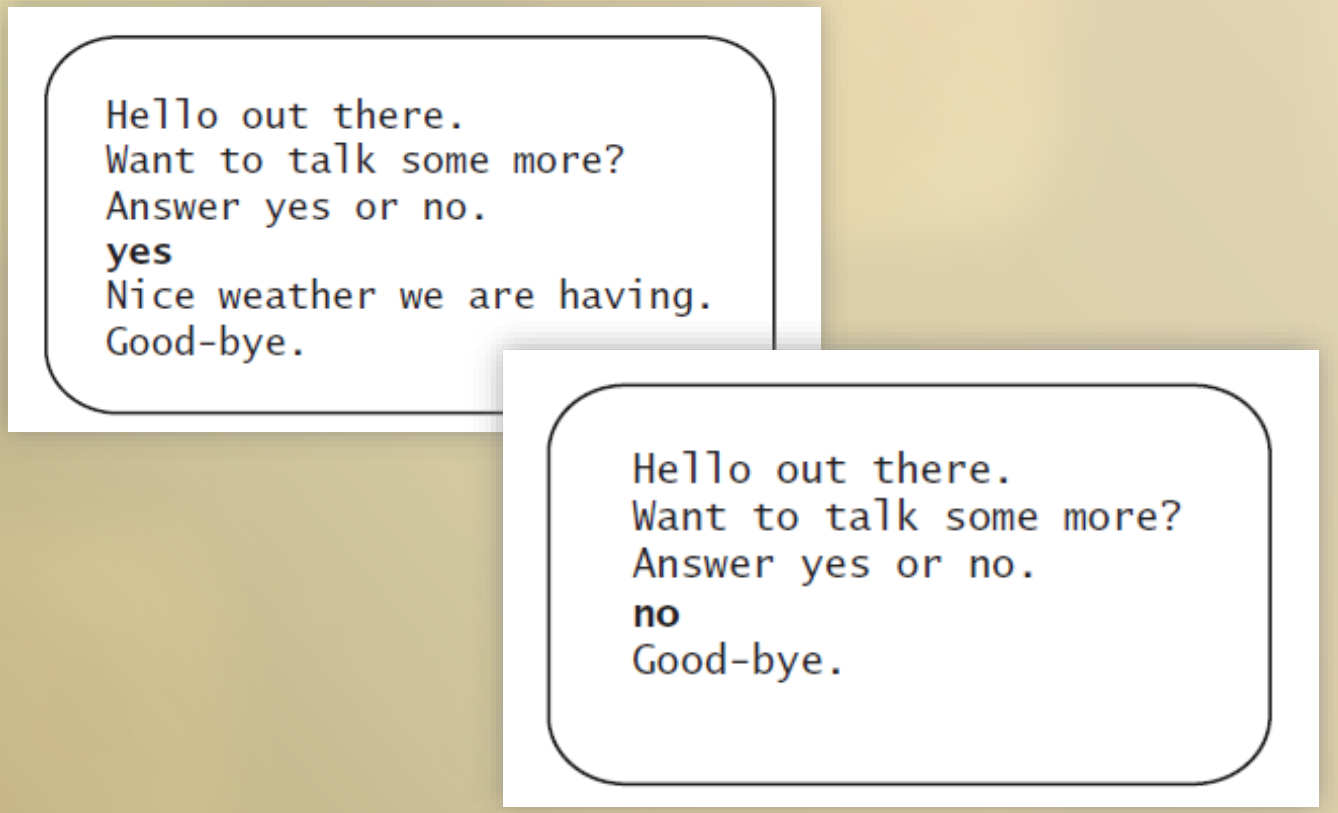


FIGURE B-1 Two possible results when running the sample program

Identifiers

- Use identifiers to name certain parts of a program
  - Consists entirely of letters, digits, the underscore character _, and the dollar sign $
  - Cannot start with a digit, must not contain a space or any other special character
- Java is case sensitive

Identifiers

- Common practice
  - Start the names of classes with uppercase letters
  - Start the names of objects, methods, and variables with lowercase letters

Reserved Words

- Some words have a special predefined meaning in Java
  - Also called keywords
  - Cannot use these words for variable names
  - Used only for the intended purpose

Variables

- Represents a memory location that stores data such as numbers and letters
  - Number or letters stored there are the *value*
  - That value can be changed
- The variable's data type specifies what kind of value may be stored
  - Primitive type
  - Reference type
  - Class type
  - Array type

Variables

- Variable declaration indicates the type of data the variable will hold
  - Write a type name
  - Followed by a list of variable names separated by commas
  - Ending with a semicolon

```
int numberOfBaskets, eggsPerBasket, totalEggs;
String myName;
```

Primitive Types

- Integers
  - Byte, int, short, long
- Floating point
  - Float, double
- Char (single chartacters)

Constants

- Integer constants
  - Optional plus sign or minus sign, no decimal or comma
- Floating-point constants
  - Optional plus sign or minus sign, no comma
  - Also using “e” for power of 10 8.5e6
- Character constants, single quotes ‘K’
- String constants, double quotes “Hi Mom”

Assignment Statements

- Use to give a value to a variable

```
amount = 3.99;
firstInitial = 'B';
score = numberOfCards + handicap;
```

- Single variable on the left-hand side of an equal sign
  - Expression on the right-hand side
  - Followed by a semicolon

Assignment Statements

- Variable declared, but not yet given a value has an undetermined value
- Possible to initialize it within the declaration.

```
int count = 0;
double taxRate = 0.075;
char grade = 'A';
int balance = 1000, newBalance;
```

Assignment Compatibilities

- Cannot put a value of one type in a variable of another type.
  - Unless value is somehow converted to match the type of the variable.
- Dealing with numbers
  - Conversion will *usually* be performed for you automatically

```
int wholeRate = 7;
double interestRate = wholeRate;
```

Type Casting

- Changing of the type of a value to some other type
- Note the *wrong* and *right* way to do this

```
double distance = 9.0;
int points = distance; // ILLEGAL
```

```
int points = (int)distance; // Casting from double to int
```

Arithmetic Operators and Expressions

- Arithmetic operators +, -, *, /, and %.
- Combine variables and constants with these operators and parentheses
  - Form an arithmetic expression
- Unary operator has only one operand

```
x = -5;
```

- Binary operator has two operands

```
total = cost + (tax * discount);
```

Arithmetic Operators and Expressions

- Type of value produced when expression evaluated
  - Depends on the types of the values being combined.
- If all items in arithmetic expression have same type, result has that type.
- If at least one of items has floating-point type, result has floating-point type.

Arithmetic Operators and Expressions

- Combine two numbers with the division operator / …
  - If at least one is a floating point, result is floating point
  - If both are integer, result is truncated integer
- When % operator has operands only of integer types
  - Result is the integer remainder of division

Parentheses and Precedence Rules

- Use parentheses to group portions of an arithmetic expression
  - Same way that you use parentheses in algebra and arithmetic
- Order of precedence
  - The unary operators +, –
  - The binary operators *, /, % (left to right)
  - The binary operators +, – (left to right)
  - Parentheses can override this order

Increment and Decrement Operators

- Increase or decrease the value of a variable by 1

  count++; equivalent to count = count + 1;

- Can be used within expressions

```
int n = 3;
int m = 4;
int result = n * (++m);
```

Incremented before *

```
int n = 3;
int m = 4;
int result = n * (m++);
```

Incremented after *

  – Text does not recommend this practice

Special Assignment Operators

- Combine simple assignment operator (=) with arithmetic operator, such as *

```
amount *= 25
```

Equivalent to

```
amount = amount * 25;
```

Named Constants

- Mechanism allows you to define and initialize a variable *and* fix the variable's value
  - Thus, it cannot be changed

```
public static final double PI = 3.14159;
```

- Good practice to place named constants
  - Near the beginning of a class
  - Outside of any method definitions.
- Typically use all uppercase for named constant

The Class **Math**

- Provides a number of standard mathematical methods.
  - Static methods
  - Write the class name, a dot, the name of the method, and a pair of parentheses
  - Most **Math** methods require that you specify items within the pair of parentheses

```
variable = Math.method_name(arguments);
```

The Class **Math**

In each of the following methods, the argument and the return value are `double`:

`Math.cbrt(x)`	Returns the cube root of x.
`Math.ceil(x)`	Returns the nearest whole number that is $\geq x$.
`Math.cos(x)`	Returns the trigonometric cosine of the angle x in radians.
`Math.exp(x)`	Returns e^x.
`Math.floor(x)`	Returns the nearest whole number that is $\leq x$.
`Math.hypot(x, y)`	Returns the square root of the sum $x^2 + y^2$.
`Math.log(x)`	Returns the natural (base e) logarithm of x.
`Math.log10(x)`	Returns the base 10 logarithm of x.
`Math.pow(x, y)`	Returns x^y.
`Math.random()`	Returns a random number that is ≥ 0 but < 1.
`Math.sin(x)`	Returns the trigonometric sine of the angle x in radians.

FIGURE B-2 Some methods in the class **Math**

The Class **Math**

`Math.sin(x)`	Returns the trigonometric sine of the angle x in radians.
`Math.sqrt(x)`	Returns the square root of x, assuming that $x \geq 0$.
`Math.tan(x)`	Returns the trigonometric tangent of the angle x in radians.
`Math.toDegrees(x)`	Returns an angle in degrees equivalent to the angle x in radians.
`Math.toRadians(x)`	Returns an angle in radians equivalent to the angle x in degrees.
In each of the following methods, the argument and the return value have the same type– either `int`, `long`, `float`, or `double`:	
`Math.abs(x)`	Returns the absolute value of x.
`Math.max(x, y)`	Returns the larger of x and y.
`Math.min(x, y)`	Returns the smaller of x and y.
`Math.round(x)`	Returns the nearest whole number to x. If x is `float`, returns an `int`; if x is `double`, returns a `long`.

FIGURE B-2 Some methods in the class **Math**

Screen Output

- Statements of the form

```
System.out.println(quarters + " quarters");
```

  send output to the screen

- To display more than one thing, simply place a + operator between them

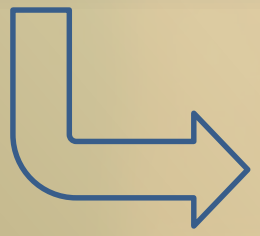
```
System.out.println("Lucky number = " + 13 +
                   "Secret number = " + number);
```

  – the + operator joins, or concatenates, two strings

Screen Output

- Every invocation of **println** ends a line of output
- If you want the output from two or more output statements to appear on a single line, use **print**

```
System.out.print("One, two,");
System.out.print(" buckle my shoe.");
System.out.println(" Three, four,");
System.out.println("shut the door.");
```



```
One, two, buckle my shoe. Three, four,
shut the door.
```

Keyboard Input
Using the Class **Scanner**

- Class **Scanner** must be imported
  - Write this line at beginning of program

```
import java.util.Scanner;
```

- Must then create a **Scanner** object

```
Scanner keyboard = new Scanner(System.in);
```

- Read integers, real numbers, strings

```
System.out.println("Please enter your height in feet and inches:");
int feet = keyboard.nextInt();
int inches = keyboard.nextInt();
String message = keyboard.nextLine();
```

The **if-else** Statement

- Meaning of **if-else** statement, same meaning it would have if read as an English sentence

```
if (balance >= 0)
   balance = balance + (INTEREST_RATE * balance) / 12;
else
   balance = balance - OVERDRAWN_PENALTY;
```

- To include more than one statement, braces

```
if (balance >= 0)
{
   System.out.println("Good for you. You earned interest.");
   balance = balance + (INTEREST_RATE * balance) / 12;
}
else
{
   System.out.println("You will be charged a penalty.");
   balance = balance - OVERDRAWN_PENALTY;
} // end if
```

The **if-else** Statement

- You can omit the **else** part

```
if (balance >= 0)
{
   System.out.println("Good for you. You earned interest.");
   balance = balance + (INTEREST_RATE * balance) / 12;
} // end if
```

- If you do, nothing happens when tested expression is false

Boolean Expressions

- An expression that is either true or false
  - As used in the previous **if-else** statement

```
if (balance >= 0)
{
   System.out.println("Good for you. You earned interest.");
   balance = balance + (INTEREST_RATE * balance) / 12;
} // end if
```

- Uses comparison operator

Boolean Expressions

Math Notation	Name	Java Operator	Java Examples
≥	Greater than or equal to	>=	`points >= 60`
≤	Less than or equal to	<=	`expenses <= income`
>	Greater than	>	`expenses > income`
<	Less than	<	`pressure < max`
=	Equal to	==	`balance == 0` `answer == 'y'`
≠	Not equal to	!=	`income != tax` `answer != 'y'`

FIGURE B-3 Java comparison operators

Logical Operators

- Enables use of boolean expression more complicated than a simple comparison

```
if ((pressure > min) && (pressure < max))
   System.out.println("Pressure is OK.");
else
   System.out.println("Warning: Pressure is out of range.");
```

- Operators
  - Operator && logical **and**
  - Operator || logical **or**
  - Operator ! logical **not**

Logical Operators

- Precedence of operators
  - The unary operators +, -, !
  - The binary arithmetic operators *, /, %
  - The binary arithmetic operators +, -
  - The comparison operators <, >, <=, >=
  - The comparison operators ==, !=
  - The logical operator &&
  - The logical operator ||
  - Can be overridden with parentheses

Nested Statements

- Can use one **if-else** statement within another **if-else** statement to get nested **if-else** statements

```
if (balance >= 0)
   if (INTEREST_RATE >= 0)
      balance = balance + (INTEREST_RATE * balance) / 12;
   else
      System.out.println("Cannot have a negative interest.");
else
   balance = balance - OVERDRAWN_PENALTY;
```

- Braces can be used to clarify or to alter nesting sequence

Multiway **if-else** Statements

- **if-else** statement has two outcomes
  - Each of these two outcomes can have an **if- else** statement with two outcomes
  - Can use nested **if-else** statements to produce any number of possible effects

```
if (balance > 0)
   System.out.println("Positive balance");
else if (balance < 0)
   System.out.println("Negative balance");
else if (balance == 0)
   System.out.println("Zero balance");
```

Multiway **if-else** Statements

- If more than one boolean expression is true
  - Only the action associated with the first true boolean expression is executed
  - Multiway **if-else** statement never performs more than one action
- Good practice to add **else** clause—without any **if** —at the end
  - Executed in case none of the boolean expressions is true

The **switch** Statement

- Multiway **if-else** statements can become unwieldy
- If choice is based on value of integer or character expression
  - **switch** statement can make code easier to read
- Begins with word **switch** followed by expression in parentheses
  - Expression must be **int, char, byte, short, String**

The **switch** Statement

```
int seatLocationCode;
< Code here assigns a value to seatLocationCode >
. . .
double price = -0.01;
switch (seatLocationCode)
{
   case 1:
      System.out.println("Balcony.");
      price = 15.00;
      break;
   case 2:
      System.out.println("Mezzanine.");
      price = 30.00;
      break;
   case 3:
      System.out.println("Orchestra.");
      price = 40.00;
      break;
   default:
      System.out.println("Unknown ticket code.");
      break;
} // end switch
```

switch statement determines the price of a ticket according to location of seat in theater

The **switch** Statement

- **switch** statement contains a list of cases, each consisting of
  - Reserved word **case**,
  - A constant, a colon, and
  - A list of statements that are actions for the case.

```
case 1:
   System.out.println("Balcony.");
   price = 15.00;
   break;
case 2:
   System.out.println("Mezzanine.");
```

The **switch** Statement

- Note the **break**; statement
  - Optional
  - If not present, execution continues on to next case

```
case 1:
   System.out.println("Balcony.");
   price = 15.00;
   break;
case 2:
   System.out.println("Mezzanine.");
```

The **switch** Statement

- Note optional **default**

```
        price = 40.00;
        break;
    default:
        System.out.println("Unknown ticket code.");
        break;
} // end switch
```

- If not present
  - Nothing happens
- Author encourages use
  - An error message helps designer find missed case

The **switch** Statement

- Possible to specify same action for multiple cases

```
char seatLocationCode;
< Code here assigns a value to seatLocationCode >
. . .
double price = -0.01;
switch (seatLocationCode)
{
   case 'B':
   case 'b':
      System.out.println("Balcony.");
      price = 15.00;
      break;
   case 'M': case 'm':
      System.out.println("Mezzanine.");
```

Enumerations

- An enumeration itemizes the values that a variable can have.
- Example: define **LetterGrade** as an enumeration

```
enum LetterGrade {A, B, C, D, F}
```

- **LetterGrade** behaves as a class type
  - Values behave as static constants

```
LetterGrade grade;
grade = LetterGrade.A;
```

Enumerations

- You can use a **switch** statement with a variable whose data type is an enumeration.

```
switch (grade)
{
   case A:
      qualityPoints = 4.0;
      break;
   case B:
      qualityPoints = 3.0;
      break;
   case C:
      qualityPoints = 2.0;
      break;
   case D:
      qualityPoints = 1.0;
      break;
   case F:
      qualityPoints = 0.0;
      break;
   default:
      qualityPoints = -9.0;
} // end switch
```

Scope

- **scope** of a variable is the portion of a program in which the variable is available.

```
{
   // counter and greeting are not available here
   . . .
   int counter = 1;
   // counter is available here
   . . .
   {
      String greeting = "Hello!";
      // Both greeting and counter are available here
      . . .
   } // end scope of greeting
   . . .
   // Only counter is available here
   . . .
} // end scope of counter
```

Loops

- Portion of a program that repeats a statement or group of statements is called a loop.
- Statement(s) to be repeated in a loop called the body
  - Each repetition of the loop body called an iteration

The **while** Statement

- General form

```
while (expression)
   statement;
```

```
int number;
. . . // Assign a value to number here
int count = 1;
while (count <= number)
{
   System.out.println(count);
   count++;
} // end while
```

while statement displays the integers from 1 to a given integer number:

The **while** Statement

- Loop may do zero iterations
  - If count <= number is false, the body of the loop is never executed

```
int number;
. . . // Assign a value to number here
int count = 1;
while (count <= number)
{
   System.out.println(count);
   count++;
} // end while
```

The **while** Statement

- Infinite loop
  - If we forget to increment count, the condition is never met

```
int number;
. . . // Assign a value to number here
int count = 1;
while (count <= number)
{
   System.out.println(count);
   count++;
} // end while
```

The **for** Statement

- General form

```
for (initialize; test; update)
   statement;
```

- Same result as while loop shown
  - **for** statement increments for the loop

```
int count, number;
. . . // Assign a value to number here
for (count = 1; count <= number; count++)
   System.out.println(count);
```

The **for** Statement

- **for** loop can perform more than one initialization
  - Use a list of initialization actions
  - Separate the actions with commas

```
int n, product;
for (n = 1, product = 1; n <= 10; n++)
   product = product * n;
```

The **for** Statement

- Using an enumeration with a **for** statement
  - Declare a variable to the left of a colon
  - To right of colon, represent values that variable will have

```
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES}
...
for (Suit nextSuit : Suit.values())
   System.out.println(nextSuit);
```

The **do-while** Statement

- Similar to the **while** statement
  - But, body of a **do-while** statement always executes at least once
- General form

```
do
    statement;
while (expression);
```

  - Be sure to include a semicolon at the end of a **do-while** statement.

The **do-while** Statement

- Be sure to include a semicolon at the end of a **do-while** statement.

```
int number;
. . . // Assign a value to number here
int count = 1;
do
{
   System.out.println(count);
   count++;
} while (count <= number);
```

Additional Loop Information

- If loop must run at least one time
  - Use **do-while**
- If loop might not be needed to execute even first time,
  - Use **while-loop**
- **Break** statement can jump out of a loop
- **Continue** statement can jump back to top of loop

The Class **String**

- Part of the package **java.lang** in the Java Class Library
- Use **String** objects to create and process strings of characters.
- Java uses the Unicode character set
  - Codes for ASCII are same in Unicode

The Class **String**

- Consider displaying this line on the screen

The word “Java” names a language and a drink!

  - *Cannot* use this code

```
System.out.println("The word "Java" names a language and a drink!");
```

  - This is a misuse of double quotes “

- Instead use escape characters

```
System.out.println("The word \"Java\" names a language and a drink!");
```

The Class **String**

`\"`	Double quote.
`\'`	Single quote (apostrophe).
`\\`	Backslash.
`\n`	New line. (Go to the beginning of the next line.)
`\r`	Carriage return. (Go to the beginning of the current line.)
`\t`	Tab. (Insert whitespace up to the next tab stop.)

FIGURE B-4 Escape characters

Concatenation of Strings

- Join two strings by using the + operator
  - The concatenation operator for strings

```
String greeting = "Hello";
String sentence = greeting + "my friend.";
System.out.println(sentence);
```

- Result displayed on screen is

```
Hellomy friend.
```

String Methods

- **String** object has methods as well as a value
  - Use these methods to manipulate string values
- **length** gets number of characters in a string
- Use the **concat** instead of the + operator

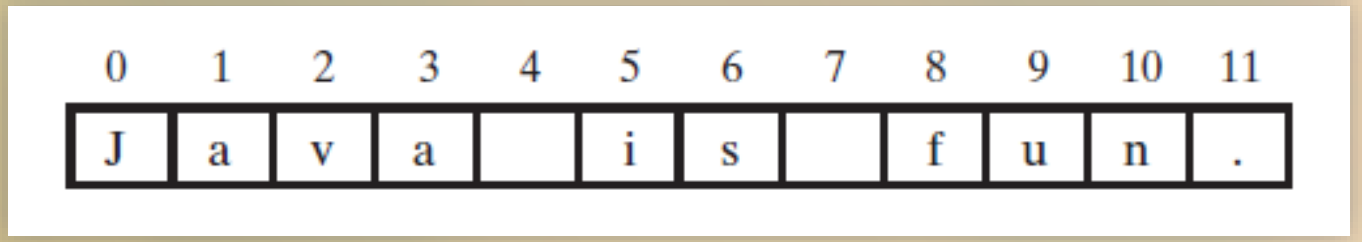


FIGURE B-5 Indices 0 through 11 for the string "Java is fun."

String Methods

- **charAt** returns the character at the index given
  - If index negative or too large, causes error
- **indexOf** tests whether string contains given substring
  - If it does, returns index at which substring begins
- **toLowerCase** replaces uppercase letters with their lowercase counterparts of argument

String Methods

- **trim** trims off leading, trailing white space
- Use method **compareTo** to compare two strings – lexicographically
  s1.compareTo (s2) returns
  - negative integer if s1 < s2
  - positive integer if s1 > s2
  - zero if s1 = s2

The Class **StringBuilder**

- **append(String s)** concatenates s to the calling object
- **delete(int before, int after)** removes the substring of this string beginning at index **start** and ending at either index **after – 1** or the end of the string
- **insert(int index, String s)** Inserts string **s** into this string at given **index**
  - Returns a reference to the result

The Class **StringBuilder**

- **replace(int start, int after, String s)** replaces substring of this string with string **s**.
  - Substring to be replaced begins at index **start** and ends at either the index **after – 1** or end of string
- **setCharAt(int index, char character)** sets character at given **index** of this string to a given **character**

Using **Scanner** to Extract Pieces of a String

- In addition to reading data from keyboard
  - Can use Scanner to process a string that defined within program.

```
String phrase = "one potato        two    potato three potato four";
Scanner scan = new Scanner(phrase);
System.out.println(scan.next());
System.out.println(scan.next());
System.out.println(scan.next());
System.out.println(scan.next());
```

- Resulting display:

```
one
potato
two
potato
```

Using **Scanner** to Extract Pieces of a String

- You can specify the delimiters that **Scanner** will use

```
String data = "one,potato,two,potato";
Scanner scan = new Scanner(data);
scan.useDelimiter(",");
System.out.println(scan.next());
System.out.println(scan.next());
System.out.println(scan.next());
System.out.println(scan.next());
```

- Resulting display:

```
one
potato
two
potato
```

Using **Scanner** to Extract Pieces of a String

`\d`	Any digit 0 through 9
`\D`	Any character other than a digit
`\s`	Any white-space character
`\S`	Any character other than white space
`\w`	Any letter, digit, or underscore
`\W`	Any character other than a letter, digit, or underscore
`.`	Any character
X	One occurrence of *X*
X?	Zero or one occurrence of *X*
*X**	Zero or more occurrences of *X*
X+	One or more occurrences of *X*
X{*n*}	Exactly *n* occurrences of *X*
X{*n*,}	At least *n* occurrences of *X*

FIGURE B-6 Some notation used to define the delimiters that Scanner uses

Using **Scanner** to Extract Pieces of a String

```
String phrase = "5 potato        6potato 7 potato more";
Scanner scan = new Scanner(phrase);
scan.useDelimiter("\\s*\\d\\s*");
System.out.println(scan.next());
System.out.println(scan.next());
System.out.println(scan.next());
```

- Resulting display:

```
potato
potato
potato more
```

Arrays

- A special kind of object that stores a finite collection of items having the same data type

```
double[] temperature = new double[7];
```

  - Left side of assignment operator declares **temperature** an array whose contents are of type double.
  - Right side uses **new** operator to request seven memory locations for array
  - Number in brackets – the index, integer value

Arrays

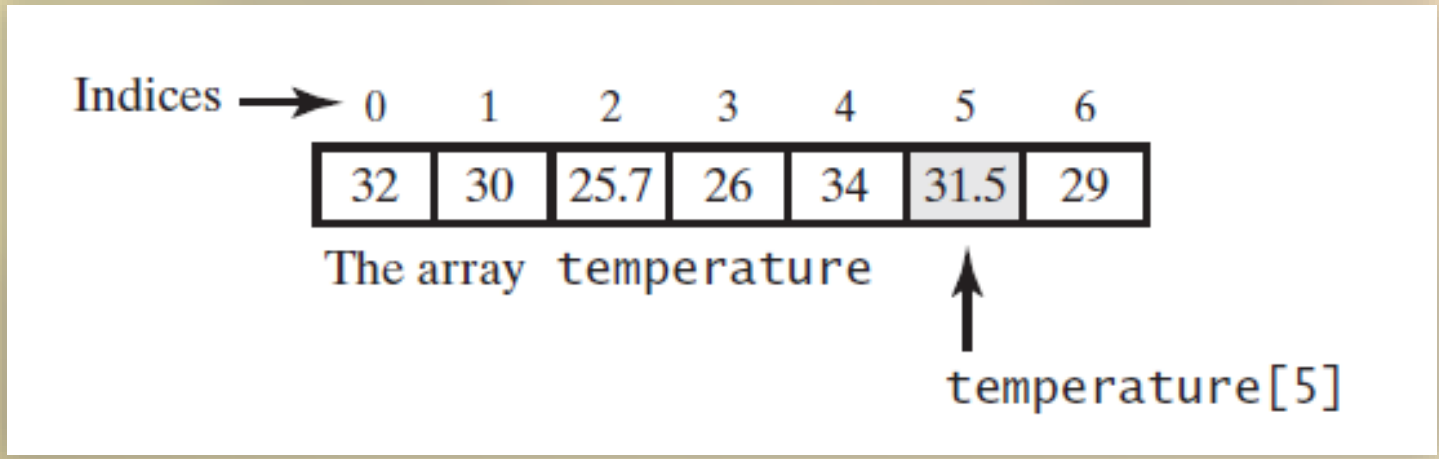


FIGURE B-7 An array of seven temperatures

- Note: array is full, each location has a value
- Arrays are hot always full – must distinguish between length and number of items currently stored

Array Parameters and Returned Values

- You can pass indexed variable as argument to a method
  - Anyplace you can pass ordinary variable of array's entry type.
- An entire array can also be a single argument to a method

```
public static void incrementArrayBy2(double[] array)
{
   for (int index = 0; index < array.length; index++)
      array[index] = array[index] + 2;
} // end incrementArrayBy2
```

Array Parameters and Returned Values

- A method can return an array

```
public static double[] incrementArrayBy2(double[] array)
{
   double[] result = new double[array.length];
   for (int index = 0; index < array.length; index++)
      result[index] = array[index] + 2;
   return result;
} // end incrementArrayBy2
```

  - Call of this method …

```
double[] originalArray = new double[10];
< Statements that place values into originalArray >
. . .

double[] revisedArray = incrementArrayBy2(originalArray);
< At this point, originalArray is unchanged. >
```

Initializing Arrays

- Provide initial values for the elements in an array when you declare it

```
double[] reading = {3.3, 15.8, 9.7};
```

- You do not explicitly state array's length.
  - Length is minimum number of locations that will hold given values

Array Index Out of Bounds

- Consider this array

```
double[] temperature = new double[7];
```

  - If index is negative or greater than 6, it is said to be “out of bounds”
- If index is an expression and out of bounds
  - Causes an **IndexOutOfBoundsException**

Use of = and == with Arrays

- Variable name of an array holds a memory address
  - Data for the array starts at that location
- Consider use of = operator with two arrays **a[ ]** and **b[ ]** where we say **a = b**
  - Result is now both **a** and **b** point to **b**'s location
  - These variables are actually aliases

Use of = and == with Arrays

- Suppose you want array **b** to have same values as array **a**, but in separate memory locations
  - Must use a loop as shown here

```
for (int index = 0; index < a.length; index++)
   b[index] = a[index];
```

Use of = and == with Arrays

- Again with arrays a[ ] and b[ ] of same type
- Now if we make the comparison a == b
  - We are testing two arrays to see if they are stored in same place in computer's memory
  - We are *not* checking for equality of contents
- Must compare the two arrays entry by entry
  - With a looping construct

Arrays and the For-Each Loop

- Can use **for-each** loop to process all the values in an array

```
int[] anArray = {1, 2, 3, 4, 5};
int sum = 0;
for (int integer : anArray)
   sum = sum + integer;
System.out.println(sum);
```

Multidimensional Arrays

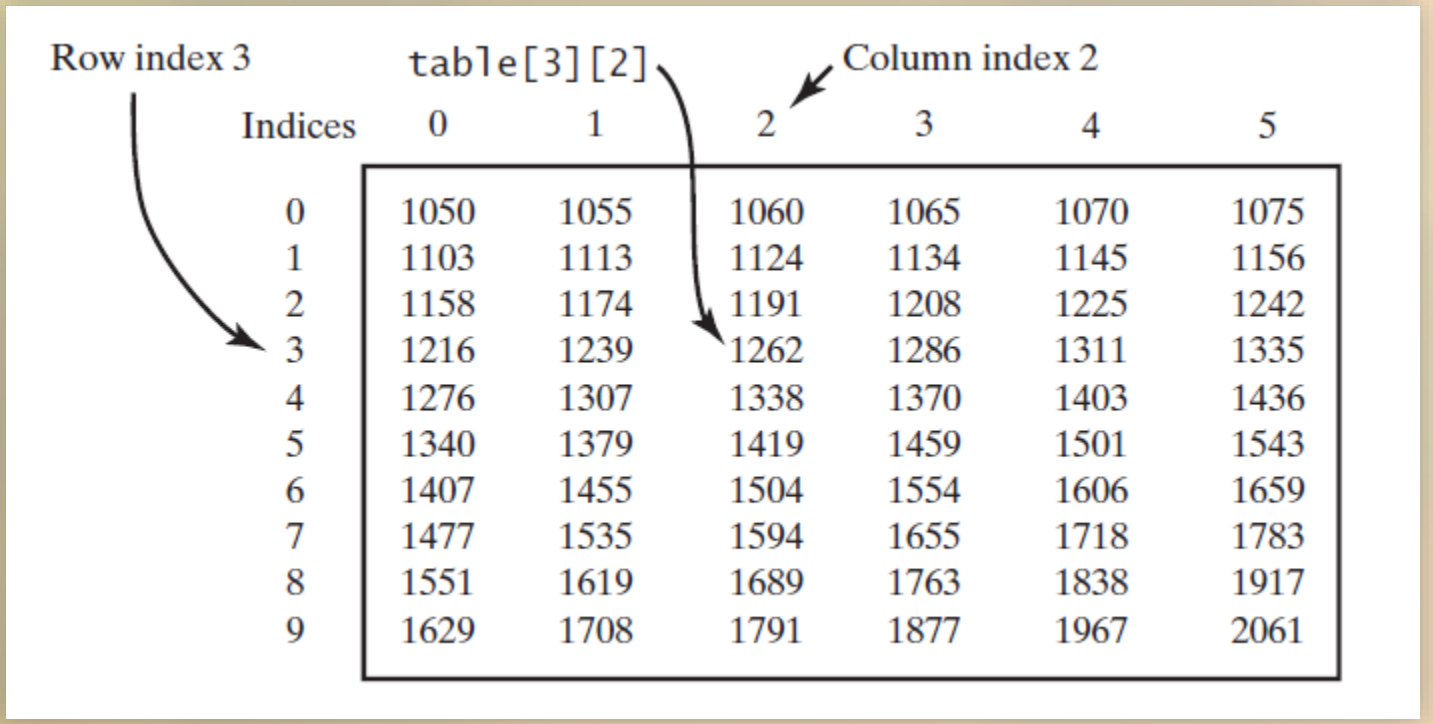
- Can have an array with more than one index
  - Could hold a table of values
- Note table on following slide
  - Effect of various interest rates on $1000 when compounded annually

Multidimensional Arrays

The effect of various interest rates on $1000 when compounded annually (rounded to whole dollars)						
Year	**5.00%**	**5.50%**	**6.00%**	**6.50%**	**7.00%**	**7.50%**
1	$1050	$1055	$1060	$1065	$1070	$1075
2	$1103	$1113	$1124	$1134	$1145	$1156
3	$1158	$1174	$1191	$1208	$1225	$1242
4	$1216	$1239	$1262	$1286	$1311	$1335
5	$1276	$1307	$1338	$1370	$1403	$1436
6	$1340	$1379	$1419	$1459	$1501	$1543
7	$1407	$1455	$1504	$1554	$1606	$1659

FIGURE B-8 A table of values

Multidimensional Arrays



Indices	0	1	2	3	4	5
0	1050	1055	1060	1065	1070	1075
1	1103	1113	1124	1134	1145	1156
2	1158	1174	1191	1208	1225	1242
3	1216	1239	1262	1286	1311	1335
4	1276	1307	1338	1370	1403	1436
5	1340	1379	1419	1459	1501	1543
6	1407	1455	1504	1554	1606	1659
7	1477	1535	1594	1655	1718	1783
8	1551	1619	1689	1763	1838	1917
9	1629	1708	1791	1877	1967	2061

FIGURE B-9 Row and column indices for an array named **table**; **table[3][2]** is the element in the fourth row and third column

Multidimensional Arrays

- A loop that will set all the values of **table** to zero

```
for (int row = 0; row < 10; row++)
   for (int column = 0; column < 6; column++)
      table[row][column] = 0;
```

- Multidimensional array can be parameter of a method

```
public static void clearArray(double[][] array)
```

  - Above loop could be placed in a method of this name

Multidimensional Arrays

- Java implements multidimensional arrays as one-dimensional arrays
  - Given

```
int[][] table = new int[10][6];
```

- Array **table** is in fact a one-dimensional array of length 10, and its entry type is **int[]**
- In other words, a multidimensional array is an array of arrays

Wrapper Classes

- An argument to a method and the assignment operator = behave differently for primitive types and class types
- To make things uniform, Java provides a wrapper class for each of primitive types
  - Enables conversion of a value of primitive type to object of corresponding class type.

Wrapper Classes

- Example: we want to convert an **int** value, such as 10, to an object of type **Integer**
  - Can be done in one of three ways

```
Integer ten = new Integer(10);
Integer fiftyTwo = new Integer("52");
Integer eighty = 80;
```

- Now use methods **equals** and **compareTo** for comparisons
  - Do not use **==** for comparisons or **=** for assignments as with primitives

Wrapper Classes

- You can use same operators that you use for arithmetic with primitives

```
Scanner keyboard = new Scanner(System.in);
System.out.print("What is his age? ");
int hisAge = keyboard.nextInt();
System.out.print("What is her age? ");
Integer herAge = keyboard.nextInt();

Integer ageDifference = Math.abs(hisAge - herAge);
System.out.println("He is " + hisAge + ", she is " + herAge +
                   ": a difference of " + ageDifference + ".");
```

Wrapper Classes

- Wrapper classes contain useful static constants
  - The largest and smallest values of type **int** are

```
Integer.MAX_VALUE and Integer.MIN_VALUE
```

  - Methods that can be used to convert a string to the corresponding numerical type

```
Double.parseDouble(theString)
```

  - Or back the other direction

```
Integer.toString(42)
```

Wrapper Classes

- **Character** is the wrapper class for the primitive type **char**
- Some of the methods include
  - **toLowerCase, toUpperCase**
  - **isLowerCase, isUpperCase**
  - **isLetter, isDigit, isWhitespace**

Java Basics

End