Chapter 1-9, 12-13, 18, 20, 23
Review Slides

CS1: Java Programming
Colorado State University

Original slides by Daniel Liang
Modified slides by Chris Wilcox

What is a Computer?
A computer consists of a CPU, memory, hard disk, floppy disk, monitor, printer, and communication devices.

Characteristics of Java
- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Companion Website
www.cs.armstrong.edu/liang/JavaCharacteristics.pdf
Declaring Variables

```java
int x;         // Declare x to be an integer variable;
double radius; // Declare radius to be a double variable;
char a;        // Declare a to be a character variable;
```

Assignment Statements

```java
x = 1;          // Assign 1 to x;
radius = 1.0;   // Assign 1.0 to radius;
a = 'A';        // Assign 'A' to a;
```

Identifiers

- An identifier is a sequence of characters that consist of letters, digits, underscores (_), and dollar signs ($).
- An identifier must start with a letter, an underscore (_), or a dollar sign ($). It cannot start with a digit.
- An identifier cannot be a reserved word. (See Appendix A, “Java Keywords,” for a list of reserved words).
- An identifier cannot be true, false, or null.
- An identifier can be of any length.
Numerical Data Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Range</th>
<th>Storage Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>$-2^{7} \leq x \leq 2^{7} - 1 (-128 \leq x \leq 127)$</td>
<td>8-bit signed</td>
</tr>
<tr>
<td>short</td>
<td>$-2^{15} \leq x \leq 2^{15} - 1 (-32768 \leq x \leq 32767)$</td>
<td>16-bit signed</td>
</tr>
<tr>
<td>int</td>
<td>$-2^{31} \leq x \leq 2^{31} - 1 (-2147483648 \leq x \leq 2147483647)$</td>
<td>32-bit signed</td>
</tr>
<tr>
<td>long</td>
<td>$-2^{63} \leq x \leq 2^{63} - 1 (-9223372036854775808 \leq x \leq 9223372036854775807)$</td>
<td>64-bit signed</td>
</tr>
<tr>
<td>float</td>
<td>Negative range: $-1.4028235E+45 \leq x \leq 1.4E-45$</td>
<td>32-bit IEEE 754</td>
</tr>
<tr>
<td>double</td>
<td>Negative range: $-1.7976931348623157E+308 \leq x \leq 1.7976931348623157E+308$</td>
<td>64-bit IEEE 754</td>
</tr>
</tbody>
</table>

Numeric Operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>34 + 1</td>
<td>35</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>34.0 - 0.1</td>
<td>33.9</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>300 * 30</td>
<td>9000</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>1.0 / 2.0</td>
<td>0.5</td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
<td>20 % 3</td>
<td>2</td>
</tr>
</tbody>
</table>

Integer Division

\(+, -, *, /, \text{ and } \%\)

5 / 2 yields an integer 2.
5.0 / 2 yields a double value 2.5

5 % 2 yields 1 (the remainder of the division).
How to Evaluate an Expression

Though Java has its own way to evaluate an expression behind the scene, the result of a Java expression and its corresponding arithmetic expression are the same. Therefore, you can safely apply the arithmetic rule for evaluating a Java expression.

3 + 4 * 4 + 5 * (4 + 3) - 1
3 + 4 * 4 + 5 * 7 – 1
3 + 16 + 5 * 7 – 1
3 + 16 + 35 – 1
19 + 35 – 1

Conversion Rules

When performing a binary operation involving two operands of different types, Java automatically converts the operand based on the following rules:

1. If one of the operands is double, the other is converted into double.
2. Otherwise, if one of the operands is float, the other is converted into float.
3. Otherwise, if one of the operands is long, the other is converted into long.
4. Otherwise, both operands are converted into int.

Type Casting

Implicit casting

```java
double d = 3;  // (type widening)
```

Explicit casting

```java
int i = (int)3.0;  // (type narrowing)
int i = (int)3.9;  // (Fraction part is truncated)
```

What is wrong? int x = 5 / 2.0;
The boolean Type and Operators

Often in a program you need to compare two values, such as whether i is greater than j. Java provides six comparison operators (also known as relational operators) that can be used to compare two values. The result of the comparison is a Boolean value: true or false.

boolean b = (1 > 2);

Relational Operators

<table>
<thead>
<tr>
<th>Java Operator</th>
<th>Mathematics Symbol</th>
<th>Name</th>
<th>Example (radius is 5)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>less than</td>
<td>radius &lt; 0</td>
<td>false</td>
</tr>
<tr>
<td>&lt;=</td>
<td>≤</td>
<td>less than or equal to</td>
<td>radius &lt;= 0</td>
<td>false</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>greater than</td>
<td>radius &gt; 0</td>
<td>true</td>
</tr>
<tr>
<td>&gt;=</td>
<td>≥</td>
<td>greater than or equal to</td>
<td>radius &gt;= 0</td>
<td>true</td>
</tr>
<tr>
<td>==</td>
<td>=</td>
<td>equal to</td>
<td>radius == 0</td>
<td>false</td>
</tr>
<tr>
<td>!=</td>
<td>≠</td>
<td>not equal to</td>
<td>radius != 0</td>
<td>true</td>
</tr>
</tbody>
</table>

Multiple Alternative if Statements

(a)  if (score >= 90.0) 
  System.out.print(“A”); 
else 
  if (score >= 80.0) 
    System.out.print(“B”); 
  else 
    ... (score >= 60.0) 
  System.out.print(“D”); 
else 
  System.out.print(“F”);

(b) This is better

if (score >= 90.0) 
  System.out.print(“A”); 
else if (score >= 80.0) 
  System.out.print(“B”); 
else if (score >= 70.0) 
  System.out.print(“C”); 
else if (score >= 60.0) 
  System.out.print(“D”); 
else 
  System.out.print(“F”);
Multi-Way if-else Statements

Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>not</td>
<td>logical negation</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
<td>logical conjunction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>exclusive or</td>
<td>logical exclusion</td>
</tr>
</tbody>
</table>

switch Statements

```java
switch (status) {
    case 0:  compute taxes for single filers;
             break;
    case 1:  compute taxes for married file jointly;
             break;
    case 2:  compute taxes for married file separately;
             break;
    case 3:  compute taxes for head of household;
             break;
    default: System.out.println("Errors: invalid status");
             System.exit(1);
}
```
**Operator Precedence**

- ()
- var++, var--
- +, - (Unary plus and minus), ++var, --var
- (type) Casting
- ! (Not)
- *, /, % (Multiplication, division, and remainder)
- +=, -= (Binary addition and subtraction)
- <, <=, >= (Relational operators)
- ==, !=; (Equality)
- ^ (Exclusive OR)
- && (Conditional AND) Short-circuit AND
- || (Conditional OR) Short-circuit OR
- =, +=, -=, *=, /=, %= (Assignment operator)

**The Math Class**

- Class constants:
  - PI
  - E
- Class methods:
  - Trigonometric Methods
  - Exponent Methods
  - Rounding Methods
  - min, max, abs, and random Methods
### ASCII Code for Commonly Used Characters

<table>
<thead>
<tr>
<th>Characters</th>
<th>Code Value in Decimal</th>
<th>Unicode Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0' to '9'</td>
<td>48 to 57</td>
<td>\u0030 to \u0039</td>
</tr>
<tr>
<td>'A' to 'Z'</td>
<td>65 to 90</td>
<td>\u0041 to \u005A</td>
</tr>
<tr>
<td>'a' to 'z'</td>
<td>97 to 122</td>
<td>\u0061 to \u007A</td>
</tr>
</tbody>
</table>

### Escape Sequences for Special Characters

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Name</th>
<th>Unicode Code</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>Backspace</td>
<td>\u0008</td>
<td>8</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
<td>\u0009</td>
<td>9</td>
</tr>
<tr>
<td>\n</td>
<td>Linefeed</td>
<td>\u000A</td>
<td>10</td>
</tr>
<tr>
<td>\f</td>
<td>Formfeed</td>
<td>\u000C</td>
<td>12</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage Return</td>
<td>\u000D</td>
<td>13</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double Quote</td>
<td>\u0022</td>
<td>34</td>
</tr>
</tbody>
</table>

### Appendix B: ASCII Character Set

ASCII Character Set is a subset of the Unicode from \u0000 to \u007f
### Methods in the Character Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isDigit(ch)</td>
<td>Returns true if the specified character is a digit.</td>
</tr>
<tr>
<td>isLetter(ch)</td>
<td>Returns true if the specified character is a letter.</td>
</tr>
<tr>
<td>isLetterOrDigit(ch)</td>
<td>Returns true if the specified character is a letter or digit.</td>
</tr>
<tr>
<td>isLowerCase(ch)</td>
<td>Returns true if the specified character is a lowercase letter.</td>
</tr>
<tr>
<td>isUpperCase(ch)</td>
<td>Returns true if the specified character is an uppercase letter.</td>
</tr>
<tr>
<td>toLowerCase(ch)</td>
<td>Returns the lowercase of the specified character.</td>
</tr>
<tr>
<td>toUpperCase(ch)</td>
<td>Returns the uppercase of the specified character.</td>
</tr>
</tbody>
</table>

### The String Type

The char type only represents one character. To represent a string of characters, use the data type called String. For example,

```java
String message = "Welcome to Java";
```

String is actually a predefined class in the Java library just like the System class and Scanner class. The String type is not a primitive type. It is known as a **reference type**. Any Java class can be used as a reference type for a variable. Reference data types will be thoroughly discussed in Chapter 9, “Objects and Classes.” For the time being, you just need to know how to declare a String variable, how to assign a string to the variable, how to concatenate strings, and to perform simple operations for strings.

### Simple Methods for String Objects

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td>Returns the number of characters in this string.</td>
</tr>
<tr>
<td>charAt(index)</td>
<td>Returns the character at the specified index from this string.</td>
</tr>
<tr>
<td>concat(s1)</td>
<td>Returns a new string that concatenates this string with string s1.</td>
</tr>
<tr>
<td>toLowerCase()</td>
<td>Returns a new string with all letters in lowercase.</td>
</tr>
<tr>
<td>toUpperCase()</td>
<td>Returns a new string with all letters in uppercase.</td>
</tr>
<tr>
<td>trim()</td>
<td>Returns a new string with whitespace characters trimmed on both sides.</td>
</tr>
</tbody>
</table>
Reading Numbers from the Keyboard

```java
Scanner input = new Scanner(System.in);
int value = input.nextInt();
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nextByte()</td>
<td>reads an integer of the byte type.</td>
</tr>
<tr>
<td>nextShort()</td>
<td>reads an integer of the short type.</td>
</tr>
<tr>
<td>nextInt()</td>
<td>reads an integer of the int type.</td>
</tr>
<tr>
<td>nextLong()</td>
<td>reads an integer of the long type.</td>
</tr>
<tr>
<td>nextFloat()</td>
<td>reads a number of the float type.</td>
</tr>
<tr>
<td>nextDouble()</td>
<td>reads a number of the double type.</td>
</tr>
</tbody>
</table>

### while Loop

```java
int count = 0;
while (count < 100) {
    System.out.println("Welcome to Java!");
    count++;
}
```

### do-while Loop

```java
do {
    // Loop body;
    Statement(s);
} while (loop-continuation-condition);
```
for Loops

```
for (initial-action; loop-continuation-condition; action-after-each-iteration) {
    // loop body;
    Statement(s);
}
```

```
int i;
for (i = 0; i < 100; i++) {
    System.out.println("Welcome to Java!");
}
```

Using `break` and `continue`

Examples for using the `break` and `continue` keywords:

- TestBreak.java

  ```java
  public class TestBreak {
    public static void main(String[] args) {
      int sum = 0;
      int number = 0;
      while (number < 20) {
        number++;
        sum += number;
        if (sum >= 100)
          break;
      }
      System.out.println("The number is ", number);
      System.out.println("The sum is ", sum);
    }
  }
  ```

- TestContinue.java

  ```java
  public class TestContinue { 
    public static void main(String[] args) { 
      int sum = 0; 
      int number = 0; 
      ... 
    } 
    System.out.println("The number is ", number); 
    System.out.println("The sum is ", sum); 
  } 
  ```

`break`

```
public class TestBreak { 
  public static void main(String[] args) { 
    int sum = 0; 
    int number = 0; 
    while (number < 20) { 
      number++;
      sum += number;
      if (sum >= 100) 
        break; 
    }
    System.out.println("The number is ", number); 
    System.out.println("The sum is ", sum); 
  } 
```
public class TestContinue {
  public static void main(String[] args) {
    int sum = 0;
    int number = 0;
    while (number < 20) {
      number++;
      if (number == 10 || number == 11)
        continue;
      sum += number;
    }
    System.out.println("The sum is "+sum);
  }
}

Formatting Output

Use the printf statement.

System.out.printf(format, items);

Where format is a string that may consist of substrings and format specifiers. A format specifier specifies how an item should be displayed. An item may be a numeric value, character, boolean value, or a string. Each specifier begins with a percent sign.

Frequently-Used Specifiers

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Output</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%b</td>
<td>a boolean value</td>
<td>true or false</td>
</tr>
<tr>
<td>%c</td>
<td>a character</td>
<td>'a'</td>
</tr>
<tr>
<td>%d</td>
<td>a decimal integer</td>
<td>200</td>
</tr>
<tr>
<td>%f</td>
<td>a floating-point number</td>
<td>45.460000</td>
</tr>
<tr>
<td>%e</td>
<td>a number in standard scientific notation</td>
<td>4.556000e+01</td>
</tr>
<tr>
<td>%s</td>
<td>a string</td>
<td>&quot;Java is cool&quot;</td>
</tr>
</tbody>
</table>

int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);

display count is 5 and amount is 45.560000
Formal Parameters

The variables defined in the method header are known as **formal parameters**.

```
public static int max(int num1, int num2) {
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```

**Actual Parameters**

When a method is invoked, you pass a value to the parameter. This value is referred to as **actual parameter or argument**.

```
int z = max(x, y);
```

**Return Value Type**

A method may return a value. The **returnValueType** is the data type of the value the method returns. If the method does not return a value, the returnValueType is the keyword `void`. For example, the returnValueType in the main method is `void`.

```
int z = max(x, y);
```
Calling Methods, cont.

public static void main(String[] args) {
    int a = 1;
    int b = 2;
    System.out.println("The maximum between " + a + " and " + b + " is " + (a > b ? a : b));
}

Scope of Local Variables

A local variable: a variable defined inside a method.

Scope: the part of the program where the variable can be referenced.

The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared before it can be used.

Introducing Arrays

Array is a data structure that represents a collection of the same types of data.

double[] myList = new double[10];

Arrays reference variable

Array element at index 5

Element value
Declaring, creating, initializing
Using the Shorthand Notation

double[] myList = {1.9, 2.9, 3.4, 3.5};

This shorthand notation is equivalent to the following statements:

double[] myList = new double[4];
myList[0] = 1.9;
myList[1] = 2.9;
myList[2] = 3.4;
myList[3] = 3.5;

Passing Arrays to Methods

public static void printArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
        System.out.print(array[i] + " ");
    }
}

Invoke the method
int[] list = {3, 1, 2, 6, 4, 2};
printArray(list);

Invoke the method
printArray(new int[]{3, 1, 2, 6, 4, 2});

Anonymous array

Passing Arrays as Arguments

- Objective: Demonstrate differences of passing primitive data type variables and array variables.
Enhanced for Loop (for-each loop)

JDK 1.5 introduced a new for loop that enables you to traverse the complete array sequentially without using an index variable. For example, the following code displays all elements in the array myList:

```java
for (double value: myList)
    System.out.println(value);
```

In general, the syntax is

```java
for (elementType value: arrayRefVar) {
    // Process the value
}
```

You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.

The Arrays.toString(list) Method

The Arrays.toString(list) method can be used to return a string representation for the list.

Linear Search

The linear search approach compares the key element, key, sequentially with each element in the array list. The method continues to do so until the key matches an element in the list or the list is exhausted without a match being found. If a match is made, the linear search returns the index of the element in the array that matches the key. If no match is found, the search returns -1.
### Linear Search Animation

<table>
<thead>
<tr>
<th>Key</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5 4 1 9 7 3 2 8</td>
</tr>
<tr>
<td>3</td>
<td>6 4 1 9 7 3 2 8</td>
</tr>
<tr>
<td>2</td>
<td>6 4 1 9 7 3 2 8</td>
</tr>
<tr>
<td>1</td>
<td>6 4 1 9 7 3 2 8</td>
</tr>
<tr>
<td>0</td>
<td>6 4 1 9 7 3 2 8</td>
</tr>
</tbody>
</table>

### Binary Search, cont.

Consider the following three cases:

- If the key is less than the middle element, you only need to search the key in the first half of the array.
- If the key is equal to the middle element, the search ends with a match.
- If the key is greater than the middle element, you only need to search the key in the second half of the array.
Selection Sort

Selection sort finds the smallest number in the list and places it first. It then finds the smallest number remaining and places it second, and so on until the list contains only a single number.

Motivations

Thus far, you have used one-dimensional arrays to model linear collections of elements. You can use a two-dimensional array to represent a matrix or a table. For example, the following table that describes the distances between the cities can be represented using a two-dimensional array.

<table>
<thead>
<tr>
<th>Distance Table (in miles)</th>
<th>Chicago</th>
<th>Boston</th>
<th>New York</th>
<th>Atlanta</th>
<th>Miami</th>
<th>Dallas</th>
<th>Houston</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago</td>
<td>0</td>
<td>983</td>
<td>747</td>
<td>714</td>
<td>1375</td>
<td>967</td>
<td>1087</td>
</tr>
<tr>
<td>Boston</td>
<td>983</td>
<td>0</td>
<td>214</td>
<td>1102</td>
<td>1549</td>
<td>1723</td>
<td>1842</td>
</tr>
<tr>
<td>New York</td>
<td>747</td>
<td>214</td>
<td>0</td>
<td>888</td>
<td>1549</td>
<td>1548</td>
<td>1627</td>
</tr>
<tr>
<td>Atlanta</td>
<td>714</td>
<td>1102</td>
<td>888</td>
<td>0</td>
<td>661</td>
<td>781</td>
<td>810</td>
</tr>
<tr>
<td>Miami</td>
<td>1375</td>
<td>1763</td>
<td>1549</td>
<td>661</td>
<td>0</td>
<td>1426</td>
<td>1187</td>
</tr>
<tr>
<td>Dallas</td>
<td>967</td>
<td>1723</td>
<td>1548</td>
<td>781</td>
<td>1426</td>
<td>0</td>
<td>239</td>
</tr>
<tr>
<td>Houston</td>
<td>1087</td>
<td>1842</td>
<td>1627</td>
<td>810</td>
<td>1187</td>
<td>239</td>
<td>0</td>
</tr>
</tbody>
</table>

Declaring, Creating, and Initializing Using Shorthand Notations

You can also use an array initializer to declare, create and initialize a two-dimensional array. For example,

```java
int[][] array = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9},
    {10, 11, 12}
};
```

Same as

```java
int[][] array = new int[4][3];
array[0][0] = 1; array[0][1] = 2; array[0][2] = 3;
array[1][0] = 4; array[1][1] = 5; array[1][2] = 6;
array[2][0] = 7; array[2][1] = 8; array[2][2] = 9;
array[3][0] = 10; array[3][1] = 11; array[3][2] = 12;
```
Lengths of Two-dimensional Arrays, cont.

```java
int[][] array = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9},
    {10, 11, 12}
};

array[4].length    // ArrayIndexOutOfBoundsException
```

Classes

```java
class Circle {
    /** The radius of this circle */
    double radius = 1.0;

    /** Construct a circle object */
    Circle() {
    }

    /** Construct a circle object */
    Circle(double newRadius) {
        radius = newRadius;
    }

    /** Return the area of this circle */
    double getArea() {
        return radius * radius * 3.14159;
    }
}
```

UML Class Diagram

- Class name: Circle
- Data fields: radius
- Constructors: Circle(), Circle(double)
- Methods: getArea()
Constructors

```java
Circle() {
}
Circle(double newRadius) {
    radius = newRadius;
}
```

Constructors are a special kind of methods that are invoked to construct objects.

Declaring/Creating Objects in a Single Step

```java
ClassName objectRefVar = new ClassName();
```

Example:
```java
Circle myCircle = new Circle();
```

Accessing Object’s Members

- Referencing the object’s data:
  ```java
  objectRefVar.data
  e.g., myCircle.radius
  ```

- Invoking the object’s method:
  ```java
  objectRefVar.methodName(arguments)
  e.g., myCircle.getArea()
  ```
Differences between Variables of Primitive Data Types and Object Types

<table>
<thead>
<tr>
<th>Primitive type</th>
<th>int i = 1</th>
<th>Primitive type int i = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object type</td>
<td>Circle c</td>
<td>Object type Circle c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c: Circle</td>
</tr>
<tr>
<td></td>
<td>radius = 1</td>
<td>Created using new Circle()</td>
</tr>
</tbody>
</table>

Instance Variables, and Methods

Instance variables belong to a specific instance.
Instance methods are invoked by an instance of the class.
Instance variables and methods are specified by omitting the static keyword.

Static Variables, Constants, and Methods

Static variables are shared by all the instances of the class.
Static methods are not tied to a specific object.
Static constants are final variables shared by all the instances of the class.
Visibility Modifiers and Accessor/Mutator Methods

By default, the class, variable, or method can be accessed by any class in the same package.

- **public**
  The class, data, or method is visible to any class in any package.

- **private**
  The data or methods can be accessed only by the declaring class.
  The get and set methods are used to read and modify private properties.

Array of Objects, cont.

```java
Circle[] circleArray = new Circle[10];
```

The this Keyword

- The **this** keyword is the name of a reference that refers to an object itself. One common use of the this keyword is reference a class’s *hidden data fields*.

- Another common use of the this keyword to enable a constructor to invoke another constructor of the same class.
Checked Exceptions vs. Unchecked Exceptions

`RuntimeException`, `Error` and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

Declaring Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

```java
public void myMethod() throws IOException
public void myMethod() throws IOException, OtherException
```

Throwing Exceptions

When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```java
throw new TheException();

TheException ex = new TheException(); throw ex;
```
Catching Exceptions

```java
try {
    statements; // Statements that may throw exceptions
} catch (Exception1 exVar1) {
    handler for exception1;
} catch (Exception2 exVar2) {
    handler for exception2;
}...
catch (ExceptionN exVarN) {
    handler for exceptionN;
}
```

The File Class

The `File` class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion. The filename is a string. The `File` class is a wrapper class for the file name and its directory path.

Obtaining file metadata and manipulating files
Text I/O

A File object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file. In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file. This section introduces how to read/write strings and numeric values from/to a text file using the Scanner and PrintWriter classes.

Writing Data Using PrintWriter

```java
java.io.PrintWriter

- PrintWriter(String filename: String)
- print(String s): void
- print(char c): void
- print(char[] cArray): void
- print(int i): void
- print(long l): void
- print(float f): void
- print(double d): void
- print(boolean b): void

Also contains the overloaded println methods.

A println method acts like a print method; additionally it prints a line separator. The line separator string is defined by the system. It is \r\n on Windows and \n on Unix.

The printf method was introduced in §4.6, “Formatting Console Output and Strings.”
```

Reading Data Using Scanner

```java
java.util.Scanner

- Scanner(String source: String)
- next(): String
- nextByte(): byte
- nextShort(): short
- nextInt(): int
- nextLong(): long
- nextFloat(): float
- nextDouble(): double
- useDelimiter(String): Scanner

- hasNext(): boolean

Creates a Scanner object to read data from the specified file.
Creates a Scanner object to read data from the specified string.
Returns true if this scanner has another token in its input.
Returns next token as a string.
Returns next token as a byte.
Returns next token as a short.
Returns next token as an int.
Returns next token as a long.
Returns next token as a float.
Returns next token as a double.
Sets this scanner’s delimiting pattern.
```

Run WriteData

Run ReadData
**Abstract Classes and Abstract Methods**

An abstract class is a class that contains at least one abstract method. An abstract method is a method that has a declaration but no implementation.

**Define an Interface**

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```java
public interface InterfaceName {
    constant declarations;
    abstract method signatures;
}
```

Example:

```java
public interface Edible {
    /** Describe how to eat */
    public abstract String howToEat();
}
```

**Interfaces vs. Abstract Classes**

In an interface, the data must be constants; an abstract class can have all types of data.

Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.
The `toString`, `equals`, and `hashCode` Methods

Each wrapper class overrides the `toString`, `equals`, and `hashCode` methods defined in the `Object` class. Since all the numeric wrapper classes and the `Character` class implement the `Comparable` interface, the `compareTo` method is implemented in these classes.

Computing Factorial

```
factorial(4) = 4 * factorial(3)
  = 4 * (3 * factorial(2))
  = 4 * (3 * (2 * factorial(1)))
  = 4 * (3 * (2 * (1 * factorial(0))))
  = 4 * (3 * (2 * 1))
  = 4 * (3 * 2)
  = 4 * (6)
  = 24
```

Trace Recursive factorial

```
Step 0: executes factorial(4)
Step 1: returns factorial(3)
Step 2: executes factorial(2)
Step 3: executes factorial(1)
Step 4: executes factorial(0)
Step 5: returns 1
Step 6: returns 1
Step 7: returns 2
Step 8: returns 6
Step 9: returns 24
```
Sierpinski Triangle
1. It begins with an equilateral triangle, which is considered to be the Sierpinski fractal of order (or level) 0, as shown in Figure (a).
2. Connect the midpoints of the sides of the triangle of order 0 to create a Sierpinski triangle of order 1, as shown in Figure (b).
3. Leave the center triangle intact. Connect the midpoints of the sides of the three other triangles to create a Sierpinski of order 2, as shown in Figure (c).
4. You can repeat the same process recursively to create a Sierpinski triangle of order 3, 4, ..., and so on, as shown in Figure (d).

Java Collection Framework hierarchy, cont.
Set and List are subinterfaces of Collection.

ArrayList and LinkedList
The ArrayList class and the LinkedList class are concrete implementations of the List interface. Which of the two classes you use depends on your specific needs. If you need to support random access through an index without inserting or removing elements from any place other than the end, ArrayList offers the most efficient collection. If, however, your application requires the insertion or deletion of elements from any place in the list, you should choose LinkedList. A list can grow or shrink dynamically. An array is fixed once it is created. If your application does not require insertion or deletion of elements, the most efficient data structure is the array.
**Insertion Sort**

```java
int[] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted
```

**Bubble Sort**

Bubble sort time: $O(n^2)$

\[
(n-1) + (n-2) + \ldots + 2 + 1 = \frac{n^2 - n}{2}
\]

**Merge Sort**
Quick Sort

Quick sort, developed by C. A. R. Hoare (1962), works as follows: The algorithm selects an element, called the pivot, in the array. Divide the array into two parts such that all the elements in the first part are less than or equal to the pivot and all the elements in the second part are greater than the pivot. Recursively apply the quick sort algorithm to the first part and then the second part.

Computational Complexity (Big O)

- $T(n)=O(1)$  // constant time
- $T(n)=O(\log n)$  // logarithmic
- $T(n)=O(n)$  // linear
- $T(n)=O(n\log n)$  // linearithmic
- $T(n)=O(n^2)$  // quadratic
- $T(n)=O(n^3)$  // cubic

Complexity Examples

http://bigocheatsheet.com/