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

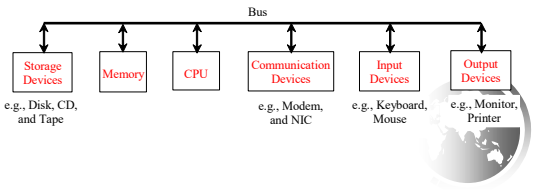
CS1: Java Programming
Colorado State University

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.









Companion Website

## 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

www.cs.armstrong.edu/liang/JavaCharacteristics.pdf

## Declaring Variables

```
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

```
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

| Name | Range | Storage Size |
|---|---|---|
| `byte` | $-2^7$ to $2^7 - 1$ (-128 to 127) | 8-bit signed |
| `short` | $-2^{15}$ to $2^{15} - 1$ (-32768 to 32767) | 16-bit signed |
| `int` | $-2^{31}$ to $2^{31} - 1$ (-2147483648 to 2147483647) | 32-bit signed |
| `long` | $-2^{63}$ to $2^{63} - 1$ (i.e., -9223372036854775808 to 9223372036854775807) | 64-bit signed |
| `float` | Negative range: -3.4028235E+38 to -1.4E-45 Positive range: 1.4E-45 to 3.4028235E+38 | 32-bit IEEE 754 |
| `double` | Negative range: -1.7976931348623157E+308 to -4.9E-324 Positive range: 4.9E-324 to 1.7976931348623157E+308 | 64-bit IEEE 754 |





# Numeric Operators

| Name | Meaning | Example | Result |
|---|---|---|---|
| `+` | Addition | `34 + 1` | `35` |
| `-` | Subtraction | `34.0 - 0.1` | `33.9` |
| `*` | Multiplication | `300 * 30` | `9000` |
| `/` | Division | `1.0 / 2.0` | `0.5` |
| `%` | Remainder | `20 % 3` | `2` |





# Integer Division

+, -, *, /, and %

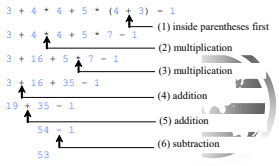
5 / 2 yields an integer 2.

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.







# 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

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

Explicit casting

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

What is wrong? int x = 5 / 2.0;

range increases

byte, short, int, long, float, double

## 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

| Java Operator | Mathematics Symbol | Name | Example (radius is 5) | Result |
|---|---|---|---|---|
| `<` | < | less than | `radius < 0` | `false` |
| `<=` | ≤ | less than or equal to | `radius <= 0` | `false` |
| `>` | > | greater than | `radius > 0` | `true` |
| `>=` | ≥ | greater than or equal to | `radius >= 0` | `true` |
| `==` | = | equal to | `radius == 0` | `false` |
| `!=` | ≠ | not equal to | `radius != 0` | `true` |




## Multiple Alternative if Statements

```
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");
```

(a)

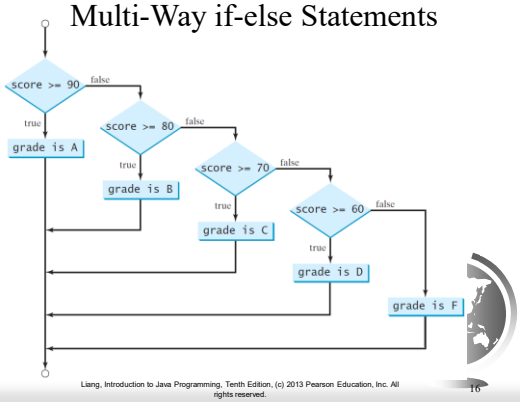
Equivalent

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");
```

(b)

# Multi-Way if-else Statements






# Logical Operators

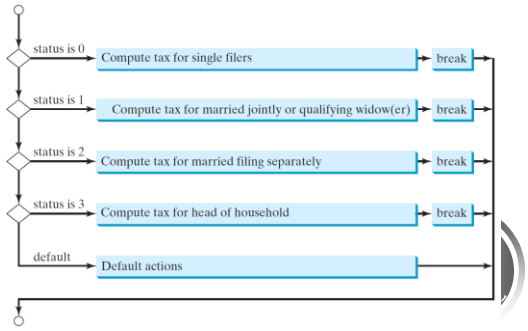
| Operator | Name | Description |
|---|---|---|
| ! | not | logical negation |
| && | and | logical conjunction |
| \|\| | or | logical disjunction |
| ^ | exclusive or | logical exclusion |



# `switch` Statements

```
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);
}
```

## `switch` Statement Flow Chart







## 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

| Characters | Code Value in Decimal | Unicode Value |
|---|---|---|
| '0' to '9' | 48 to 57 | \u0030 to \u0039 |
| 'A' to 'Z' | 65 to 90 | \u0041 to \u005A |
| 'a' to 'z' | 97 to 122 | \u0061 to \u007A |



# Escape Sequences for Special Characters

| *Escape Sequence* | *Name* | *Unicode Code* | *Decimal Value* |
|---|---|---|---|
| \b | Backspace | \u0008 | 8 |
| \t | Tab | \u0009 | 9 |
| \n | Linefeed | \u000A | 10 |
| \f | Formfeed | \u000C | 12 |
| \r | Carriage Return | \u000D | 13 |
| \\ | Backslash | \u005C | 92 |
| \" | Double Quote | \u0022 | 34 |



# Appendix B: ASCII Character Set

ASCII Character Set is a subset of the Unicode from \u0000 to \u007f

TABLE B.1 ASCII Character Set in the Decimal Index

| | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | nul | soh | stx | etx | eot | enq | ack | bel | bs | ht |
| **1** | nl | vt | ff | cr | so | si | dle | dcl | dc2 | dc3 |
| **2** | dc4 | nak | syn | etb | can | em | sub | esc | fs | gs |
| **3** | rs | us | sp | ! | " | # | $ | % | & | ' |
| **4** | ( | ) | * | + | , | - | . | / | 0 | 1 |
| **5** | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; |
| **6** | < | = | > | ? | @ | A | B | C | D | E |
| **7** | F | G | H | I | J | K | L | M | N | O |
| **8** | P | Q | R | S | T | U | V | W | X | Y |
| **9** | Z | [ | \ | ] | ^ | _ | ` | a | b | c |
| **10** | d | e | f | g | h | i | j | k | l | m |
| **11** | n | o | p | q | r | s | t | u | v | w |
| **12** | x | y | z | { | \| | } | ~ | del | | |

# Methods in the Character Class

| Method | Description |
|---|---|
| `isDigit(ch)` | Returns true if the specified character is a digit. |
| `isLetter(ch)` | Returns true if the specified character is a letter. |
| `isLetterOfDigit(ch)` | Returns true if the specified character is a letter or digit. |
| `isLowerCase(ch)` | Returns true if the specified character is a lowercase letter. |
| `isUpperCase(ch)` | Returns true if the specified character is an uppercase letter. |
| `toLowerCase(ch)` | Returns the lowercase of the specified character. |
| `toUpperCase(ch)` | Returns the uppercase of the specified character. |

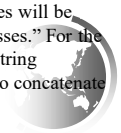


# The String Type

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

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

| Method | Description |
|---|---|
| `length()` | Returns the number of characters in this string. |
| `charAt(index)` | Returns the character at the specified index from this string. |
| `concat(s1)` | Returns a new string that concatenates this string with string s1. |
| `toUpperCase()` | Returns a new string with all letters in uppercase. |
| `toLowerCase()` | Returns a new string with all letters in lowercase. |
| `trim()` | Returns a new string with whitespace characters trimmed on both sides. |

# Reading Numbers from the Keyboard

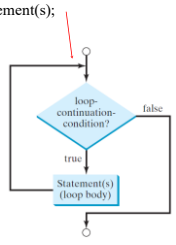
```
Scanner input = new Scanner(System.in);
int value = input.nextInt();
```

| Method | Description |
|---|---|
| `nextByte()` | reads an integer of the `byte` type. |
| `nextShort()` | reads an integer of the `short` type. |
| `nextInt()` | reads an integer of the `int` type. |
| `nextLong()` | reads an integer of the `long` type. |
| `nextFloat()` | reads a number of the `float` type. |
| `nextDouble()` | reads a number of the `double` type. |

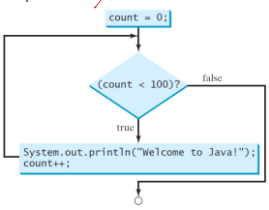



# `while` Loop

```
while (loop-continuation-condition) {
  // loop-body;
  Statement(s);
}
```



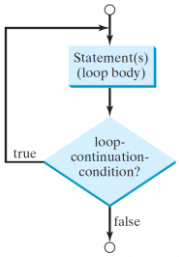

```
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java!");
  count++;
}
```







# `do-while` Loop

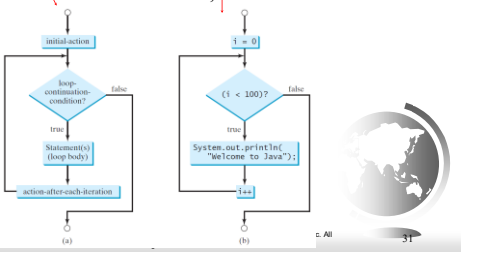



```
do {
  // Loop body;
  Statement(s);
} while (loop-continuation-condition);
```

# 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

  TestBreak Run

- TestContinue.java

  TestContinue Run





# 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);
  }
}
```

## continue

```
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

| Specifier | Output | Example |
|---|---|---|
| %b | a boolean value | true or false |
| %c | a character | 'a' |
| %d | a decimal integer | 200 |
| %f | a floating-point number | 45.460000 |
| %e | a number in standard scientific notation | 4.556000e+01 |
| %s | a string | "Java is cool" |

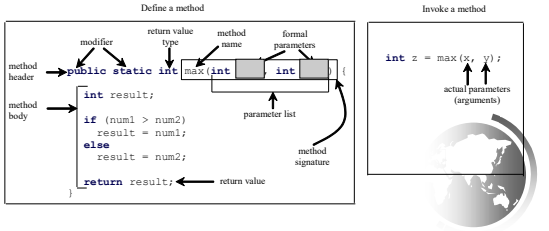
```
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);

display            count is 5 and amount is 45.560000
```

items

# Formal Parameters

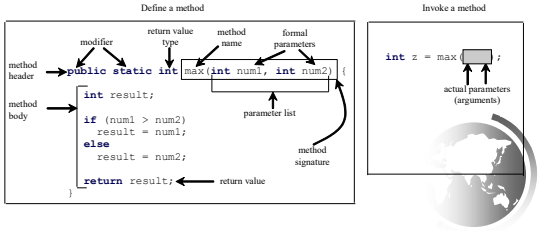
The variables defined in the method header are known as *formal parameters*.






# Actual Parameters

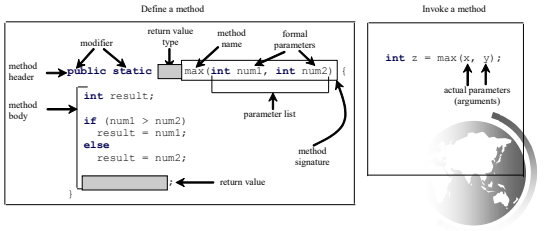
When a method is invoked, you pass a value to the parameter. This value is referred to as *actual parameter or argument*.





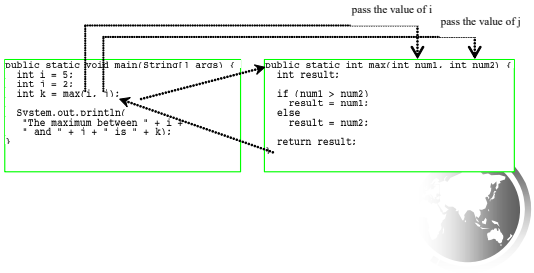

# Return Value Type

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.



animation

## Calling Methods, cont.






## Scope of Local Variables

A local variable: a variable defined inside a method.

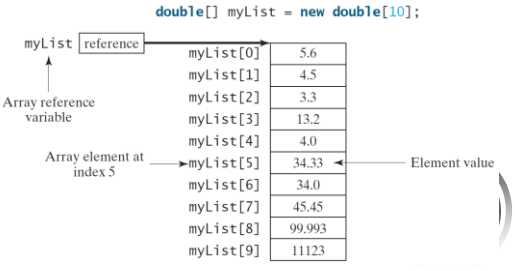
Scope: the part of the program where the variable can be referenced.

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.



## 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.

TestPassArray Run

## 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:

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

In general, the syntax is

```
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.

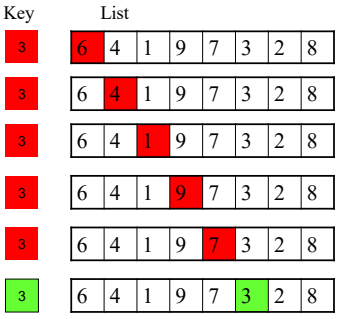



## Linear Search

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.

animation

## Linear Search Animation







## Binary Search, cont.

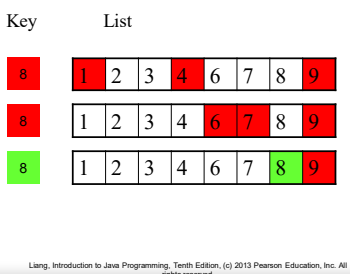
Consider the following three cases:

- 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.



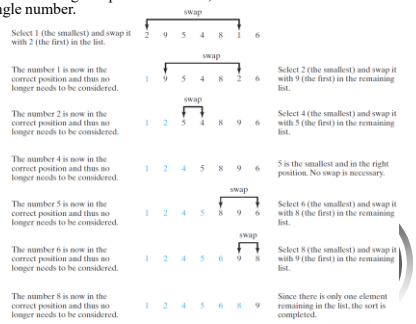

animation

## Binary Search



## 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.

Distance Table (in miles)

| | Chicago | Boston | New York | Atlanta | Miami | Dallas | Houston |
|---|---|---|---|---|---|---|---|
| Chicago | 0 | 983 | 787 | 714 | 1375 | 967 | 1087 |
| Boston | 983 | 0 | 214 | 1102 | 1763 | 1723 | 1842 |
| New York | 787 | 214 | 0 | 888 | 1549 | 1548 | 1627 |
| Atlanta | 714 | 1102 | 888 | 0 | 661 | 781 | 810 |
| Miami | 1375 | 1763 | 1549 | 661 | 0 | 1426 | 1187 |
| Dallas | 967 | 1723 | 1548 | 781 | 1426 | 0 | 239 |
| Houston | 1087 | 1842 | 1627 | 810 | 1187 | 239 | 0 |



## 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,

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

Same as

```
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.

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

array.length
array[0].length
array[1].length
array[2].length
array[3].length

array[4].length ArrayIndexOutOfBoundsException



# Classes

```
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;
  }
}
```

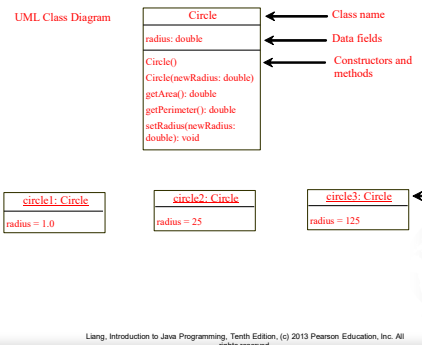
Data field

Constructors

Method



# UML Class Diagram




## Constructors

```
Circle() {
}
```

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

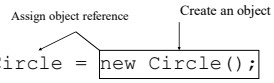
```
Circle(double newRadius) {
  radius = newRadius;
}
```



## Declaring/Creating Objects in a Single Step

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

Example:






## Accessing Object's Members

- Referencing the object's data:

  `objectRefVar.data`

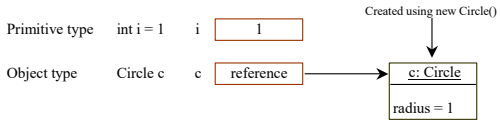
  *e.g.,* `myCircle.radius`

- Invoking the object's method:

  `objectRefVar.methodName(arguments)`

  *e.g.,* `myCircle.getArea()`

## Differences between Variables of Primitive Data Types and Object Types







## 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`

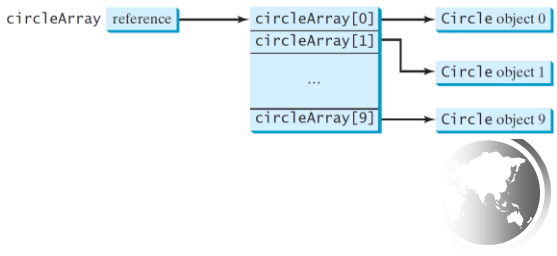
  The data or methods can be accessed only by the declaring class.

The get and set methods are used to read and modify private properties.



## Array of Objects, cont.

```
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*.

```
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,

```
throw new TheException();

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

# Catching Exceptions

```
try {
  statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVar3) {
  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 properties and manipulating file

| java.io.File | |
|---|---|
| +File(pathname: String) | Creates a File object for the specified path name. The path name may be a directory or a file. |
| +File(parent: String, child: String) | Creates a File object for the child under the directory parent. The child may be a file name or a subdirectory. |
| +File(parent: File, child: String) | Creates a File object for the child under the directory parent. The parent is a File object. In the preceding constructor, the parent is a string. |
| +exists(): boolean | Returns true if the file or the directory represented by the File object exists. |
| +canRead(): boolean | Returns true if the file represented by the File object exists and can be read. |
| +canWrite(): boolean | Returns true if the file represented by the File object exists and can be written. |
| +isDirectory(): boolean | Returns true if the File object represents a directory. |
| +isFile(): boolean | Returns true if the File object represents a file. |
| +isAbsolute(): boolean | Returns true if the File object is created using an absolute path name. |
| +isHidden(): boolean | Returns true if the file represented in the File object is hidden. The exact definition of *hidden* is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period(.) character. |
| +getAbsolutePath(): String | Returns the complete absolute file or directory name represented by the File object. |
| +getCanonicalPath(): String | Returns the same as getAbsolutePath() except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows). |
| +getName(): String | Returns the last name of the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getName() returns test.dat. |
| +getPath(): String | Returns the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getPath() returns c:\book\test.dat. |
| +getParent(): String | Returns the complete parent directory of the current directory or the file represented by the File object. For example, new File("c:\\book\\test.dat").getParent() returns c:\book. |
| +lastModified(): long | Returns the time that the file was last modified. |
| +length(): long | Returns the size of the file, or 0 if it does not exist or if it is a directory. |
| +listFile(): File[] | Returns the files under the directory for a directory File object. |
| +delete(): boolean | Deletes the file or directory represented by this File object. The method returns true if the deletion succeeds. |
| +renameTo(dest: File): boolean | Renames the file or directory represented by this File object to the specified name represented in dest. The method returns true if the operation succeeds. |
| +mkdir(): boolean | Creates a directory represented in this File object. Returns true if the the directory is created successfully. |
| +mkdirs(): boolean | Same as mkdir() except that it creates directory along with its parent directories if the parent directories do not exist. |

# 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.io.PrintWriter | |
|---|---|
| +PrintWriter(filename: String) | Creates a PrintWriter for the specified file. |
| +print(s: String): void | Writes a string. |
| +print(c: char): void | Writes a character. |
| +print(cArray: char[]): void | Writes an array of character. |
| +print(i: int): void | Writes an int value. |
| +print(l: long): void | Writes a long value. |
| +print(f: float): void | Writes a float value. |
| +print(d: double): void | Writes a double value. |
| +print(b: boolean): void | Writes a boolean value. |
| 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. |
| Also contains the overloaded printf methods. | The printf method was introduced in §4.6, "Formatting Console Output and Strings." |

WriteData Run

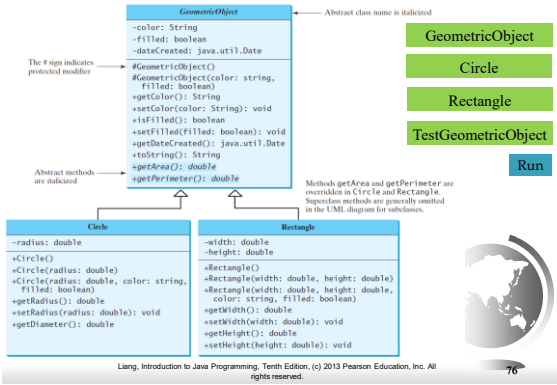



# Reading Data Using Scanner

| java.util.Scanner | |
|---|---|
| +Scanner(source: File) | Creates a Scanner object to read data from the specified file. |
| +Scanner(source: String) | Creates a Scanner object to read data from the specified string. |
| +close() | Closes this scanner. |
| +hasNext(): boolean | Returns true if this scanner has another token in its input. |
| +next(): String | Returns next token as a string. |
| +nextByte(): byte | Returns next token as a byte. |
| +nextShort(): short | Returns next token as a short. |
| +nextInt(): int | Returns next token as an int. |
| +nextLong(): long | Returns next token as a long. |
| +nextFloat(): float | Returns next token as a float. |
| +nextDouble(): double | Returns next token as a double. |
| +useDelimiter(pattern: String): Scanner | Sets this scanner's delimiting pattern. |

ReadData Run

# Abstract Classes and Abstract Methods




GeometricObject

Circle

Rectangle

TestGeometricObject

Run





# Define an Interface

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

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

Example:

```
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.

| | *Variables* | *Constructors* | *Methods* |
|---|---|---|---|
| Abstract class | No restrictions. | Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator. | No restrictions. |
| Interface | All variables must be `public static final`. | No constructors. An interface cannot be instantiated using the new operator. | All methods must be public abstract instance 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.



animation

# Computing Factorial

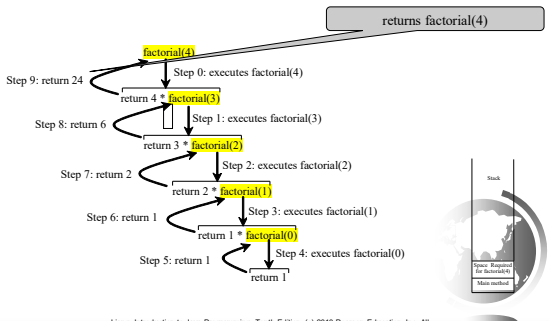
factorial(0) = 1;
factorial(n) = n*factorial(n-1);

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



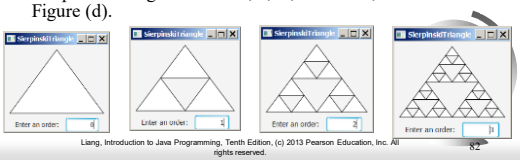
animation

# Trace Recursive factorial



# Sierpinski Triangle

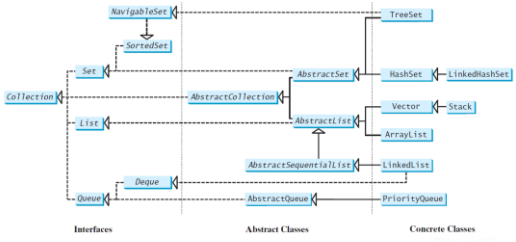
1. It begins with an equilateral triangle, which is considered to be the Sierpinski fractal of order (or level) 0, as shown in Figure (a).
2. Connect the midpoints of the sides of the triangle of order 0 to create a Sierpinski triangle of order 1, as shown in Figure (b).
3. Leave the center triangle 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.







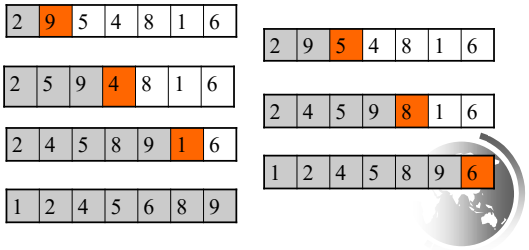
# ArrayList and LinkedList

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.

animation

# Insertion Sort

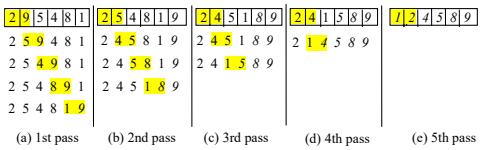
int[] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted








# Bubble Sort




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

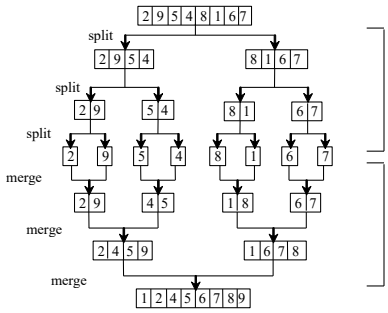
$$(n-1)+(n-2)+\ldots+2+1=\frac{n^2}{2}-\frac{n}{2}$$

BubbleSort Run





# Merge Sort




MergeSort Run

# 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.





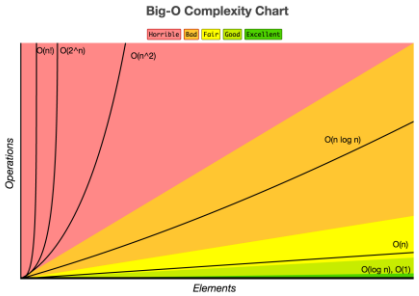
# Computational Complexity (Big O)

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





# Complexity Examples




http://bigocheatsheet.com/