Module 2

Basic Concepts

Object-oriented Programming

- Classes are the most important language feature that make *object-oriented programming* (*OOP*) possible
- Programming in Java consists of defining a number of classes
  - Every program is a class
  - All helping software consists of classes
  - All programmer-defined types are classes
- Classes are central to Java

Classes, Objects, and Methods

- A *class* is the name for a type whose values are objects
- *Objects* are entities that store data and take actions
  - Objects of the `String` class store data consisting of strings of characters
- The actions that an object can take are called *methods*
  - Methods can return a value of a single type and/or perform an action
  - All objects within a class have the same methods, but each can have different data values

Classes, Objects, and Methods

- *Invoking* or *calling a method*: a method is called into action by writing the name of the calling object, followed by a dot, followed by the method name, followed by parentheses
  - This is sometimes referred to as *sending a message to the object*
  - The parentheses contain the information (if any) needed by the method
  - This information is called an *argument* (or *arguments*)

`System.out.println`

- Java programs work by having things called *objects* perform actions
  - **`System.out`**: an object used for sending output to the screen
- The actions performed by an object are called *methods*
  - **`println`**: the method or action that the **`System.out`** object performs

A Sample Java Application

Display 1.1 A Sample Java Program

Name of class (program)

The `main` method

```
1  public class FirstProgram
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("Hello reader.");
6          System.out.println("Welcome to Java.");
7          System.out.println("Let's demonstrate a simple calculation.");
8          int answer;
9          answer = 2 + 2;
10         System.out.println("2 plus 2 is " + answer);
11     }
12 }
```

SAMPLE DIALOGUE 1

```
Hello reader.
Welcome to Java.
Let's demonstrate a simple calculation.
2 plus 2 is 4
```

System.out.println

- *Invoking* or *calling* a method: When an object performs an action using a method
  - Also called *sending a message* to the object
  - Method invocation syntax (in order): an object, a dot (period), the method name, and a pair of parentheses
  - Arguments: Zero or more pieces of information needed by the method that are placed inside the parentheses

```
System.out.println("This is an argument");
```

A Class Is a Type

- A class is a special kind of programmer-defined type, and variables can be declared of a class type
- A value of a class type is called an object or *an instance of the class*
  - If A is a class, then the phrases “x is of type A," “x is an object of the class A," and “x is an instance of the class A" mean the same thing
- A class determines the types of data that an object can contain, as well as the actions it can perform

Primitive Type Values vs. Class Type Values

- A primitive type value is a single piece of data
- A class type value or object can have multiple pieces of data, as well as actions called *methods*
  - All objects of a class have the same methods
  - All objects of a class have the same pieces of data (i.e., name, type, and number)
  - For a given object, each piece of data can hold a different value

Primitive Types

Display 1.2 Primitive Types

TYPE NAME	KIND OF VALUE	MEMORY USED	SIZE RANGE
`boolean`	`true` or `false`	1 byte	not applicable
`char`	single character (Unicode)	2 bytes	all Unicode characters
`byte`	integer	1 byte	−128 to 127
`short`	integer	2 bytes	−32768 to 32767
`int`	integer	4 bytes	−2147483648 to 2147483647
`long`	integer	8 bytes	−9223372036854775808 to 9223372036854775807
`float`	floating-point number	4 bytes	$-3.40282347 \times 10^{+38}$ to $-1.40239846 \times 10^{-45}$
`double`	floating-point number	8 bytes	$\pm 1.76769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$

The Contents of a Class Definition

- A class definition specifies the data items and methods that all of its objects will have
- These data items and methods are sometimes called *members* of the object
- Data items are called *fields* or *instance variables*
- Instance variable declarations and method definitions can be placed in any order within the class definition

Instantiation of an Object

- An object of a class is named or declared by a variable of the class type:

```
ClassName  classVar;
```

- The `new` operator must then be used to create the object and associate it with its variable name:

```
classVar = new ClassName();
```

- These can be combined as follows:

```
ClassName classVar = new ClassName();
```

Constructors

- A *constructor* is a special kind of method that is designed to initialize the instance variables for an object:

```
Public ClassName(anyParameters){code}
```

  - A constructor must have the same name as the class
  - A constructor has no type returned, not even **`void`**
  - Constructors are typically overloaded

Constructors

- A constructor is called when an object of the class is created using **`new`**

```
ClassName objectName = new ClassName(anyArgs);
```

  - The name of the constructor and its parenthesized list of arguments (if any) must follow the **`new`** operator
  - This is the **only** valid way to invoke a constructor: a constructor cannot be invoked like an ordinary method
- If a constructor is invoked again (using **`new`**), the first object is discarded and an entirely new object is created
  - If you need to change the values of instance variables of the object, use *mutator methods* instead

You Can Invoke Another Method in a Constructor

- The first action taken by a constructor is to create an object with instance variables
- Therefore, it is legal to invoke another method within the definition of a constructor, since it has the newly created object as its calling object
  - For example, mutator methods can be used to set the values of the instance variables
  - It is even possible for one constructor to invoke another

Include a No-Argument Constructor

- If you do not include any constructors in your class, Java will automatically create a *default* or *no-argument* constructor that takes no arguments, performs no initializations, but allows the object to be created
- If you include even one constructor in your class, Java will not provide this default constructor
- If you include any constructors in your class, be sure to provide your own no-argument constructor as well

Instance Variables and Methods

- Instance variables can be defined as in the following two examples
  - Note the **`public`** modifier (for now):

```
public String  instanceVar1;
public int  instanceVar2;
```

- In order to refer to a particular instance variable, preface it with its object name as follows:

```
objectName.instanceVar1
objectName.instanceVar2
```

Default Variable Initializations

- Instance variables are automatically initialized in Java
  - `boolean` types are initialized to false
  - Other primitives are initialized to the zero of their type
  - Class types are initialized to `null`
- However, it is a better practice to explicitly initialize instance variables in a constructor
- Note: Local variables are not automatically initialized

Instance Variables and Methods

- Method definitions are divided into two parts: a *heading* and a *method body:*

```
public void myMethod()
{
    code  to perform some action
    and/or compute a value
}
```

Heading

Body

- Methods are invoked using the name of the calling object and the method name as follows:

```
classVar.myMethod();
```

- Invoking a method is equivalent to executing the method body

More About Methods

- There are two kinds of methods:
  - Methods that compute and return a value
  - Methods that perform an action
    - This type of method does not return a value, and is called a `void` method
- Each type of method differs slightly in how it is defined as well as how it is (usually) invoked

More About Methods

- A method that returns a value must specify the type of that value in its heading:

```
public typeReturned methodName(paramList)
```

- A **`void`** method uses the keyword **`void`** in its heading to show that it does not return a value :

```
public void methodName(paramList)
```

Terminology Comparisons

- Other high-level languages have constructs called procedures, methods, functions, and/or subprograms
  - These types of constructs are called *methods* in Java
  - All programming constructs in Java, including *methods*, are part of a *class*

The `this` Parameter

- All instance variables are understood to have `<the calling object>.` in front of them
- If an explicit name for the calling object is needed, the keyword `this` can be used
  - `myInstanceVariable` always means and is always interchangeable with `this.myInstanceVariable`

The `this` Parameter

- `this` ***must*** be used if a parameter or other local variable with the same name is used in the method
  - Otherwise, all instances of the variable name will be interpreted as local

```
int someVariable = this.someVariable
```

local **instance**

The `this` Parameter

- The `this` parameter is a kind of hidden parameter
- Even though it does not appear on the parameter list of a method, it is still a parameter
- When a method is invoked, the calling object is automatically plugged in for `this`
- A Constructor has a `this` Parameter

Variable Declarations

- Variable declarations in Java are similar to those in other programming languages
  - Simply give the type of the variable followed by its name and a semicolon

```
int answer;
```

Variable Declarations

- Every variable in a Java program must be *declared* before it is used
  - A variable declaration tells the compiler what kind of data (type) will be stored in the variable
  - The type of the variable is followed by one or more variable names separated by commas, and terminated with a semicolon
  - Variables are typically declared just before they are used or at the start of a block (indicated by an opening brace `{`)
  - Basic types in Java are called *primitive types*

```
int numberOfBeans;
double oneWeight, totalWeight;
```

The methods **equals** and **toString**

- Java expects certain methods, such as **equals** and **toString**, to be in all, or almost all, classes
- The purpose of **equals**, a **boolean** valued method, is to compare two objects of the class to see if they satisfy the notion of "being equal"
  - Note: You cannot use **==** to compare objects

```
public boolean equals(ClassName objectName)
```

- The purpose of the **toString** method is to return a **String** value that represents the data in the object

```
public String toString()
```

Identifiers

- *Identifier*: The name of a variable or other item (class, method, object, etc.) defined in a program
  - A Java identifier must not start with a digit, and all the characters must be letters, digits, or the underscore symbol
  - Java identifiers can theoretically be of any length
  - Java is a case-sensitive language: `Rate`, `rate`, and `RATE` are the names of three different variables

Constants

- *Constant* (or *literal*): An item in Java which has one specific value that cannot change
  - Constants of an integer type may not be written with a decimal point (e.g., `10`)
  - Constants of a floating-point type can be written in ordinary decimal fraction form (e.g., `367000.0` or `0.000589`)
  - Constant of a floating-point type can also be written in *scientific* (or *floating-point*) *notation* (e.g., `3.67e5` or `5.89e-4`)
    - Note that the number before the `e` may contain a decimal point, but the number after the `e` may not

Constants

- Constants of type `char` are expressed by placing a single character in single quotes (e.g., `'Z'`)
- Constants for strings of characters are enclosed by double quotes (e.g., `"Welcome to Java"`)
- There are only two `boolean` type constants, `true` and `false`
  - Note that they must be spelled with all lowercase letters

Naming Constants

- Instead of using "anonymous" numbers in a program, always declare them as named constants, and use their name instead

```
public static final int INCHES_PER_FOOT = 12;
public static final double RATE = 0.14;
```

  - This prevents a value from being changed inadvertently
  - It has the added advantage that when a value must be modified, it need only be changed in one place
  - Note the naming convention for constants: Use all uppercase letters, and designate word boundaries with an underscore character

Expressions

- In Java, the equal sign (=) is used as the *assignment operator*
  - The variable on the left side of the assignment operator is *assigned the value* of the expression on the right side of the assignment operator

```
answer = 2 + 2;
```

- In Java, the plus sign (+) can be used to denote addition (as above) or concatenation
  - Using +, two strings can be connected together

```
System.out.println("2 plus 2 is " + answer);
```

Expressions

- In Java, the assignment statement is used to change the value of a variable
  - The equal sign (=) is used as the assignment operator
  - An assignment statement consists of a variable on the left side of the operator, and an *expression* on the right side of the operator

```
Variable = Expression;
```

  - An *expression* consists of a variable, number, or mix of variables, numbers, operators, and/or method invocations

```
temperature = 98.6;
count = numberOfBeans;
```

Expressions

- When an assignment statement is executed, the expression is first evaluated, and then the variable on the left-hand side of the equal sign is set equal to the value of the expression

```
distance = rate * time;
```

- Note that a variable can occur on both sides of the assignment operator

```
count = count + 2;
```

- The assignment operator is automatically executed from right-to-left, so assignment statements can be chained

```
number2 = number1 = 3;
```

Initializations

- A variable that has been declared but that has not yet been given a value by some means is said to be *uninitialized*
- In certain cases an uninitialized variable is given a default value
  - It is best not to rely on this
  - Explicitly initialized variables have the added benefit of improving program clarity

Initializations

- The declaration of a variable can be combined with its initialization via an assignment statement

```
int count = 0;
double distance = 55 * .5;
char grade = 'A';
```

  - Note that some variables can be initialized and others can remain uninitialized in the same declaration

```
int initialCount = 50, finalCount;
```

Shorthand Assignment Statements

- Shorthand assignment notation combines the *assignment operator* (**=**) and an *arithmetic operator*
- It is used to change the value of a variable by adding, subtracting, multiplying, or dividing by a specified value
- The general form is

Variable Op = Expression

which is equivalent to

Variable = Variable Op (Expression)

  - The **Expression** can be another variable, a constant, or a more complicated expression
  - Some examples of what **Op** can be are **+**, **-**, *****, **/**, or **%**

Shorthand Assignment Statements

Example:	Equivalent To:
`count += 2;`	`count = count + 2;`
`sum -= discount;`	`sum = sum - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time / rushFactor;`
`change %= 100;`	`change = change % 100;`
`amount *= count1 + count2;`	`amount = amount * (count1 + count2);`

Assignment Compatibility

- In general, the value of one type cannot be stored in a variable of another type

```
int intVariable = 2.99; //Illegal
```

  - The above example results in a type mismatch because a **`double`** value cannot be stored in an **`int`** variable
- However, there are exceptions to this

```
double doubleVariable = 2;
```

  - For example, an **`int`** value can be stored in a **`double`** type

Assignment Compatibility

- More generally, a value of any type in the following list can be assigned to a variable of any type that appears to the right of it

  `byte`→`short`→`int`→`long`→`float`→`double`

  `char` 

  - Note that as your move down the list from left to right, the range of allowed values for the types becomes larger
- An explicit *type cast* is required to assign a value of one type to a variable whose type appears to the left of it on the above list (e.g., **`double`** to **`int`**)
- Note that in Java an **`int`** cannot be assigned to a variable of type **`boolean`**, nor can a **`boolean`** be assigned to a variable of type **`int`**

Arithmetic Operators and Expressions

- As in most languages, *expressions* can be formed in Java using variables, constants, and arithmetic operators
  - These operators are + (addition), – (subtraction), * (multiplication), / (division), and % (modulo, remainder)
  - An expression can be used anyplace it is legal to use a value of the type produced by the expression

Arithmetic Operators and Expressions

- If an arithmetic operator is combined with **`int`** operands, then the resulting type is **`int`**
- If an arithmetic operator is combined with one or two **`double`** operands, then the resulting type is **`double`**
- If different types are combined in an expression, then the resulting type is the right-most type on the following list that is found within the expression

  `byte→short→int→long→float→double`

  `char`

  - Exception: If the type produced should be **`byte`** or **`short`** (according to the rules above), then the type produced will actually be an **`int`**

Parentheses and Precedence Rules

- An expression can be *fully parenthesized* in order to specify exactly what subexpressions are combined with each operator
- If some or all of the parentheses in an expression are omitted, Java will follow *precedence* rules to determine, in effect, where to place them
  - However, it's best (and sometimes necessary) to include them

Precedence Rules

Display 1.3 Precedence Rules

Highest Precedence

First: the unary operators: +, –, ++, ––, and !

Second: the binary arithmetic operators: *, /, and %

Third: the binary arithmetic operators: + and –

Lowest Precedence

Precedence and Associativity Rules

- When the order of two adjacent operations must be determined, the operation of higher precedence (and its apparent arguments) is grouped before the operation of lower precedence

  `base + rate * hours` is evaluated as

  `base + (rate * hours)`

- When two operations have equal precedence, the order of operations is determined by *associativity* rules

Precedence and Associativity Rules

- Unary operators of equal precedence are grouped right-to-left

  `+-+rate` is evaluated as `+(-(+rate))`

- Binary operators of equal precedence are grouped left-to-right

  `base + rate + hours` is evaluated as

  `(base + rate) + hours`

- Exception: A string of assignment operators is grouped right-to-left

  `n1 = n2 = n3;` is evaluated as `n1 = (n2 = n3);`

Integer and Floating-Point Division

- When one or both operands are a floating-point type, division results in a floating-point type

  `15.0/2` evaluates to `7.5`

- When both operands are integer types, division results in an integer type
  - Any fractional part is discarded
  - The number is not rounded

    `15/2` evaluates to `7`

- Be careful to make at least one of the operands a floating-point type if the fractional portion is needed

The % Operator

- The `%` operator is used with operands of type `int` to recover the information lost after performing integer division
  - `15/2` evaluates to the quotient `7`
  - `15%2` evaluates to the remainder `1`
- The `%` operator can be used to count by 2's, 3's, or any other number
  - To count by twos, perform the operation `number % 2`, and when the result is `0`, `number` is even

Type Casting

- A *type cast* takes a value of one type and produces a value of another type with an "equivalent" value
  - If `n` and `m` are integers to be divided, and the fractional portion of the result must be preserved, at least one of the two must be type cast to a floating-point type **before** the division operation is performed

```
double ans = n / (double)m;
```

  - Note that the desired type is placed inside parentheses immediately in front of the variable to be cast
  - Note also that the type and value of the variable to be cast does not change

More Details About Type Casting

- When type casting from a floating-point to an integer type, the number is truncated, not rounded
  - `(int)2.9` evaluates to `2`, not `3`
- When the value of an integer type is assigned to a variable of a floating-point type, Java performs an automatic type cast called a *type coercion*

```
double d = 5;
```

- In contrast, it is illegal to place a `double` value into an `int` variable without an explicit type cast

```
int i = 5.5; // Illegal
int i = (int)5.5 // Correct
```

Increment and Decrement Operators

- The *increment operator* (`++`) adds one to the value of a variable
  - If `n` is equal to `2`, then `n++` or `++n` will change the value of `n` to `3`
- The *decrement operator* (`--`) subtracts one from the value of a variable
  - If `n` is equal to `4`, then `n--` or `--n` will change the value of `n` to `3`

Increment and Decrement Operators

- When either operator precedes its variable, and is part of an expression, then the expression is evaluated using the changed value of the variable
  - If `n` is equal to `2`, then `2*(++n)` evaluates to `6`
- When either operator follows its variable, and is part of an expression, then the expression is evaluated using the original value of the variable, and only then is the variable value changed
  - If `n` is equal to `2`, then `2*(n++)` evaluates to `4`

The Class `String`

- There is no primitive type for strings in Java
- The class `String` is a predefined class in Java that is used to store and process strings
- Objects of type `String` are made up of strings of characters that are written within double quotes
  - Any quoted string is a constant of type `String`

```
"Live long and prosper."
```

- A variable of type `String` can be given the value of a `String` object

```
String blessing = "Live long and
  prosper.";
```

Concatenation of Strings

- *Concatenation*: Using the `+` operator on two strings in order to connect them to form one longer string
  - If `greeting` is equal to `"Hello "`, and `javaClass` is equal to `"class"`, then `greeting + javaClass` is equal to `"Hello class"`
- Any number of strings can be concatenated together
- When a string is combined with almost any other type of item, the result is a string
  - `"The answer is " + 42` evaluates to `"The answer is 42"`

String Methods

- The `String` class contains many useful methods for string-processing applications
  - A `String` method is called by writing a `String` object, a dot, the name of the method, and a pair of parentheses to enclose any arguments
  - If a `String` method returns a value, then it can be placed anywhere that a value of its type can be used

```
String greeting = "Hello";
int count = greeting.length();
System.out.println("Length is " +
  greeting.length());
```

  - Always count from zero when referring to the *position* or *index* of a character in a string

Some Methods in the Class `String`

Display 1.4 Some Methods in the Class `String`

`int length()`

Returns the length of the calling object (which is a string) as a value of type `int`.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.length()` returns 6.

`boolean equals(Other_String)`

Returns `true` if the calling object string and the *Other_String* are equal. Otherwise, returns `false`.

EXAMPLE

After program executes `String greeting = "Hello";`
`greeting.equals("Hello")` returns `true`
`greeting.equals("Good-Bye")` returns `false`
`greeting.equals("hello")` returns `false`

Note that case matters. `"Hello"` and `"hello"` are not equal because one starts with an uppercase letter and the other starts with a lowercase letter.

(continued)

Some Methods in the Class `String`

Display 1.4 Some Methods in the Class `String`

`boolean equalsIgnoreCase(`*Other_String*`)`

Returns `true` if the calling object string and the *Other_String* are equal, considering uppercase and lowercase versions of a letter to be the same. Otherwise, returns `false`.

EXAMPLE

After program executes `String name = "mary!";`
`greeting.equalsIgnoreCase("Mary!")` returns `true`

`String toLowerCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to lowercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toLowerCase()` returns `"hi mary!"`.

(continued)

Some Methods in the Class `String`

Display 1.4 Some Methods in the Class `String`

`String toUpperCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to uppercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toUpperCase()` returns `"HI MARY!"`.

`String trim()`

Returns a string with the same characters as the calling object string, but with leading and trailing white space removed. Whitespace characters are the characters that print as white space on paper, such as the blank (space) character, the tab character, and the new-line character `'\n'`.

EXAMPLE

After program executes `String pause = "   Hmm   ";`
`pause.trim()` returns `"Hmm"`.

(continued)

Some Methods in the Class `String`

Display 1.4 Some Methods in the Class `String`

`char charAt(`*Position*`)`

Returns the character in the calling object string at the *Position*. Positions are counted 0, 1, 2, etc.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.charAt(0)` returns `'H'`, and
`greeting.charAt(1)` returns `'e'`.

`String substring(`*Start*`)`

Returns the substring of the calling object string starting from *Start* through to the end of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2)` returns `"cdefG"`.

(continued)

Some Methods in the Class `String`

Display 1.4 Some Methods in the Class `String`

`String substring(`*Start*`, `*End*`)`

Returns the substring of the calling object string starting from position *Start* through, but not including, position *End* of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned, but the character at position *End* is not included.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2, 5)` returns `"cde"`.

`int indexOf(`*A_String*`)`

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns −1 if *A_String* is not found.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.indexOf("Mary")` returns 3, and
`greeting.indexOf("Sally")` returns −1.

(continued)

Some Methods in the Class `String`

Display 1.4 Some Methods in the Class `String`

`int indexOf(A_String, Start)`

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string that occurs at or after position *Start*. Positions are counted 0, 1, 2, etc. Returns −1 if *A_String* is not found.

EXAMPLE

After program executes `String name = "Mary, Mary quite contrary";`
`name.indexOf("Mary", 1)` returns 6.
The same value is returned if 1 is replaced by any number up to and including 6.
`name.indexOf("Mary", 0)` returns 0.
`name.indexOf("Mary", 8)` returns −1.

`int lastIndexOf(A_String)`

Returns the index (position) of the last occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns −1, if *A_String* is not found.

EXAMPLE

After program executes `String name = "Mary, Mary, Mary quite so";`
`greeting.indexOf("Mary")` returns 0, and
`name.lastIndexOf("Mary")` returns 12.

(continued)

Some Methods in the Class `String`

Display 1.4 Some Methods in the Class `String`

`int compareTo(A_String)`

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering. Lexicographic order is the same as alphabetical order but with the characters ordered as in Appendix 3. Note that in Appendix 3 all the uppercase letters are in regular alphabetical order and all the lowercase letters are in alphabetical order, but all the uppercase letters precede all the lowercase letters. So, lexicographic ordering is the same as alphabetical ordering provided both strings are either all uppercase letters or both strings are all lowercase letters. If the calling string is first, it returns a negative value. If the two strings are equal, it returns zero. If the argument is first, it returns a positive number.

EXAMPLE

After program executes `String entry = "adventure";`
`entry.compareTo("zoo")` returns a negative number,
`entry.compareTo("adventure")` returns `0`, and
`entry.compareTo("above")` returns a positive number.

(continued)

Some Methods in the Class `String`

Display 1.4 Some Methods in the Class `String`

`int compareToIgnoreCase(A_String)`

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering, treating uppercase and lowercase letters as being the same. (To be precise, all uppercase letters are treated as if they were their lowercase versions in doing the comparison.) Thus, if both strings consist entirely of letters, the comparison is for ordinary alphabetical order. If the calling string is first, it returns a negative value. If the two strings are equal ignoring case, it returns zero. If the argument is first, it returns a positive number.

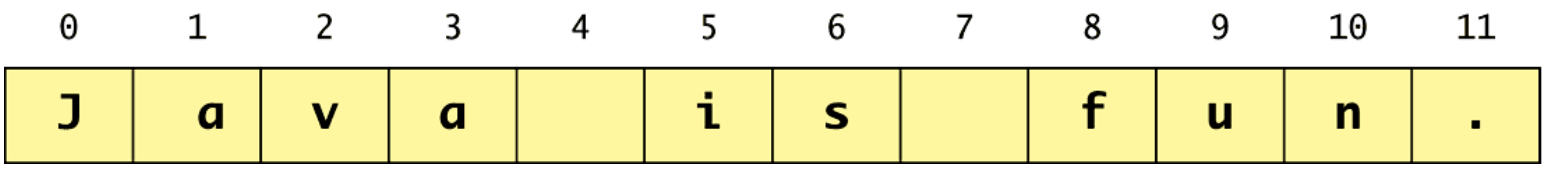
EXAMPLE

After program executes `String entry = "adventure";`
`entry.compareToIgnoreCase("Zoo")` returns a negative number,
`entry.compareToIgnoreCase("Adventure")` returns `0`, and
`"Zoo".compareToIgnoreCase(entry)` returns a positive number.

String Indexes

Display 1.5 String Indexes

The 12 characters in the string `"Java is fun."` have indexes `0` through `11`.



Notice that the blanks and the period count as characters in the string.

Escape Sequences

- A backslash (\) immediately preceding a character (i.e., without any space) denotes an *escape sequence* or an *escape character*
  - The character following the backslash does not have its usual meaning
  - Although it is formed using two symbols, it is regarded as a single character

Escape Sequences

Display 1.6 Escape Sequences

```
\" Double quote.
\' Single quote.
\\ 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. White space up to the next tab stop.
```

String Processing

- A `String` object in Java is considered to be immutable, i.e., the characters it contains cannot be changed
- There is another class in Java called `StringBuffer` that has methods for editing its string objects
- However, it is possible to change the value of a `String` variable by using an assignment statement

```
String name = "Soprano";
name = "Anthony " + name;
```

Character Sets

- *ASCII*: A character set used by many programming languages that contains all the characters normally used on an English-language keyboard, plus a few special characters
  - Each character is represented by a particular number
- *Unicode*: A character set used by the Java language that includes all the ASCII characters plus many of the characters used in languages with a different alphabet from English

`main` is a `void` Method

- A program in Java is just a class that has a `main` method
- When you give a command to run a Java program, the run-time system invokes the method `main`
- Note that `main` is a `void` method, as indicated by its heading:

```
public static void main(String[] args)
```

`return` Statements

- The body of both types of methods contains a list of declarations and statements enclosed in a pair of braces

```
public <void or typeReturned> myMethod()
{
    declarations
    statements
}
```

Body

`return` Statements

- The body of a method that returns a value must also contain one or more `return` statements
  - A `return` statement specifies the value returned and ends the method invocation:

```
return Expression;
```

  - `Expression` can be any expression that evaluates to something of the type returned listed in the method heading

`return` Statements

- A `void` method need not contain a `return` statement, unless there is a situation that requires the method to end before all its code is executed
- In this context, since it does not return a value, a `return` statement is used without an expression:

```
return;
```

Method Definitions

- An invocation of a method that returns a value can be used as an expression anyplace that a value of the `typeReturned` can be used:

```
typeReturned tRVariable;
tRVariable =
  objectName.methodName();
```

- An invocation of a `void` method is simply a statement:

```
objectName.methodName();
```

Any Method Can Be Used As a `void` Method

- A method that returns a value can also perform an action
- If you want the action performed, but do not need the returned value, you can invoke the method as if it were a `void` method, and the returned value will be discarded:

```
objectName.returnedValueMethod();
```

Testing Methods

- Each method should be tested in a program in which it is the only untested program
  - A program whose only purpose is to test a method is called a *driver program*
- One method often invokes other methods, so one way to do this is to first test all the methods invoked by that method, and then test the method itself
  - This is called *bottom-up testing*
- Sometimes it is necessary to test a method before another method it depends on is finished or tested
  - In this case, use a simplified version of the method, called a *stub,* to return a value for testing

The Fundamental Rule for Testing Methods

- *Every method should be tested in a program in which every other method in the testing program has already been fully tested and debugged*

Preconditions and Postconditions

- The *precondition* of a method states what is assumed to be true when the method is called
- The *postcondition* of a method states what will be true after the method is executed, as long as the precondition holds
- It is a good practice to always think in terms of preconditions and postconditions when designing a method, and when writing the method comment

Naming Conventions

- Start the names of variables, methods, and objects with a lowercase letter, indicate "word" boundaries with an uppercase letter, and restrict the remaining characters to digits and lowercase letters

  `topSpeed` `bankRate1` `timeOfArrival`

- Start the names of classes with an uppercase letter and, otherwise, adhere to the rules above

  `FirstProgram` `MyClass` `String`