
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS390 Foundamental Programming Practices (FPP)
Professor Paul Corazza

Lecture 2:
Fundamental Programming Structures In Java

Wholeness of the Lesson

Java is an object-oriented programming language that supports both primitive and object data types. These data types make it possible to store data in memory and modify it or perform computations on it to produce useful output. Execution of a program is an example of the "flow of knowledge"—the intelligence that has been coded into the program has a chance to be expressed when the program executes.

Maharishi's Science of Consciousness locates three components to any kind of knowledge: the knower, the object of knowledge, and the process of knowing. These can be found in the structure of a Java program: the "knower" aspect of the program is the intelligence underlying the creation of Java objects—a Java *class*. The *data* that a program works on, which is stored in program variables of either primitive or object type, is the "object of knowledge." And the Java methods, which act on the data, are the "process of knowing."

Outline of Topics

- First Java program and the *Reference Example*
- Data Types:
  - The Primitive Types
  - Other data types at the basis of object creation
- Operators In Java
  - Arithmetic Operators
  - Increment and Decrement Operators
  - Relational And Boolean Operators
  - Bitwise Operators
- Java Strings
- Control Flow:
  - Conditional Logic
  - While Loops
  - For loops
  - The switch Statement
- Arrays

Introducing Java

- Begin by studying a simple example:

```
public class Hello {
    public static void main(String[] args) {
            System.out.println("Hello World!");
    }
}
```

Things to understand:

- public
- class
- static
- void
- main
- String[]
- System, System.out, System.out.println (vs System.out.print)
- delimiters: ;, }, { ("blocks")
- capitalization conventions

A Java Application

- Java applications are *object-oriented*. This means that, unlike C, a Java program works by invoking multiple objects that then interact to produce results
- We provide a sample to give a feeling for how a Java program works. The details about it will be explained in Lesson 3. For now, it is possible to learn quite a bit by just playing with the syntax to create other programs.
- See the code in the package

  `lesson2.basics.typicalprogram`

  This code will be referred to in future lessons as the

  reference example

Comments In Java

- commenting out a line with //
- commenting out a block with /* … */
- commenting using javadoc format /** … */
- some javadoc keywords @author, @since, @param @return
- javadoc in Eclipse
- running javadoc (demo) - Project > Generate javadoc
- ***Style:*** Every significant method you write should be documented with comments, javadoc style. (This is also true of every Java class you create.)
- See how this is done in the ***reference example.***

Data Types: The Primitive Types

- Every variable must have a declared type
- Eight primitive types: `int`, `short`, `long`, `byte`, `float`, `double`, `char`, `boolean`

Type	Storage Requirement	Range (Inclusive)
`int`	4 bytes	-2^{31} to $2^{31} - 1$
`short`	2 bytes	-2^{15} to $2^{15} - 1$
`long`	8 bytes	-2^{63} to $2^{63} - 1$
`byte`	1 byte	-2^{7} to $2^{7} - 1$ (-128 to 127)
`float`	4 bytes	6 - 7 significant (decimal) digits
`double`	8 bytes	15 significant (decimal) digits

- `boolean` has just 2 values: *true* and *false*. (Unlike C, not the same as 1 and 0.)

The char Type

- To represent a character literal in Java, for commonly used characters, simply place the character between single quotes: 'A' represents the letter A.

- Characters can also be represented using a *unicode* representation – this is useful for characters that cannot be typed directly from a keyboard.

  Examples:

  - the ordinary letter 'A' is represented in this notation by `'\u0041'`
  - the Chinese character 终 by `'\u7ec8'` (Zhōng)

- Occasionally, a character can be represented only if two of Java's unicode characters are concatenated together – these are called *supplementary characters,* and typically show up only in very specialized applications.

  Example:

  the symbol for the set ℤ of integers in mathematics is represented by the pair `"\ud835\udd6b"`

The char Type (continued)

- To compute the unicode value of a basic Java character, cast it to an int (and convert to hex notation)

```
char c = 'A';
int unicodeVal = (int)c;  // this is in base 10
String hexVal = Integer.toHexString(unicodeVal); //value = 41

char c = '终';
int unicodeVal = (int)c;  // this is in base 10
String hexVal = Integer.toHexString(unicodeVal); //value = 7ec8
```

- To render a character in output, pass in the Java unicode to `System.out.println`.

```
System.out.println('\u7ec8');  //output is 终
System.out.println("\ud835\udd6b"); //output is 𝕫
```

- To see the full Unicode character map, go to `en.wikibooks.org/wiki/Unicode/Character_reference/`
- See demo code: `lesson2.charsandstrings.Main`

Escape Characters

- \u is an example of an *escape character*. Other common escape characters are used to represent special characters:

  \b – backspace

  \t – tab

  \n – newline

  \r – carriage return

  \" – double quote

  \' – single quote

  \\ - backslash

```
System.out.println("After waving, he said \"hello\"");
```

//output: After waving, he said "hello"

Variables In Java

- Variables in Java store values, like strings, numbers, and other data
- A variable in Java always has a type; a variable is *declared* by displaying the type, followed by the variable name.
- Examples of declaring variables (see also the ***reference example***)

```
double salary;
int amount;
boolean found;
```

- Variable names consist of digits, letters and underscores, but may not begin with a digit. (More precise criteria are available in the documentation on the `isJavaIdentifierStart` and `isJavaIdentifierPart` methods of the `Character` class.)

Variables In Java

- *Variable Initialization.* A variable is initialized by using the *assignment operator* (=) to specify a value for a declared variable.

  Example:

  ```
  int sum;
  sum = 0;
  ```

  OR

  ```
  int sum = 0;
  ```

- *Coding Style:*
  - variable names begin with lower case letter
  - variable names composed of multiple words written so that each new word (except the first) begins with a capital letter, but all other letters are lower case
    Example:

    ```
    myExamScore
    ```

  - underscores should *not* be used typically in variable names
  - for *constants*, capitals and underscores are used (discussed later)

    ```
    CONSTANTS_LIKE_THIS
    ```

Introducing Other Data Types: Reading Console Input

- We have examined primitive data types (int, char, boolean, etc) and how variables are declared using these types. Most of the data types used in Java, however, are not built into the language, but are created through *class definitions*. This will be the topic of Lesson 3 – see the ***reference example*.**

- To introduce working with console input, we give another example a class definition: the `Scanner` **class.**

Samples

- Scanner (as of j2se5.0)

```
//can think of Scanner as a special data type
Scanner sc = new Scanner(System.in);
System.out.print("Type your name: ");
System.out.println("you wrote: " + sc.nextLine());
System.out.print("Type your age: ");
System.out.println("your age: " + sc.nextInt());
sc.close();  //don't forget to close
```

```
//output
Type your name: Jim Stevens
you wrote:    Jim Stevens
Type your age: 36
your age: 36
```

See package lesson2.scannerandreader

- System.in and Readers (jdk1.1) (behaves like Scanner approach)

```
BufferedReader in = null;
String input = null;
in = new BufferedReader(new
    InputStreamReader(System.in));
System.out.print("Type your name: ");
input = in.readLine();
System.out.println("You wrote " + input);
in.close();
```

- JOptionPane (jdk1.2) (creates a GUI window for input)

```
String input =
    JOptionPane.showInputDialog("Type
        your name");
```

Main Point

Variables in Java are *declared* and *initialized* to provide room in RAM for the data that is to be stored. Pure consciousness manifests as individuals in space.

Operators In Java: Arithmetic Operators

- Standard binary operations represented in Java by `+, -, *, /`. Also the modulus operator `%`. Note: In Java, to compute `-5/2` (integer division) and `-5 % 2`, remove the minus sign, compute, and then insert the minus sign again:

```
-5/2   = - ( 5/2 ) = -2
-5 % 2 = - ( 5 % 2 ) = -1
```

Warning! This way of calculating modulus is an old mistake that is found in most procedural languages – the computation differs from the usual mathematical definition of modulus – in math, the modulus is always a nonnegative number.

=> **Java 8** corrects this with `Math.floorMod`:

```
Math.floorMod(-5, 2) = -5 (mod 2) = 1
```

See the `lesson2.modulus` package for this lesson.

- Division by 0: for `int`s, an exception is thrown; for floating point numbers, the value is `NaN`

- The operators `+=, *=, /=, -=, %=`

Operators In Java: Increment and Decrement Operators

- Variables having primitive numeric type can be incremented and decremented using "++" and "--" respectively (char types can be incremented like this too, but it is not a good practice to do this)

- Example:

```
int k = 1;
k++;   //new value of k is 2 (postfix form)
++k;   //new value of k is 3 (prefix form)
```

- Difference between postfix and prefix forms arises when used in expressions – prefix form is evaluated *before* evaluation, postfix form *after* evaluation

Operators (continued)

- Example:

```
int k = 0;
int m = 3 * k++;  //m equals 0, k equals 1

int q = 0;
int n = 3 * ++q;  //n equals 3, q equals 1
```

- Commonly used in for loops (coming up soon) but also in traversing arrays (also coming up soon). These are standard uses; mostly this style should be avoided for the sake of readability.

Operators In Java: Relational And Boolean Operators

- **Relational:** == (equals), != (not equals), < (less than), <= (less than or equal to), >, >= (greater than, greater than or equal to)
- **Logical:** &&, ||, ! . Short-circuit evaluation.

&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

If.. then	T	F
T	T	F
F	T	T

- **Ternary:** *condition* ? *expression1* : *expression2* – evaluates to expression1 if condition is true, expression2 otherwise
- Example:

```
CustomerStatus =
     (income > 100000) ?
           PLATINUM : SILVER;
```

is equivalent to this logic:

```
IF( income > 100000 )
       customerStatus = PLATINUM
ELSE
       customerStatus = SILVER
```

Operators In Java: Bitwise Operators

- & (and), | (or), ^ (xor), ~ (not), << (left shift), >> (right shift)

&	1	0
1	1	0
0	0	0

|	1	0
1	1	1
0	1	0

^	1	0
1	0	1
0	1	0

~1 = 0

~0 = 1

- Examples of << and >>

0000 1111 >> 2 = 0000 0011 (right shift by 2 is same as dividing by 4)
[15 >> 2 = 15/4 = 3]

0000 1111 << 2 = 0011 1100 (left shift by 2 is same as multiplying by 4)
[15 << 2 = 15 * 4 = 60]

Mathematical Constants And Functions

- Special math functions and constants are available in Java by using the syntax
  `Math.`<*constant*> and `Math.`<*function*>

- Examples:

```
Math.PI    (the number pi - approximately 3.14159)

Math.pow(a,x)   (the number a raised to the power x)

Math.sqrt(x)   (the square root of x)
```

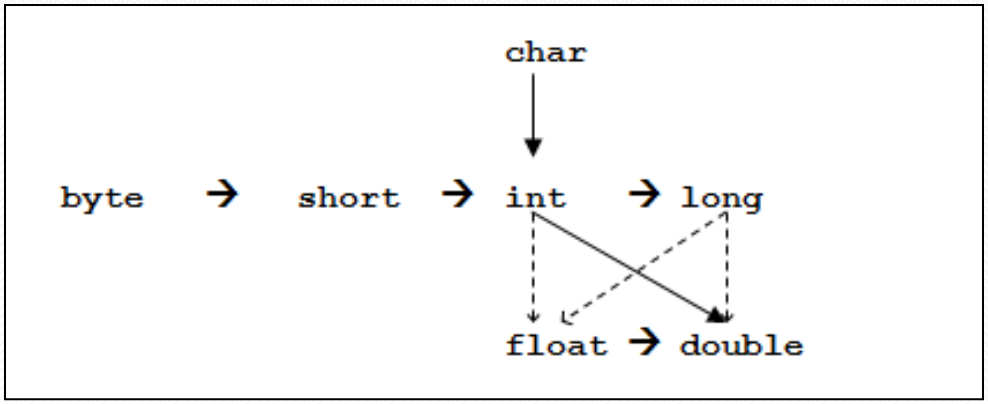
- For this course, we have a `RandomNumbers` class (which uses the Java Random class). Its methods can be accessed in the same way the methods of `Math` class can.

- Examples:

```
//produces a randomly generated int
int n = RandomNumbers.getRandomInt();

//produces a randomly generated int in the range 3..11, inclusive.
int m = RandomNumbers.getRandomInt(3,11);
```

Conversions Between Numeric Types



- Solid arrows indicate automatic type conversions that do not entail information loss
- `int` to `float`, `long` to `float`, and `long` to `double` are automatic, preserve number of digits to left of decimal, but may lose precision:

Example:

```
int n = 123456789;
float f = n;  //f is 123456792.000000,
```

See `lesson2.datatypeconversion.DataConversion`

- When values of different type are combined (via addition, multiplication or other operations), a type conversion occurs to arrive at just one common type.

  Most important cases:
  - a double combined with another primitive numeric type results in a double
  - an int combined with a smaller type (byte, short) results in an int.

- (A complete list of rules is given on p. 60 of Core Java, Vol 1, 10th ed.)

- Other conversions can be "forced" by casting.

  Example:

```
double x = 9.997;
int y = (int) x;  //y has value 9
```

- "Rounding" is usually preferable to casting and is done by using the round function of the Math class:

```
Example:
double x = 9.997;
int nx = (int)Math.round(x); //round returns a long,
 //so cast is necessary
```

- It is possible to cast a `long` to an `int`, an `int` to a `byte`, and so forth. In such cases, casting is accomplished by removing as many high-order bits as necessary to produce a number of the narrower type.

```
Example:
int x = 130;              ( = 000. . .00 1000 0010 as an int)
byte b = (byte)x;         ( = 1000 0010 as a byte = -126)
```

- *Automatic promotion of integral types.* When a binary operation (like +, *, or any shift operator) is applied to values of type `byte` or `short`, the types are promoted to `int` before the computation is carried out.

Example: The following produces a compiler error. Why?

```
byte x = 5;
byte y = 7;
byte z = x + y;
```

Operator Precedence and Association Conventions

Examples:

`a && b || c` means `(a && b) || c` (operator precedence)

`a += b += c` means `a += (b += c)` (association to the right)

- See page 64 of Core Java, Vol 1, 10th ed. for a list of all the rules
- Important precedence rules to know:
  - *, / , % precede + and –
  - ++ and -- have almost the highest precedence
  - = (assignment) associates to the right: `a = b = c` means `a = (b = c)`
  - *Tip:* be able to use the chart on p. 64 to read syntax

Example. Use the precedence rules table (below and on p.53) to evaluate:

```
7 & 13 >> 2 ^ 5
```

Solution:

```
7 & 13 >> 2 ^ 5
        =(7 & (13 >> 2)) ^ 5 (preced rules)
        =(7 & 3) ^ 5)       (since 13>>2 =3)
        = 00000011 ^ 00000101
        = 00000110
        = 6
```

Table 3-4. Operator precedence

Operators	Associativity
[] . () (method call)	left to right
! ~ ++ +(unary) -(unary) ()(cast) new	right to left
* / %	left to right
+ -	left to right
<< >> >>>	left to right
< <= > >= instanceof	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?: (ternary operator)	right to left
= += -= *= /= %= &= |= ^= <<= >>= >>>=	right to left

Main Point

Variables of primitive type can be combined to form expressions through the use of *operators*. The Java syntax requires one to observe rules for forming expressions – precedence rules, type conversion rules, and others. Pure consciousness, likewise, also has laws that govern its self-combining. The self-combining dynamics of pure consciousness – like the self-interacting dynamics of the unified field – give rise to "everything": All thoughts, all knowledge, and, if the unified field is really the same as pure consciousness, all manifest existence.

Java Strings

- A `String` is a sequence (technically, an array) of characters – therefore, formally, "String" is not a built-in data type (unlike `int` and `float`)

- A `String` can be created using a string literal.

  Example:

```
String name = "Jennifer";
String emtpy = "";
```

- Java `Strings` are *immutable*. This means that it is not possible to change the values of the characters within a `String`.

Java Strings: The `charAt` and `length` Methods

- Thinking of a Java `String` as a sequence of characters, the `charAt` method extracts the character at a specified position in this sequence:
  - `"Hello".charAt(1)  //value is 'e'`
  - For special characters, care is needed. For example: Suppose a `String expr` stores the `String`

```
"ℤ is the set of integers"
```

In this case, the expression

```
expr.charAt(1)
```

has value `'\uDD6B'` since it is the second character of the pair `\ud835\uDD6B` that represents the character ℤ.

- The `length()` method returns the number of Java characters in a `String`. When only basic characters are present, the length is just the number of characters. But care is needed with *special* characters.
  - The value of

    ```
    "Hello".length()
    ```

    is 5

    However, the value of

    ```
    "𝕫_𝕫".length()
    ```

    is 5, not 3.

String Functions: substring, indexOf, startsWith, +, equals, StringJoiner

- Examples of how the `String` functions are used:
  - `substring`

```
String name = "Robert";
String nickname = name.substring(0,3);// "Rob"
String whole = name.substring(0,name.length()); //"Robert"
String first = name.substring(0,1); //"R"
String empty = name.substring(0,0);  // ""
```

  - `indexOf`

```
String name = "Robert";
int posOfT = name.indexOf('t'); //5
int posOfSubstr = name.indexOf("bert"); //2
```

- `startsWith`

```
String name = "Robert";
boolean result = name.startsWith(“Rob");//true
boolean result2 = name.startsWith("R"); //true
boolean result3 = name.startsWith("bert"); //false
```

- `+` (concatenation) – creates a new String

```
String name = "Robert";
String space = " ";
String lastName = "Stevens";
String fullname = name + space + lastName;// "Robert Stevens"
```

- `equals`

```
String name = "Robert";
boolean equal = name.equals("Robert"); //true
boolean refEqual = (name == "Robert");  //true, but be careful
```

Note: `equals` and + are illustrated in the ***Reference Example***

- `StringJoiner`

  This relative of the `String` class can be used to produce a formatted sequence of `String`s.

  Example. The `String`

  `"[George:Sally:Fred]"`

  may be constructed as follows:

```
StringJoiner sj = new StringJoiner(":", "[", "]");
sj.add("George").add("Sally").add("Fred");
String desiredString = sj.toString();
```

Formatted Console Output

- j2se5.0 introduced C-like formatting features with `System.out.printf` and `String.format`
- Use `System.out.printf` to print formatted output directly to the console
- Use `String.format`, with the same formatting options, to store formatted String in memory, perhaps to be sent to the console or a file (for example) at a later time
- Can be combined with `Date` formatting
- For jdk1.4 and before, the `MessageFormat` class is used for formatting `Strings` – see sample code below
- A complete list of conversion characters (like s, d) can be found on p. 83, Core Java, 10th edition.

Samples

```
System.out.printf("You owe me $%f \n", 195.50f);
System.out.printf("You owe me $%.2f \n", 195.50f);
System.out.printf("You owe me $%7.2f \n", 195.50f);
```

You owe me $195.500000
You owe me $195.50
You owe me $ 195.50

```
String name = "Bob";
int age = 30;
System.out.printf("Happy birthday %s. I can't believe you're
%d.",name,age);
```

Happy birthday Bob. I can't believe you're 30.

```
String oweMe = String.format("You owe me %.2f dollars", 196f);
String oweMe2 = String.format("You owe me %d dollars", 196);
System.out.println(oweMe);
System.out.println(oweMe2);
```

You owe me 196.00 dollars
You owe me 196 dollars

Samples

```
String date = String.format("Today's date: %tD", new Date());
System.out.println(date);
```

Today's date: 09/09/05

```
//formatting in jdk1.4 - uses arrays, explained soon
Object [] params = {"animal", "dog"};
String stringWithParameter =
    "Look at that {0} -- it looks like a {1}.";
System.out.println("original string: " + stringWithParameter);
System.out.println("formatted string: " +
MessageFormat.format(stringWithParameter,params));
```

original string: "Look at that {0} -- it looks like a {1}."
formatted string: Look at that animal -- it looks like a dog.

Control Flow: Conditional Logic

- The conditional statement in Java has the following form:

```
if(condition) statement
```

Here, condition is any boolean statement (statement that evaluates to true of false)

- The *statement* may in fact be an entire block of code, in this form:

```
if(condition) {
      statement1;
      statement2;
      …
}
Example:

      if(sales >= target) {
            performance = "Satisfactory";
            bonus = 100;
      }
```

Control Flow: Conditional Logic

- Another form of conditionals is the *"if...else"* form:

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

```
Example:
    if(sales >= target) {
          performance = "Satisfactory";
          bonus = 100;
    }
    else {
          performance = "Unsatisfactory";
          bonus = 0;
    }
```

See the ***reference example*** (the Main class).

- Can have repeated “else if”’s .

Example:

```
if(sales >= 2 * target) {
          performance = "Excellent";
          bonus = 100;
}
else if (sales >= target {
          performance = "Satisfactory";
          bonus = 50;
}
else { //sales < target
          performance = "Unsatisfactory";
          bonus = 0;
}
```

An “else” is associated with nearest previous “if”. Therefore, these statements are read by the compiler as:

```
if(sales >= 2 * target) {
          performance = "Excellent";
          bonus = 100;
}
else {
          if (sales >= target) {
                    performance = "Satisfactory";
                    bonus = 500;
          }
          else { //sales < target
                    performance = "Unsatisfactory";
                    bonus = 0;
          }
}
```

Control Flow: While Loops

- The general form of a `while` loop is

  `while (`*`condition`*`)` *`statement`*

  where *condition* is a boolean expression.

- The general form of a `do…while` loop is

  ```
  do statement while (condition)
  ```

- Typically, `do…while` is used in place of `while` when it is necessary for `statement` to execute at least once (even if `condition` is always false).

Examples

```
//while loop
while(balance < goal) {
  balance += payment;
  double interest
     = balance * interestRate/100;
  balance += interest;
  years++;
}
System.out.println(years + "
years");
```

```
//do..while loop
Scanner sc = new Scanner(System.in);
do{
  System.out.print("Payment amount? ");
  payment = sc.nextDouble();
  balance += payment;
  double interest = balance * interestRate / 100;
  balance += interest;
  years++;
  System.out.println("Your balance: " +
     balance);
  System.out.println(
      "Make another payment? (Y/N)");
  input = sc.next();
}
while(input.equals("Y"));
```

Examples – the while(true) Construct

- Use the `while(true)` form when the `statement` requires processing before a condition can be evaluated. To exit the loop, use a `break` statement.

Example:

```
Scanner sc = new Scanner(System.in);
while(true) {
    System.out.print ("Enter a positive number: ");
    int value = sc.nextInt();
    if(value <= 0){
        break;
    }
}
System.out.println("The value you enter must be
positive.");
```

while(true) - continued

- Also used sometimes in creating a server (for a client/server system); in this case, the while loop never stops (until the server itself stops):
- *Using* `break` *in while loops.* When a `break` statement occurs, the while loop is exited and execution resumes as it would if the condition in the while loop had just failed. When possible, use `while` *without* a `break` statement (by selecting the condition for the `while` loop carefully) – sometimes though `break` statements are necessary.

Control Flow: for Loops

- General form of the `for` loop:

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

- Sample code: see the *reference example*, `Main` class.
- All three parts of the `for` expression are optional. The expression

  `for( ; ; )` *statement*

  means the same as

  `while(true)` *statement*

Examples

```
//standard
for(int i = 0; i < max; ++i) {
     //do something
}
```

Note: Since `i` is declared in the for expression, it cannot be referenced outside of the for block. If you need to use it outside the block, this code should be used:

```
int i;
for(i = 0; i < max; ++i) {
     //do something
}
//now i can be referenced here
```

or, equivalently,

```
int i=0;
for( ; i < max; ++i) {
     //do something
}
//now i can be referenced here
```

Examples - continued

More than one variable can be initialized, and more than one increment statement can be used; commas separate such statements.

```
for(int i = 1, j = max; i * j <= balance; i++, j--) {
    //do something
}
```

Complex conditions are allowed in the condition slot:

```
for(int i = 0; (i+1) * value > min && i * value <
max; i = i +2) {
    //do something
}
```

Nested for loops – example

```
for(int i = 0; i < n; ++i){
    for(int j = 0; j < n; ++j){
          System.out.printf("%-3s","*");
    }
    System.out.println();
}
// '-' flag means "left justify" within field

//output for n = 5

* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Examples - continued

```
for(int i = 0; i < n; ++i){
    for(int j = 0; j <= i; ++j){
          System.out.printf("%-3s","*");
    }
    System.out.println();

}

//output for n = 5

*
* *
* * *
* * * *
* * * * *
```

Control Flow: The switch Statement

- The `switch` statement is a convenient shorthand for writing "`if..else`" statements, when the values being tested are `int`s, `char`s, `String`s, or `enum`s. (Note: `enum`s will be discussed in Lesson 3.)
- General form of the `switch` statement:

```
switch(val) {
     case x:
               statement_x;
               break;
     case y:
               statement_y;
               break;
     …

     default:
               default_statement;
}
```

- The `break` in each case ensures that only one case is exectuted. If you forget to insert the break, later cases will continue to be tested and executed.
- A `default` case should typically be provided, to handle all cases not specified in the case statements.

“Fall-through” Behavior

- “Fallthrough behavior” occurs when break statements are omitted: Cases are examined and, as soon as a match is found, the corresponding statement is executed, and all subsequent case statements are also executed, until a break is encountered. If no matches are found, then the default statement is executed if there is one.
- Example of “fallthrough behavior”:

```
Scanner sc = new Scanner(System.in);
System.out.print("Pick an integer in the range 1..9");
int val = sc.nextInt();
System.out.println();
switch(val) {
    case 2:
    case 4:
    case 6:
    case 8:
        System.out.println("You chose an even number.");
        break;
    default:
        System.out.println("You chose an odd number.");
}
```

Main Point

Control flow is supported in Java via the *if..else, for, while, do..while, switch* [and also *for each*] language elements. Loops are the CS analogue to the self-referral performance at the basis of all creation, whereas branching logic mirrors the tree-like hierarchy of natural laws that guide the activity in each layer of creation.

Arrays

An array is a data structure that stores a collection of values of the same type and that supports *random access* of its elements (the element at position `i` in an array `arr` is retrieved using the syntax `arr[i]`).

- *Declaration of arrays*

```
int[] arr;
```

- *Initialization of arrays*

```
int[] arr = new int[100];
```

100 cells, numbered 0 to 99, are created and by default, each cell contains the value 0. All numeric arrays (for primitive types) are filled with their own version of 0 when initialized. String arrays (and arrays of objects of other kinds) are filled with the value `null` (more on this later).

Arrays

- *Setting values in an array*

```
arr[5] = 30;
```

- *Retrieving values in an array*

```
int positionFour = arr[4];
```

- *Length of an array.* This is the size determined at initialization and may not be changed.

```
int len = arr.length;  // len is 4
```

- Note: Arrays are used in the ***Reference Example***

Application of Arrays-the split function of the String class

- Use `split` to break up a `String` into `tokens` based on a set of *delimiters*.
- The statement

```
String[] parsedVals =  s.split(",");
```

will split the String s into tokens, using "," as delimiter, and will place the tokens in the array `parsedVals`

Example:

```
String s = "hello,how,are,you,today";
String[] parsedVals =  s.split(",");
```

The elements of `parsedVals` are:

```
hello
how
are
you
today
```

- .

- To treat more than one character as a delimiter for a `String`, list them in a String, using *bitwise or* (|) as a separator. To indicate a dot (.), you must use a double backslash (in Java's regular expression syntax, '\.' matches a dot; in Java, a backslash must be coded by "\\").

```
String t = "Hello,strings can be fun. They have many uses."
String[] result = t.split(",|\\.| ");  // 3 delimiters here
```

Here, `result` stores the array:

```
["Hello", "strings", "can", "be","fun", " ", "They", "have", "many", "uses"]
```

- In general, the argument passed to split can be any *regular expression* – see the `JavaDocs` discussion of `String.split` for an explanation of the many options.

- See demo code in package lesson2.split.

- NOTE: As of jdk1.4, the `split` method replaces the use of the class `StringTokenizer`

The *for each* Loop

```
int [] arr = {4, 5, 12, 25};
for(int x: arr) {
        System.out.println(x);
}
```

- Syntax:

  for(*variable* : *collection*) *statement*

  (As with ordinary for loops, the variable declaration can occur inside or outside the for expression)

  The *collection* must either be an array or an instance of a class that implements the `Iterable` interface (more on this later)

- ***Best Practice:*** Whenever there is a choice, use a *for each* loop in place of an ordinary *for* loop. The syntax is easier to read and doesn't rely on irrelevant information. (For instance, in this example, the *index* of each element in the array is not relevant for the task of printing the array elements)

Array Initializers and Anonymous Arrays

- When first created, can initialize an array like this (called an *array initializer*):

```
int[] somePrimes = {2, 3, 5, 7, 9, 11};

String[] names = {"Bob", "Harry", "Sue"};
```

But, the following is not legal:

```
String[] favoriteTeams = new String[2];
teams = {"Sonics", "Mets"};  //compiler error
```

- Anonymous arrays

```
new int[] { 17, 19, 23, 29 };
```

One application: permits initialization like an array initializer even after an array has been declared:

```
String[] favoriteTeams = new String[2];
teams = new String[]{"Sonics", "Mets", "Bulls"}; //change in size is ok
```

Array Copying and Sorting

To copy the cells from one array to another array, create a new (empty) array of the same size (or larger), and use `System.arraycopy`. To sort, use the `Arrays.sort` function.

Signatures:

```
System.arraycopy(from, fromIndex, to, toIndex, count)
Arrays.sort(arr)
```

Examples:

```
int[] smallPrimes = { 2, 3, 5, 7, 11};
int[] copy = new int[5];
System.arraycopy(smallPrimes, 0, copy, 0, 5);

int[] smallPrimes = { 2, 3, 5, 7, 11};
int[] luckyNums = {350, 400, 150, 200, 250};
System.arraycopy(smallPrimes, 1, luckyNums, 3, 2);

//luckyNums is now [350, 400, 150, 3, 5]
//now sort
Arrays.sort(luckyNums);
//luckyNums is now [3, 5, 150, 350, 400]
```

Commandline Parameters

The `main` method is designed to read input from the user when the program is executed.

```
class ParameterExample {
      public static void main(String[] args) {
                int len = 0;
                if(args != null) len = args.length;
                for(int i = 0; i < len; ++i) {
                    System.out.println("position " + i + ": " + args[i]);
                }
      }
}
```

Sample run of this code:

```
java ParameterExample Hello Goodbye
//output
position 0: Hello
position 1: Goodbye
```

Commandline parameters can be inserted into a Run Configuration in Eclipse. Right click class > Run As > Run Configurations, set name of configuration, add Program Arguments in Arguments tab. See demo in package: lesson2.commandlineparams

Introduction to Static Methods

- We have seen that the `main` method in a Java class is static and have discussed briefly what this means.
- In Java, static methods can call other static methods. Static methods are *utility methods* – designed to do some computation or processing, accepting inputs returning outputs, which support the main flow of the application. They can be used whether or not an instance of their enclosing class has been created. We will discuss static methods, and methods generally, in Lesson 3.

Example: See the `StringGame` demo in

`lesson2.staticdemo`

Avoiding Costly Concatenation of Strings with StringBuilder

- **Example**: You are writing an application that will receive an unknown number of `Strings` as command-line arguments. These `Strings`, when pieced together, will form a sentence. Your job is to concatenate all these `Strings` and output to console the final sentence, with the correct sentence structure. (Since we are assuming just one sentence is formed, the only adjustments we need to make to the input are to put spaces between the words and a period at the end.)

First Try

```
public static void main(String[] args) {
    if(args == null || args.length == 0) {
        System.out.println("<no input>");
    }
    String finalSentence = "";
    len = args.length;
    for(int i = 0; i < len-1; ++i) {
        finalSentence += (args[i] + " ");  //inefficient
    }
    finalSentence += (args[len-1] + ".");
    System.out.println(finalSentence);
}
//NOTE: couldn't do this with a "for each" loop
```

Problem: Concatenation becomes very slow with many arguments because each concatenation creates a new `String` (which requires allocating new memory for the new object), and compared to other steps, this is a costly operation.

Solution: `StringBuilder`

StringBuilder represents a "growable String" – can append characters and `Strings` without significant cost.

Note: `StringBuilder` is designed to be used for single-threaded applications – it is not thread-safe. This means that a single `StringBuilder` instance must not be shared between two or more competing threads. If multithreaded access is needed, a class with the same method names, `StringBuffer`, can be used, but it is less efficient in the single-threaded case.

Better Solution

```
public static void main(String[] args) {
    if(args == null || args.length == 0) {
        System.out.println("<no input>");
    }
    StringBuilder finalSentence = new StringBuilder();
    len = args.length;
    for(int i = 0; i < length-1; ++i) {
        finalSentence.append(args[i]);
        finalSentence.append(" ");  //much more efficient
    }
    finalSentence.append(args[len-1]);
    finalSentence.append(".");

    // Convert the StringBuilder to a String at the end.
    String finalSentenceAsString = finalSentence.toString();
    System.out.println(finalSentenceAsString);
}
```

Multidimensional Arrays

- Declaration:

```
int[][] twoD;
```

- Initialization:

```
int[][] twoDspecified = new int[3][5];  //3 int[] arrays
// each with 5 elements

int[][] twoDunspecified = new  int[3][]; //3 int[] arrays
//each of unspecified length

//ragged array
twoDunspecified[0] = new int[2];
twoDunspecified[1] = new int[3];
twoDunspecified[2] = new int[5];
```

- Array initializers

```
String[][] teams{
    {"Joe", "Bob", "Frank", "Steve"},
    {"Jon", "Tom","David","Ralph"},
    {"Tim", "Bev","Susan","Dennis"}
};

//specifies a 3 x 4 array
//teams.length is 3
//teams[i].length is 4 (whenever 0<= i <= 2)
//teams[1][2] has value “David” (row 1,
//column 2, start counting from 0)
```

Main Point

Arrays in Java support storage of multiple objects of the same type. Java supports multi-dimensional and ragged arrays; array copy and sort functions (accessible through the System and Arrays classes); and supports convenient forms of declaration and initialization. All CS data structures mirror the "existence" aspect of consciousness – the nervous system – whereas the *contents* of these structures mirrors the "intelligence" aspect; the pure potentiality of a data structure is as if brought to life by filling it with real data.

Summary

- We introduced the *Reference Example,* which shows how a Java program is composed of objects interacting with each other. More explanation of syntax will be given in Lesson 3.
- Java uses variables to store data, and all variables are given a *type.* The built-in types in Java are the primitive types (char, boolean, int, byte, short, double, float) together with more complex *object* types, to be discussed more in Lesson 3.Data Types:
- Data having primitive type can be manipulated using Java's operators, like +, * (arithmetic), &&, || (logical), and &, | (bitwise)
- An important data type in Java is a String, which provides many string manipulation operations, like `substring, +, indexOf, startsWith, charAt`.
- Procedural flow in a Java program is controlled by conditional logic (if..then..else, switch, and the ternary operator) and loops (for, forEach, while, do..while)
- Data in a Java program is stored in memory using *arrays,* which can be one- or multi-dimensional

Connecting the Parts of Knowledge With the Wholeness of Knowledge

1. In Java, variables of primitive type can be combined using operators to form expressions, which may be evaluated to produce well-defined output values.

2. On a broader scale, objects in Java are "combined" by way of "messages" between objects, which collectively result in the behavior of a Java application.

---

3. **Transcendental Consciousness:** Pure consciousness is the field beyond type and interaction; it is the field of *unbounded awareness* and *infinite silence*.
4. **Wholeness moving within itself**: In Unity Consciousness, one observes that this unbounded silent quality of awareness is spontaneously present at all levels of action in the world, and not just relegated to the transcendental field.