JAVA PROGRAMMING: CLASS 3

Rewrite: “Types” definition, Operator Classes
Java Control Structures
Abstract Algorithms
Maven
GUI Programming / 2D Graphics
Hands-on
Assignment 2
DEFINITION: TYPE

‘Type’ is usually defined with examples.

A stand-alone definition:

- A **type** is a data classifying name. It directs the compiler to accept a particular set of values to be assigned to a variable.
- Primitive types: boolean, char, short, int, long, float, double. Assignment: compiler accepts any value of that type.
- Reference types: Any class or interface name. Assignment: compiler accepts any instantiated object of that type.
OPERATOR CLASSES

‘Java Tutorial’ contains code examples for each class
- [http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html)

Others:
- Accessors: Array Access, Object Access
- Method Invocation
- Cast
- New
## OPERATOR CLASSES

<table>
<thead>
<tr>
<th>Operator Group</th>
<th>Applicable Types or Program Identifiers*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>All</td>
</tr>
<tr>
<td>Unary</td>
<td>Primitive except boolean</td>
</tr>
<tr>
<td>Equality</td>
<td>All</td>
</tr>
<tr>
<td>Relational</td>
<td>Primitive except boolean</td>
</tr>
<tr>
<td>Conditional (Logical)</td>
<td>Boolean</td>
</tr>
<tr>
<td>Type Comparison (instanceof)</td>
<td>Reference types</td>
</tr>
<tr>
<td>Bitwise</td>
<td>All primitives</td>
</tr>
<tr>
<td>Bitshift</td>
<td>All primitives except boolean</td>
</tr>
<tr>
<td>Accessors</td>
<td>Reference Types (array uses [], otherwise .)</td>
</tr>
</tbody>
</table>
## OPERATOR CLASSES

<table>
<thead>
<tr>
<th>Operator Group</th>
<th>Applicable Types or Program Identifiers*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Invocation</td>
<td>*Method Names</td>
</tr>
<tr>
<td>Cast</td>
<td>Any compatible type (numeric primitives to numeric primitives, reference types according to rules)</td>
</tr>
<tr>
<td>New</td>
<td>*Class Names</td>
</tr>
</tbody>
</table>
## OPERATORS AND WHEN TO USE THEM

<table>
<thead>
<tr>
<th>Operator Group</th>
<th>Generally when to use it</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>To store computational results</td>
</tr>
<tr>
<td>Unary</td>
<td>Simplify: Increment/decrement, arithmetic negation</td>
</tr>
<tr>
<td>Equality</td>
<td>Control flow</td>
</tr>
<tr>
<td>Relational</td>
<td>Arithmetic comparisons</td>
</tr>
<tr>
<td>Conditional (Logical)</td>
<td>Control flow</td>
</tr>
<tr>
<td>Type Comparison (instanceof)</td>
<td>Test type of a reference variable</td>
</tr>
<tr>
<td>Bitwise</td>
<td>Performance; Compact switch representation; compression, encryption</td>
</tr>
<tr>
<td>Bitshift</td>
<td>Performance: fast arithmetic, Various specialized algorithms e.g. checksums</td>
</tr>
</tbody>
</table>
## OPERATORS AND WHEN TO USE THEM

<table>
<thead>
<tr>
<th>Operator Group</th>
<th>Generally when to use it</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessors</td>
<td>Get the data behind the reference type.</td>
</tr>
<tr>
<td>Method Invocation</td>
<td>Method Invocation</td>
</tr>
<tr>
<td>Cast</td>
<td>To enable otherwise incompatible types to participate in an expression or statement</td>
</tr>
<tr>
<td>New</td>
<td>To instantiate an object of a given type and return a reference to it</td>
</tr>
</tbody>
</table>
OPERATOR ASSOCIATION EXAMPLES

</operator-association-examples>
blocks group statements and provide scope to variables declared inside them:

```java
int x = 50;
{
    int y = x; // earlier outer block variables are visible here
}
// y leaves scope and stack memory is reclaimed
}
// x leaves scope and stack memory is reclaimed
```
LOOPS: WHILE

Performs one statement or a block of statements repeatedly as long as a given boolean expression evaluates to true. The test is performed before each iteration. This while loop will repeat until the user enters 0:

```java
int x = FileIO.readInt(); // prime the while loop
while ( x != 0 ) {
    System.out.println("x is now = " + x);
    int x = FileIO.readInt();
}
assert x == 0;
```
**DO...WHILE**

```java
do {
    int x = FileIO.readInInt();
    System.out.println("x is now = " + x);
} while (x != 0)
assert x == 0;
```

The do loop is guaranteed to execute at least once and does not require priming. With a ‘while’ loop, the first iteration has to be done before the loops start.
‘IF’ STATEMENT

The “if” statement checks a boolean condition (with short-circuiting logic for && and ||) and executes a statement or block of statements once if the boolean expression is true.

    boolean isGreen = true;
    if (isGreen) {
        doSomething();
    }
‘IF-ELSE’ STATEMENT

If-else evaluates a boolean expression like ‘if’ and performs one statement or a block of statements if true. If false, the block immediately after the ‘else’ is executed instead. Only one System.out.println will ever be printed.

```java
boolean isGreen = true;
if (isGreen) {
    System.out.println("It’s Green");
} else {
    System.out.println("It’s Not Green");
}
```
DEFINITE ASSIGNMENT

With an ‘if-else if’, the compiler can’t tell that an assignment is always going to occur resulting in a compilation error if an uninitialized variable is later used.

```java
int x;
boolean test = false;
if (test == true) {
    x = 1;
} else if (test == false) {
    x = 2;
}
int y = x; // error: compiler doesn't realize x must be set in the ‘if-else if’ above.
```

See also: </example-definite-assignment>
ASSIGNMENT HAS A SIDE EFFECT

Two sequential ‘if’ statements with the same code inside can produce a different result than an if-else because in the former case the block that’s run can change the value of the tested variable resulting in the second block running, too.

See book p74 for example.
FOR LOOP: (EXPLICIT ITERATION)

```
int y = 0;
for (int i = 1; i <= 10; i++) {
    y = y + i;
}
System.out.println(" y = " + y);
```

'y' above is called an accumulator because it accumulates a value over each loop iteration. The for loop shown above is equivalent to:

```
int i = 0;
while ( ++i <= 10 ) {
    y = y + i;
}
```
FOR LOOP: EXPLICIT ITERATION

for ( optional-comma-separated-assignments ; boolean expression; optional-updates-to-loop-variables ) {
   // statements
}

A few things to notice:

- You can initialize zero or more loop variables in the optional-comma-separated-assignments.
- You can update zero or more loop variables in the optional-updates-to-loop-variables.
- The loop body’s statements should generally avoid updating the loop variables; it’s probably not what you want.
**FOR LOOP: (FOR-EACH SYNTAX)**

```java
int x[] = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for (int i : x) { // iterates over the array
    if (y >100) {
        break; // exit the innermost for loop; note: break and continue work for any loop
    }
    if (i==9) {
        continue; // go to next iteration
    }
    y = y + i;
}
System.out.println(" y = "+ y);
```
String userChoice = TEXTIO.getInString();
switch (userChoice) {
    case (“deposit”):
        display.showDepositTicket();
        break;
    case (“withdrawal”):
        display.showWithdrawalTicket();
        break;
    case (“inquiry”):
    case (“balance”): // falls through to default after showing account balance
        display.showAccountBalance();
        break;
    default:
        showBankAdvertisement();
}
SWITCH STATEMENT

• Somewhat easier to read than if-then-else
• ‘case’ constant is limited to byte, short, int, char, String, and enum types.
• Be careful not to omit a ‘break’ where necessary.
EXCEPTIONS ( TRY..CATCH )

FileInputStream in = null;
try {
    in = new FileInputStream("input.txt");  // This can throw an IO Exception if file not found
    while ((c = in.read()) != -1) {  // read() can also throw an IO Exception
        System.out.print(c);
    }
}
finally {  // this runs after try-catch, even if “return” occurs
    System.out.println("Completed file read try-catch block");
    if (in != null) {
        in.close();
    }
}
ABSTRACT ALGORITHMS

An algorithm is an effective method of correctly solving a problem in a finite number of steps.

If it doesn’t meet both those criteria, it isn’t an algorithm.

Algorithms are therefore referred to as “totally correct”.

Methods of solving problems that guarantee correctness, but not necessarily in a finite number of steps, are “partially correct”.

ABSTRACT ALGORITHMS

• When solving a problem in any programming language, start here.
• Think through an abstract solution in natural language (see examples in chapter 2).
• For now, refine the solution in steps until you can implement each step in Java.
• Later, we’ll talk about class design where we’ll modify this process.
APACHE MAVEN

“Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.”

Create a Java Project, then “Configure > Convert to Maven Project”. This will create a pom.xml (default packaging should be jar).

A jarfile is a java archive that contains the .class files that hold the compiled Java bytecode.

Right click on pom.xml, then “Run As > Maven install”.

Maven knows how to package your project jarfile. You can use it to package your code (find the output jarfile under the target directory).

Next class we’ll see:
- How POM dependencies provide access to the Central Repository. You can search it here: http://search.maven.org/
- How Maven can set up a skeleton project structure.
GUI PROGRAMMING (BASIC 2D GRAPHICS)

• A Graphics Context is a class used to create a 2D drawing area inside a Swing window.

• One Swing component you can draw in is JFrame. There are a selection of other components available in the Swing toolkit.

• X-coordinate runs along top line of the window increasing from left-right. Y-coordinate runs along the left side, increasing from top-down.

• You can specify the height and width of the JFrame among other properties.

Draw 2D images:
  ▪ Lines, Circles, ovals, arcs, images, polygons, rectangles (see Graphics2D API)

Can draw outlines or fill them.
Can set the container’s color. Anything drawn uses that color until it’s changed.
HANDS-ON

• Try out each loop construct to compute the sum of the first N integers. Assert the expected result after each.

• Install the GraphicsDriver, Graphics2DTemplate example classes to a project in your IDE workspace.

• Create a Graphics object with size 1000 x 1000 and draw various shapes in it.

• Draw a 10 x 10 grid with one color for the even rows and another for the odd rows.
ASSIGNMENT 2 (DUE: FEB 12, 2015)

- Part 1: Write an algorithm to set each element of an integer 12 x 12 two dimensional array with the product of the row and column number. Note that row and column number are indexed from 1, no zero.
  1. Print the sum of each of the rows.
  2. Tally and print the sum of all the row sums.
  3. Verify the tally in #2 with an assertion.
  4. Repeat 2-3 for the columns.
  5. Refine the algorithm until you can easily translate it into Java code.
  6. Submit your abstract algorithm, any refinements, and your source code.

- Part 2: Extend your program from Part 1 to draw a 2D 12 x 12 square grid filling each square with a color according to the 2D array element’s value from Part 1 and this strategy:
  - Red if the value of the square is divisible by 2
  - Yellow if the value of the square is divisible by 3
  - Green if the value of the square is divisible by 5
  - Blue otherwise