Building Java Programs

Lecture 1: Java Review

reading: Ch. 1-9
A Java program (1.2)

public class name {
    public static void main(String[] args) {
        statement;
        statement;
        ...
        statement;
    }
}

- Every executable Java program consists of a **class**, that contains a **method** named **main**, that contains the **statements** (commands) to be executed.
System.out.println

- A statement that prints a line of output on the console.
  - pronounced "print-linn"
  - sometimes called a "println statement" for short

- Two ways to use System.out.println:
  - System.out.println("text");
    Prints the given message as output.
  - System.out.println();
    Prints a blank line of output.
Static methods (1.4)

- **static method**: A named group of statements.
  - denotes the *structure* of a program
  - eliminates *redundancy* by code reuse

- **procedural decomposition**: dividing a problem into methods

- Writing a static method is like adding a new command to Java.

```
class
    method A
      statement
      statement
      statement

    method B
      statement
      statement

    method C
      statement
      statement
      statement
```
Declaring a method

*Gives your method a name so it can be executed*

- **Syntax:**
  
  ```java
  public static void name() {
    statement;
    statement;
    ...
    statement;
  }
  ```

- **Example:**
  
  ```java
  public static void printWarning() {
    System.out.println("This product causes cancer");
    System.out.println("in lab rats and humans.");
  }
  ```
Calling a method

Executes the method's code

- Syntax:
  
  ```
  name();
  ```
  
  - You can call the same method many times if you like.

- Example:
  
  ```
  printWarning();
  ```

- Output:
  
  This product causes cancer in lab rats and humans.
When a method is called, the program's execution...

- "jumps" into that method, executing its statements, then
- "jumps" back to the point where the method was called.

```java
public class MethodsExample {
    public static void main(String[] args) {
        message1();
        message2();
        System.out.println("Done with main.");
    }
    
    public static void message1() {
        System.out.println("This is message1.");
    }
    public static void message2() {
        System.out.println("This is message2.");
        message1();
        System.out.println("Done with message2.");
    }
    public static void message1() {
        System.out.println("This is message1.");
    }
}
```
Java's primitive types (2.1)

- **primitive types**: 8 simple types for numbers, text, etc.

  - Java also has **object types**, which we'll talk about later

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>integers</td>
<td>42, -3, 0, 926394</td>
</tr>
<tr>
<td>double</td>
<td>real numbers</td>
<td>3.1, -0.25, 9.4e3</td>
</tr>
<tr>
<td>char</td>
<td>single text characters</td>
<td>'a', 'X', '?', '\n'</td>
</tr>
<tr>
<td>boolean</td>
<td>logical values</td>
<td>true, false</td>
</tr>
</tbody>
</table>

- Why does Java distinguish integers vs. real numbers?
Expressions

- **expression**: A value or operation that computes a value.
  - **Examples**: $1 + 4 \times 5$
    $\quad (7 + 2) \times 6 / 3$
    $\quad 42$
  - The simplest expression is a *literal value*.
  - A complex expression can use operators and parentheses.
Integer division with /

• When we divide integers, the quotient is also an integer.
  • 14 / 4 is 3, not 3.5

\[
\begin{array}{c@{\hspace{1cm}}c@{\hspace{1cm}}c@{\hspace{1cm}}c}
3 & 4 & 52 \\
4 & 14 & 10 & 45 \\
12 & 40 & 27 & 1425 \\
2 & 5 & 135 & 75 \\
\end{array}
\]

• More examples:
  • 32 / 5 is 6
  • 84 / 10 is 8
  • 156 / 100 is 1

• Dividing by 0 causes an error when your program runs.
Integer remainder with %

- The % operator computes the remainder from integer division.
  - \(14 \% 4\) is 2
  - \(218 \% 5\) is 3

\[
\begin{array}{c}
14 \\
\hline
4 \\
-12 \\
\hline
2
\end{array}
\]

\[
\begin{array}{c}
218 \\
\hline
5 \\
-20 \\
\hline
18 \\
\hline
15 \\
\hline
3
\end{array}
\]

- Applications of % operator:
  - Obtain last digit of a number: \(230857 \% 10\) is 7
  - Obtain last 4 digits: \(658236489 \% 10000\) is 6489
  - See whether a number is odd: \(7 \% 2\) is 1, \(42 \% 2\) is 0

What is the result?
- \(45 \% 6\)
- \(2 \% 2\)
- \(8 \% 20\)
- \(11 \% 0\)
Precedence

• **precedence**: Order in which operators are evaluated.
  • Generally operators evaluate left-to-right.
    \[1 - 2 - 3 \text{ is } (1 - 2) - 3 \text{ which is } -4\]
  • But \(^*/\%\) have a higher level of precedence than \(+-\)
    \[1 + 3 \ast 4 \text{ is } 13\]
    \[6 + 8 / 2 \ast 3\]
    \[6 + 4 \ast 3\]
    \[6 + 12 \text{ is } 18\]
  • Parentheses can force a certain order of evaluation:
    \[(1 + 3) \ast 4 \text{ is } 16\]
  • Spacing does not affect order of evaluation
    \[1 + 3 \ast 4 - 2 \text{ is } 11\]
String concatenation

- **string concatenation**: Using `+` between a string and another value to make a longer string.

  - "hello" + 42 is "hello42"
  - 1 + "abc" + 2 is "1abc2"
  - "abc" + 1 + 2 is "abc12"
  - 1 + 2 + "abc" is "3abc"
  - "abc" + 9 * 3 is "abc27"
  - "1" + 1 is "11"
  - 4 - 1 + "abc" is "3abc"

- Use `+` to print a string and an expression's value together.
  - `System.out.println("Grade: " + (95.1 + 71.9) / 2);`
  - **Output**: Grade: 83.5
Variables (2.2)

- **variable**: A piece of the computer's memory that is given a name and type, and can store a value.

- A variable can be declared/initialized in one statement.

- Syntax:
  
  ```
  type name = value;
  ```

- ```double myGPA = 3.95;```  

- ```int x = (11 % 3) + 12;```
Type casting

- **type cast**: A conversion from one type to another.
  - To promote an `int` into a `double` to get exact division from `/`
  - To truncate a `double` from a real number to an integer

**Syntax:**

```
(type) expression
```

**Examples:**
```
double result = (double) 19 / 5;  // 3.8
int result2 = (int) result;       // 3
int x = (int) Math.pow(10, 3);   // 1000
```
Increment and decrement

shortcuts to increase or decrease a variable's value by 1

Shorthand

variable++;
variable--;

Equivalent longer version

variable = variable + 1;
variable = variable - 1;

int x = 2;
x++; // x = x + 1;

double gpa = 2.5;
gpa--; // gpa = gpa - 1;
Modify-and-assign operators

*shortcuts to modify a variable's value*

<table>
<thead>
<tr>
<th>Shorthand</th>
<th>Equivalent longer version</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>variable += value;</code></td>
<td><code>variable = variable + value;</code></td>
</tr>
<tr>
<td><code>variable -= value;</code></td>
<td><code>variable = variable - value;</code></td>
</tr>
<tr>
<td><code>variable *= value;</code></td>
<td><code>variable = variable * value;</code></td>
</tr>
<tr>
<td><code>variable /= value;</code></td>
<td><code>variable = variable / value;</code></td>
</tr>
<tr>
<td><code>variable %= value;</code></td>
<td><code>variable = variable % value;</code></td>
</tr>
</tbody>
</table>

x += 3;  // x = x + 3;
gpa -= 0.5;  // gpa = gpa - 0.5;
number *= 2;  // number = number * 2;
for loops (2.3)

```java
for (initialization; test; update) {
    statement;
    statement;
    ...
    statement;
}
```

- Perform `initialization` once.
- Repeat the following:
  - Check if the `test` is true. If not, stop.
  - Execute the `statements`.
  - Perform the `update`.
System.out.print

• Prints without moving to a new line
  • allows you to print partial messages on the same line

```java
int highestTemp = 5;
for (int i = -3; i <= highestTemp / 2; i++) {
    System.out.print((i * 1.8 + 32) + " ");
}
```

• Output:
  26.6  28.4  30.2  32.0  33.8  35.6
Nested loops

- **nested loop**: A loop placed inside another loop.

```java
for (int i = 1; i <= 4; i++) {
    for (int j = 1; j <= 5; j++) {
        System.out.print((i * j) + "\t");
    } System.out.println(); // to end the line
}
```

- Output:
  
  1    2    3    4    5
  2    4    6    8    10
  3    6    9   12   15
  4    8   12   16   20

- Statements in the outer loop's body are executed 4 times.
  - The inner loop prints 5 numbers each time it is run.
Variable scope

- **scope**: The part of a program where a variable exists.
  - From its declaration to the end of the `{ }` braces
  - A variable declared in a `for` loop exists only in that loop.
  - A variable declared in a method exists only in that method.

```java
public static void example() {
    int x = 3;
    for (int i = 1; i <= 10; i++) {
        System.out.println(x);
        // i no longer exists here
    }
    // x ceases to exist here
}
```

x's scope

i's scope
Class constants (2.4)

- **class constant**: A value visible to the whole program.
  - value can only be set at declaration
  - value can't be changed while the program is running

- Syntax:
  ```java
  public static final type name = value;
  ```
  - name is usually in ALL_UPPER_CASE

- Examples:
  ```java
  public static final int DAYS_IN_WEEK = 7;
  public static final double INTEREST_RATE = 3.5;
  public static final int SSN = 658234569;
  ```
Parameters (3.1)

- **parameter**: A value passed to a method by its caller.

  - Instead of `lineOf7, lineOf13, write line` to draw any length.
    - When *declaring* the method, we will state that it requires a parameter for the number of stars.
    - When *calling* the method, we will specify how many stars to draw.
Passing parameters

• Declaration:
  public static void name (type name, ..., type name) {
    statement(s);
  }

• Call:
  methodName (value, value, ..., value);

• Example:
  public static void main(String[] args) {
    sayPassword(42);    // The password is: 42
    sayPassword(12345); // The password is: 12345
  }

  public static void sayPassword(int code) {
    System.out.println("The password is: " + code);
  }
# Java's `Math` class (3.2)

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Math.abs(value)</code></td>
<td>absolute value</td>
</tr>
<tr>
<td><code>Math.round(value)</code></td>
<td>nearest whole number</td>
</tr>
<tr>
<td><code>Math.ceil(value)</code></td>
<td>rounds up</td>
</tr>
<tr>
<td><code>Math.floor(value)</code></td>
<td>rounds down</td>
</tr>
<tr>
<td><code>Math.sqrt(value)</code></td>
<td>square root</td>
</tr>
<tr>
<td><code>Math.log10(value)</code></td>
<td>logarithm, base 10</td>
</tr>
<tr>
<td><code>Math.max(value1, value2)</code></td>
<td>larger of two values</td>
</tr>
<tr>
<td><code>Math.min(value1, value2)</code></td>
<td>smaller of two values</td>
</tr>
<tr>
<td><code>Math.pow(base, exp)</code></td>
<td><code>base</code> to the <code>exp</code> power</td>
</tr>
<tr>
<td><code>Math.sqrt(value)</code></td>
<td>square root</td>
</tr>
<tr>
<td><code>Math.sin(value)</code></td>
<td>sine/cosine/tangent of an angle in radians</td>
</tr>
<tr>
<td><code>Math.cos(value)</code></td>
<td></td>
</tr>
<tr>
<td><code>Math.tan(value)</code></td>
<td></td>
</tr>
<tr>
<td><code>Math.toDegrees(value)</code></td>
<td>convert degrees to radians and back</td>
</tr>
<tr>
<td><code>Math.toRadians(value)</code></td>
<td></td>
</tr>
<tr>
<td><code>Math.random()</code></td>
<td>random double between 0 and 1</td>
</tr>
</tbody>
</table>

## Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.E</td>
<td>2.7182818...</td>
</tr>
<tr>
<td>Math.PI</td>
<td>3.1415926...</td>
</tr>
</tbody>
</table>
Return (3.2)

- **return**: To send out a value as the result of a method.
  - The opposite of a parameter:
    - Parameters send information *in* from the caller to the method.
    - Return values send information *out* from a method to its caller.

```
main
  Math.abs(42) - 42
  Math.round(2.71) -> 3
```

```
-42   42
Math.abs(42)   2.71
main
  3
Math.round(2.71)
```
Returning a value

```java
public static type name(parameters) {
    statements;
    ...
    return expression;
}

• Example:

    // Returns the slope of the line between the given points.
    public static double slope(int x1, int y1, int x2, int y2) {
        double dy = y2 - y1;
        double dx = x2 - x1;
        return dy / dx;
    }
```
Strings (3.3)

- **string**: An object storing a sequence of text characters.
  
  ```java
  String name = "text";
  String name = expression;
  ```

- Characters of a string are numbered with 0-based *indexes*:
  
  ```java
  String name = "P. Diddy";
  ```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>P</td>
<td>.</td>
<td>D</td>
<td>i</td>
<td>d</td>
<td>d</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

- The first character's index is always 0
- The last character's index is 1 less than the string's length
- The individual characters are values of type `char`
String methods

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexOf(str)</td>
<td>index where the start of the given string appears in this string (-1 if it is not there)</td>
</tr>
<tr>
<td>length()</td>
<td>number of characters in this string</td>
</tr>
<tr>
<td>substring(index1, index2) or substring(index1)</td>
<td>the characters in this string from index1 (inclusive) to index2 (exclusive); if index2 omitted, grabs till end of string</td>
</tr>
<tr>
<td>toLowerCase()</td>
<td>a new string with all lowercase letters</td>
</tr>
<tr>
<td>toUpperCase()</td>
<td>a new string with all uppercase letters</td>
</tr>
</tbody>
</table>

- These methods are called using the dot notation:

```java
String gangsta = "Dr. Dre";
System.out.println(gangsta.length()); // 7
```
### String test methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals(str)</td>
<td>whether two strings contain the same characters</td>
</tr>
<tr>
<td>equalsIgnoreCase(str)</td>
<td>whether two strings contain the same characters, ignoring upper vs. lower case</td>
</tr>
<tr>
<td>startsWith(str)</td>
<td>whether one contains other's characters at start</td>
</tr>
<tr>
<td>endsWith(str)</td>
<td>whether one contains other's characters at end</td>
</tr>
<tr>
<td>contains(str)</td>
<td>whether the given string is found within this one</td>
</tr>
</tbody>
</table>

```java
String name = console.next();
if (name.startsWith("Dr.")) {
    System.out.println("Are you single?");
} else if (name.equalsIgnoreCase("LUMBERG")) {
    System.out.println("I need your TPS reports.");
}
```
The `equals` method

- Objects are compared using a method named `equals`.

```java
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name.equals("Barney")) {
    System.out.println("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```

- Technically this is a method that returns a value of type `boolean`, the type used in logical tests.
Type `char` (4.4)

- `char`: A primitive type representing single characters.
  - Each character inside a `String` is stored as a `char` value.
  - Literal `char` values are surrounded with apostrophe (single-quote) marks, such as `a`, `4`, `\n`, or `\'`

- It is legal to have variables, parameters, returns of type `char`

```
char letter = 'S';
System.out.println(letter);  // S
```

- `char` values can be concatenated with strings.

```
char initial = 'P';
System.out.println(initial + " Diddy");  // P Diddy
```
char vs. String

- "h" is a String
  'h' is a char (the two behave differently)

- String is an object; it contains methods

```java
String s = "h";
s = s.toUpperCase();    // 'H'
int len = s.length();   // 1
char first = s.charAt(0);  // 'H'
```

- char is primitive; you can't call methods on it

```java
char c = 'h';
c = c.toUpperCase();    // ERROR: "cannot be dereferenced"
```

- What is s + 1? What is c + 1?
- What is s + s? What is c + c?
**System.out.printf (4.4)**

System.out.printf("**format string**", parameters);

- A format string contains *placeholders* to insert parameters into it:
  - `%d` an integer
  - `%f` a real number
  - `%s` a string
  - `%8d` an integer, 8 characters wide, right-aligned
  - `%8d` an integer, 8 characters wide, left-aligned
  - `%.4f` a real number, 4 characters after decimal
  - `%6.2f` a real number, 6 characters wide, 2 after decimal

- Example:

  ```java
  int x = 3, y = 2;
  System.out.printf("(%d, %d)\n", x, y); // (3, 2)
  System.out.printf("%4d %4.2f\n", x, y); // 3 2.00
  ```
DrawingPanel (3G)

"Canvas" objects that represents windows/drawing surfaces

- To create a window:
  
  ```java
  DrawingPanel name = new DrawingPanel(width, height);
  ```

  Example:
  ```java
  DrawingPanel panel = new DrawingPanel(300, 200);
  ```

- The window has nothing on it.
  - We can draw shapes and lines on it using another object of type Graphics.
"Pen" objects that can draw lines and shapes

- Access it by calling `getGraphics` on your `DrawingPanel`.
  
  ```java
  Graphics g = panel.getGraphics();
  ```

- Draw shapes by calling methods on the `Graphics` object.
  
  ```java
  g.fillRect(10, 30, 60, 35);
  g.fillOval(80, 40, 50, 70);
  ```
## Graphics methods

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>g.drawLine(x1, y1, x2, y2)</code></td>
<td>line between points ((x1, y1), (x2, y2))</td>
</tr>
<tr>
<td><code>g.drawOval(x, y, width, height)</code></td>
<td>outline largest oval that fits in a box of size (width \times height) with top-left at ((x, y))</td>
</tr>
<tr>
<td><code>g.drawRect(x, y, width, height)</code></td>
<td>outline of rectangle of size (width \times height) with top-left at ((x, y))</td>
</tr>
<tr>
<td><code>g.drawString(text, x, y)</code></td>
<td>text with bottom-left at ((x, y))</td>
</tr>
<tr>
<td><code>g.fillOval(x, y, width, height)</code></td>
<td>fill largest oval that fits in a box of size (width \times height) with top-left at ((x, y))</td>
</tr>
<tr>
<td><code>g.fillRect(x, y, width, height)</code></td>
<td>fill rectangle of size (width \times height) with top-left at ((x, y))</td>
</tr>
<tr>
<td><code>g.setColor(Color)</code></td>
<td>set Graphics to paint any following shapes in the given color</td>
</tr>
</tbody>
</table>
Color

- Create one using Red-Green-Blue (RGB) values from 0-255

  ```java
  Color name = new Color(red, green, blue);
  ```

- Example:

  ```java
  Color brown = new Color(192, 128, 64);
  ```

- Or use a predefined `Color` class constant (more common)

  ```java
  Color.CONSTANT_NAME
  ```

  where `CONSTANT_NAME` is one of:

  - BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, or YELLOW
Scanner (3.3)

- System.out
  - An object with methods named `println` and `print`
- System.in
  - not intended to be used directly
  - We use a second object, from a class `Scanner`, to help us.

- Constructing a `Scanner` object to read console input:
  ```java
  Scanner name = new Scanner(System.in);
  ```
- Example:
  ```java
  Scanner console = new Scanner(System.in);
  ```
Scanner methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nextInt()</td>
<td>reads a token of user input as an int</td>
</tr>
<tr>
<td>nextDouble()</td>
<td>reads a token of user input as a double</td>
</tr>
<tr>
<td>next()</td>
<td>reads a token of user input as a String</td>
</tr>
<tr>
<td>.nextLine()</td>
<td>reads a line of user input as a String</td>
</tr>
</tbody>
</table>

- Each method waits until the user presses Enter.
  - The value typed is returned.

```
System.out.print("How old are you? "); // prompt
int age = console.nextInt();
System.out.println("You'll be 40 in " +
       (40 - age) + " years.");
```

- **prompt**: A message telling the user what input to type.
Testing for valid input (5.3)

- **Scanner** methods to see what the next token will be:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNext()</td>
<td>returns <code>true</code> if there are any more tokens of input to read</td>
</tr>
<tr>
<td>hasNextInt()</td>
<td>returns <code>true</code> if there is a next token and it can be read as an <code>int</code></td>
</tr>
<tr>
<td>hasNextDouble()</td>
<td>returns <code>true</code> if there is a next token and it can be read as a <code>double</code></td>
</tr>
<tr>
<td>hasNextLine()</td>
<td>returns <code>true</code> if there are any more lines of input to read</td>
</tr>
</tbody>
</table>

- These methods do not consume input; they just give information about the next token.
- Useful to see what input is coming, and to avoid crashes.
Cumulative sum (4.1)

- A loop that adds the numbers from 1-1000:

```java
int sum = 0;
for (int i = 1; i <= 1000; i++) {
    sum = sum + i;
}
System.out.println("The sum is " + sum);
```

**Key idea:**
- Cumulative sum variables must be declared *outside* the loops that update them, so that they will exist after the loop.
if/else (4.2)

Executes one block if a test is true, another if false

```java
if (test) {
    statement(s);
} else {
    statement(s);
}
```

**Example:**
```java
double gpa = console.nextDouble();
if (gpa >= 2.0) {
    System.out.println("Welcome to Mars University!");
} else {
    System.out.println("Application denied.");
}
```
Relational expressions

- A test in an if is the same as in a for loop.

  ```java
  for (int i = 1; i <= 10; i++) {
    ...
  }
  if (i <= 10) {
    ...
  }
  ```

- These are boolean expressions, seen in Ch. 5.

- Tests use relational operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equals</td>
<td>1 + 1 == 2</td>
<td>true</td>
</tr>
<tr>
<td>!=</td>
<td>does not equal</td>
<td>3.2 != 2.5</td>
<td>true</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>10 &lt; 5</td>
<td>false</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>10 &gt; 5</td>
<td>true</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>126 &lt;= 100</td>
<td>false</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>5.0 &gt;= 5.0</td>
<td>true</td>
</tr>
</tbody>
</table>
Logical operators: &&, ||, !

- Conditions can be combined using *logical operators*:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
<td>(2 == 3) &amp;&amp; (-1 &lt; 5)</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td>!</td>
<td>not</td>
<td>!(2 == 3)</td>
<td>true</td>
</tr>
</tbody>
</table>

- "Truth tables" for each, used with logical values \( p \) and \( q \):

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>q</td>
<td>p &amp;&amp; q</td>
<td>p</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>!p</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
Type boolean (5.2)

- **boolean**: A logical type whose values are *true* and *false*.
  - A **test** in an **if**, **for**, or **while** is a **boolean** expression.
  - You can create **boolean** variables, pass **boolean** parameters, return **boolean** values from methods, ...

```java
boolean minor = (age < 21);
boolean expensive = iPhonePrice > 200.00;
boolean iLoveCS = true;
if (minor) {
    System.out.println("Can't purchase alcohol!");
}
if (iLoveCS || !expensive) {
    System.out.println("Buying an iPhone");
}
```
De Morgan's Law

- **De Morgan's Law:**
  Rules used to *negate* or *reverse* boolean expressions.
  - Useful when you want the opposite of a known boolean test.

<table>
<thead>
<tr>
<th>Original Expression</th>
<th>Negated Expression</th>
<th>Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \land b )</td>
<td>( \neg a \lor \neg b )</td>
<td>( \neg (a \land b) )</td>
</tr>
<tr>
<td>( a \lor b )</td>
<td>( \neg a \land \neg b )</td>
<td>( \neg (a \lor b) )</td>
</tr>
</tbody>
</table>

- **Example:**

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Negated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( { x = 7 \land y &gt; 3 } ) | ( { x \neq 7 \lor y \leq 3 } )</td>
<td>if ( { x \neq 7 \lor y \leq 3 } ) | ( { x = 7 \land y &gt; 3 } )</td>
</tr>
</tbody>
</table>
if/else Structures

- Exactly 1 path: (mutually exclusive)

```java
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else {
    statement(s);
}
```

- 0 or 1 path:

```java
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else if (test) {
    statement(s);
}
```

- 0, 1, or many paths: (independent tests, not exclusive)

```java
if (test) {
    statement(s);
} if (test) {
    statement(s);
} if (test) {
    statement(s);
} if (test) {
    statement(s);
} if (test) {
    statement(s);
} if (test) {
    statement(s);
} if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else {
    statement(s);
}
```
Fencepost loops (4.1)

- **fencepost problem**: When we want to repeat two tasks, one of them \( n \) times, another \( n-1 \) or \( n+1 \) times.
  - Add a statement outside the loop to place the initial "post."
  - Also called a *fencepost loop* or a "loop-and-a-half" solution.

- Algorithm template:

  ```
  place a post.
  for (length of fence - 1) {
    place some wire.
    place a post.
  }
  ```
Fencepost method solution

• Write a method `printNumbers` that prints each number from 1 to a given maximum, separated by commas.

For example, the call:
   `printNumbers(5);`

should print:
   `1, 2, 3, 4, 5`

• Solution:
   ```java
   public static void printNumbers(int max) {
       System.out.print(1);
       for (int i = 2; i <= max; i++) {
           System.out.print(", " + i);
       }
       System.out.println(); // to end the line
   }
   ```
while loops (5.1)

- **while loop**: Repeatedly executes its body as long as a logical test is true.

  ```java
  while (test) {
    statement(s);
  }
  ```

- **Example**:
  ```java
  int num = 1;  // initialization
  while (num <= 200) {  // test
    System.out.print(num + " ");
    num = num * 2;  // update
  }
  ```

- **OUTPUT**:
  1 2 4 8 16 32 64 128
do/while loops (5.4)

- **do/while loop**: Executes statements repeatedly while a condition is true, testing it at the end of each repetition.

  ```java
  do {
    statement(s);
  } while (test);
  ```

- **Example:**

  ```java
  // prompt until the user gets the right password
  String phrase;
  do {
    System.out.print("Password: ");
    phrase = console.next();
  } while (!phrase.equals("abracadabra"));
  ```
The Random class (5.1)

- A Random object generates pseudo-random* numbers.
  - Class Random is found in the java.util package.

```java
import java.util.*;

Example:

```Random rand = new Random();
int randomNumber = rand.nextInt(10);  // 0-9```
"Boolean Zen"

- Students new to boolean often test if a result is true:
  
  ```java
  if (bothOdd(7, 13) == true) { // bad
      ...
  }
  ```

- But this is unnecessary and redundant. Preferred:
  
  ```java
  if (bothOdd(7, 13)) { // good
      ...
  }
  ```

- A similar pattern can be used for a false test:
  
  ```java
  if (bothOdd(7, 13) == false) { // bad
      if (!bothOdd(7, 13)) { // good
          ...
      }
  ```
"Boolean Zen", part 2

- Methods that return boolean often have an if/else that returns true or false:

```java
public static boolean bothOdd(int n1, int n2) {
    if (n1 % 2 != 0 && n2 % 2 != 0) {
        return true;
    } else {
        return false;
    }
}
```

- Observation: The if/else is unnecessary.
  - Our logical test is itself a boolean value; so return that!

```java
public static boolean bothOdd(int n1, int n2) {
    return (n1 % 2 != 0 && n2 % 2 != 0);
}
```
break (5.4)

- **break statement**: Immediately exits a loop.
  - Can be used to write a loop whose test is in the middle.
  - Such loops are often called "forever" loops because their header's boolean test is often changed to a trivial `true`.

```java
while (true) {
    statement(s);
    if (test) {
        break;
    }
    statement(s);
}
```

- Some programmers consider `break` to be bad style.
Reading files (6.1)

- To read a file, pass a `File` when constructing a `Scanner`.
  
  ```java
  Scanner name = new Scanner(new File("file name"));
  ```

  Example:
  ```java
  File file = new File("mydata.txt");
  Scanner input = new Scanner(file);
  ```

  or, better yet:
  ```java
  Scanner input = new Scanner(new File("mydata.txt"));
  ```
The **throws clause**: Keywords on a method's header that state that it may generate an exception.

**Syntax:**

```java
public static type name(params) throws type {
```

**Example:**

```java
public class ReadFile {
    public static void main(String[] args) throws FileNotFoundException {
    
    ```

**Like saying, "I hereby announce that this method might throw an exception, and I accept the consequences if it happens."**
Input tokens (6.2)

- **token**: A unit of user input, separated by whitespace.
  - A **Scanner** splits a file's contents into tokens.

- If an input file contains the following:

<table>
<thead>
<tr>
<th>Token</th>
<th>Type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>int, double, String</td>
</tr>
<tr>
<td>3.14</td>
<td>double, String</td>
</tr>
<tr>
<td>&quot;John Smith&quot;</td>
<td>String</td>
</tr>
</tbody>
</table>

The **Scanner** can interpret the tokens as the following types:
Files and input cursor

• Consider a file `numbers.txt` that contains this text:

```
308.2
   14.9 7.4  2.8

3.9 4.7   -15.4
  2.8
```

• A Scanner views all input as a stream of characters:

```
308.2\n   14.9 7.4  2.8\n3.9 4.7   -15.4\n  2.8\n```

• **input cursor**: The current position of the Scanner.
Consuming tokens

- **consuming input**: Reading input and advancing the cursor.
- Calling `nextInt` etc. moves the cursor past the current token.

```java
double x = input.nextDouble();    // 308.2
String s = input.next();          // "14.9"
```
Scanner exceptions

- InputMismatchException
  - You read the wrong type of token (e.g. read "hi" as int).

- NoSuchElementException
  - You read past the end of the input.

Finding and fixing these exceptions:
- Read the exception text for line numbers in your code (the first line that mentions your file; often near the bottom):

```java
Exception in thread "main" java.util.NoSuchElementException
  at java.util.Scanner.throwFor(Scanner.java:838)
  at java.util.Scanner.next(Scanner.java:1347)
  at CountTokens.sillyMethod(CountTokens.java:19)
  at CountTokens.main(CountTokens.java:6)
```
Output to files (6.4)

- **PrintStream**: An object in the `java.io` package that lets you print output to a destination such as a file.
  - Any methods you have used on `System.out` (such as `print`, `println`) will work on a `PrintStream`.

- **Syntax**:

  ```java
  PrintStream name = new PrintStream(new File("file name"));
  ```

- **Example**:

  ```java
  PrintStream output = new PrintStream(new File("out.txt"));
  output.println("Hello, file!");
  output.println("This is a second line of output.");
  ```
System.out and PrintStream

- The console output object, System.out, is a PrintStream.

```java
PrintStream out1 = System.out;
PrintStream out2 = new PrintStream(new File("data.txt"));
out1.println("Hello, console!");  // goes to console
out2.println("Hello, file!");    // goes to file
```

- A reference to it can be stored in a PrintStream variable.
  - Printing to that variable causes console output to appear.

- You can pass System.out as a parameter to a method expecting a PrintStream.
  - Allows methods that can send output to the console or a file.
Arrays (7.1)

- **array**: object that stores many values of the same type.
- **element**: One value in an array.
- **index**: A 0-based integer to access an element from an array.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>12</td>
<td>49</td>
<td>-2</td>
<td>26</td>
<td>5</td>
<td>17</td>
<td>-6</td>
<td>84</td>
<td>72</td>
<td>3</td>
</tr>
</tbody>
</table>

- element 0
- element 4
- element 9
Array declaration

type[] name = new type[length];

- Example:
  int[] numbers = new int[10];

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Accessing elements

name[index] // access
name[index] = value; // modify

- Example:

numbers[0] = 27;
numbers[3] = -6;
System.out.println(numbers[0]);
if (numbers[3] < 0) {
    System.out.println("Element 3 is negative.");
}

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>-6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Out-of-bounds

- Legal indexes: between 0 and the array's length - 1.
- Reading or writing any index outside this range will throw an ArrayIndexOutOfBoundsException.

Example:

```java
int[] data = new int[10];
System.out.println(data[0]); // okay
System.out.println(data[9]); // okay
System.out.println(data[-1]); // exception
System.out.println(data[10]); // exception
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
The length field

• An array's length field stores its number of elements.

```java
for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + " ");
}
// output: 0 2 4 6 8 10 12 14
```

• It does not use parentheses like a String's .length().
Quick array initialization

type[] name = {value, value, ... value};

- Example:
  int[] numbers = {12, 49, -2, 26, 5, 17, -6};

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>12</td>
<td>49</td>
<td>-2</td>
<td>26</td>
<td>5</td>
<td>17</td>
<td>-6</td>
</tr>
</tbody>
</table>

- Useful when you know what the array's elements will be.
- The compiler figures out the size by counting the values.
### The Arrays class

**Class Arrays in package java.util** has useful static methods for manipulating arrays:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binarySearch(array, value)</code></td>
<td>returns the index of the given value in a sorted array (&lt; 0 if not found)</td>
</tr>
<tr>
<td><code>equals(array1, array2)</code></td>
<td>returns <code>true</code> if the two arrays contain the same elements in the same order</td>
</tr>
<tr>
<td><code>fill(array, value)</code></td>
<td>sets every element in the array to have the given value</td>
</tr>
<tr>
<td><code>sort(array)</code></td>
<td>arranges the elements in the array into ascending order</td>
</tr>
<tr>
<td><code>toString(array)</code></td>
<td>returns a string representing the array, such as &quot;[10, 30, 17]&quot;</td>
</tr>
</tbody>
</table>
Arrays as parameters

- Declaration:
  ```java
  public static type methodName(type[] name) {
  ```

  - Example:
    ```java
    public static double average(int[] numbers) {
    ```
    ```java
      ...
    ```
  }

- Call:
  ```java
  methodName(arrayName);
  ```

- Example:
  ```java
  int[] scores = {13, 17, 12, 15, 11};
  double avg = average(scores);
  ```
Arrays as return

• Declaring:
  ```java
  public static type[] methodName(parameters) {
    // Example:
    public static int[] countDigits(int n) {
      int[] counts = new int[10];
      ...
      return counts;
    }
  }
  ```

• Calling:
  ```java
  type[] name = methodName(parameters);
  ```

• Example:
  ```java
  public static void main(String[] args) {
    int[] tally = countDigits(229231007);
    System.out.println(Arrays.toString(tally));
  }
  ```
Value semantics (primitives)

- **value semantics**: Behavior where values are copied when assigned to each other or passed as parameters.
  - When one primitive variable is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

```java
int x = 5;
int y = x;    // x = 5, y = 5
y = 17;      // x = 5, y = 17
x = 8;       // x = 8, y = 17
```

- $x$ is a primitive variable.
- $y$ is a primitive variable.
- When $x$ is assigned to $y$, the value of $x$ is copied.
- Modifying the value of $x$ does not affect $y$.
- When $y$ is modified, the value of $x$ remains unchanged.

Reference semantics (objects)

- **reference semantics**: Behavior where variables actually store the address of an object in memory.
  - When one reference variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.
  - Modifying the value of one variable will affect others.

```java
int[] a1 = {4, 5, 2, 12, 14, 14, 9};
int[] a2 = a1;      // refer to same array as a1
a2[0] = 7;
System.out.println(a1[0]); // 7
```

```
<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>
```
Null

- **null**: A reference that does not refer to any object.
  - Fields of an object that refer to objects are initialized to `null`.
  - The elements of an array of objects are initialized to `null`.

```java
String[] words = new String[5];
DrawingPanel[] windows = new DrawingPanel[3];
```
Null pointer exception

- **dereference**: To access data or methods of an object with the dot notation, such as `s.length()`.
- It is illegal to dereference `null` (causes an exception).
- `null` is not any object, so it has no methods or data.

```java
String[] words = new String[5];
System.out.println("word is: " + words[0]);
words[0] = words[0].toUpperCase();
```

Output:
word is: null
Exception in thread "main"
java.lang.NullPointerException
    at Example.main(Example.java:8)
Classes and objects (8.1)

- **class**: A program entity that represents either:
  1. A program / module, or
  2. A template for a new type of objects.

- The `DrawingPanel` class is a template for creating `DrawingPanel` objects.

- **object**: An entity that combines state and behavior.
  - **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.
Fields (8.2)

- **field**: A variable inside an object that is part of its state.
  - Each object has *its own copy* of each field.
  - **encapsulation**: Declaring fields `private` to hide their data.

- Declaration syntax:

  ```java
  private type name;
  ```

- Example:

  ```java
  public class Student {
      private String name; // each object now has
      private double gpa; // a name and gpa field
  }
  ```
Instance methods

- **instance method**: One that exists inside each object of a class and defines behavior of that object.

  ```java
  public type name(parameters) {
      statements;
  }
  ```

- same syntax as static methods, but without `static` keyword

Example:

```java
public void shout() {
    System.out.println("HELLO THERE!");
}
```
A Point class

```java
public class Point {
    private int x;
    private int y;

    // Changes the location of this Point object.
    public void draw(Graphics g) {
        g.fillOval(x, y, 3, 3);
        g.drawString("(" + x + ", " + y + ")", x, y);
    }
}
```

- Each Point object contains data fields named x and y.
- Each Point object contains a method named draw that draws that point at its current x/y position.
The implicit parameter

- **implicit parameter**: The object on which an instance method is called.
  - During the call `p1.draw(g);` the object referred to by `p1` is the implicit parameter.
  - During the call `p2.draw(g);` the object referred to by `p2` is the implicit parameter.

- The instance method can refer to that object's fields.
  - We say that it executes in the *context* of a particular object.
  - `draw` can refer to the `x` and `y` of the object it was called on.
Kinds of methods

- Instance methods take advantage of an object's state.
  - Some methods allow clients to access/modify its state.

- **accessor**: A method that lets clients examine object state.
  - Example: A `distanceFromOrigin` method that tells how far a `Point` is away from (0, 0).
  - Accessors often have a non-`void` return type.

- **mutator**: A method that modifies an object's state.
  - Example: A `translate` method that shifts the position of a `Point` by a given amount.
Constructors (8.4)

- **constructor**: Initializes the state of new objects.

  ```java
  public type(parameters) {
  statements;
  }
  ```

  **Example:**
  ```java
  public Point(int initialX, int initialY) {
  x = initialX;
  y = initialY;
  }
  ```

  - runs when the client uses the `new` keyword
  - does not specify a return type; implicitly returns a new object
  - If a class has no constructor, Java gives it a `default constructor` with no parameters that sets all fields to 0.
**toString method (8.6)**

- tells Java how to convert an object into a String
  ```java
  public String toString() {
      code that returns a suitable String;
  }
  ```
- Example:
  ```java
  public String toString() {
      return "(" + x + ", " + y + ")";
  }
  ```
- called when an object is printed/concatenated to a String:
  ```java
  Point p1 = new Point(7, 2);
  System.out.println("p1: " + p1);
  ```
- Every class has a `toString`, even if it isn't in your code.
  - Default is class's name and a hex number: `Point@9e8c34`
**this keyword (8.7)**

- **this**: A reference to the implicit parameter.
  - *implicit parameter*: object on which a method is called

- Syntax for using **this**:
  - To refer to a field:
    ```java
    this.field
    ```
  - To call a method:
    ```java
    this.method(parameters);
    ```
  - To call a constructor from another constructor:
    ```java
    this(parameters);
    ```
Static methods

- **static method**: Part of a class, not part of an object.
  - shared by all objects of that class
  - good for code related to a class but not to each object's state
  - does not understand the *implicit parameter*, this; therefore, cannot access an object's fields directly
  - if public, can be called from inside or outside the class

- Declaration syntax:

  ```java
  public static type name(parameters) {
    statements;
  }
  ```
Inheritance (9.1)

- **inheritance**: A way to form new classes based on existing classes, taking on their attributes/behavior.
  - a way to group related classes
  - a way to share code between two or more classes

- One class can *extend* another, absorbing its data/behavior.
  - ** superclass**: The parent class that is being extended.
  - **subclass**: The child class that extends the superclass and inherits its behavior.
    - Subclass gets a copy of every field and method from superclass
Inheritance syntax (9.1)

```java
public class name extends superclass {

    // Example:
    public class Secretary extends Employee {
        ...
    }

    // By extending Employee, each Secretary object now:
    - receives a `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` method automatically
    - can be treated as an `Employee` by client code (seen later)
```
Overriding methods (9.1)

- **override**: To write a new version of a method in a subclass that replaces the superclass's version.
- No special syntax required to override a superclass method. Just write a new version of it in the subclass.

```java
public class Secretary extends Employee {
    // overrides getVacationForm in Employee class
    public String getVacationForm() {
        return "pink";
    }
    ...
}
```
super keyword (9.3)

- Subclasses can call overridden methods with `super`

  `super. method(parameters)`

- Example:

  ```java
  public class LegalSecretary extends Secretary {
      public double getSalary() {
          double baseSalary = super.getSalary();
          return baseSalary + 5000.0;
      }
  }
  ```
Polymorphism

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.
  - Example: `System.out.println` can print any type of object.
    - Each one displays in its own way on the console.

- A variable of type $T$ can hold an object of any subclass of $T$.

  ```java
  Employee ed = new LegalSecretary();
  ```

  - You can call any methods from `Employee` on `ed`.
  - You can *not* call any methods specific to `LegalSecretary`.

- When a method is called, it behaves as a `LegalSecretary`.

  ```java
  System.out.println(ed.getSalary()); // 55000.0
  System.out.println(ed.getVacationForm()); // pink
  ```