Compiling and executing

In IDEs like NetBeans and Eclipse compilation can be dynamic as you edit the source code. Compilation on the command line:

```
javac classname.java
```

produces bytecode named `classname.class`

Program is executed within an IDE via menu entry, a shortcut or a button on the interface:

- **NetBeans**: Run->Run Project, F6, 
- **Eclipse**: Run->Run, <shift><command> F11,

From command line:

```
java classname
```
Errors

Syntax errors
The easiest ones.

- Mistake in language syntax.
  - In IDE line will be marked, detail displayed when hovering over error mark. From command line error displayed in the terminal.

Run time errors
A little more difficult, but usually obvious.

- Mistake that causes program to crash.
  - In IDE error displayed in output window. From command line error displayed in terminal.

Logic errors
Can be really difficult to correct.

- No syntax or run time error, but program produces wrong result.
/*
 * This is a simple Java program
 */

public class Welcome {
    public static void main(String[] args) {
        // Display a message to screen
        System.out.println("Welcome to Java!");
    }
}

- Each class in separate file: ClassName.java
- visibility class ClassName { class body }
- Class body: variables and methods
- Method: visibility type methodName { method body }
Statements

- Each statement terminated by semicolon ;
- Block: statements grouped by enclosing them in brackets

Style / Conventions:

- Indent statements to improve readability
- Space before and after operators
- Blocks can either be *next-line*
  
  ```java
  class Test {
  }
  ```

  or *end-of-line*
  
  ```java
  class Test {
  }
  ```
Identifiers

- Class, variable and method names
- Alphanumeric ('a–z', 'A–Z' and '0–9'), _ and $
  $ Cannot start with digit
- Reserved words: public, class, [main] (appendix A)

**Style / Conventions:**
- Use descriptive names, avoid abbreviations
  
  numberOfStudents
  daysInWeek
- Class names start with capital letter, others start with lowercase
- Medial capitalization or *CamelCase*: each new word capitalized
- Constants all caps with words separated by underscore _
Comments

Comments make code more readable. Comments allow others to understand the logic involved. This does not only simplify the work for colleagues, but simplifies the work for you when you are working on someone else's code or your own.

Each file should start with comments explaining what the code does. For this class the first set of comments should be your name, class and section and assignment number.

Comment block starts with /* and ends with */. It is good practice to start each line in the comment block with *.

    /*
     *  Line in comment block
     */

Comment lines start with //

    // Comment line

Javadoc allows to generate API documentation in HTML format from Java code. The documentation is contained in a comment block that starts with /**. 
Variables
Used to store values that might change during the running of the program. Can be initialize when declared.

\[
\text{type name } = \text{ initial value}\], \text{name};
\]

Constants
Used to store values that will not change. **Must be initialized.**

\[
\text{final type name } = \text{ value};
\]

**Primitive data types:**

- **byte**: 8-bit signed, \(-2^7 \text{ to } 2^7-1\)
- **short**: 16-bit signed, \(-2^{15} \text{ to } 2^{15}-1\)
- **int**: 32-bit signed, \(-2^{31} \text{ to } 2^{31}-1\)
- **long**: 64-bit signed, \(-2^{63} \text{ to } 2^{63}-1\)
- **float**: 32-bit IEEE
- **double**: 64-bit IEEE
- **char**: 16-bit Unicode, \'\u0000\' to \'\uffff\'
- **boolean**: true or false
Numerical Operators

Standard arithmetic operators:

+   addition
−   subtraction
*   multiplication
/   division

Additional operator:

%   remainder (mod)

Precedence:

1. Expressions in parenthesis.
2. Multiplication, division, remainder – left-to-right.
3. Addition, subtraction – left-to-right.
Console Output

To display output to console use

```
System.out.print
or
System.out.println  // append new line
```

The method can handle all primitive data types and String.

```
System.out.println(“Test”);
System.out.println(1 + 3);
```
Console Input

To read from console use Scanner class:

1. Initialize scanner with

   ```java
   Scanner scan = new Scanner(System.in);
   ```

2. Read next item with

   ```java
   next()                 - String token(*)
   nextBoolean()         - boolean
   nextDouble()          - double
   nextFloat()           - float
   nextInt()             - int
   nextLine()            - String
   nextLong()            - long
   ```

   ```java
   int intValue = scan.nextInt();
   ```

(*) Token is string terminated by `<SPACE>`,`<TAB>`,`<CR>` and other termination characters (see documentation for details).
Augmented assignment operators

Operator combined with `=` is short-hand for operator, followed by assignment

+ = variable += 1; // add and store in place
- = variable -= 1; // subtract and store in place
* = variable *= 2; // multiply and store in place
/ = variable /= 2; // divide and store in place
% = variable %= 3; // mod and store in place

Note: no space between operator and `=`
Increment and Decrement operators

Special operators `++` and `--` increment and decrement variable.

Pre-increment

```plaintext
++var
temp = ++var;  // var += 1; temp = var;
```

Post-increment

```plaintext
var++
temp = var++;  // temp = var; var += 1;
```

Pre-decrement

```plaintext
--var
temp = --var;  // var -= 1; temp = var;
```

Post-decrement

```plaintext
var--
temp = var--;  // temp = var; var -= 1;
```
Numerical Literals

A *literal* is a constant value that appears directly in the program. They are used when initializing variables and constants and as assignments to variables through the code.

```java
int intValue = 3;
long longValue = 1000000L;
double doubleValue = 3.5;
float floatValue = 4.73F;
boolean boolValue = true;

boolValue = false;
```

Value cannot exceed maximum capacity of data type:

```java
short shortValue = 16777216; // compiler error
```

Integer literal is of type *int*. To specify an integer of *long* type, append an `L` (lowercase `l` is allowed, but it's discouraged).

Real literal is of type *double*. To specify a real of *float* type, append an `F` (or lowercase `f`). You can specify a *double* type by appending a `D` (or lowercase `d`).
Conversion Rules

When performing an operation involving two operands of different types, Java automatically promotes (widens) the operand based on the following rules:

1. If one of the operands is a `double`, the other is converted to `double`.
2. If one operand is a `float`, the other is converted into `float`.
3. If one operand is a `long`, the other is converted into `long`.
4. Both operands are converted into `int`.

In assignment statements, the right-hand side is promoted as long as the left-hand side is wider:

```java
long longVar = 10;  // int 10 promoted to long
int intVar = 123L;  // Compiler error
```
Type casting

A value can be forced to fit in a narrower type by using explicit casting:

```java
int intValue = (int)3.9;       // real part truncated
float floatValue = (float)5.732;
```

Some castings are not meaningful and will give unexpected results:

```java
long shortVar = (short)1000000;  // what is the value ?
```
Output In A Dialog

To display values in a dialog use method `showMessageDialog` of the `JOptionPane` class. Two basic versions:

```java
JOptionPane.showMessageDialog(null, value);

JOptionPane.showMessageDialog(null, 
    value,
    title,
    JOptionPane.INFORMATION_MESSAGE);
```

In order to use the `JOptionPane` class you need to import it. Either just this class or the whole `javax.swing` package:

```java
import javax.swing.JOptionPane;
or
import javax.swing.*;
```

Your IDE should add the import statement automatically.
Input In A Dialog

To read values in a *dialog* use method `showInputDialog` of the `JOptionPane` class. Two basic versions:

```java
JOptionPane.showMessageDialog(message);

JOptionPane.showMessageDialog(null,
                       message,
                       title,
                       JOptionPane.QUESTION_MESSAGE);
```

In order to use the `JOptionPane` class you need to import it.

`showMessageDialog` returns a String.

```java
String answer = JOptionPane.showMessageDialog("Value: ");
```
Convenience type classes

For each primitive data type, the Java library contains convenience wrapper classes that perform conversion and other operations on the value.

To convert from a String to a primitive data type you can use the parse methods available in the convenience classes:

```java
String answer = JOptionPane.showMessageDialog("Value: ");
int intValue = Integer.parseInt(answer);
double doubleValue = Double.parseDouble(answer);
```
Character Data Type

Characters are stored as integers which are mapped using an encoding scheme. Java follows the Unicode encoding scheme. The first 16 bits are used for the standard Unicode characters. The first 128 characters are the ASCII (American Standard Code for Information Interchange) characters, roughly those that are present on your keyboard. Appendix B of your book has the full ASCII character set.

Representation

Characters can be represented using three different methods:

- The 4 digit hexadecimal Unicode value preceded by \u
- The decimal or hexadecimal value cast to a char
- The character enclosed in single quotes (')

```java
char letterA = '\u0041';       // Unicode
char letterA = (char)65;       // Decimal value
char letterA = (char)0X41;     // Hexadecimal value
char letterA = 'A';            // The ASCII character
```
Escape characters

The special character \ (backslash) indicates an escape character, a character that has special meaning:

\b  Backspace      \u0008
\t  Tab            \u0009
\n  Linefeed       \u000A
\f  Formfeed      \u000C
\r  Carriage Return \u000D
\\  Backslash      \u005C
""  Double Quote   \u0022

Character Operations

Since characters are integers, we can perform arithmetic operations on them:

```java
char letter = 'A';   // ASCII A
++letter;            // ASCII B
```
A string of characters can be handled using the String class. The class contains a number of methods that allow for manipulation of the character string:

```java
String theString = "Test it";
boolean equal = theString.equalsIgnoreCase("test it");
int index = theString.lastIndexOf("st");
String upper = theString.toUpperCase();
```

Addition operator can be used to concatenate strings. Augmented addition operator works too:

```java
String theString = "Starting";
theString = theString + " string";
theString += " for exercise";
```
Imports

For the compiler to find the class being referenced, that class needs to be *imported*. You can import the specific class or the whole package containing the class

```java
import java.util.*; // whole package
import java.util.Scanner; // just the Scanner class
```

It is usually best to include the specific class.