Java: Additional Notes and Tips to Koushik's course - Java Essentials

General Facts

- First version released in 1995
- Initially developed by Sun Microsystems. In 2010, Oracle acquired Sun Microsystems, including Java platform.
- Compiled code (byte code) is not machine-executable, need to be translated again into machine code
- Platform independent compiled code (C, C++ need different compilers for OS's)
- Platform dependent runtime (C, C++ don't need runtime)
- Object-oriented language
- Familiar to C syntax

Java Development and Execution

Java byte code

- Instruction set meant for execution by JVM

JVM

- Required for running byte code
- Platform dependent
- Contains byte code loader, verifier, interpreter, and Just-In-Time (JIT) compiler
  - **Interpreter** executes byte code instructions directly, translating byte code (.class) line-by-line into machine code, and hand it to the OS.
  - **JIT compiler** detects frequently executed sections of byte code (hotspots) at runtime, pre-compile them into machine code in advance before the line is reached by the interpreter, and place them in the memory for ease of access. This resulting in better performance compared to using interpreter alone.

JRE

- Required for running Java applications
- Platform dependent
- Includes JVM

- Contains Java standard libraries
- Language agnostic: can run Groovy, Scala, Kotlin

JDK

- Required for Java development
- Platform dependent
- Includes JRE
- Contains tools and utilities for development, compiling, packaging, distribution, etc.

Pros and Cons of JVM model

Pros

- Platform independent compiled code
- Secure: JVM acts as a sandbox. It establishes boundaries between the code and the machine, preventing the code from accessing or manipulating dangerous parts of the machine.

Cons

- A bit slow start up due to line-by-line interpretation (compensated with JIT optimization after start up)

GraalVM

- GraalVM is a full-scale JDK distribution that follows the JVM specification and offers more.
- GraalVM JDK can simply be used as a drop-in replacement for other JDKs.
- Offers two ways of optimizing JVM (refer to the official link) as:

1. **Option 1: running compiled byte code using GraalVM as VM**
   - It uses the Java HotSpot VM as platform.
   - GraalVM replaces the last-tier optimizing compiler in the HotSpot JVM (C2) with the Graal compiler, which includes several new optimizations.
   - GraalVM is itself written in Java (in contrary to Java Hotspot VM which is written in C, C++) which accelerates maintenance and delivering new optimizations.
   - Several companies, such as Oracle Cloud, Twitter, and Facebook are running large-scale Java applications on GraalVM JDK to increase performance, reduce resource usage, and lower deployment costs.

2. **Option 2: running native executables compiled with 'Native Image'**
    - Native Image is a GraalVM tool that convert byte code into native executable of Java applications, that are self-contained and thus no longer require the JVM.
    - The image generation process employs static analysis to find all code reachable from the main Java method and then performs full ahead-of-time (AOT) compilation. At build time, it also performs snapshotting, so, at run time, the application starts much faster with pre-populated heap.
    - The resulting native binary contains the whole program in machine code form that is ready for execution instantly at startup with the following features:
        - Instant startup due to pre-initializing the JDK and user code at build time
        - Reduced memory and CPU usage due to minimized code execution overhead
        - Small packaging due to the AOT approach and slim runtime components
        - Reduced attack surface due to code elimination and the AOT approach

Java Core

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

Special method declaration that is recognized as the entry point for Java program.

`var`

- `var` can only be used for local variables, including loop variables in enhanced for loops and resource variables in try-with-resources statements. It cannot be used for fields, method parameters, or method return types.
- A `var` declaration **must** be initialized at the time of declaration.
- Java compiler automatically **refers** the variable's **static** type at **compile time** based on the type of the initializer expression assigned to the variable.

Primitive Types

- **Integers:** `byte`, `short`, `int`, and `long`
- **Floating Points:** `float` and `double`
- **Character:** `char`
- **Boolean:** `boolean`

`byte`

- 8-bit signed two's complement integer
- The leftmost bit (bit #8) - a.k.a. most significant bit (MSB) - determines whether the number is positive or negative. If MSB is 0, the number is positive or zero. If MSB is 1, the number is negative.
- The rest of 7 bits makes up the number's absolute value.
- For positive numbers, the 7 bits are interpreted straightforwardly.
- For negative numbers, the 7 bits are stored in a special way of signed two's complement.

  1. Take the absolute value of the number.
  2. Convert this absolute value to its binary representation.
  3. Flip all the bits (change 0 to 1 and 1 to 0) – this is called the one's complement.
  4. Add 1 to the result.

  Example, to find the two's complement of -5:

  1. Absolute value: 5
  2. Binary of 5: 00000101
  3. One's complement (flip bits): 11111010
  4. Add 1: 11111011

  ```
  jshell> Integer.toBinaryString(5)
  $1 ==> "101"

  jshell> Integer.toBinaryString(-5)
  $2 ==> "11111111111111111111111111111011"
  ```

- The maximum value is `Byte.MAX_VALUE` $= 2^7 - 1 = 127$
- The minimum value is `Byte.MIN_VALUE` $= -2^7 = -128$
- No actual literals, need casting: `(byte) 0x0A` for `10`

`short`

- 16-bit signed two's complement integer
- Follows MSB principle
- The maximum value is `Short.MAX_VALUE` $= 2^{15} - 1 = 32,767$
- The minimum value is `Short.MIN_VALUE` $= -2^{15} = -32,768$
- No actual literals, need casting: `(short) 0123` for `83`

`int`

- 32-bit signed two's complement integer
- Follows MSB principle
- The maximum value is `Integer.MAX_VALUE` $= 2^{31} - 1 = 2,147,483,647$ (billions)
- The minimum value is `Integer.MIN_VALUE` $= -2^{31} = -2,147,483,648$
- Decimal literals: `0` for 0, `1101` for $1,101$
- Literal separator: `_` anywhere but at the start and the end, e.g., `1_101` for $1,101$
- Octal literals: `00` for 0, `01` for 1, `010` for 8
- Binary literals: `0b0` for 0, `0b1010` for 10, `0b10_1010` for 42, `0b1111_1111_1111_1111` for $65,535$ (can use both `b` or `B`)
- Hexadecimal literals: `0x0` for 0, `0xFF` for 255, `0x7fff_ffff` for $2^{31} - 1$ (can use both `x` or `X`)

`long`

- 64-bit signed two's complement integer
- Follows MSB principle
- The maximum value is `Long.MAX_VALUE` $= 2^{63} - 1 = 9,223,372,036,854,775,807$ (quintillion - 6 commas)
- The minimum value is `Long.MIN_VALUE` $= -2^{63} = -9,223,372,036,854,775,808$
- Literal separator, decimal, octal, binary, and hexadecimal literals are the same as `int` but..
- Long literals have `L` or `l` trailing.
- Hexadecimal literal: `0x7fff_ffff_ffff_ffffL` for $2^{63} - 1$

`float`

- 32-bit single-precision floating-point number
- Follows MSB principle
- 23 rightmost bits (bit 22-0) represent the fractional part of the numbers (Significand/Mantissa).
- 8 in-between bits (bit 30-23) represent the exponent of the number.
- The maximum value is `Float.MAX_VALUE` $= 3.4028235 \times 10^{38}$.
- The minimum value is `-Float.MAX_VALUE` $= -3.4028235 \times 10^{38}$.

- The smallest positive value is `Float.MIN_VALUE` $= 2^{-149} \approx 1.4 \times 10^{-45}$.
- Literal separator and Exponential notation literal are the same as `double`.
- Float literals must have `F` or `f` trailing.
- Cast hexadecimal literal: `0x0` for 0.0, `0x0f` for $1f.0$.

`double`

- 64-bit double-precision floating-point number
- 52 rightmost bits (bit 51-0) represent the fractional part of the numbers (Significand/Mantissa).
- 11 in-between bits (bit 62-52) represent the exponent of the number.
- The maximum value is `Double.MAX_VALUE` $= 1.7976931348623157 \times 10^{308}$
- The minimum value is `-Double.MAX_VALUE` $= -1.7976931348623157 \times 10^{308}$
- The smallest positive value is `Double.MIN_VALUE` $= 2^{-1074} \approx 4.94065645841247 \times 10^{-324}$
- Double literals have `D` or `d` trailing.
- Literal separator: `_` as in `int` but also not allowed near the decimal point, e.g., `123_0.1_2_0` for 1230.12
- Exponential notation literal: `123_0.1_2_0E-02` for 12.3012 (can use both `E` or `e`)
- Cast hexadecimal literal: `0x0` for 0.0, `0x0d` for 13.0

`char`

- Stores a single character
- 16-bit unsigned integer under the hood, corresponding to Unicode code units
- The maximum value is `Character.MAX_VALUE` = `\uffff` $= 2^{16} - 1 = 65,535$
- The minimum value is `Character.MIN_VALUE` = `\u0000` $= 0$
- Character literal: `'a'`
- Unicode escape sequence: `'\u0000'` (must have four hexadecimal digits)
- Escape sequences: `'\n'` for 'newline', `'\t'` for 'tab', `'\''` for single quote, `'\\'` for backslash
- Integer literals (representing Unicode value): `65` for `'a'`, `0101` for `'a'`, `0x41` for `'a'`

`boolean`

- Two possible values: `true` and `false`
- JVM encodes boolean array components using 1 to represent `true` and 0 to represent `false`.
- Compilers map boolean values to JVM type of int.
- Literals: `true` and `false`

Strongly static typing

- The declared variable's type and the assigned value's type should be *compatible*.
- Typically, smaller-capacity source can be converted to larger-capacity destination.

```
// This is fine because int can be converted to double
int i = 123;
double d = i;

// This is not fine
double d = 10;
int i = d;
```

- For smaller-capacity destination, it is incompatible and therefore needs casting.

Casting

- An unary operator used for explicit type conversion
- Has the syntax of `(<destination_type>) <source>`
- Java can squeeze certain sources into the destination to make it work even if it is not ideal, causing **lossy conversion**.
- Throws an exception at runtime when the runtime cannot 'make it work'

```
// Lossy conversion, works but not accurate
double d = 10.5;
int i = (int) d; // Get i = 10
```

Precision Loss

- Sometimes, the assigned source are detected as compatible with the destination but it actually is *not*.

```
int i = 3 / 2;
double d = 3 / 2;
```

```
// i = d = 1 because the 3 / 2 expression is an (int / int) which cannot hold fractions
resulting in 1 evaluated value, regardless of the declared type.
```

Automatic Type Promotion

- In an expression with multiple types, the smaller-capacity types are all converted to the largest-capacity type of the expression.

```
double d = 10;
int i = 5;
var c = d * i
// c becomes a double
```

Arrays

- Array's size is static. Once created, its size cannot be changed.
- Cannot be cast from one type of array to another.
- Array Initializer is a shorthand for creating an array at the declaration point:

```
int[] nums = {1, 2, 3};
```

- For multi-dimension array initialization, one must specify the size of the first dimension at least. This allows multi-width columns or jagged array.

```
int[][] nums = new int[2][];
nums[0] = new int[4];
nums[1] = new int[3];
// nums = { {0, 0, 0, 0}, {0, 0, 0} }
```

Variable Memory Allocation

- Primitive types are allocated with some space of the type's size instantly at the variable declaration.
- Arrays, Strings and Objects are allocated with space of the reference only, at the variable declaration, then the actual space for the objects are allocated later when the variable is instantiated or initialized.

Expressions, Statements, and Blocks

Statements

- Controls the sequence of execution of a program.
- Executed for their effect and *do not* have values.
- Some statements contain other statements.

- Some statements contain expressions.
- E.g., `if`, `assert`, `switch`, `while`, `do`, `for`, `break`, `continue`, `return`, `throw`, `synchronized`, `try`, block (inside a method only)
- **Abrupt of completion**: the `break`, `yield`, `continue`, `return` statements and an exception thrown cause a transfer of control that may prevent normal completion of expressions, statements, and blocks.

Blocks

- A *block* consists of a pair of curly braces which might or might not contain *block statements*

  ```
  Block:
  { {BlockStatement} }
  ```

- A *block statement* consists of statements, local variable declarations, and local class and interface declarations within braces.

  ```
  BlockStatement:
    LocalVariableDeclarationStatement
    ClassDeclaration
    Statement
  ```

- E.g., method body, constructor body, initializer, block inside method body

Operators

Arithmetic Operators

- Binary Arithmetic Operators (require two operands): `+`, `-`, `*`, `/`, and `%`
- Unary Arithmetic Operators (require one operand): `!`, `+`, `-`, `++`, and `--` (post/pre increment/decrement or postfix/prefix operators)

The Equality and Relational Operators

- `==`, `!=`, `>`, `>=`, `<`, and `<=`
- `instanceof` - type comparison operator
  - Syntax: `<object_variable> instanceof <class_name>`, evaluates to `boolean`
  - Tests the actual runtime type of objects.
  - Never causes a cast.
  - `null` is not an instance of any types

The Conditional Operators

- `&&`, `||`, and `?:`

Bitwise Operators

- Performs a logical operation on each corresponding bit of two operands.
- Mimics boolean as 1 for `true` and 0 for `false`.
- **Caveat**: all the bit wise operators automatically promote `byte` and `short` to `int` before performing operations.
- **Bitwise AND** (`&`): If both bits are 1, the result bit is 1; otherwise, it's 0.
- **Bitwise OR** (`|`): If at least one bit is 1, the result bit is 1; otherwise, it's 0.
- **Bitwise XOR** (`^`) (exclusive OR): If the bits are different, the result bit is 1; otherwise, it's 0.
- **Bitwise NOT** (`~`): Flips the bits of a single operand. Every 0 becomes 1, and every 1 becomes 0.

```
byte a = 0b1010;
byte b = 0b1001;
// a & b = 0b1000 = 8
// a | b = 0b1011 = 11
// a ^ b = 0b0011 = 3

// ~a != 0b0101
// ~a = 0b1111_0101 = -11

// TODO: why these two methods of finding the value of negative integer from
// two's complement representation yield the same results?
method 1: "minus 1 from a, negating, interpret value, then add minus sign"
          (reverse engineering storing negatives from positives)
method 2: "negating, adding 1, interpret value as a positive integral, then
          add a minus sign" (chatGPT)

proof:
case 1: 10...0 (n number of 0)
  method 1: 01...10 -> 10...01 -> (binary stored of a negative) ->
            01...10 is the positive -> 2^n - 2 -> 2 - 2^n
  method 2: 01...1 -> 10...0 -> looping
```

Bit Shift Operators

- `<number> << <shift_positions>` - signed left shift: shift the binary pattern of the `<number>` to the left for `<shift_positions>`, replacing vacant positions on the right with `0`.

```
jshell> Integer.toBinaryString(8)
$1 ==> "1000"
```

```
jshell> Integer.toBinaryString(8 << 2)
$2 ==> "100000"
```

- `<number> >> <shift_positions>` - signed right shift: shift the binary pattern of the `<number>` to the right for `<shift_positions>`, replacing vacant positions on the left with the sign bit.

```
jshell> Integer.toBinaryString(8 >> 2)
$3 ==> "10"

jshell> Integer.toBinaryString(-8)
$4 ==> "11111111111111111111111111111000"

jshell> Integer.toBinaryString(-8 >> 2)
$5 ==> "11111111111111111111111111111110"
```

- `<number> >>> <shift_positions>` - unsigned right shift: shift the binary pattern of the `<number>` to the right for `<shift_positions>`, replacing vacant positions on the left with `0`.

```
jshell> Integer.toBinaryString(-8 >>> 2)
$6 ==> "11111111111111111111111111111110"

// Only 30 digits is displayed above
// Check if the replacing digits were 0 or not

jshell> String s = "00" + Integer.toBinaryString(-8 >>> 2)
s ==> "00111111111111111111111111111110"

jshell> Integer.parseInt(s, 2) == (-8 >>> 2)
$8 ==> true

jshell> Integer.parseInt(s, 2)
$9 ==> 1073741822

jshell> -8 >>> 2
$10 ==> 1073741822
```

Assignment Operators:

- `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`

Blocks and Variable Scoping

- Variables exist from where they are declared and end when the end of scope is reached.
- Scopes are bound by curly braces `{}` or blocks.

```
public static void main(String[] args) {
  {
    int i = 1;
  }
  System.out.println(i); // This does not compile
}
```

Switch statement

- Regular switch statement:
  - Needs `break` to stop fall-through.
  - Cases can be grouped together.

```
int a = 3;
switch (a) {
  case 0:
    System.out.println("a == 0");
    break;
  case 1:
    System.out.println("a == 1");
    break;
  case 2, 3, 4:
    System.out.println("1 < a < 5");
    break;
  default:
    System.out.println("other");
}
```

- Switch expression:
  - `->` eliminates the need for explicit `break`, preventing bugs.
  - `yield` for returning values in case blocks.

```
int a = 3;
String result;
result = switch (a) {
    case 0 -> {
        result = "a == " + 0;
        yield result;
    }
    case 1 -> {
        result = "a == " + 1;
        yield result;
    }
    case 2, 3, 4 -> "1 < a < 5";
    default -> "other";
};
System.out.println("The result is " + result);
```

For-Each Loop

- Has the following syntax:

```
for (Type elementVariable : iterableOrArray) {
    // Stuff
}
```

- Can be used with `Iterable` objects or arrays.
- Internally uses an `Iterator` for `Iterable` objects.
- Does not allow modifying the `Collection` object during iteration, or else it throws `ConcurrentModificationException`. For such operations, use an Iterator explicitly instead.

`var`

- Used for local variable declaration only.
- Compiler will infer the type of the variable from its initializer.
- The bytecode will contain the fully resolved, explicit type information.

`null`

- The `null` expression that belong to a special *null* type, which has no name.
- Because the *null* type has no name, it is impossible to declare a variable of the *null* type or to cast to the *null* type.
- The `null` reference is the only possible value of an expression of *null* type.
- The `null` reference can always be cast to any reference type.
- In practice, the programmer can just pretend that `null` is merely a special literal that can be of any reference type.
- See more on `null` at this Stack Overflow thread

Variable Shadowing

- A variable declared in an inner scope has the same name as a variable declared in an outer scope.
- The inner variable "shadows" or hides the outer variable, making the outer variable temporarily inaccessible within that specific inner scope.
- For example, subclass - inner scope & superclass - outer scope, method - inner scope & instance - outer scope

`this` keyword

- Can mean two things: a class's constructor and an instance of a class
- As a constructor, `this(<constructor_arguments>)` can only be called in another constructor.
- As an instance, `this` can only be accessed within instance's members: variables, methods, and inner class (needs to refer with `<outer_class_name.this>`).

Copy constructor

- A constructor that takes a parameter of another instance of the same type, and assigns all the new instance's fields with the argument's fields.

Call by Value vs Call by Reference

- Java has only call by value mechanism, which is copying the values of variables into method arguments, *not* passing the memory addresses of the variables themselves.

Nested Classes

Static inner class

- Instantiating an inner class as

```
Outer.Inner inner = new Outer.Inner();
```

Inner Class

- Instantiating an inner class as

```
Outer outer = new Outer();
Outer.Inner inner = outer.new Inner();
```

- Inner object can access members of the enclosing class object - `outer`.

Local Class

- Is a class inside any block: if, loop, method, etc.
- A local class can access local variables and parameters of the enclosing block that are final or *effectively final*. A variable or parameter whose value is never changed after its initialization is effectively final.

```
public static void main(String[] args) {
    int i = 10;

    class Foo {
        int x = i;
```

```
        }

        // i = 20; // This is not allowed as it will make 'i' not effectively final.
    }
```

- Since Java SE 8, if you declare the local class in a method, it can access the method's parameters.

Anonymous Class

- Enable declaration and instantiation of a class at the same time.
- Similar to local classes, but do not have a name.
- Use them if you need to use a local class only once.
- While local classes are class declarations, anonymous classes are expressions, must be part of a statement.
- See this example from Oracle Tutorial

```
public class HelloWorldAnonymousClasses {

    interface HelloWorld {
        public void greet();
        public void greetSomeone(String someone);
    }

    public void sayHello() {
        // Anonymous class declaration and instantiation
        HelloWorld spanishGreeting = new HelloWorld() {
            String name = "mundo";
            public void greet() {
                greetSomeone("mundo");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hola, " + name);
            }
        };
        spanishGreeting.greet();
    }

    public static void main(String... args) {
        HelloWorldAnonymousClasses myApp =
            new HelloWorldAnonymousClasses();
        myApp.sayHello();
    }
}
```

- The anonymous class expression consists of the following:
  1. The new operator
  2. The name of an interface to implement or a class to extend
  3. Parentheses that contain the arguments to a constructor, just like a normal class instantiation.

  Note: When implementing an interface, there is no constructor, so an empty pair of parentheses is used.

Class Inherence

- Subclass cannot have a method with the same name and parameters as the superclass' but with:
  - Weaker-privilege access modifier
  - Different return type
- Subclass' instantiation involves calling the superclass' constructor first, with two different cases:
  - If the invoked subclass' constructor does not call any of superclass' constructor explicitly, the no-args constructor of the superclass will get called.
  - If the invoked subclass' constructor calls a superclass' constructor explicitly, then it's straightforward.

`final`

- Before Java 17, `final` can be used at a class declaration to prevent extensions.
- Can be used at variables declaration to prohibit re-assignments.
- Can be used at methods declaration to prevent overriding by subclasses.

Sealed Class

- `sealed` evolved from `final` on classes, enabling and requiring further inheritance.
- `sealed` classes must have declared `permits` listing at least one subclass. *Why? Because the `final` already does the job otherwise.*
- The subclasses of the sealed class must be declared with one of the three following keywords: `final`, `sealed`, and `non-sealed`.
- `final` is allowed on every classes.
- `non-sealed` is not allowed on classes that do not have a sealed superclass.

Interface

- **Variables** declared in an interface are implicitly `public`, `static`, and `final`.
- **Initializer blocks** are not allowed in an interface.
- **Non-default methods** declared in an interface are implicitly `public` and `abstract`. (`protected` not allowed)
- **Default methods** declared in an interface are implicitly `public` and non-`static`. (`protected` and `private` not allowed)
- **Static methods** declared in an interface are implicitly `public` and *need* bodies. (`protected` not allowed)
- `private` non-default methods and `private` static methods are allowed.

Default Methods Clash - Diamond Problem

- A class implementing more than one interfaces that have default methods of the same method signature.
  1. If the methods have the same return type, the class needs to override the particular methods. However, it can call methods of the interface using `<interface_name>.super.<method_call>`.
  2. If the methods have different return types, the code will not compile, saying that the implemented interfaces have clashing methods. Same goes with *non-default* methods of the same method signature.

`Throwable`

- Top superclass of `Exception`s and `Error`s.
- Contains methods like `getCause()`, `getMessage()`, `printStackTrace()`, etc.

`Exception`

- Exceptions represent conditions that a program can reasonably anticipate and handle.
- Usually are issues rooted from the program's logic or external factors, e.g., file not found, invalid input.
- Designed for catching and handling with `try-catch` blocks, recovering, re-routing executions, or informing users.

Checked Exceptions

- Require handling at compile time: using `try-catch` blocks or `throws`.
- Examples: `IOException`, `SQLException`.

Unchecked Exceptions (Runtime Exceptions)

- Do not require handling at compile time.
- Usually represent issues rooted from runtime factors.
- Examples: `NullPointerException`, `IllegalArgumentException`, `FileNotFoundException`

`Error`

- Errors represent serious, unrecoverable problems that are typically beyond the control of the program.
- Usually indicate issues with the JVM or the environment.
- Designed for logging the issue and terminating the program, not handling.
- Examples: `OutOfMemoryError`, `StackOverflowError`, `NoClassDefFoundError`.

Type Casting

- Performed by compiler.

Implicit (Automatic) Casting

- Also known as widening or upcasting.
- Assigning a smaller data type to a larger data type.
- Safe and not data losable.
- Examples: `int` to `double`, `long` to `float`, `String` to `Object`.

Explicit (Manual) Casting

- Also known as narrowing or down casting.
- Converting a larger data type to a smaller type.
- Can lead to data loss.
- Examples: `double` to `int`.