IT602: Object-Oriented Programming

Lecture - 03

Language Fundamentals

Arpit Rana

25th Jan 2022

Basic Language Elements

A programming language is defined by -

- ***grammar rules*** that specify: how syntactically legal constructs can be formed using the *language elements*
- ***semantic definition*** that specifies the meaning of syntactically legal constructs

Basic Language Elements

Language elements are categorised as -

- ***Low-level language elements*** (*lexical tokens or tokens*) are the building blocks for more complex constructs,

  e.g. *identifiers, keywords, numbers, operators, and special characters.*

- ***High-level language elements*** are formed using low-level elements,

  e.g. *expressions, statements, methods, and classes*

Low-level Language Elements

Identifiers

- A name in a program is called an **identifier**.
- An identifier is used to denote classes, methods, variables, and labels.
  - An identifier is sequence of characters, where each character can be a letter or a digit.
  - The first letter must be a letter, a underscore, or a dollar
  - Underscore (_) and any currency symbol (e.g. $) are also allowed as letters in identifier names.
- e.g. `number, Number, sum_$, $$_100, _007, gru8, all@hands, grand-sum`

Low-level Language Elements

Keywords

- Keywords are reserved words that are predefined in the language and cannot be used to denote other entities.

abstract	default	if	private	this
assert	do	implements	protected	throw
boolean	double	import	public	throws
break	else	instanceof	return	transient
byte	enum	int	short	try
case	extends	interface	static	void
catch	final	long	strictfp	volatile
char	finally	native	super	while
class	float	new	switch	
continue	for	package	synchronized	

Low-level Language Elements

Separators

- These tokens have meaning depending on the context in which they are used.
- e.g. `brackets ([]), parentheses (()), dot operator (.), @, double colon (::), semicolon (;)`

Low-level Language Elements

Literals

- A literal denotes a constant value that remains unchanged in the program. For example -
- `integer literal: 2000, 0, -7, (8, 0b1000, 010, 0x8), 35L (or l) (long integer literal)`
- `floating point literal: 3.14, -3.14, 0.5, 0.49D (or d) (double literal), 0.49F (or f) (Float literal)`

Low-level Language Elements

Literals

- Character (enclosed with single quotes): ‘a’, ‘e’, ‘;’, ‘\n’, ‘\t’ (escape sequences)
- Boolean: true, false
- String (enclosed with double quotes and must be in one line): “abba”, “for”
  - All string literals are objects of class **String**.

Low-level Language Elements

Whitespace

- A whitespace is a sequence of spaces, tabs, form feeds (`'\f'`), and line terminator characters (`'\n', '\r', '\n\r'`) in Java source file.

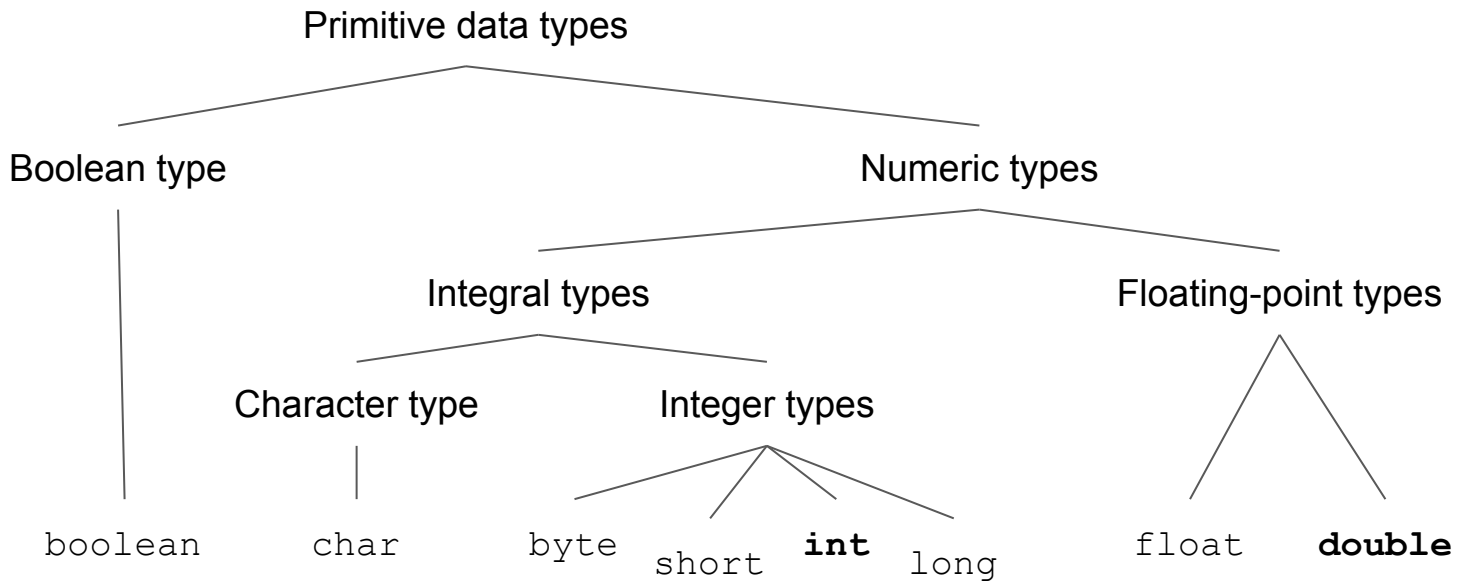
Low-level Language Elements

Comments

- A program can be documented by inserting comments at relevant places in the source code.
- Comments are for documentation purpose only and are ignored by the compiler.
- Java provides three types of comments -
  - `A single-line comment: // . . .`
  - `A multiple-line comment: /* . . . */`
    `(nesting of this type of comment is a CTE)`
  - `A documentation (javadoc) comment: /** . . . */`

Primitive Data Types

Each primitive data type defines **the range of values** in the data type, and **operations on these values** are defined by special operators in the language.



Primitive Data Types

Each primitive data type also has a corresponding *wrapper class* that can be used to represent a primitive value as an object.

Data type	Width (bits)	Minimum value, maximum value	Wrapper class
boolean	not applicable	true, false	Boolean
byte	8	-2^7, 2^7-1	Byte
short	16	-2^{15}, $2^{15}-1$	Short
char	16	0x0, 0xffff	Character
int	32	-2^{31}, $2^{31}-1$	Integer
long	64	-2^{63}, $2^{63}-1$	Long
float	32	±1.40129846432481707e-45f, ±3.40282347663852886e+38f	Float
double	64	±4.94065645841246544e-324, ±1.79769313486231570e+308	Double

Variable Declarations

A `variable` stores a value of a particular type.

- It has a *name*, a *type*, and an associated *value*.
- It can store a value of a primitive data types and reference values of objects (called *reference variables or references*).

Variable Declarations

Declaring and Initializing Variables

- Variable declarations are used to specify the type and the name of variables.
- This determines their memory allocation and the values that can be stored in them.

```
char a, b, c;           // a, b and c are character variables.
double area;            // area is a floating-point variable.
boolean flag;           // flag is a boolean variable.
```

Variable Declarations

Declaring and Initializing Variables

- A declaration can also be combined with an initialization expression to specify an appropriate initial value for the variable.
- Such declarations are called ***declaration statements***.

```
int i = 10,              // i is an int variable with initial value 10.
    j = 0b101;           // j is an int variable with initial value 5.
long big = 2147483648L;  // big is a long variable with specified initial value.
```

Variable Declarations

Reference Variables

- A variable declaration that specifies a ***reference type*** declares a reference variable..
- A reference type can be of **a class, an array, an interface name, or an enum type**.
- A reference variable can be used to ***manipulate*** the object denoted by the reference value.

```
Pizza yummyPizza;   // Variable yummyPizza can reference objects of class Pizza.

Pizza yummyPizza = new Pizza("Hot&Spicy"); // Declaration statement
```

Initial Values for Variables

Default Values for Fields

- If no explicit initialization is provided to variables of primitive data types or to reference variables, they are initialized to **`default`** values as below.

Data type	Default value
`boolean`	`false`
`char`	`'\u0000'`
Integer (`byte`, `short`, `int`, `long`)	`0L` for `long`, `0` for others
Floating-point (`float`, `double`)	`0.0F` or `0.0D`
Reference types	`null`

Initial Values for Variables

Default Values to *`class`* (`static` and `instance`) variables

- **`static`** variables are initialized with the default value of its type when the class is loaded.
- **`instance`** variables are initialized with the default values of their types when the class is instantiated.

```
public class Light {
  // Static variable
  static int counter;      // Default value 0 when class is loaded

  // Instance variables:
  int     noOfWatts = 100; // Explicitly set to 100
  boolean indicator;       // Implicitly set to default value false
  String  location;        // Implicitly set to default value null

  public static void main(String[] args) {
    Light bulb = new Light();
    System.out.println("Static variable counter:     " + Light.counter);
    System.out.println("Instance variable noOfWatts: " + bulb.noOfWatts);
    System.out.println("Instance variable indicator: " + bulb.indicator);
    System.out.println("Instance variable location:  " + bulb.location);
  }
}
```

Initializing Local Variables

Local Variables of Primitive Data Types

- Variables that are declared in methods, constructors, and blocks are called ***local*** variables of that construct.
- Local variables are not initialized implicitly when they are allocated memory at method invocation i.e. when the execution of a method begins. **They must be explicitly initialized before being used.**

```
public class TooSmartClass {
  public static void main(String[] args) {
    int weight = 10, thePrice;                          // (1) Local variables

    if (weight <  10) thePrice = 1000;
    if (weight >  50) thePrice = 5000;
    if (weight >= 10) thePrice = weight*10;             // (2) Always executed
    System.out.println("The price is: " + thePrice);    // (3) Compile-time
error!
  }
}
```

Initializing Local Variables

Local Reference Variables

- Local reference variables are also bound by the same initialization rule. **They must be explicitly initialized before being used.**
- In the example below, initializing `importantMessage` to `null` will generate a *run-time* error as still the variable does not denote any object.

```
public class VerySmartClass {
  public static void main(String[] args) {
    String importantMessage;       // Local reference variable

    System.out.println("The message length is: " +
                       importantMessage.length());  // Compile-time error!
  }
}
```

Lifetime of Variables

Lifetime (also called scope) of a variable is the time a variable is accessible during execution.

- It depends on the context in which the variable is declared.
  - **Instance variables**: members of a class, which are created for each object of the class and exist as long as the object they belong to is in use at runtime.
  - **Static variables**: members of a class, which belong only to the class. They are created when the class is loaded and exist as long as the class is available at runtime.
  - **Local variables**: declared in methods, constructors, and blocks, and are created for each execution of the construct. After the execution of the construct completes, local variables are no longer accessible.

Next lecture -
Operators & Expressions