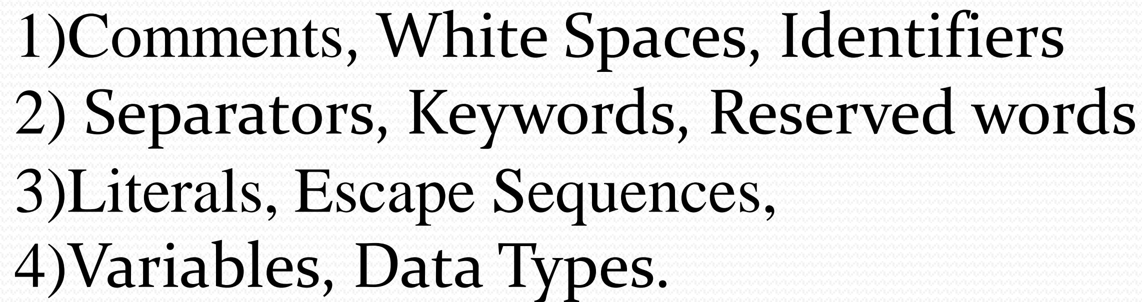
ADVANCED OBJECT ORIENTED PROGRAMMING

Engr. Anees Ahmed Soomro
Assistant Professor
CS QUEST Nawabshah

https://anees-soomro.neocities.org


1)Comments, White Spaces, Identifiers
2) Separators, Keywords, Reserved words
3)Literals, Escape Sequences,
4)Variables, Data Types.

Comments, White Spaces, Identifiers,

Comments

- A program can be documented by inserting comments at relevant places.
- These comments are for documentation purposes and are ignored by the compiler.
- Java provides three types of comments to document a program:
- A single-line comment: // … to the end of the line
- A multiple-line comment: /* … */
- A documentation (Javadoc) comment: /** … */

Single-line Comment

- All characters after the comment-start sequence.
- // through to the end of the line constitute a single-line comment.
- // This comment ends at the end of this line.
- int age;   // From comment-start sequence to the end of the line is a comment.

Multiple-line Comment

- A multiple-line comment, as the name suggests, can span several lines.
- Such a comment starts with /* and ends with */.
- /* A comment on several lines.*/

- The comment-start sequences (//, /*, /**) are not treated differently from other characters when occurring within comments, and are thus ignored. This means trying to nest multiple-line comments will result in compile time error:
- /* Formula for alchemy. gold = wizard.makeGold(stone); /* But it only works on Sundays. */*/ The second occurrence of the comment-start sequence /* is ignored. The last occurrence of the sequence */ in the code is now unmatched, resulting in a syntax error.

Documentation Comment

- A documentation comment is a special-purpose comment that when placed before class or class member declarations can be extracted and used by the javadoc tool to generate HTML documentation for the program.
- Documentation comments are usually placed in front of classes, interfaces, methods and field definitions. Groups of special tags can be used inside a documentation comment to provide more specific information. Such a comment starts with /** and ends with */:

```
/**
*  This class implements a gizmo.
*  @author K.A.M.
*  @version 2.0
*/
```

- For details on the javadoc tool, see the documentation for the tools in the Java 2 SDK.

White Spaces

- A white space is a sequence of spaces, tabs, form feeds, and line terminator characters in a Java source file. Line terminators can be newline, carriage return, or carriage return-newline sequence.
- A Java program is a free-format sequence of characters that is tokenized by the compiler, that is, broken into a stream of tokens for further analysis.
- Separators and operators help to distinguish tokens, but sometimes white space has to be inserted explicitly as separators.
- For example, the identifier **classRoom** will be interpreted as a single token, unless white space is inserted to distinguish the keyword class from the identifier **Room**.
- White space aids not only in separating tokens, but also in formatting the program so that it is easy for humans to read. The compiler ignores the white spaces once the tokens are identified.

Identifiers

- A name in a program is called an identifier. Identifiers can be used to denote classes, methods, variables, and labels.
- In Java an identifier is composed of a sequence of characters, where each character can be either a letter, a digit, a connecting punctuation (such as underscore _), or any currency symbol (such as $, ¢, ¥, or £). However, the first character in an identifier cannot be a digit.
- Since Java programs are written in the Unicode character set (see p. 23), the definitions of letter and digit are interpreted according to this character set.
- Identifiers in Java are case sensitive, for example, **price** and **Price** are two different identifiers.

Examples of Legal Identifiers:

- number, Number, sum_$, bingo, $$_100, mål, grüß

Examples of Illegal Identifiers:

- 48chevy, all@hands, grand-sum
- The name 48chevy is not a legal identifier as it starts with a digit. The character @ is not a legal character in an identifier. It is also not a legal operator so that all@hands cannot not be interpreted as a legal expression with two operands. The character - is also not a legal character in an identifier.
- However, it is a legal operator so grand-sum could be interpreted as a legal expression with two operands.

Escape Sequences

- Certain escape sequences define special character values as shown in Table 2.7.
- These escape sequences can be single-quoted to define character literals.
- For example, the character literals '\t' and '\u0009' are equivalent.
- However, the character literals '\u000a' and '\u000d' should not be used to represent newline and carriage return in the source code.
- These values are interpreted as line-terminator characters by the compiler, and will cause compile time errors.
- One should use the escape sequences '\n' and '\r', respectively, for correct interpretation of these characters in the source code.
- Table 2.7 **Escape Sequence**

Escape Sequence	Unicode Value	Character
\b	\u0008	Backspace (BS)
\t	\u0009	Horizontal tab (HT or TAB)
\n	\u000a	Linefeed (LF) a.k.a., Newline (NL)
\f	\u000c	Form feed (FF)
\r	\u000d	Carriage return (CR)
\'	\u0027	Apostrophe-quote
\"	\u0022	Quotation mark
\\	\u005c	Backslash

- We can also use the escape sequence \ddd to specify a character literal by octal value, where each digit d can be any octal digit (0–7), as shown in Table 2.8. The number of digits must be three or fewer, and the octal value cannot exceed \377, that is, only the first 256 characters can be specified with this notation.
- **Table 2.8. Examples of Escape Sequence \ddd**

Escape Sequence \ddd	Character Literal
'\141'	'a'
'\46'	'&'
'\60'	'0'

Keywords

- Keywords are reserved identifiers that are predefined in the language and cannot be used to denote other entities. All the keywords are in lowercase, and incorrect usage results in compilation errors.
- Keywords currently defined in the language are listed in Table 2.1. In addition, three identifiers are reserved as predefined literals in the language: the null reference and the Boolean literals true and false (see Table 2.2). Keywords currently reserved, but not in use, are listed in Table 2.3. All these reserved words cannot be used as identifiers. The index contains references to relevant sections where currently defined keywords are explained.

Table 2.1. Keywords in Java

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

Table 2.2. Reserved Literals in Java

null	true	false

Table 2.3. Reserved Keywords not Currently in Use

const	goto

Literals

- A literal denotes a constant value, that is, the value a literal represents remains unchanged in the program. Literals represent numerical (integer or floating-point), character, boolean or string values. In addition, there is the literal null that represents the null reference.

Table 2.4. Examples of Literals	
Integer	**2000 0 -7**
Floating-point	**3.14 -3.14 .5 0.5**
Character	**'a' 'A' '0' ':' '-' ')'**
Boolean	**true false**
String	**"abba" "3.14" "for" "a piece of the action"**

Integer Literals

- Integer data types are comprised of the following primitive data types: int, long, byte, and short.
- The default data type of an integer literal is always int, but it can be specified as long by appending the suffix L (or l) to the integer value. Without the suffix, the long literals 2000L and 0l will be interpreted as int literals. There is no direct way to specify a short or a byte literal.
- In addition to the decimal number system, integer literals can also be specified in octal (base 8) and hexadecimal (base 16) number systems. Octal and hexadecimal numbers are specified with 0 and 0x (or 0X) prefix respectively.
- Examples of decimal, octal and hexadecimal literals are shown in Table 2.5. Note that the leading 0 (zero) digit is not the uppercase letter O. The hexadecimal digits from a to f can also be specified with the corresponding uppercase forms (A to F). Negative integers (e.g. -90) can be specified by prefixing the minus sign (-) to the magnitude of the integer regardless of number system (e.g., -0132 or -0X5A).
- Java does not support literals in binary notation.

Table 2.5. Examples of Decimal, Octal, and Hexadecimal Literals

Decimal	Octal	Hexadecimal
8	010	0x8
10L	012L	0XaL
16	020	0x10
27	033	0x1B
90L	0132L	0x5aL
-90	-0132	-0X5A
2147483647 (i.e., 2^{31}-1)	017777777777	0x7fffffff
-2147483648 (i.e., -2^{31})	-020000000000	-0x80000000
1125899906842624L (i.e., 2^{50})	040000000000000000L	0x4000000000000L

Floating-point Literals

- Floating-point data types come in two flavors: float or double.
- The default data type of a floating-point literal is double, but it can be explicitly designated by appending the suffix D (or d) to the value.
- A floating-point literal can also be specified to be a float by appending the suffix F (or f).
- Floating-point literals can also be specified in scientific notation, where E (or e) stands for Exponent. For example, the double literal 194.9E-2 in scientific notation is interpreted as $194.9*10^{-2}$ (i.e., 1.949).

Examples of double Literals

- 0.0 0.0d 0D
- 0.49 .49 .49D
- 49.0 49. 49D
- 4.9E+1 4.9E+1D 4.9e1d 4900e-2 .49E2

Examples of float Literals

- 0.0F 0f 0.49F .49F 49.0F 49.F 49F 4.9E+1F 4900e-2f .49E2F
- Note that the decimal point and the exponent are optional and that at least one digit must be specified.
- **Boolean Literals**
- The primitive data type boolean represents the truth-values true or false that are denoted by the reserved literals true or false, respectively.

Character Literals

- A character literal is quoted in single-quotes (') and have the primitive data type char.
- Characters in Java are represented by the 16-bit Unicode character set, which subsumes the 8-bit ISO-Latin-1 and the 7-bit ASCII characters. In Table 2.6, note that digits (0 to 9), upper-case letters (A to Z), and lower-case letters (a to z) have contiguous Unicode values. Any Unicode character can be specified as a four-digit hexadecimal number (i.e., 16 bits) with the prefix \u.

Table 2.6. Examples of Unicode Values

Character Literal	Char Literal with Unicode value	Character
' '	'\u0020'	Space
'0'	'\u0030'	0
'1'	'\u0031'	1
'9'	'\u0039'	9
'A'	'\u0041'	A
'B'	'\u0042'	B
'Z'	'\u005a'	Z
'a'	'\u0061'	a
'b'	'\u0062'	b
'z'	'\u007a'	z
'Ñ'	'\u0084'	Ñ
'å'	'\u008c'	å
'ß'	'\u00a7'	

Lifetime of Variables

Lifetime of a variable, that is, the time a variable is accessible during execution, is determined by the context in which it is declared. We distinguish between lifetime of variables in three contexts:

Instance variables— members of a class and created for each object of the class. In other words, every object of the class will have its own copies of these variables, which are local to the object. The values of these variables at any given time constitute the state of the object. Instance variables exist as long as the object they belong to exists.

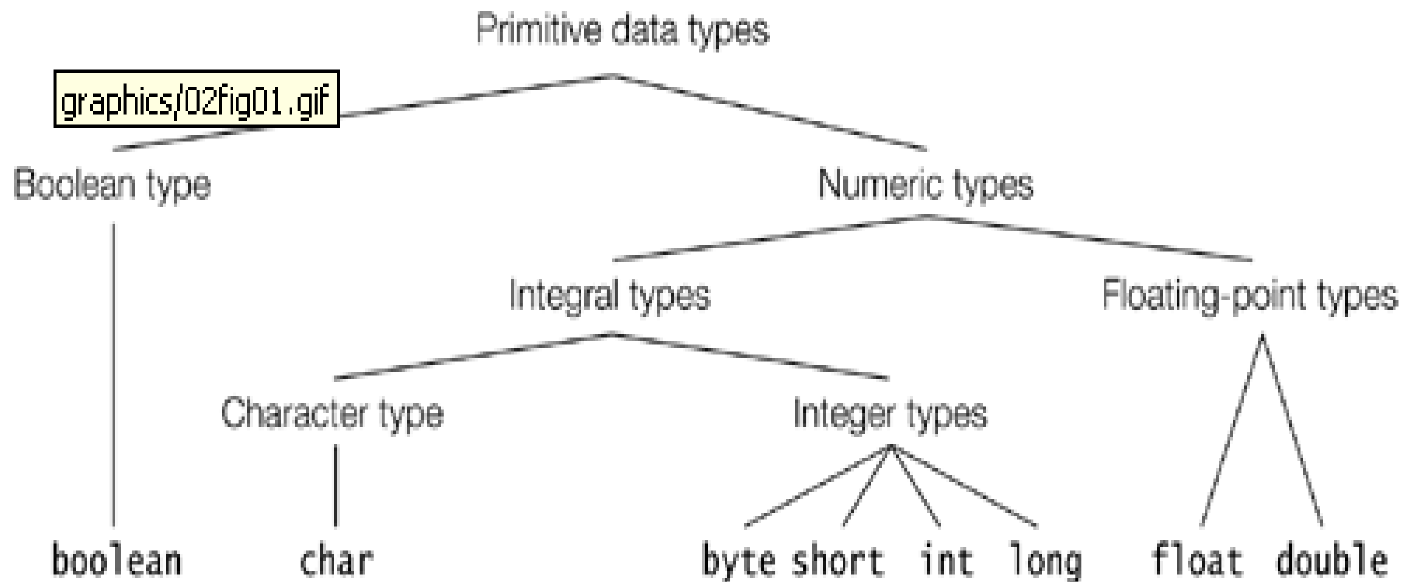
Static variables— also members of a class, but not created for any object of the class and, therefore, belong only to the class (see Section 4.10, p. 144). They are created when the class is loaded at runtime, and exist as long as the class exists.

Local variables (also called method automatic variables)— declared in methods and in blocks and created for each execution of the method or block. After the execution of the method or block completes, local (non-final) variables are no longer accessible.

Primitive Data Types

Figure 2.1 gives an overview of the primitive data types in Java.

Figure 2.1. Primitive Data Types in Java



Primitive data types in Java can be divided into three main categories:

Integer Types

Primitive data types in Java can be divided into three main categories:

Integral types— represent signed integers (byte, short, int, long) and unsigned character values (char)

Floating-point types (float, double)— represent fractional signed numbers

Boolean type (boolean)— represent logical values

Primitive data values are not objects. 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 .

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

Table 2.9. Range of Integer Values

Data Type	Width (bits)	Minimum value MIN_VALUE	Maximum value MAX_VALUE
Byte	8	-2^{7} (-128)	$2^{7}-1$ (+127)
Short	16	-2^{15} (-32768)	$2^{15}-1$ (+32767)
Integer	32	-2^{31} (-2147483648)	$2^{31}-1$ (+2147483647)

Table 2.13. Summary of Primitive Data Types

Data Type	Width (bits)	Minimum Value, Maximum Value	Wrapper Class
boolean	not applicable	true, false (no ordering implied)	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, ±3.40282346638528860e+38f	Float
double	64	\'b14.94065645841246544e-324, \'b11.79769313486231570e+308	Double

Variable Declarations

A variable stores a value of a particular type. A variable has a name, a type, and a value associated with it.

In Java, variables can only store values of primitive data types and references to objects.

Variables that store references to objects are called reference variables.

Declaring and Initializing Variables

Variable declarations are used to specify the type and the name of variables.

This implicitly determines their memory allocation and the values that can be stored in them.

We show some examples of declaring variables that can store primitive values:

```
char a, b, c;            // a, b and c are character variables.
double area;
boolean flag;
// flag is a boolean variable. The first declaration above is equivalent to the
//following three declarations:
char a;char b;char c;
```

A declaration can also include initialization code to specify an appropriate initial value for the variable:

```
int i = 10,              // i is an int variable with initial value 10.
 j = 101;                // j is an int variable with initial value 101.
long big = 2147483648L;  // big is a long variable with specified initial value.
```

Object Reference Variables

An object reference is a value that denotes an object in Java. Such reference values can be stored in variables and used to manipulate the object denoted by the reference value.

The declaration determines what objects a reference variable can denote. Before we can use a reference variable to manipulate an object, it must be declared and initialized with the reference value of the object.

```
Pizza yummyPizza;   // Variable yummyPizza can reference objects of class Pizza.
Hamburger bigOne,   // Variable bigOne can reference objects of class
Hamburger smallOne; // and so can variable smallOne.
```

```
int a,b;
a= b=40;
```