
The Java Language Rules
Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/Core-Java

Lexical Elements

- Java source code consists of words or symbols called ***lexical elements*** or ***tokens***.
- Include line terminators, whitespace, comments, keywords, identifiers, separators, operators, and literals.
- The words or symbols in the Java programming language are comprised of the Unicode character set.

Unicode

- Is the universal character set with the first 128 characters being the same as those in the ASCII, American Standard Code for Information Interchange.
- Provides a unique number for every character, given all platforms, programs, and languages.
- Unicode, see also
  - https://home.unicode.org/

Unicode (Cont.)

Printable ASCII Characters

- ASCII reserves code 32 (spaces) and codes 33 to 126(letters, digits, punctuation marks, and a few others) for printable characters.

32	SP	48	0	64	@	80	P	96	‘	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	“	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	‘	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o		

Non-Printable ASCII Characters

- ASCII reserves decimal numbers 0-31 and 127 for control characters.

00	NUL	07	BEL	14	SO	21	NAK	28	FS
01	SOH	08	BS	15	SI	22	SYN	29	GS
02	STX	09	HT	16	DLE	23	ETB	30	RS
03	ETX	10	NL	17	DC1	24	CAN	31	US
04	EOT	11	VT	18	DC2	25	EM	127	DEL
05	ENQ	12	NP	19	DC3	26	SUB		
06	ACK	13	CR	20	DC4	27	ESC		

- ASCII 10 is a newline or linefeed.
- ASCII 13 is a carriage return.

Comments

- A single-line comment begins with two forward slashes and ends immediately before the line terminator character.

```
//  A comment on a single line
```

- A multiple-line comment begins with a forward slash, immediately followed by an asterisk, and ends with an asterisk immediately followed by a forward slash.

```
/*   A comment that can span

multiple lines just like this   */
```

- In Java, comments cannot be nested.

```
/* This is /* not permissible */ in Java */
```

Comments - JavaDoc

- Is a documentation generator from Sun Microsystems.
- Generate API documentation in HTML format from Java source code.

```
/**     documenting comment    */
```

- Refer to
  - Windows : docs\technotes\tools\windows\javadoc.html
  - Solaris or Linux : docs/technotes/tools/solaris/javadoc.html
  - http://www.javaexpert.co.kr/entry/19

Comments – JavaDoc Cont.

```
1  import java.util.Date;
2
3  /**
4   * JavaDoc Demo Class
5   * @author JavaMan
6   * @version 1.0
7   */
8  public class JavaDocDemo {
9      /**
10      * Java Application Entry Point Method
11      * @param args String array
12      * @return No return value
13      */
14     public static void main(String[] args) {
15         System.out.println("Hello, Today is : ");
16         System.out.println(new Date());
17     }
18 }
```

All Classes

JavaDocDemo

Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class JavaDocDemo

java.lang.Object
  └ **JavaDocDemo**

public class **JavaDocDemo**
extends java.lang.Object

JavaDoc Demo Class

Version:
1.0

Author:
JavaMan

Constructor Summary

JavaDocDemo()

```
$ javadoc -author -version -d ./docs JavaDocDemo.java
```

Separators

- Nine ASCII characters delimit program parts.
- ( ), { }, and [ ] are used in pairs.

```
(   )   {   }   [   ]   ;   ,   .
```

Statements

- Small, logical actions taken on a method.
- Is a single command that performs some activity when executed by the Java interpreter.
- A single line of code terminated by a semicolon(`;`)
- Include expression, empty, block, conditional, iteration, transfer of control, exception handling, variable, labeled, assert, and synchronized.
- Reserved Java words used in statements are **`if`**, **`else`**, **`switch`**, **`while`**, **`do`**, **`for`**, **`for/in`**, **`break`**, **`continue`**, **`return`**, **`synchronized`**, **`throw`**, **`try`**, **`catch`**, **`finally`**, and **`assert`**.

```
total = a + b + c + d + e + f;
```

Blocks

- A group of statements is called a block or statement block.
- A block of statements is enclosed in curly braces({ and }).
- Variables and classes declared in the block are called local variables and local classes.
- In blocks, one statement is interpreted at a time in the order in which it was written or in the order of flow control.
- Can nest block statements.

Whitespaces

- Characters within the source code that do not influence the program's execution
- Any amount of whitespace is allowed in a Java program.
- e.g.

    spaces, tabs, newlines,… etc.

Identifiers

- Are names given to a variable, class, or method.
- Are made up of Java letters[1].
- The first character of an identifier must be **`a Unicode letter`**, underscore(**`_`**), or dollar sign(**`$`**).
- The second and subsequent characters must be **`a Unicode letter`**, **`_`** , **`$`**, or **`0~9`**.
- Are case sensitive and have no maximum length.
- Cannot have the same Unicode character sequence as any keyword, **`boolean`** or **`null`** literal.

1. A Java letter is a character for which `Character.isJavaIdentifierStart(int)` returns `true`.

Keywords and Reserved Words

- Cannot be used as identifiers for classes, methods, variables, and so forth.
- Must be used in the correct order.
- https://docs.oracle.com/en/java/javase/17/
- https://docs.oracle.com/javase/specs/jls/se17/html/index.html

Java Keywords

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

Variables and Constants

- Variables - The values put into variables can be changed at any time.
- Constants - The values put into constants cannot be changed.

Computer Data Storage

- Computers use 0s and 1s(binary) to store data.
- Humans use base 10.
- 0s and 1s are stored in bits(8 bits make a byte).
- A bit can only be on(1) or off(0).
- Values are evaluated by multiplying 2 to a power.

Computer Data Storage (Cont.)

- Java technology uses different data types that store different number of bits.
- Range of values: (-2^{x-1} to $2^{x-1}-1$) where x is the number of bits.
- A data type storing 8 bits would have:
  - 2^8 possible values
  - Highest and lowest values of (-2^7) and (2^7-1)
- Zero is positive.
- Leftmost bit is reserved for the sign: 0 for positive and 1 for negative.

Fundamental Types

- Java technology uses ***data types*** with predefined
  - Storage sizes
  - Kinds of data they can store
- Include the Java primitive types, their corresponding wrapper classes, and reference types.
- Java 5.0 and beyond provide for automatic conversion between these primitive and reference types through ***autoboxing*** and ***unboxing***.

Primitive Types

- Are always the specified precision, regardless of the underlying hardware precisions(e.g., 32- or 64-bit).

Type	Detail	Storage	Range
`boolean`	`true` or `false`	1 bit	Not applicable
`char`	Unicode character	2 bytes	\u0000 to \uFFFF
`byte`	integer	1 byte	-128 to 127
`short`	integer	2 bytes	-32768 to 32767
`int`	integer	4 bytes	-2147483648 to 2147483647
`long`	integer	8 bytes	-2^{63} to $2^{63}-1$
`float`	floating point	4 bytes	$1.4e^{-45}$ to $3.4e^{+38}$
`double`	floating point	8 bytes	$5e^{-324}$ to $1.8e^{+308}$

- **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, and **`double`** are all signed. Type **`char`** is unsigned.

Literals for Primitive Types

- Are source code representation of values.
- All primitive types, except `boolean`, can accept character, decimal, hexadecimal, octal, and Unicode literal formats, as well as character escape sequences.
- Where appropriate, the literal value is automatically cast or converted.
- Remember that bits are lost during truncation.

`boolean` Type & Boolean Literals

- Decision must be true/false, on/off, or yes/no.
- Literals are expressed as either `true` or `false`.
- Print format : `%b`

```
boolean isReady = true;
boolean isSet = new Boolean(false);
```

Sample Code – Boolean Literal

```
System.out.println(true);
boolean boolValue1 = false;
System.out.printf("boolValue = %b\n", boolValue1);
boolean boolValue2 = new Boolean(true);
System.out.println("boolValue2 = " + boolValue2);
//Compile Error. Why?
//boolean boolValue2 = 1;
//System.out.println(boolValue2);
```

`char` Type & Character Literals

- A character literal is either a single character or an ***escape sequence*** contained within single quotes.
- Line terminators are not allowed.
- **`char`** represents each character as a series of 16 bits.
- **`char`** is based on Unicode: 65,535 characters (contains first 128 characters of ASCII).
- See also http://www.unicode.org
- Print format : **`%c`**

```
char charValue1 = 'a';
Character charValue2 = new Character('\'');
```

`char` literals – Escape Sequence

Escape	meant	Unicode
`\n`	New line	\u000a
`\t`	Tab	\u0009
`\b`	Back space	\u0008
`\r`	Carriage return	\u000d
`\f`	Form feed	\u000c
`\\`	Back slash	\u005c
`\'`	Single quote mark	\u0027
`\"`	Double quote mark	\u0022
`\ddd`	Octal number	\0 ~ \377
`\udddd`	Unicode character	\u0000 ~ \uFFFF

Refer to
http://java.sun.com/docs/books/jls/third_edition/html/lexical.html#101089

`char` literals – Escape Sequence (Cont.)

- Different line terminators are used for different platforms to achieve a newline.
- The `println()` method, which includes a line break, is a better solution than hard coding `\n` and `\r`, when used appropriately.

Operating System	Newline
POSIX-compliant operating systems (i.e., Solaris, Linux) and Mac OS X	LF (\n)
Mac OS up to version 9	CR (\r)
Microsoft Windows	CR+LF (\r\n)

Sample Code – Character Literal

```
System.out.printf("%c", 'A');
System.out.printf("%c", 65);
System.out.printf("%c", 'A' + 3);
System.out.printf("%c", '\n');
System.out.printf("%c", 97);
System.out.printf("%c", 'b');
System.out.printf("%c", 'a' + 2);
System.out.printf("%c", '\b');
System.out.printf("%c", 'M');
//Exception Occurrance. Why?
//System.out.printf("%c", 65.5);
System.out.println();
char charValue = '\u0042';
System.out.println("charValue = " + charValue);
```

Integral Type & Literals

- Can be expressed in decimal, hexadecimal, and octal.
- Print format : ***%d***, ***%x***, ***%o***

```
int intValue = 34567;   //decimal
```

- Hexadecimal literals begin with **0x** or **0X**, followed by the ASCII digits **0** through **9** and the letters **a** through **f** (or **A** through **F**).

```
int hexValue = 0X64; //hexadecimal
```

- Octal literals begin with a zero followed by one or more ASCII digits **0** through **7**.

```
int octalValue = 0144; //octal
```

- Binary literals begin with **0b** or **0B**, followed by **0**, and **1**(since JDK 7).

```
int binaryValue = 0b01110111; //binary
```

Integral Type & Literals (Cont.)

- Default is a `int`.
- To define an integer as type `long`, suffix it with an ASCII letter `L` (preferred and more readable) or `l`.

```
long longValue = 500L;
```

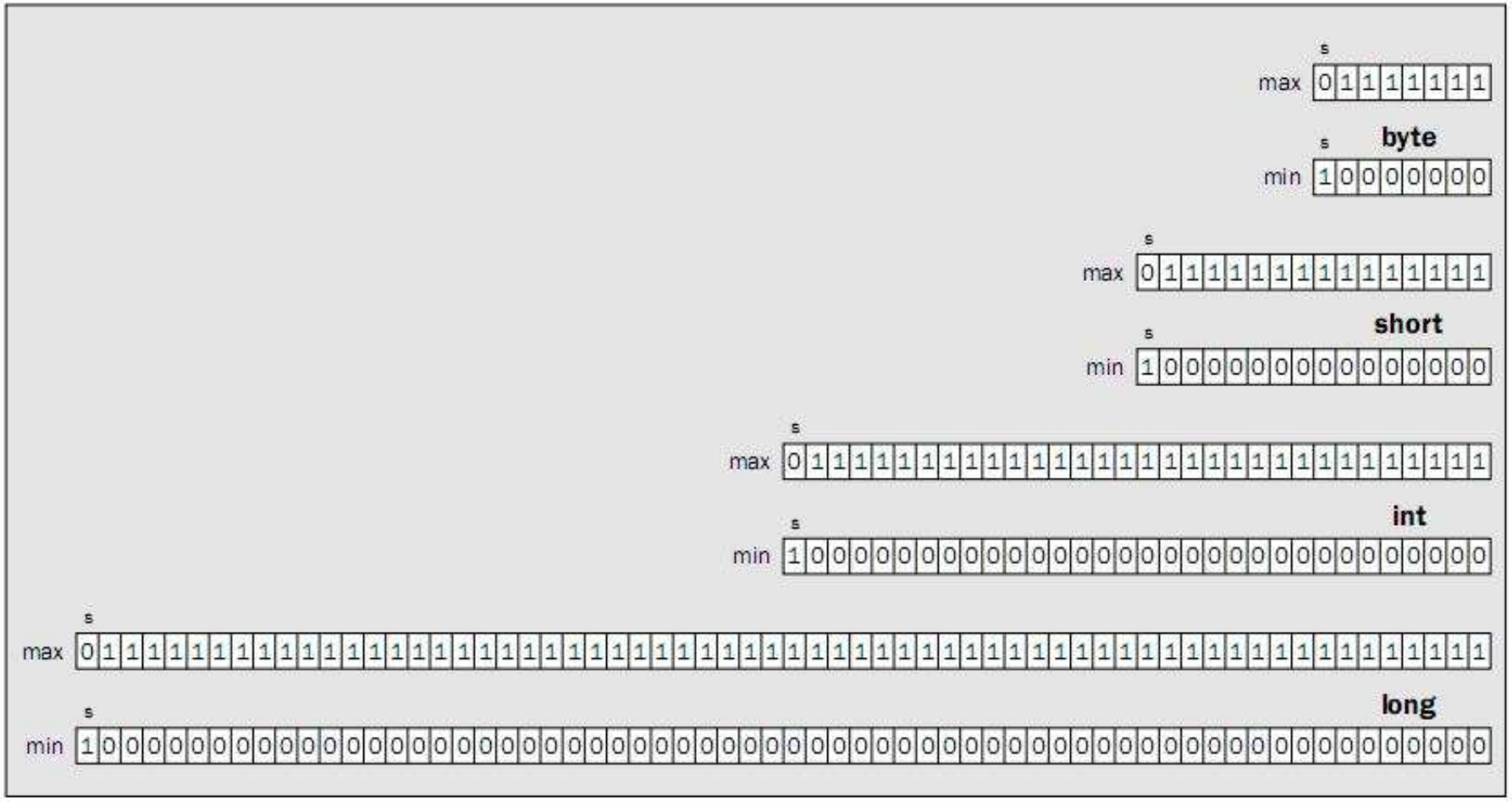
- The `short` primitive has a four byte signed integer as its valid literal. If an explicit cast is not performed, the integer is implicitly cast to two bytes.

```
short seatingCapacity = 17157;
short vipSeats = (short) 500;
```

- Using underscore(_) (since JDK 7).

```
long su = 100_000_00000_00000_00L;
```

Integral Type & Literals (Cont.)



Sample Code – Integral Literal

```
System.out.printf("%d\n", 10);
System.out.printf("%d\n", 10 + 20);
//Compile Error, Why?
//System.out.printf("%d\n", 2147483648);
System.out.printf("%d\n", 13);
System.out.printf("%o\n", 13);
System.out.printf("%x\n", 13);
System.out.printf("%d\n", 015);
System.out.printf("%o\n", 015);
System.out.printf("%x\n", 015);
System.out.printf("%d\n", 0xC);
System.out.printf("%o\n", 0xC);
System.out.printf("%x\n", 0xC);
//Exception Occurrance, Why?
//System.out.printf("%d\n", 12.5);
```

Sample Code – Integral Variable

```
1  //Integral Literal Demo 2nd
2  public class IntLiteralDemo1 {
3      public static void main(String[] args) {
4          byte b1 = 127;
5          System.out.println("b1 =" + b1);
6          short sh = 2500;
7          System.out.println("sh = " + sh);
8          int su = 23L;
9          System.out.println("su = " + su);
10     }
11 }
```

Floating Point Type & Literals

- A valid floating-point literal requires a whole number and/or a fractional part, decimal point, and type suffix.
- An exponent prefaced by an **`e`** or **`E`**, **`P`** or **`p`** is optional.
- Fractional parts and decimals are not required when exponents or type suffixes are applied.
- Highest or lowest values cannot be determined.

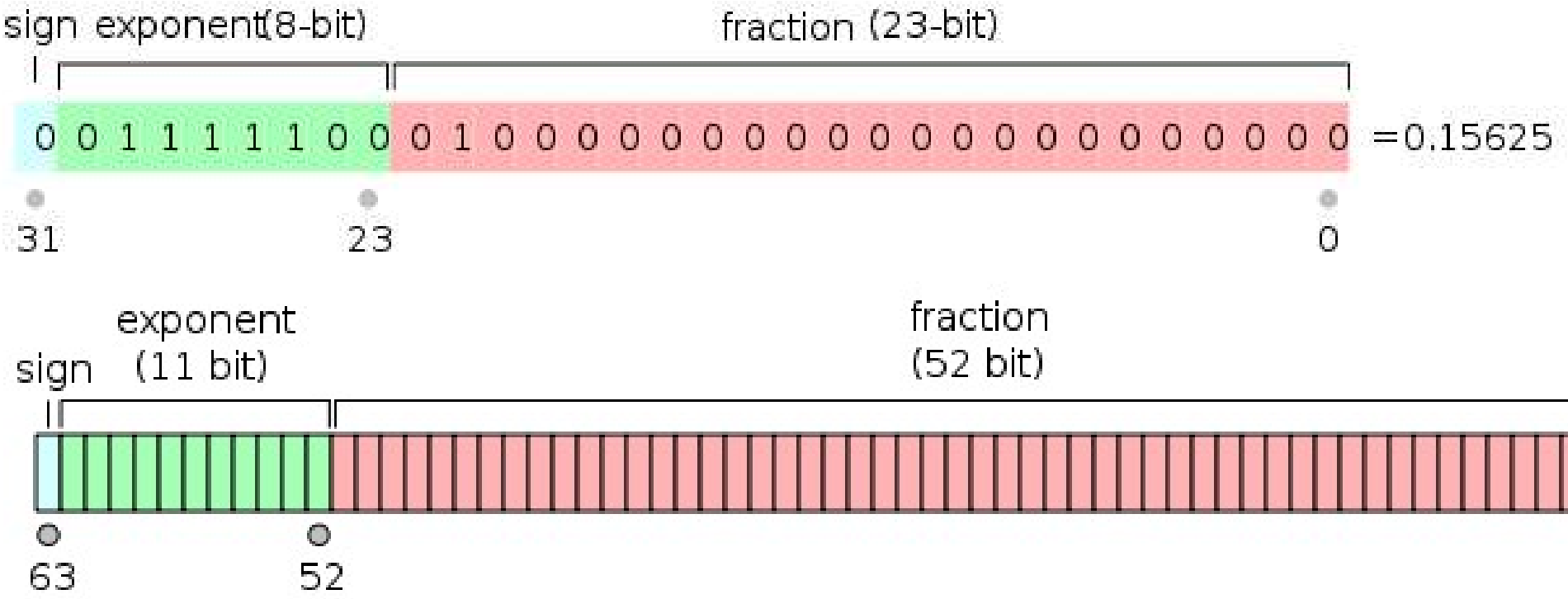
```
[whole-number].[fractional_part][e|E exp][d|D|f|F]
```

Floating Point Type & Literals (Cont.)

- Default is **`double`**.
- Type suffices for doubles are **`d`** or **`D`**; suffices for **`floats`** are **`f`** or **`F`**.
- Print format : **`%f`**, **`%e`**
- No explicit cast is necessary for an **`int`** literal because an **`int`** fits in a **`float`**.

```
float  floatValue = 400;
float  floatValue1 = 9.15f;
double avg = 89.5;
double  doubleValue1 = 3.12d;
double doubleValue2 = 0xA1.27p5;
```

Floating Point Type & Literals – IEEE 754



http://steve.hollasch.net/cgindex/coding/ieeefloat.html
http://en.wikipedia.org/wiki/IEEE_754-1985

Sample Code – Floating Literal

```
System.out.printf("%f\n", 10.5);
System.out.printf("%f\n", 23.1456789);
System.out.printf("%f\n", 34.5E+02);
System.out.printf("%f\n", 1234.567);
System.out.printf("%e\n", 10.5);
System.out.printf("%e\n", 123.456789);
System.out.printf("%e\n", 12.45678e+03);
//Exception Occurrance, Why?
//System.out.printf("%f\n", 'A');
//System.out.printf("%f\n", 10);
System.out.print((-0.0) + Double.NaN);
```

Sample Code – Floating Literal

```
2 public class DoubleDemo {
3     public static void main(String[] args) {
4         double num = 3.14;
5         long num1 = Double.doubleToRawLongBits(num);
6         String str = Long.toBinaryString(num1);
7         System.out.println(str);
8     }
9 }
```

```
1000000000010010001111010111000010100011110101110000101000111111
```

12.375

```
1000000001010001100000000000000000000000000000000000000000000000
```

-12.375

```
1100000000101000110000000000000000000000000000000000000000000000
```

`String` Type & Literals

- Stores words and sentences.
- Is not a primitive data type; it is a *class*.
- Are the only class you can build objects from without using **`new`**.
- Has its literal enclosed in double quotes(" ").

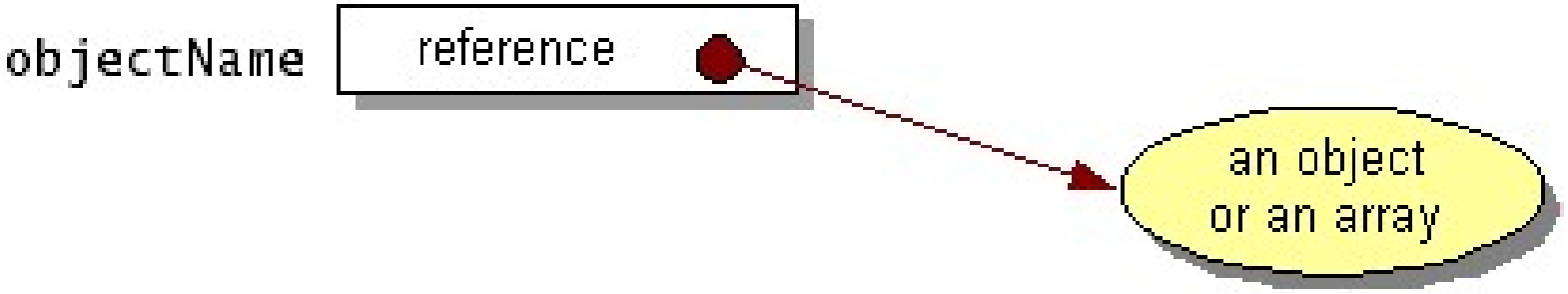
```
String stringValue1 = new String("Valid literal.");
String stringValue2 = "Valid.\nMoving to next line.";
String stringValue3 = "Joins str" + "ings";
String stringValue4 = "\"Escape Sequences\"\r";
```

Sample Code – String Literal

```
1 //String Literal Demo
2 public class StringLiteralDemo {
3     public static void main(String[] args) {
4         System.out.printf("%s\n","Hello,World");
5         System.out.printf("%s\n","Hello,World" + 128);
6     }
7 }
```

Reference Types

- Are used to store addresses of objects.
- Provide a means to access those objects stored somewhere in memory.
- The memory locations are irrelevant to programmers.
- All reference types are a subclass of type `java.lang.Object`.
- Can only store the address of objects of their own type.



Reference Types (Cont.)

- Annotation
  - Provides a way to associate metadata (data about data) with program elements.
- Array
  - Provides a fixed-size data structure that stores data elements of the same type
- Class
  - Models something in the real world and consists of a set of values that holds data and a set of methods that operates on the data.
- Enumeration
  - A reference for a set of objects that represents a related set of choices.
- Interface
  - Provides a public API and is “implemented” by Java classes.

Comparing Reference Types to Primitive Types

Reference Type	Primitive types
Unlimited number of reference types, as they are user-defined.	Consists of `boolean` and numeric types : `char`, `byte`, `short`, `int`, `long`, `float`, and `double`
Memory location stores a reference to the data	Memory location stores actual data
Reference copy	Value copy
Call by reference	Call by value

Sample Code – Reference Type

```java
//Reference Type Demo
public class ReferenceDemo {
    public static void main(String[] args) {
        int su = 5;
        System.out.println("su = " + su);
        int num = su;
        System.out.println("num = " + num);
        su = su * 3;
        System.out.println("su = " + su);
        System.out.println("num = " + num);
    }
}
```

```java
//Reference Type Demo 2nd
public class ReferenceDemo1 {
    public static void main(String[] args) {
        Test t1 = new Test();
        Test t2 = new Test();
        t1.su = 5;
        t2.su = t1.su;
        System.out.println("t1.su = " + t1.su);
        System.out.println("t2.su = " + t2.su);
        ////////////////////////변경후////////////////////////////
        t2 = t1;
        t1.su = t1.su * 3;
        System.out.println("t1.su = " + t1.su);
        System.out.println("t2.su = " + t2.su);
    }
}
class Test{
    public int su;
}
```

Constants

- For values that cannot change once assigned:

```
final double SALES_TAX = 6.25;
```

- Cannot be changed except in original location.
- Use **`final`** keyword to make unchangeable.
- Use all capital letters and underscores for identifier.
- The compiler will give an error message if an attempt is made to change the constant's value.

Naming Conventions – Class names

Naming conventions are used to make Java programs more readable.

It is important to use meaningful and unambiguous names comprised of ASCII letters.

- Should be ***nouns***, as they represent "things" or "objects".
- Should be mixed case with only the first letter of each word capitalized.
- Examples:

```
class ClassOne { }
class MyPetRhinoceros { }
```

Naming Conventions – Interface names

- Should be ***adjectives***.
- Should be end with “able” or “ible” whenever the interface provides a capability.
- Otherwise, should be ***nouns***.
- Follows the same capitalization convention as class names.
- Examples:

```
interface Serializable { }
interface SystemPanel { }
```

Naming Conventions – Enumeration names

- Should be follow the conventions of class names.
- The enumeration set of objects(choices) should be all uppercase letters.
- Examples:

```
enum Battery {CRITICAL, LOW, CHARGED, FULL }
```

Naming Conventions – Method names

- Should contain a verb, as they are used to make an object take action.
- Should be mixed case, beginning with a lowercase letter.
- The first letter of each internal word should be capitalized.
- Adjectives and nouns may be included.
- Always ends with a pair of parentheses.
- Examples:

```
int locate () { }  //verb
String getWayPoint() { } //verb and noun
```

Naming Conventions – Instance and Static Variable names

- Should be ***nouns***.
- Should follow the same capitalization convention as method names.
- Starts with a lowercase letter, no separating character between words.
- Examples:

```
String wayPoint;
int total;
```

Naming Conventions – Parameter and Local Variable names

- Should be descriptive lowercase single words, acronyms, or abbreviations.
- If multiple words are necessary, they should follow the same capitalization convention as method names.
- Temporary variable names may be single letters such as **`i`**, **`j`**, **`k`**, **`m`**, and **`n`** for integers and **`c`**, **`d`**, and **`e`** for characters.
- Examples:

```
public void printHotSpot (String spot) {
  String bestSpot = spot;
  System.out.print("Fish here : " + bestSpot);
}
```

Naming Conventions – Generic Type Parameter names

- Should be uppercase single letters.
- The letter **`T`** for type is typically recommended.
- The Collections Framework makes extensive use of generics.
- **`E`** is used for collection elements.
- **`S`** is used for service loaders.
- **`K`** and **`V`** are used for map keys and values.
- Examples:

```
public interface Map <K, V> {
    V put(K key, V value);
}
```

Naming Conventions – Constant names

- Have uppercase letters for all the words in the phrase.
- Multiples words should be separated by underscores(_).
- Examples:

```
final int CONSTANT_ONE = 27;
final double PI = 3.141592654;
```

Naming Conventions – Package names

- Should be unique and consist of lowercase letters.
- Underscores may be used if necessary.
- Publicly available packages should be the reversed Internet domain name of the organization.
- Beginning with a single-word top-level domain name(i.e. **com**, **net**, **org**, or **kr**), followed by the name of the organization and the project or division.
- Examples:

```
package com.examle.libs;
```