



# The Java Language Rules

**Bok, Jong Soon**  
**[javaexpert@nate.com](mailto: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.)

← → ↺ ↻ unicode.org/charts/ ☆ 📄 🗂️ 🏠 오류 ⋮			
UN Code Charts Tech Site   Site Map   Search			
Unicode 15.1 Character Code Charts			
SCRIPTS   SYMBOLS & PUNCTUATION   NAME INDEX			
Find chart by hex code: <input type="text"/> Go Help Conventions Terms of Use			
Scripts			
European Scripts	African Scripts	South Asian Scripts	Indonesian & Philippine Scripts
Armenian	Adlam	Ahom	Balinese
Armenian Ligatures	Bamum	Bengali and Assamese	Batak
Carian	Bamum Supplement	Bhaiksuki	Buginese
Caucasian Albanian	Bassa Vah	Brahmi	Buhid
Cypriot Syllabary	Coptic	Chakma	Hanunoo
Cypro-Minoan	Coptic in Greek block	Devanagari	Javanese
Cyrillic	Coptic Epact Numbers	Devanagari Extended	Kawi
Cyrillic Supplement	Egyptian Hieroglyphs	Devanagari Extended-A	Makasar
Cyrillic Extended-A	Egyptian Hieroglyph Format Controls	Dives Akuru	Rejang
Cyrillic Extended-B	Ethiopic	Dogra	Sundanese
Cyrillic Extended-C	Ethiopic Supplement	Grantha	Sundanese Supplement
Cyrillic Extended-D	Ethiopic Extended	Gujarati	Tagalog
Elbasan	Ethiopic Extended-A	Gunjala Gondi	Tagbanwa
Georgian	Ethiopic Extended-B	Gurmukhi	East Asian Scripts
Georgian Extended	Medefaidrin	Kaithi	Bopomofo
Georgian Supplement	Mende Kikakui	Kannada	Bopomofo Extended
Glagolitic	Meroitic	Kharoshthi	CJK Unified Ideographs (Han) (38MB)
Glagolitic Supplement	Meroitic Cursive	Khojki	CJK Extension A (7.6MB)
Gothic	Meroitic Hieroglyphs	Khudawadi	CJK Extension B (31MB)
Greek	N'Ko	Lepcha	CJK Extension C
Greek Extended	Osmanya	Limbu	CJK Extension D
Ancient Greek Numbers	Tifinagh	Mahajani	CJK Extension E
Latin	Vai	Malayalam	CJK Extension F
Basic Latin (ASCII)	Middle Eastern Scripts	Masaram Gondi	CJK Extension G
Latin-1 Supplement	Anatolian Hieroglyphs	Meetei Mayek	CJK Extension H
Latin Extended-A	Arabic	Meetei Mayek Extensions	CJK Extension I
Latin Extended-B	Arabic Supplement	Modi	(see also Unihan Database)
Latin Extended-C	Arabic Extended-A	Mro	CJK Compatibility Ideographs
Latin Extended-D	Arabic Extended-B	Multani	CJK Compatibility Ideographs Supplement
Latin Extended-E	Arabic Extended-C	Nag Mundari	CJK Radicals / Kangxi Radicals
Latin Extended-F	Arabic Presentation Forms-A	Nandinagari	CJK Radicals Supplement
Latin Extended-G	Arabic Presentation Forms-B	Newa	CJK Strokes
Latin Extended Additional	Aramaic, Imperial	Ol Chiki	Ideographic Description Characters
Latin Ligatures	Avestan	Oriya (Odia)	Hangul Jamo
Fullwidth Latin Letters	Chorasmian	Saurashtra	Hangul Jamo Extended-A
IPA Extensions	Cuneiform	Sharada	Hangul Jamo Extended-B
Phonetic Extensions	Cuneiform Numbers and Punctuation	Siddham	Hangul Compatibility Jamo

# 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 (Cont.)

## 라인 주석

```
public class FirstSample {  
    public static void main(String[] args) {  
        String message = "Welcome to Java World!"; // 메시지  
        System.out.println(message);  
    }  
}
```

## 블록 주석

```
public class FirstSample {  
    public static void main(String[] args) {  
        /*  
        String message = "Welcome to Java World!";  
        System.out.println(message);  
        */  
        System.out.println("olleh~ 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
<a href="#">PREV CLASS</a>	<a href="#">NEXT CLASS</a>			<a href="#">FRAMES</a>	<a href="#">NO FRAMES</a>
<a href="#">SUMMARY: NESTED   FIELD   CONSTR   METHOD</a>			<a href="#">DETAIL: FIELD   CONSTR   METHOD</a>		
<h2>Class JavaDocDemo</h2>					
java.lang.Object └─ JavaDocDemo					
public class <b>JavaDocDemo</b> extends java.lang.Object					
JavaDoc Demo Class					
<b>Version:</b> 1.0					
<b>Author:</b> JavaMan					
<h3>Constructor Summary</h3>					
<a href="#">JavaDocDemo()</a>					

```
$ 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<sup>1</sup>.
- 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.

<sup>1</sup>. 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 | <i>goto</i> | 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         |
| <i>const</i> | 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.)

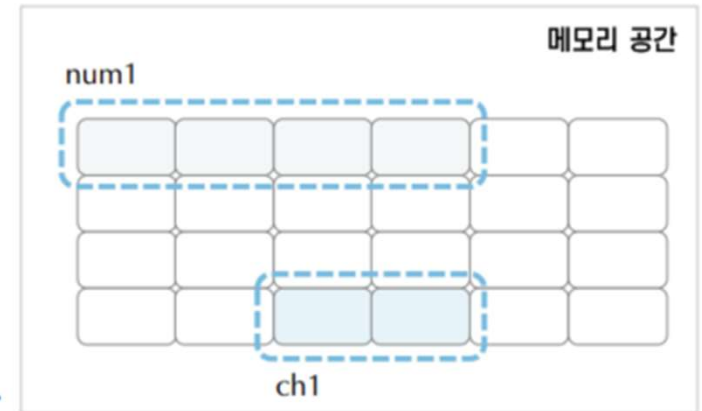
- Java technology uses different data types that store different number of bits.
- Range of values:  $(-2^{x-1} \text{ to } 2^{x-1}-1)$  where  $x$  is the number of bits.
- A data type storing 8 bits would have:
  - $2^8$  possible values
  - Highest and lowest values of  $(-2^7)$  and  $(2^7-1)$
-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

# Variables and Constants (Cont.)

✓ 변수란 데이터를 담는 그릇이며, 데이터의 저장과 참조를 위해 메모리 공간을 할당 받습니다.

- 변수는 자료형과 변수의 이름을 사용하여 선언합니다.

```
int num1;  
char ch1;
```



✓ 변수 이름

- 첫 글자는 문자만 올 수 있으며, 이어서 문자 또는 숫자가 올 수 있습니다.
- 문자는 대소문자를 구분하며, 유니코드로 정의된 어떠한 문자도 사용할 수 있습니다.
- 특수문자는 \$와 \_만 쓸 수 있습니다.  
단, \$는 컴파일러가 생성하므로 사용하지 않습니다.
- 자바의 예약어는 사용할 수 없습니다.  
예: boolean, class, switch, return 등

```
int num1;  
num1 = 10;  
System.out.println(num1);
```

```
int num1 = 10;  
System.out.println(num1);
```

```
int num1;  
System.out.println(num1); //Compile Error
```

✓ 변수 초기화

- 선언한 변수에 처음으로 값을 할당하는 것을 변수 초기화라고 합니다.
- 선언과 동시에 변수를 초기화할 수 있습니다.
- 초기화하지 않은 변수를 사용하면 자바 컴파일러는 에러를 발생시킵니다.

# Variables and Constants (Cont.)

✓ 변수는 사용 위치에 따라 4가지 유형으로 구분합니다.

- 지역변수(Local variables)
- 매개변수(Parameter variables)
- 인스턴스 변수(Instance variables)
- 클래스 변수(Class variables)

```
public class VariableTypes {  
  
    /** 클래스 변수 */  
    public static int classVar = 1;  
  
    /** 인스턴스 변수 */  
    private int instanceVar;  
  
    /**  
     * @param paramVar 매개변수  
     */  
    public static void main(String[] paramVar) {  
  
        // 지역변수  
        int localVal = 10;  
    }  
}
```

## Variables and Constants (Cont.)

- 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.



# Variables and Constants (Cont.)

✓ 변수 중에서 그 값을 바꿀 수 없는 변수를 상수라고 합니다.

- final 키워드를 사용하여 상수를 표시합니다.
- final로 선언된 변수는 값이 한번 할당되면 변경할 수 없습니다.

```
final double PI = 3.14;  
System.out.println("PI is " + PI);
```

✓ 클래스 상수를 선언하면 여러 메소드에서 사용할 수 있습니다.

- 클래스 상수는 static final 키워드로 선언합니다.

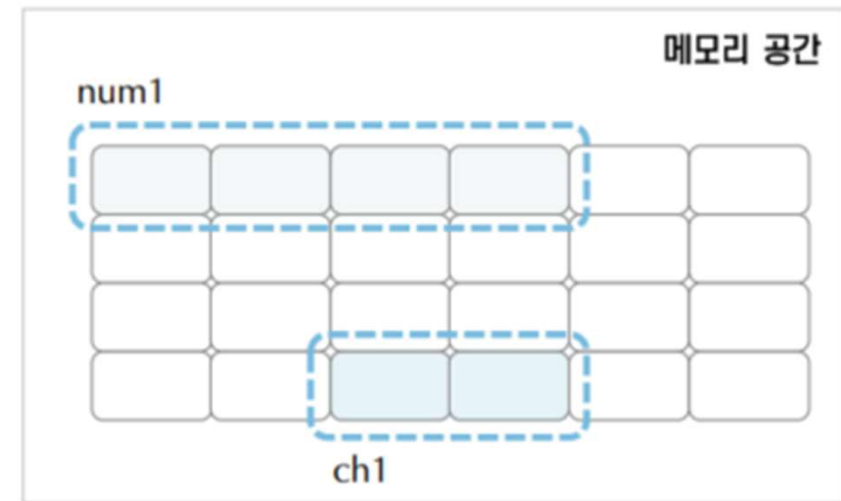
```
public class Variable {  
    public static final double PI = 3.14;  
  
    public static void main(String[] args) {  
        System.out.println("PI is " + PI);  
    }  
}
```

*Variable.java*

# 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*.

```
int num1;  
char ch1;
```



# Primitive Types

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

| Type           | Detail                      | Storage | Range                        |
|----------------|-----------------------------|---------|------------------------------|
| <b>boolean</b> | <b>true</b> or <b>false</b> | 1 bit   | Not applicable               |
| <b>char</b>    | Unicode character           | 2 bytes | \u0000 to \uFFFF             |
| <b>byte</b>    | integer                     | 1 byte  | -128 to 127                  |
| <b>short</b>   | integer                     | 2 bytes | -32768 to 32767              |
| <b>int</b>     | integer                     | 4 bytes | -2147483648 to 2147483647    |
| <b>long</b>    | integer                     | 8 bytes | $-2^{63}$ to $2^{63} - 1$    |
| <b>float</b>   | floating point              | 4 bytes | $1.4e^{-45}$ to $3.4e^{+38}$ |
| <b>double</b>  | 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 Type & Character Literals (Cont.)

- ✓ 문자형은 개별 문자를 나타내는 자료형입니다.
- ✓ Java는 유니코드라는 표준을 사용하며 2바이트로 문자를 표현합니다. ([www.unicode.org](http://www.unicode.org) 참고)
- ✓ 유니코드의 문자체계는 2바이트(16비트)이므로 유니코드 표현법과 16진수 표기법은 같습니다.

J 한 A さ

```
char ch1 = 'J';

char han = '한';
System.out.println(han);

char english = 0x0041; // A
System.out.println(english);

char hangul = 0xAC00; // 가
System.out.println(hangul);

char japanese = 0x3055; // さ
System.out.println(japanese);
```

CharSample.java

|   | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 |
|---|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | NAI | DAI | SP  | 0   | @   | P   | `   | p   |
| 1 | BOH | DC1 | !   | 1   | A   | Q   | a   | q   |
| 2 | STX | DC2 | "   | 2   | B   | R   | b   | r   |
| 3 | ETX | DC3 | #   | 3   | C   | S   | c   | s   |
| 4 | EOF | DC4 | \$  | 4   | D   | T   | d   | t   |
| 5 | END | NAK | %   | 5   | E   | U   | e   | u   |

|   | AC0 | AC1 | AC2 | AC3 | AC4 | AC5 | AC6 | AC7 |
|---|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 가   | 감   | 겐   | 갸   | 갈   | 각   | 갬   | 거   |
| 1 | 각   | 갸   | 갬   | 갯   | 갼   | 갽   | 갾   | 거   |
| 2 | 갾   | 갼   | 갽   | 갻   | 갼   | 갽   | 갾   | 거   |
| 3 | 갾   | 갼   | 갽   | 갻   | 갼   | 갽   | 갾   | 거   |
| 4 | 간   | 갸   | 갬   | 갯   | 갼   | 갽   | 갾   | 거   |

|   | 304 | 305 | 306 | 307 | 308 | 309 |
|---|-----|-----|-----|-----|-----|-----|
| 0 | ぐ   | だ   | ば   | む   | る   |     |
| 1 | あ   | け   | ち   | ば   | め   | ゑ   |
| 2 | あ   | げ   | ぢ   | ひ   | も   | を   |
| 3 | い   | こ   | っ   | び   | ゃ   | ん   |

## char literals – Escape Sequence

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

Refer to

[http://java.sun.com/docs/books/jls/third\\_edition/html/lexical.html#101089](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 Occurrence. 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`.
- 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;
```

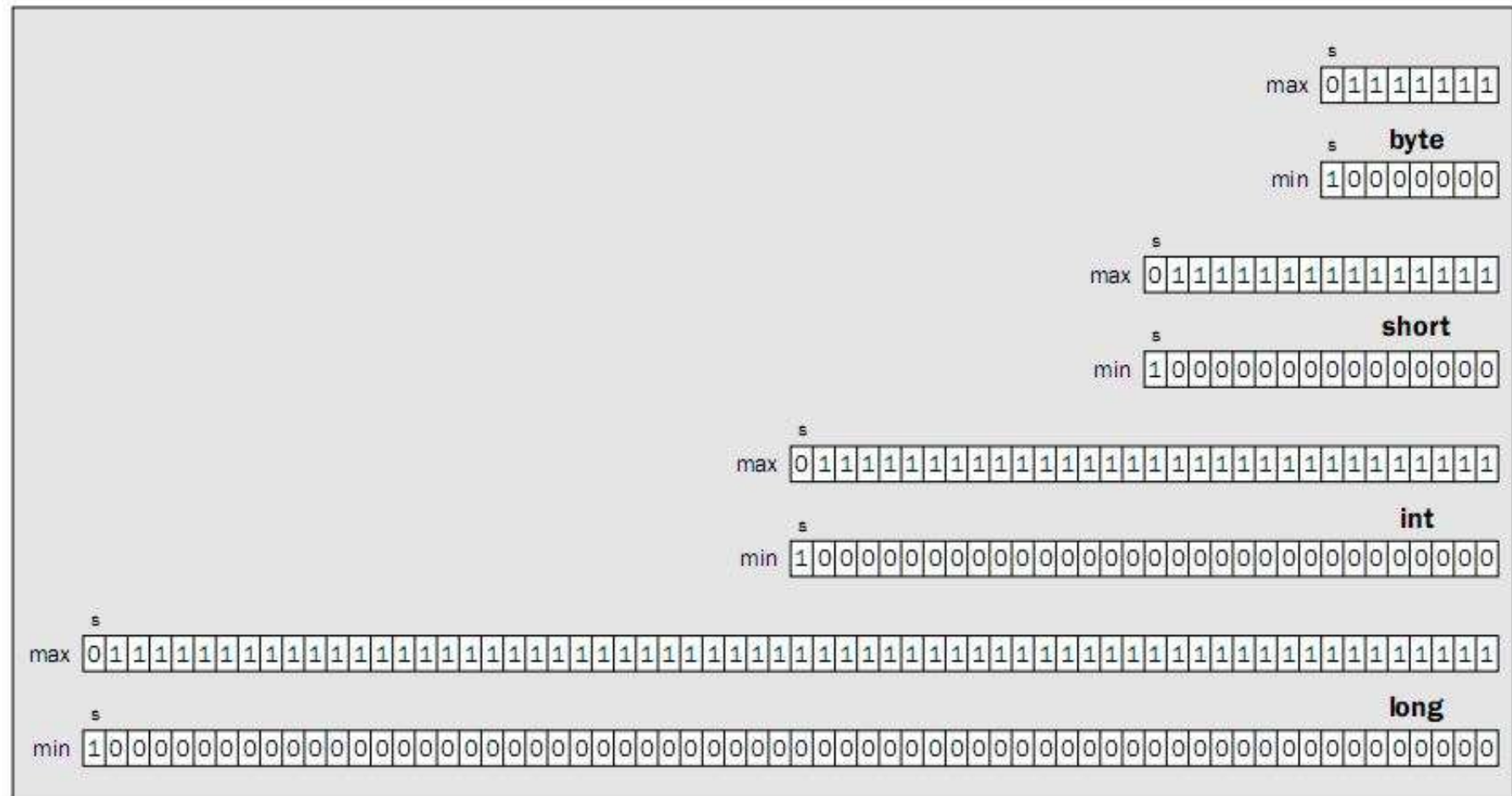
```
short vipSeats = (short) 500;
```

- 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 Occurrence, 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;
```

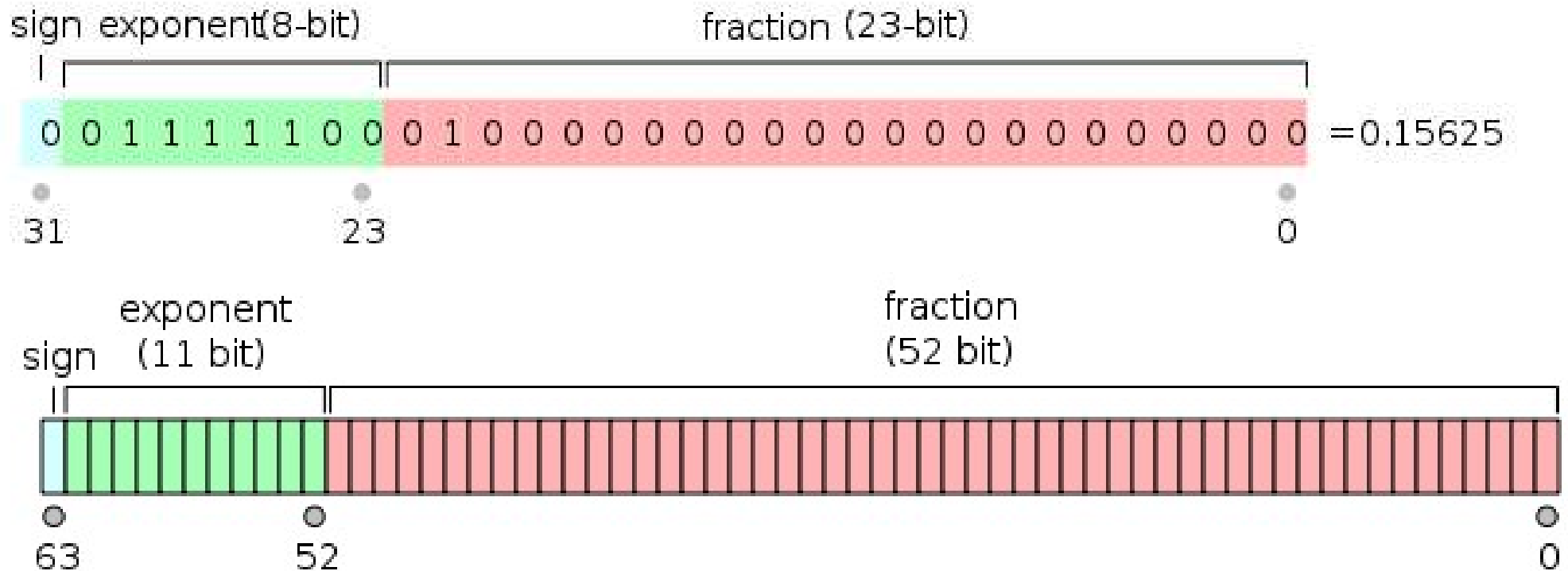
```
float floatValue1 = 9.15f;
```

```
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](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 Occurrence, 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 }
```

100000000001001000111101011100001010001111010111000010100011111

**12.375**

```
10000000010100011000
```

**-12.375**

```
11000000001010001100
```



## 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(" ").

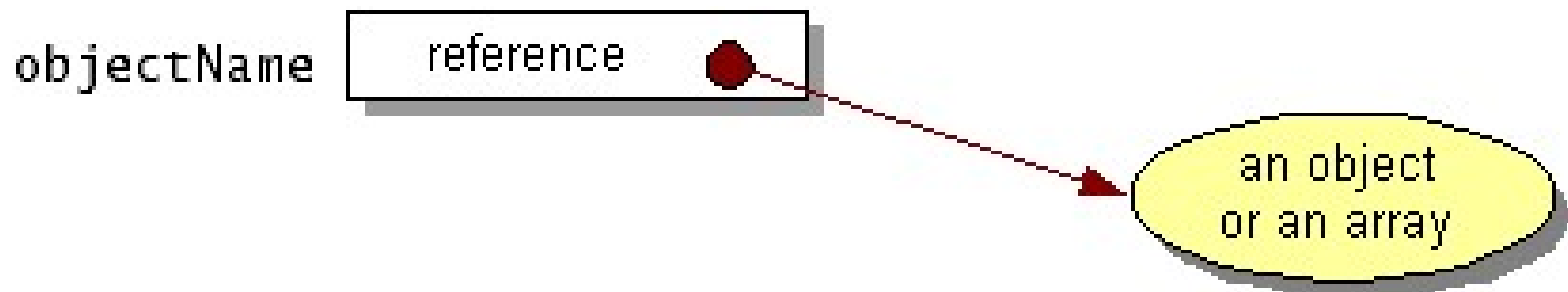
```
String stringValue1 = new String("Valid literal.");  
String stringValue2 = "Valid.\nMoving to next line.";   
String stringValue3 = "Joins str" + "ings";  
String stringValue4 = "\"Escape Sequences\"\\r";
```

## 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.)

✓ JVM이 관리하는 메모리 공간은 크게 3가지 영역으로 나눌 수 있습니다

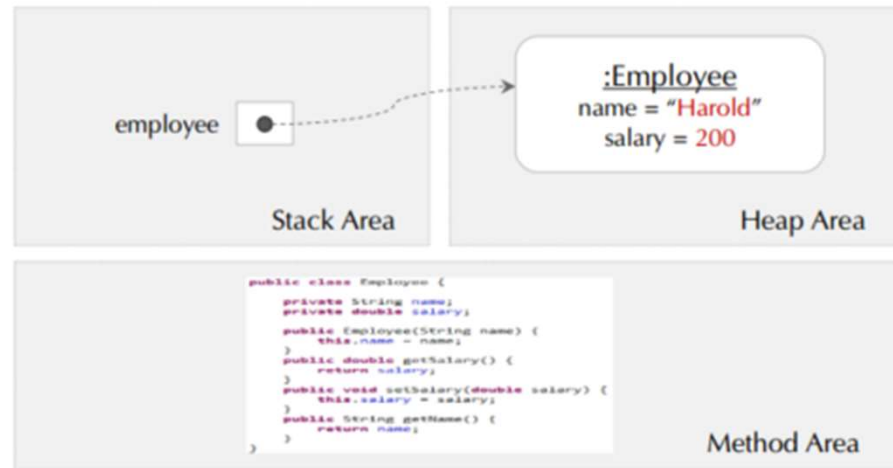
- 메소드 영역(Method Area) : 메소드의 바이트코드, static 변수
- 스택 영역(Stack Area) : 바이트 코드를 읽어 이곳에 지역변수, 매개변수 등을 할당합니다. 자동으로 초기화되지 않습니다.
- 힙 영역(Heap Area) : 배열 공간이나 클래스에 포함된 멤버를 할당하는 영역으로, Garbage Collection 대상이 됩니다.



JVM 메모리 모델

“모든 객체는 힙(Heap)영역에 저장됩니다.”

“GC는 자동으로 참조되지 않는 객체를 메모리에서 제거합니다.”



# 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 <b>boolean</b> and numeric types : <b>char</b> , <b>byte</b> , <b>short</b> , <b>int</b> , <b>long</b> , <b>float</b> , and <b>double</b> |
| 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

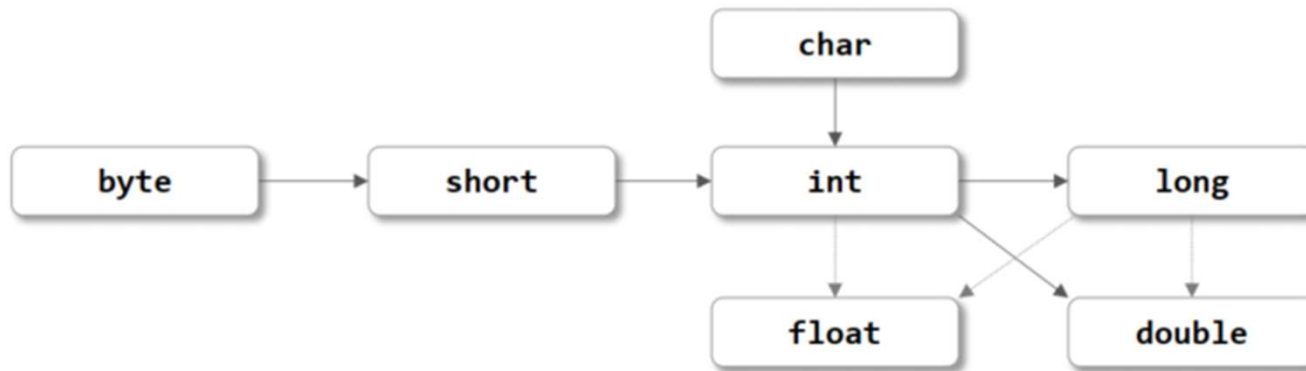
```
1 //Reference Type Demo
2 public class ReferenceDemo {
3     public static void main(String[] args) {
4         int su = 5;
5         System.out.println("su = " + su);
6         int num = su;
7         System.out.println("num = " + num);
8         su = su * 3;
9         System.out.println("su = " + su);
10        System.out.println("num = " + num);
11    }
12 }
```

```
1 //Reference Type Demo 2nd
2 public class ReferenceDemo1 {
3     public static void main(String[] args) {
4         Test t1 = new Test();
5         Test t2 = new Test();
6         t1.su = 5;
7         t2.su = t1.su;
8         System.out.println("t1.su = " + t1.su);
9         System.out.println("t2.su = " + t2.su);
10        //////////////////////////////////변경 후////////////////////////////////////
11        t2 = t1;
12        t1.su = t1.su * 3;
13        System.out.println("t1.su = " + t1.su);
14        System.out.println("t2.su = " + t2.su);
15    }
16 }
17 class Test{
18     public int su;
19 }
```

# Primitive Type Conversion

## ✓ 암묵적 형 변환 (Implicit Conversion)

- Java는 데이터가 손실되지 않거나, 손실이 제한적인 범위 내에서 암묵적으로 형을 변환합니다.
- 자료형이 다른 변수에 값을 할당하는 경우 (float f = 10;)
- 자료형이 다른 값을 연산하는 경우 (double d = 10.0 + 20;)



## ✓ 명시적 형 변환 (Explicit Conversion)

- 명시적으로 형 변환하는 것을 캐스팅(Casting)이라고 합니다.
- 자동 형 변환 규칙에 위배되는 상황임에도 형 변환이 필요한 경우 강제로 자료형을 변환할 수 있습니다. (데이터가 손실될 수 있음)

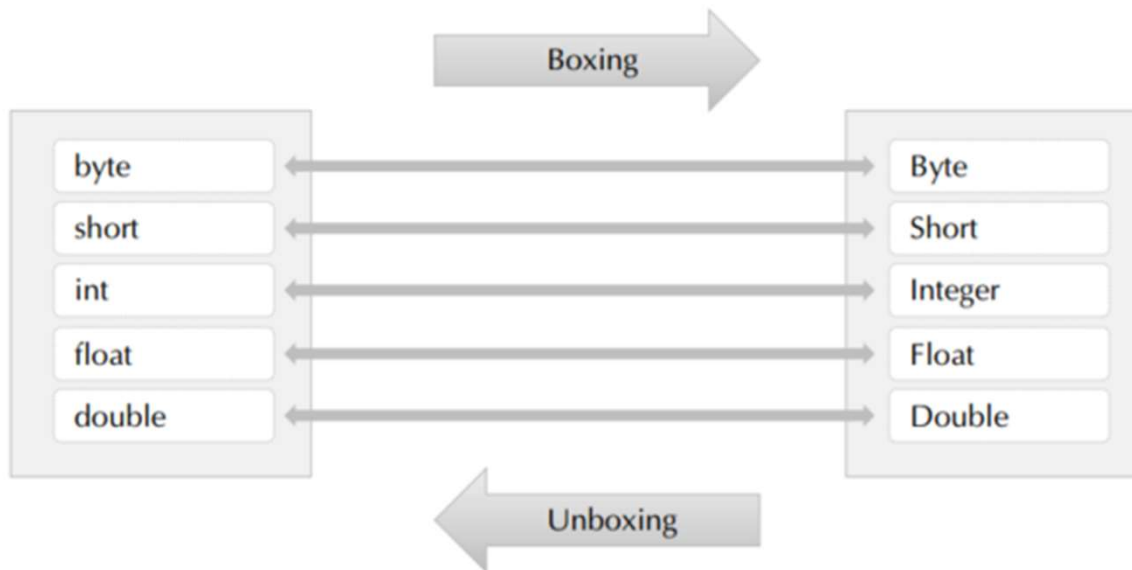
```
int i1 = 20.5; // Error
int i2 = (int) 20.5; // 20
```

*Casting.java*



# Object Wrapper

- ✓ Java는 기본 자료형 값을 객체로 사용할 수 있는 Object Wrapper 클래스를 제공합니다.
- ✓ Object Wrapper 클래스의 종류입니다.
  - Integer, Long, Float, Double, Short, Byte, Character, Boolean
  - Integer, Long, Float, Double, Short, Byte는 상위 클래스로 Number를 상속합니다.
- ✓ Wrapper 객체는 불변(immutable) 객체로 인스턴스에 저장된 값을 바꿀 수 없습니다.
- ✓ 기본 자료형을 Wrapper 객체로 변환하는 것을 Boxing 이라 하고, 그 반대를 Unboxing 이라 합니다.



# 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.
- Multiple 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.example.libs;
```