Data Types, Variable and Operator

Outline of the Presentation

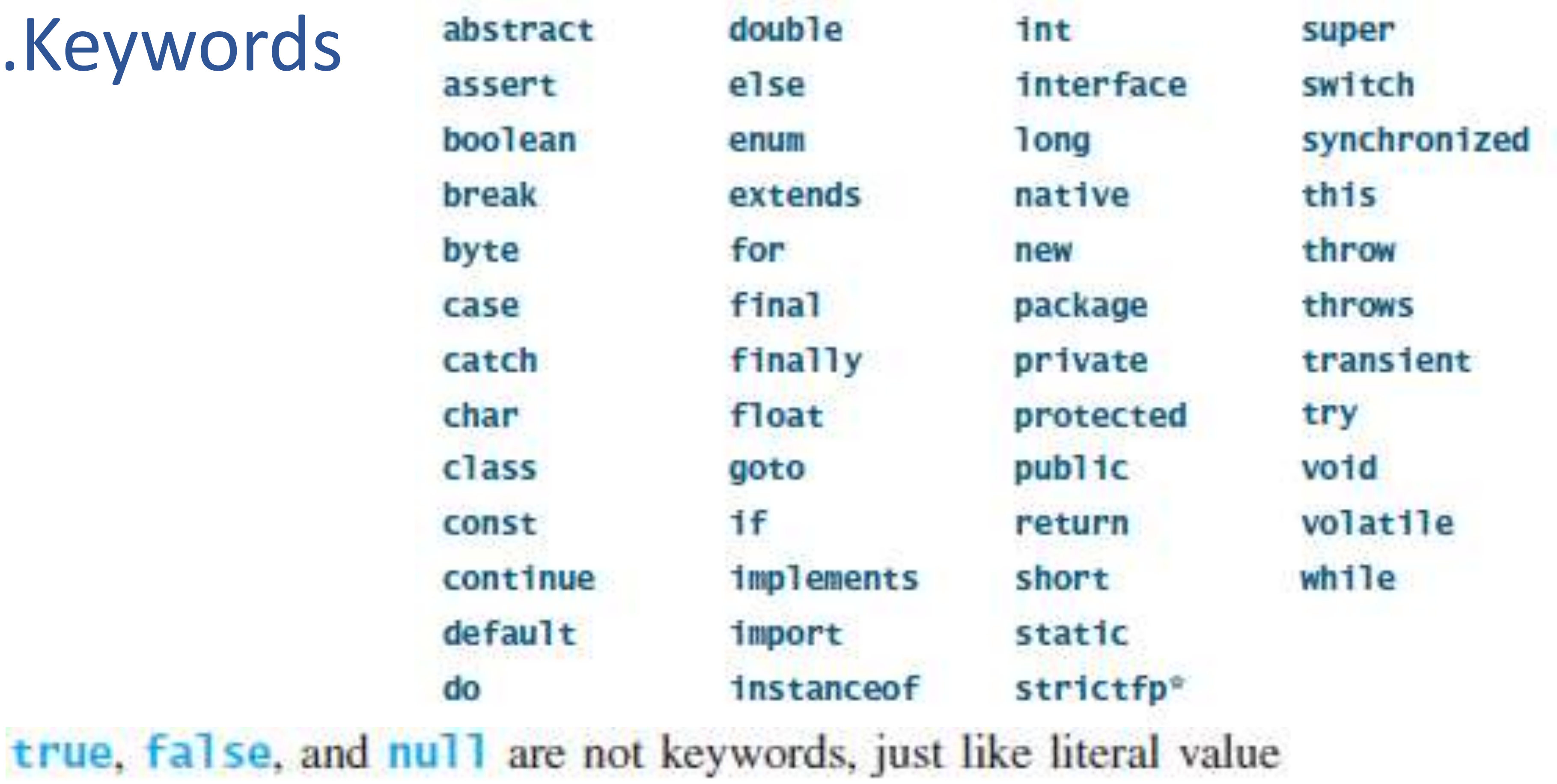
- ✓ Pre requisite (Do it yourself)
- ✓ Identifiers and naming Convention
- ✓ Data types and Variables
- ✓ Type conversion and casting, type promotion

Pre requisite (Do it Yourself)

1.Concepts of OOPs

(definition and examples of abstraction, classes, objects, encapsulation, inheritance and polymorphism)

2.Keywords



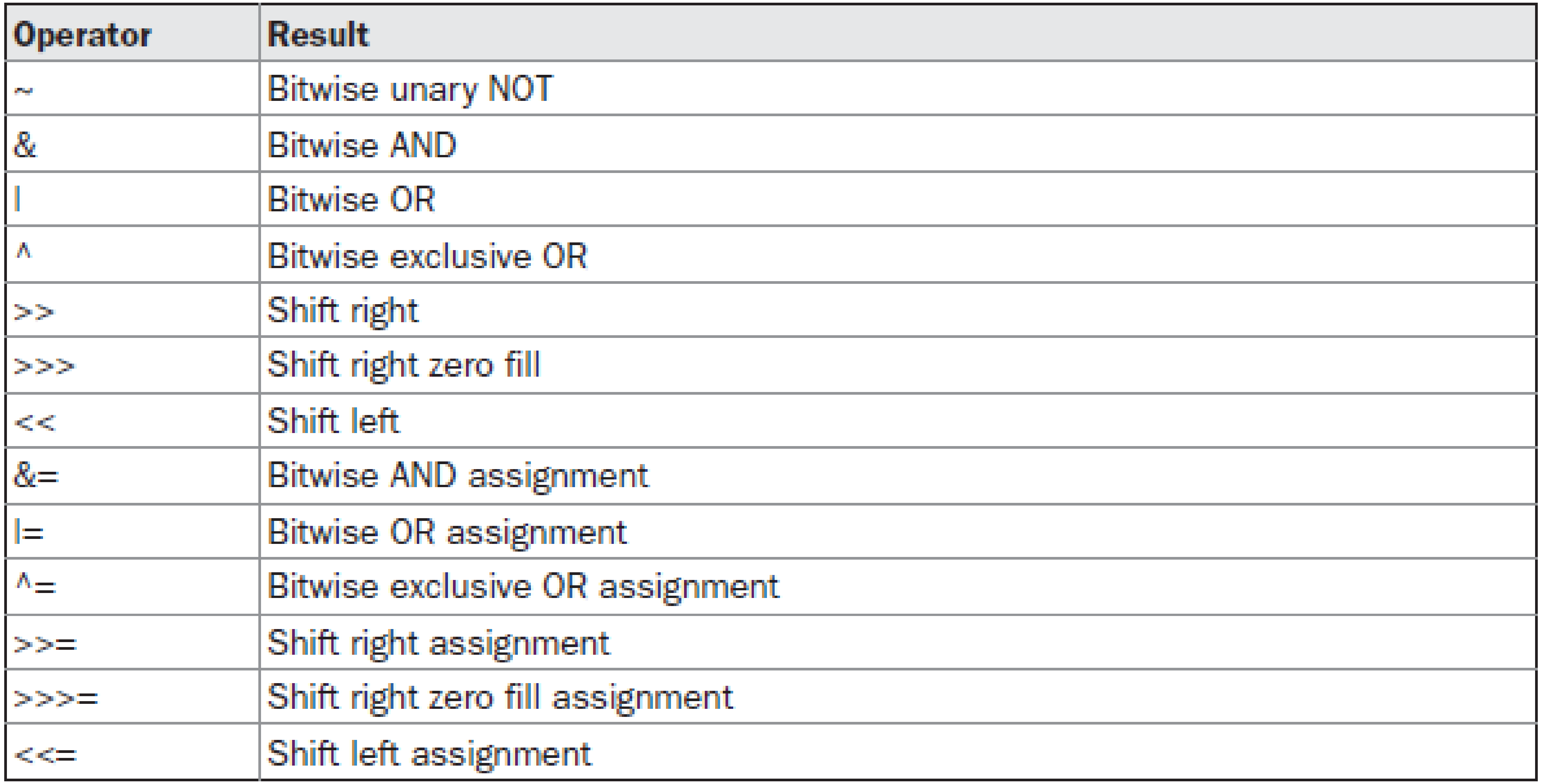
abstract	double	int	super
assert	else	interface	switch
boolean	enum	long	synchronized
break	extends	native	this
byte	for	new	throw
case	final	package	throws
catch	finally	private	transient
char	float	protected	try
class	goto	public	void
const	if	return	volatile
continue	implements	short	while
default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

true, false, and null are not keywords, just like literal value

3. Operators – Arithmetic Operators

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

3. Operators – Bitwise Operators

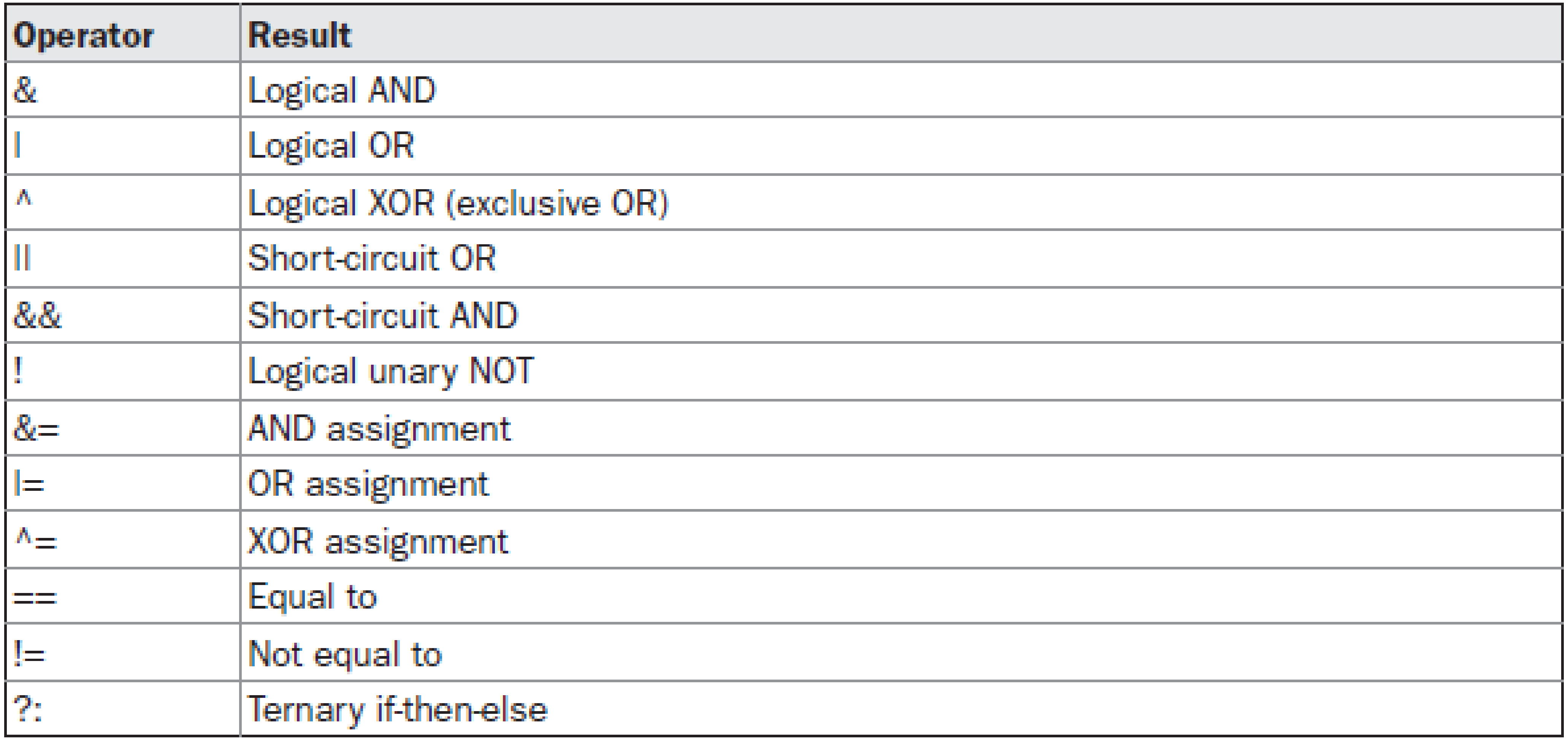


Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

3. Operators – Relational Operators

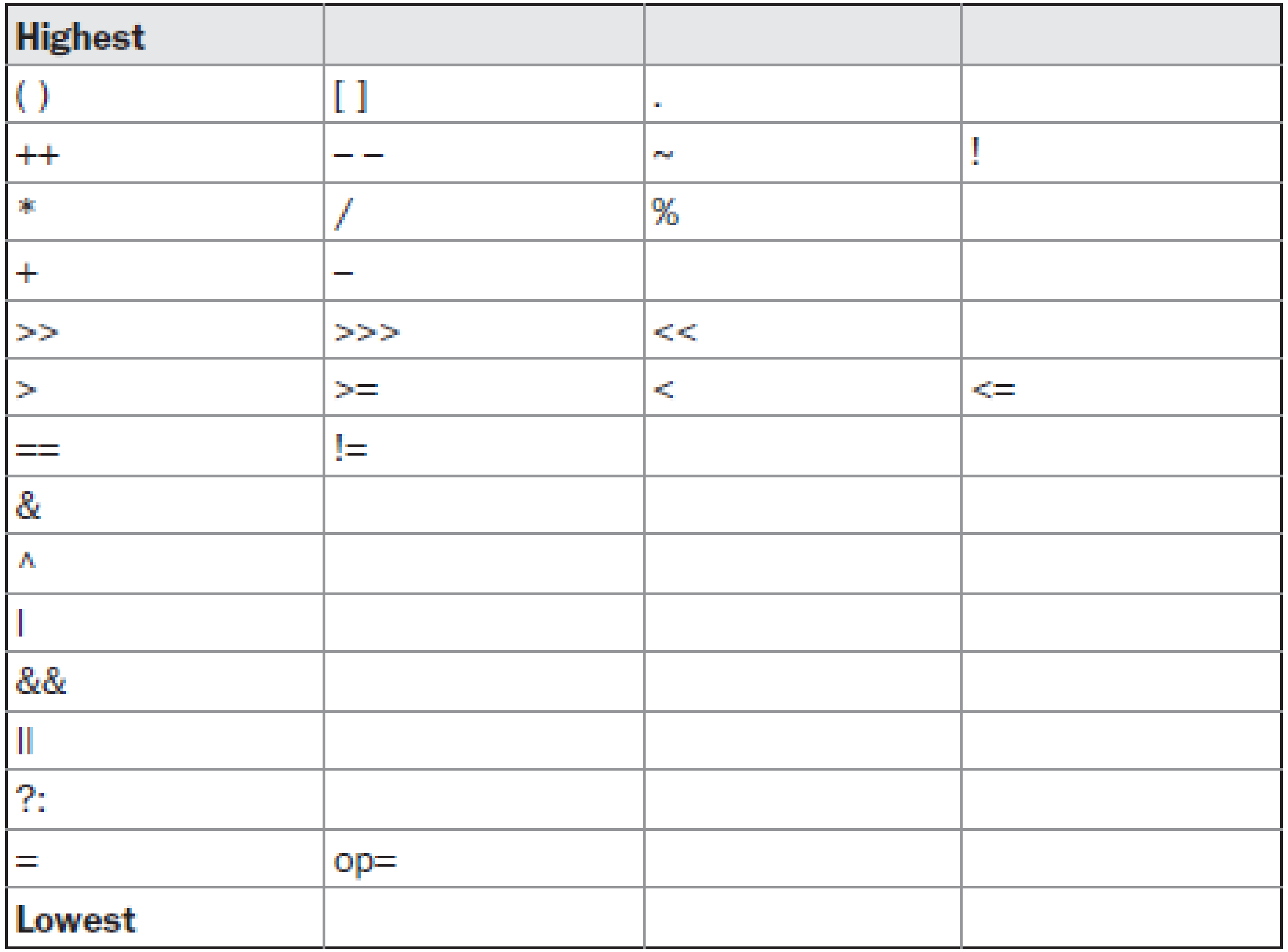
Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

3. Operators – Boolean Logical Operators



Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

3. Precedence of Operators



Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

The Left Shift

- The left shift operator,<<, shifts all of the bits in a value to the left a specified number of times.

  value << num

- Example:
  - 01000001 65
  - << 2
  - 00000100 4

The Right Shift

- The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times.

  value >> num
- Example:
  - 00100011 35
  - >> 2
  - 00001000 8

- When we are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit.

- This is called *sign extension and serves to preserve* the sign of negative numbers when you shift them right.

The Unsigned Right Shift

- In these cases, to shift a zero into the high-order bit no matter what its initial value was. This is known as an unsigned shift.
- To accomplish this, we will use Java's unsigned, shift-right operator, >>>, which always shifts zeros into the high-order bit.

- Example:
  - 11111111 11111111 11111111 11111111 –1 in binary as an int
  - >>>24
  - 00000000 00000000 00000000 11111111 255 in binary as an int

Example

```
int a = 2;
System.out.println(a<<2);       //8
System.out.println(a>>2);       //0

int b = -512;
System.out.println(b>>1);       //-256
System.out.println(b>>24);      //-1
System.out.println(b>>>24);     //255
```

```
/*
2 -->    00000000 00000000 00000000 00000010
 >>2 -->00000000 00000000 00000000 00000000     //after right-shift by 2

512 --> 00000000 00000000 00000010 00000000

        11111111 11111111 11111101 11111111  //1's complement
-512--> 11111111 11111111 11111110 00000000  //2's complement

-512 >> 1 -->  (MSB will be preserved)
        11111111 11111111 11111111 00000000     //it is a -ve no. since MSB is "1"
                                                //so take 2's complement again
        00000000 00000000 00000000 11111111  //1's complement
        00000000 00000000 00000001 00000000  //2's complement

Similarly,
-512--> 11111111 11111111 11111110 00000000

-512 >>24 -->  (MSB will be preserved)
        11111111 11111111 11111111 11111111   //it is a -ve no. since MSB is "1"
                                                //so take 2's complement again

        00000000 00000000 00000000 00000000 //1's complement
        00000000 00000000 00000000 00000001 //2's complement
```

Point to remember

- **When no. of shifts is increased by more than 31 bits, then following formula is used to calculate the answer:**

actual shifts to perform = given no. of shifts – 32

Eg: 2 << 32 → apply above formula → 32 – 32 = 0

Hence, no shift is performed on value 2.

2 << 33 → apply above formula → 33 – 32 = 1

Hence, apply only 1 shift on value 2.

- **If no. of shifts to perform are large (eg: 64), then use above formula where you have to keep subtracting given no. of shifts by 32 until you get a resulting no. < 32 (or perform the mod on given no. of shifts by 32 and use the remainder to perform that many shifts).**

Eg: 2 << 64 → using above formula → 64 – 32 = **32**

→ **32** – 32 = 0 (which is < 32)

Hence, no shift is performed on value 2.

Bitwise Operator Compound Assignments

- combines the assignment with the bitwise operation.
- Similar to the algebraic operators
- For example, the following two statements, which shift the value in a right by four bits, are equivalent:

```
a = a >> 4;
a >>= 4;
```

- Likewise,

```
a = a | b;
a |= b;
```

Java Unary Operator Example: ++ and --

```
class OperatorExample{
public static void main(String args[]){
int x=10;
System.out.println(x++);//10 (11)
System.out.println(++x);//12
System.out.println(x--);//12 (11)
System.out.println(--x);//10
}}
```

Output:

```
10
12
12
10
```

Java Unary Operator Example: ~ and !

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=-10;
boolean c=true;
boolean d=false;
System.out.println(~a);//-11 (minus of total positive value which starts from 0)
System.out.println(~b);//9 (positive of total minus, positive starts from 0)
System.out.println(!c);//false (opposite of boolean value)
System.out.println(!d);//true
}}
```

Output:

```
-11
9
false
true
```

Java Arithmetic Operator Example

```java
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

Output:

```
15
5
50
2
0
```

Java Arithmetic Operator Example: Expression

```
class OperatorExample{
public static void main(String args[]){
System.out.println(10*10/5+3-1*4/2);
}}
```

Output:

```
21
```

Java Shift Operator Example: Left Shift

```
class OperatorExample{
public static void main(String args[]){
System.out.println(10<<2);//10*2^2=10*4=40
System.out.println(10<<3);//10*2^3=10*8=80
System.out.println(20<<2);//20*2^2=20*4=80
System.out.println(15<<4);//15*2^4=15*16=240
}}
```

Output:

```
40
80
80
240
```

Java Shift Operator Example: Right Shift

```
class OperatorExample{
public static void main(String args[]){
System.out.println(10>>2);//10/2^2=10/4=2
System.out.println(20>>2);//20/2^2=20/4=5
System.out.println(20>>3);//20/2^3=20/8=2
}}
```

Output:

```
2
5
2
```

Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check second condition if first condition is false. It checks second condition only if first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a<c);//false && true = false
System.out.println(a<b&a<c);//false & true = false
}}
```

Output:

```
false
false
```

Java AND Operator Example: Logical && vs Bitwise &

```java
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a++<c);//false && true = false
System.out.println(a);//10 because second condition is not checked
System.out.println(a<b&a++<c);//false && true = false
System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
false
10
false
11
```

Java OR Operator Example: Logical || and Bitwise |

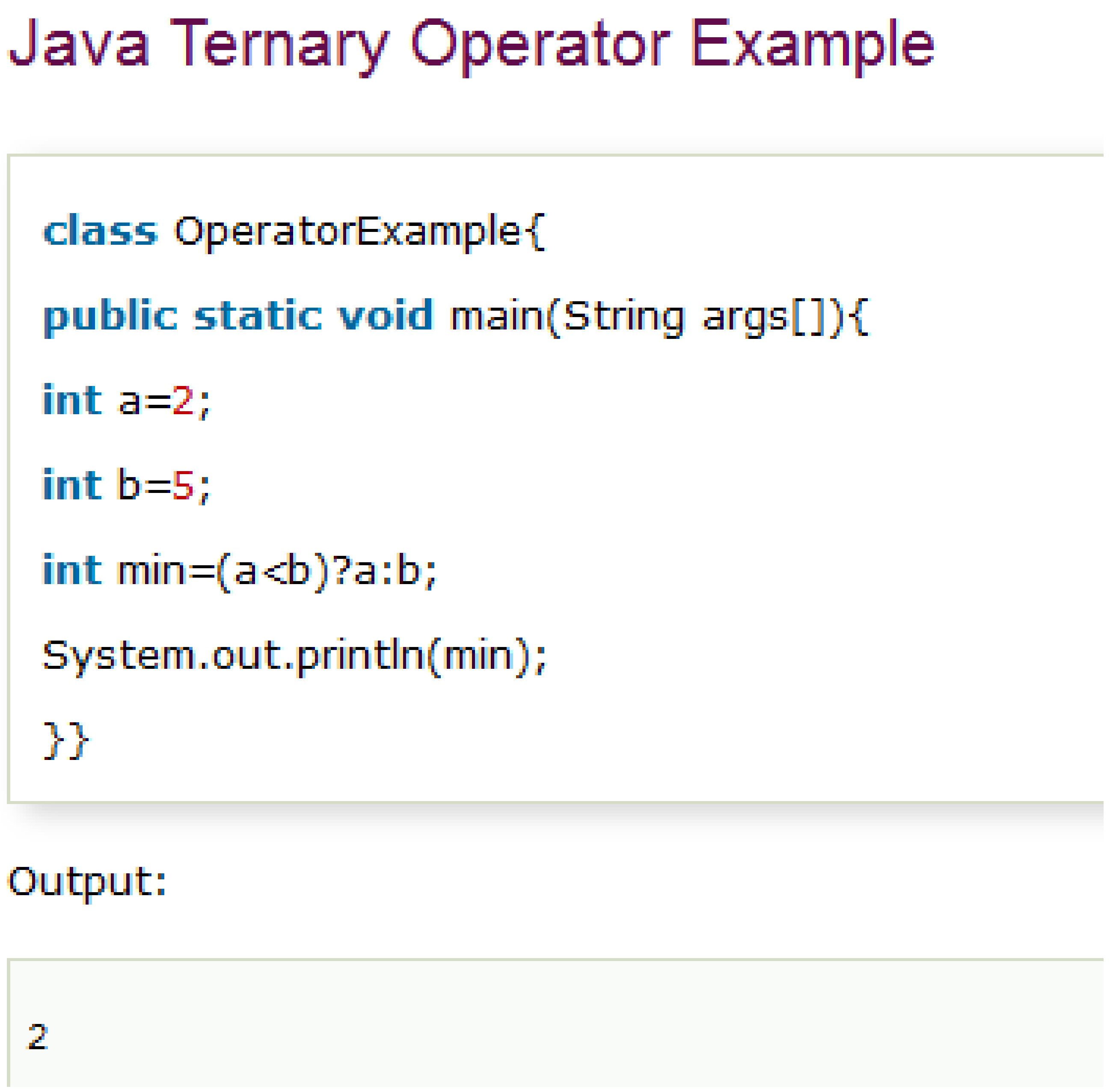
The logical || operator doesn't check second condition if first condition is true. It checks second condition only if first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a>b||a<c);//true || true = true
System.out.println(a>b|a<c);//true | true = true
//|| vs |
System.out.println(a>b||a++<c);//true || true = true
System.out.println(a);//10 because second condition is not checked
System.out.println(a>b|a++<c);//true | true = true
System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
true
true
true
10
true
11
```



Java Ternary Operator Example

```
class OperatorExample{
public static void main(String args[]){
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

```
2
```

Java Assignment Operator Example

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
```

Output:

```
14
16
```

5. Lexical issues

(literals, identifiers, separators, comments)

Identifiers and Naming Conventions

Names of things that appear in the program are called identifiers.

Rules:

- ✓ An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs ($).
- ✓ An identifier must start with a letter, an underscore (_), or a dollar sign ($). It cannot start with a digit.
- ✓ An identifier cannot be a reserved word.
- ✓ An identifier cannot be true, false, or null.
- ✓ An identifier can be of any length.

*Java is case sensitive

*Do not name identifiers with the **$** character. By convention, the **$** character should be used only in mechanically generated source code

*Descriptive identifiers make programs easy to read

Identifiers and Naming Conventions

Names of things that appear in the program are called identifiers.

Rules:

- ✓ An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs ($).
- ✓ An identifier must start with a letter, an underscore (_), or a dollar sign ($). It cannot start with a digit.
- ✓ An identifier cannot be a reserved word.
- ✓ An identifier cannot be true, false, or null.
- ✓ An identifier can be of any length.

Which of the following identifiers are valid?

applet, Applet, a++, —a, 4#R, $4, #44, apps

Identifiers and Naming Conventions

Conventions (not rules)

- ✓ Use lowercase for variables and methods. If a name consists of several words, concatenate them into one, making the first word lowercase and capitalizing the first letter of each subsequent word—for example, the variable **radius** and the method **computeArea()**.
- ✓ Capitalize the first letter of each word in a class name—for example, the class name Circle
- ✓ Capitalize every letter in a constant, and use underscores between words—for example, the constants **PI** and **MAX_VALUE**

* Do not choose class names that are already used in the Java library. For example, since the **Math** class is defined in Java, you should not name your class **Math**.

* Avoid using abbreviations for identifiers. Using complete words is more descriptive. For example, **numberOfStudents** is better than **numStuds**, **numOfStuds**, or **numOfStudents**

Datatype and Variables (do it yourself)

1. Declaring Variables

   datatype variablename;

   datatype variablename 1, variablename 2, …. variablename n;

2. Initializing Variables

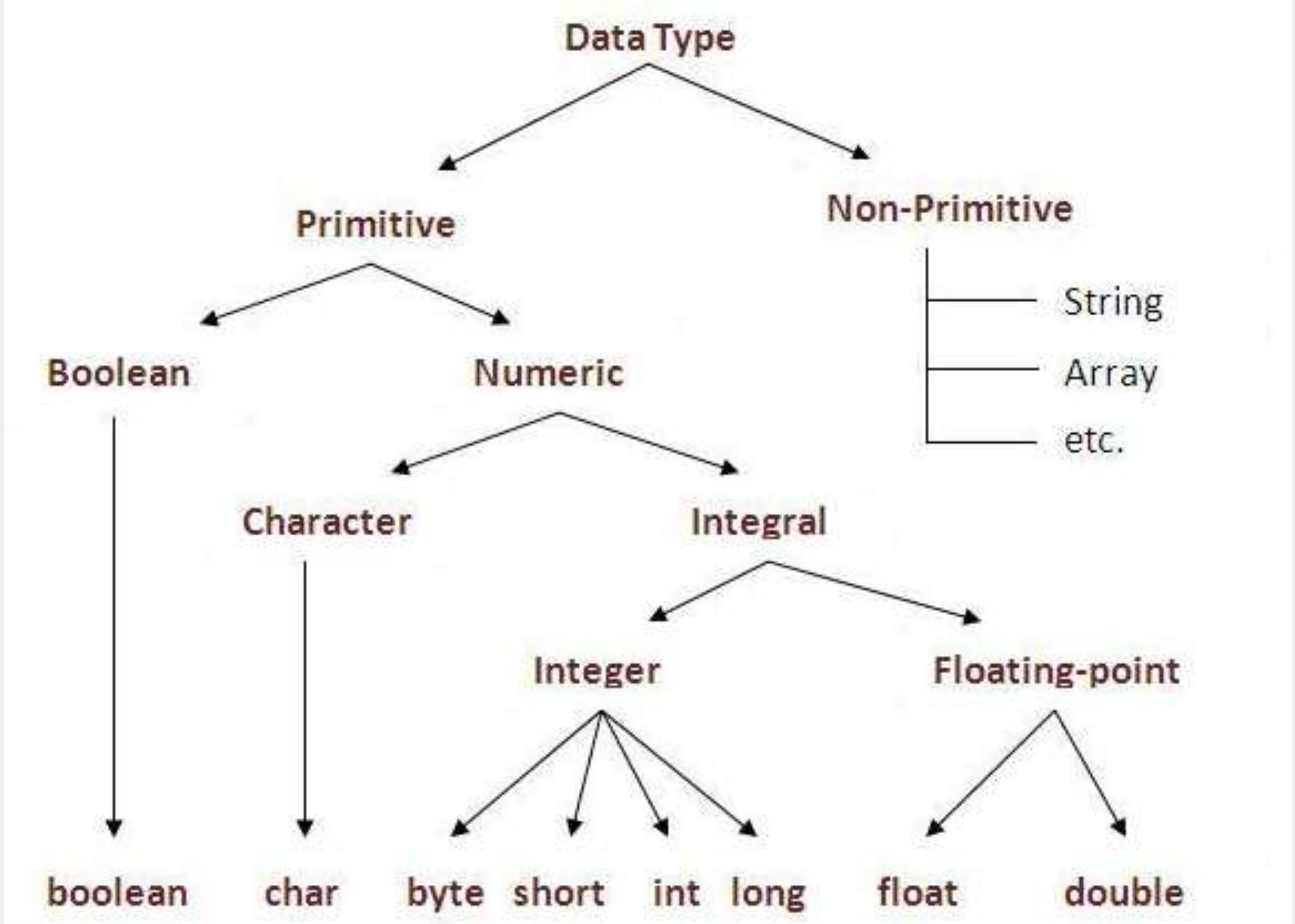
   datatype variablename = value;

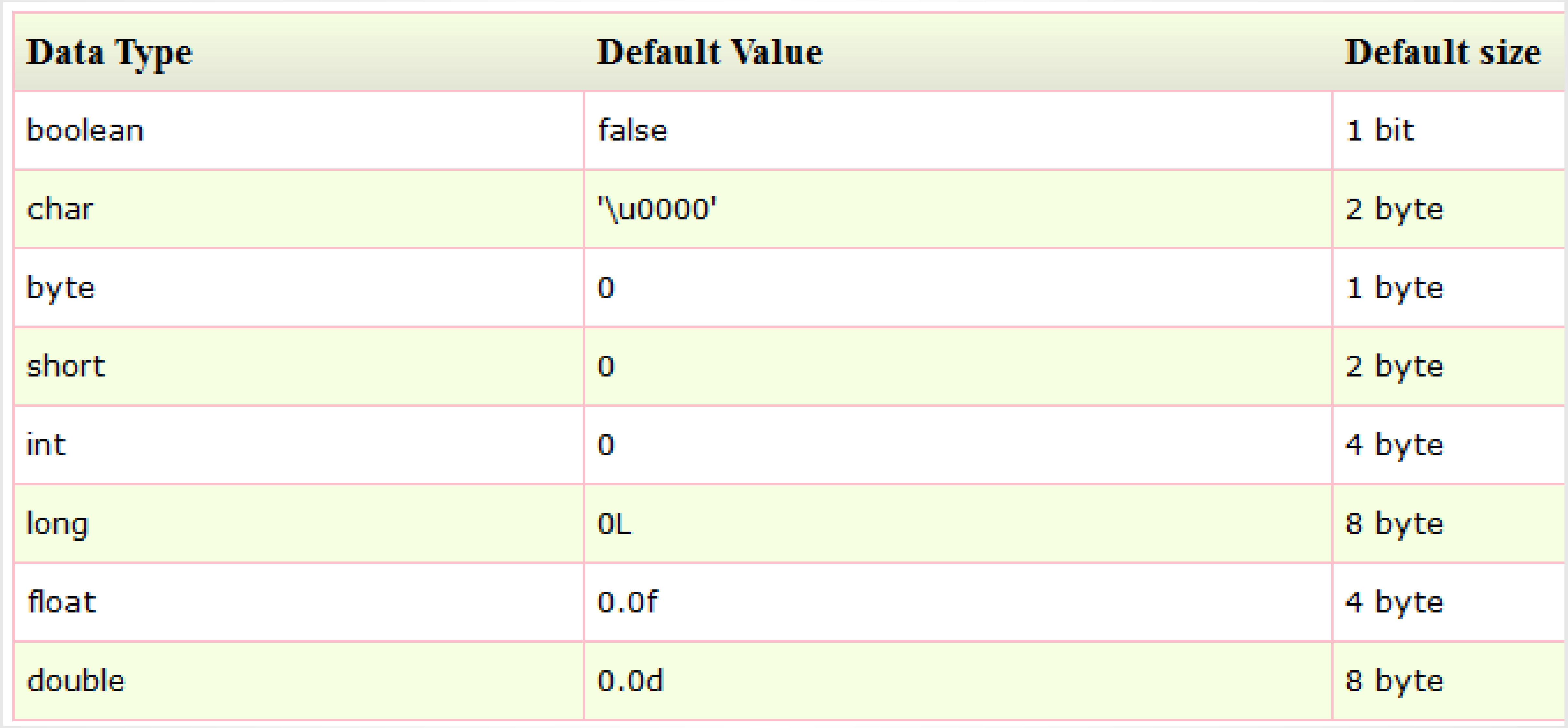
3. Defining Constants

   final datatype CONSTANTNAME = VALUE;

<u>Data types</u>

1. Numeric
2. Character
3. Boolean

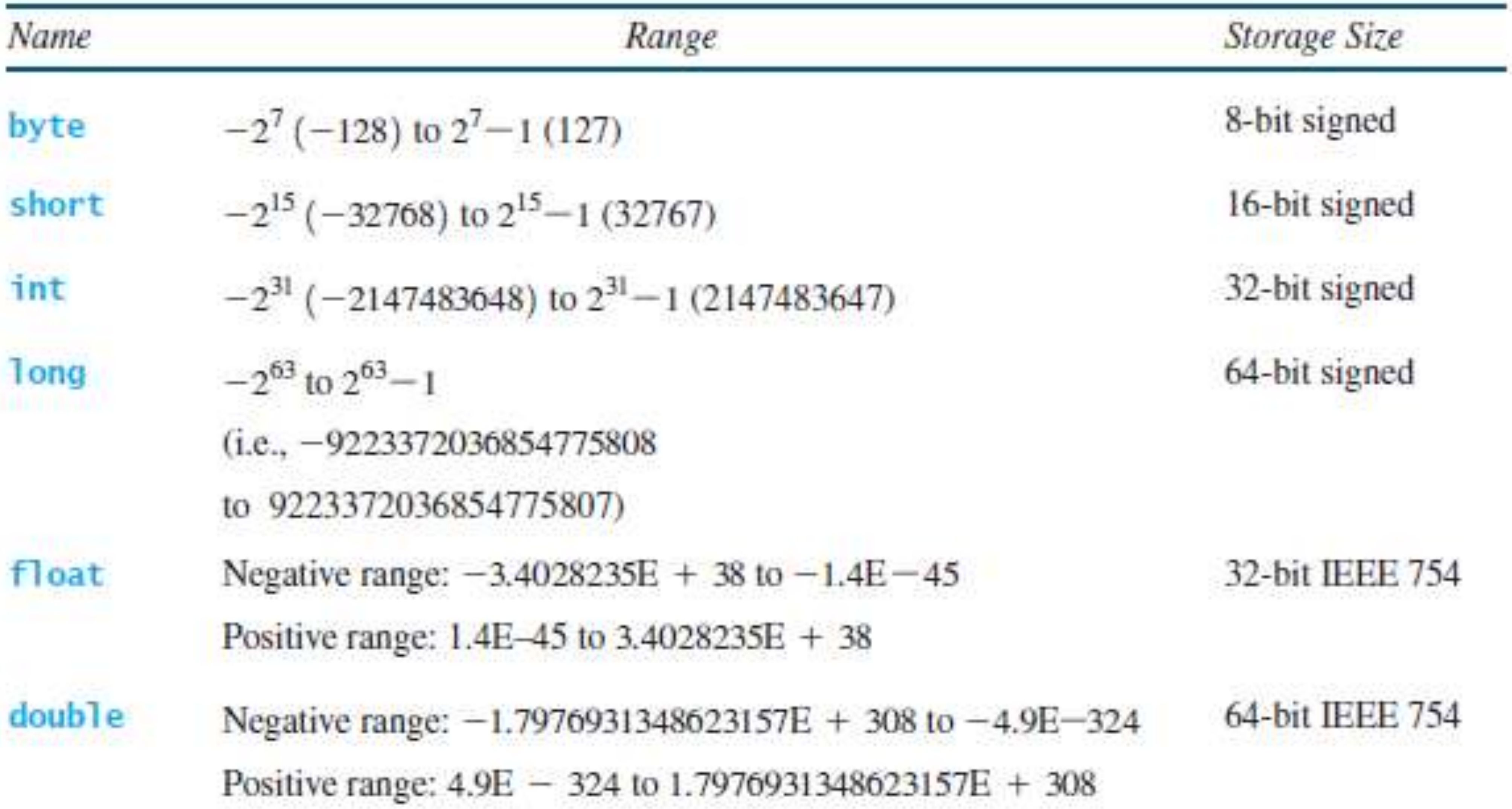

Data Type
Primitive
Non-Primitive
String
Array
etc.
Boolean
Numeric
Character
Integral
Integer
Floating-point
boolean
char
byte
short
int
long
float
double



Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Datatype and Variables (do it yourself)

1. Numeric datatypes



Name	*Range*	*Storage Size*
`byte`	-2^7 (−128) to 2^7-1 (127)	8-bit signed
`short`	-2^{15} (−32768) to $2^{15}-1$ (32767)	16-bit signed
`int`	-2^{31} (−2147483648) to $2^{31}-1$ (2147483647)	32-bit signed
`long`	-2^{63} to $2^{63}-1$ (i.e., −9223372036854775808 to 9223372036854775807)	64-bit signed
`float`	Negative range: −3.4028235E + 38 to −1.4E−45 Positive range: 1.4E–45 to 3.4028235E + 38	32-bit IEEE 754
`double`	Negative range: −1.7976931348623157E + 308 to −4.9E−324 Positive range: 4.9E − 324 to 1.7976931348623157E + 308	64-bit IEEE 754

Datatype and Variables (do it yourself)

2. Character datatype

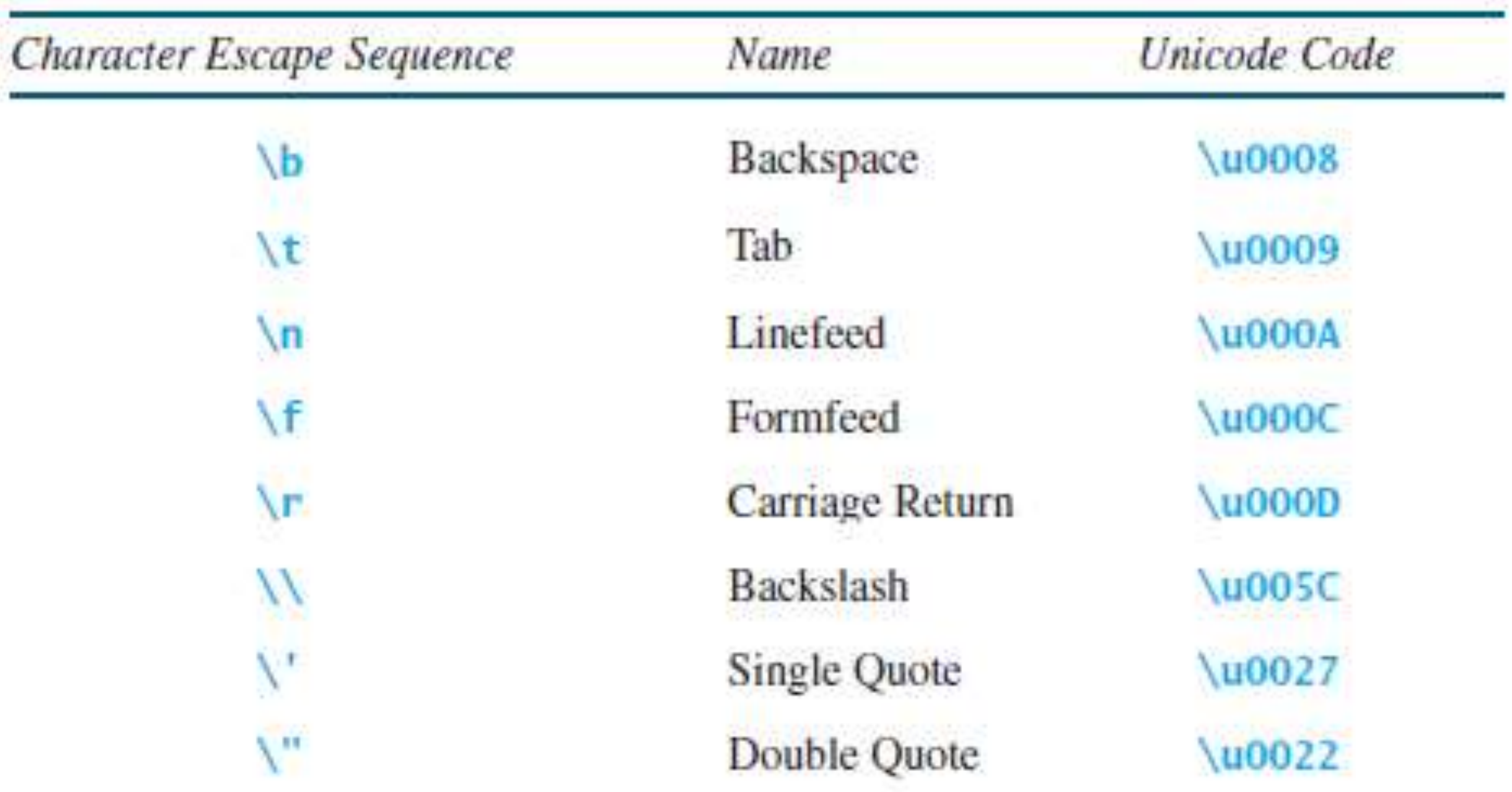
```
char letter = 'A';
char numChar = '4';
```

- ✓ Java supports Unicode
- ✓ 65,536 characters possible in a 16-bit encoding
- ✓ 16-bit Unicode takes two bytes.
- ✓ The range of a char is 0 to 65,535. There are no negative chars.
- ✓ The standard set of characters known as ASCII still ranges from 0 to 127 as always.
- ✓ The increment and decrement operators can also be used on char variables to get the next or preceding Unicode character

.

2. Character datatype

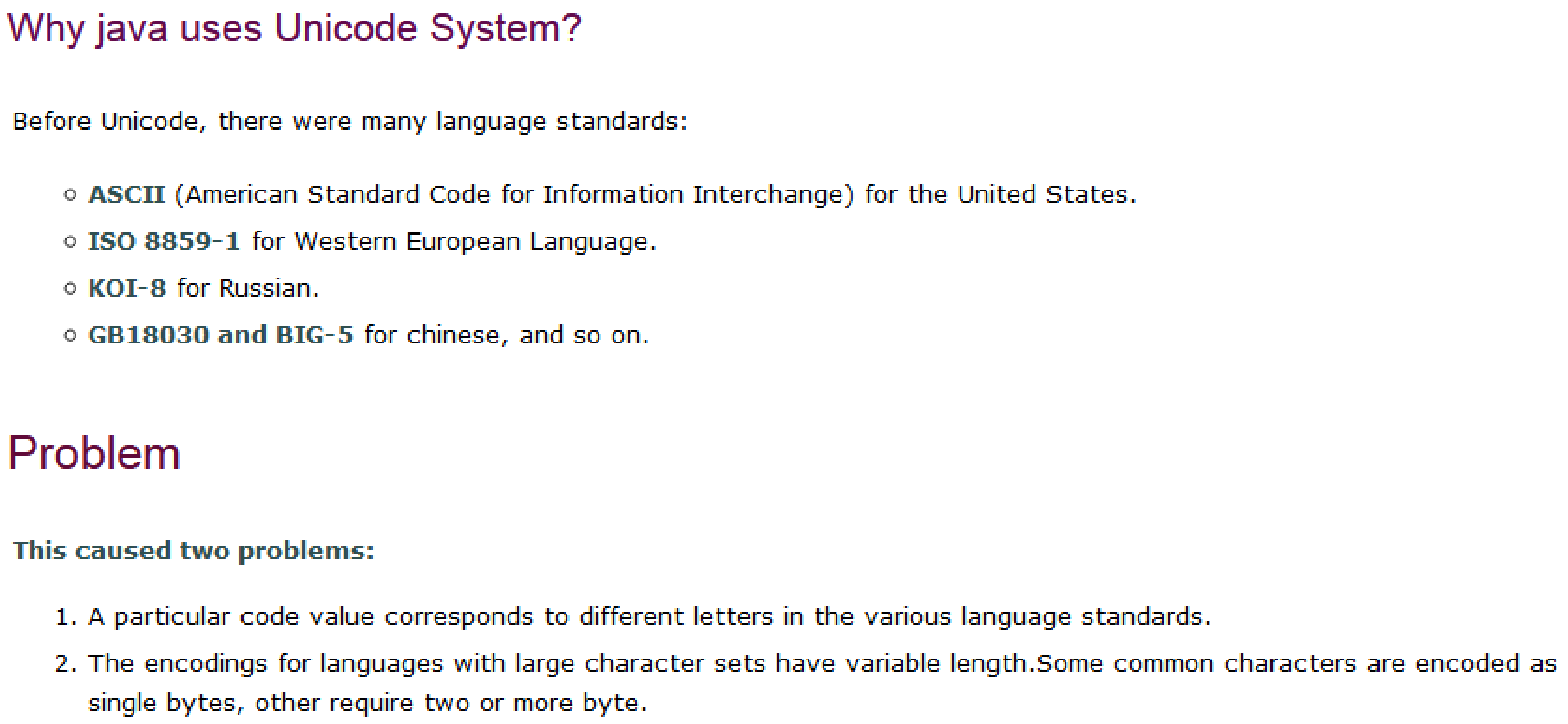
✓ Escape Sequences for special characters



Character Escape Sequence	*Name*	*Unicode Code*
\b	Backspace	\u0008
\t	Tab	\u0009
\n	Linefeed	\u000A
\f	Formfeed	\u000C
\r	Carriage Return	\u000D
\\	Backslash	\u005C
\'	Single Quote	\u0027
\"	Double Quote	\u0022

About Unicode

- Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages.



Why java uses Unicode System?

Before Unicode, there were many language standards:

- **ASCII** (American Standard Code for Information Interchange) for the United States.
- **ISO 8859-1** for Western European Language.
- **KOI-8** for Russian.
- **GB18030 and BIG-5** for chinese, and so on.

Problem

This caused two problems:

1. A particular code value corresponds to different letters in the various language standards.
2. The encodings for languages with large character sets have variable length.Some common characters are encoded as single bytes, other require two or more byte.

Solution

To solve these problems, a new language standard was developed i.e. Unicode System.

In unicode, character holds 2 byte, so java also uses 2 byte for characters.

lowest value:\u0000

highest value:\uFFFF

Datatype and Variables (do it yourself)

3. Boolean datatype

- ✓ The boolean data type is used to declare Boolean variables.
- ✓ A boolean variable can hold one of the two values: true and false
- ✓ true and false are literals

Automatic type promotion in expressions

Consider the following line of code:

```
    byte b = 50;
     b = b * 2;
```

Automatic type promotion in expressions

Consider the following line of code:

```
    byte b = 50;
     b = b * 2;              // Error! Cannot assign an int to a byte!
```

The code is attempting to store 50 * 2, a perfectly valid byte value, back into a byte variable. However, <u>because the operands were automatically promoted to int when the expression was evaluated, the result has also been promoted to int.</u>

Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast.

Exercise

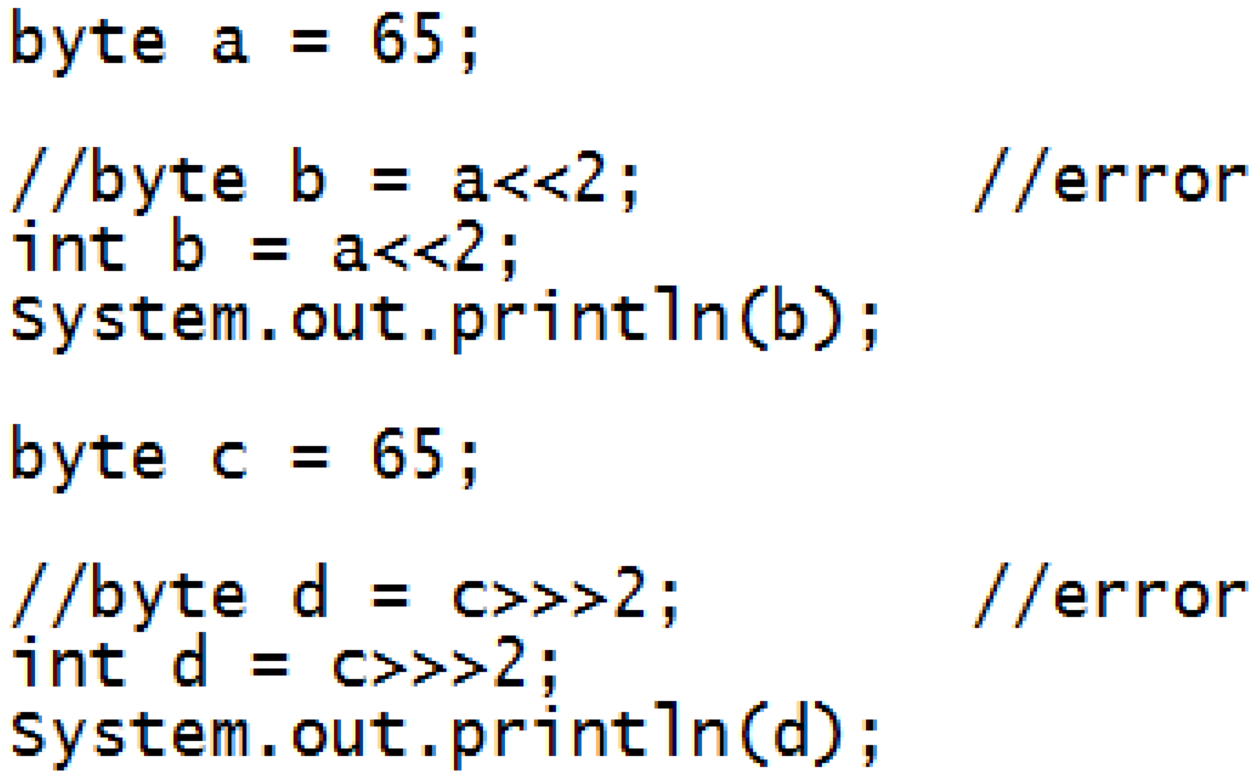
```
1) What is output of the following code:
        byte a = 5;
        byte b = 10;
        byte c = a + b;
        System.out.println(c);


2) What is output of the following code:
        byte a = 5;
        byte b = 10 * a;
        System.out.println(b);


3) What is output of the following code:
        byte a = 7;
        byte b = (byte)20 * a;
        System.out.println(b);


4) What is output of the following code:
        byte a = 7;
        byte b = (byte) (20 * a);
        System.out.println(b);
```

Example 1



```
byte a = 65;

//byte b = a<<2;            //error
int b = a<<2;
System.out.println(b);

byte c = 65;

//byte d = c>>>2;           //error
int d = c>>>2;
System.out.println(d);
```

Example 2

```
                byte e = -65;
                byte f = (byte)(e>>>24);
                System.out.println(f);

/*65 --> 00000000 00000000 00000000 01000001
 -65 --> 11111111 11111111 11111111 10111111
           >>>24
         00000000 00000000 00000000 11111111 --> (byte of this) --> 11111111

 -65 --> 11111111 11111111 11111111 10111111
           >>24
         11111111 11111111 11111111 11111111 --> (byte of this) --> 11111111
*/
                int g = 128;
                int h = g<<4;
                System.out.println(h);

                char ch = 'a';
                int bb = ch>>2;
                System.out.println(bb);
```

Type Conversion, Casting and Promotion

1. Numeric Type Conversions

Consider the following statements:

```
byte i = 100;
long k = i * 3 + 4;
double d = i * 3.1 + k / 2;
```

When performing a binary operation involving two operands of different types, Java automatically converts the operand based on the following rules:

1. All byte, short, and char values are promoted to int.
2. if one operand is a long, the whole expression is promoted to long.
3. If one operand is a float, the entire expression is promoted to float.
4. If any of the operands is double, the result is double.
5. Although char and short are having same size, char to short and short to char conversion is NARROWING. byte to char conversion is also Narrowing.

Type Conversion, Casting and Promotion

1. Numeric Type Conversions

The result of 1/2 is 0 but result of 1.0/2 is 0.5

```
                  range increases
          ------------------------------>
byte, short, int, long, float, double
```

Casting converts a value of one data type into a value of another data type

* widening a type – 1. Casting a variable of a type with a small range to a variable of a type with a larger range
2. can be performed automatically (implicit casting)

* narrowing a type – 1. Casting a variable of a type with a large range to a variable of a type with a smaller range
2. must be performed explicitly (explicit casting)

Note: Not all the types are compatible,Ex. boolean is not compatible with numeric or char types.

Type Conversion, Casting and Promotion

1. Numeric Type Conversions

The result of 1/2 is 0 but result of 1.0/2 is 0.5

$$\xrightarrow{\texttt{range increases}}$$

```
byte, short, int, long, float, double
```

Implicit casting

double d = 3; (type widening)

Explicit casting

int i = (int)3.0; (type narrowing)

int i = (int)3.9; (Fraction part is truncated)

What is wrong? int x = 5 / 2.0; It would work if int x = (int)(5 / 2.0);

2. Character Type Conversions

- ✓ A **char** can be cast into any numeric type, and vice versa
- ✓ When an integer is cast into a char, only its lower 16 bits of data are used; the other part is ignored

```
int i = 65;
char ch = (char)i;
System.out.println(ch); // ch is character A
```

- ✓ When a floating-point value is cast into a char, the floating-point value is first cast into an int, which is then cast into a char

```
char ch = (char)65.25; // decimal 65 is assigned to ch
System.out.println(ch); // ch is character A
```

- ✓ Implicit casting can be used if the result of a casting fits into the target variable. Otherwise, explicit casting must be used.

```
int i = 'a';                    //correct
byte b = i;       // incorrect   byte b = (byte) i; //correct
```

2. Character Type Conversions

- ✓ All numeric operators can be applied to char operands. A char operand is automatically cast into a number if the other operand is a number or a character. If the other operand is a string, the character is concatenated with the string.

```
class Test
    { public static void main(String [] ar)
        {
            int i = '2' + '3';                    // (int)'2' is 50 and (int)'3' is 51
            System.out.println("i is " + i);      // i is 101
            int j = 2 + 'a';                      // (int)'a' is 97
            System.out.println("j is " + j);      // j is 99
            System.out.println(j + " is the Unicode for character "+ (char)j);
            System.out.println("Chapter " + '2');
           }
      }
```

- O/P is
    - i is 101
    - j is 99
    - 99 is the Unicode for character c
    - Chapter 2

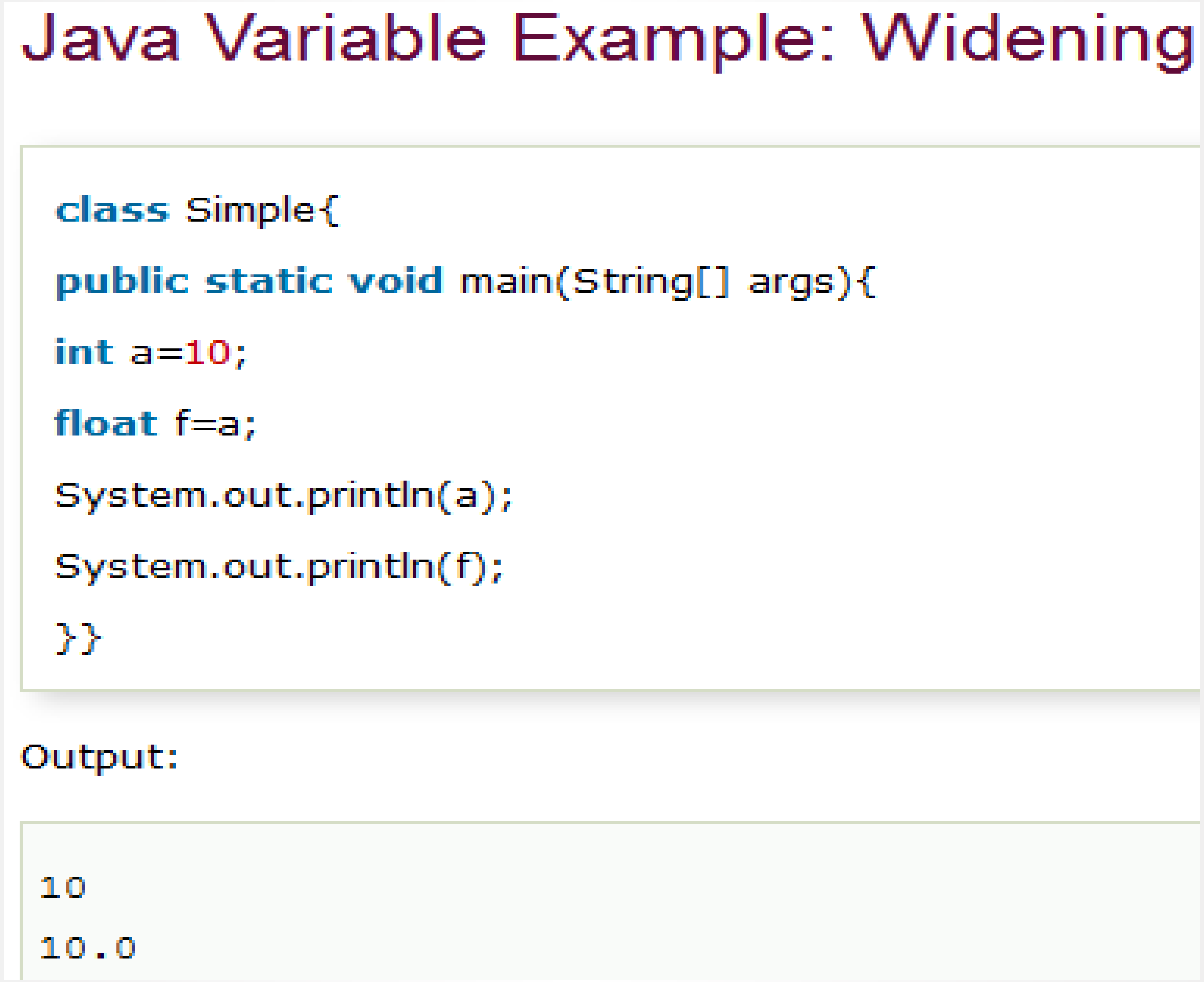
Example 1

```
char ch = 'A';
short s = ch;    /*error since char and short are not compatible with each
                   other. Hence they have to be type-casted while assinging
                   to each other.
                 */

short s = 32;
char ch = s;     //error


float f = 234222222222222223L;    /* no error since value will be stored in
                                     exponential form (i.e. power of 10)
                                  */

System.out.println(f);  //prints 2.34222228E17
long l = (long)f;
System.out.println(l);  //prints 234222227556401152
```

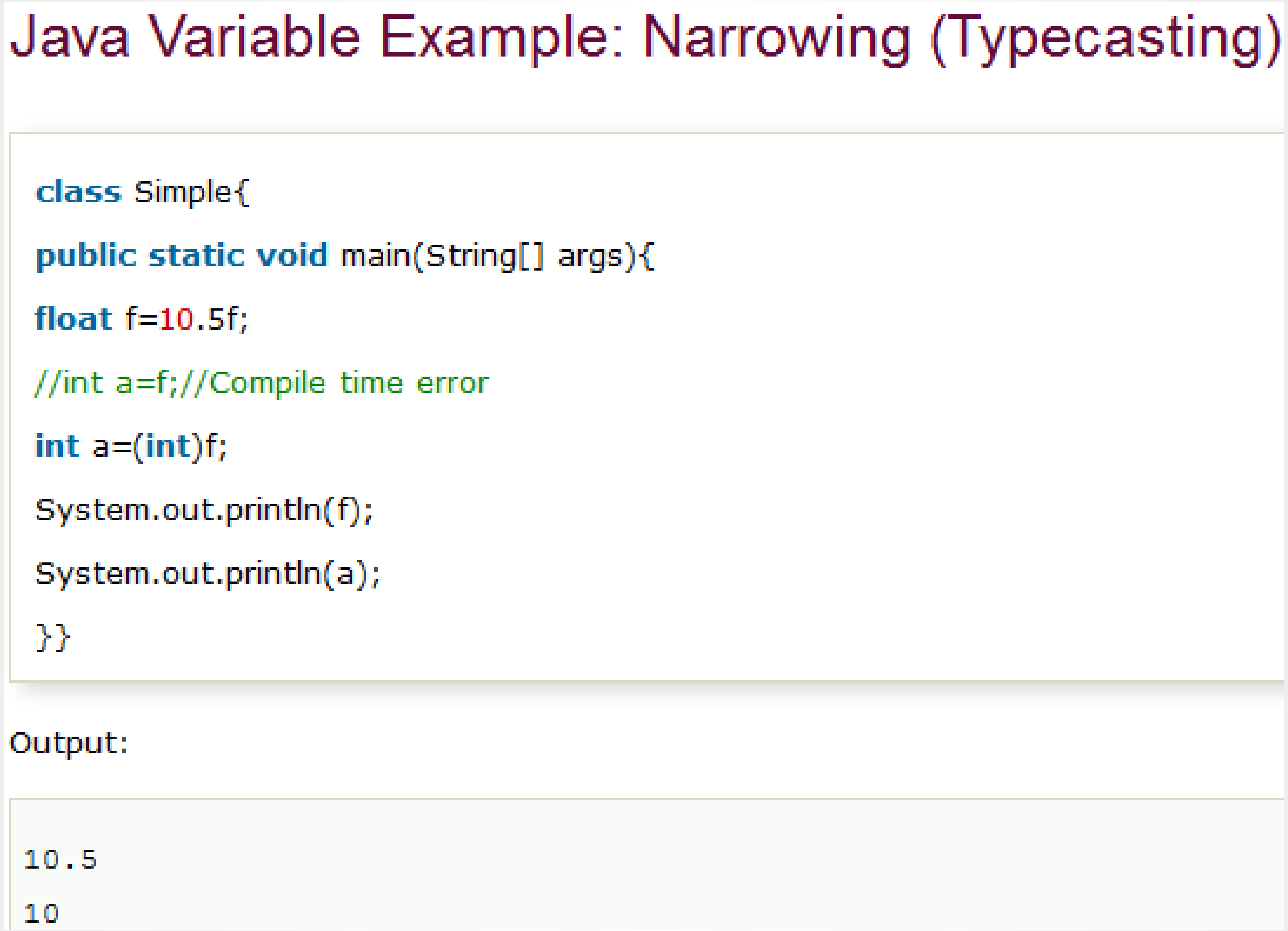


Java Variable Example: Widening

```
class Simple{
public static void main(String[] args){
int a=10;
float f=a;
System.out.println(a);
System.out.println(f);
}}
```

Output:

```
10
10.0
```

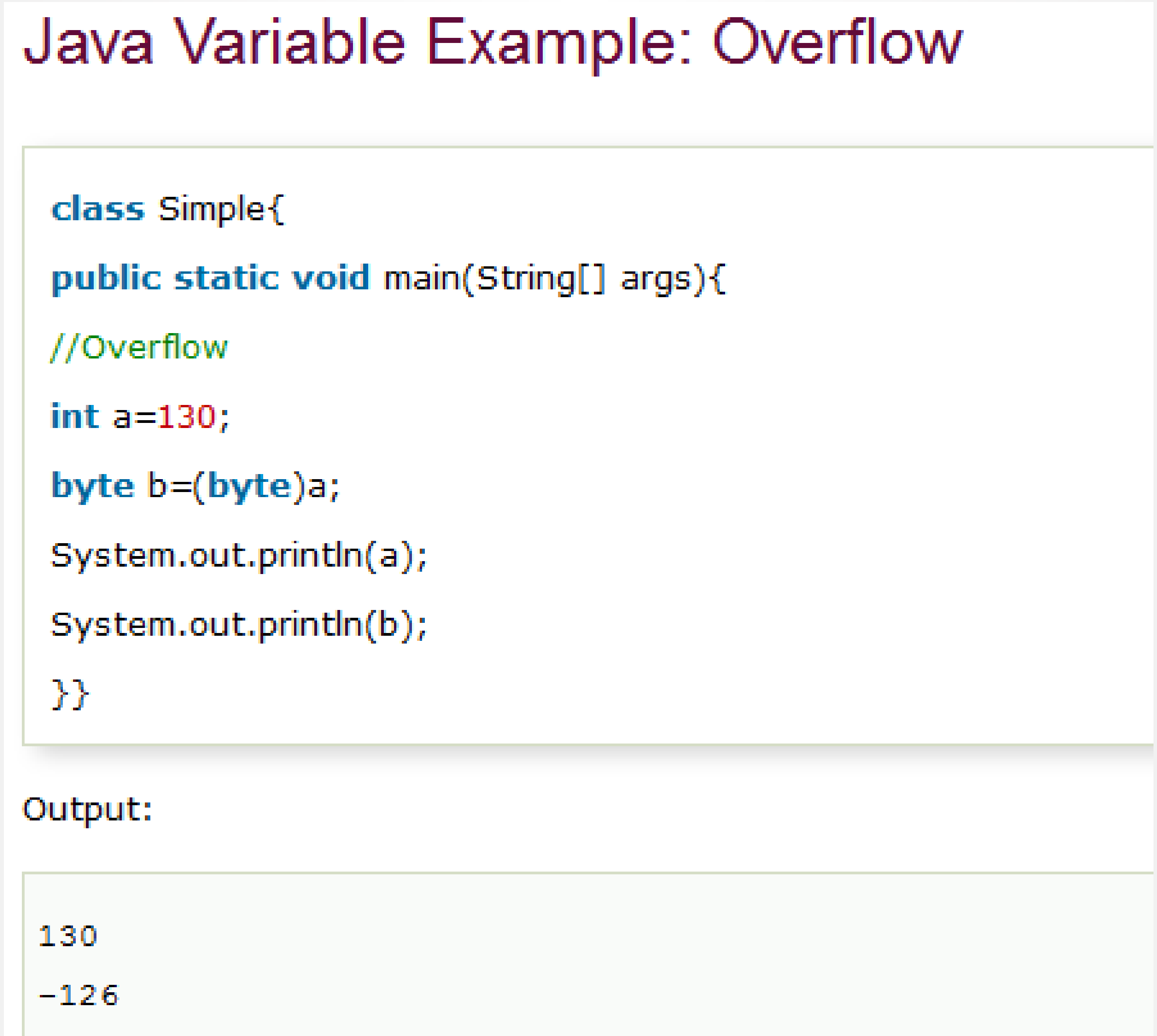


Java Variable Example: Narrowing (Typecasting)

```
class Simple{
public static void main(String[] args){
float f=10.5f;
//int a=f;//Compile time error
int a=(int)f;
System.out.println(f);
System.out.println(a);
}}
```

Output:

```
10.5
10
```

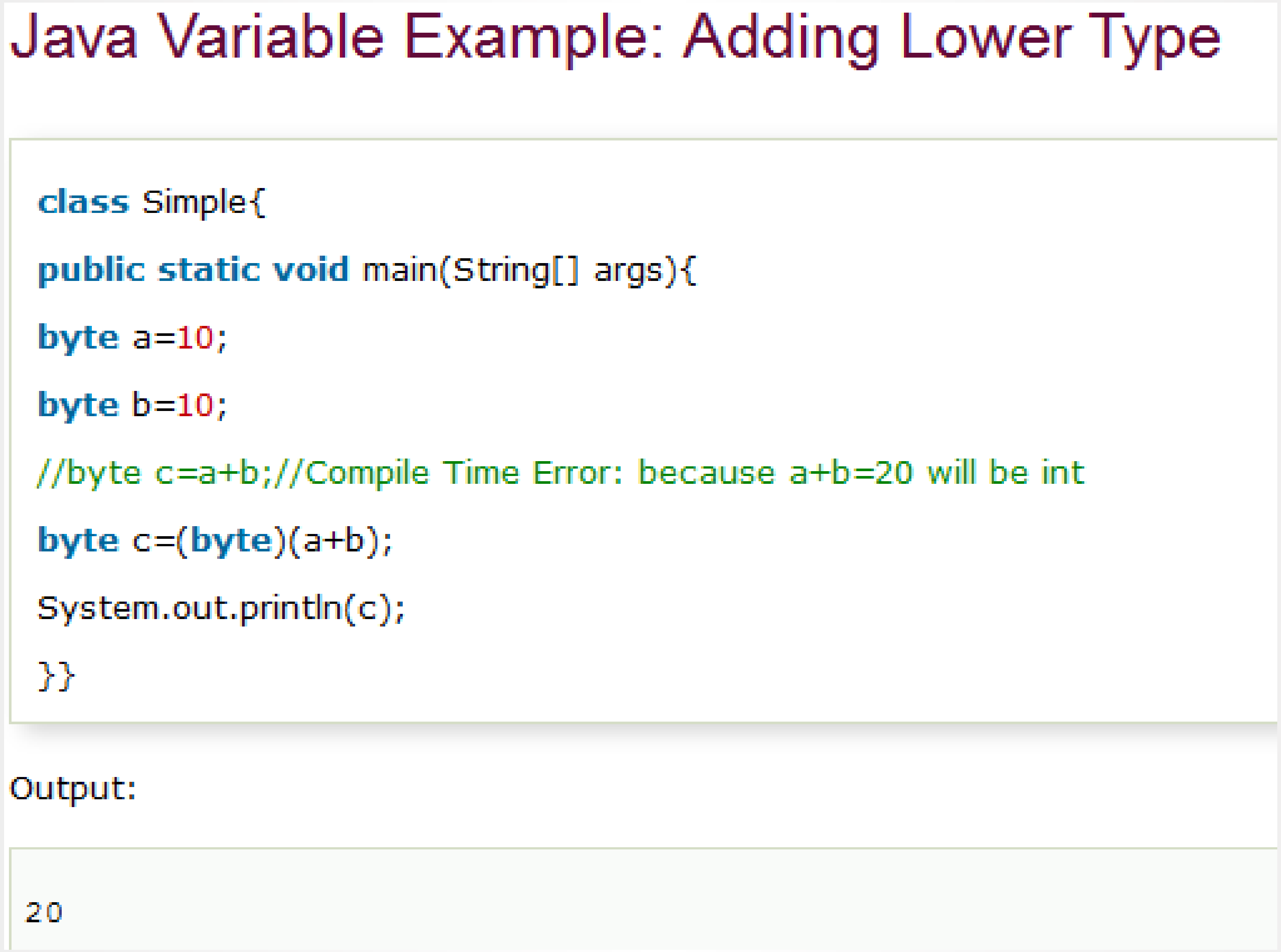


Java Variable Example: Overflow

```
class Simple{
public static void main(String[] args){
//Overflow
int a=130;
byte b=(byte)a;
System.out.println(a);
System.out.println(b);
}}
```

Output:

```
130
-126
```



Java Variable Example: Adding Lower Type

```
class Simple{
public static void main(String[] args){
byte a=10;
byte b=10;
//byte c=a+b;//Compile Time Error: because a+b=20 will be int
byte c=(byte)(a+b);
System.out.println(c);
}}
```

Output:

```
20
```

4. Control Statements

1. Selection Statements
    - (a) if else
    - (b) nested ifs
    - (c) if-else-if ladder
    - (d) switch-case
2. Looping Statements
    - (a) while
    - (b) do-while
    - (c) for --- (comma operator)
    - (d) for-each (to be discussed in arrays)
3. Jumping Statement
    - (a) break
    - (b) continue
    - (c) return

Selection Statements

- Java supports two selection statements: **if** and **switch.**

if statement

```
if (condition) statement1;
else statement2;
```

- Each statement may be a single statement or a compound statement enclosed in curly braces (block).
- The condition is any expression that returns a **boolean value.**
- **The else** clause is optional.
- If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed.
- In no case will both statements be executed.

Nested ifs

- A nested if is an if statement that is the target of another if or else.
- In nested ifs an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.

```
if (i == 10) {
            if (j < 20) a = b;
            if (k > 100) c = d;      // this if is
            else a = c;              // associated with this else
            }
else a = d;                          // this else refers to if(i == 10)
```

The if-else-if Ladder

- A sequence of nested ifs is the if-else-if ladder.

```
if(condition)
  statement;
else if(condition)
  statement;
else if(condition)
  statement;
...
else
  statement;
```

- The if statements are executed from the top to down.

switch

- The switch statement is Java’s multi-way branch statement.
- provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
- provides a better alternative than a large series of **if-else-if statements.**

```
switch (expression) {
        case value1:
                // statement sequence
                break;
        case value2:
                // statement sequence
                break;
        ...
        case valueN:
                // statement sequence
                break;
        default:
                // default statement sequence
                }
```

- **The expression must be of type byte, short, int, or char**.
- Each of the values specified in the case statements must be of a type compatible with the expression.
- Each case value must be a unique literal (i.e. constant not variable).
- Duplicate case values are not allowed.
- The value of the expression is compared with each of the literal values in the case statements.
- If a match is found, the code sequence following that case statement is executed.
- If none of the constants matches the value of the expression, then the default statement is executed.
- The default statement is optional.

- If no case matches and no default is present, then no further action is taken.
- The break statement is used inside the switch to terminate a statement sequence.
- When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement.

```
class SampleSwitch {
  public static void main(String args[]) {
        for(int i=0; i<6; i++)
                switch(i) {
                        case 0:
                                System.out.println("i is zero.");
                                break;
                        case 1:
                                System.out.println("i is one.");
                                break;
                        case 2:
                                System.out.println("i is two.");
                                break;
                        default:
                                System.out.println("i is greater than 2.");
                           }
                        }
```

Nested switch Statements

- When a switch is used as a part of the statement sequence of an outer switch. This is called a nested switch.

```
switch(count) {
        case 1:
                switch(target) { // nested switch
                        case 0:
                                System.out.println("target is zero");
                                break;
                        case 1: // no conflicts with outer switch
                                System.out.println("target is one");
                                break;
                              }
                        break;
        case 2: // ...
```

Difference between ifs and switch

- switch can only test for equality, whereas if can evaluate any type of Boolean expression. That is, the switch looks only for a match between the value of the expression and one of its case constants.

- A switch statement is usually more efficient than a set of nested ifs.

- Note: No two case constants in the same switch can have identical values. Of course, a switch statement and an enclosing outer switch can have case constants in common.

ITERATION STATEMENTS (LOOPS)

Iteration Statements

- In Java, iteration statements (loops) are:
  - for
  - while, and
  - do-while

- A loop repeatedly executes the same set of instructions until a termination condition is met.

While Loop

- While loop repeats a statement or block while its controlling expression is true.
- The condition can be any Boolean expression.
- The body of the loop will be executed as long as the condition expression is true.
- When condition becomes false, control passes to the next line of code immediately following the loop.

```
while(condition)
 {
    // body of loop
 }
```

```
class While
    {
        public static void main(String args[]) {
        int n = 10;
        char a = 'G';
        while(n > 0)
                {
                  System.out.print(a);
                  n--;
                  a++;
                        }
          }
    }
```

- The body of the loop will not execute even once if the condition is false.

- The body of the while (or any other of Java’s loops) can be empty. This is because a null statement (one that consists only of a semicolon) is syntactically valid in Java.

do-while

- The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

```
do {
        // body of loop
   } while (condition);
```

- Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression.
- If this expression is true, the loop will repeat. Otherwise, the loop terminates.

for Loop

```
for (initialization; condition; iteration)
    {
      // body
    }
```

- Initialization portion sets the value of loop control variable.
- Initialization expression is only executed once.
- Condition must be a Boolean expression. It usually tests the loop control variable against a target value.
- Iteration is an expression that increments or decrements the loop control variable.

The for loop operates as follows.

- When the loop first starts, the initialization portion of the loop is executed.
- Next, condition is evaluated. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
- Next, the iteration portion of the loop is executed.

```
class ForTable
        {
                public static void main(String args[])
                 {
                        int n;
                        int x=5;
                        for(n=1; n<=10; n++)
                          {
                           int p = x*n;
                           System.out.println(x+"*"+n +"="+ p);
                          }
                 }
        }
```

What will be the output?

```
class Loop
  {
        public static void main(String args[])
          {
                for(int i=0; i<5; i++);
                      System.out.println (i++);
           }
   }
```

Output: **Compile error**

Explanation: variable “i” declared inside for loop is destroyed as soon as the loop finishes executing. Hence, “variable undeclared” error in line: System.out.println (i++);

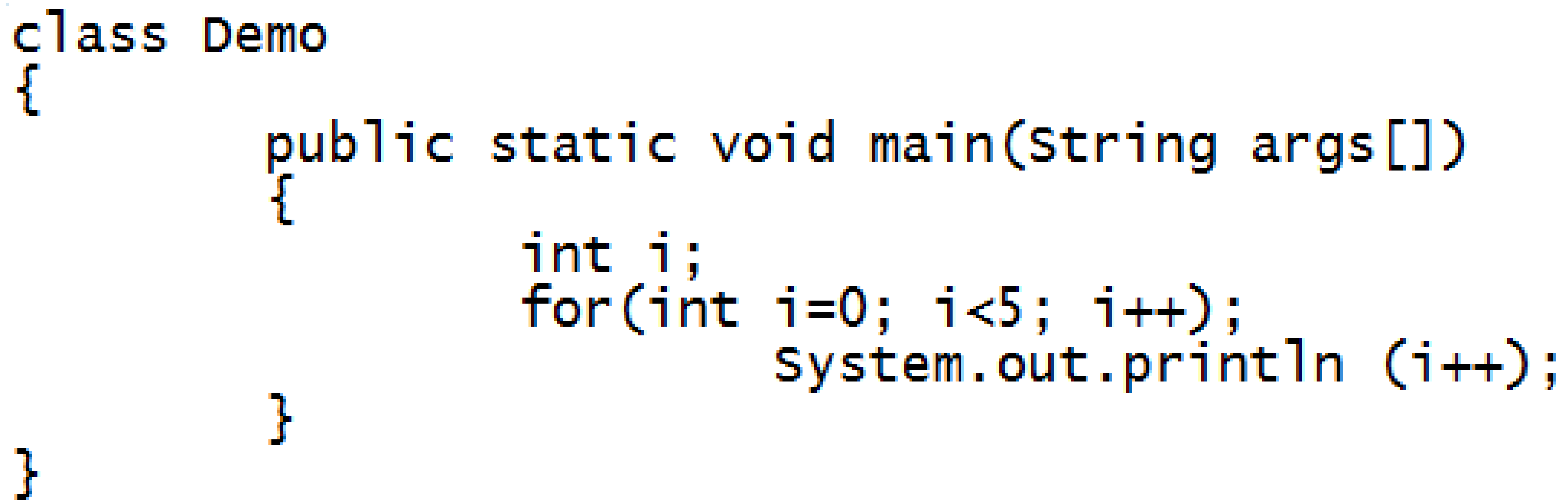
Declaring loop control variable inside loop

- We can declare the variable inside the initialization portion of the for.

```
for ( int i=0; i<10; i++)
        {
          System.out.println(i);
        }
```

- Note: The scope of this variable i is limited to the for loop and ends with the for statement.

What will be the output?



```
class Demo
{
        public static void main(String args[])
        {
                int i;
                for(int i=0; i<5; i++);
                        System.out.println (i++);
        }
}
```

Output: **Compile error**

Explanation: variable “i” re-declared inside for loop

Using multiple variables in a for loop

- More than one statement in the initialization and iteration portions of the for loop can be used.

Example 1:

```
class var2 {
        public static void main(String arr[]) {
                int a, b;
                b = 5;
                for(a=0; a<b; a++) {
                        System.out.println("a = " + a);
                        System.out.println("b = " + b);
                        b--;
                }
        }
}
```

- Comma (separator) is used while initializing multiple loop control variables.

Example 2:

```
class var21
      {
            public static void main(String arr[]) {
                  int x, y;
                  for(x=0, y=5; x<=y; x++, y--) {
                        System.out.println("x=  " +   x);
                        System.out.println(“y =  " +  y);
                  }
            }
      }
```

- Initialization and iteration can be moved out from for loop.

Example 3:

```
class Loopchk
{
   public static void main(String arr[])
   {
         for(int i=1, j=5; i>0 && j>2; i++, j--)
               System.out.println("i is: "+ i + "and j is: "+j);
         }
 }
```

For-Each Version of the for Loop

- Beginning with JDK 5, a second form of **for** was defined that implements a “for-each” style loop.
- For-each is also referred to as the **enhanced for loop.**
- Designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish.

for (type itr-var : collection) statement-block

- *type* specifies the type.
- *itr-var* specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.
- The collection being cycled through is specified by collection.
- With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var.
- The loop repeats until all elements in the collection have been obtained.

```
class ForEach
{
	public static void main(String arr[])
	{
			int num[] = { 1, 2, 3, 4, 5 };
			int sum = 0;
			for(int i : num)
			{
				System.out.println("Value is: " + i);
				sum += i;
			}
			System.out.println("Sum is:  " + sum);
	}
}
```

Iterating Over Multidimensional Arrays

```
class ForEachMArray {
        public static void main(String args[]) {
                int sum = 0;
                int num[][] = new int[3][5];
        // give num some values
                for(int i = 0; i < 3; i++)
                        for(int j=0; j < 5; j++)
                                num[i][j] = (i+1)*(j+1);
        // use for-each for to display and sum the values
                for(int x[] : num) {
                        for(int y : x) {
                                System.out.println("Value is: " + y);
                                sum += y;
                                  }
                               }
                System.out.println("Summation: " + sum);
```

Exercise

Wap to store records of "n" students, where each student record will contain his subject-marks

Assume that each student has pursued variable number of subject-courses.

Display all the records using enhanced for loop.

Nested Loops

```
class NestedLoop
  {
          public static void main(String arr[])
                   {
                            int i, j;
                            for(i=0; i<10; i++)
                            {
                                     for(j=i; j<10; j++)
                                            System.out.print(“* ”);
                                     System.out.println(  );
                            }
                   }
  }
```

Jump Statements

- Java supports three jump statements:
    - break
    - continue
    - return
- These statements transfer control to another part of the program.

- Break: break statement has three uses.
    - terminates a statement sequence in a switch statement
    - used to exit a loop
    - used as a “civilized” form of goto

Example 1:

```
class BreakLoop
  {
          public static void main(String arr[])
                    {
                              for(int i=0; i<100; i++)
                              {
                                        if(i == 10)
                                              break;        // terminate loop if i is 10
                                        System.out.println("i: " + i);
                              }
                              System.out.println("Loop complete.");
                    }
   }
```

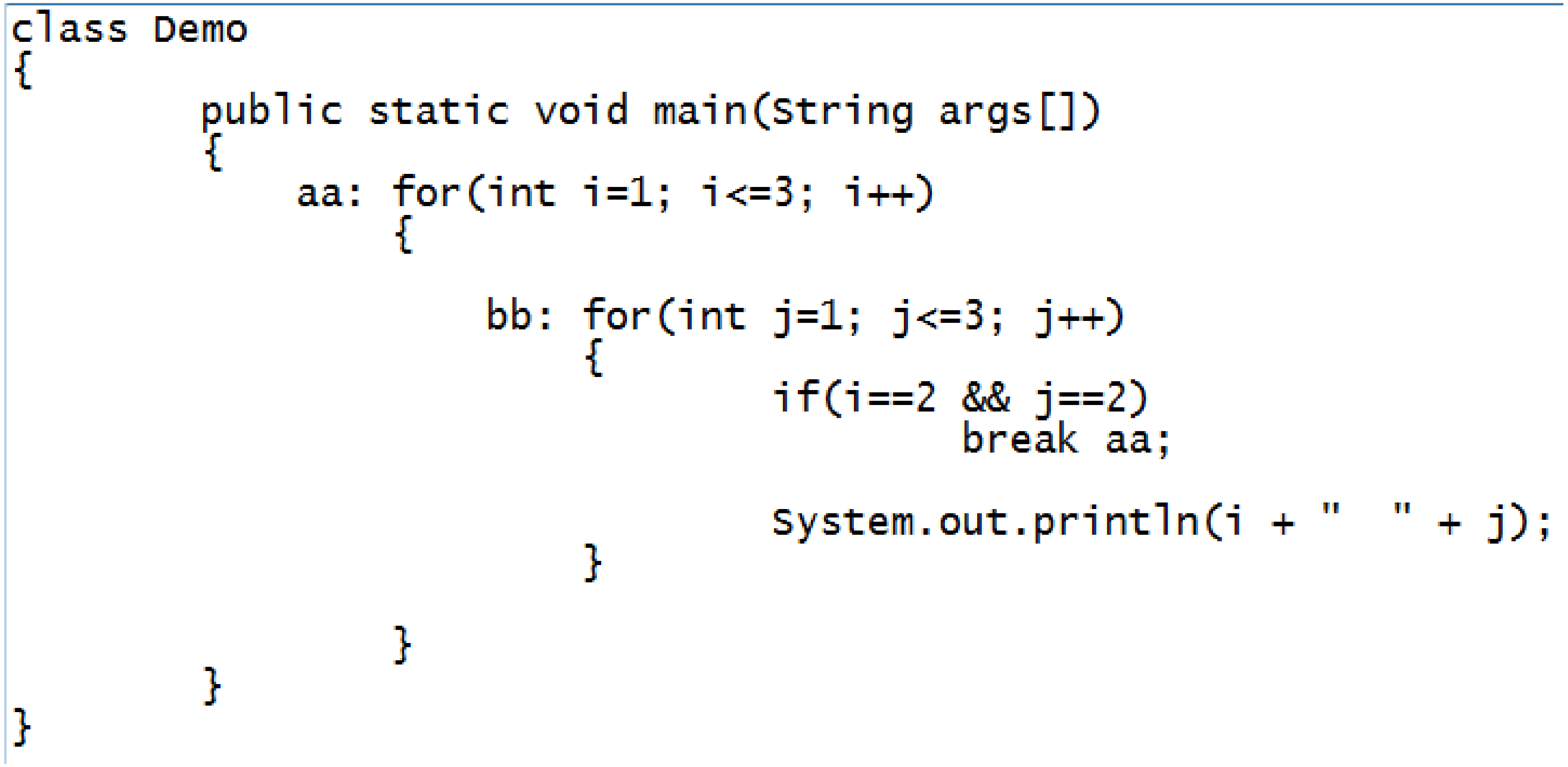
- Java defines an expanded form of the break statement.
- By using this form of break, we can break out of one or more blocks of code.
- These blocks need not be part of a loop or a switch. They can be any block.
- Further, we can specify precisely where execution will resume, because this form of **break works with a label.**

break *label;*

- When this form of break executes, control is transferred out of the named block. The labeled block must enclose the break statement.

```
class BreakLoop
  {
        public static void main(String arr[])
        {
           outer: for(int i=0; i<3; i++)
              {
                     System.out.print("Pass " + i + ": ");
                     for(int j=0; j<100; j++)
                     {
                          if(j == 10)
                                break outer;     // exit both loops
                          System.out.print(j + " ");
                     }
                     System.out.println("This will not print");
              }
              System.out.println("Loops complete.");
        }
```

What will be the output?



```
class Demo
{
        public static void main(String args[])
        {
            aa: for(int i=1; i<=3; i++)
                {

                    bb: for(int j=1; j<=3; j++)
                        {
                                if(i==2 && j==2)
                                        break aa;

                                System.out.println(i + "   " + j);
                        }

                }
        }
}
```

Output:

```
1  1

1  2

1  3

2  1
```

What will be the output?

```
class Demo
{
          public static void main(String args[])
          {
               aa: for(int i=1; i<=3; i++)
                    {

                         bb: for(int j=1; j<=3; j++)
                              {
                                        if(i==2 && j==2)
                                                  continue aa;

                                        System.out.println(i + "   " + j);
                              }

                    }

          }
}
```

Output:

```
1  1
1  2
1  3
2  1
3  1
3  2
3  3
```

If you use **break bb;**, it will break inner loop only which is the default behavior of any loop.

```
public class LabeledForExample {
public static void main(String[] args) {
    aa:
        for(int i=1;i<=3;i++){
            bb:
                for(int j=1;j<=3;j++){
                    if(i==2&&j==2){
                        break bb;
                    }
                    System.out.println(i+" "+j);
                }
        }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

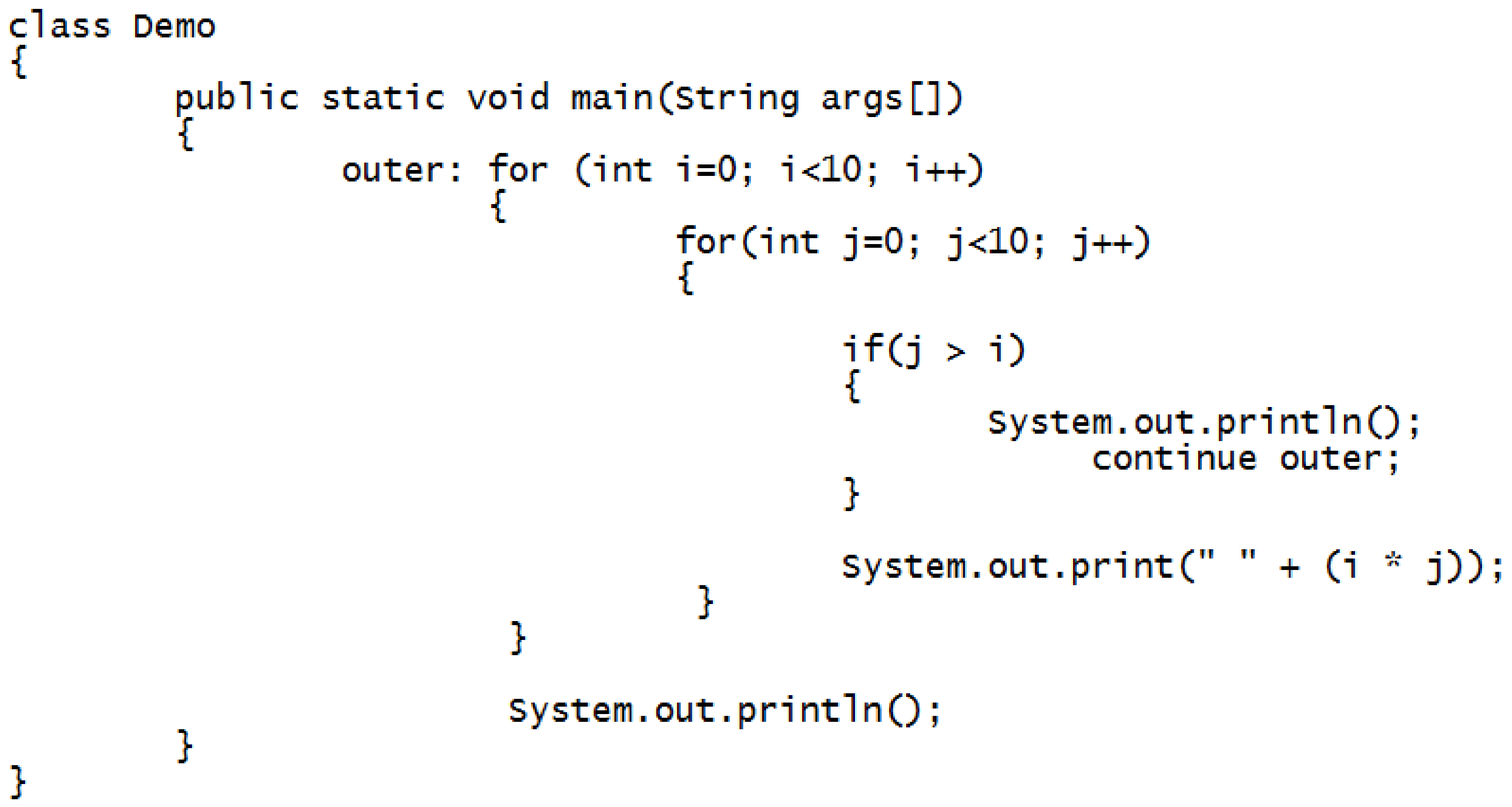
continue

- In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop.
- In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression.
- For all three loops, any intermediate code is bypassed.

- Example:

```
class Continue
{
        public static void main(String args[])
        {
                for(int i=0; i<5; i++)
                {
                        System.out.println(i + " ");
                         if (i%2 == 0)
                                continue;
                        System.out.println("No Continue");
                }
        }
}
```

- Similar to “break” statement, “continue” may specify a label to describe which enclosing loop to continue.



```
class Demo
{
        public static void main(String args[])
        {
                outer: for (int i=0; i<10; i++)
                       {
                                for(int j=0; j<10; j++)
                                {

                                        if(j > i)
                                        {
                                               System.out.println();
                                                    continue outer;
                                        }

                                        System.out.print(" " + (i * j));
                                 }
                        }

                        System.out.println();
        }
}
```

return

- The return statement is used to explicitly return from a method.
- It causes program control to transfer back to the caller of the method.

Example:

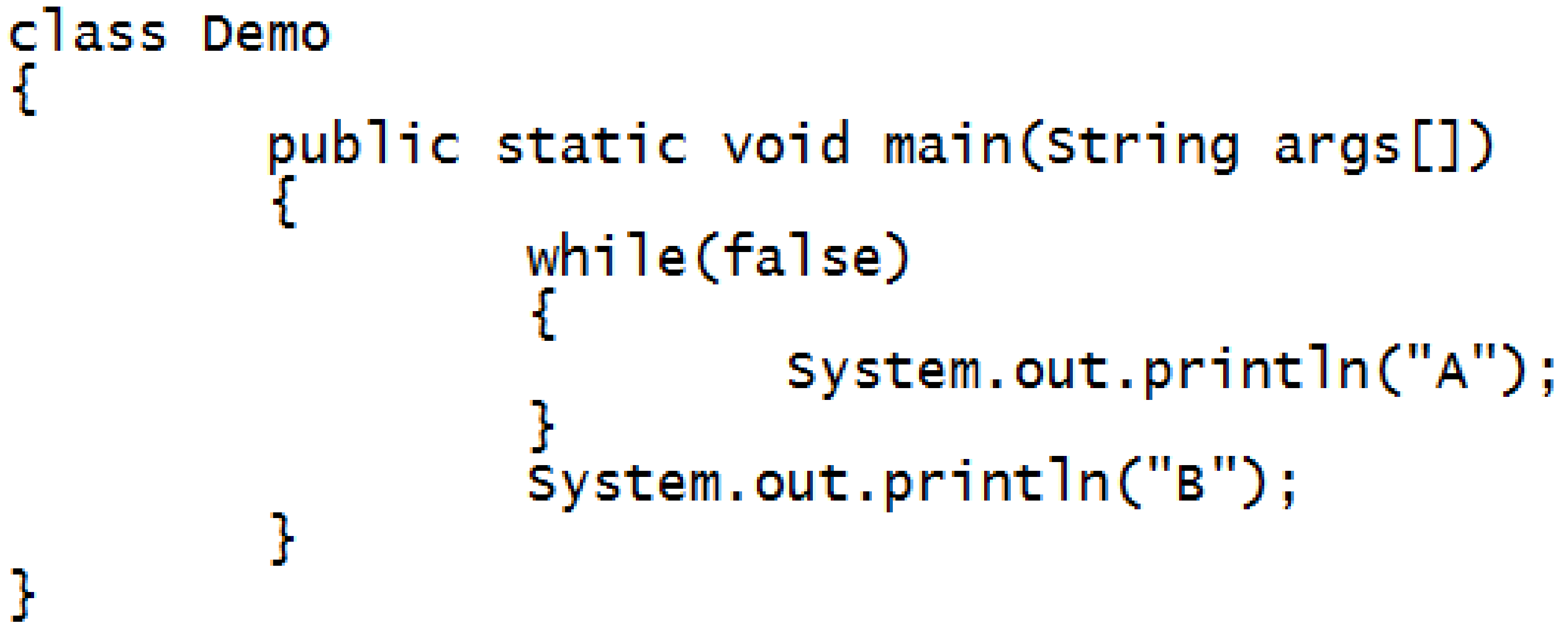
```
class Return {
        public static void main(String args[]) {
                boolean t = true;
                int i;
                System.out.println("Before the return.");
                if(t)
                {
                        i = 2;
                        return;                 // return to caller

                        System.out.println("This won't execute.“ + i);
                }
        }
```

If you have written the “return” statement (**without any condition**) before any other statement **in same block**, then Java compiler would flag “un-reachable code” error because the compiler would know that the next statement (after return statement) would never execute.

“code un-reachable” error

What is the output of following:



```
class Demo
{
        public static void main(String args[])
        {
                while(false)
                {
                        System.out.println("A");
                }
                System.out.println("B");
        }
}
```