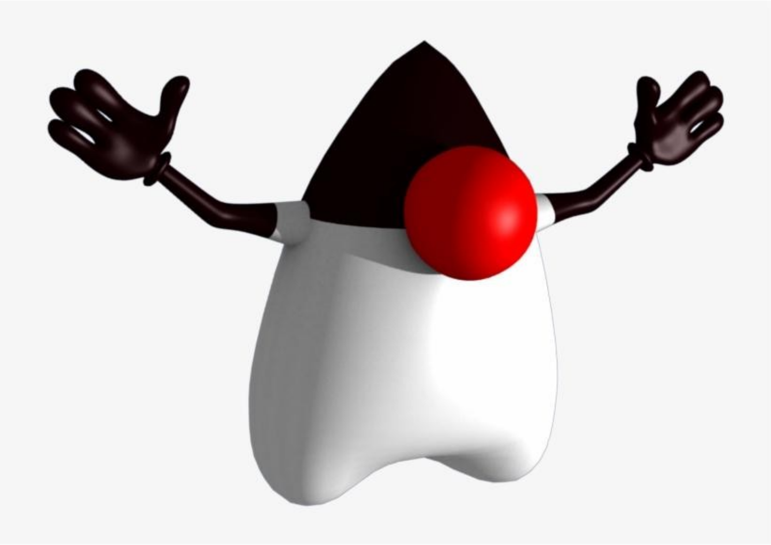
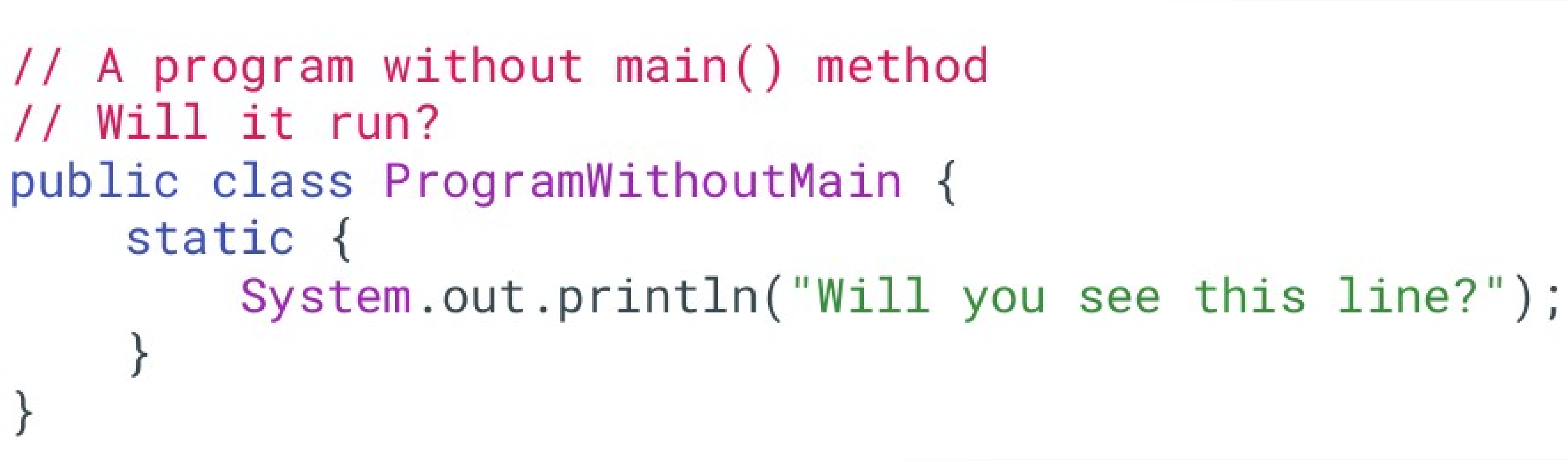
Java Program Constructs



sourcemind

A Program without main()



```java
// A program without main() method
// Will it run?
public class ProgramWithoutMain {
    static {
        System.out.println("Will you see this line?");
    }
}
```

Class Loading

- The JVM loads certain classes from the class path (the path that contains Java classes)
- You can invoke a main() method of a class that exists the the class path
- After JVM finds and loads a class, it executes the static block
- Was the execution of ProgramWithoutMain successful?

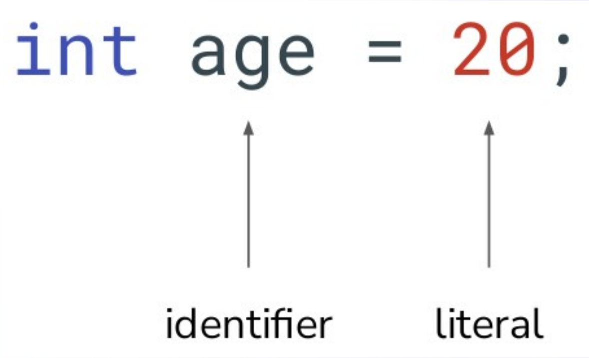
Java Program Constructs

Java Program Constructs

- Literal values and identifiers
- Java data types
- Expressions and operators
- Statements
- Methods
- Classes
- Packages

Java Literals and Identifiers

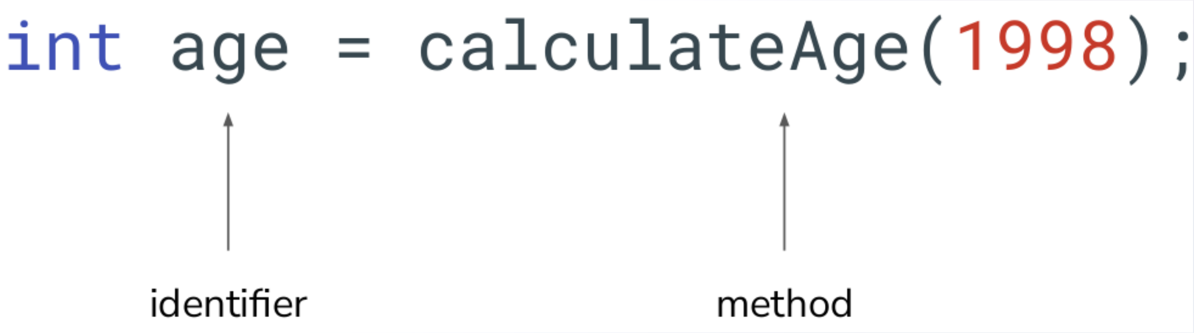
- A literal in Java is representation of a value, such as the value of a person’s name, age and height.
- An identifier is used for finding other Java elements, including literals. Simply,a name given to some part of a Java program.



```
int age = 20;
```

Java Literals and Identifiers

- An identifier can be used to refer not only to a literal value (e.g. 20) but also a calculated result from a method and other elements as well.
- Example: Given birth year, calculate age and use “age” identifier to access it.
- Sometimes we can also use the word “variable” for “identifier”.

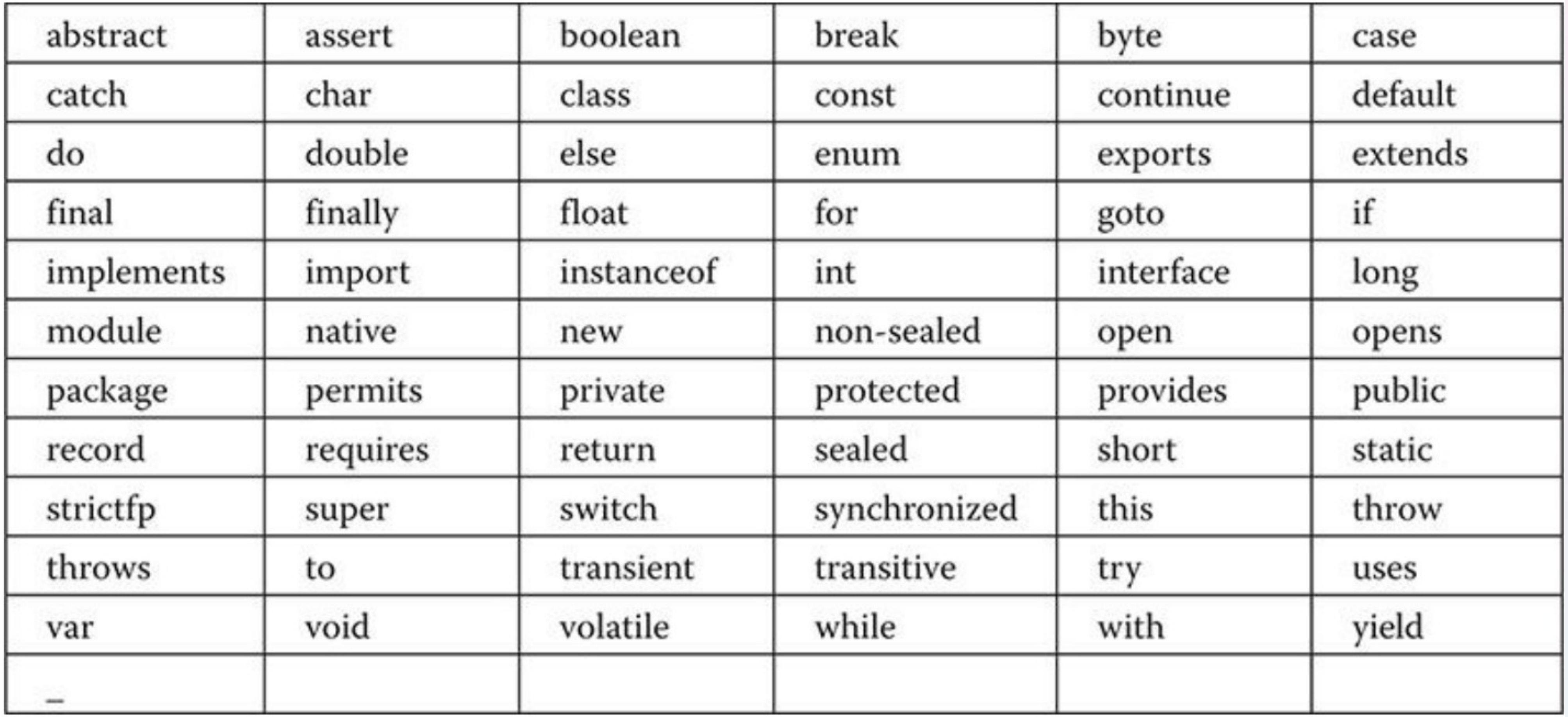


```java
int age = calculateAge(1998);
```

Valid Identifier Names

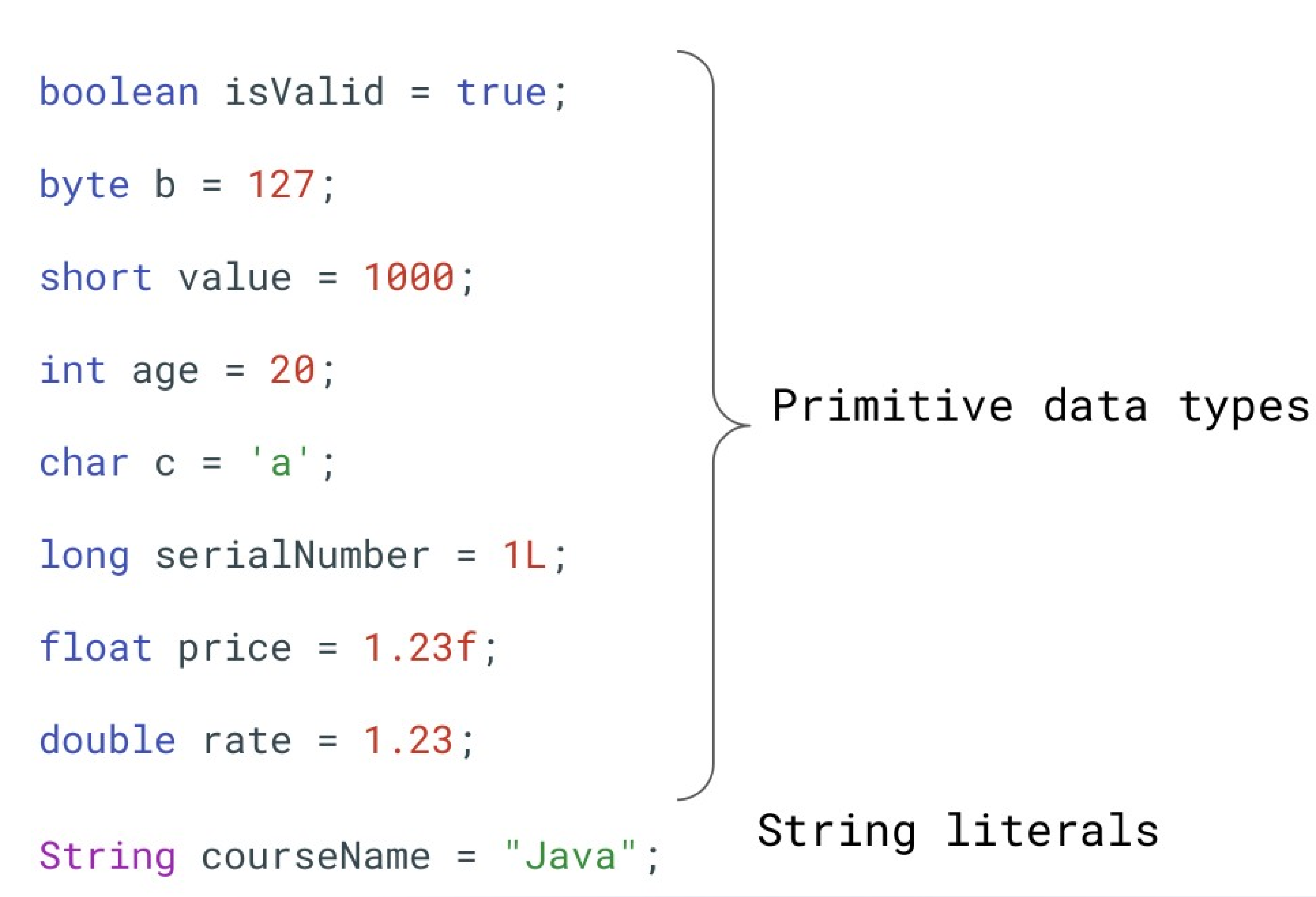
- A sequence of:
  - Uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.
- The dollar-sign character is not intended for general use.
- They must not begin with a number.
- Cannot use Java keywords (next slide).
- Cannot be: true, false, null.
- Valid:
  - Avg, avg, _avg, $avg, average_1, $_average_1
- Invalid:
  - 1num, average-1, my/value,

Java keywords



abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	exports	extends
final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long
module	native	new	non-sealed	open	opens
package	permits	private	protected	provides	public
record	requires	return	sealed	short	static
strictfp	super	switch	synchronized	this	throw
throws	to	transient	transitive	try	uses
var	void	volatile	while	with	yield
_					

Java Data Types



```java
boolean isValid = true;

byte b = 127;

short value = 1000;

int age = 20;

char c = 'a';

long serialNumber = 1L;

float price = 1.23f;

double rate = 1.23;

String courseName = "Java";
```

Primitive data types

String literals

Java: Strongly Typed Language

- Every variable and expression has a type
- double x = 1; // Type is defined as a double.
- x + 2 // The expression is evaluated as a double.
- int y = x + 2; // No automatic conversion, since integer is smaller than double.
- boolean t = 0; // incompatible types: int cannot be converted to boolean
- The Java compiler checks all the type definitions and assignments during compile time.
- What about:
- String a = “1”;
- String b = a + 1; // What is the value of b?

Exercise #5

- Use Java interactive shell (jshell) to:

1.Define variables to hold some data

- Name, age, hobbies, walking distance per day, grade, passing grade,...

2.Print some of the variables to the console output.

3.Define the following integer literals. What is the difference between them?

- int a = 17;
- int b = 021;
- int c = 0x11;
- int d = 0b10001;

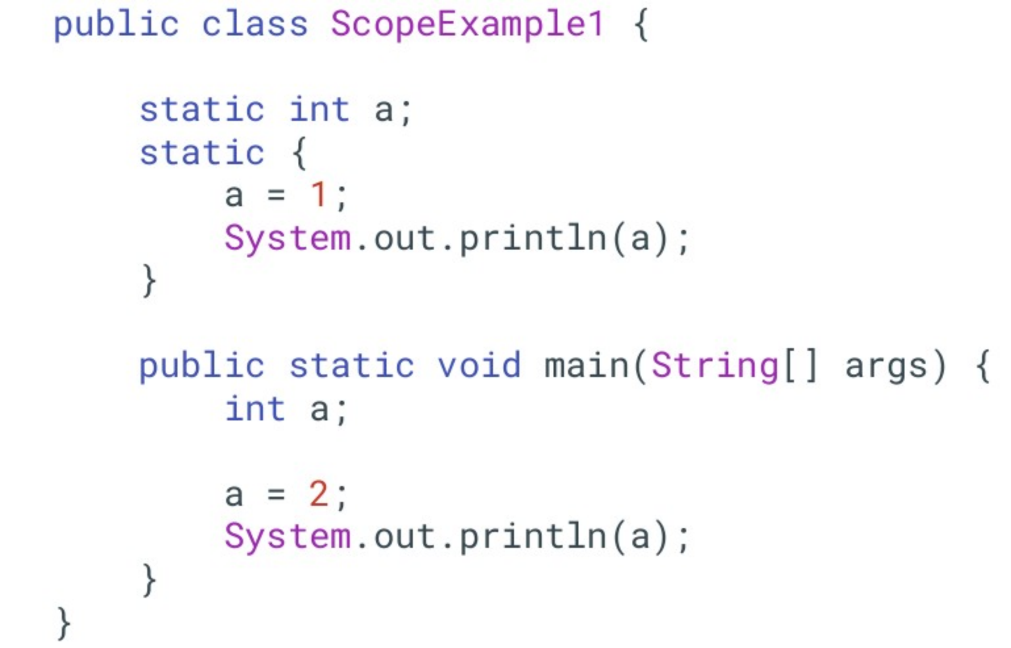
4.Find the maximum values you can store in byte, int and long

Primitive Data Types vs. String

- Why some data types are defined lowercase but “String” is defined uppercase?
  - int, double, char,... **String**
- Java has two categories of data types:
  - Primitive data
  - Objects
- The power of Java and other OOP languages is their ability to define and use custom made objects based on custom made data types
- Custom made data types = Class
- Java has a set of predefined classes such as **String, Date, File**
- By convention, all Java classes are defined in camel case **SomethingLikeThis**

Variables Scope

- Each scope is defined by curly braces { }
- Two major scopes:
    - Class scope
    - Method scope
- Minor scopes: Nested scopes in major scopes

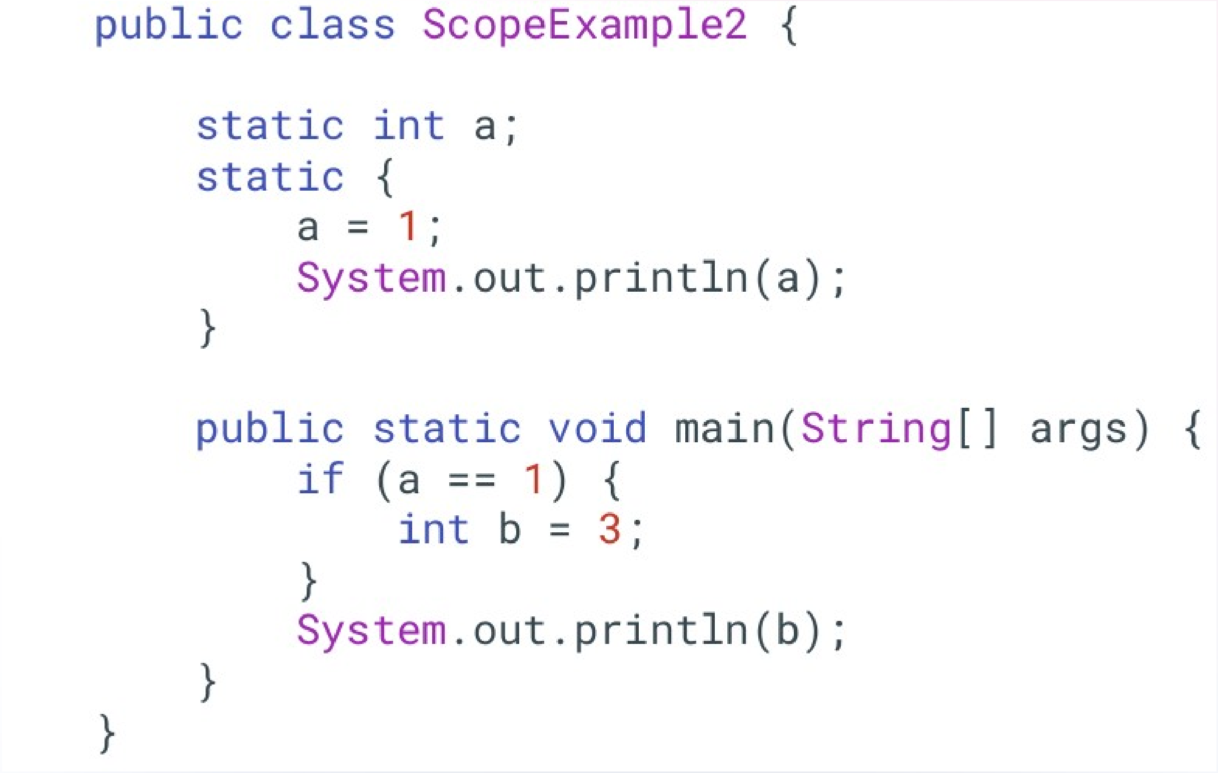


```
public class ScopeExample1 {

    static int a;
    static {
        a = 1;
        System.out.println(a);
    }

    public static void main(String[] args) {
        int a;

        a = 2;
        System.out.println(a);
    }
}
```

sourcemind

Variables Scope

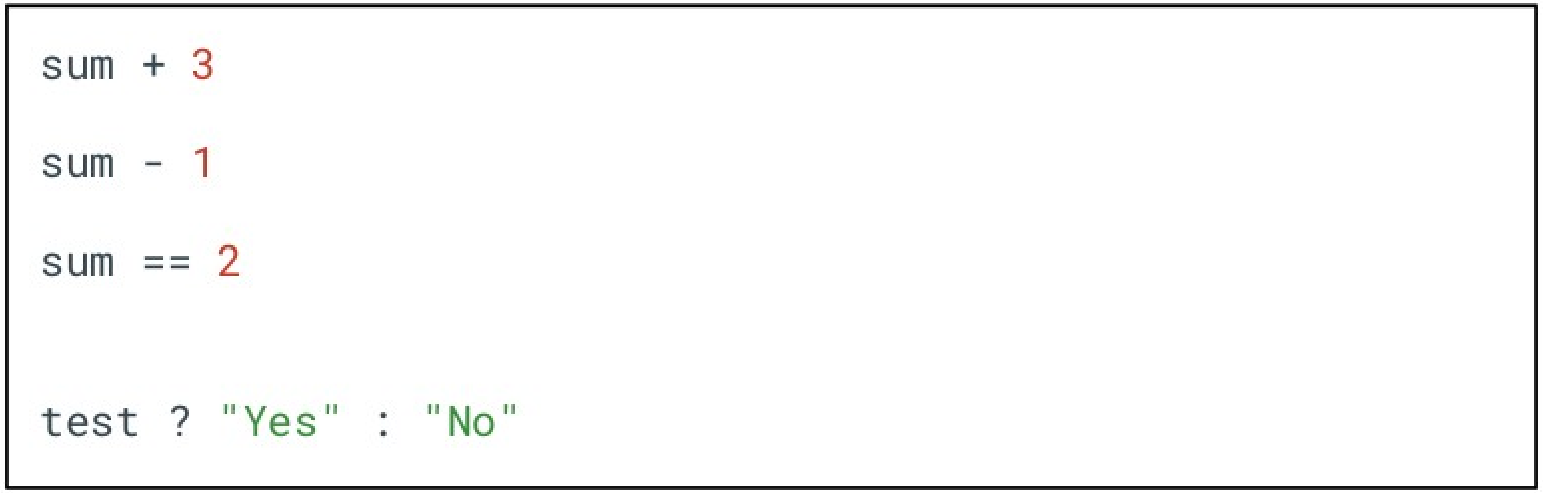
- Variables defined in major scopes
- are visible in their nested scopes.
- The reverse is not correct.



```
public class ScopeExample2 {

    static int a;
    static {
        a = 1;
        System.out.println(a);
    }

    public static void main(String[] args) {
        if (a == 1) {
            int b = 3;
        }
        System.out.println(b);
    }
}
```

Expressions and Operators

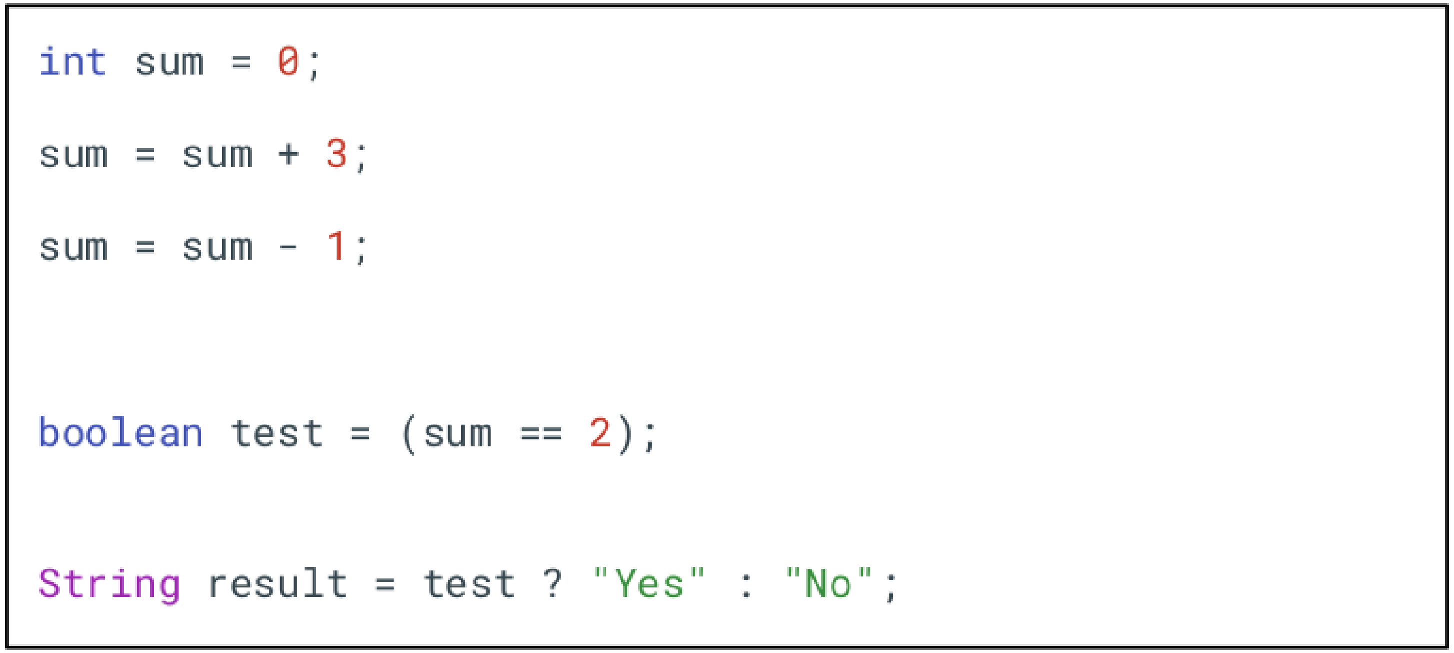
- Operators manipulate the values of variables and literals and produce new values.
- Expressions are composed of operators and literals or variables.



```
sum + 3

sum - 1

sum == 2


test ? "Yes" : "No"
```

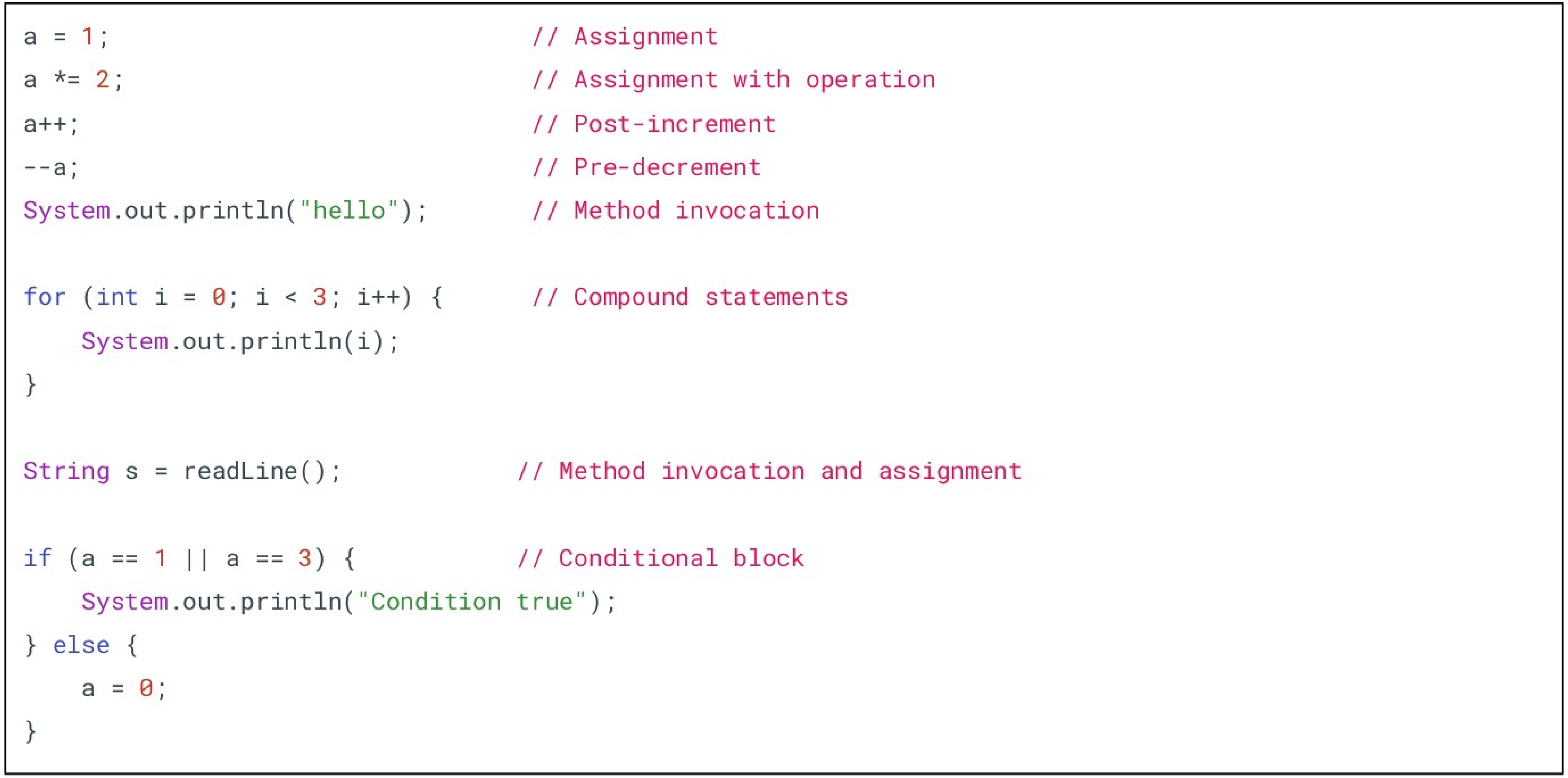
Statements

- A statement is a basic unit of execution in the Java language — it expresses
- a single piece of intent by the programmer.
- A statement changes the state of program or one or a set of variables.



```java
int sum = 0;

sum = sum + 3;

sum = sum - 1;


boolean test = (sum == 2);


String result = test ? "Yes" : "No";
```

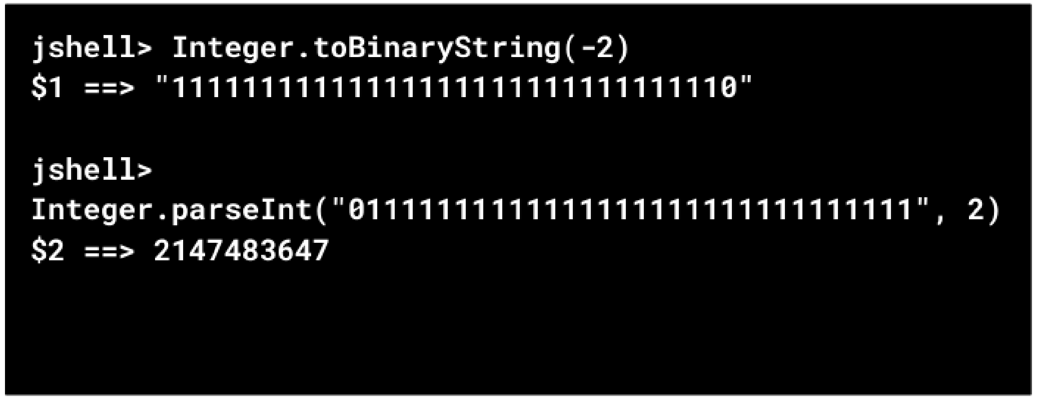
Statements



```
a = 1;                             // Assignment
a *= 2;                            // Assignment with operation
a++;                               // Post-increment
--a;                               // Pre-decrement
System.out.println("hello");       // Method invocation

for (int i = 0; i < 3; i++) {      // Compound statements
    System.out.println(i);
}

String s = readLine();            // Method invocation and assignment

if (a == 1 || a == 3) {           // Conditional block
    System.out.println("Condition true");
} else {
    a = 0;
}
```

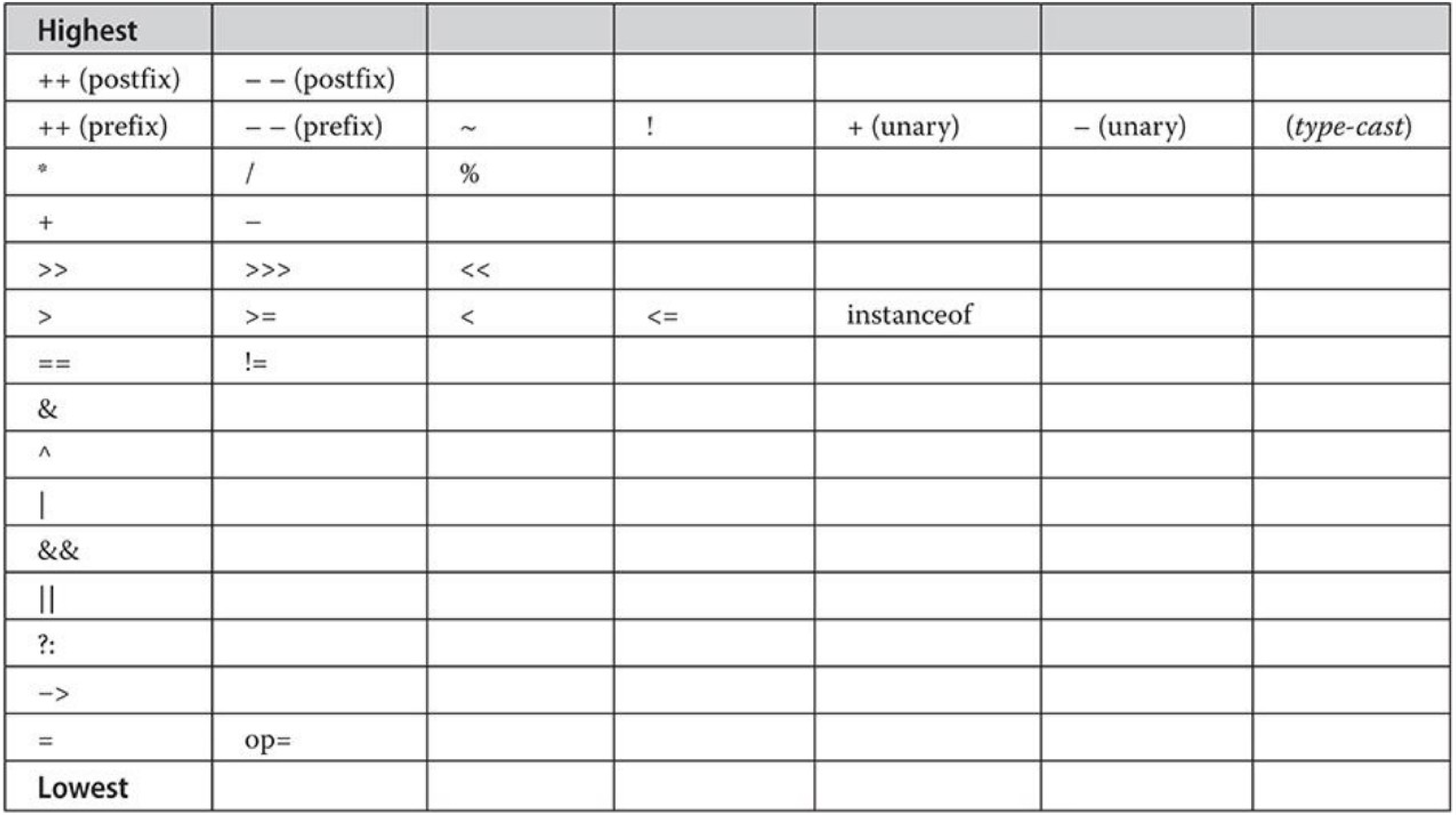
Let’s practice

```
public class Statements {
    public static void main(String[] args) {
        int a = 1;
        int b = a++;
        System.out.println(a);
        System.out.println(b);
        int c = ++a;
        System.out.println(a);
        System.out.println(c);
        System.out.println(1 + 1);
        System.out.println("1" + "1");
        if (-2 >> 1 < -2 >>>1)
            System.out.println("OK");
        if (1 == 2)
            System.out.println("Not");
            System.out.println("OK");
    }
}
```



```
jshell> Integer.toBinaryString(-2)
$1 ==> "11111111111111111111111111111110"

jshell>
Integer.parseInt("01111111111111111111111111111111", 2)
$2 ==> 2147483647
```

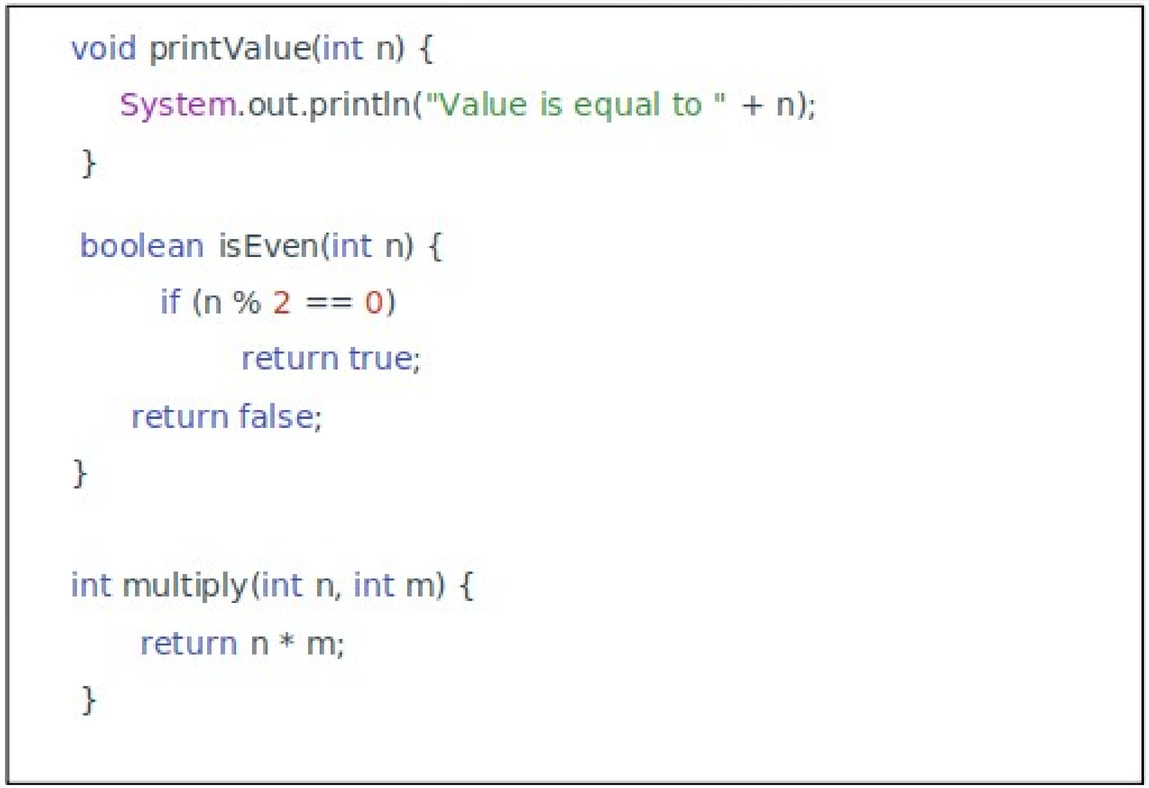
Operator Precedence



Highest						
++ (postfix)	– – (postfix)					
++ (prefix)	– – (prefix)	~	!	+ (unary)	– (unary)	(*type-cast*)
*	/	%				
+	–					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
–>						
=	op=					
Lowest						

Methods

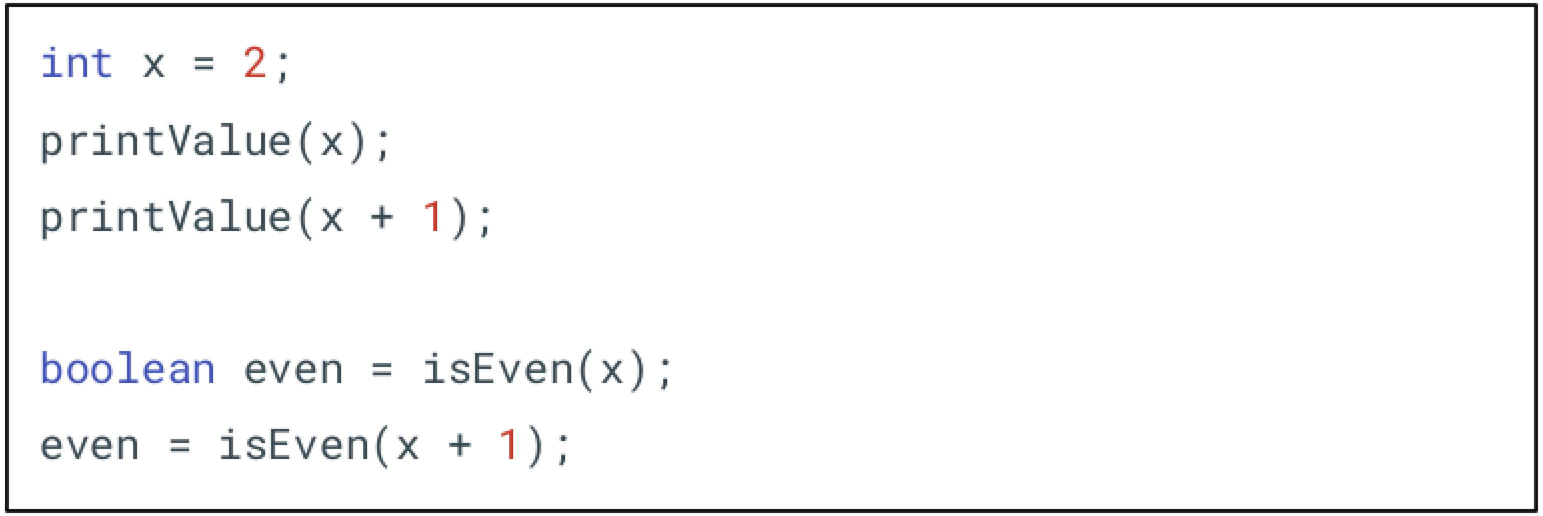
- A method is a named sequence of Java statements that can be invoked by other Java code. When a method is invoked, it is passed zero or more values known as arguments. The method performs some computations and, optionally, returns a value



```
void printValue(int n) {
    System.out.println("Value is equal to " + n);
 }

 boolean isEven(int n) {
        if (n % 2 == 0)
               return true;
     return false;
}


int multiply(int n, int m) {
      return n * m;
 }
```

Methods

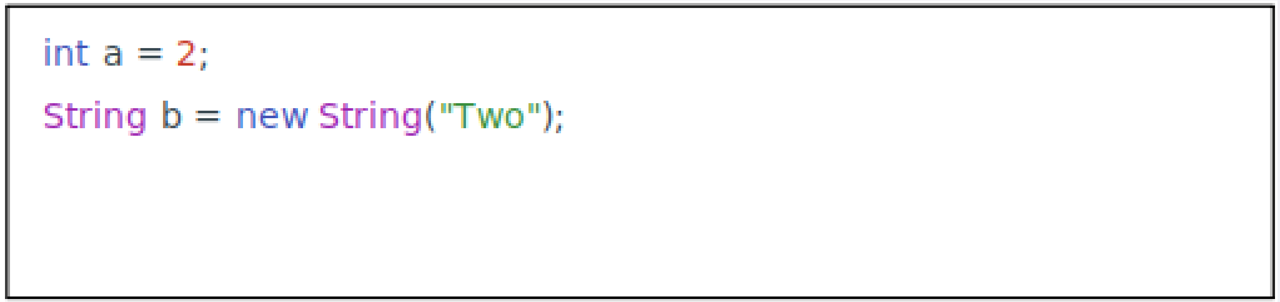
- A defined method can be used in other places of Java program – In order to reuse code.



```java
int x = 2;
printValue(x);
printValue(x + 1);

boolean even = isEven(x);
even = isEven(x + 1);
```

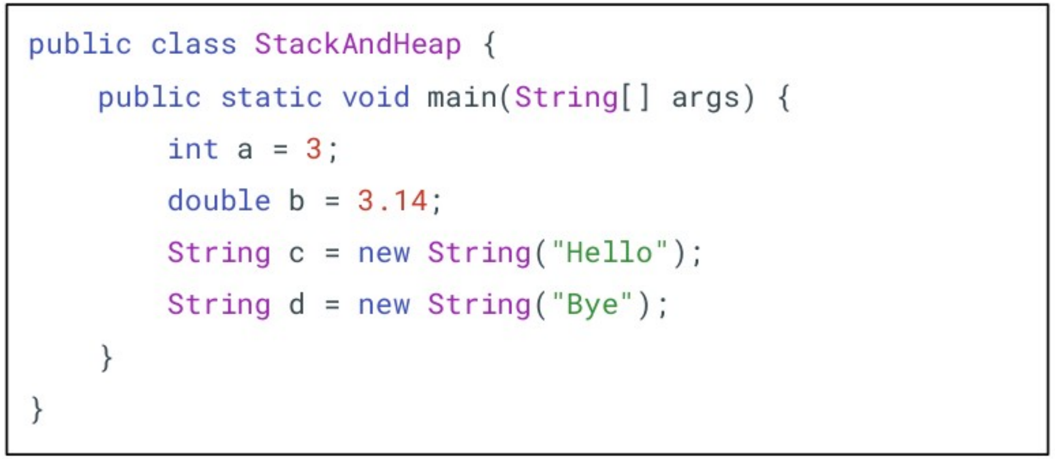
Java Stack and Heap

- Variables in JVM are organized into two sections: Stack and Heap
- Primitive variables and literals are stored in Stack
- Objects are stored in Heap
- However, object “references” are stored in Stack
- Distinguish between the two concepts:
  - Value
  - Reference

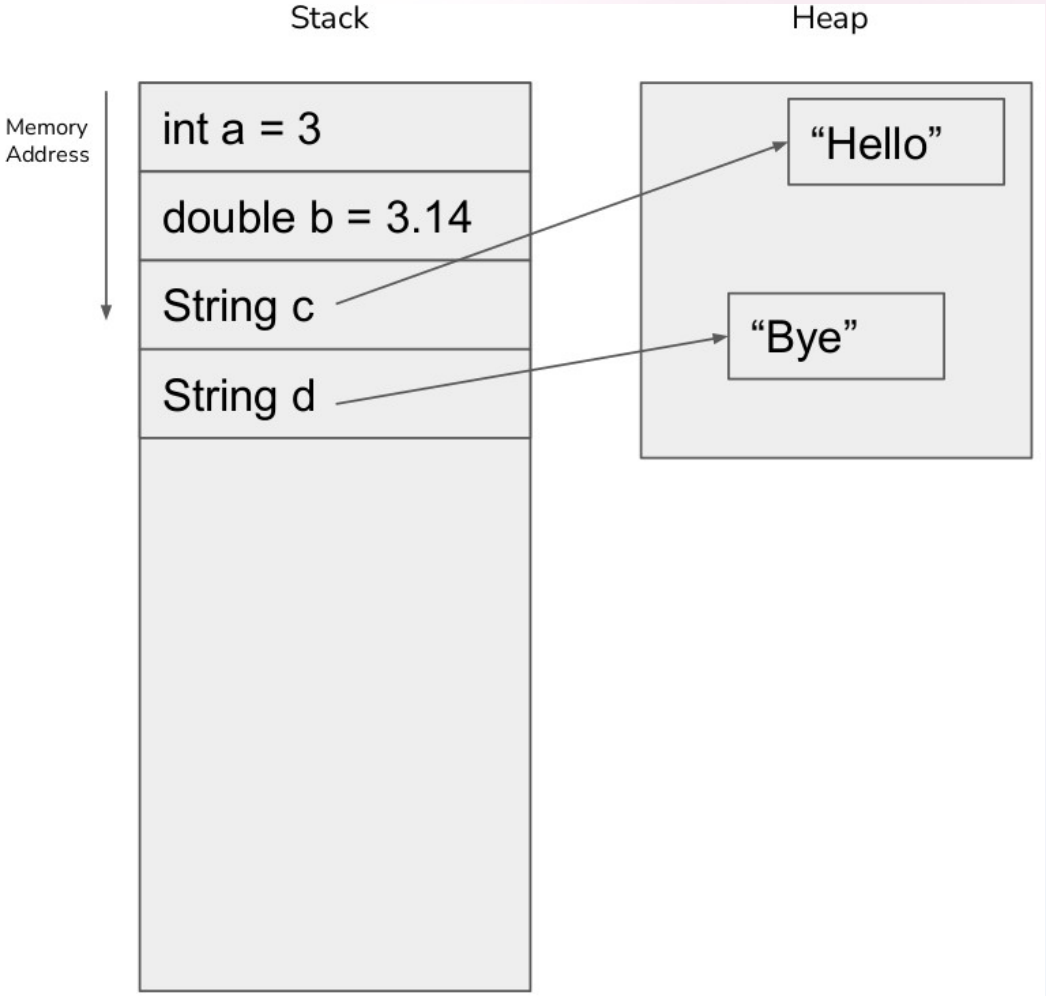


```
int a = 2;
String b = new String("Two");
```

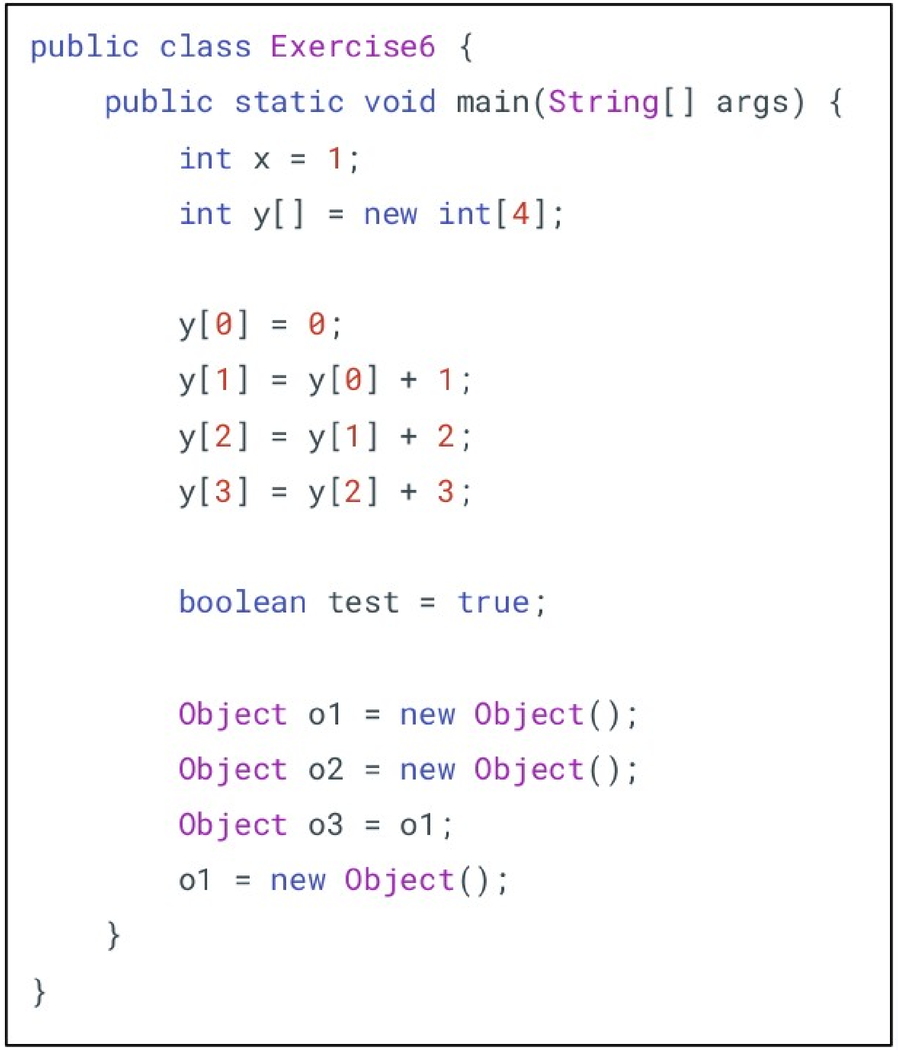
Java Stack and Heap



```java
public class StackAndHeap {
    public static void main(String[] args) {
        int a = 3;
        double b = 3.14;
        String c = new String("Hello");
        String d = new String("Bye");
    }
}
```



Exercise #6



```java
public class Exercise6 {
    public static void main(String[] args) {
        int x = 1;
        int y[] = new int[4];

        y[0] = 0;
        y[1] = y[0] + 1;
        y[2] = y[1] + 2;
        y[3] = y[2] + 3;

        boolean test = true;

        Object o1 = new Object();
        Object o2 = new Object();
        Object o3 = o1;
        o1 = new Object();
    }
}
```

- Use the following tool to investigate what is stored in JVM stack and heap when running the provided source code.

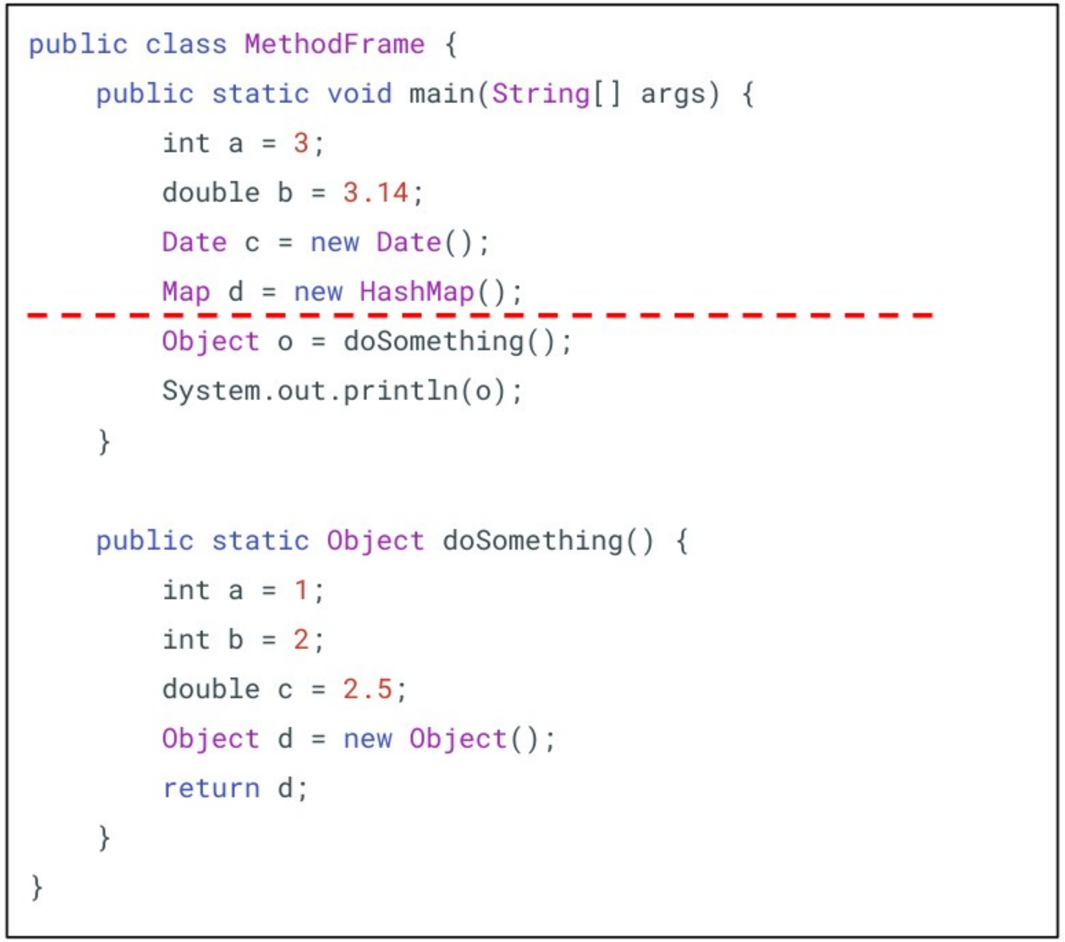
JVM memory visualizer

https://cscircles.cemc.uwaterloo.ca/java_visualize/

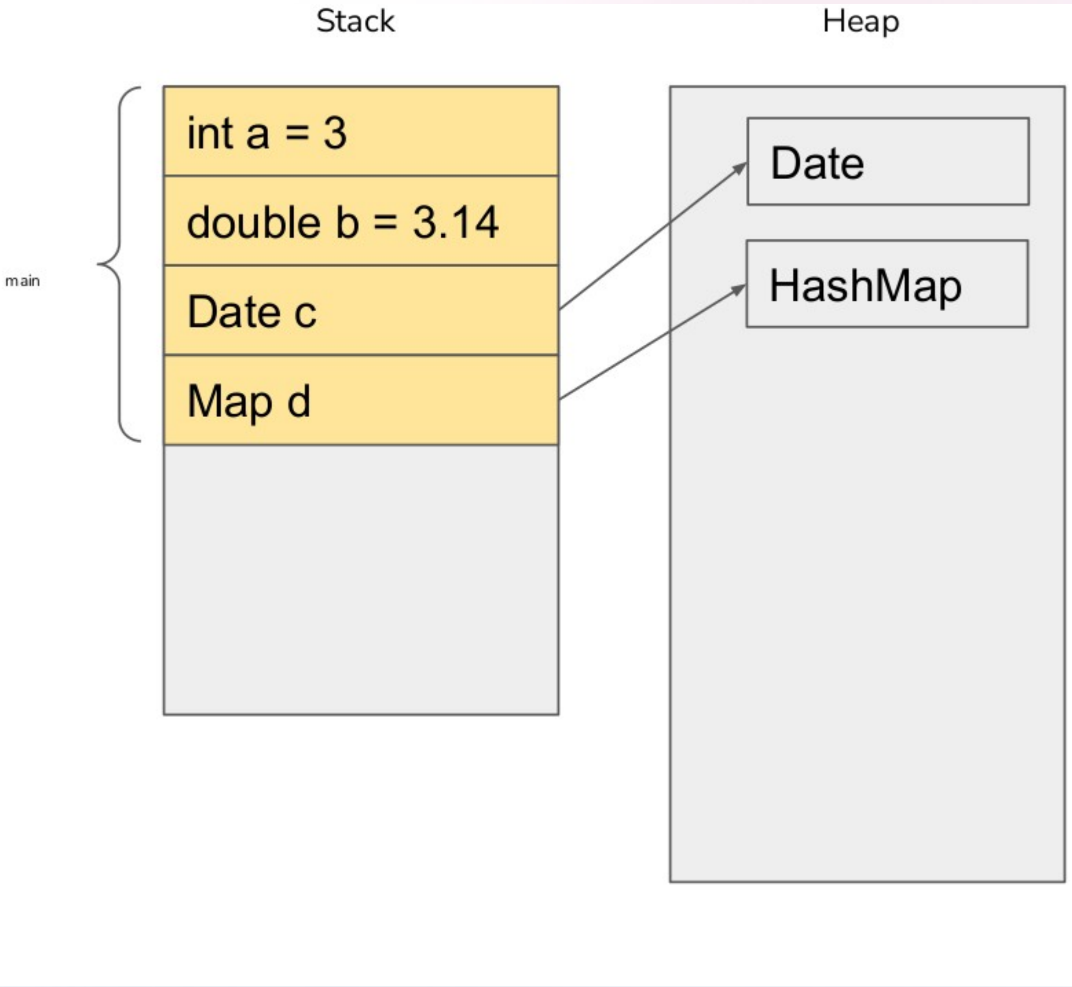
Method Variables in Stack

- Each method execution has its own “stack frame”
- Heap space is available to all methods
- Primitive data types are created in stack
- Objects and arrays are created in heap

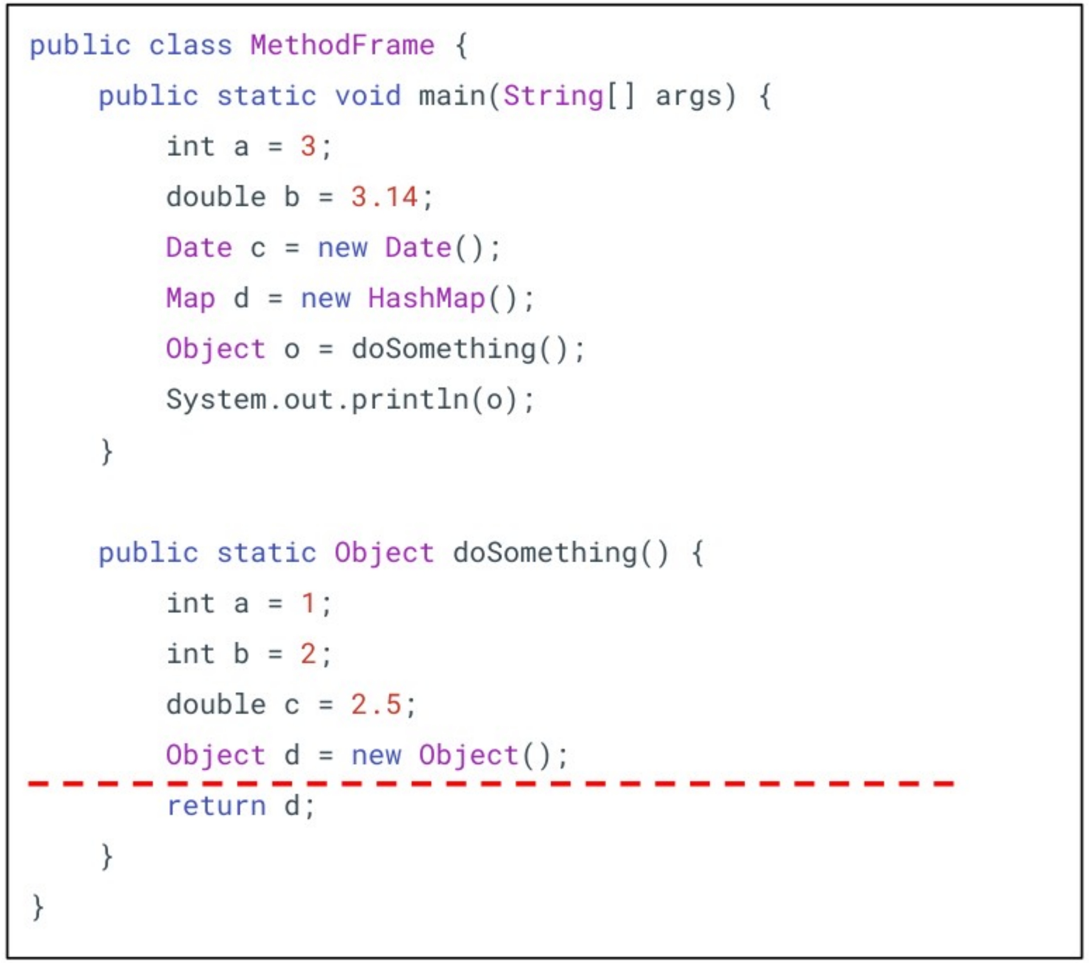
Method Variables in Stack



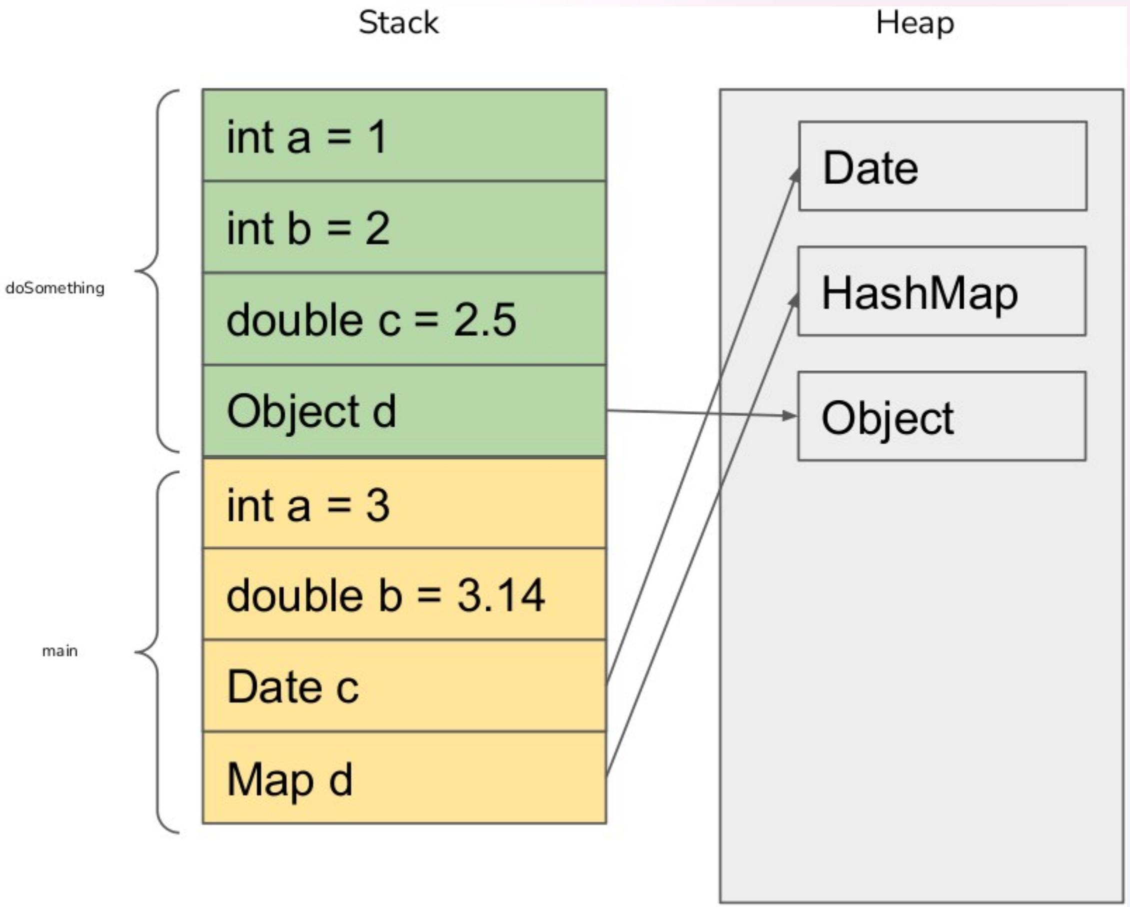
```
public class MethodFrame {
    public static void main(String[] args) {
        int a = 3;
        double b = 3.14;
        Date c = new Date();
        Map d = new HashMap();
        Object o = doSomething();
        System.out.println(o);
    }

    public static Object doSomething() {
        int a = 1;
        int b = 2;
        double c = 2.5;
        Object d = new Object();
        return d;
    }
}
```



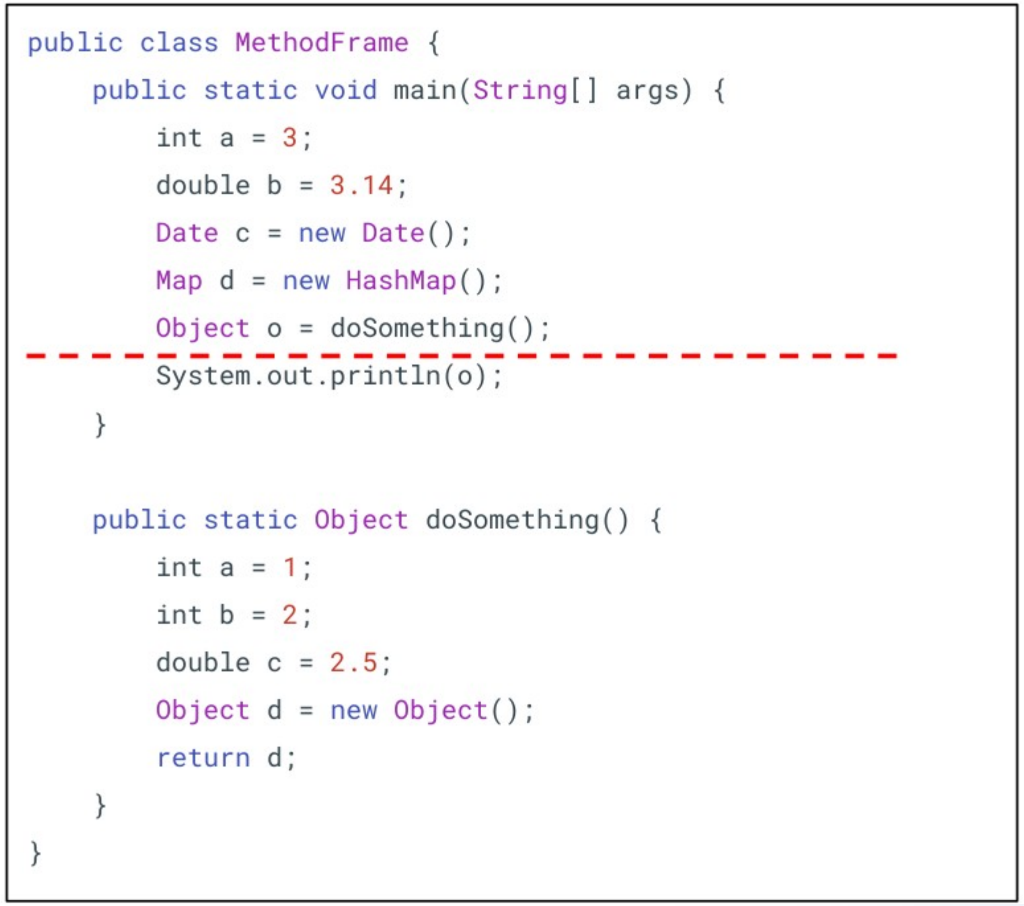
Method Variables in Stack



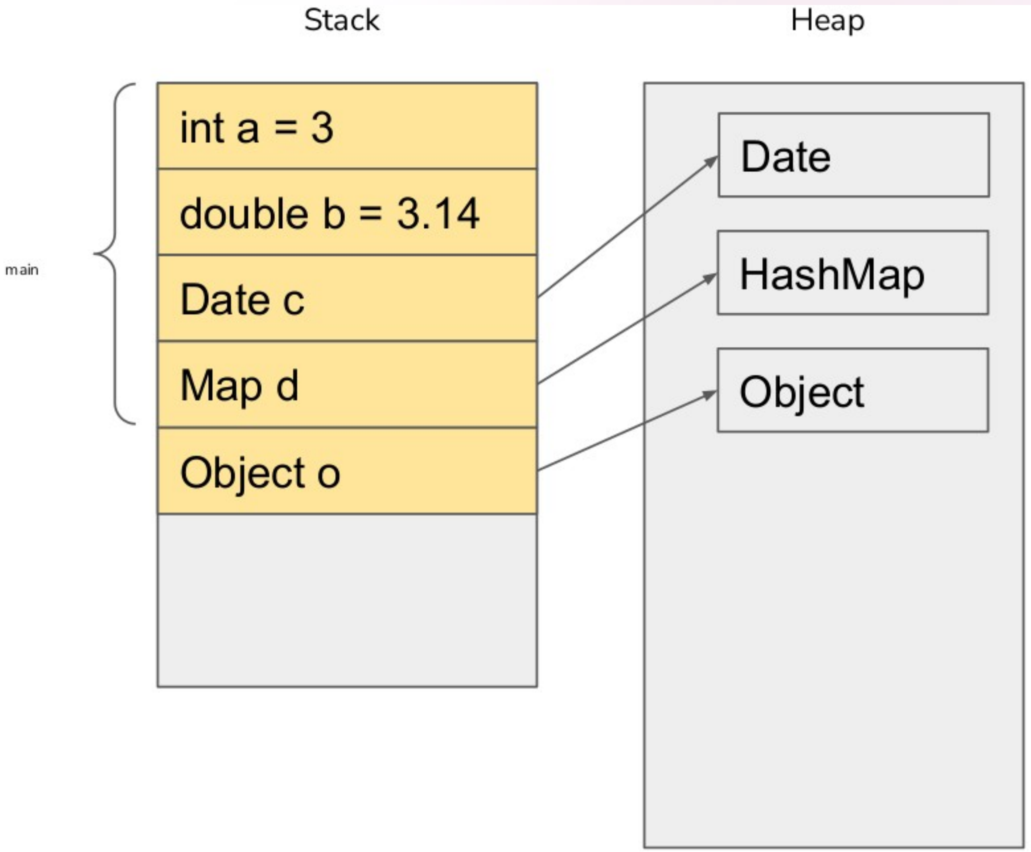
```
public class MethodFrame {
    public static void main(String[] args) {
        int a = 3;
        double b = 3.14;
        Date c = new Date();
        Map d = new HashMap();
        Object o = doSomething();
        System.out.println(o);
    }

    public static Object doSomething() {
        int a = 1;
        int b = 2;
        double c = 2.5;
        Object d = new Object();
        return d;
    }
}
```



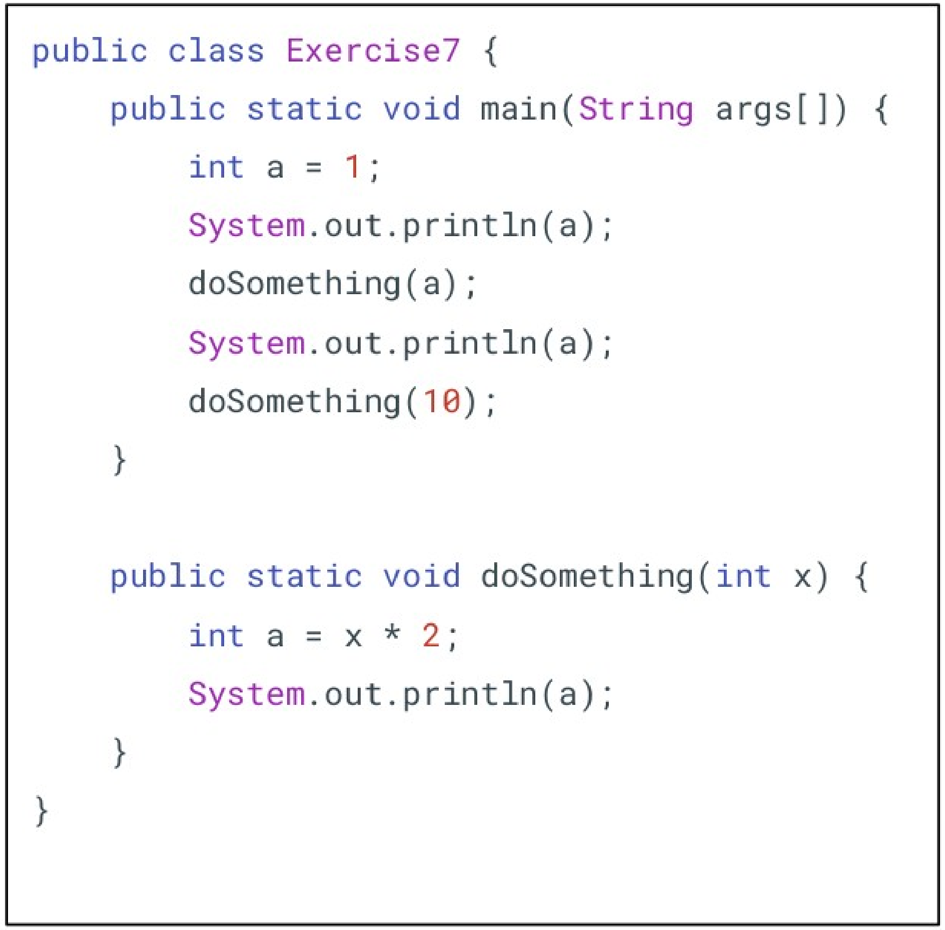
Method Variables in Stack



```
public class MethodFrame {
    public static void main(String[] args) {
        int a = 3;
        double b = 3.14;
        Date c = new Date();
        Map d = new HashMap();
        Object o = doSomething();
        System.out.println(o);
    }

    public static Object doSomething() {
        int a = 1;
        int b = 2;
        double c = 2.5;
        Object d = new Object();
        return d;
    }
}
```



Exercise #7



```java
public class Exercise7 {
    public static void main(String args[]) {
        int a = 1;
        System.out.println(a);
        doSomething(a);
        System.out.println(a);
        doSomething(10);
    }

    public static void doSomething(int x) {
        int a = x * 2;
        System.out.println(a);
    }
}
```

- Use the following tool to investigate what is stored in JVM stack and heap when running the provided source code.

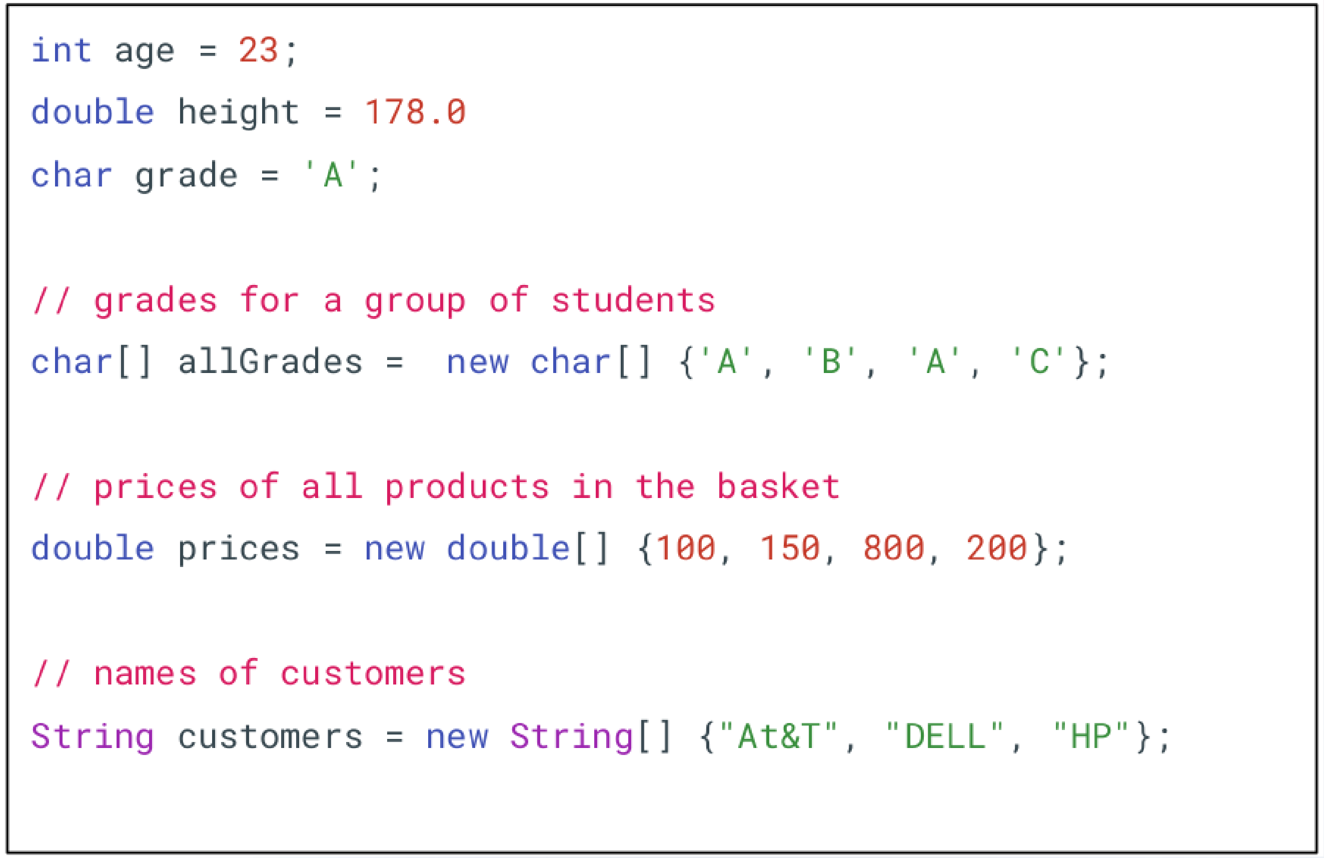
JVM memory visualizer

https://cscircles.cemc.uwaterloo.ca/java_visualize/

sourcemind

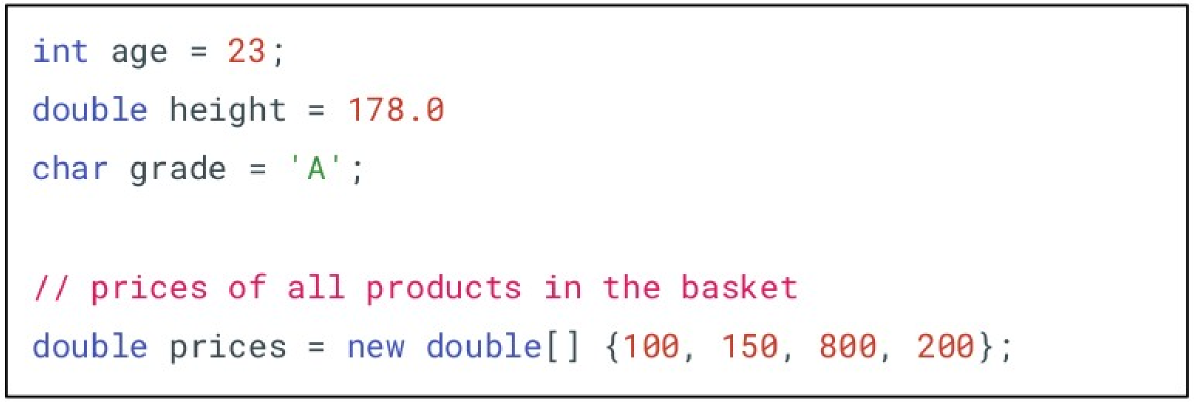
Arrays

- Sometimes we need to work with a group of similar values
- Semantically belonging to the same category
- Same type

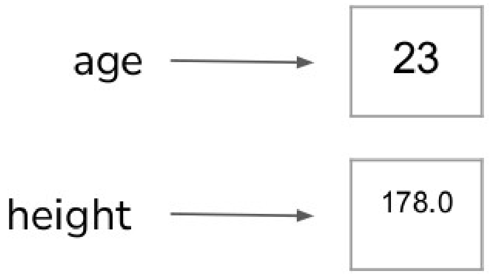


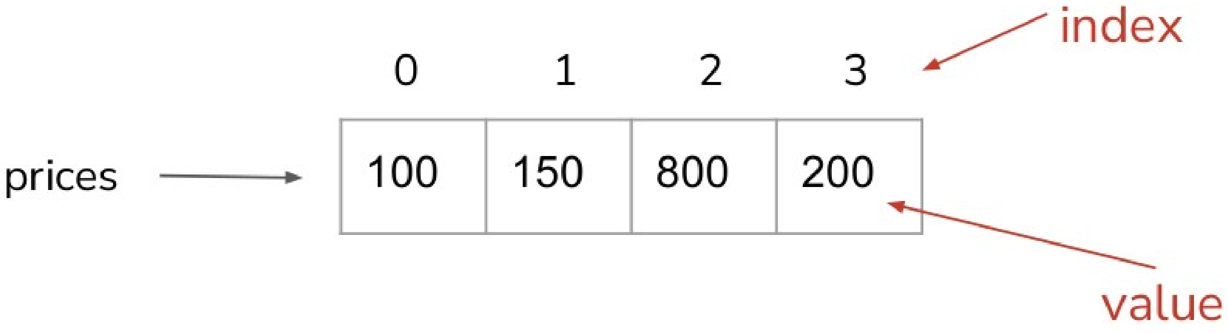
```
int age = 23;
double height = 178.0
char grade = 'A';

// grades for a group of students
char[] allGrades =  new char[] {'A', 'B', 'A', 'C'};

// prices of all products in the basket
double prices = new double[] {100, 150, 800, 200};

// names of customers
String customers = new String[] {"At&T", "DELL", "HP"};
```

Arrays – visualize

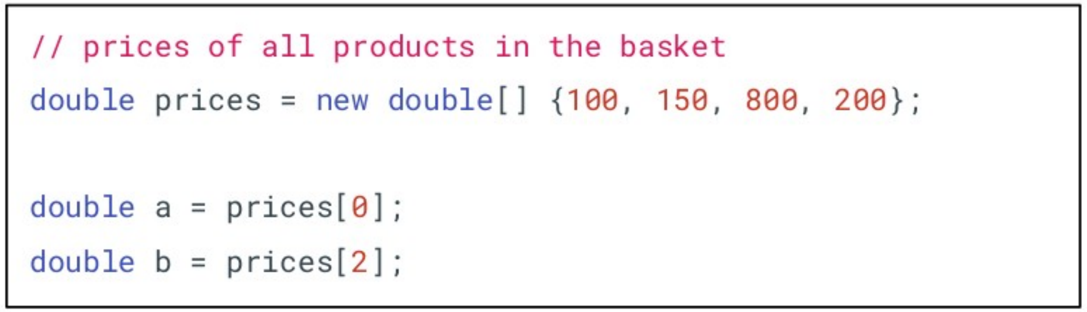


```
int age = 23;
double height = 178.0
char grade = 'A';

// prices of all products in the basket
double prices = new double[] {100, 150, 800, 200};
```

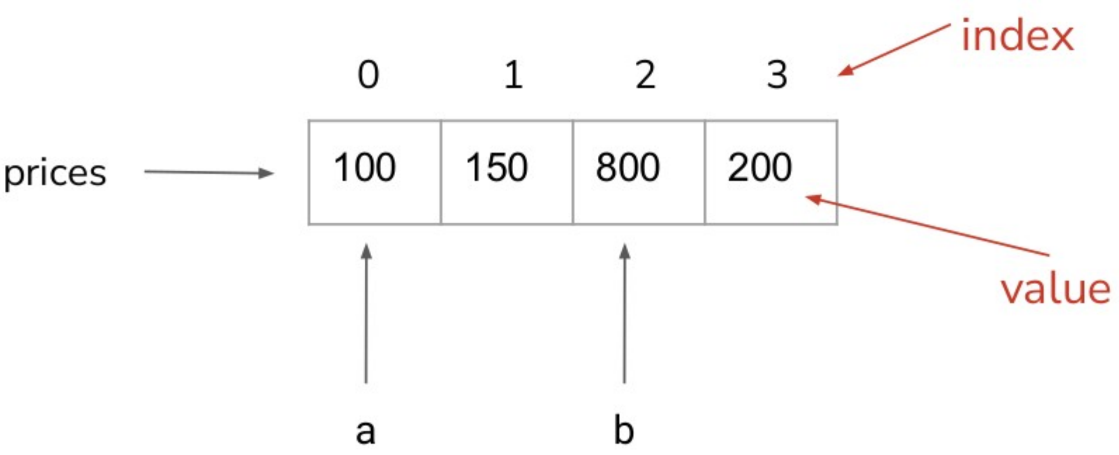




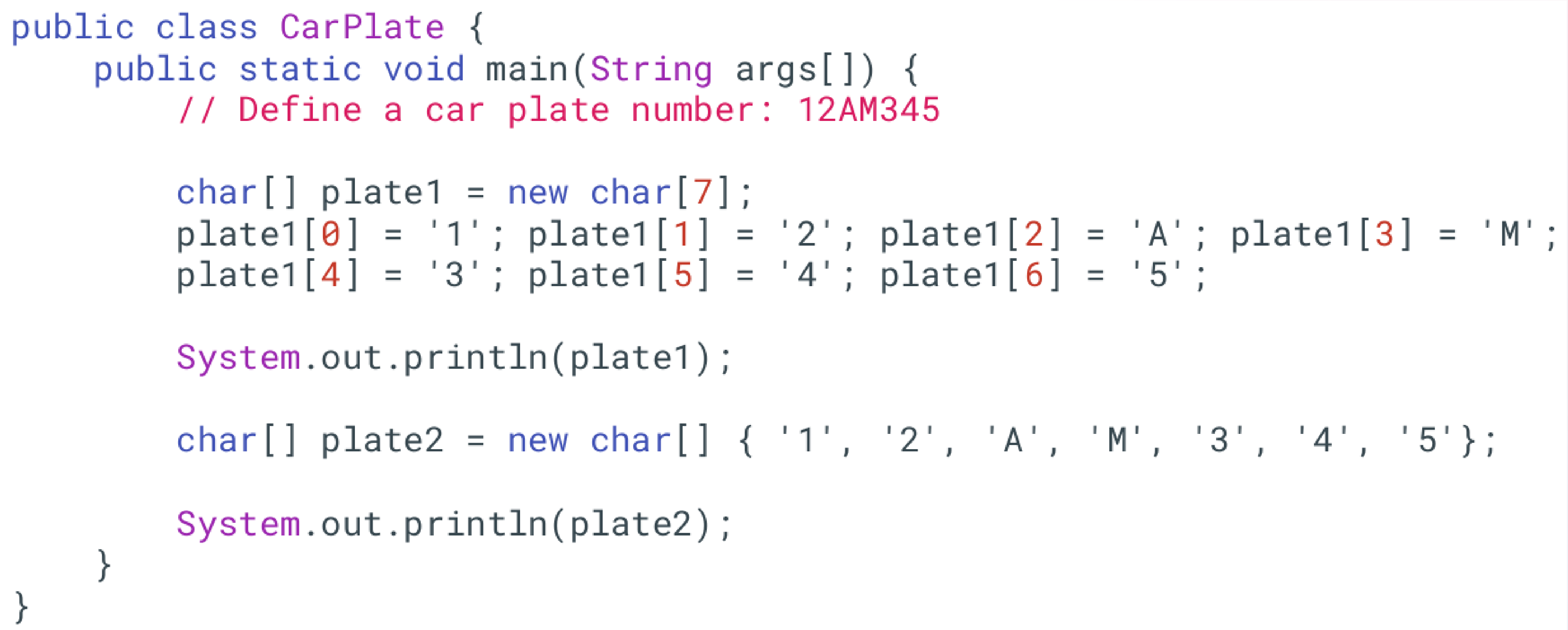
Arrays – indexes



```
// prices of all products in the basket
double prices = new double[] {100, 150, 800, 200};

double a = prices[0];
double b = prices[2];
```

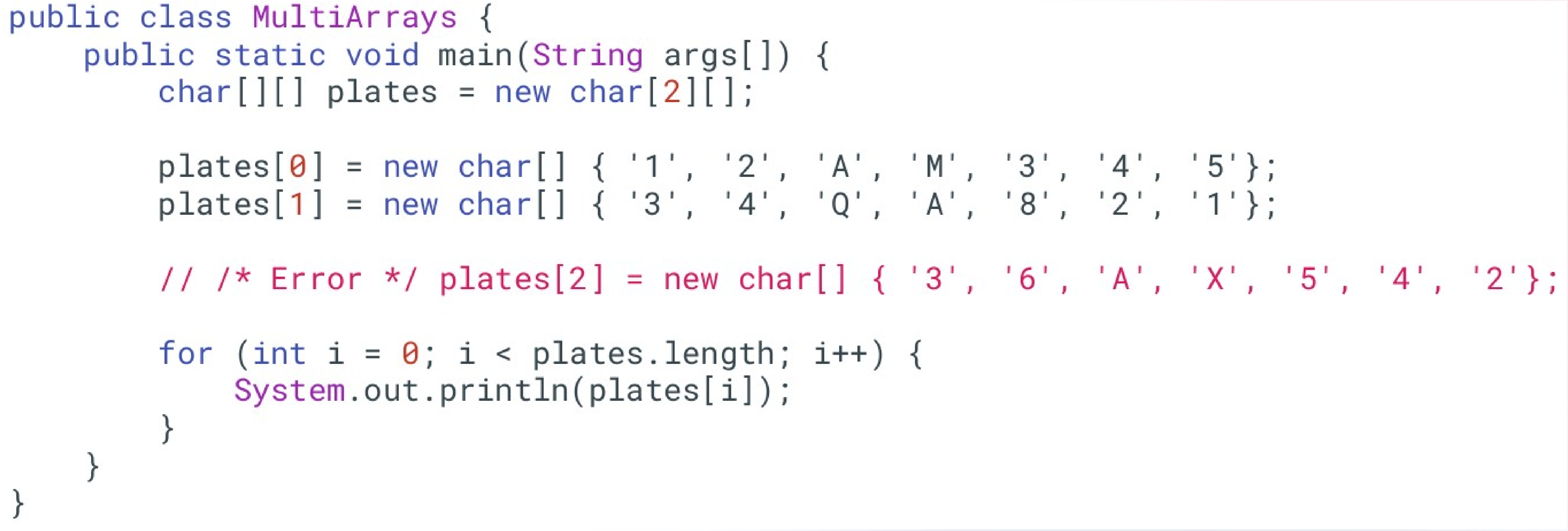


Example: Car Plates



```java
public class CarPlate {
    public static void main(String args[]) {
        // Define a car plate number: 12AM345

        char[] plate1 = new char[7];
        plate1[0] = '1'; plate1[1] = '2'; plate1[2] = 'A'; plate1[3] = 'M';
        plate1[4] = '3'; plate1[5] = '4'; plate1[6] = '5';

        System.out.println(plate1);

        char[] plate2 = new char[] { '1', '2', 'A', 'M', '3', '4', '5'};

        System.out.println(plate2);
    }
}
```

Multidimensional arrays



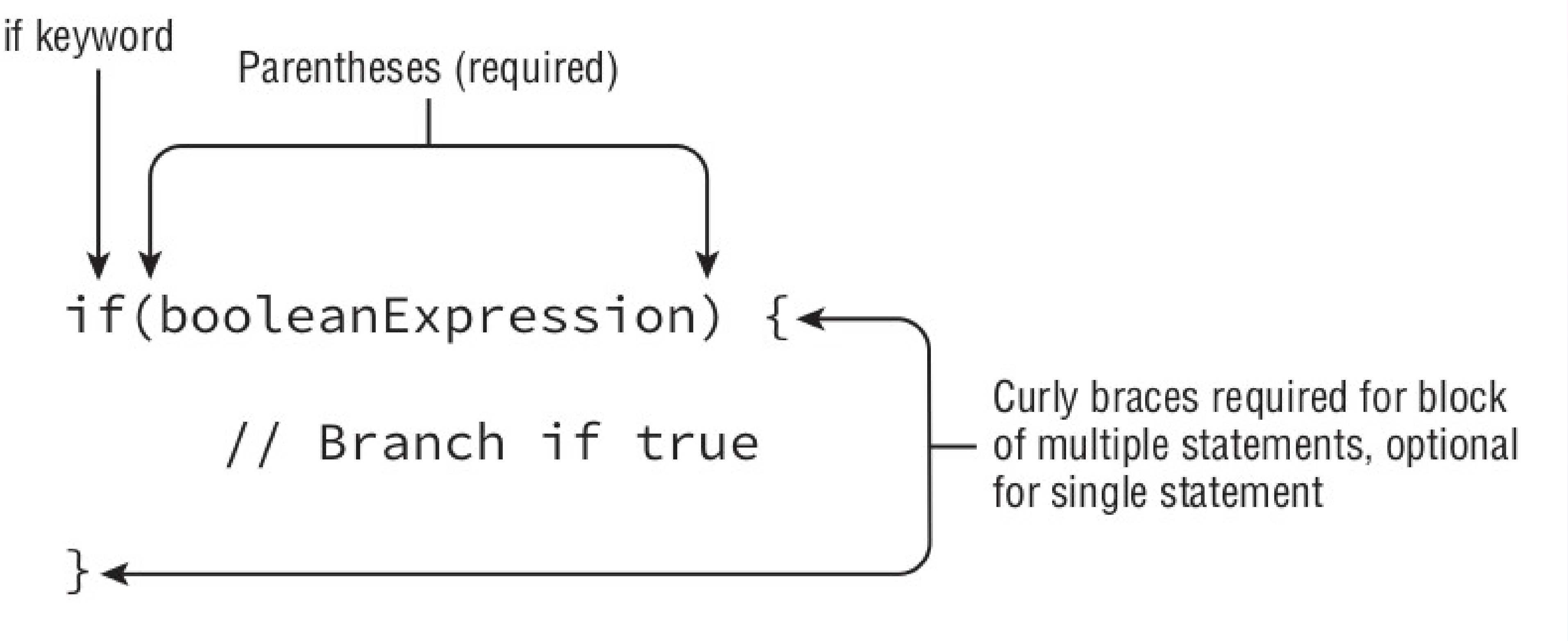
```
public class MultiArrays {
    public static void main(String args[]) {
        char[][] plates = new char[2][];

        plates[0] = new char[] { '1', '2', 'A', 'M', '3', '4', '5'};
        plates[1] = new char[] { '3', '4', 'Q', 'A', '8', '2', '1'};

        // /* Error */ plates[2] = new char[] { '3', '6', 'A', 'X', '5', '4', '2'};

        for (int i = 0; i < plates.length; i++) {
            System.out.println(plates[i]);
        }
    }
}
```

Homework 2

- Modify the ManyCarPlates program in order to ensure that there are no duplicate car plates in the final list.

Control structures

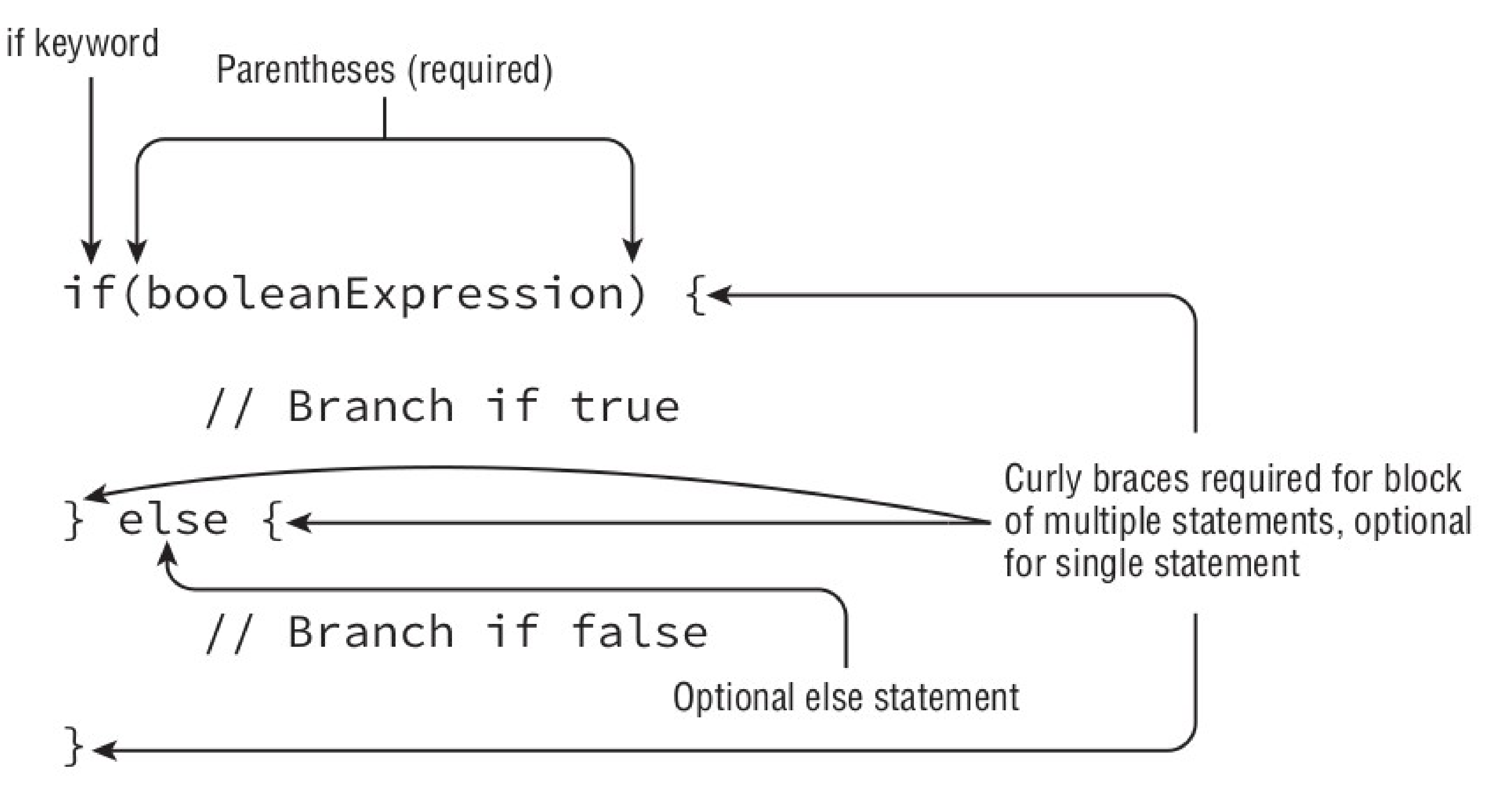
The if-then Statement



```
if(booleanExpression) {

    // Branch if true

}
```

if keyword
Parentheses (required)
Curly braces required for block of multiple statements, optional for single statement

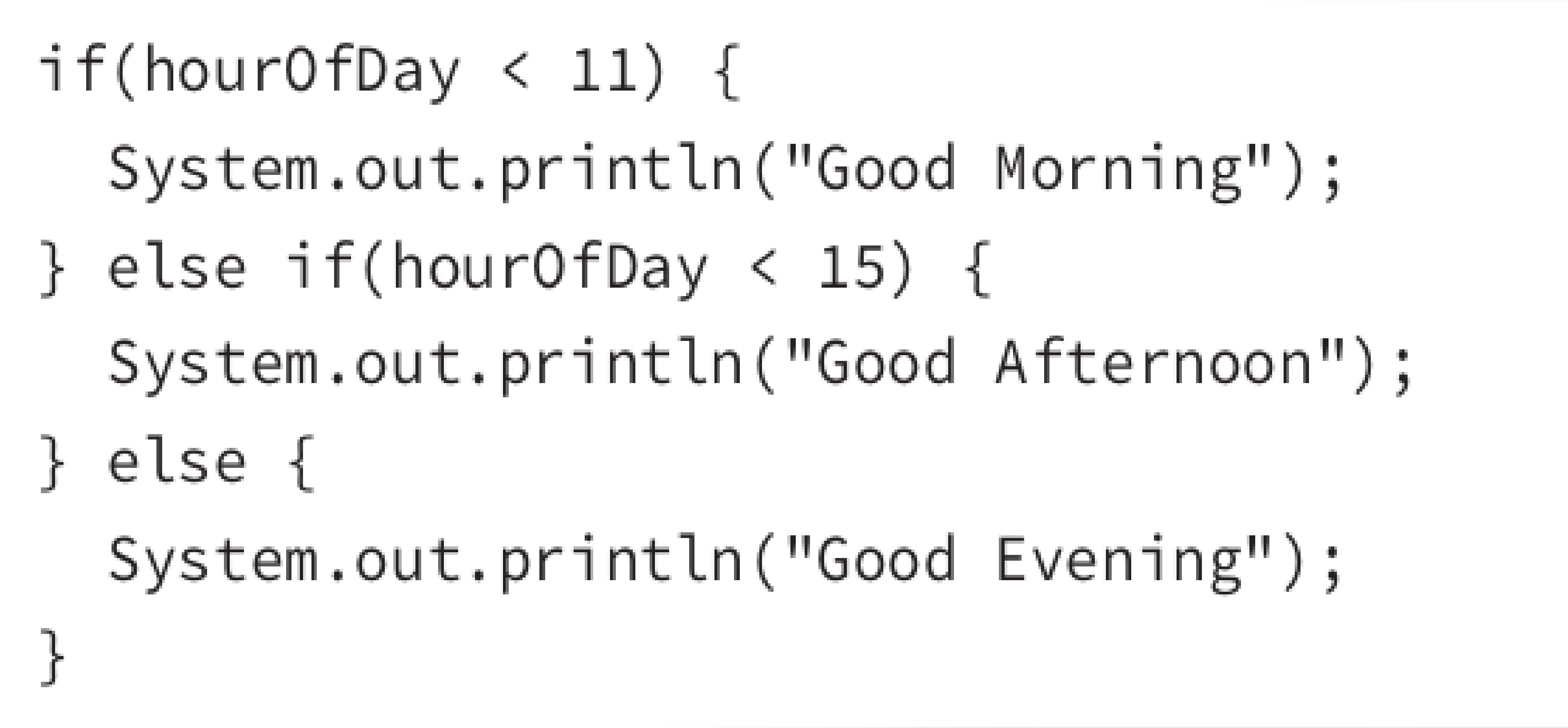
The if-then-else Statement



```
if(booleanExpression) {
      // Branch if true
} else {
      // Branch if false
}
```

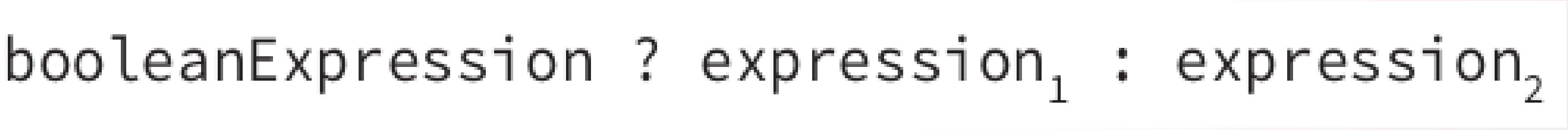
if keyword
Parentheses (required)
Curly braces required for block of multiple statements, optional for single statement
Optional else statement

The if-then-else Statement

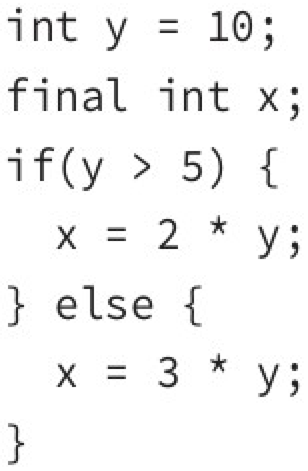


```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
} else if(hourOfDay < 15) {
  System.out.println("Good Afternoon");
} else {
  System.out.println("Good Evening");
}
```

Ternary Operator

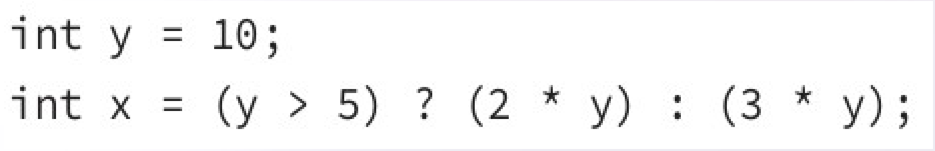


booleanExpression ? $expression_1$: $expression_2$



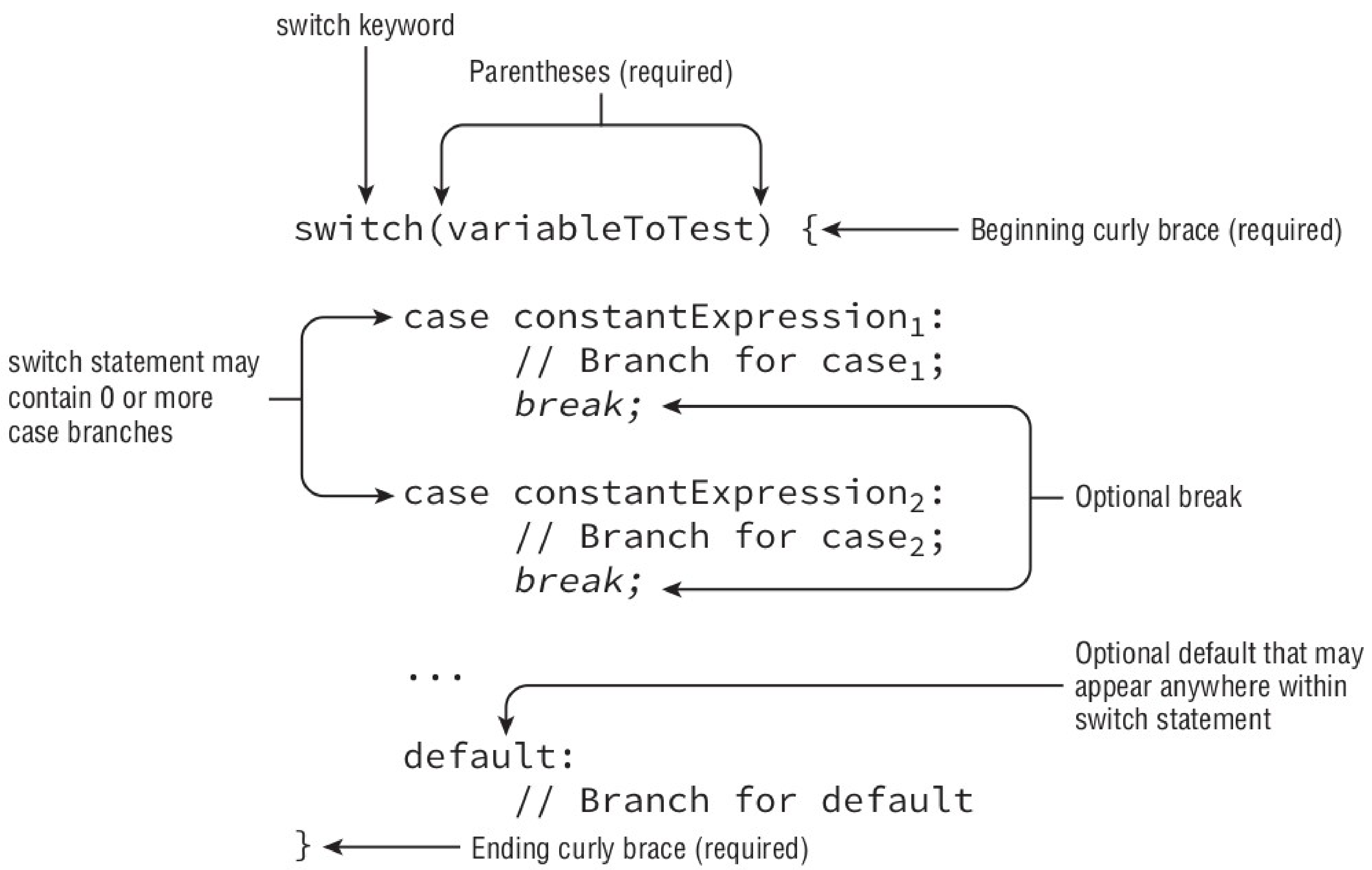
```java
int y = 10;
final int x;
if(y > 5) {
  x = 2 * y;
} else {
  x = 3 * y;
}
```

=>



```java
int y = 10;
int x = (y > 5) ? (2 * y) : (3 * y);
```

The switch Statement



```
switch(variableToTest) {

    case constantExpression1:
        // Branch for case1;
        break;

    case constantExpression2:
        // Branch for case2;
        break;

    ...

    default:
        // Branch for default
}
```

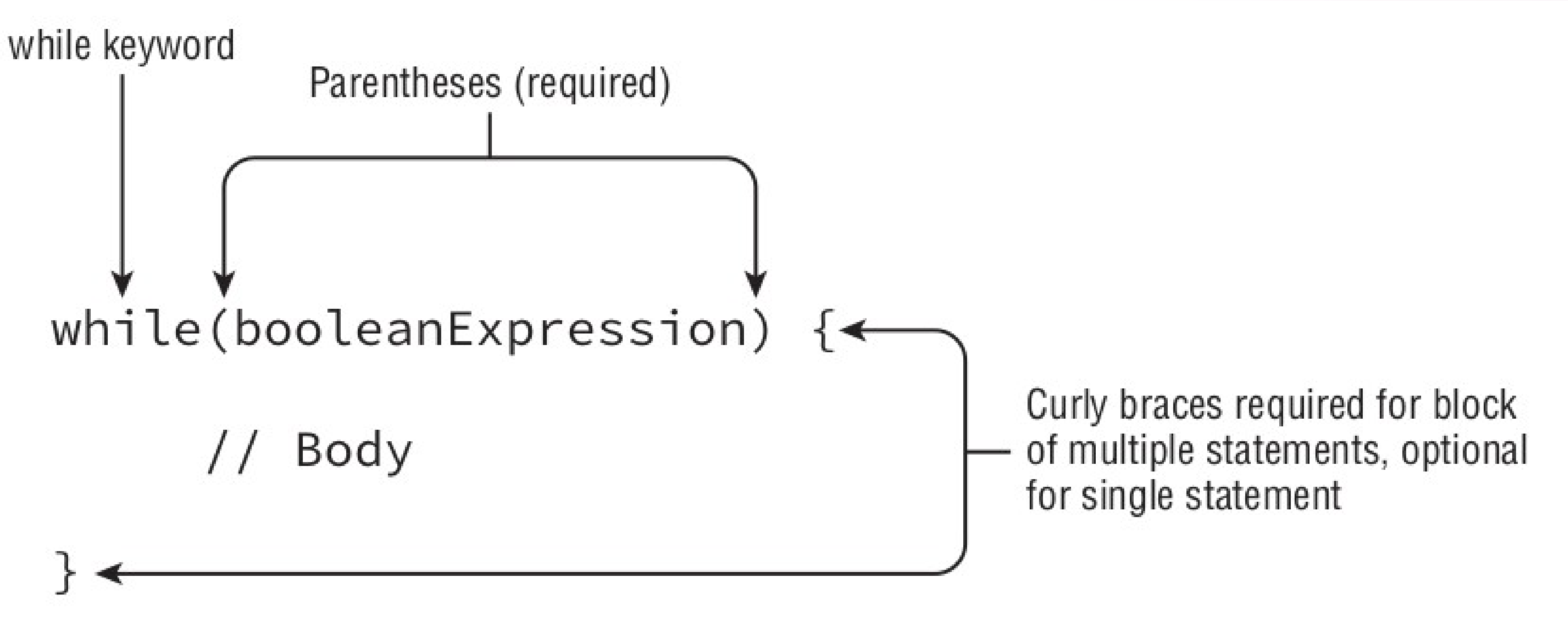
switch keyword
Parentheses (required)
Beginning curly brace (required)
switch statement may contain 0 or more case branches
Optional break
Optional default that may appear anywhere within switch statement
Ending curly brace (required)

The switch Statement

Data types supported by switch statements include the following:

- int and Integer
- byte and Byte
- short and Short
- char and Character
- int and Integer
- String
- enum values
- Note that boolean and long , and their associated wrapper classes, are not supported by switch statements.

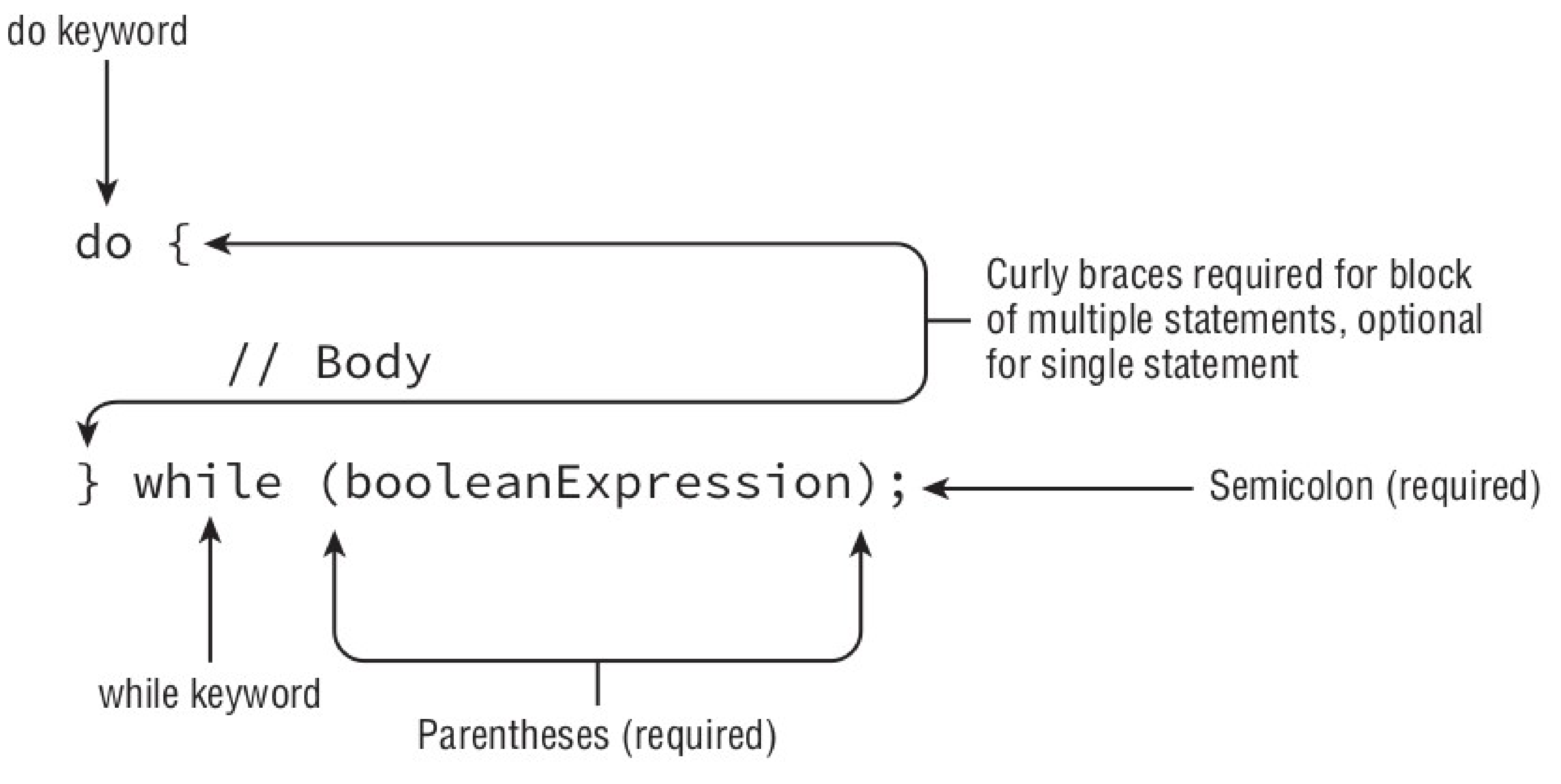
The while Statement



```
while(booleanExpression) {

      // Body

}
```

while keyword
Parentheses (required)
Curly braces required for block of multiple statements, optional for single statement

The do-while Statement



do keyword

```
do {

    // Body

} while (booleanExpression);
```

Curly braces required for block of multiple statements, optional for single statement

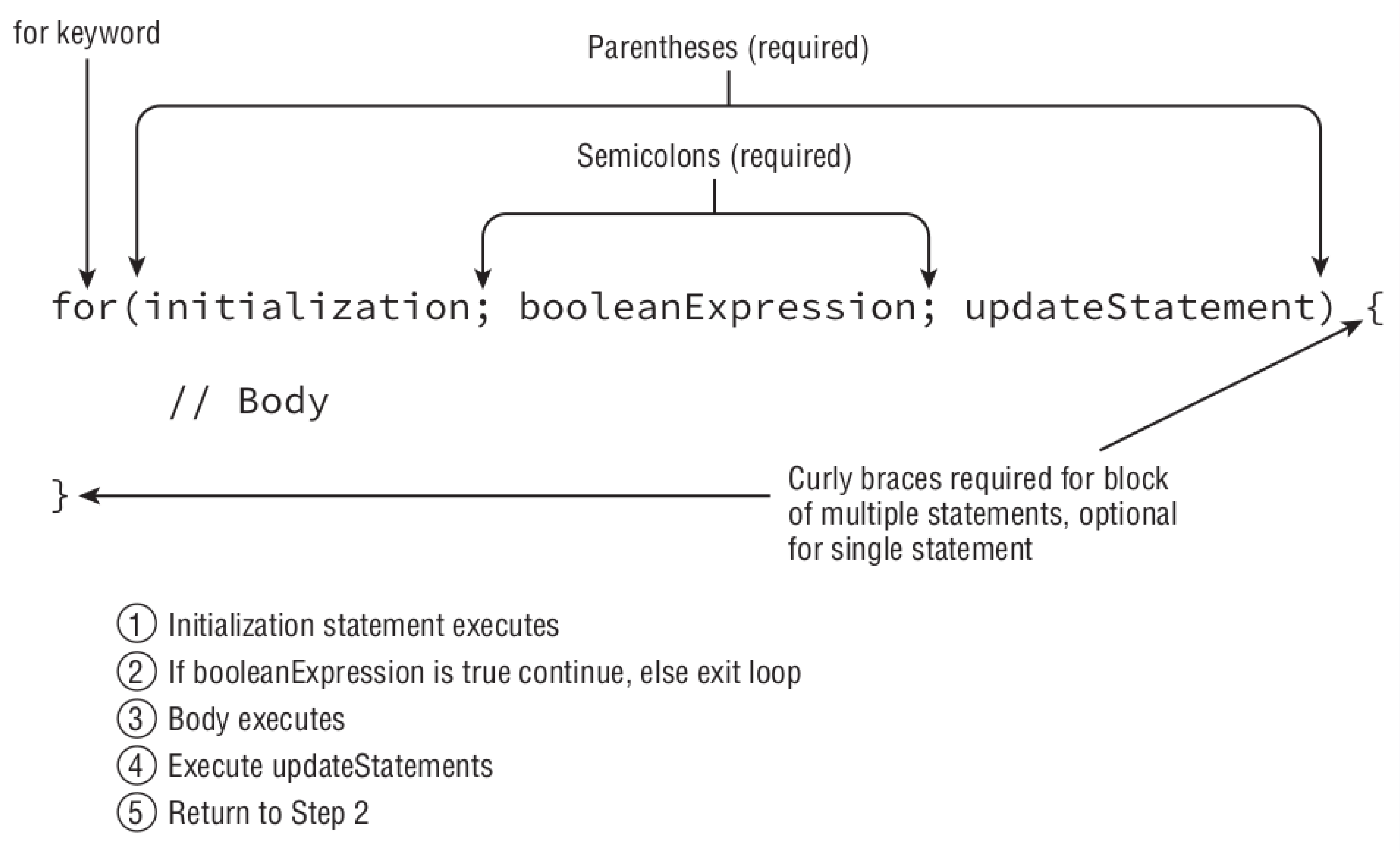
Semicolon (required)

while keyword

Parentheses (required)

sourcemind

The for Statement

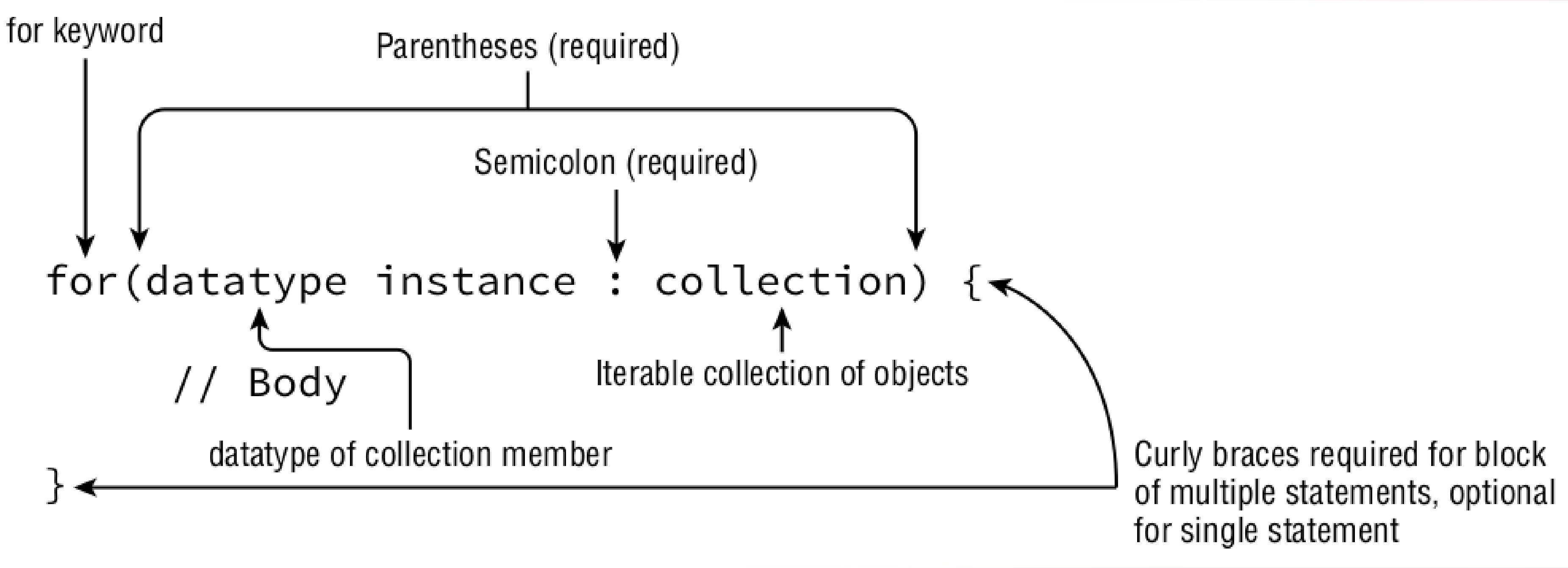


for keyword
Parentheses (required)
Semicolons (required)
Curly braces required for block of multiple statements, optional for single statement

```
for(initialization; booleanExpression; updateStatement) {

    // Body

}
```

① Initialization statement executes
② If booleanExpression is true continue, else exit loop
③ Body executes
④ Execute updateStatements
⑤ Return to Step 2

The for-each Statement

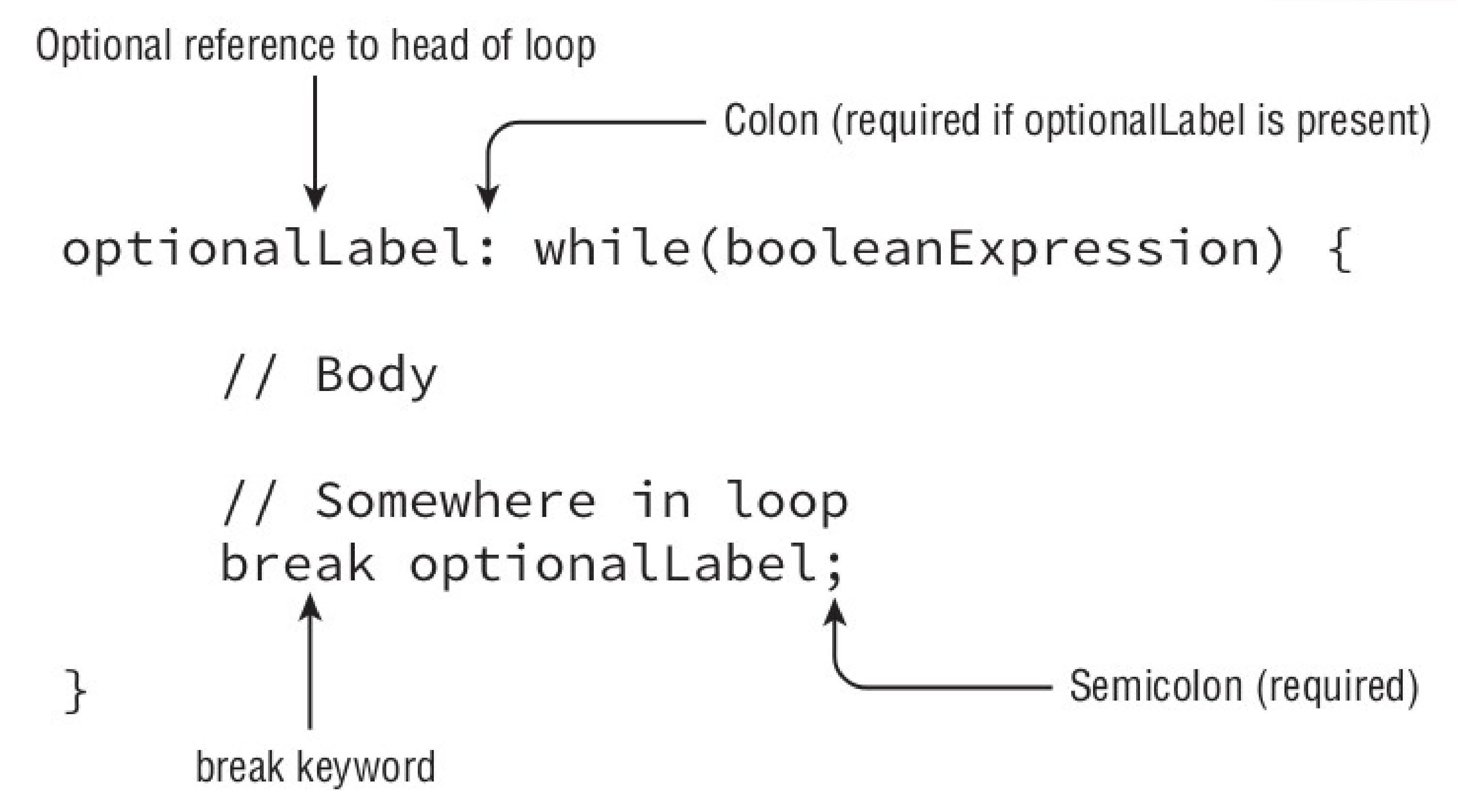


```
for(datatype instance : collection) {
    // Body
}
```

for keyword
Parentheses (required)
Semicolon (required)
Iterable collection of objects
datatype of collection member
Curly braces required for block of multiple statements, optional for single statement

The break Statement

- A break statement transfers the flow of control out to the enclosing statement

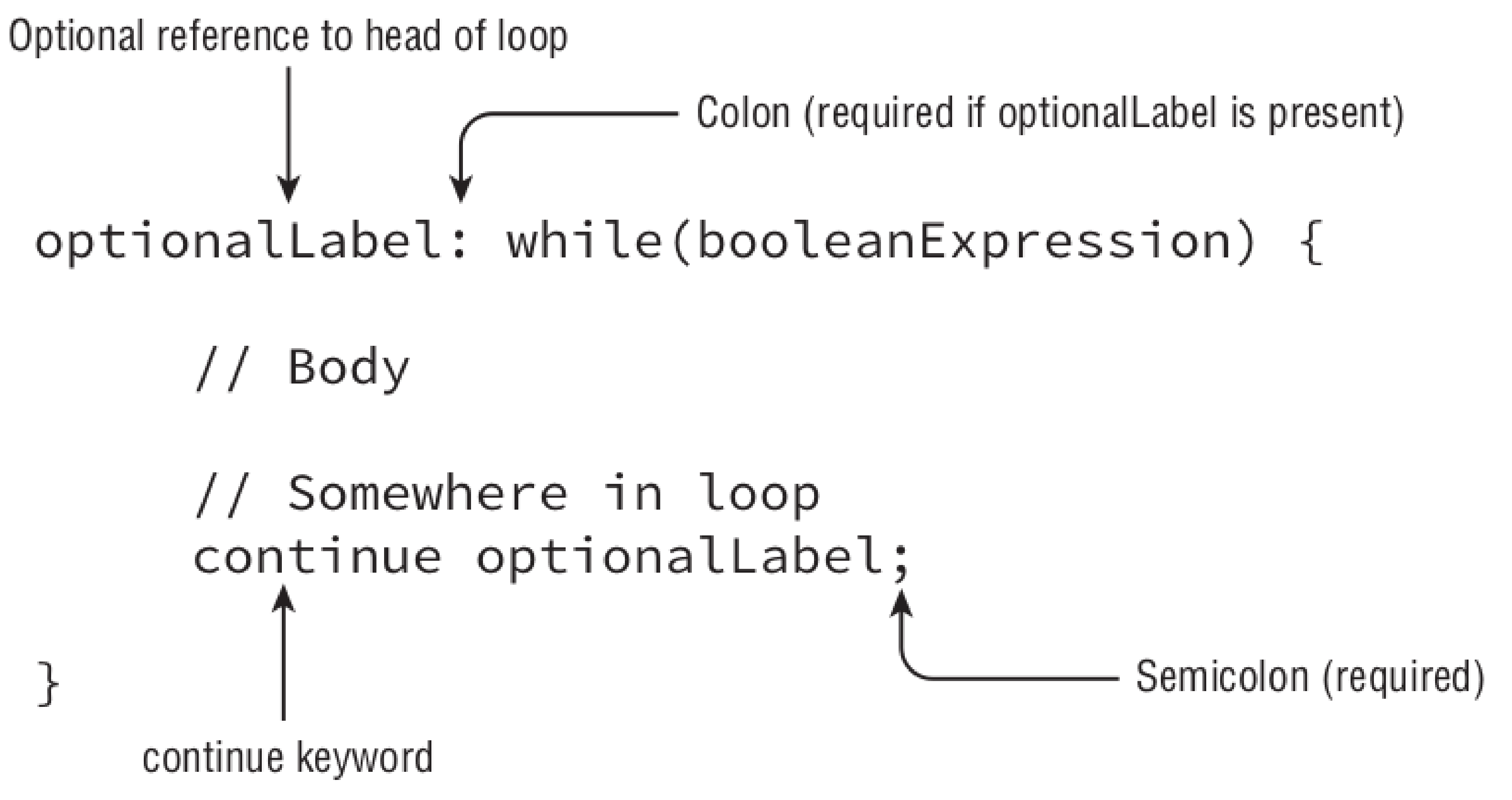


```
optionalLabel: while(booleanExpression) {

     // Body

     // Somewhere in loop
     break optionalLabel;

}
```

Optional reference to head of loop
Colon (required if optionalLabel is present)
Semicolon (required)
break keyword

The continue Statement

- A statement that causes flow to finish the execution of the current loop



Optional reference to head of loop

Colon (required if optionalLabel is present)

```
optionalLabel: while(booleanExpression) {

     // Body

     // Somewhere in loop
     continue optionalLabel;

}
```

continue keyword

Semicolon (required)

End of session 1

sourcemind