Recapitulate CSE160: Java basics, types, statements, arrays and methods

CSE260, Computer Science B: Honors

Stony Brook University

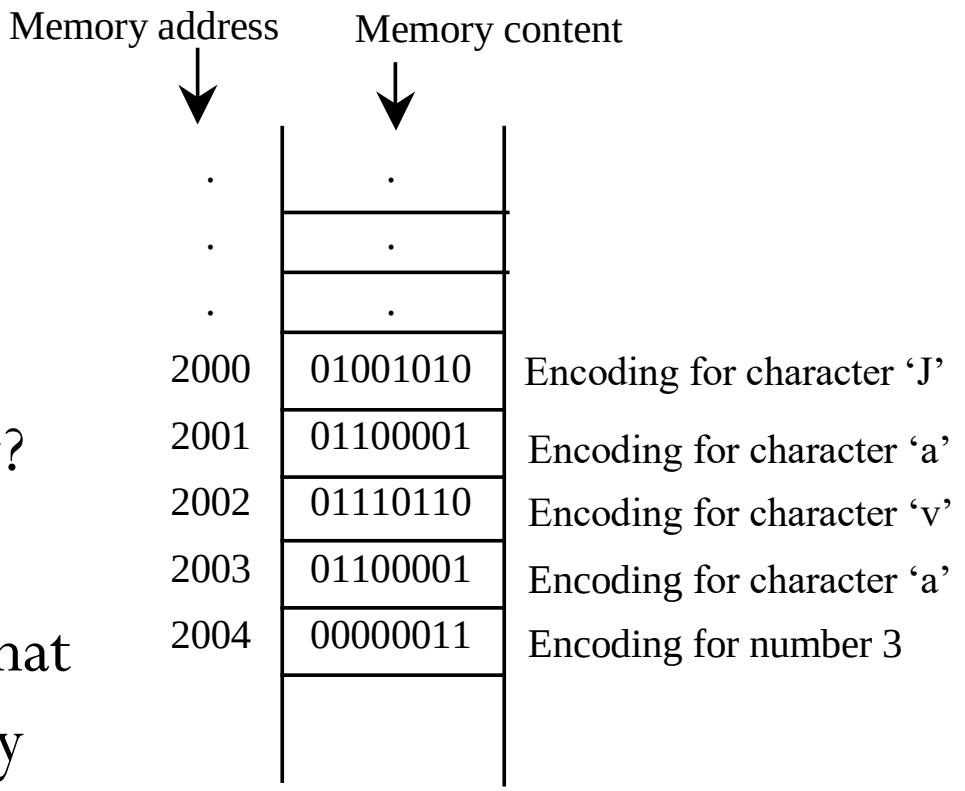
http://www.cs.stonybrook.edu/~cse260

Objectives

- Refresh information from CSE160

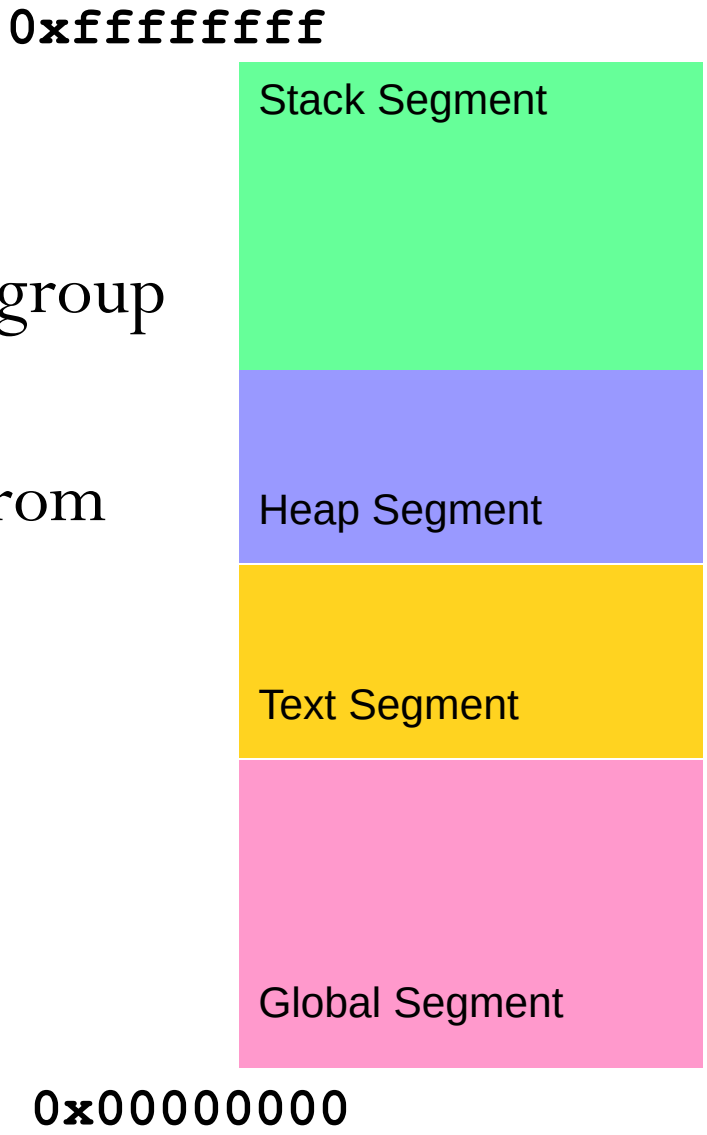
How Data is Stored?

- What's binary?
  - a base-2 number system
- What do humans use?
  - base-10
  - Why?
- Why do computers like binary?
  - electronics
    - easier to make hardware that stores and processes binary numbers than decimal numbers
    - more efficient: space & cost



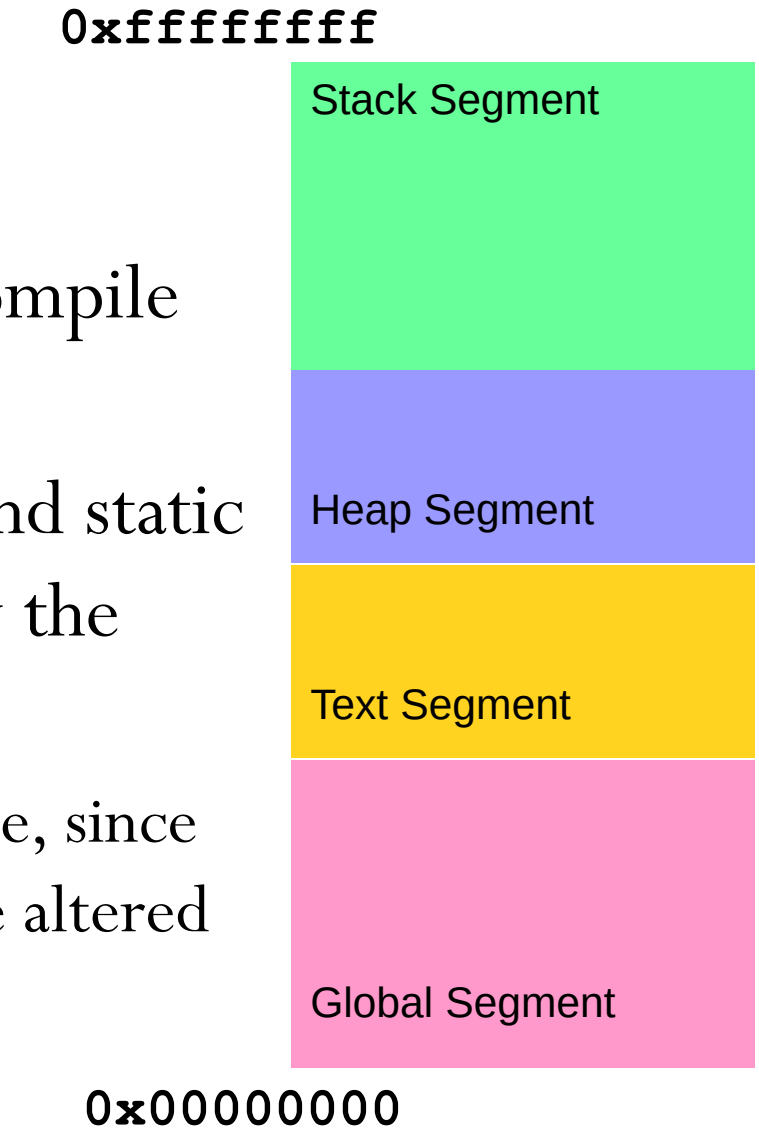
What is memory?

- A giant array of bytes
- Data is byte addressable
  - we can access or change any byte (group of 8 bits) independently as needed
- How do we assign data to/get data from memory?
  - in Java we don't
  - the JVM does
  - using memory addresses
- We use object ids/references



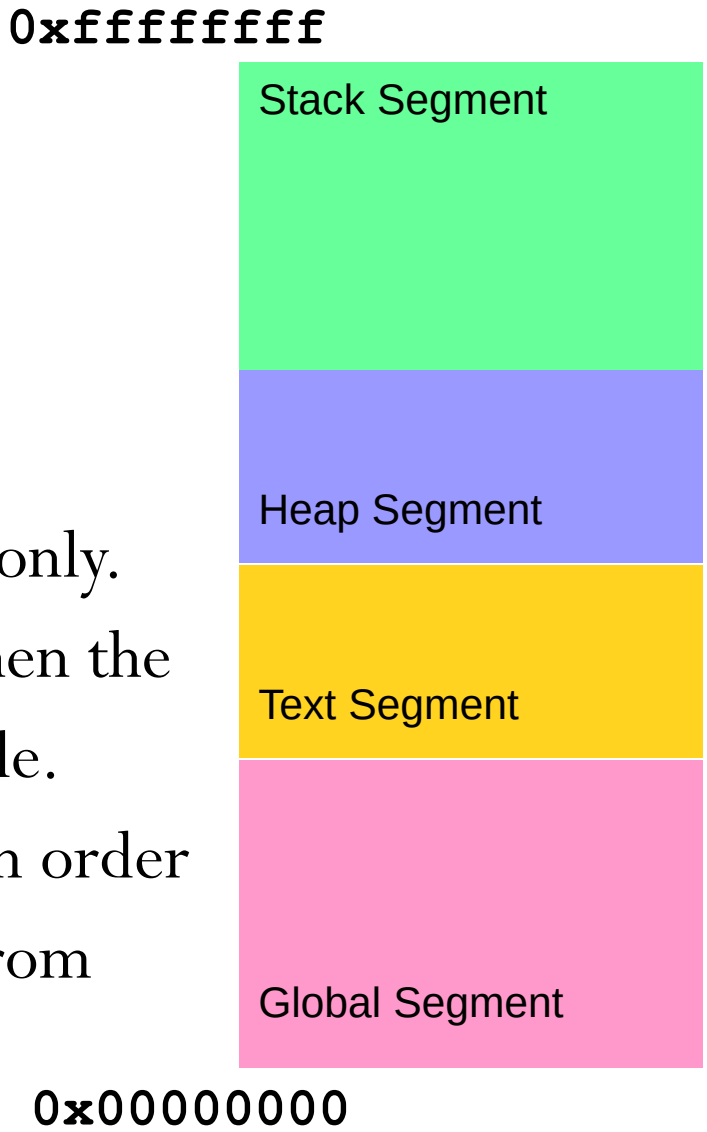
What goes in each memory segment?

- Global Segment
  - data that can be reserved at compile time
  - contains the global variables and static variables that are initialized by the programmer
    - The **data segment** is read-write, since the values of the variables can be altered at run-time.



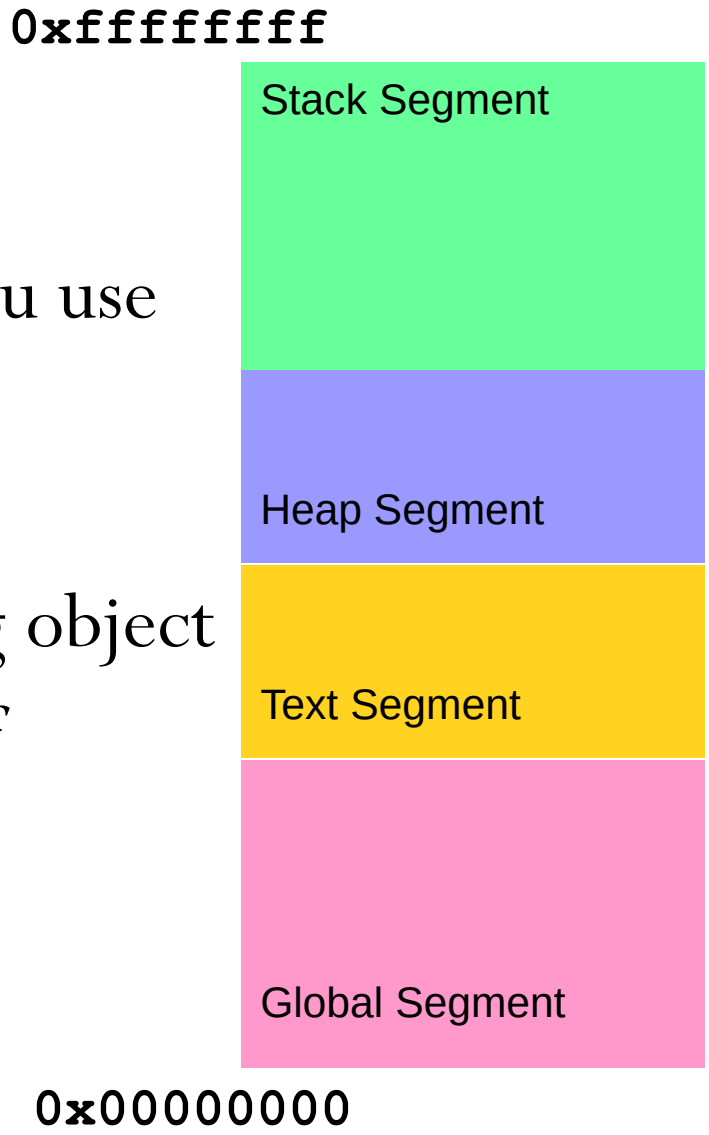
What goes in each memory segment?

- Text Segment
  - Also called **code segment**
  - stores program instructions
    - contains executable instructions
  - It has a fixed size and is usually read-only.
  - If the text section is not read-only, then the architecture allows self-modifying code.
  - It is placed below the heap or stack in order to prevent heap and stack overflows from overwriting it.



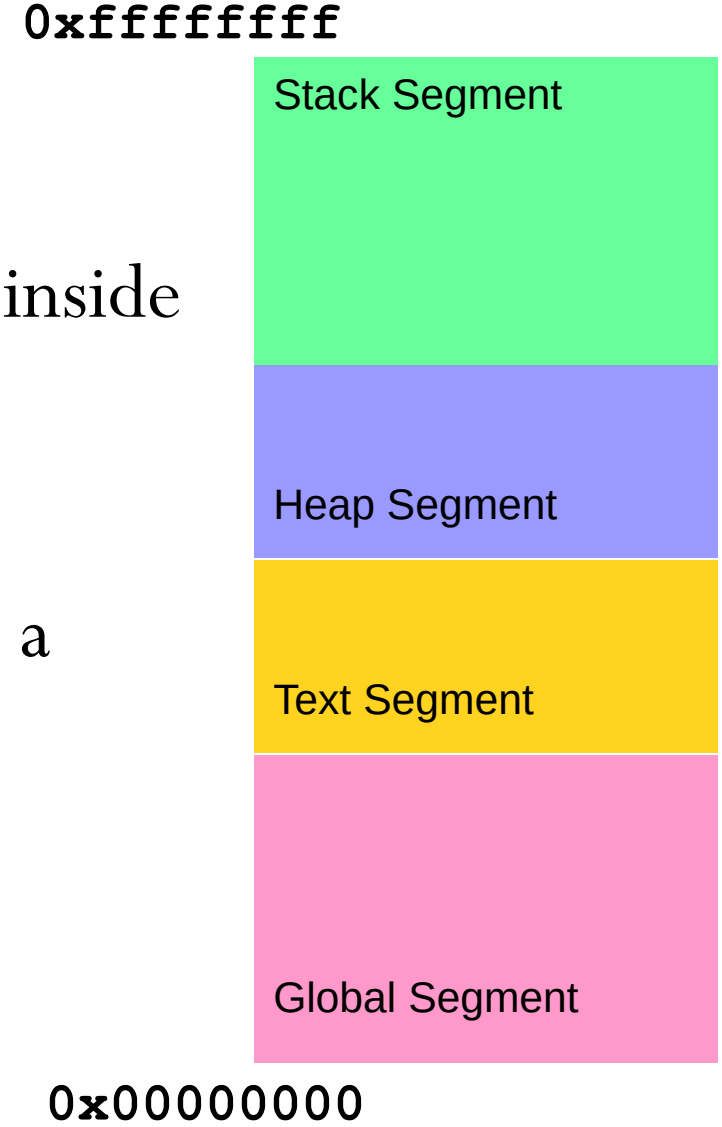
What goes in each memory segment?

- Heap Segment
  - for dynamic data (whenever you use new)
  - data for constructed objects
  - persistent as long as an existing object variable references this region of memory
    - Java, C#, Python, etc.
    - Automatic Garbage Collection



What goes in each memory segment?

- Stack Segment
  - temporary variables declared inside methods
  - method arguments
  - removed from memory when a method returns



Anatomy of a Java Program

- Comments
- Reserved words
- Modifiers
- Statements
- Blocks
- Classes
- Methods
- The main method

Modifiers

Java uses certain reserved words called modifiers that specify the **properties** of the data, methods, and classes and how they can be used

- Examples: **`public`**, **`static`**, **`private`**, **`final`**, **`abstract`**, **`protected`**
- A **`public`** datum, method, or class can be accessed by other programs
- A **`private`** datum or method cannot be accessed by other programs

Variable, class, and method names

- What's an API?
  - Application Programming Interface
  - a library of code to use
- Names
  - For Variables, Classes, and Methods
  - From 2 sources:
    - your own classes, variables, and methods
    - the Oracle/Sun (or someone else's) API
  - Your Identifiers (Names) – Why name them?
    - they are your data and commands
    - you'll need to reference them elsewhere in your program

```
int myVariable = 5; // Declaration
myVariable = myVariable + 1; // Using the variable
```

Rules for Identifiers

- Should contain only letters, numbers, & '_'
  - '$' is allowed, but only for special use
- Cannot begin with a digit!
- Uppercase and lowercase letters are considered to be different characters
- Examples:
  - Legal: **`myVariable, my_class, my4Var`**
  - Illegal: **`4myVariable, my class, my!Var, @#$myClass`**

Common Java Naming Conventions

- Variables & Methods start with lower case letters: **`x, toString`**
- Classes start with upper case letters: **`Person`**
- Variables and Class identifiers should generally be nouns
- Method identifiers should be verbs
- Use Camel notation: **`myVariable, MyClass`**
- Although it is legal, do not begin with '_' (underscore).
- Use descriptive names: **`LinkedList, compareTo`**

```
area = PI * radius * radius;
```

Programming Errors

- Syntax / Compiler Errors
  - Detected by the compiler
- Runtime Errors
  - Causes the program to abort
- Logic Errors
  - Produces incorrect result

Syntax Error

```
public class ShowSyntaxError {
  public static void main(String[] args) {
    i = 30; // Detected by the compiler
    System.out.println(i + 4);
  }
}
```

Runtime Error

```
public class ShowRuntimeError {
  public static void main(String[] args) {
    int i = 1 / 0;    // Division with 0
  }
}
```

Logic Errors

```
public class ShowLogicError {
  // Determine if a number is between 1 and 100 inclusively
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int number = input.nextInt();
    // Display the result
    System.out.println(
      "The number is between 1 and 100, inclusively: " +
          ((1 < number) && (number < 100)) );
        // Wrong result if the entered number is 1 or 100
    System.exit(0);
  }
}
```

Logic Errors Debugging

- Logic errors are called *bugs*
- The process of finding and correcting errors is called debugging
- Methods:
  - hand-trace the program (i.e., catch errors by reading the program),
  - insert print statements in order to show the values of the variables
  - for a large, complex program, the most effective approach for debugging is to use a debugger utility

Debugger

Debugger is a program that facilitates debugging. You can use a debugger to:

- Execute a single statement at a time.
- Trace into or stepping over a method.
- Set breakpoints.
- Display variables.
- Display call stack.
- Modify variables.

Java's Primitive Types

- Integers (whole numbers)
  - **`byte`**–1 byte (-128 to 127)
  - **`short`** –2 bytes (-32768 to 32767)
  - **`int`**–4 bytes (-2147483648 to 2147483647)
  - **`long`**–8 bytes (-9223372036854775808 to 9223372036854775807)
- Real Numbers
  - **`float`**–4 bytes
  - **`double`**–8 bytes
- **`char`**–2 bytes
  - stores a single character (Unicode 2)
- **`boolean`**–stores **`true`** or **`false`** (uses 1-bit or byte)

Arithmetic Operators

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo/Remainder (integer operands only)
++	Increment by one
--	Decrement by one
+=	Increment by specified amount
-=	Decrement by specified amount
*=	Multiply by specified amount
/=	Divide by specified amount

Division

- Integer division:
  - 8/3 = 2
- Double division:
  - 8.0/3.0 = 2.6666666666666667
  - 8.0/3 = 2.6666666666666667
  - 8/3.0 = 2.6666666666666667

Arithmetic Operators

- **Division operator (evaluate full expression first, then assignment):**

```
double average = 100.0/8.0;    //12.5
average = 100.0/8;             //12.5
average = 100/8;               //12.0
int sumGrades = 100/8;         //12
sumGrades = 100.0/8.0;         //ERROR
sumGrades = (int)100.0/8.0;    //ERROR
sumGrades = (int)(100.0/8.0); //12
int fifty_percent = 50/100;    //0
double fiftyPercent = 50/100; //0.0
fiftyPercent = 50.0/100.0;     //0.5
```

Increment and Decrement Operators

```
int i = 10;
int newNum = 10 * i++;
```

Same effect as

```
int newNum = 10 * i;
i = i + 1;
```

```
int i = 10;
int newNum = 10 * (++i);
```

Same effect as

```
i = i + 1;
int newNum = 10 * i;
```

Packages

- To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages.
- The types that are part of the Java platform are members of various packages that bundle classes by function: fundamental classes are in *java.lang*, classes for reading and writing (input and output) are in *java.io*, and so on.
  - You can put your types in packages too.
    - To create a package, you choose a name for the package and put a package statement with that name at the top of *every source file* that contains the types (e.g., classes, interfaces). In file Circle.java:

```
package edu.stonybrook.cse160;
public class Circle {
    ...
}
```

Packages

- To use a public package member from outside its package, you must do one of the following:
  - Refer to the member by its fully qualified name

```
java.util.Scanner input =
  new java.util.Scanner(System.in);
```

  - Import the package member

```
import java.util.Scanner;
```

  - Import the member's entire package

```
import java.util.*;
```

Packages

- Packages appear to be hierarchical, but they are not.
  - Importing **`java.awt.*`** imports all of the types in the java.awt package, but it does not import **`java.awt.color`**, **`java.awt.font`**, or any other **`java.awt.xxxx`** packages.
  - If you plan to use the classes and other types in **`java.awt.color`** as well as those in **`java.awt`**, you must import both packages with all their files:

```
import java.awt.*;
import java.awt.color.*;
```

Setting the CLASSPATH System Variable

- In Windows: **`set CLASSPATH=C:\users\george\java\classes`**
- In Unix-based OS:

```
%CLASSPATH=/home/george/java/classes;
export CLASSPATH
```

Text

- **How do we store text?**
  - Numerically (using its code)
  - Each character is stored in memory as a number
  - Standard character sets: old ASCII & Unicode
    - ASCII uses 1 byte per character
      - ‘A’ is 65

Unicode Format

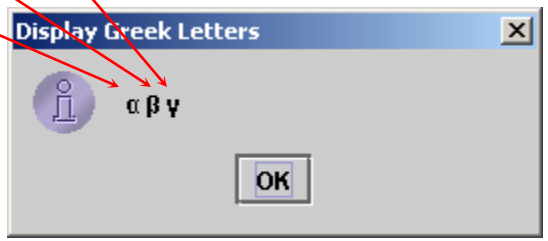
Java characters use *Unicode* UTF-16

16-bit encoding

Unicode takes two bytes, preceded by \u, expressed in four hexadecimal numbers that run from '\u0000' to '\uFFFF'.

Unicode can represent `65535 + 1 characters`.

Unicode \u03b1 \u03b2 \u03b3 for three Greek letters



Character Data Type

Four hexadecimal digits.

```
char letter = 'A'; (ASCII)
char numChar = '4'; (ASCII)
char letter = '\u0041'; (Unicode)
char numChar = '\u0034'; (Unicode)
```

The increment and decrement operators can also be used on char variables to get the next or preceding Unicode character.

- the following statements display character **b**:

```
char ch = 'a';
System.out.println(++ch);
```

The `boolean` Type and Operators

- Often in a programs you need to compare values:

  if `x` is greater than `y`

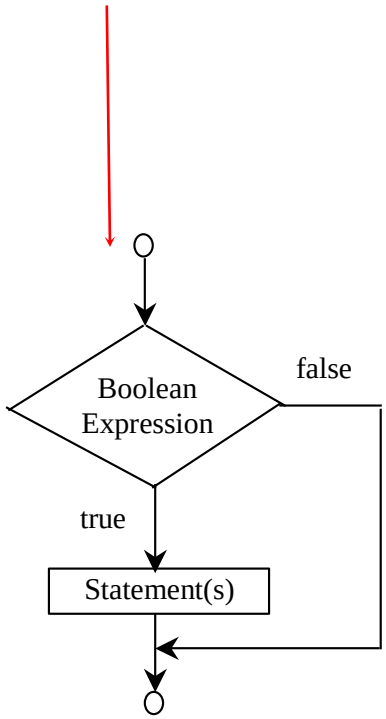
- Java provides six comparison operators (relational operators) to compare two values: `<`, `<=` , `>`, `>=`, `==` and `!=`
- The result of the comparison is a Boolean value: true or false.

```
boolean b = (1 > 2);
```

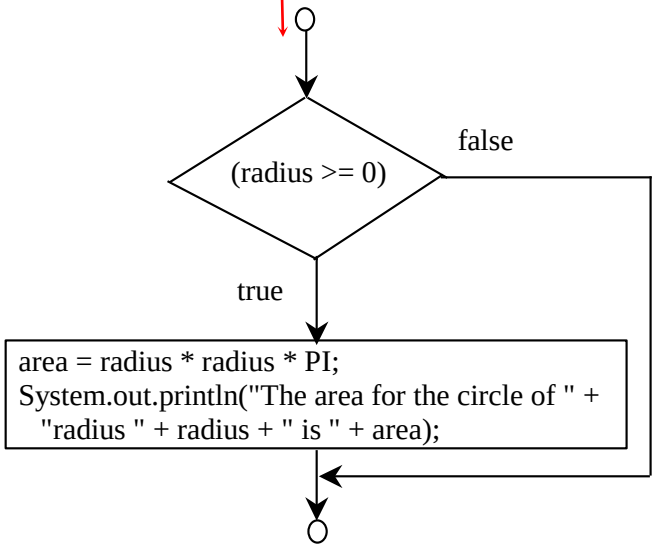
One-way `if` Statements

```
if (boolean-
  expression) {
   statement(s);
}
```

```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area"
   +" for the circle of radius "
   + radius + " is " + area);
}
```



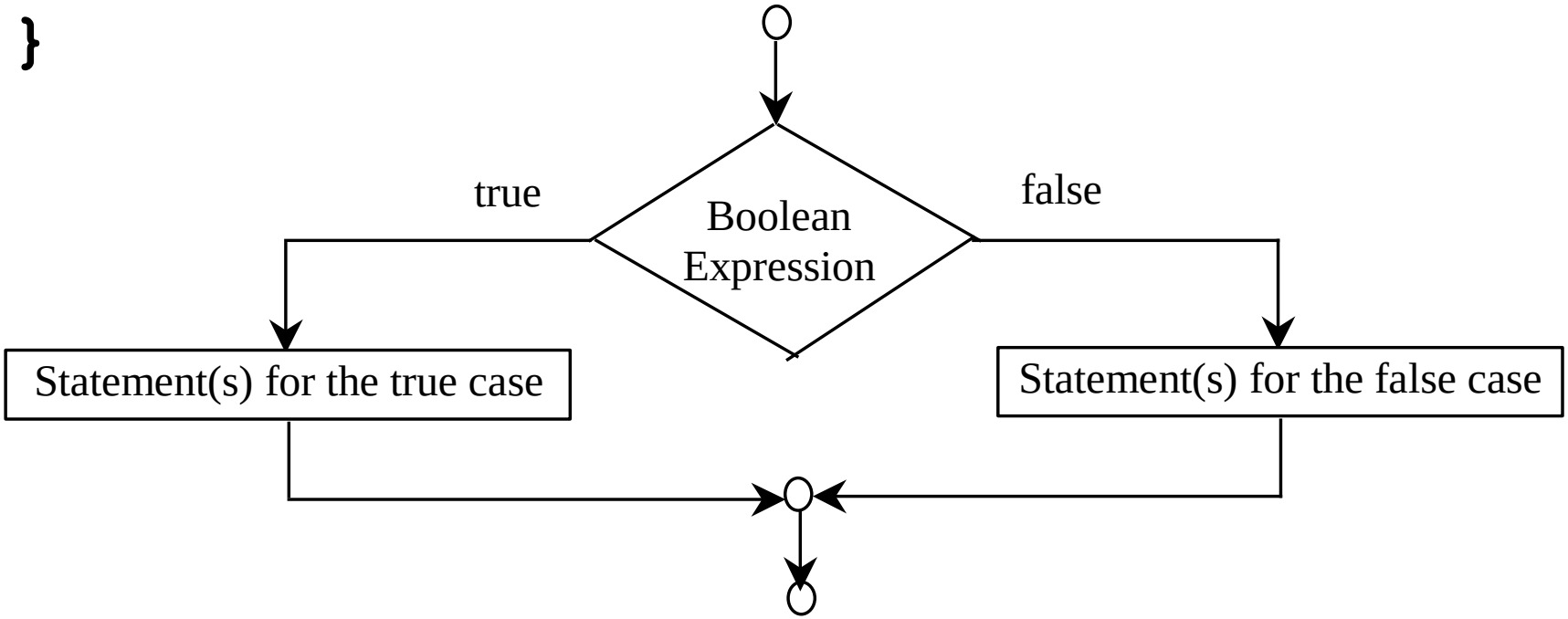
(A)



(B)

Two-way `if` Statement

```
if (boolean-expression) {
  statement(s)-for-the-true-case;
} else {
  statement(s)-for-the-false-case;
}
```



Logical Operators

Operator	*Name*
!	not
&&	and
||	or
^	exclusive or

Determining Leap Year

This program first prompts the user to enter a year as an int value and checks if it is a leap year.

A year is a leap year if it is divisible by 4 but not by 100, or it is divisible by 400.

```
(year % 4 == 0 && year % 100 != 0)
  || year % 400 == 0
```

The unconditional & and | Operators

- The & operator works exactly the same as the && operator, and the | operator works exactly the same as the || operator with one exception:
  - the & and | operators always evaluate both operands

The unconditional & and | Operators

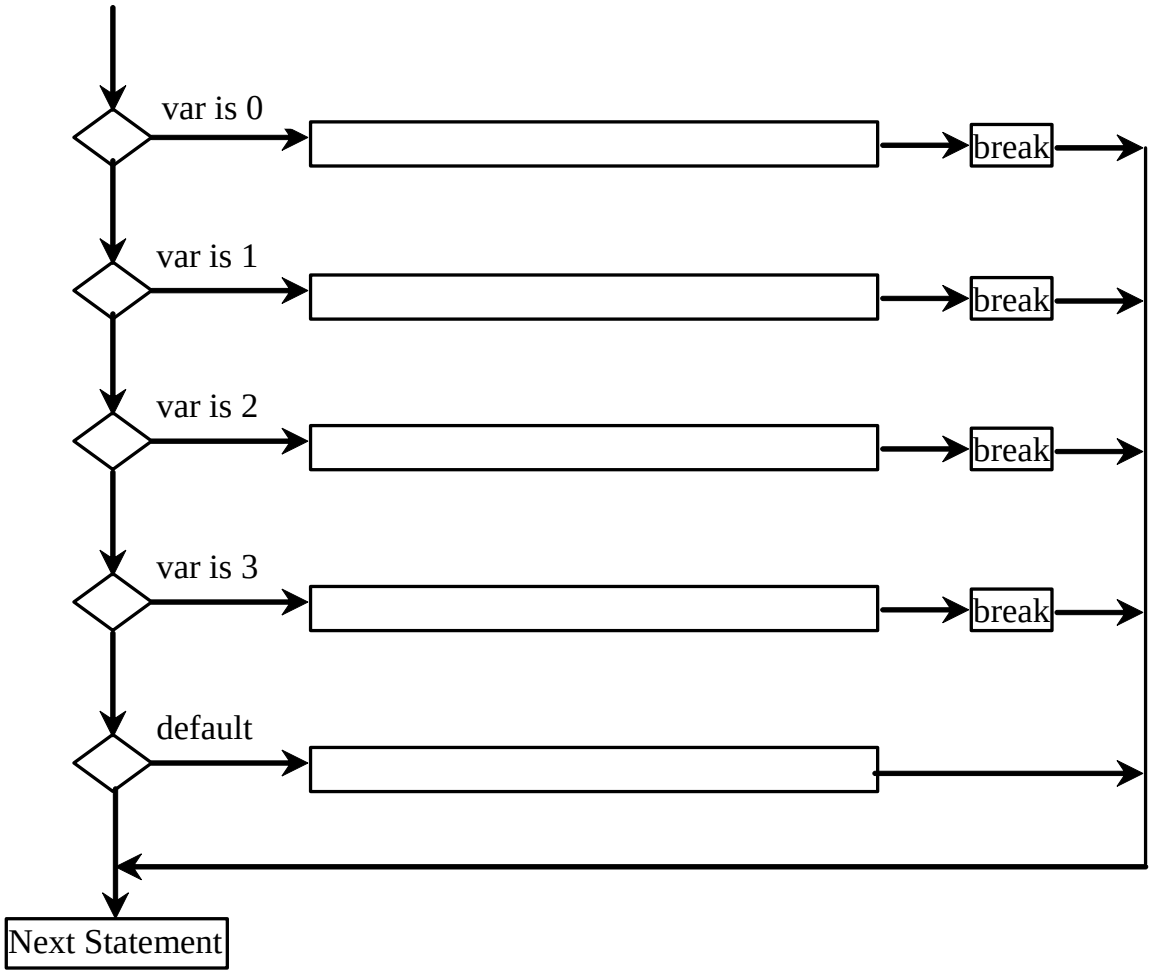
If x is 1, what is x after these expressions:

```
(x > 1) && (x++ < 10)          1

(x > 1) & (x++ < 10)           2


(1 == x) || (10 > x++)?        1

(1 == x) | (10 > x++)?         2
```

switch Statements

```
switch (var) {
   case 0:          ...;
                    break;
   case 1:          ...;
                    break;
   case 2:          ...;
                    break;
   case 3:          ...;
                    break;
   default:         ...;
}
```

`switch` Statement Flow Chart



switch Statement Rules

char, byte, short, int, String

value1, ..., and valueN are **constant expressions** of the **same data type** as the value of the switch-expression

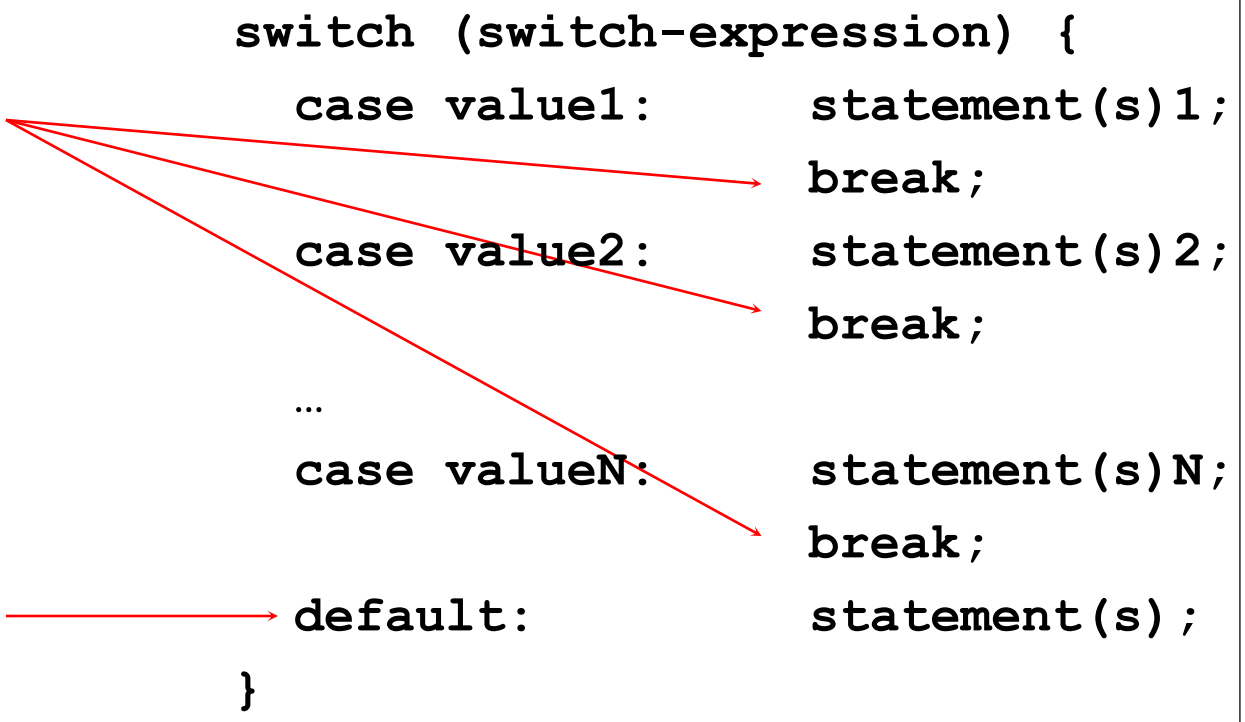
constant = they cannot contain variables in the expression, such as x+y

```
switch (switch-expression) {
  case value1:      statement(s)1;
                    break;
  case value2:      statement(s)2;
                    break;
  …
  case valueN:      statement(s)N;
                    break;
  default:          statement(s);
}
```

switch Statement Rules

break is optional, but it terminates the remainder of the switch statement

default is optional - executed when none of the specified cases matches the switch-expression.



execution in sequential order

Static methods

- What does **static mean?**
  - associates a method with a particular class name
  - any method can call a **static method either:**
    - directly from within same class OR
    - using class name from outside class

The `Math` Class

- Class constants:
  - `PI`
  - `E`
- Class methods:
  - Trigonometric Methods
  - Exponent Methods
  - Rounding Methods
  - min, max, abs, and random Methods

Trigonometric Methods

- `sin(double a)`
- `cos(double a)`
- `tan(double a)`
- `acos(double a)`
- `asin(double a)`
- `atan(double a)`

Radians

- **Examples:**

```
Math.sin(0) returns 0.0
Math.sin(Math.PI / 6)
  returns 0.5
Math.sin(Math.PI / 2)
  returns 1.0
Math.cos(0) returns 1.0
Math.cos(Math.PI / 6)
  returns 0.866
Math.cos(Math.PI / 2)
  returns 0
```

Exponent Methods

- **`exp(double a)`**
  Returns `e` raised to the power of `a`.
- **`log(double a)`**
  Returns the natural logarithm of `a`.
- **`log10(double a)`**
  Returns the 10-based logarithm of `a`.
- **`pow(double a, double b)`**
  Returns `a` raised to the power of `b`.
- **`sqrt(double a)`**
  Returns the square root of `a`.

- **Examples:**

```
Math.exp(1) returns 2.71
Math.log(2.71)
   returns 1.0
Math.pow(2, 3)
   returns 8.0
Math.pow(3, 2)
   returns 9.0
Math.pow(3.5, 2.5)
   returns 22.91765
Math.sqrt(4) returns 2.0
Math.sqrt(10.5)
   returns 3.24
```

Rounding Methods

- **`double ceil(double x)`**

  x rounded up to its nearest integer. This integer is returned as a double value.

- **`double floor(double x)`**

  x is rounded down to its nearest integer. This integer is returned as a double value.

- **`double rint(double x)`**

  x is rounded to its nearest integer. If x is equally close to two integers, the even one is returned as a double.

- **`int round(float x)`**

  Return (int)Math.floor(x+0.5).

- **`long round(double x)`**

  Return (long)Math.floor(x+0.5).

Rounding Methods Examples

Math.ceil(2.1) returns 3.0
Math.ceil(2.0) returns 2.0
Math.ceil(-2.0) returns -2.0
Math.ceil(-2.1) returns -2.0
Math.floor(2.1) returns 2.0
Math.floor(2.0) returns 2.0
Math.floor(-2.0) returns -2.0
Math.floor(-2.1) returns -3.0
Math.round(2.6f) returns 3
Math.round(2.0) returns 2
Math.round(-2.0f) returns -2
Math.round(-2.6) returns -3

`min`, `max`, and `abs`

- **`max(a, b)`** and **`min(a, b)`**

  Returns the maximum or minimum of two parameters.

- **`abs(a)`**

  Returns the absolute value of the parameter.

- **`random()`**

  Returns a random `double` value
  in the range [0.0, 1.0).

- **Examples:**

```
Math.max(2, 3)
  returns 3
Math.max(2.5, 3)
  returns 3.0
Math.min(2.5,3.6)
  returns 2.5
Math.abs(-2)
  returns 2
Math.abs(-2.1)
  returns 2.1
```

The **random** Method

Generates a random **double** value greater than or equal to 0.0 and less than 1.0 (**0 <= Math.random() < 1.0**)

Examples:

```
(int)(Math.random() * 10)
```
→ Returns a random integer between 0 and 9.

```
50 + (int)(Math.random() * 50)
```
→ Returns a random integer between 50 and 99.

In general,

```
a + Math.random() * b
```
→ Returns a random number between a and a + b, excluding a + b.

Generating Random Characters

```
(char)((int)'a' + Math.random() * ((int)'z' - (int)'a' + 1))
```

- All numeric operators can be applied to the char operands
  - The char operand is cast into a number if the other operand is a number or a character.
  - So, the preceding expression can be simplified as follows:

```
(char)('a' + Math.random() * ('z' - 'a' + 1))
```

Comparing and Testing Characters

```
if (ch >= 'A' && ch <= 'Z')
  System.out.println(ch + " is an uppercase letter");

if (ch >= 'a' && ch <= 'z')
  System.out.println(ch + " is a lowercase letter");

if (ch >= '0' && ch <= '9')
  System.out.println(ch + " is a numeric character");
```

How objects are stored?

- You must understand that in Java, every object/reference variable stores a memory address
  - 32 bit numbers (4 bytes)

  OR
  - 64 bit numbers (8 bytes)
- These addresses point to memory locations where the objects' data is stored

The String Type

- The **`char`** type only represents one character
- To represent a string of characters, use the data type called **`String`**:

```
String message = "Welcome to Java";
```

`String` is a predefined class in the Java library just like the **`System`** class

http://java.sun.com/javase/8/docs/api/java/lang/String.html

- The **`String`** type is NOT a primitive type
  - The **`String`** type is a *reference type*
    - A String variable is a reference variable, an "*address*" which points to an object storing the value or actual text

Strings are immutable!

- There are no methods to change them once they have been created
  - any new assignment will assign a new String to the old variable

```
String word = "Steven";
word = word.substring(0, 5);
```

  - the variable word is now a reference to a new String that contains "**Steve**"

Useful String functions

- **`charAt, equals, equalsIgnoreCase, compareTo, startsWith, endsWith, indexOf, lastIndexOf, replace, substring, toLowerCase, toUpperCase, trim`**
- **`s.equals(t)`**
  - returns **`true`** if **`s`** and **`t`** have same letters and sequence
  - **`false`** otherwise

Comparing Strings

- Don't use '==' to compare Strings
  - it compares their memory addresses and not actual strings (character sequences)
  - Instead use the **`equals`**/1 method supplied by the String class

Comparing Strings

```
String word1 = new String("Hello");
String word2 = new String("Hello");
if (word1 == word2){
  System.out.println(true);
} else {
  System.out.println(false);
}
```

Result?

Comparing Strings

```
String word1 = new String("Hello");
String word2 = new String("Hello");
if (word1 == word2){
  System.out.println(true);
} else {
  System.out.println(false);
}
```

false

Why? Two different addresses!

Comparing Strings

```
String word1 = new String("Hello");
String word2 = new String("Hello");
if (word1.equals(word2)){
  System.out.println(true);
} else {
  System.out.println(false);
}
```

true

Same content!

Comparing Strings

```
String word1 = "Hello";
String word2 = "Hello";
if (word1 == word2){
  System.out.println(true);
} else {
  System.out.println(false);
}
```

true

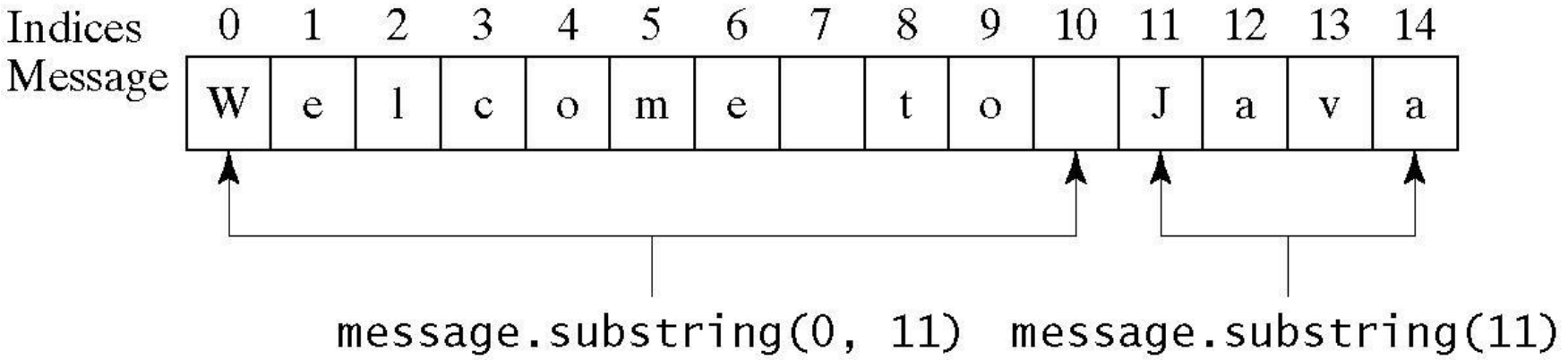
- Interned Strings: Only one instance of "Hello" is stored
  - word1 and word2 will have the same address

Comparing Strings

Method	Description
`equals(s1)`	Returns true if this string is equal to string `s1`.
`equalsIgnoreCase(s1)`	Returns true if this string is equal to string `s1`; it is case insensitive.
`compareTo(s1)`	Returns an integer greater than `0`, equal to `0`, or less than `0` to indicate whether this string is greater than, equal to, or greater than `s1`.
`compareToIgnoreCase(s1)`	Same as `compareTo` except that the comparison is case insensitive.
`startsWith(prefix)`	Returns true if this string starts with the specified prefix.
`endsWith(suffix)`	Returns true if this string ends with the specified suffix.

Obtaining Substrings

Method	Description
`substring(beginIndex)`	Returns this string's substring that begins with the character at the specified `beginIndex` and extends to the end of the string, as shown in Figure 4.2.
`substring(beginIndex, endIndex)`	Returns this string's substring that begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`, as shown in Figure 9.6. Note that the character at `endIndex` is not part of the substring.



Finding a Character or a Substring in a String

Method	Description
`indexOf(ch)`	Returns the index of the first occurrence of `ch` in the string. Returns `-1` if not matched.
`indexOf(ch, fromIndex)`	Returns the index of the first occurrence of `ch` after `fromIndex` in the string. Returns `-1` if not matched.
`indexOf(s)`	Returns the index of the first occurrence of string `s` in this string. Returns `-1` if not matched.
`indexOf(s, fromIndex)`	Returns the index of the first occurrence of string `s` in this string after `fromIndex`. Returns `-1` if not matched.
`lastIndexOf(ch)`	Returns the index of the last occurrence of `ch` in the string. Returns `-1` if not matched.
`lastIndexOf(ch, fromIndex)`	Returns the index of the last occurrence of `ch` before `fromIndex` in this string. Returns `-1` if not matched.
`lastIndexOf(s)`	Returns the index of the last occurrence of string `s`. Returns `-1` if not matched.
`lastIndexOf(s, fromIndex)`	Returns the index of the last occurrence of string `s` before `fromIndex`. Returns `-1` if not matched.

Conversion between Strings and Numbers

```
String intString = "15";
String doubleString = "56.77653";

int intValue =
    Integer.parseInt(intString);
double doubleValue =
    Double.parseDouble(doubleString);

String s2 = "" + intValue;
```

Formatting Output

The printf statement:

```
System.out.printf(format, items);
```

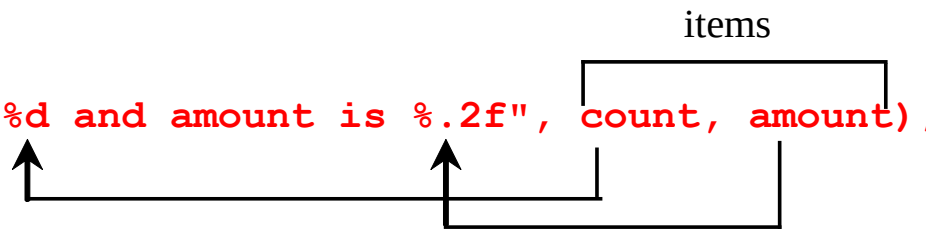
format is a string that may consist of substrings and format **specifiers**

- A format specifier begins with a percent sign and specifies how an item should be displayed: a numeric value, character, boolean value, or a string

Frequently-Used Specifiers

Specifier	Output	Example
%b	a boolean value	true or false
%c	a character	'a'
%d	a decimal integer	200
%f	a floating-point number	45.460000
%e	a number in standard scientific notation	4.556000e+01
%s	a string	"Java is cool"

```
int count = 5;
double amount = 45.5678;
System.out.printf("count is %d and amount is %.2f", count, amount);
```



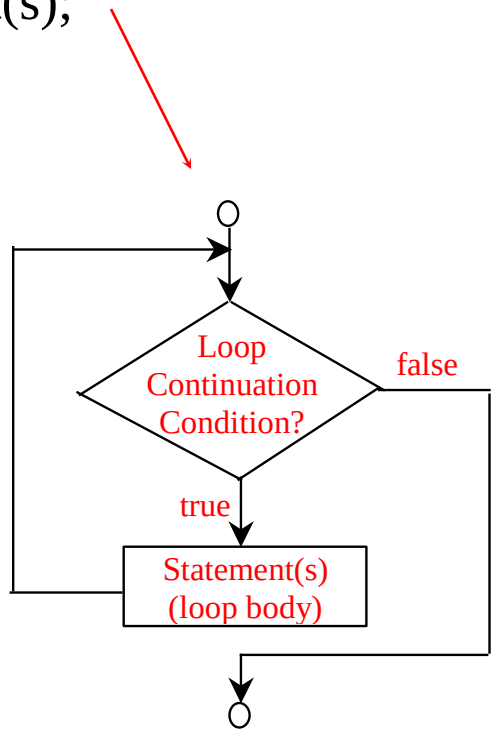
Displays: `count is 5 and amount is 45.56`

Java and iteration

- We have 3 types of iterative statements
  - a **`while`** loop
  - a **`do`** … **`while`** loop
  - a **`for`** loop
- All 3 can be used to do similar things
- Which one should you use?
  - a matter of individual preference/convenience

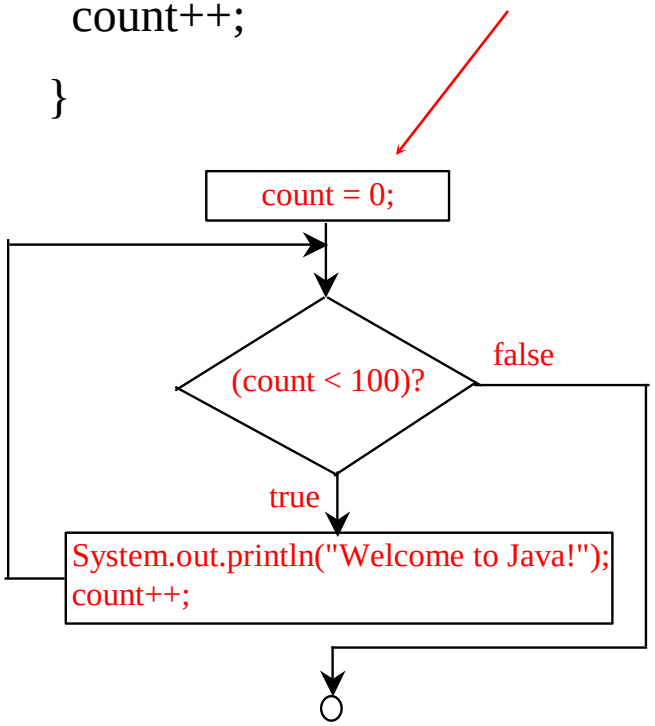
`while` Loop Flow Chart

```
while (loop-continuation-condition) {
  // loop-body;
  Statement(s);
}
```



(A)

```
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java!");
  count++;
}
```



(B)

Caution: equality for reals

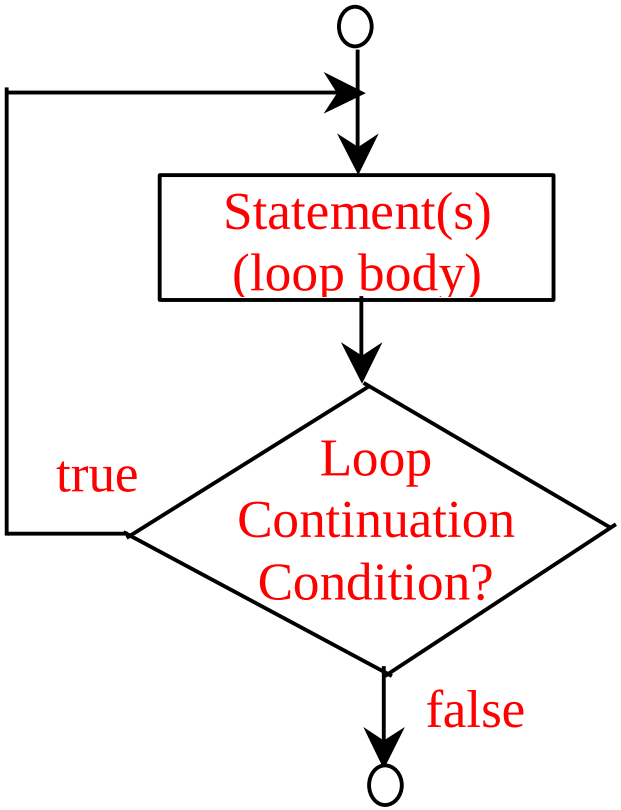
- **Don't use floating-point values for equality checking** in a loop control - floating-point values are **approximations** for some values
- Example: the following code for computing 1 + 0.9 + 0.8 + … + 0.1:

```
double item = 1; double sum = 0;
while (item != 0) { // No guarantee item will be 0 or 0.0
  sum += item;
  item -= 0.1;
}
System.out.println(sum);
```

- Variable item starts with 1 and is reduced by 0.1 every time the loop body is executed
- The loop should terminate when item becomes 0
- There is no guarantee that item will be exactly 0, because the floating-point arithmetic is approximated
  - 0.1 is not represented exactly: 0.1 = 1/16 + 1/32 + 1/256 + 1/512 + 1/4096 + 1/8192 + …
- It is actually an infinite loop!

do-while Loop

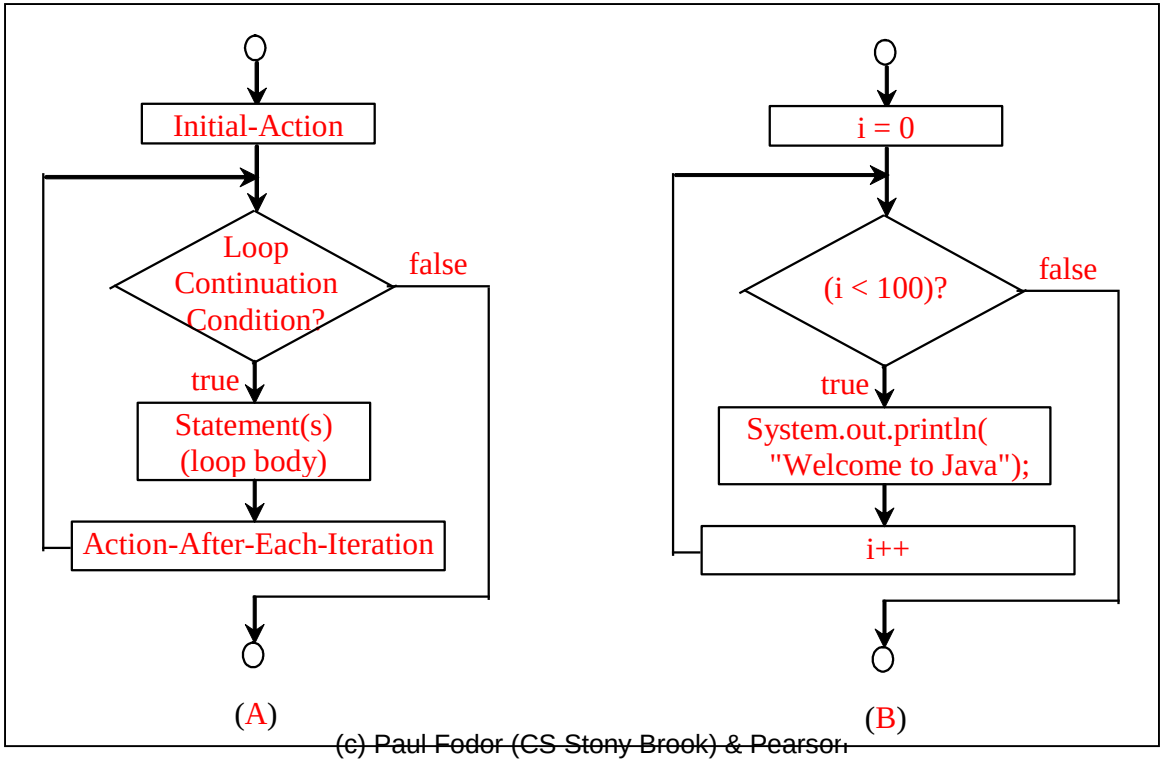
```
do {
  // Loop body;
  Statement(s);
} while (loop-continuation-condition);
```



for Loops

```
for (initial-action;
        loop-continuation-condition;
        action-after-each-iteration) {
    // loop body;
    Statement(s);
}
```

```
int i;
for (i = 0; i < 100; i++){
  System.out.println(
     "Welcome to Java!");
}
```



`for` loops and counting

- **`for`** loops are popular for counting loops
  - through the indices of a string
  - through the indices of an array (later)
  - through iterations of an algorithm
- Good for algorithms that require a known number of iterations
  - counter-controlled loops

`for` loops

The initial-action in a **`for`** loop can be a list of zero or more comma-separated expressions

The action-after-each-iteration in a **`for`** loop can be a list of zero or more comma-separated statements

```
for(int i = 1; i < 100; System.out.println(i++));


for(int i = 0, j = 0; (i + j < 10); i++, j++){
   // Do something
}
```

Infinite loops

If the loop-continuation-condition in a **`for`** loop is omitted, it is implicitly **`true`**

```
for ( ; ; ) {
  // Do something
}
```

(a)

Equivalent

```
while (true) {
  // Do something
}
```

(b)

Keywords **break** and **continue**

- You can also use **break** in a loop to immediately terminate the loop:

```
public static void main(String[] args) {
  int sum = 0;
  int number = 0;
  while (number < 20) {
   number++;
   sum += number;
   if (sum >= 100) // increments until the sum is
        break;     // greater than 100
 }
 System.out.println("The number is " + number);
 System.out.println("The sum is " + sum);
}
```

```
The number is 14
The sum is 105
```

Keywords **break** and **continue**

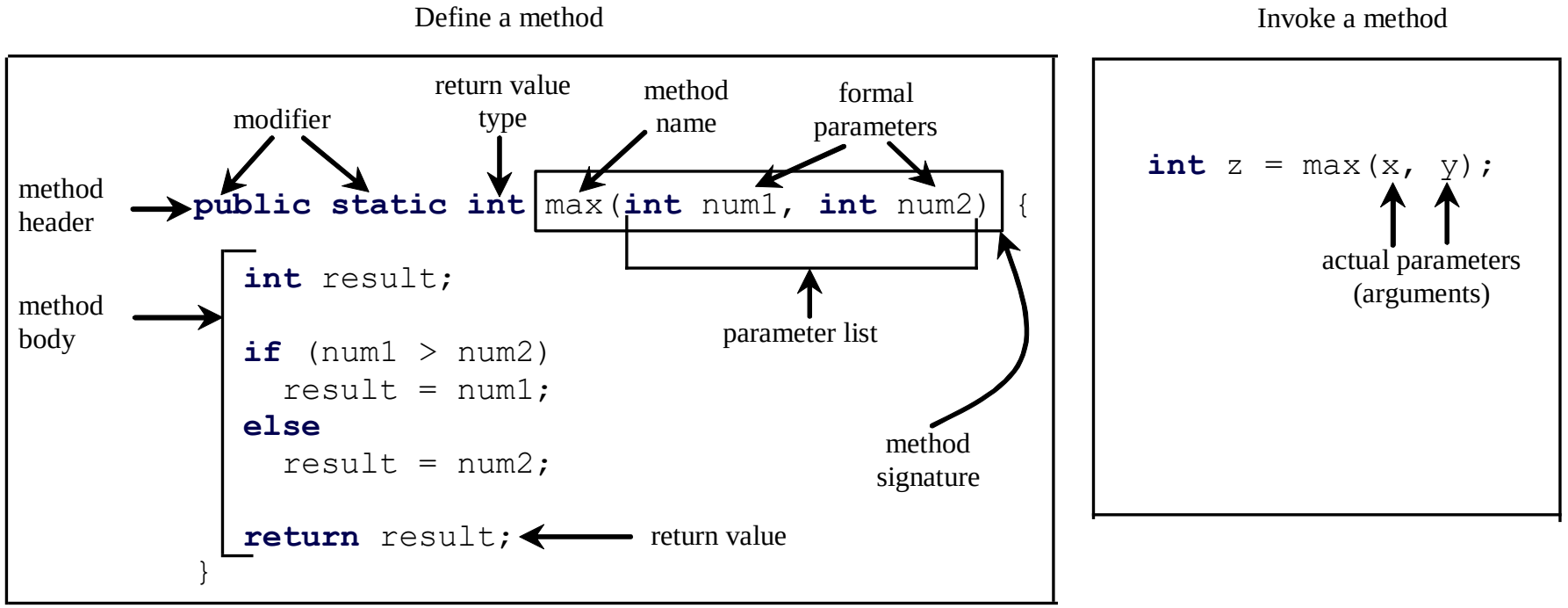
- You can also use **continue** in a loop to end the current iteration and program control goes to the end of the loop body (and continues the loop):

```
public static void main(String[] args) {
  int sum = 0;
  int number = 0;
  while (number < 20) { // adds integers from 1 to 20
   number++;              // except 10 and 11 to sum
   if (number ==10 || number == 11)
          continue;
   sum += number;
 }
 System.out.println("The number is " + number);
 System.out.println("The sum is " + sum);
}
```

```
The number is 20
The sum is 189
```

Defining Methods

- A *method* is a collection of statements that are grouped together to perform an operation

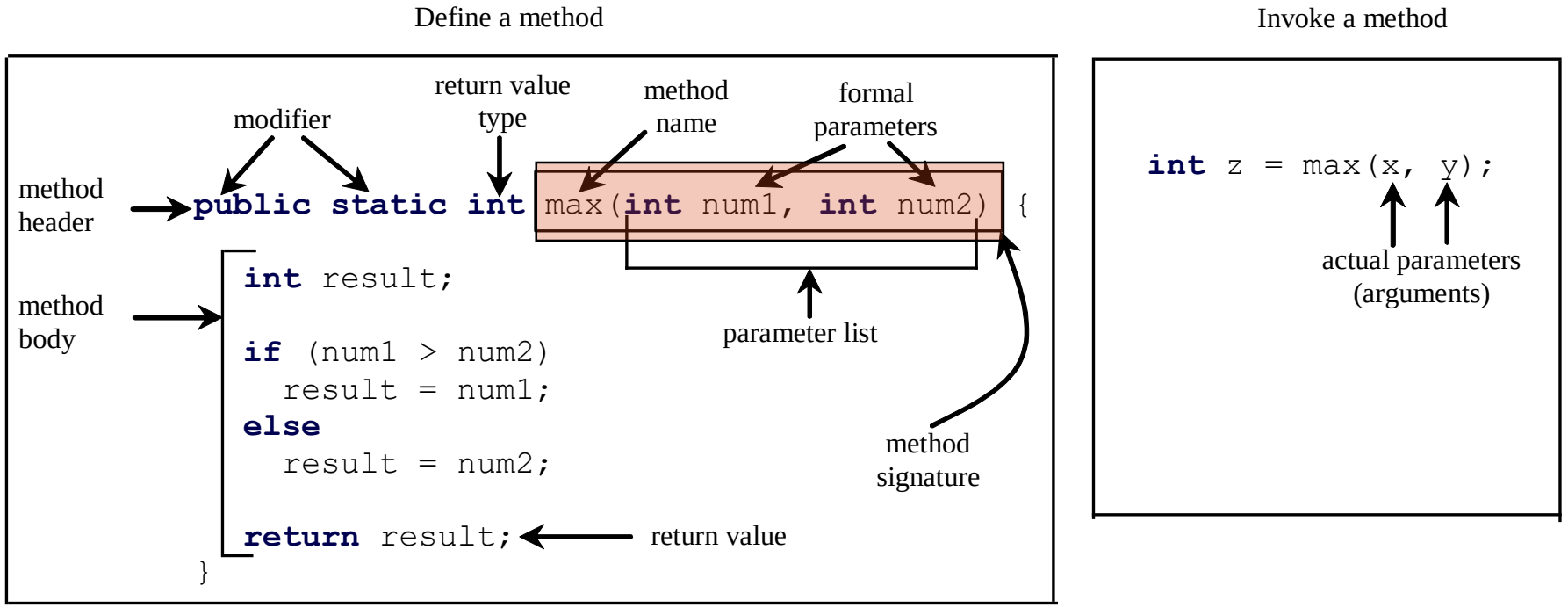


Why write methods?

- To shorten your programs
  - avoid writing identical code twice or more
- To modularize your programs
  - fully tested methods can be trusted
- To make your programs more:
  - readable
  - reusable
  - testable
  - debuggable
  - extensible
  - adaptable

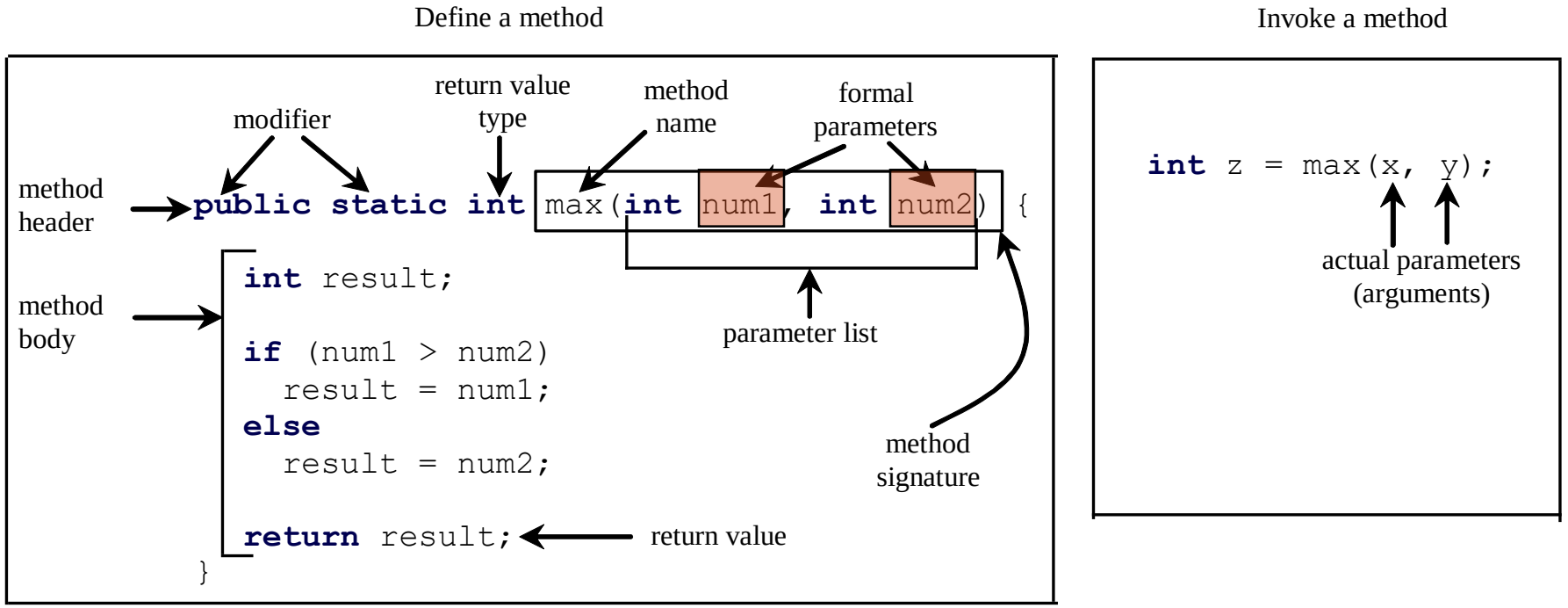
Method Signature

- *Method signature* is the combination of the method name and the parameter list.



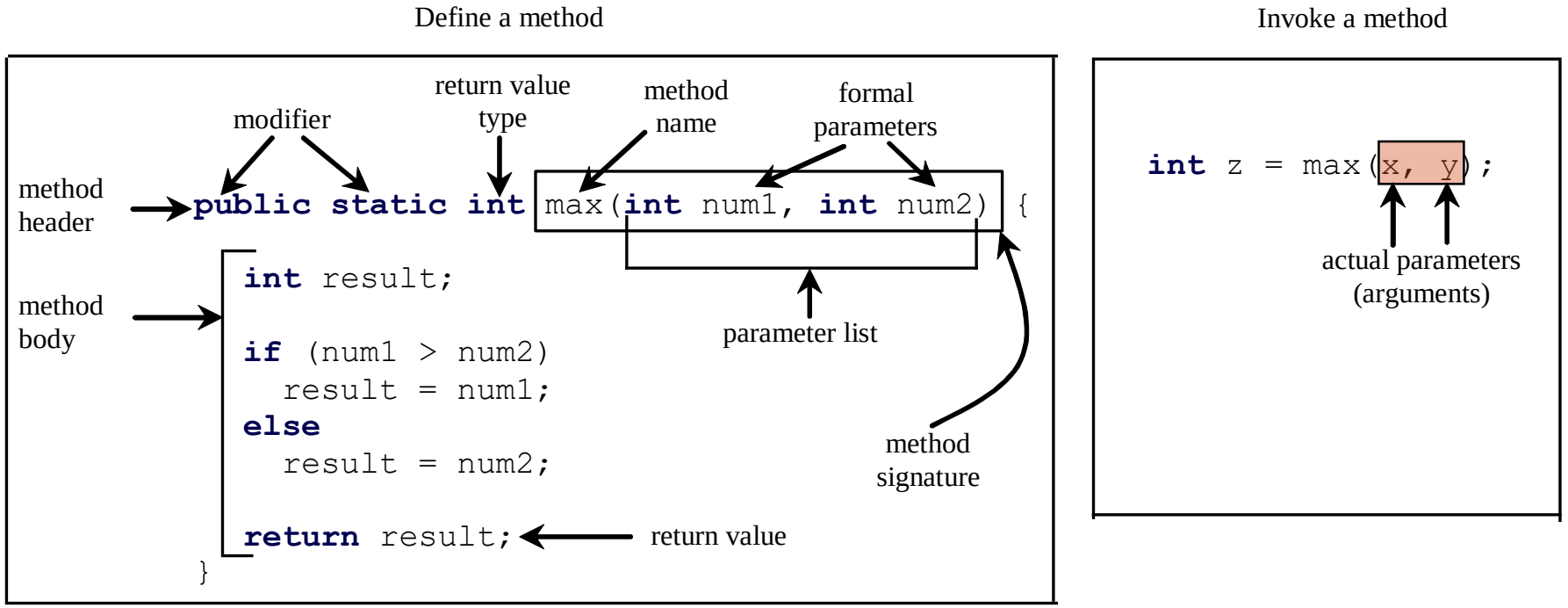
Formal Parameters

- The variables defined in the method header are known as *formal parameters*.



Actual Parameters

- When a method is invoked, you pass a value to the parameter: *actual parameter or argument*.



CAUTION: all execution paths

- A **`return`** statement is required for a value-returning method

The method shown below has a compilation error because the Java compiler thinks it possible that this method does not return any value

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else if (n < 0)
    return -1;
}
```

(a)

Should be

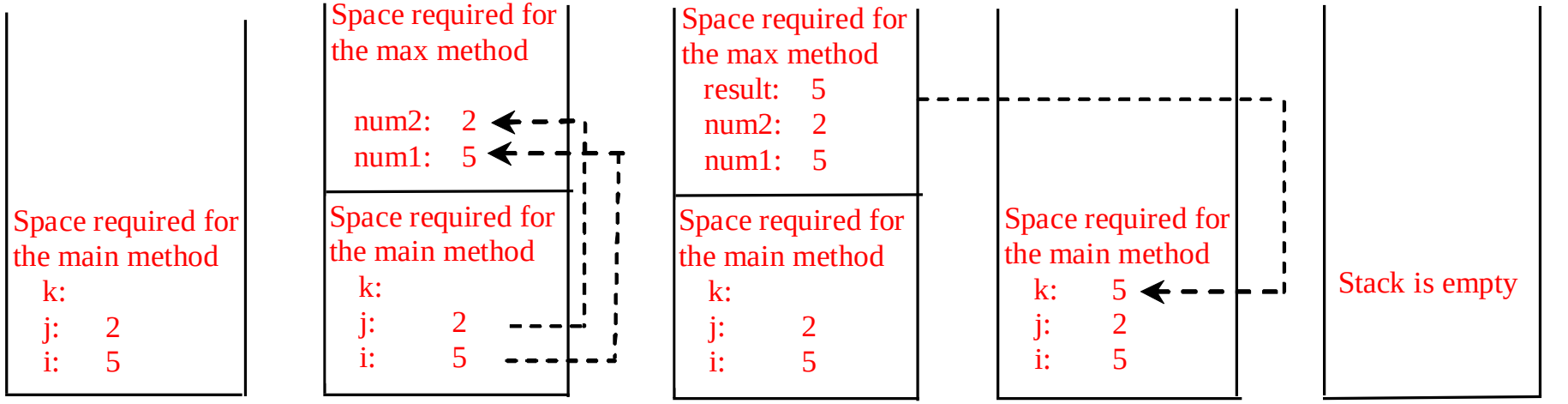
```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else
    return -1;
}
```

(b)

To fix this problem, delete **`if(n < 0)`** in (a), so that the compiler will see a **`return`** statement to be reached regardless of how the **`if`** statement is evaluated.

Call Stacks

Methods are executed using a stack data structure



(a) The main method is invoked.

(b) The max method is invoked.

(c) The max method is being executed.

(d) The max method is finished and the return value is sent to k.

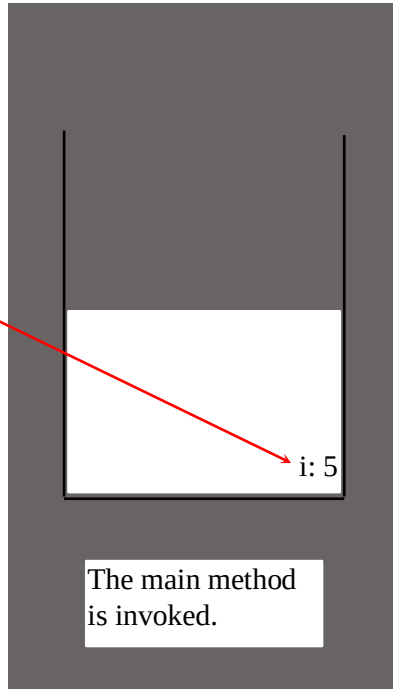
(e) The main method is finished.

Trace Call Stack

i is declared and initialized

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

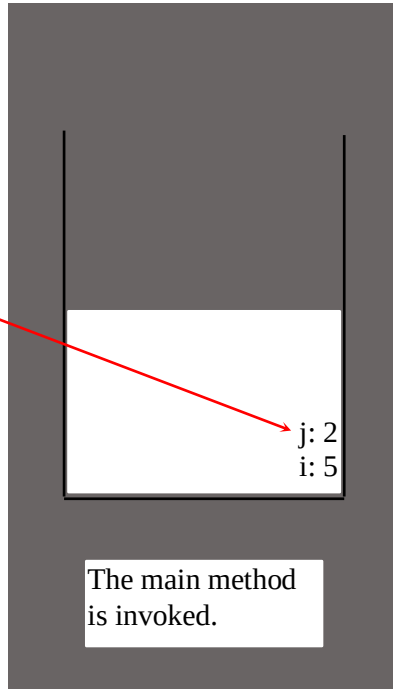


Trace Call Stack

j is declared and initialized

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

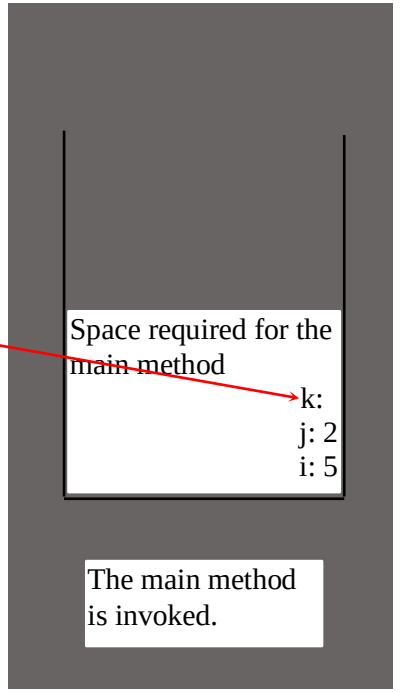


Trace Call Stack

Declare k

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

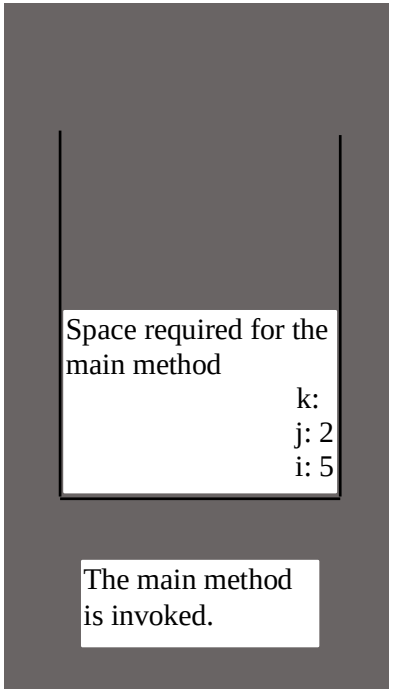


Trace Call Stack

Invoke max(i, j)

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

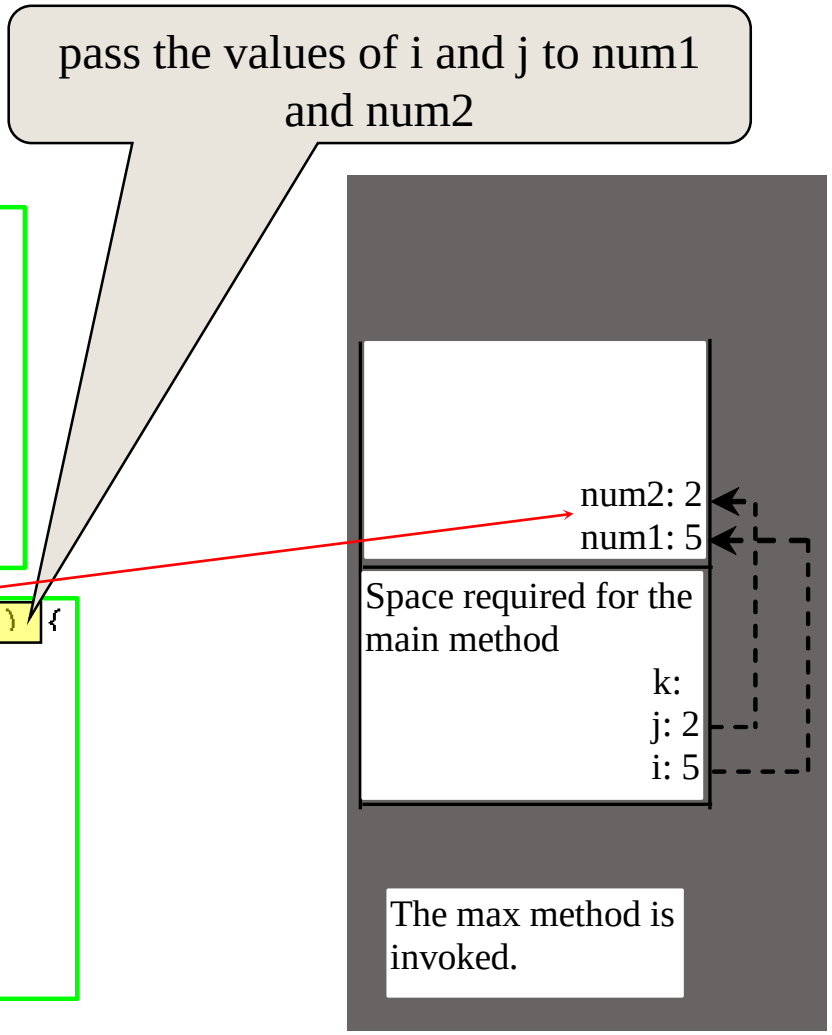
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Trace Call Stack

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

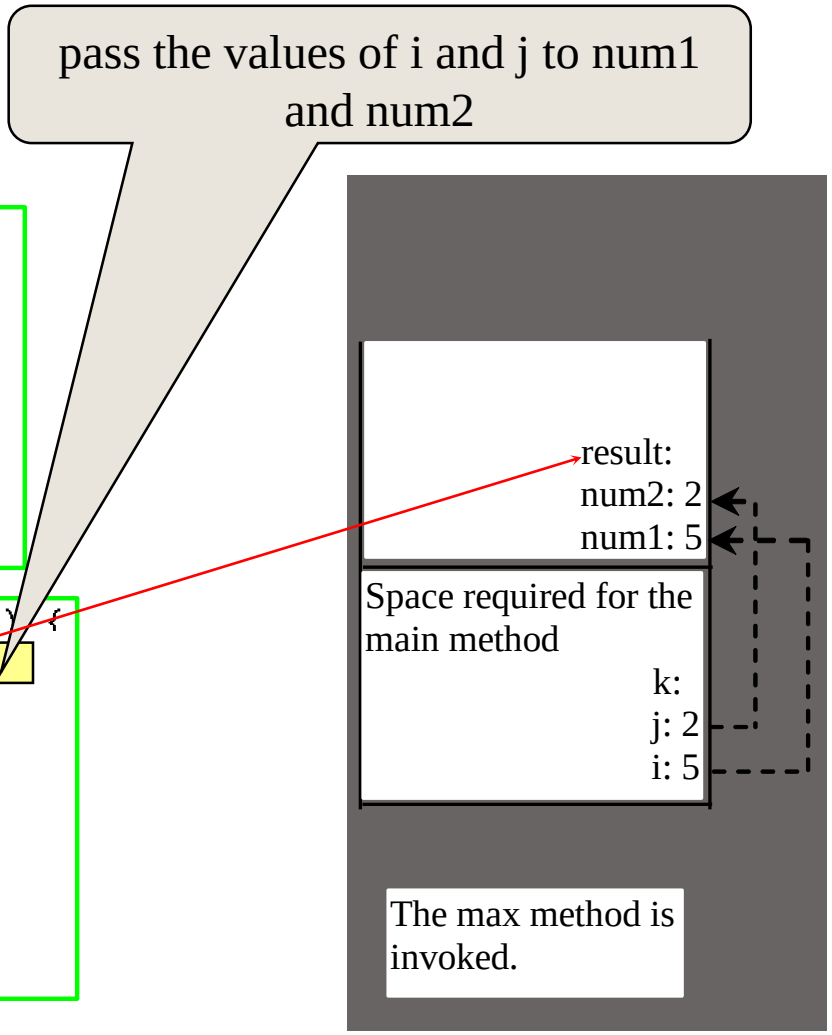
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Trace Call Stack

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

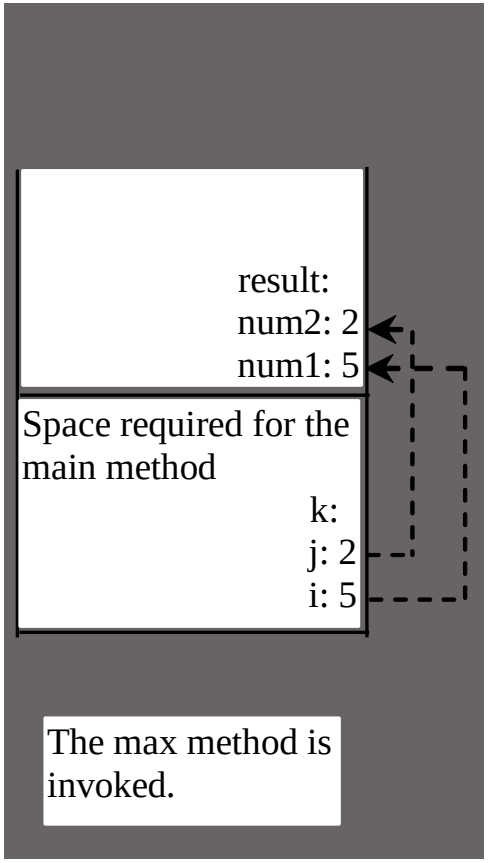


Trace Call Stack

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

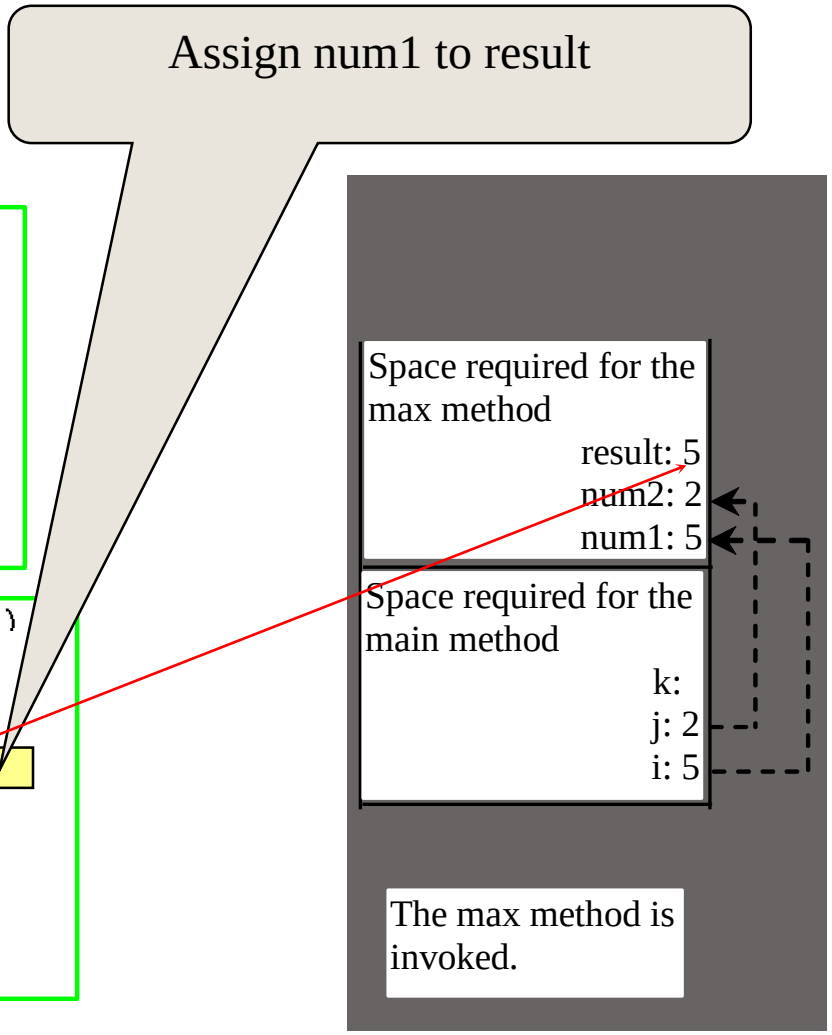
(num1 > num2) is true



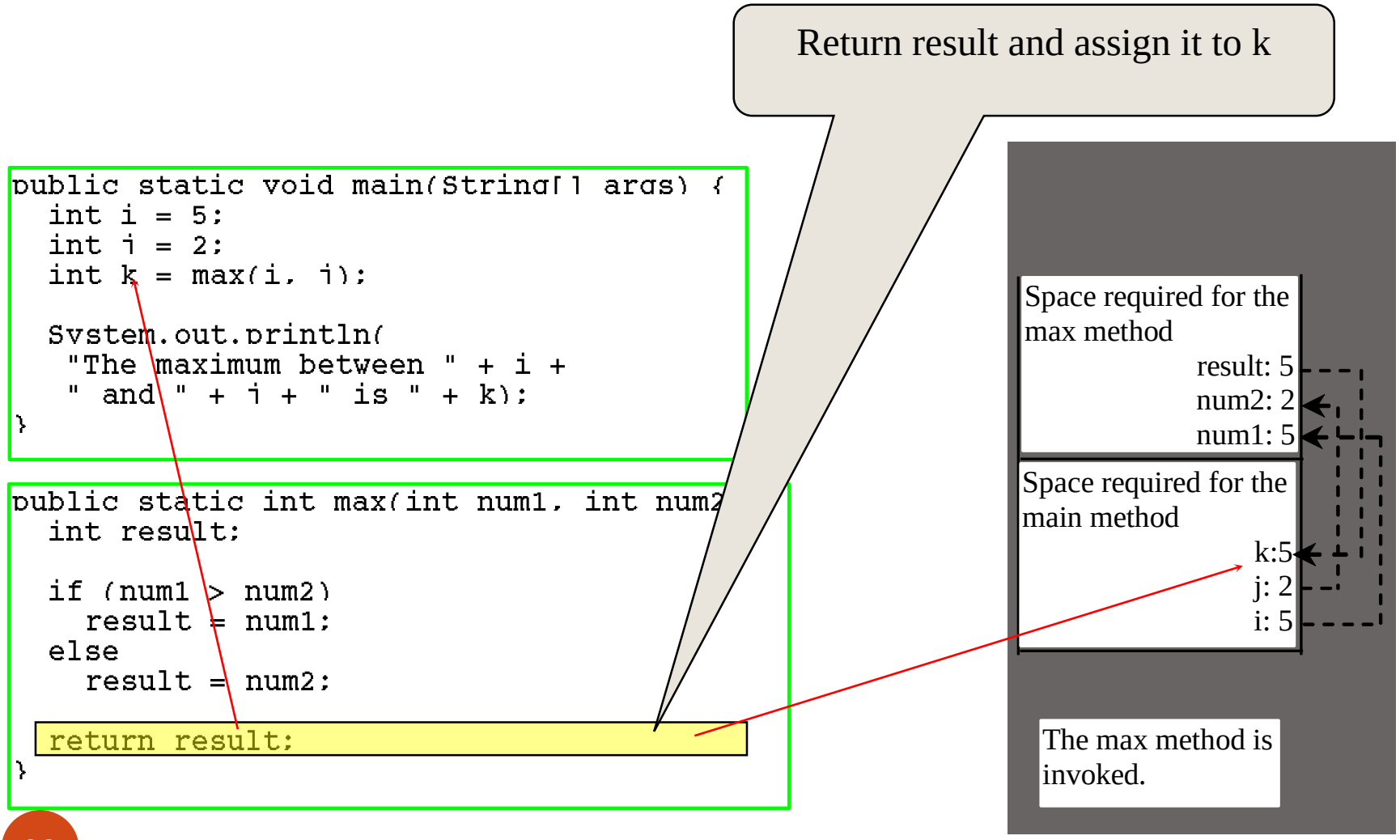
Trace Call Stack

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2)
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Trace Call Stack

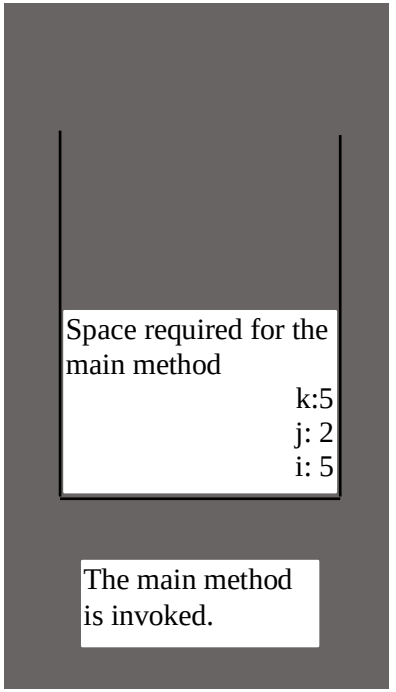


Trace Call Stack

Execute print statement

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

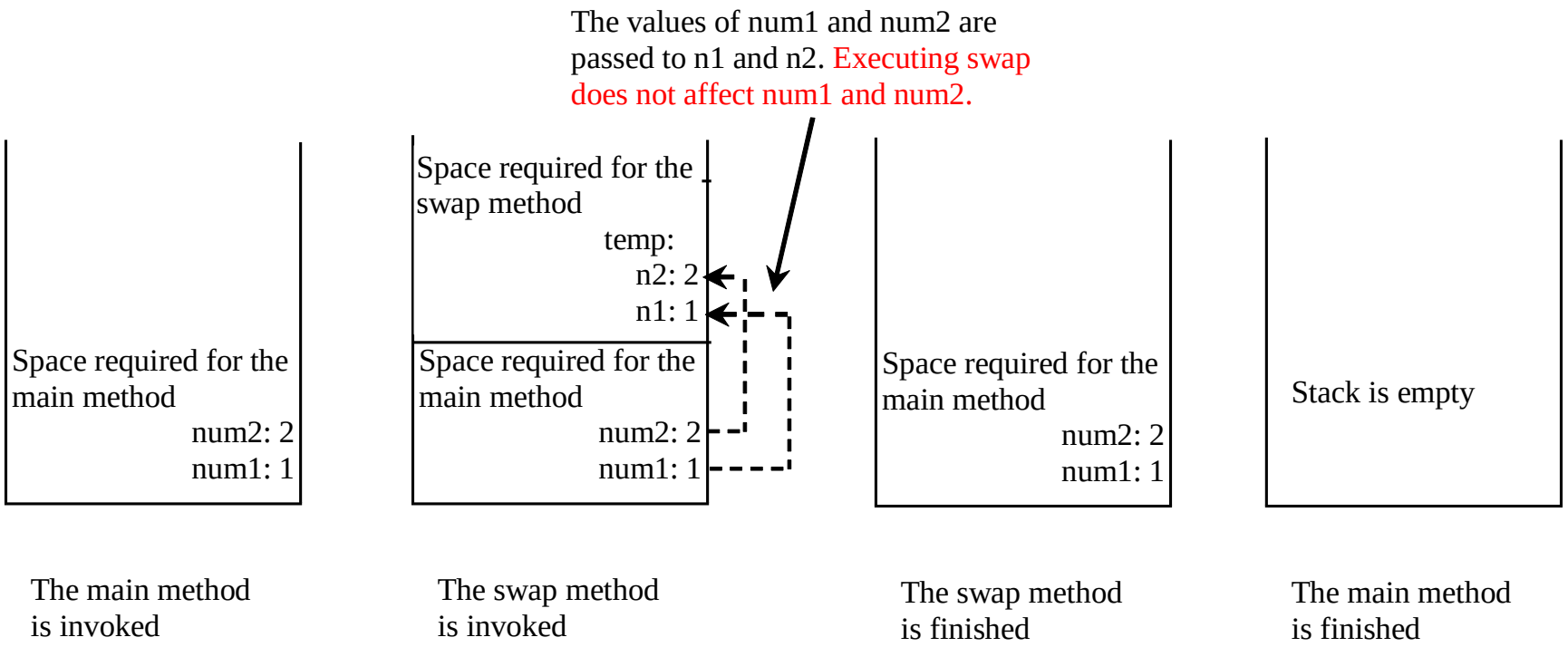
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Call-by-value

- Method formal arguments are ***copies of the original data***
- Consequence?
  - methods cannot assign („=“) new values to primitive type formal arguments and affect the original passed variables.
- Why?
  - changing argument values changes the copy, not the original.

Swap case for Call-by-value



Overloading

- Method overloading is the ability to create multiple methods of the same name with different implementations.

```
// Overload the name max for different invocations
public static int max(int x, int y){
        return (x>y) ? x : y;
}

public static double max(double x, double y){
        return (x>y) ? x : y;
}

public static void main(String[] args) {
        System.out.println(max(1,2)); // will call max(int,int)
        System.out.println(max(3.5,4.7)); // will call max(double,double)
}
```

Overloading & Ambiguous Invocation

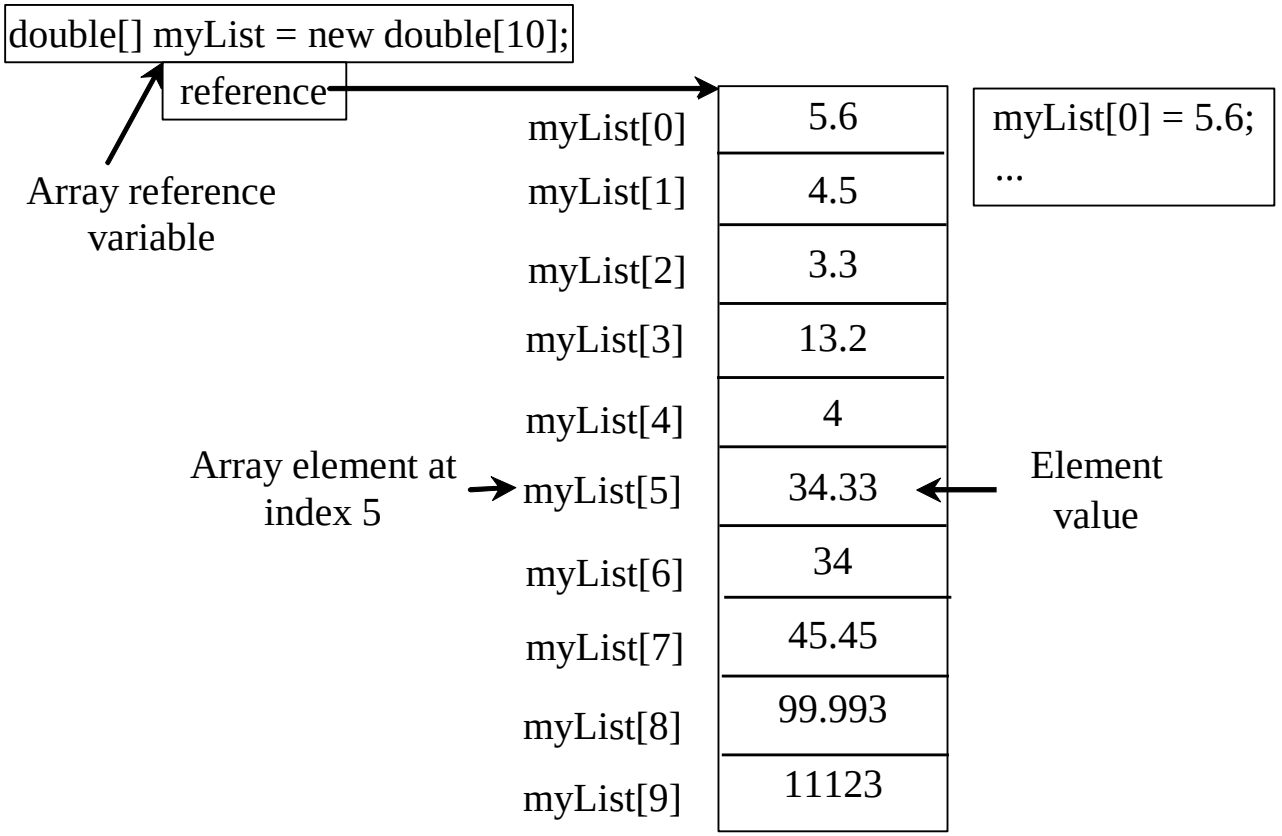
- Sometimes there may be two or more possible matches for an invocation of a method, but the compiler cannot determine the most specific match.
  - This is referred to as *ambiguous invocation*.
- Ambiguous invocation is a compilation error.

Overloading & Ambiguous Invocation

```
public class AmbiguousOverloading {
  public static void main(String[] args) {
    System.out.println(max(1, 2));
  }
  public static double max(int num1, double num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
  public static double max(double num1, int num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
}
```

Introducing Arrays

An *array* is a data structure that represents a collection of the same type of data



Default Values

- When an array is created, its elements are assigned the default value of

  0 for the numeric primitive data types,

  '\u0000' for char types, and

  false for boolean types.

Indexed Variables

- The array elements are accessed through the index
  - The array indices are **0-*based***, i.e., it starts from **0** to **arrayRefVar.length - 1**
- Each element in the array is represented using the following syntax, known as an *indexed variable*:

```
arrayRefVar[index];
```

Array Initializers

- Declaring, creating, initializing in one step:

```
double[] myList = {1.9, 2.9, 3.4, 3.5};
```

This shorthand syntax must be in one statement

Enhanced for Loop (for-each loop)

JDK 1.5 introduced a new for loop that enables you to traverse the complete array sequentially without using an index variable.

- For example, the following code displays all elements in the array myList:

```
for (double value: myList)
      System.out.println(value);
```

In general, the syntax is

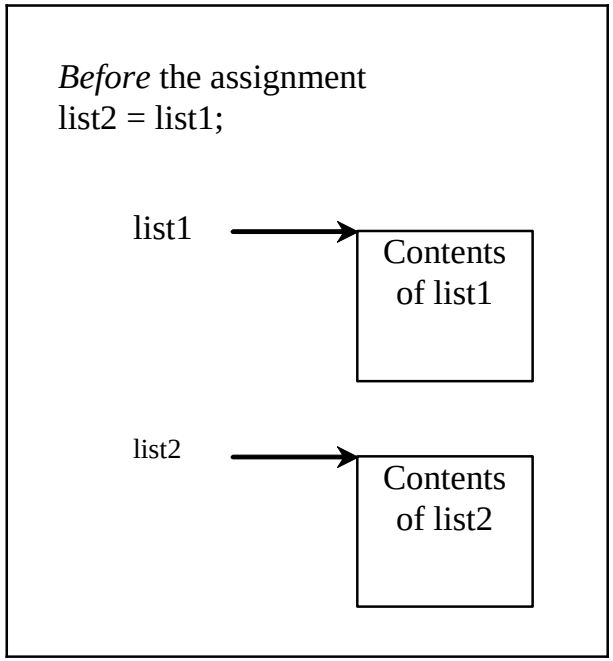
```
for (elementType value: arrayRefVar) {
      // Process the value
}
```

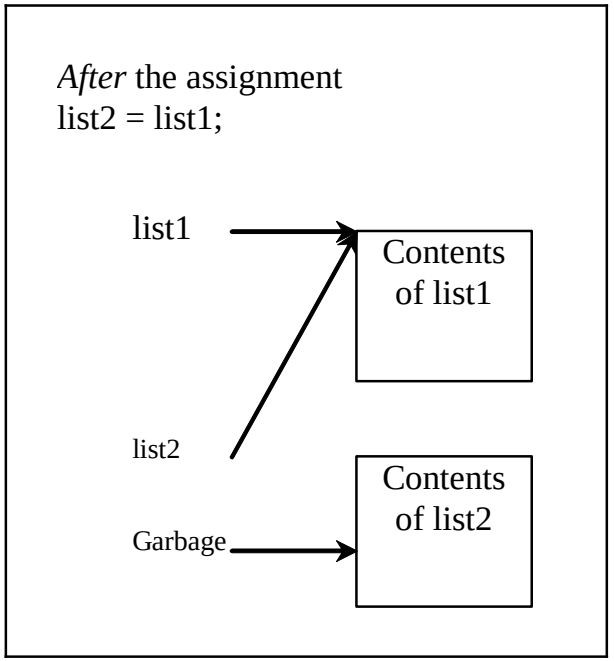
Note: You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.

Copying Arrays

- Often, in a program, you need to duplicate an array or a part of an array.
- **Using the assignment statement (=), you re-direct the pointer:**

```
list2 = list1;
```





- **You don't copy with "=" !**

Copying Arrays

- Using a loop:

```
int[] sourceArray={2, 3, 1, 5, 10};
int[] targetArray=new int[sourceArray.length];
for (int i = 0; i < sourceArray.length; i++)
   targetArray[i] = sourceArray[i];
```

The arraycopy Utility

```
System.arraycopy(sourceArray,
  src_pos, targetArray, tar_pos,
  length);
```

Example:

```
System.arraycopy(sourceArray, 0,
  targetArray, 0, sourceArray.length);
```

Passing Arrays to Methods

```
public static void printArray(int[] array) {
  for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
  }
}
```

```
Invoke the method
int[] list = {3, 1, 2, 6, 4, 2};
printArray(list);
```

OR

```
Invoke the method
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Anonymous array

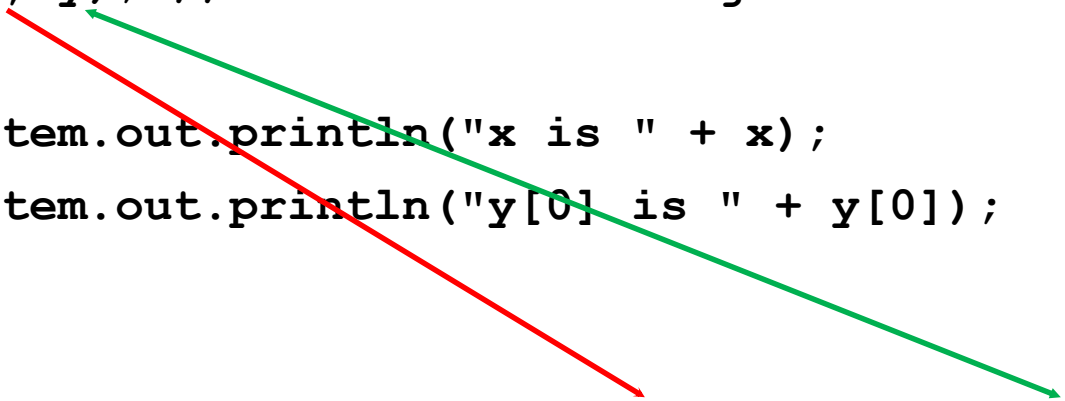
Pass By Value

Java uses ***pass by value*** to pass arguments to a method.

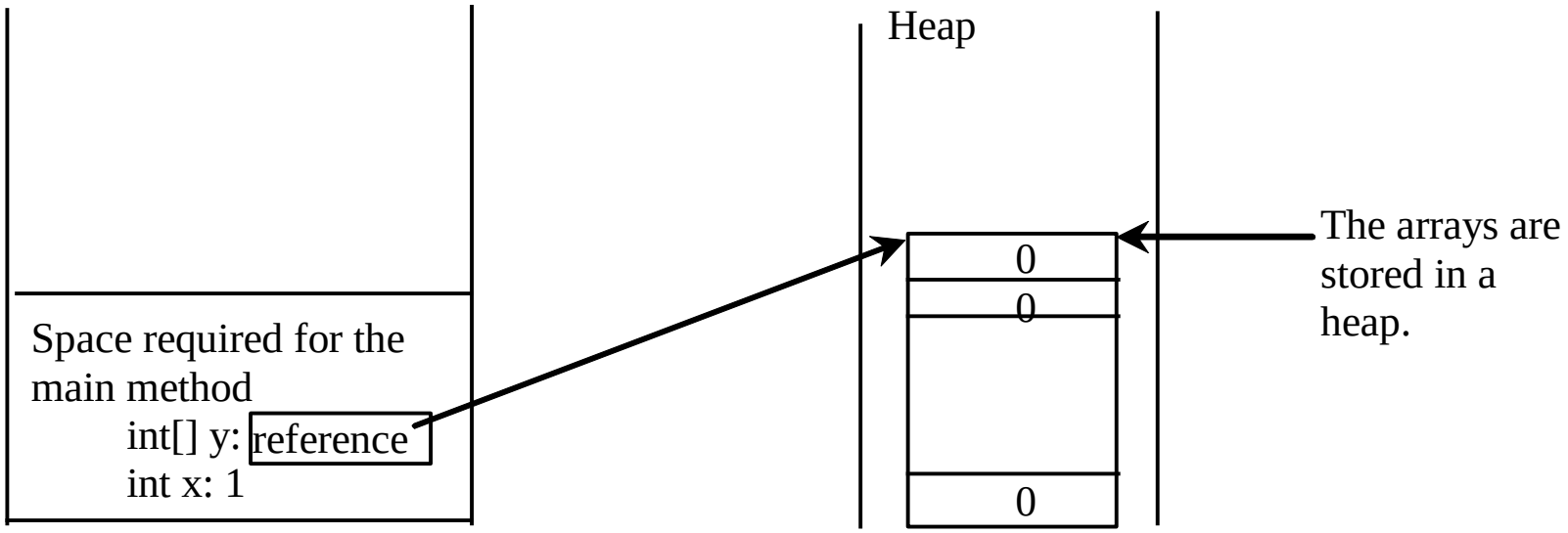
- For a parameter of a primitive type value, the actual value is passed.
  - Changing the value of the local parameter inside the method does not affect the value of the variable outside the method.
- For a parameter of an array type, the value of the parameter contains a reference to an array; this reference is passed to the method.
  - Any changes to the array that occur inside the method body will affect the original array that was passed as the argument.

Simple Example

```
public class Test {
  public static void main(String[] args) {
    int x = 1; // x represents an int value
    int[] y = new int[10]; // y represents an array of int values

    m(x, y); // Invoke m with arguments x and y

    System.out.println("x is " + x);          // x is 1
    System.out.println("y[0] is " + y[0]);    // y[0] is 5555
  }

  public static void m(int number, int[] numbers) {
    number = 1001; // Assign a new value to number
    numbers[0] = 5555; // Assign a new value to numbers[0]
  }
}
```

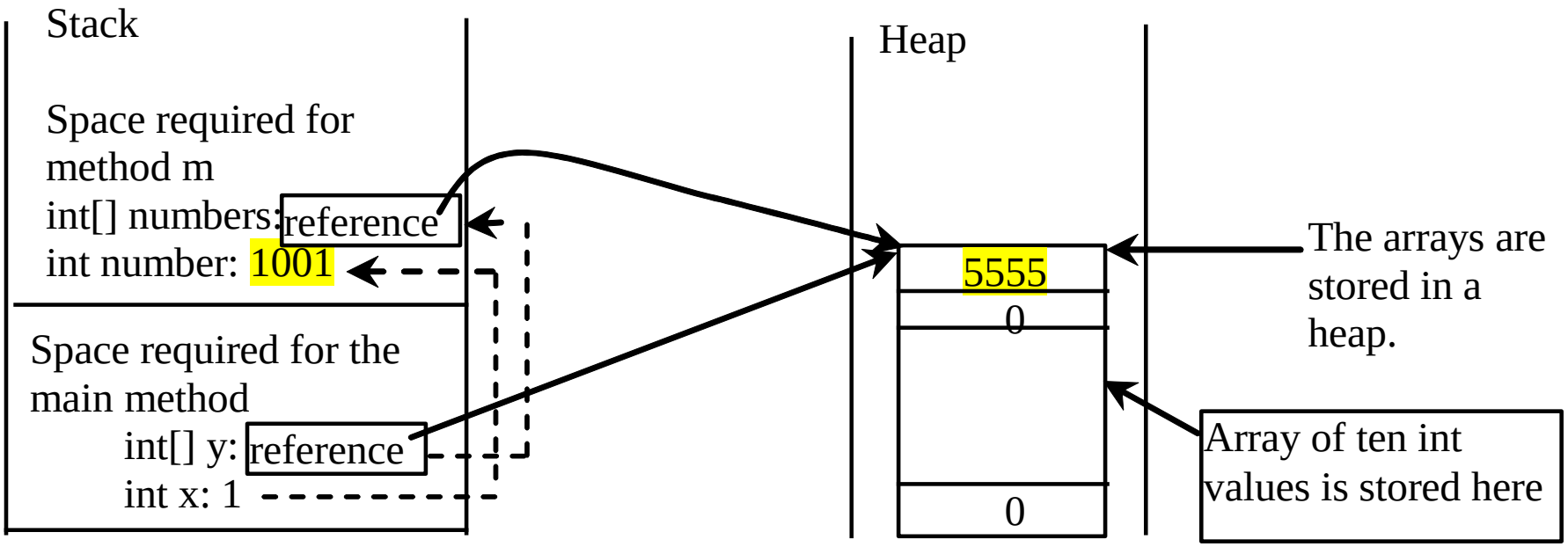


The Call Stack and Heap



The JVM stores the array in an area of memory, called *heap*, which is used for dynamic memory allocation where blocks of memory are allocated and freed in an arbitrary order.

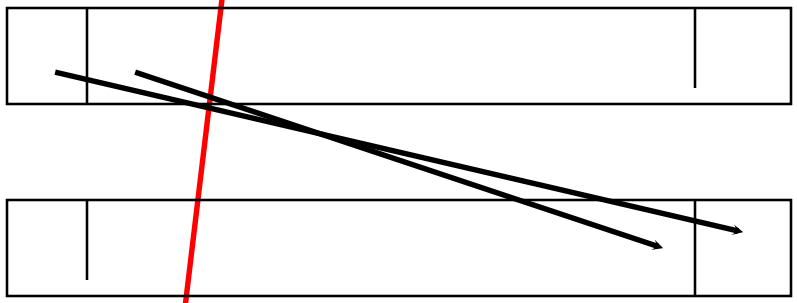
The Call Stack and Heap



When invoking m(x, y), the values of x and y are passed to number and numbers. Since y contains the reference value to the array, numbers now contains the same reference value to the same array.

Returning an Array from a Method

```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];

  for (int i = 0, j = result.length - 1;
       i < list.length; i++, j--) {
    result[j] = list[i];
  }

  return result;
}
```

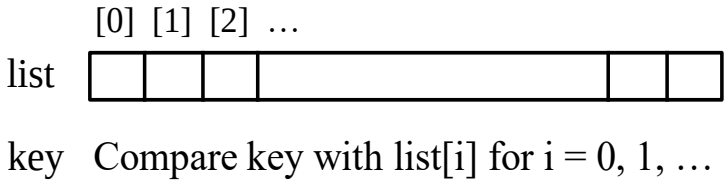



```
int[] list1 = new int[]{1, 2, 3, 4, 5, 6};
int[] list2 = reverse(list1);
```

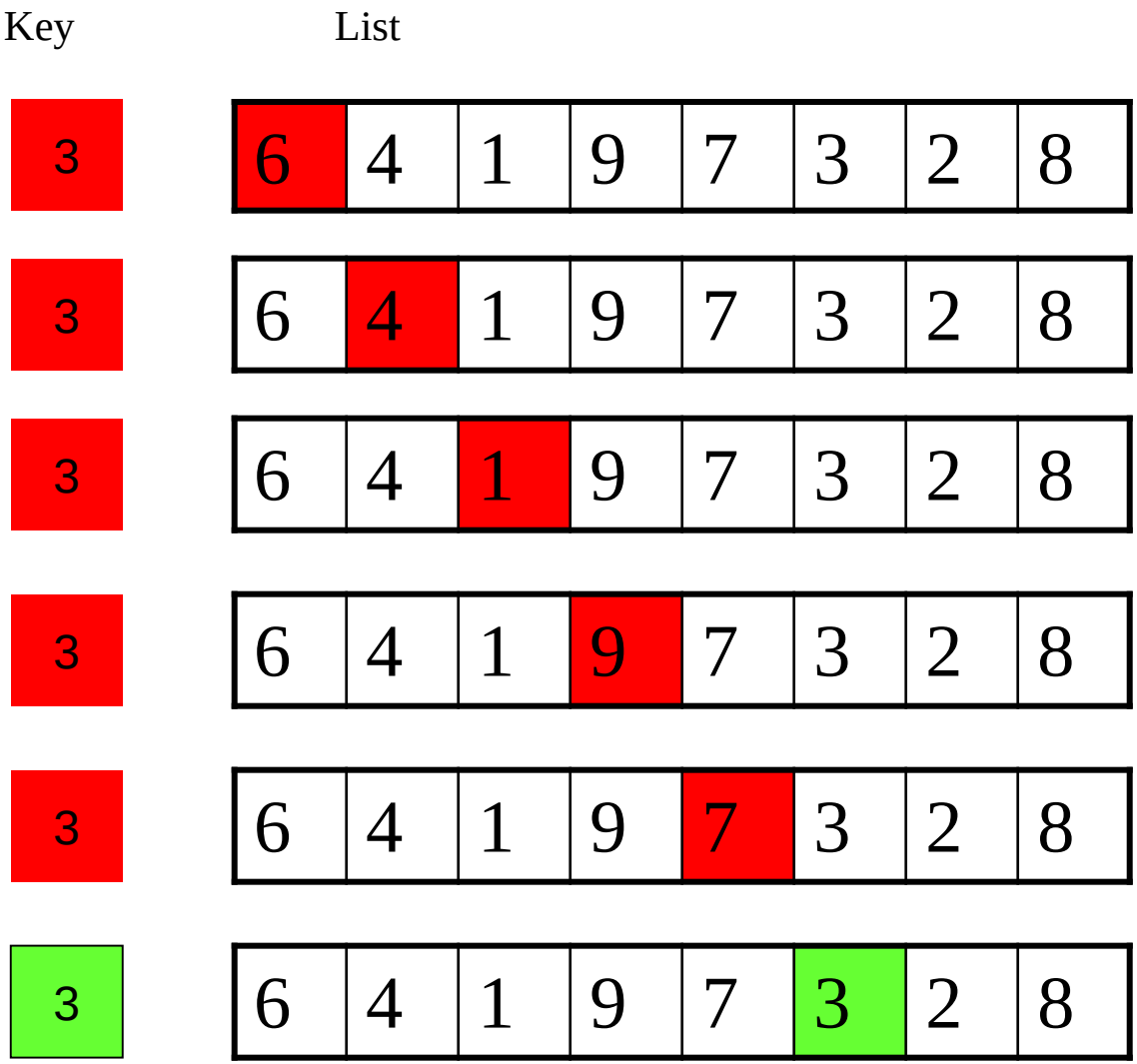
Searching Arrays

- Searching is the process of looking for a specific element in an array

```
public static int linearSearch(int[] list, int key)
```



Linear Search Example



From Idea to Solution

```
public static int linearSearch(int[] list, int key) {
  for (int i = 0; i < list.length; i++)
    if (key == list[i])
      return i;
  return -1;
}
```

```
int[] list = {6,4,1,9,7,3,2,8};
int i = linearSearch(list, 3);  // returns 5
int j = linearSearch(list, -4); // returns -1
int k = linearSearch(list, 4); // returns 1
```

Binary Search

- If an array is already ordered, then it is cheaper to find an element
  - Assume that the array is in ascending order. e.g., 1, 2, 3, 4, 6, 7, 8, 9

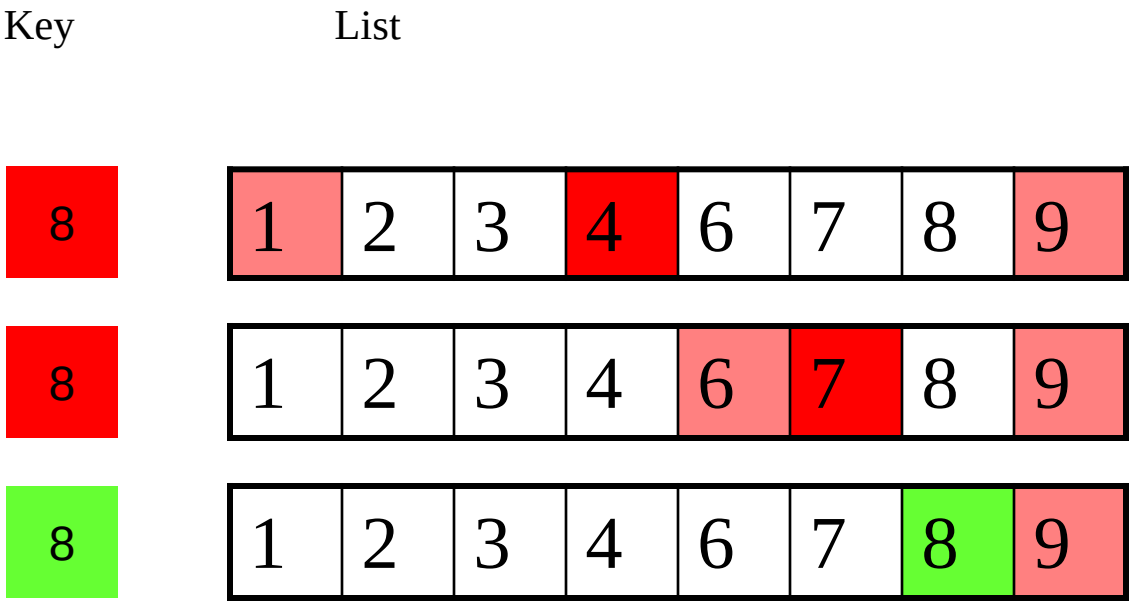
The binary search first compares the key (e.g., 8) with the element in the middle of the array.

Binary Search

Consider the following three cases:

- If the key is less than the middle element, you only need to search the key in the **first half** of the array.
- If the key is equal to the middle element, the search ends with a match.
- If the key is greater than the middle element, you only need to search the key in the second half of the array.

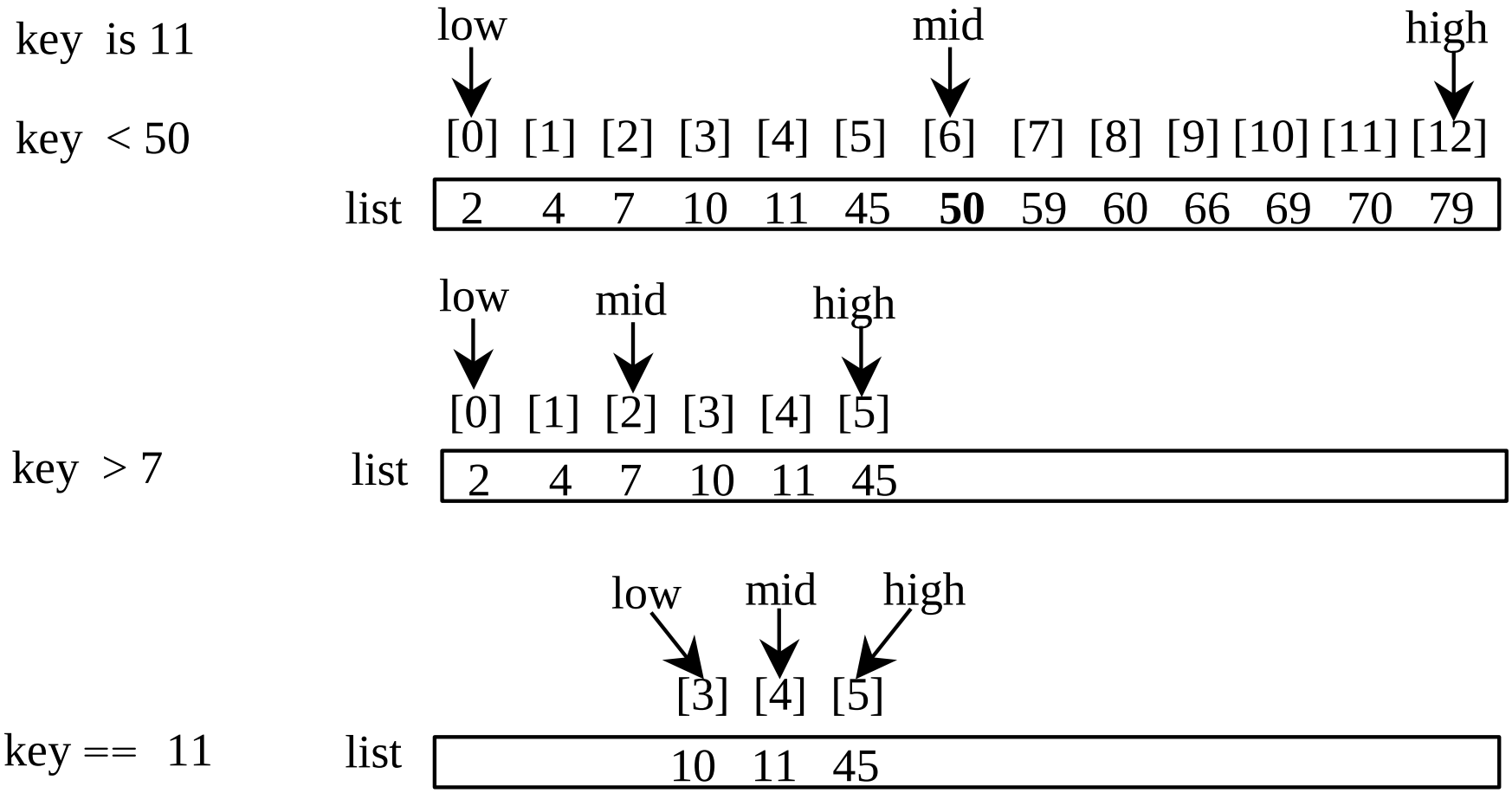
Binary Search



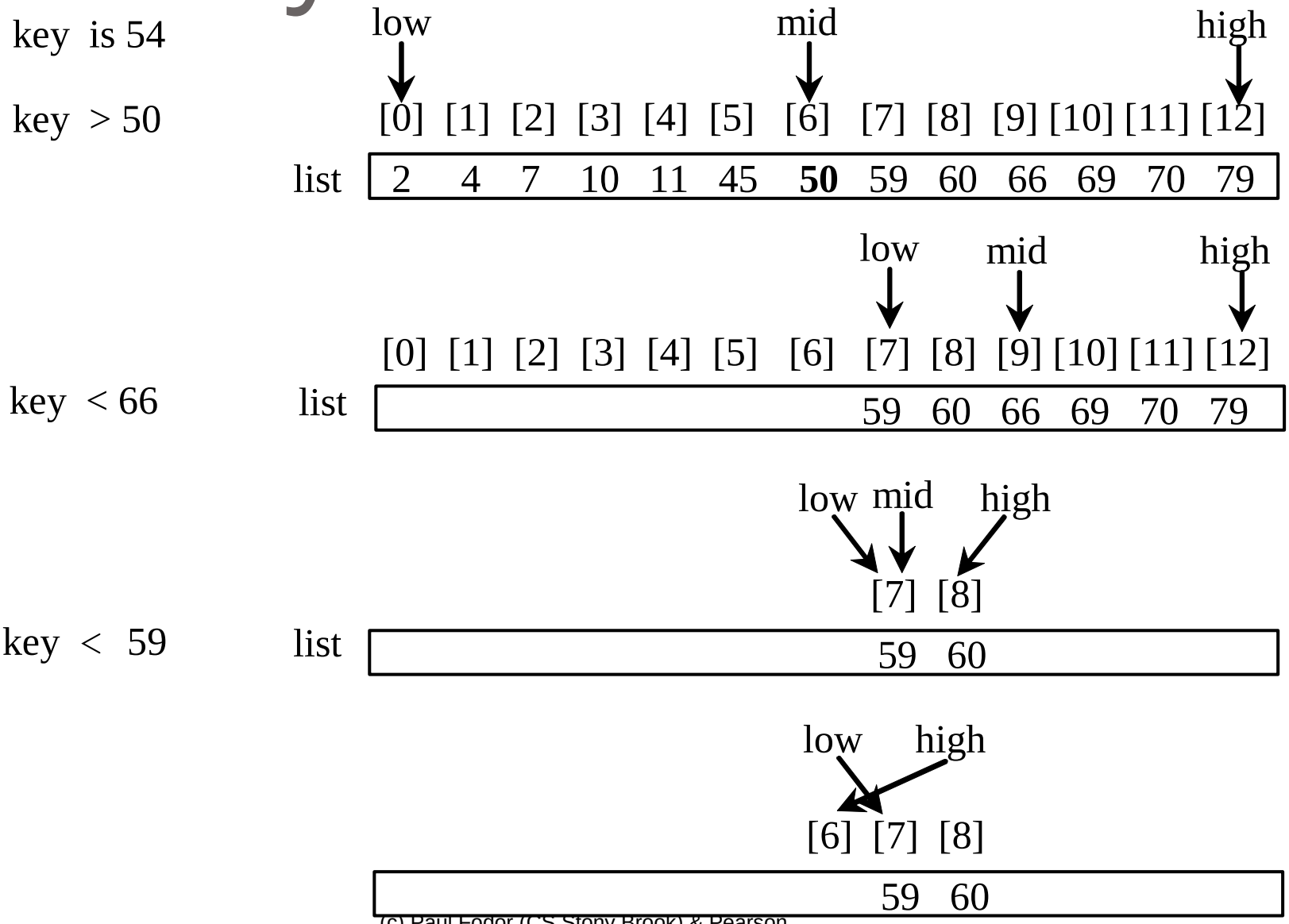
From Idea to Solution

```
/** Use binary search to find the key in the list */
public static int binarySearch(int[] list, int key) {
  int low = 0;
  int high = list.length - 1;
  while (high >= low) {
    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key == list[mid])
      return mid;
    else
      low = mid + 1;
  }
  return -1 - low;
}
```

Binary Search



Binary Search



The Arrays.binarySearch Method

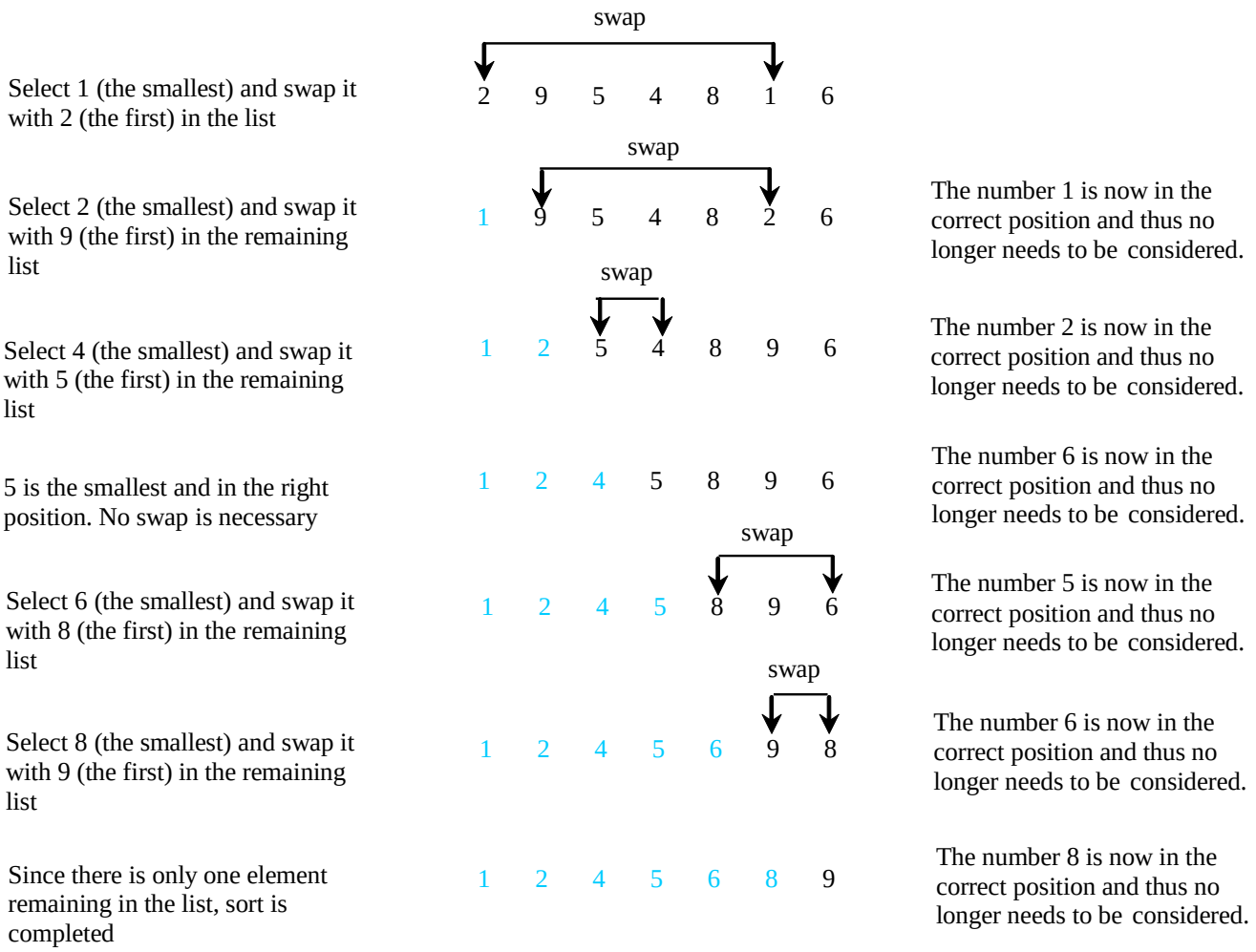
- Java provides several overloaded binarySearch methods for searching a key in an array of int, double, char, short, long, and float in the java.util.Arrays class.

```
int[] list = {1, 2, 3, 4, 6, 7, 8, 9};
System.out.println("Index is " +
  java.util.Arrays.binarySearch(list, 11));
```

Return is 4

Selection Sort

Selection sort finds the smallest number in the list and places it first. It then finds the smallest number in the remaining list and places it second, and so on until the list contains only a single number. Sort the list {2, 9, 5, 4, 8, 1, 6} using selection sort would be:



From Idea to Solution

```
for (int i = 0; i < list.length; i++) {
  select the smallest element in list[i..listSize-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration apply on list[i+1..listSize-1]
}
```

From Idea to Solution

```
for (int i = 0; i < list.length; i++) {
  select the smallest element in list[i..listSize-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration apply on list[i+1..listSize-1]
}
```

Expand

```
double currentMin = list[i];
int currentMinIndex = i;
for (int j = i+1; j < list.length; j++) {
  if (currentMin > list[j]) {
    currentMin = list[j];
    currentMinIndex = j;
  }
}
```

Wrap it in a Method

```
/** The method for sorting numbers */
public static void selectionSort(double[] list) {
    for (int i = 0; i < list.length; i++) {
      // Find the minimum in the list[i..list.length-1]
      double currentMin = list[i];
      int currentMinIndex = i;
      for (int j = i + 1; j < list.length; j++) {
        if (currentMin > list[j]) {
          currentMin = list[j];
          currentMinIndex = j;
        }
      }
      // Swap list[i] with list[currentMinIndex] if necessary;
      if (currentMinIndex != i) {
        list[currentMinIndex] = list[i];
        list[i] = currentMin;
      }
    }
  }
```

Insertion Sort

The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.

```
int[] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted
```

Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 to the sublist.

Step2: The sorted sublist is {2, 9}. Insert 5 to the sublist.

Step 3: The sorted sublist is {2, 5, 9}. Insert 4 to the sublist.

Step 4: The sorted sublist is {2, 4, 5, 9}. Insert 8 to the sublist.

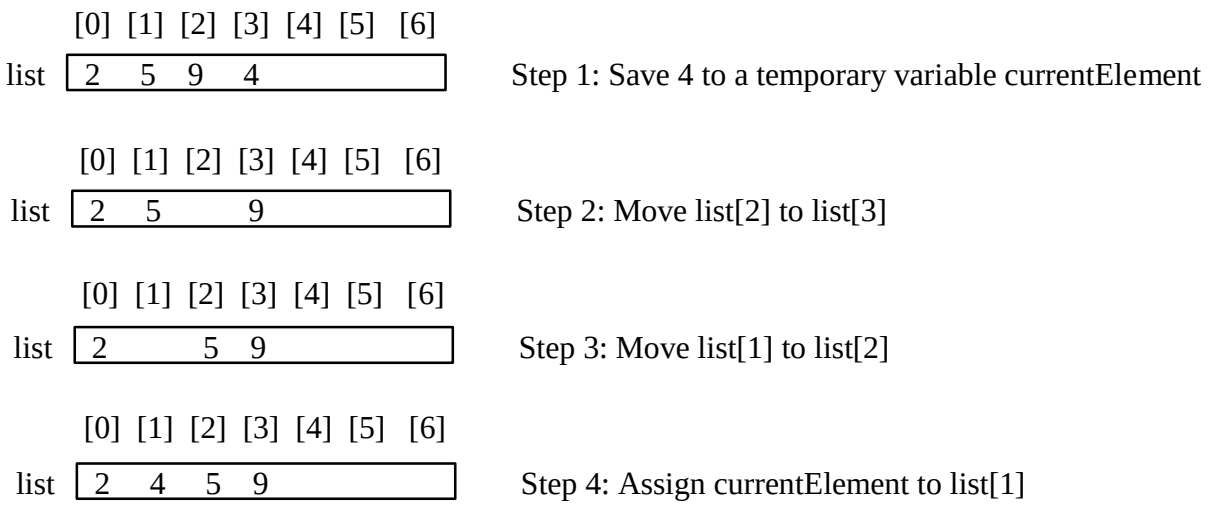
Step 5: The sorted sublist is {2, 4, 5, 8, 9}. Insert 1 to the sublist.

Step 6: The sorted sublist is {1, 2, 4, 5, 8, 9}. Insert 6 to the sublist.

Step 7: The entire list is now sorted

How to Insert?

The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.



From Idea to Solution

```
for (int i = 1; 1; i < list,length; i++) {
  insert list[i] into a sorted sublist list[0..i-1] so that
          list[0..i] is sorted
}
```

```
public static void insertionSort(double[] list){
        for(int i=1; i<list.length; i++){
                //insert list[i] in the sorted sublist list[0,i-1]
                // find the position
                int pos;
                for(pos=0; pos<i; pos++)
                        if(list[pos]>list[i])
                                break;
                double temp = list[i];
                // shift right elements from pos to i-1
                for(int j=i; j>pos; j--)
                        list[j] = list[j-1];
                list[pos] = temp;
        }
}
public static void main(String[] args) {
        double[] list1 = new double[]{8, 2, 3, 4};
        insertionSort(list1);
        print(list1);
}
public static void print(double[] list){
        for(double x:list)    System.out.print(x + " ");
}
```

The Arrays.sort Method

- Since sorting is frequently used in programming, Java provides several overloaded sort methods for sorting an array of int, double, char, short, long, and float in the java.util.Arrays class. For example, the following code sorts an array of numbers and an array of characters.

```
double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};
java.util.Arrays.sort(numbers);

char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};
java.util.Arrays.sort(chars);
```

Declaring Variables of Two-dimensional Arrays and Creating Two-dimensional Arrays

```
int[][] matrix = new int[10][10];
```

or

```
int matrix[][] = new int[10][10];
```

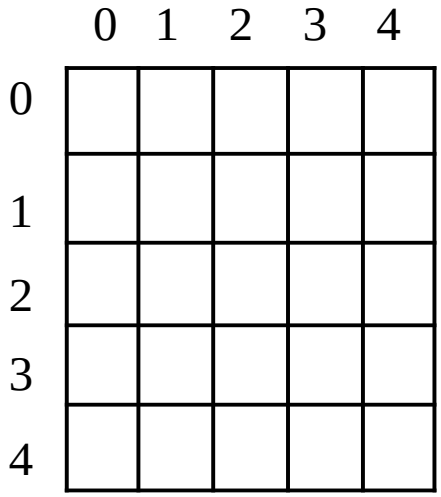
- **Indexed variables:**

```
matrix[0][0] = 3;
```

- **Length:**

```
for (int i = 0; i < matrix.length; i++)
  for (int j = 0; j < matrix[i].length; j++)
    matrix[i][j] = (int)(Math.random() * 1000);
```

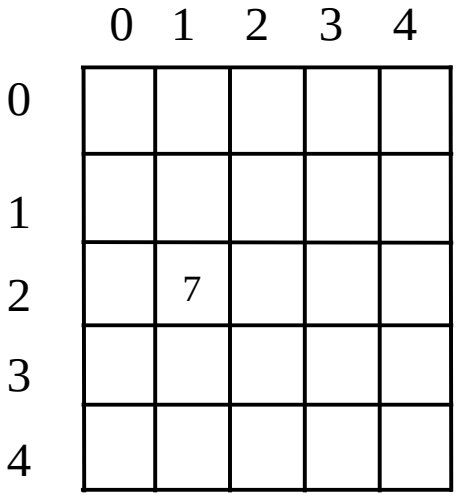
Two-dimensional Array Lengths



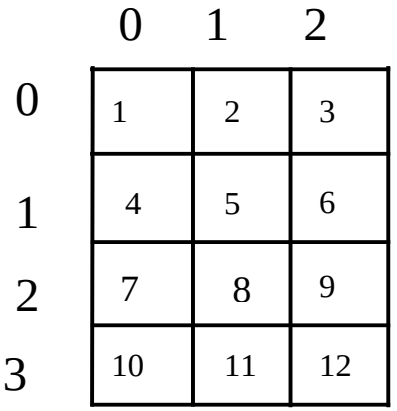

```
matrix = new int[5][5];
```

matrix.length? 5

matrix[0].length? 5




```
matrix[2][1] = 7;
```



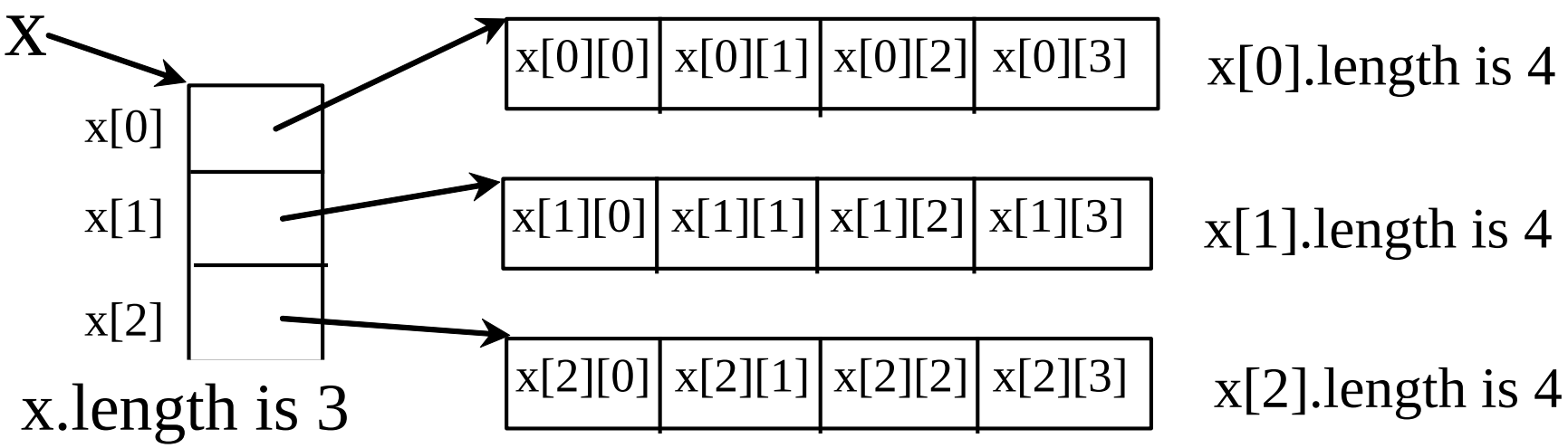

```
int[][] array = {
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9},
   {10, 11, 12}
};
```

array.length? 4

array[0].length? 3

Lengths of Two-dimensional Arrays

```
int[][] x = new int[3][4];
```



Declaring, Creating, and Initializing Using Shorthand Notations

You can also use an array initializer to declare, create and initialize a two-dimensional array. For example,

```
int[][] array = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

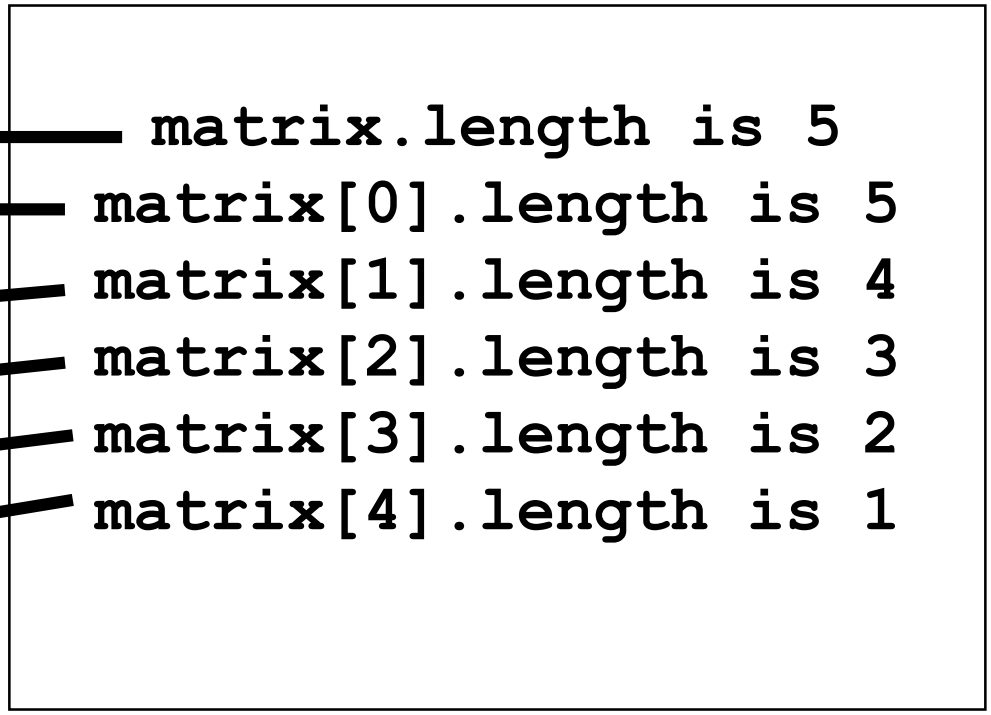
Same as

```
int[][] array = new int[4][3];
array[0][0] = 1; array[0][1] = 2; array[0][2] = 3;
array[1][0] = 4; array[1][1] = 5; array[1][2] = 6;
array[2][0] = 7; array[2][1] = 8; array[2][2] = 9;
array[3][0] = 10; array[3][1] = 11; array[3][2] = 12;
```

Ragged Arrays

- A ragged array is an array where rows can have different lengths:

```
int[][] matrix = {
  {1, 2, 3, 4, 5},
  {2, 3, 4, 5},
  {3, 4, 5},
  {4, 5},
  {5}
};
```



Ragged Arrays

Storing a ragged array:

```
int[][] triangleArray = {
   {1, 2, 3, 4, 5},
   {1, 2, 3, 4},
   {1, 2, 3},
   {1, 2},
   {1}
};
```