Java Basics and Structure

CIS 2103 | Object Oriented Programming

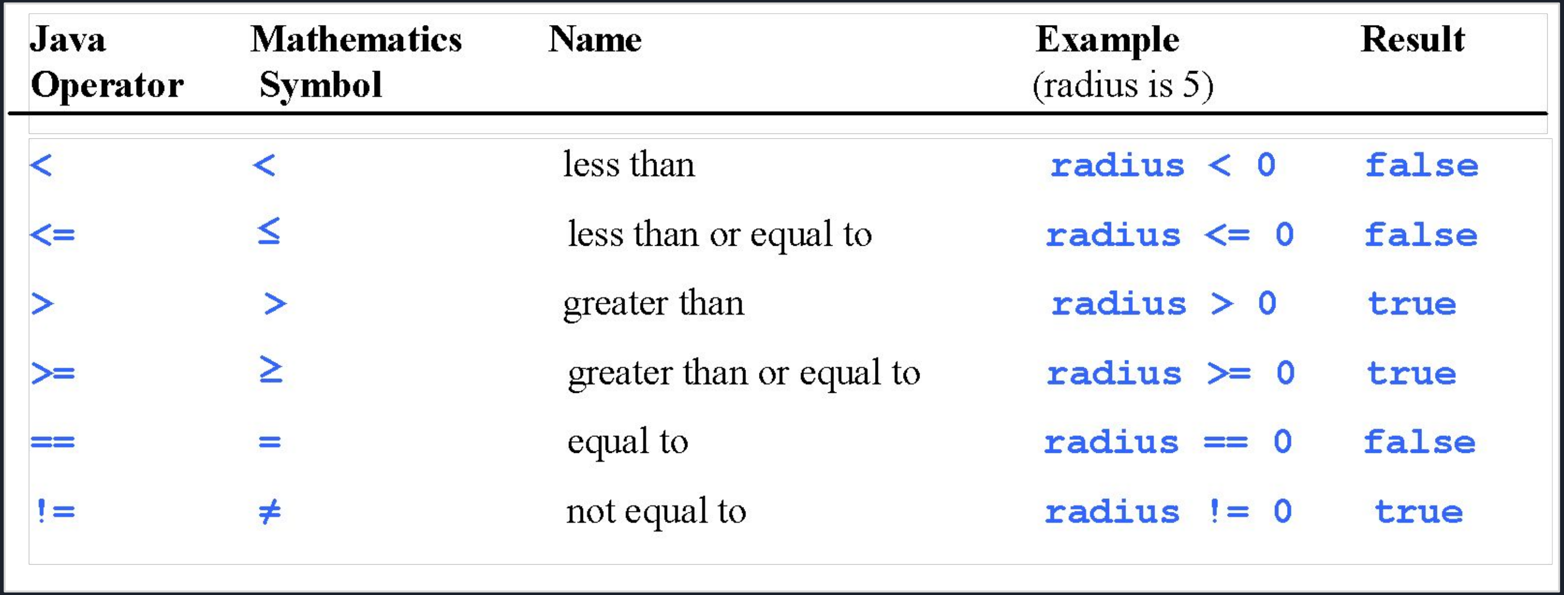
Control Structures

The `boolean` Type and Operators

Often in a program you need to compare two values, such as whether i is greater than j. Java provides six comparison operators (also known as relational operators) that can be used to compare two values. The result of the comparison is a Boolean value: true or false.

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

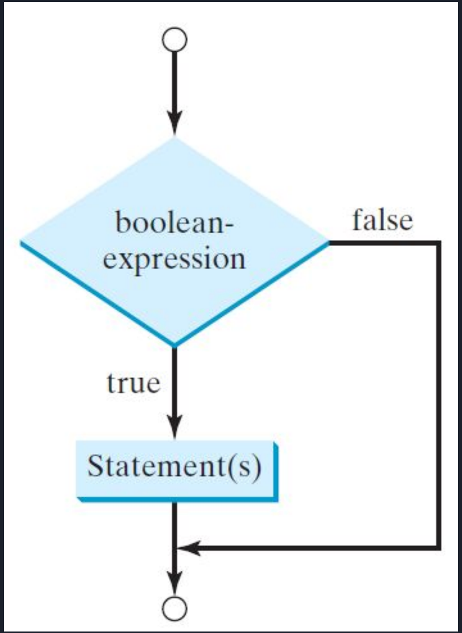
Relational Operators



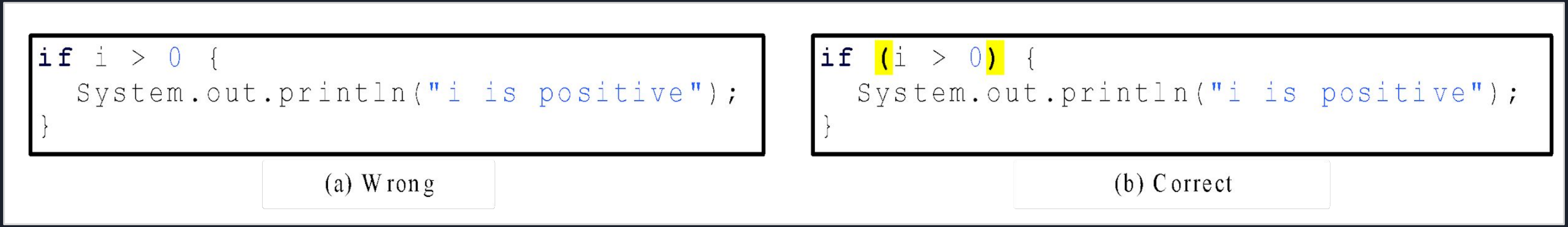
Java Operator	Mathematics Symbol	Name	Example (radius is 5)	Result
`<`	$<$	less than	`radius < 0`	`false`
`<=`	$\leq$	less than or equal to	`radius <= 0`	`false`
`>`	$>$	greater than	`radius > 0`	`true`
`>=`	$\geq$	greater than or equal to	`radius >= 0`	`true`
`==`	$=$	equal to	`radius == 0`	`false`
`!=`	$\neq$	not equal to	`radius != 0`	`true`

One-way `if` Statements

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



NOTE

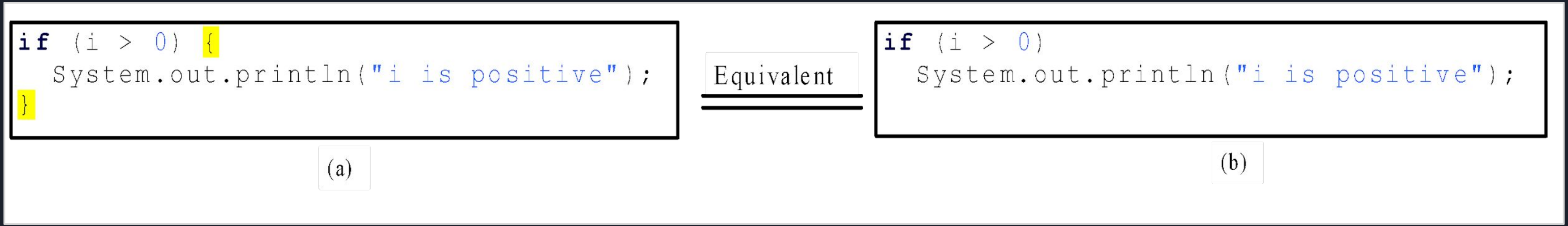


```
if i > 0 {
  System.out.println("i is positive");
}
```

(a) Wrong

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(b) Correct



```
if (i > 0) {
  System.out.println("i is positive");
}
```

(a)

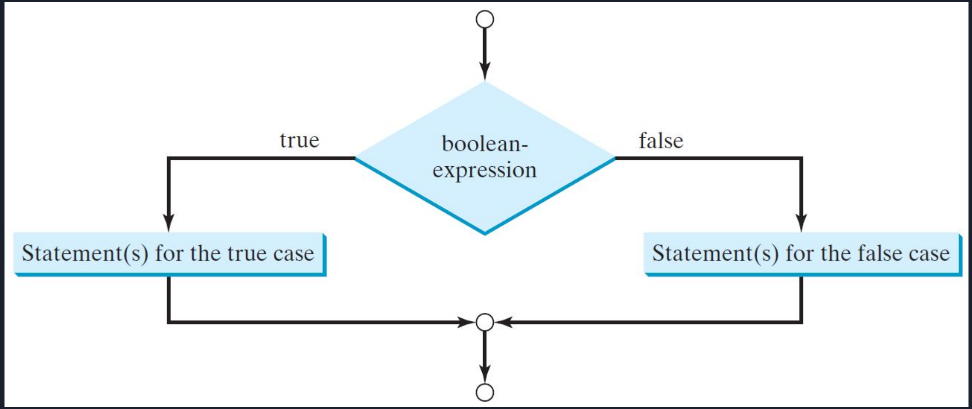
Equivalent

```
if (i > 0)
  System.out.println("i is positive");
```

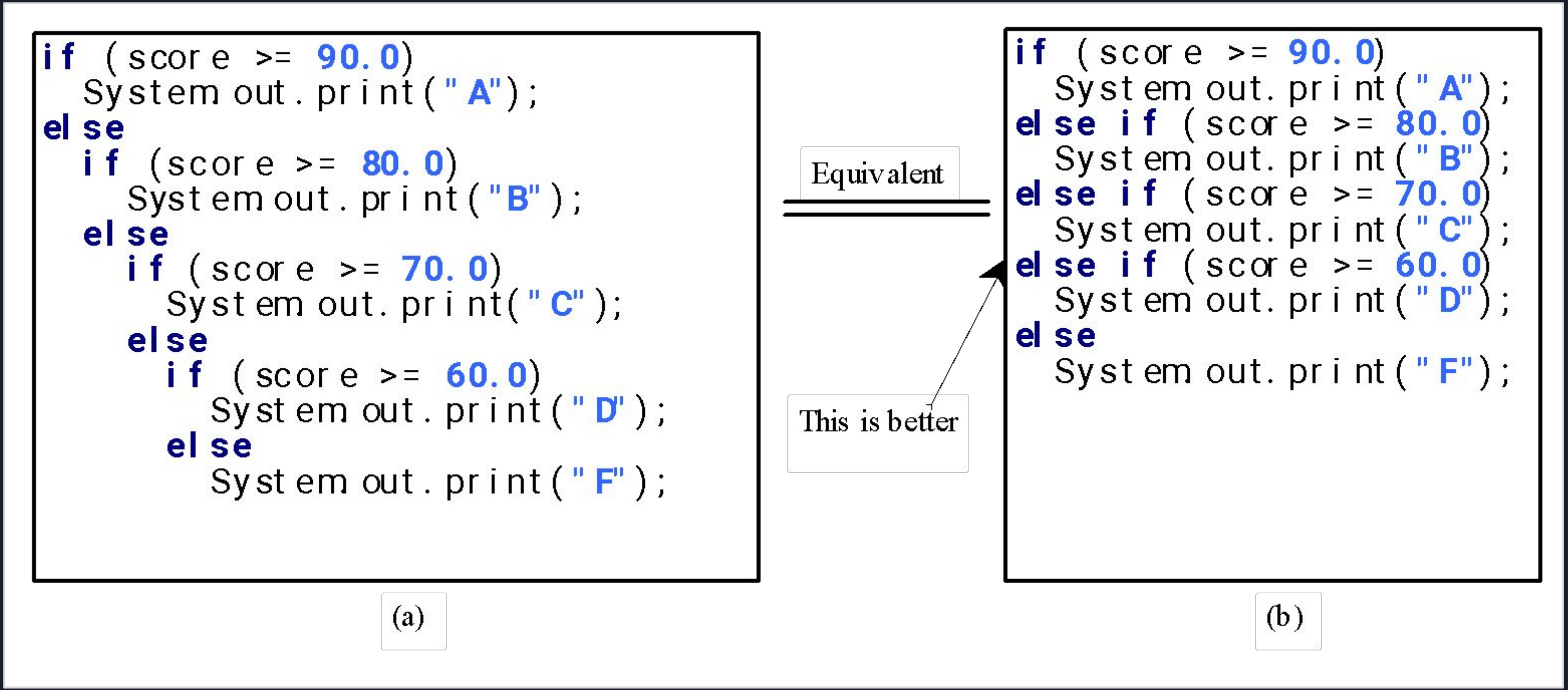
(b)

The Two-way `if` Statement

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


true
boolean-
expression
false
Statement(s) for the true case
Statement(s) for the false case

Multiple Alternative if Statements



```
if (score >= 90.0)
  System.out.print("A");
else
  if (score >= 80.0)
    System.out.print("B");
  else
    if (score >= 70.0)
      System.out.print("C");
    else
      if (score >= 60.0)
        System.out.print("D");
      else
        System.out.print("F");
```

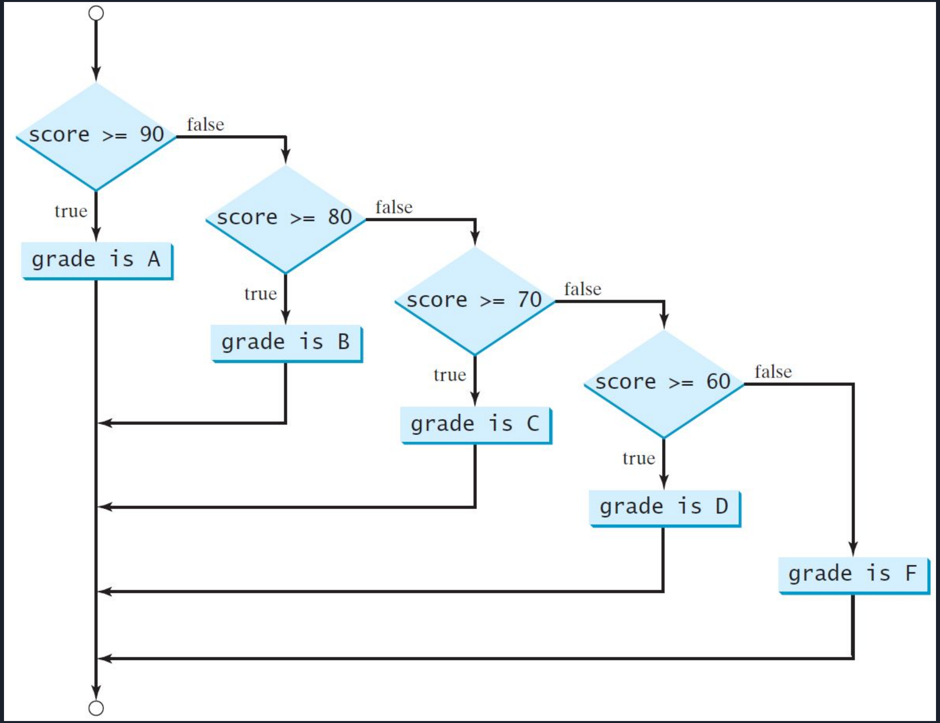
(a)

Equivalent
This is better

```
if (score >= 90.0)
  System.out.print("A");
else if (score >= 80.0)
  System.out.print("B");
else if (score >= 70.0)
  System.out.print("C");
else if (score >= 60.0)
  System.out.print("D");
else
  System.out.print("F");
```

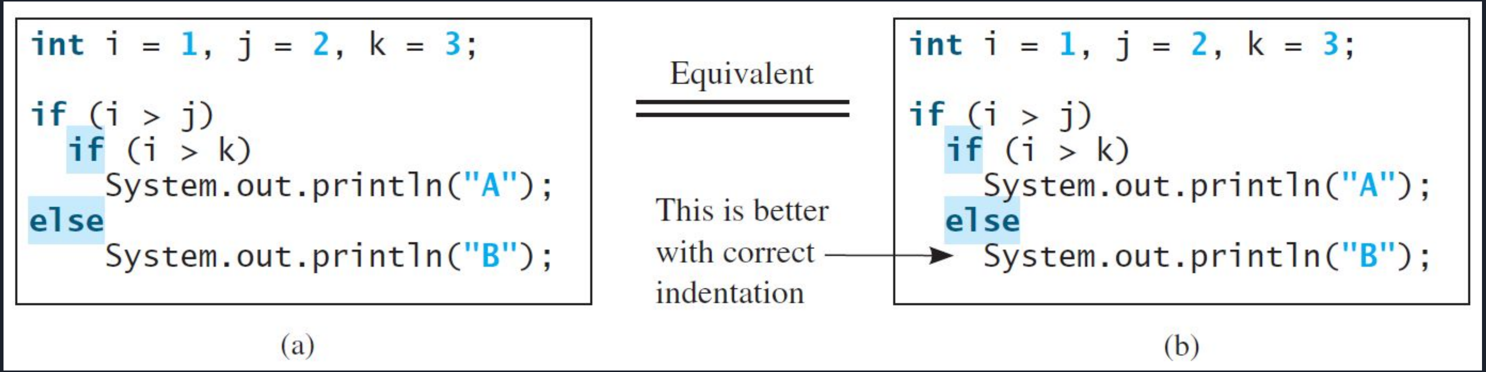
(b)

Multi-Way if-else Statements



Note

- The <u>else</u> clause matches the most recent <u>if</u> clause in the same block.



```
int i = 1, j = 2, k = 3;

if (i > j)
  if (i > k)
    System.out.println("A");
else
    System.out.println("B");
```

(a)

Equivalent

This is better with correct indentation

```
int i = 1, j = 2, k = 3;

if (i > j)
  if (i > k)
    System.out.println("A");
  else
    System.out.println("B");
```

(b)

Note

Nothing is printed from the preceding statement. To force the else clause to match the first if clause, you must add a pair of braces:

```
int i = 1;
 int j = 2;
 int k = 3;
 if (i > j) {
   if (i > k)
     System.out.println("A");
 }
 else
   System.out.println("B");
```

This statement prints B.

Common Errors

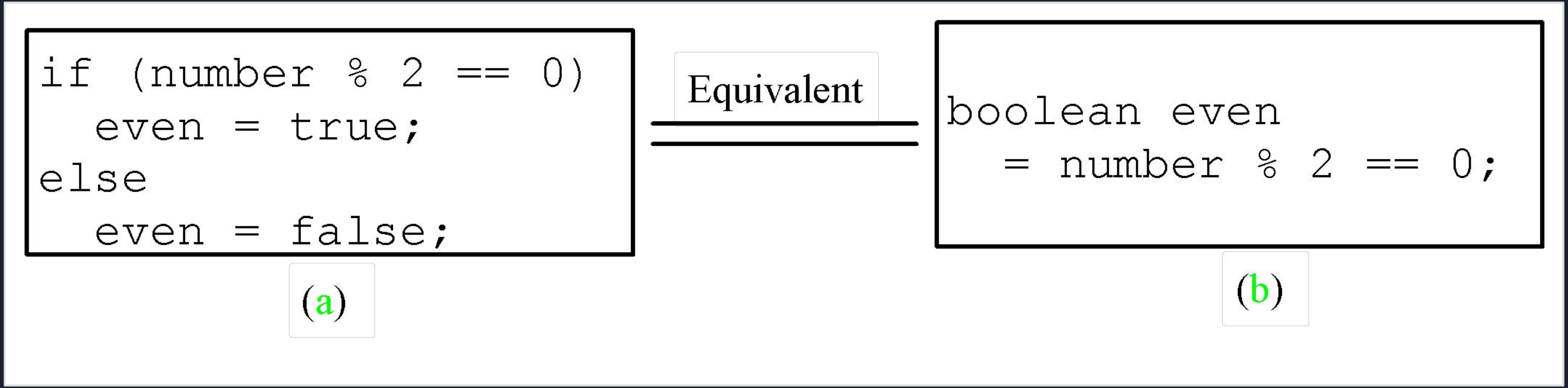
Adding a semicolon at the end of an if clause is a common mistake.

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

This mistake is hard to find, because it is not a compilation error or a runtime error, it is a logic error.

This error often occurs when you use the next-line block style.

TIP



```
if (number % 2 == 0)
  even = true;
else
  even = false;
```

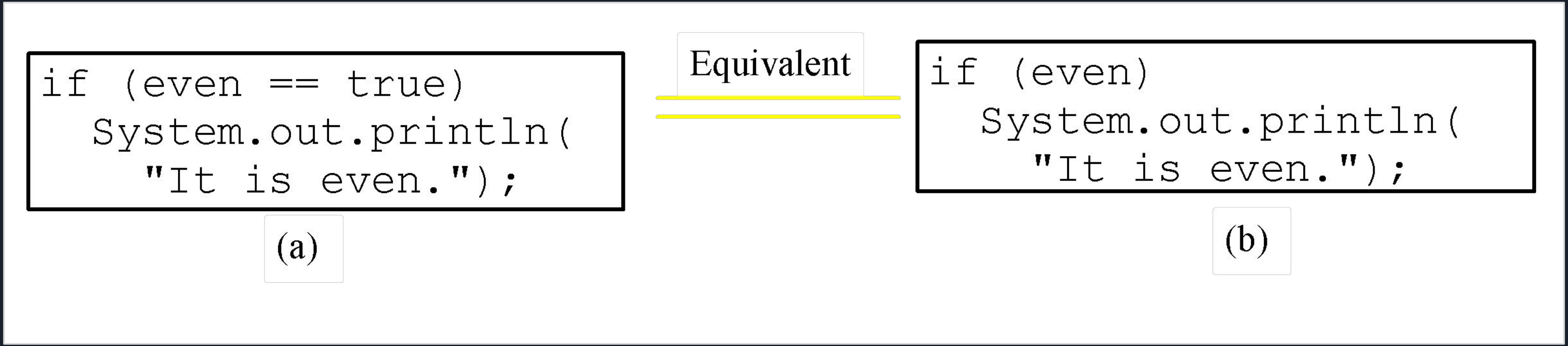
(a)

Equivalent

```
boolean even
  = number % 2 == 0;
```

(b)

CAUTION



```
if (even == true)
  System.out.println(
    "It is even.");
```

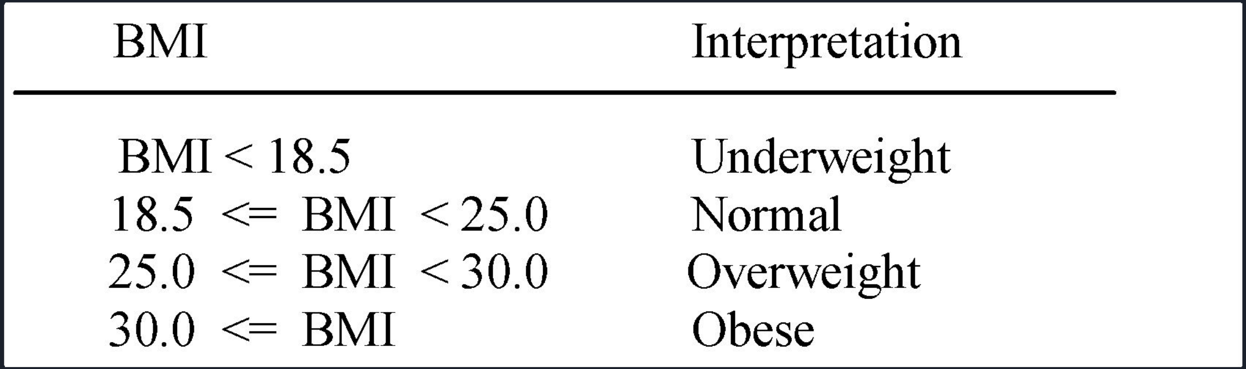
```
if (even)
  System.out.println(
    "It is even.");
```

Demo: A Simple Math Learning Tool

- This example creates a program to let a first grader practice additions. The program randomly generates two single-digit integers number1 and number2 and displays a question such as “What is 7 + 9?” to the student. After the student types the answer, the program displays a message to indicate whether the answer is correct or incorrect.

Example: Body Mass Index

Body Mass Index (BMI) is a measure of health on weight. It can be calculated by taking your weight in kilograms and dividing by the square of your height in meters. The interpretation of BMI for people 16 years or older is as follows:



BMI	Interpretation
BMI < 18.5	Underweight
18.5 <= BMI < 25.0	Normal
25.0 <= BMI < 30.0	Overweight
30.0 <= BMI	Obese

Logical Operators

Operator	Name	Description
!	not	logical negation
&&	and	logical conjunction
| |	or	logical disjunction
^	exclusive or	logical exclusion

Truth Table for Operator !

p	!p	Example (assume age = 24, weight = 140)
true	**false**	!(age > 18) is false, because (age > 18) is true.
false	**true**	!(weight == 150) is true, because (weight == 150) is false.

Truth Table for Operator &&

p_1	p_2	p_1 && p_2	Example (assume age = 24, weight = 140)
false	false	false	(age <= 18) && (weight < 140) is false, because (age > 18) and (weight <= 140) are both false.
false	true	false	
true	false	false	(age > 18) && (weight > 140) is false, because (weight > 140) is false.
true	true	true	(age > 18) && (weight >= 140) is true, because both (age > 18) and (weight >= 140) are true.

Truth Table for Operator | |

p_1	p_2	p_1 || p_2	Example (assume age = 24, weight = 140)
false	false	false	
false	true	true	(age > 34) || (weight <= 140) is true, because (age > 34) is false, but (weight <= 140) is true.
true	false	true	(age > 14) || (weight >= 150) is false, because (age > 14) is true.
true	true	true	

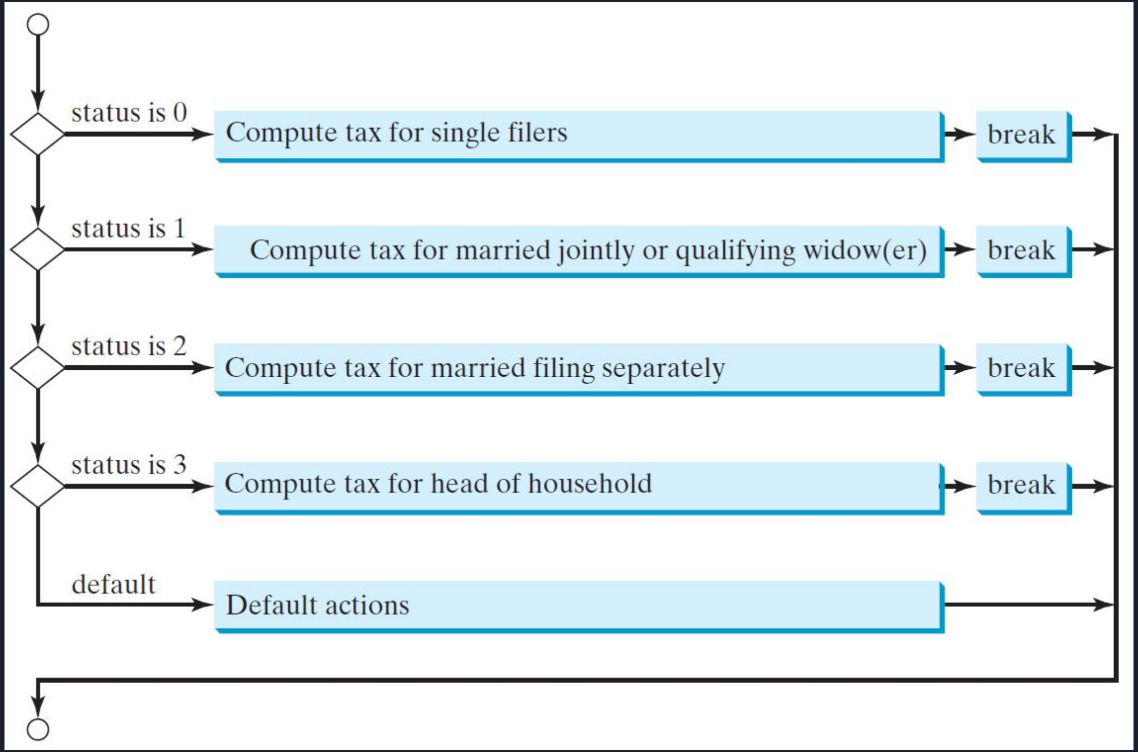
Truth Table for Operator ^

p_1	p_2	p_1 ^ p_2	Example (assume age = 24, weight = 140)
false	**false**	**false**	(age > 34) ^ (weight > 140) is true, because (age > 34) is false and (weight > 140) is false.
false	**true**	**true**	(age > 34) ^ (weight >= 140) is true, because (age > 34) is false but (weight >= 140) is true.
true	**false**	**true**	(age > 14) ^ (weight > 140) is true, because (age > 14) is true and (weight > 140) is false.
true	**true**	**false**	

`switch` Statements

```
switch (status) {
  case 0:  compute taxes for single filers;
           break;
  case 1:  compute taxes for married file jointly;
           break;
  case 2:  compute taxes for married file separately;
           break;
  case 3:  compute taxes for head of household;
           break;
  default: System.out.println("Errors: invalid status");
           System.exit(1);
}
```

`switch` Statement Flow Chart


status is 0
Compute tax for single filers
break
status is 1
Compute tax for married jointly or qualifying widow(er)
break
status is 2
Compute tax for married filing separately
break
status is 3
Compute tax for head of household
break
default
Default actions

Conditional Expressions

```
if (x > 0)
  y = 1
else
  y = -1;
```

is equivalent to

```
y = (x > 0) ? 1 : -1;
(boolean-expression) ? expression1 : expression2
```

Ternary operator
Binary operator
Unary operator

Conditional Operator

```
boolean-expression ? exp1 : exp2
```

Operator Precedence

- `var++, var--`
- `+, - (Unary plus and minus), ++var,--var`
- `(type) Casting`
- `! (Not)`
- `*, /, % (Multiplication, division, and remainder)`
- `+, - (Binary addition and subtraction)`
- `<, <=, >, >= (Relational operators)`
- `==, !=; (Equality)`
- `^ (Exclusive OR)`
- `&& (Conditional AND) Short-circuit AND`
- `|| (Conditional OR) Short-circuit OR`
- `=, +=, -=, *=, /=, %= (Assignment operator)`

Operator Precedence and Associativity

- The expression in the parentheses is evaluated first. (Parentheses can be nested, in which case the expression in the inner parentheses is executed first.) When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule.
- If operators with the same precedence are next to each other, their associativity determines the order of evaluation. All binary operators except assignment operators are left-associative.

Operator Associativity

When two operators with the same precedence are evaluated, the ***associativity*** of the operators determines the order of evaluation. All binary operators except assignment operators are left-associative.

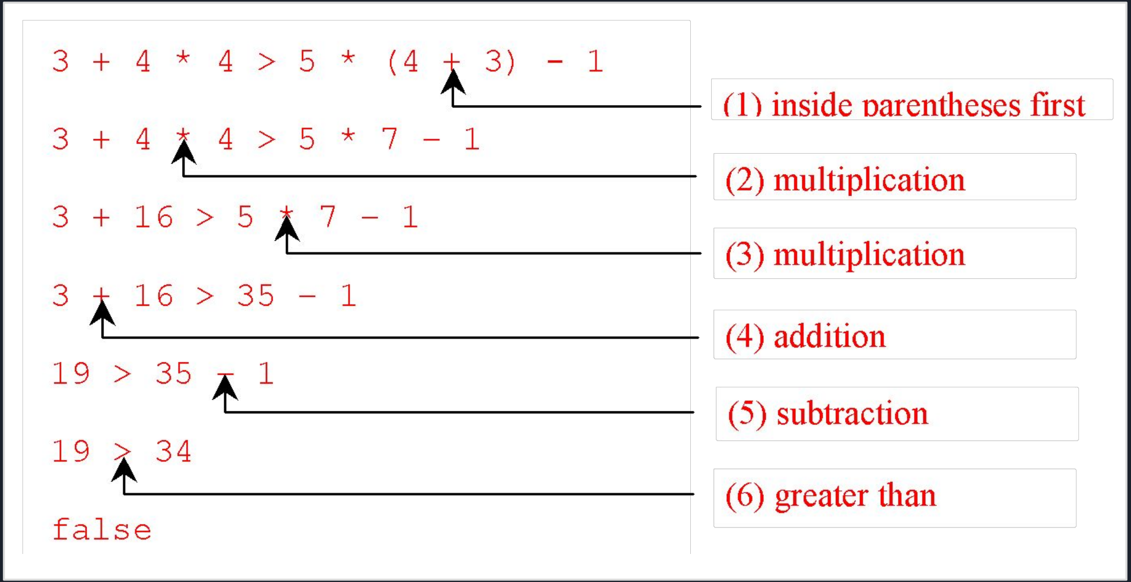
a – b + c – d is equivalent to ((a – b) + c) – d

Assignment operators are right-associative. Therefore, the expression

a = b += c = 5 is equivalent to a = (b += (c = 5))

Example

- Applying the operator precedence and associativity rule, the expression 3 + 4 * 4 > 5 * (4 + 3) - 1 is evaluated as follows:



```
3 + 4 * 4 > 5 * (4 + 3) - 1
3 + 4 * 4 > 5 * 7 - 1
3 + 16 > 5 * 7 - 1
3 + 16 > 35 - 1
19 > 35 - 1
19 > 34
false
```

Looping

Motivations

Suppose that you need to print a string (e.g., "Welcome to Java!") a hundred times. It would be tedious to have to write the following statement a hundred times:

```
System.out.println("Welcome to Java!");
```

So, how do you solve this problem?

Opening Problem

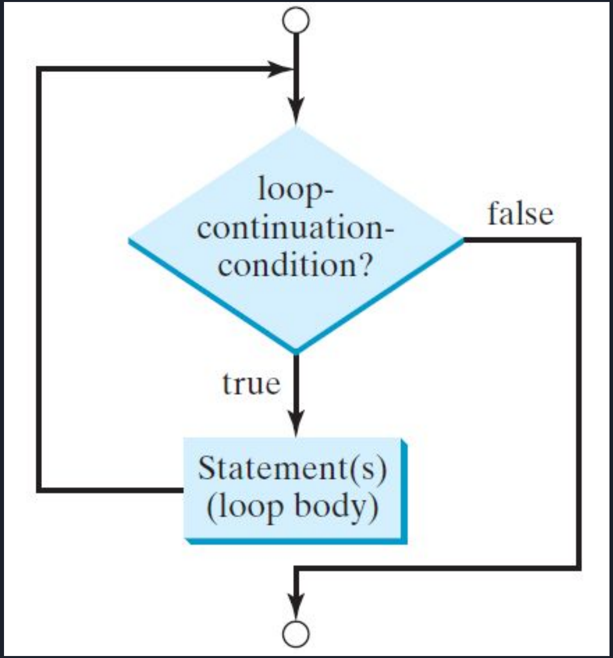
```
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");

…

…

…
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
```

Introducing while Loops

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

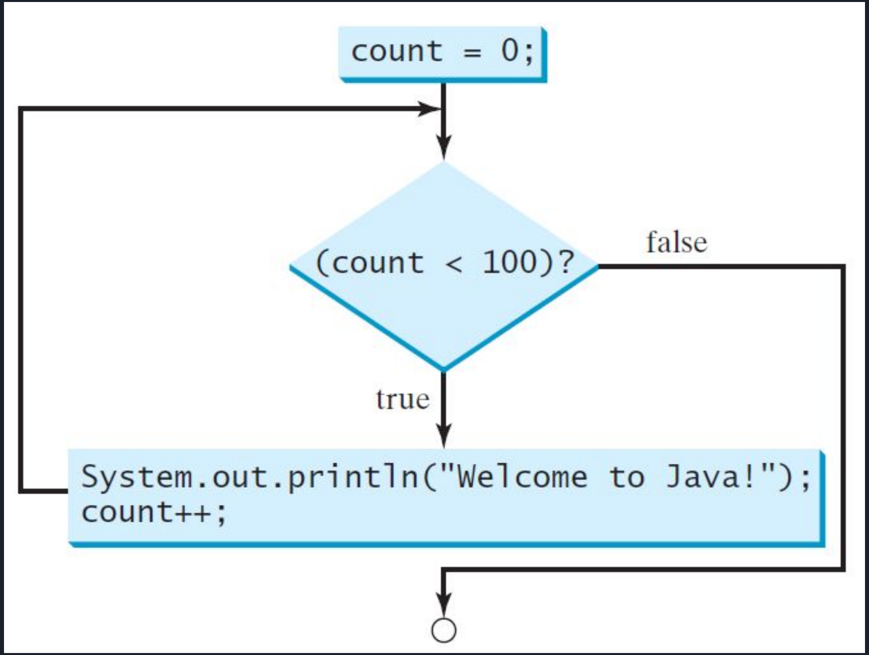
`while` Loop Flow Chart

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



`while` Loop Flow Chart

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



Ending a Loop with a Sentinel Value

- Often the number of times a loop is executed is not predetermined. You may use an input value to signify the end of the loop. Such a value is known as a sentinel value.
- Write a program that reads and calculates the sum of an unspecified number of integers. The input 0 signifies the end of the input.

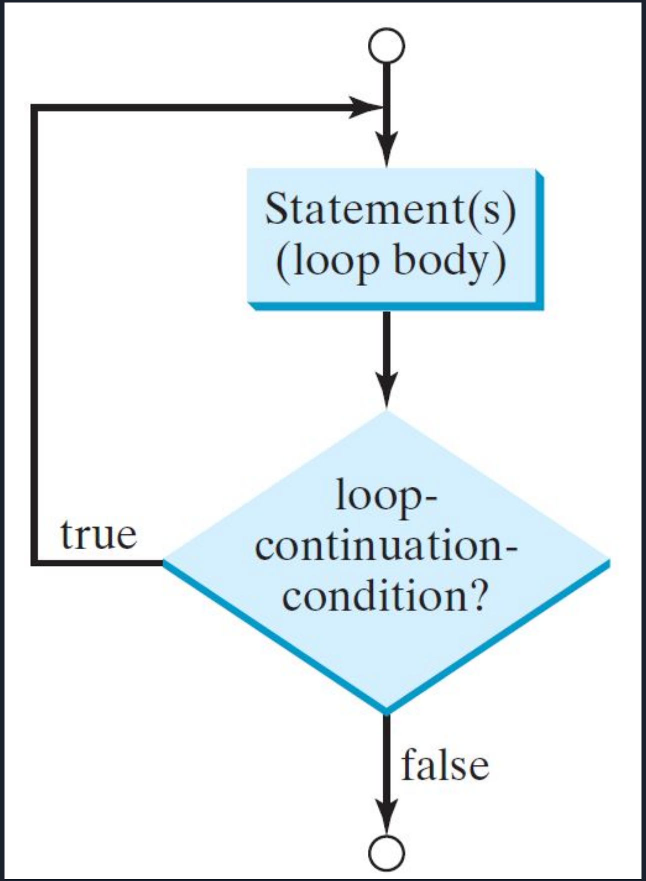
Caution

- Don’t use floating-point values for equality checking in a loop control. Since floating-point values are approximations for some values, using them could result in imprecise counter values and inaccurate results. Consider the following code for computing 1 + 0.9 + 0.8 + ... + 0.1:

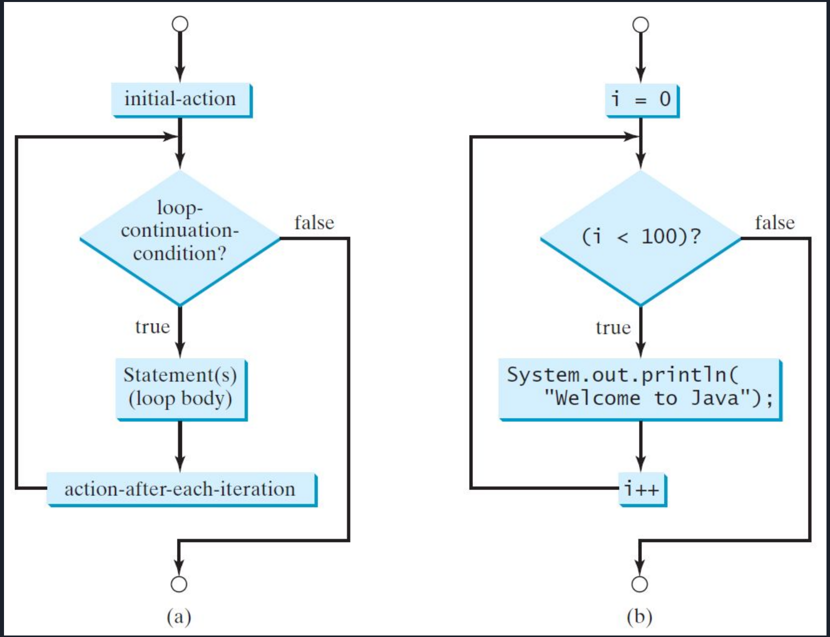
```
double item = 1; double sum = 0;
while (item != 0) { // No guarantee item will be 0
  sum += item;
  item -= 0.1;
}
System.out.println(sum);
```

do-while Loop

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



for Loops


initial-action
loop-continuation-condition?
false
true
Statement(s) (loop body)
action-after-each-iteration
(a)
i = 0
(i < 100)?
false
true
System.out.println("Welcome to Java");
i++
(b)

```
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!");
}
```

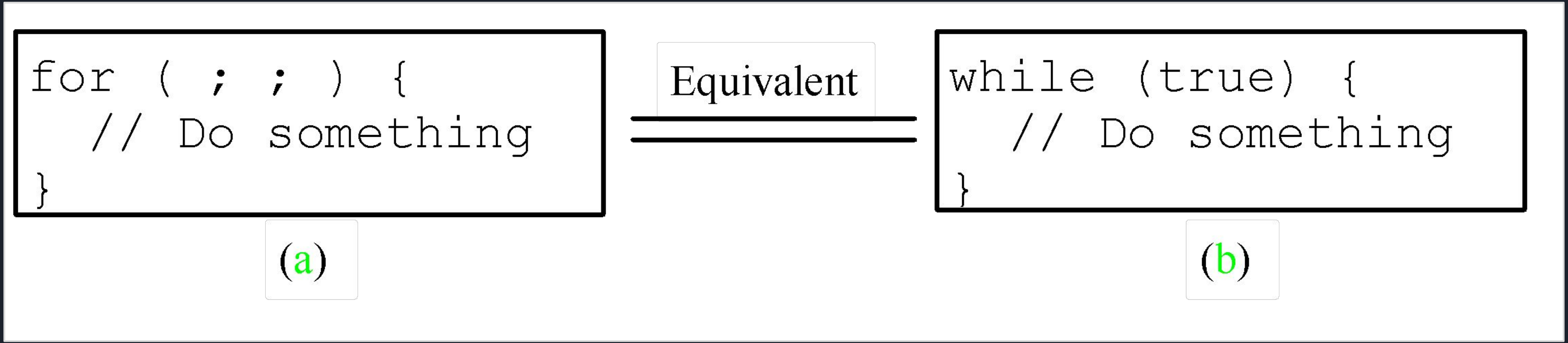
Note

- 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. Therefore, the following two for loops are correct. They are rarely used in practice, however.

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


for (int i = 0, j = 0; (i + j < 10); i++, j++) {

  // Do something

}
```

Note

- If the loop-continuation-condition in a for loop is omitted, it is implicitly true. Thus the statement given below in (a), which is an infinite loop, is correct. Nevertheless, it is better to use the equivalent loop in (b) to avoid confusion:



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

Equivalent

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

(a) (b)

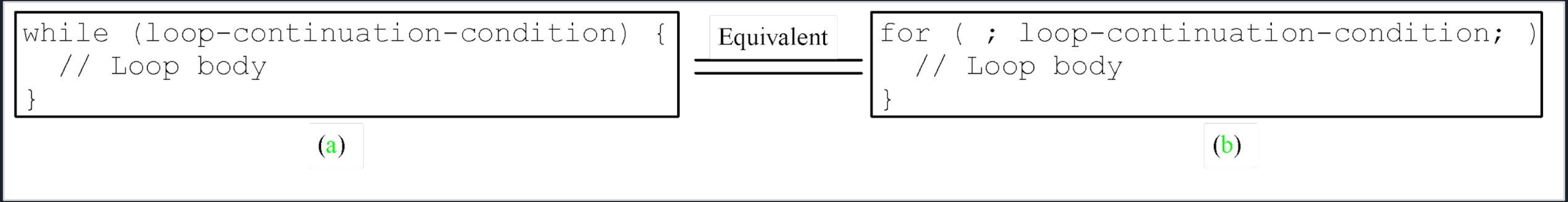
Caution

- Adding a semicolon at the end of the for clause before the loop body is a common mistake, as shown below:

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

Which Loop to Use?

The three forms of loop statements, while, do-while, and for, are expressively equivalent; that is, you can write a loop in any of these three forms. For example, a while loop in (a) in the following figure can always be converted into the following for loop in (b):



```
while (loop-continuation-condition) {
  // Loop body
}
```

(a)

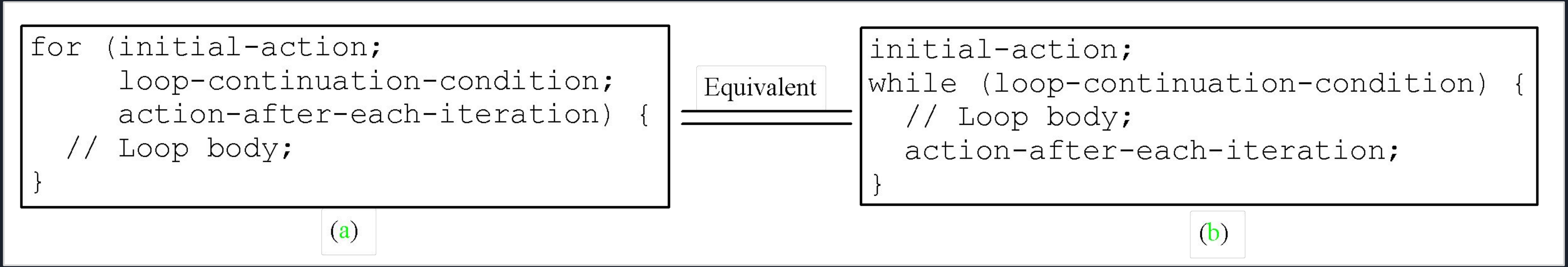
Equivalent

```
for ( ; loop-continuation-condition; )
  // Loop body
}
```

(b)

Which Loop to Use?

A for loop in (a) in the following figure can generally be converted into the following while loop in (b).



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

(a)

Equivalent

```
initial-action;
while (loop-continuation-condition) {
  // Loop body;
  action-after-each-iteration;
}
```

(b)

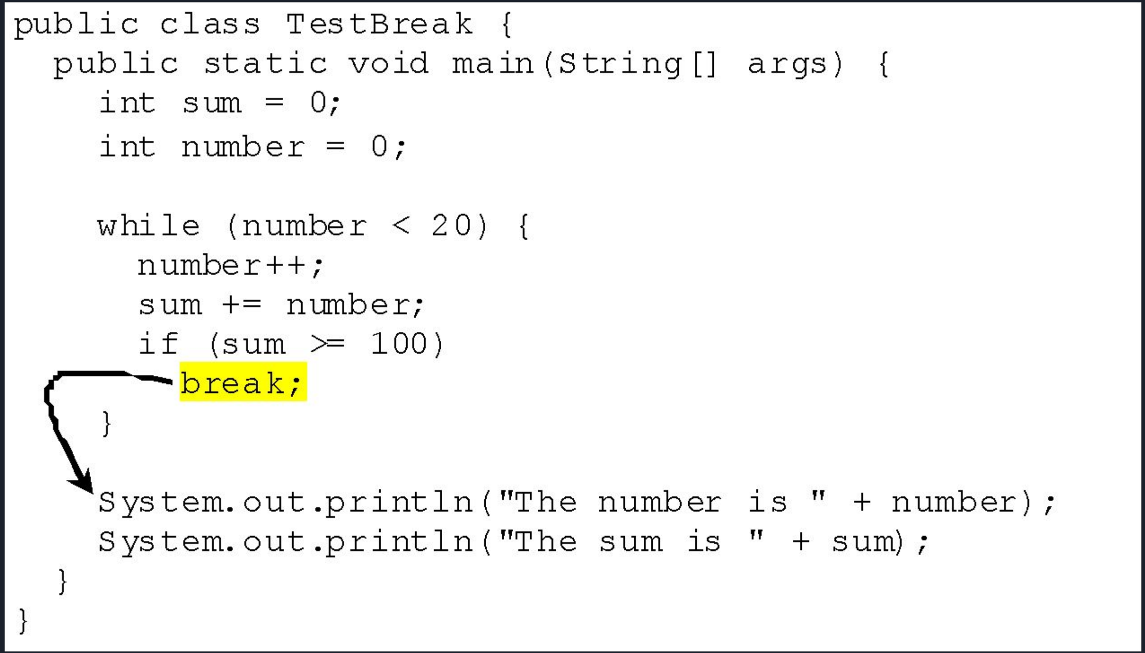
Recommendations

- Use the one that is most intuitive and comfortable for you. In general, a for loop may be used if the number of repetitions is known, as, for example, when you need to print a message 100 times. A while loop may be used if the number of repetitions is not known, as in the case of reading the numbers until the input is 0. A do-while loop can be used to replace a while loop if the loop body has to be executed before testing the continuation condition.

Using break and continue

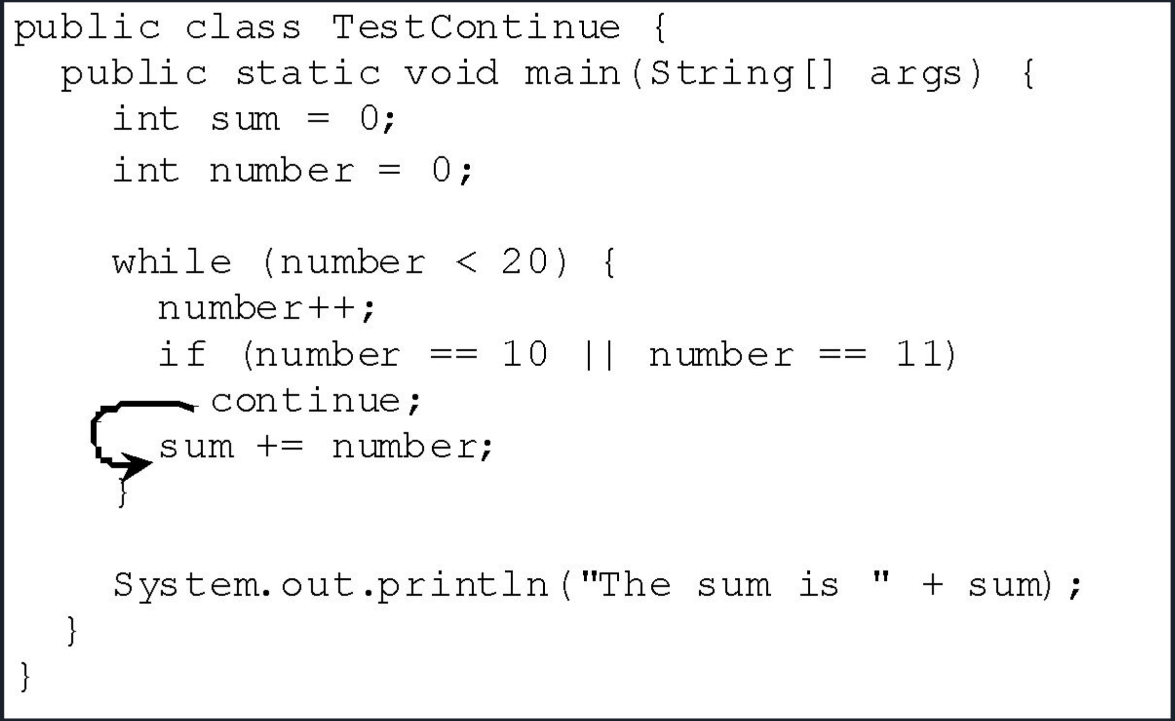
- Examples for using the `break` and `continue` keywords:

break



```
public class TestBreak {
  public static void main(String[] args) {
    int sum = 0;
    int number = 0;

    while (number < 20) {
      number++;
      sum += number;
      if (sum >= 100)
        break;
    }

    System.out.println("The number is " + number);
    System.out.println("The sum is " + sum);
  }
}
```

continue



```
public class TestContinue {
  public static void main(String[] args) {
    int sum = 0;
    int number = 0;

    while (number < 20) {
      number++;
      if (number == 10 || number == 11)
        continue;
      sum += number;
    }

    System.out.println("The sum is " + sum);
  }
}
```

Methods

Opening Problem

- Find the sum of integers from 1 to 10, from 20 to 30, and from 35 to 45, respectively.

Problem

```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 30; i++)
  sum += i;
System.out.println("Sum from 20 to 30 is " + sum);

sum = 0;
for (int i = 35; i <= 45; i++)
  sum += i;
System.out.println("Sum from 35 to 45 is " + sum);
```

Problem

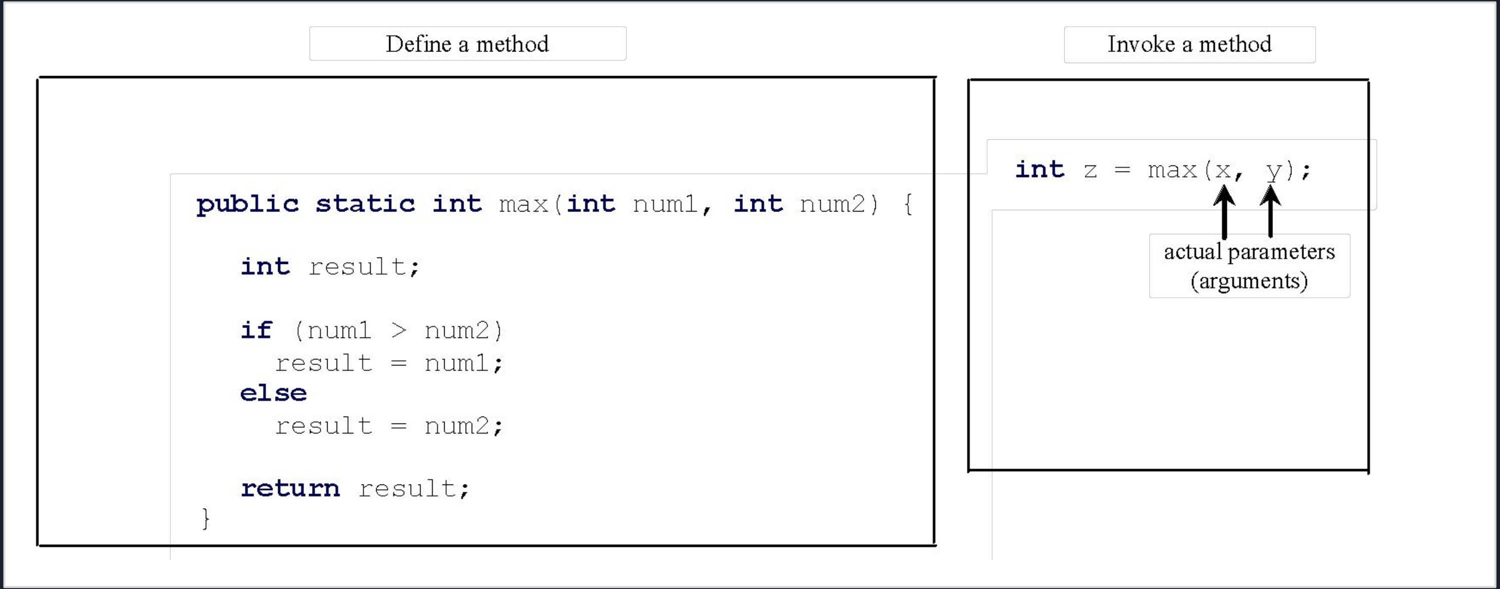
```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 30; i++)
  sum += i;
System.out.println("Sum from 20 to 30 is " + sum);

sum = 0;
for (int i = 35; i <= 45; i++)
  sum += i;
System.out.println("Sum from 35 to 45 is " + sum);
```

Solution

```java
public static void main(String[] args) {
  System.out.println("Sum from 1 to 10 is " + sum(1,
10));
  System.out.println("Sum from 20 to 30 is " + sum(20,
30));
  System.out.println("Sum from 35 to 45 is " + sum(35,
45));
}

public static int sum(int i1, int i2) {
  int sum = 0;
  for (int i = i1; i <= i2; i++)
    sum += i;
  return sum;
}
```

Defining Methods

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

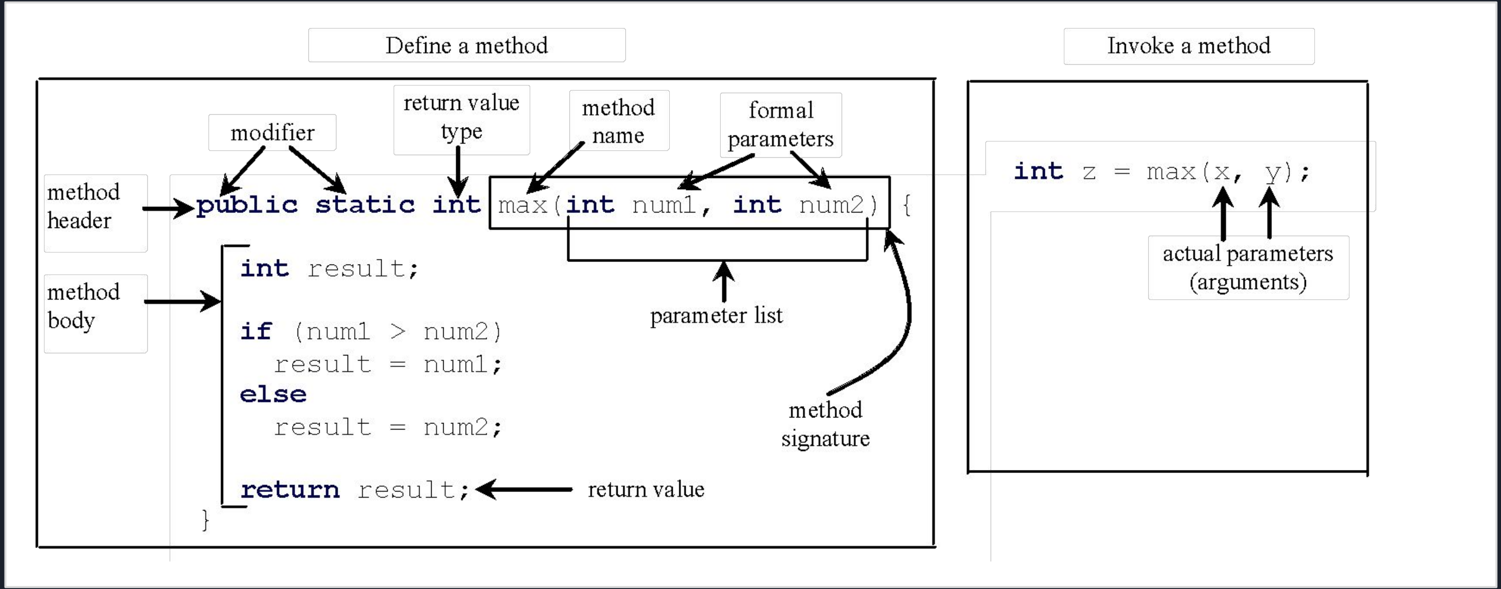


```
public static int max(int num1, int num2) {

   int result;

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

   return result;
}
```

```
int z = max(x, y);
```

Defining Methods

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

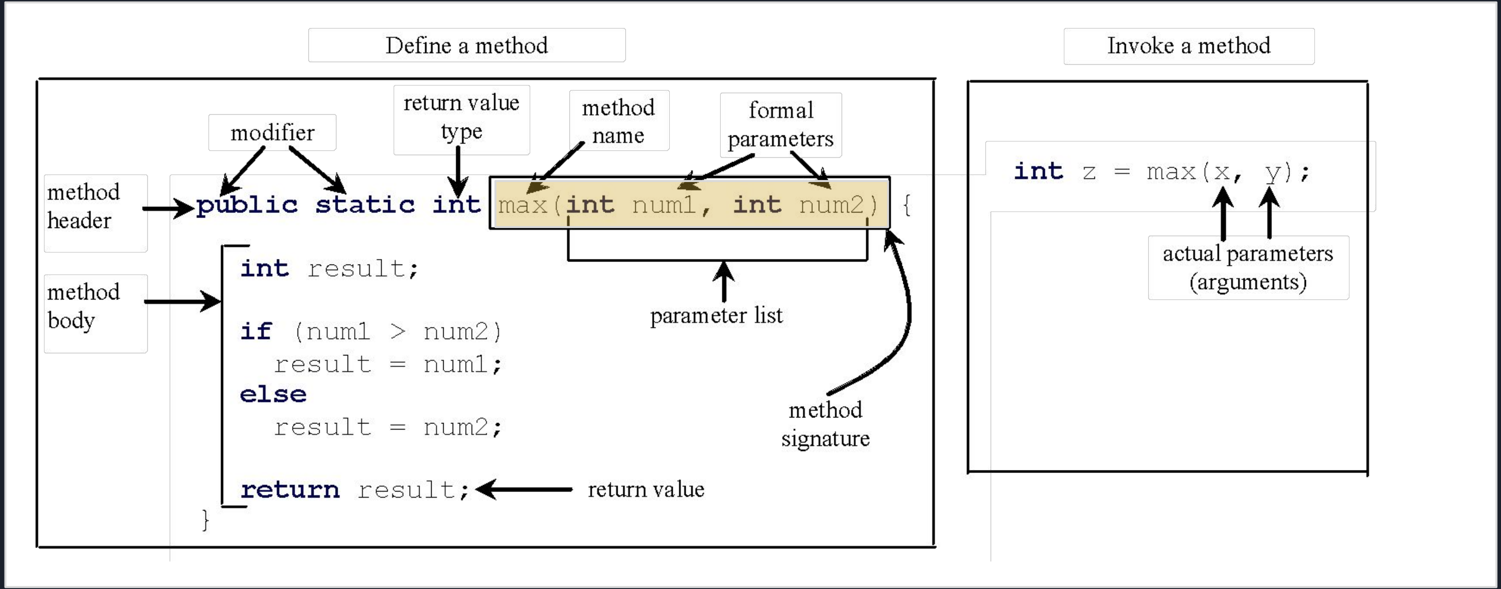


```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```

```
int z = max(x, y);
```

Method Signature

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

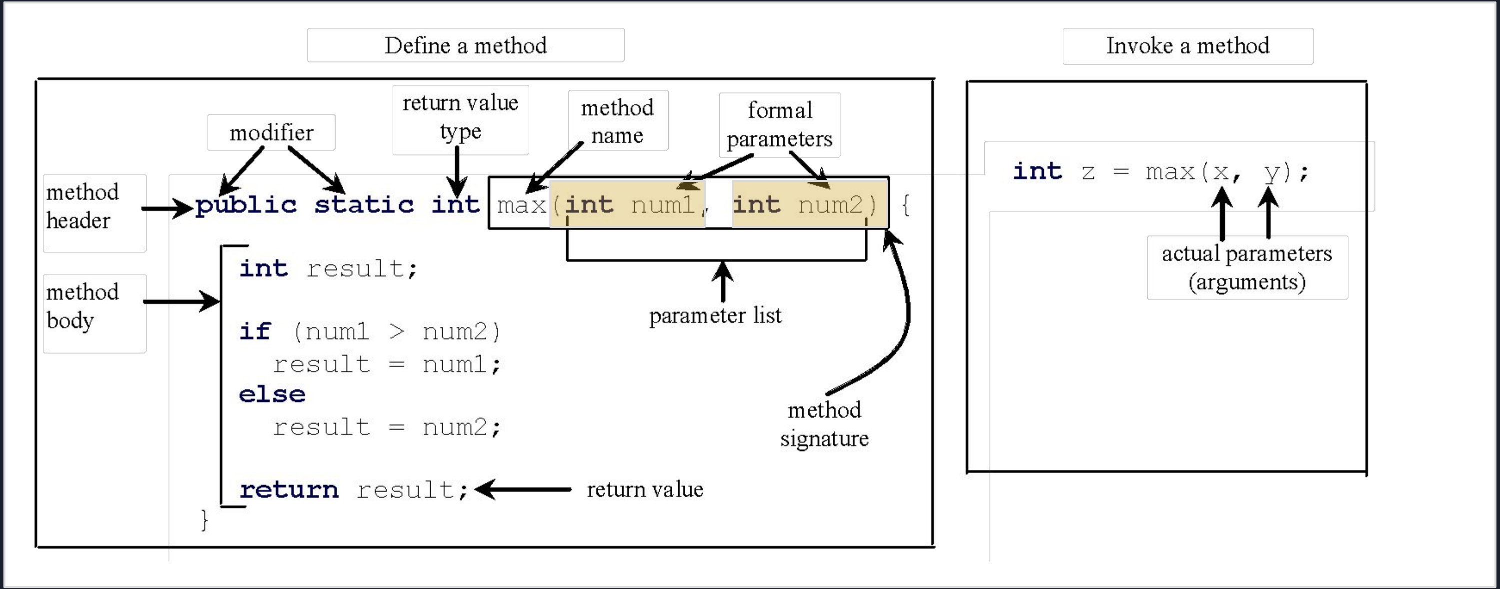


```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```

```
int z = max(x, y);
```

Formal Parameters

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

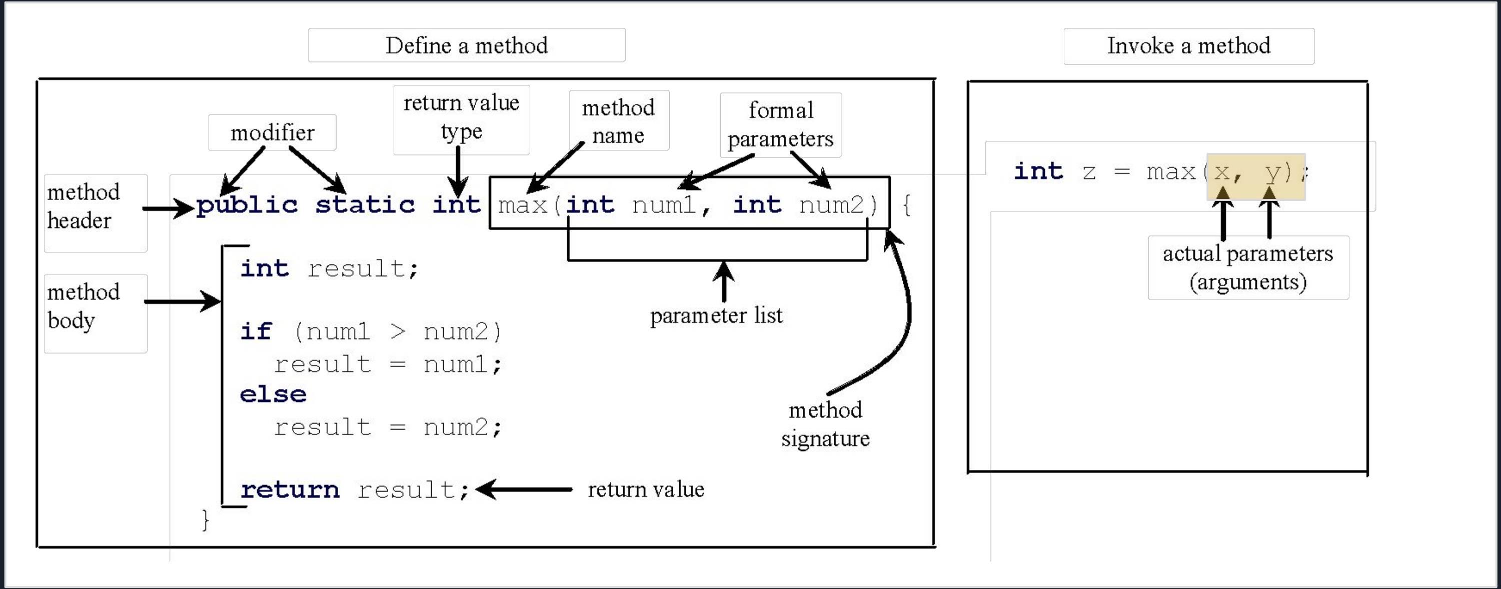


```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```

```
int z = max(x, y);
```

Actual Parameters

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

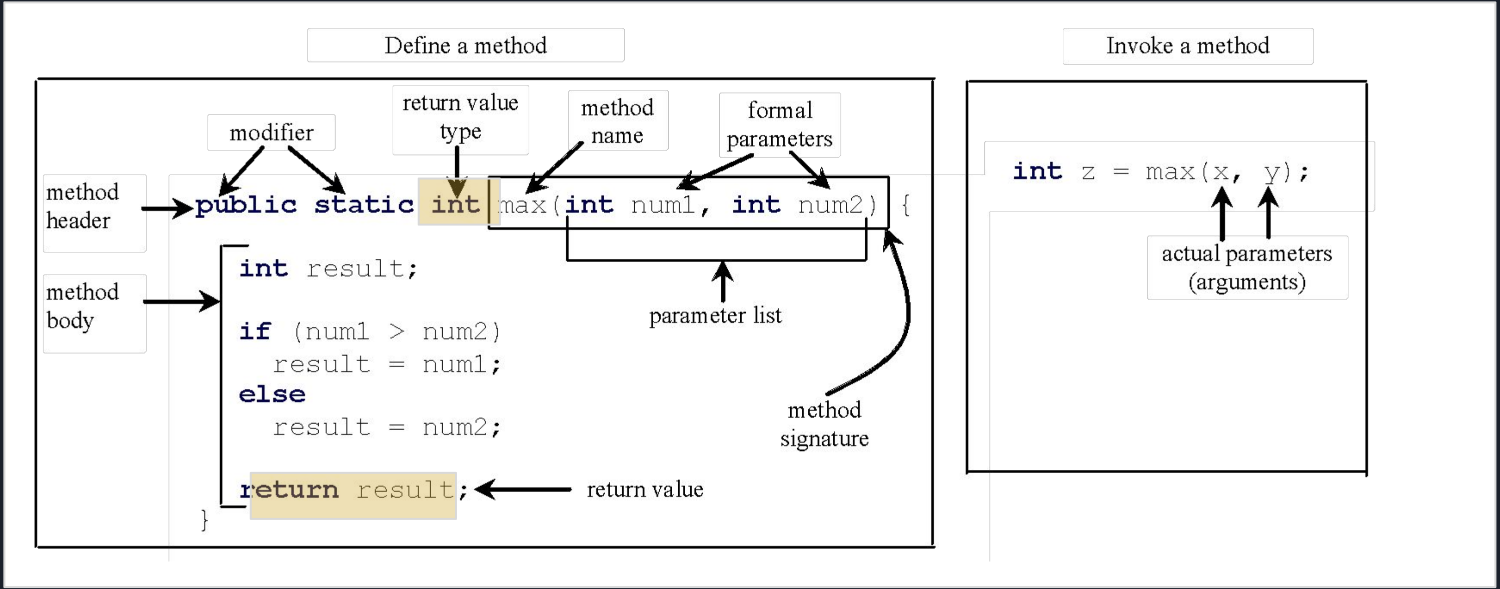


```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```

```
int z = max(x, y);
```

Return Value Type

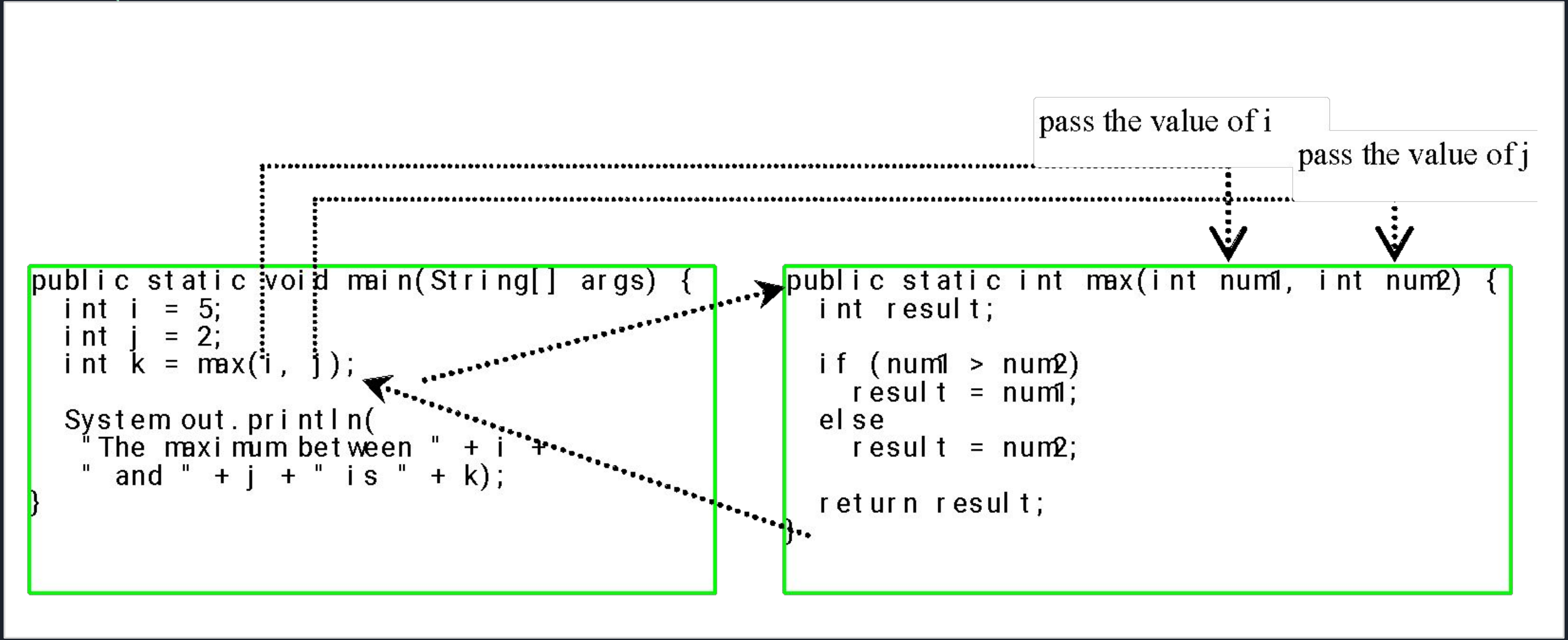
- A method may return a value. The returnValueType is the data type of the value the method returns. If the method does not return a value, the returnValueType is the keyword void. For example, the returnValueType in the main method is void.



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

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

  return result;
}
```

```
int z = max(x, y);
```

Calling Methods

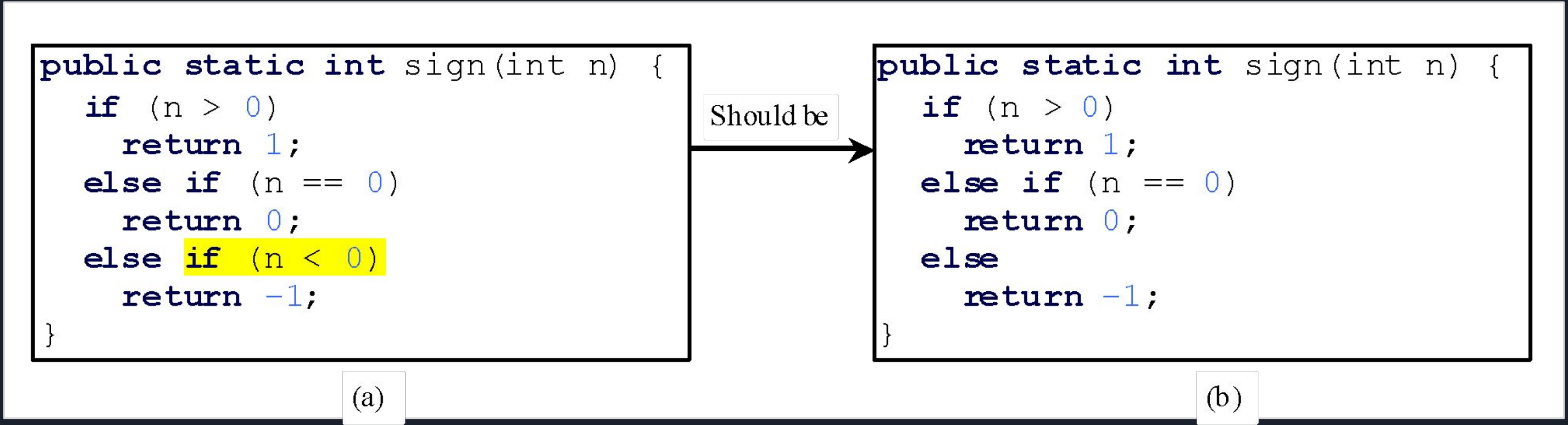


```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  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;
}
```

CAUTION

- A return statement is required for a value-returning method. The method shown below in (a) is logically correct, but it 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;
}
```

Should be

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

(a) (b)

Reuse Methods from Other Classes

- NOTE: One of the benefits of methods is for reuse. The max method can be invoked from any class besides TestMax. If you create a new class Test, you can invoke the max method using ClassName.methodName (e.g., TestMax.max).

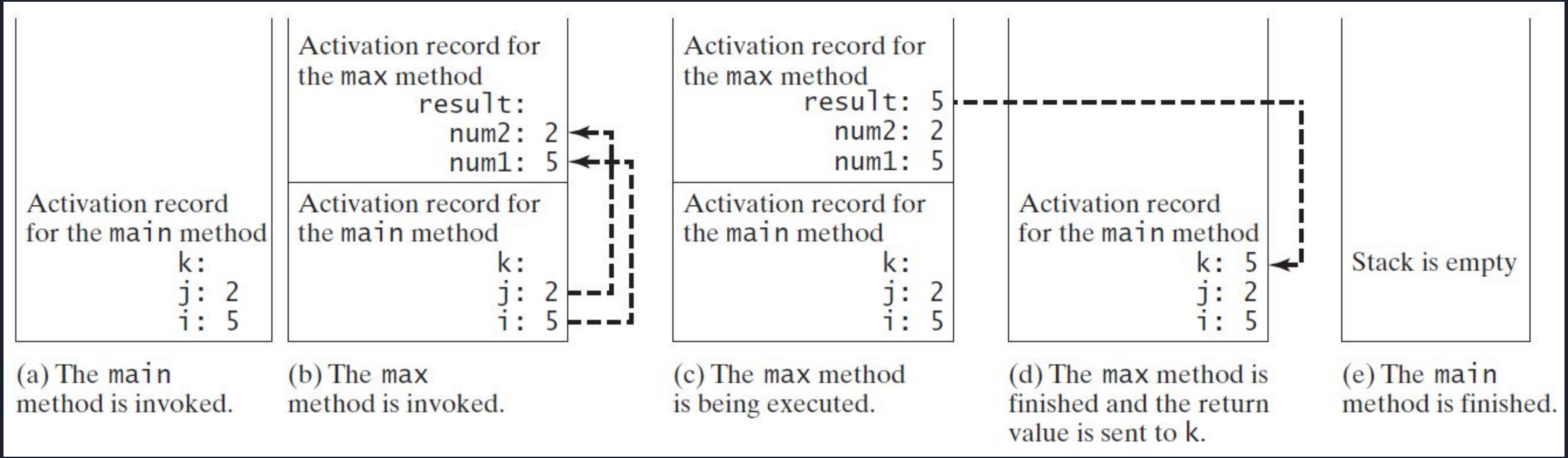
void Method Example

- This type of method does not return a value. The method performs some actions.

Passing Parameters

```
public static void nPrintln(String message, int n) {
  for (int i = 0; i < n; i++)
    System.out.println(message);
}
```

Call Stacks


Activation record for the main method
k:
j: 2
i: 5
Activation record for the max method
result:
num2: 2
num1: 5
Activation record for the main method
k:
j: 2
i: 5
Activation record for the max method
result: 5
num2: 2
num1: 5
Activation record for the main method
k:
j: 2
i: 5
Activation record for the main method
k: 5
j: 2
i: 5
Stack is empty

(a) The main method is invoked.

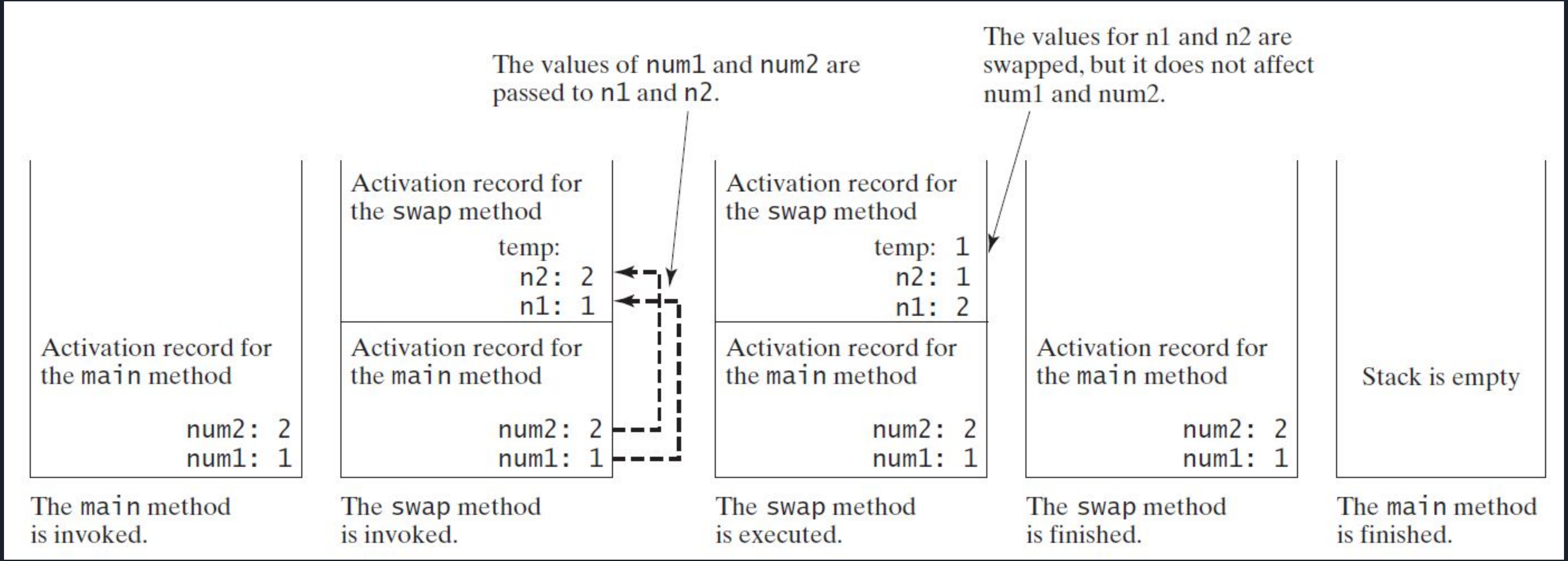
(b) The max method is invoked.

(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.

Pass by Value


The values of num1 and num2 are passed to n1 and n2.
The values for n1 and n2 are swapped, but it does not affect num1 and num2.
Activation record for the main method
num2: 2
num1: 1
The main method is invoked.
Activation record for the swap method
temp:
n2: 2
n1: 1
Activation record for the main method
num2: 2
num1: 1
The swap method is invoked.
Activation record for the swap method
temp: 1
n2: 1
n1: 2
Activation record for the main method
num2: 2
num1: 1
The swap method is executed.
Activation record for the main method
num2: 2
num1: 1
The swap method is finished.
Stack is empty
The main method is finished.

Modularizing Code

- Methods can be used to reduce redundant coding and enable code reuse. Methods can also be used to modularize code and improve the quality of the program.

Overloading Methods

- Overloading the `max` Method

```
public static double max(double num1, double num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

Ambiguous Invocation

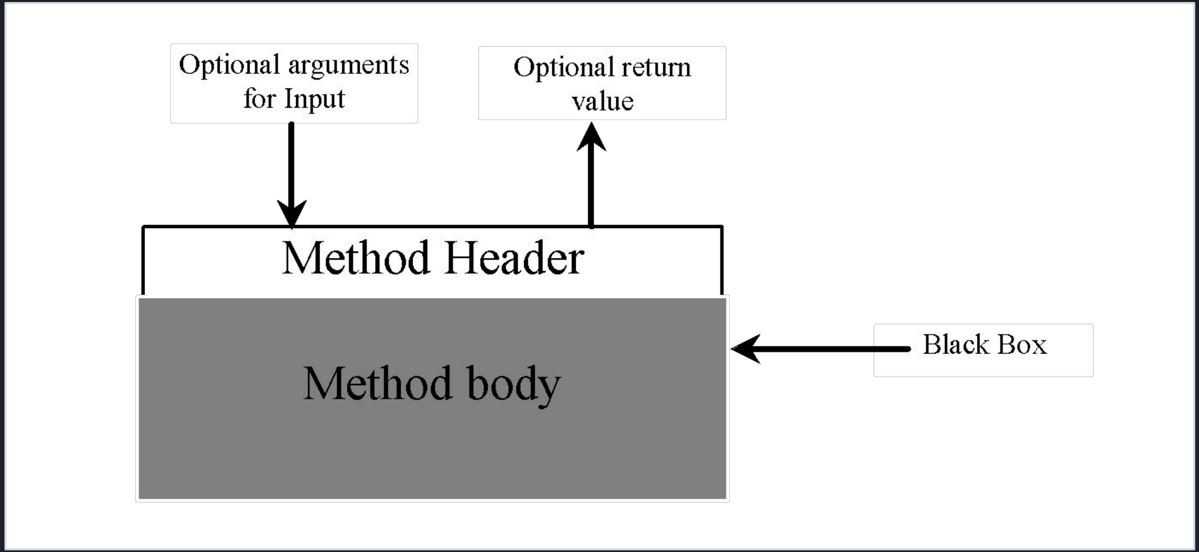
- Sometimes there may be two or more possible matches for an invocation of a method, but the compiler cannot determine the most specific match. This is referred to as *ambiguous invocation*. Ambiguous invocation is a compile error.

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;
  }
}
```

Method Abstraction

- You can think of the method body as a black box that contains the detailed implementation for the method.



Benefits of Methods

- Write a method once and reuse it anywhere.
- Information hiding. Hide the implementation from the user.
- Reduce complexity.

Scope of Local Variables

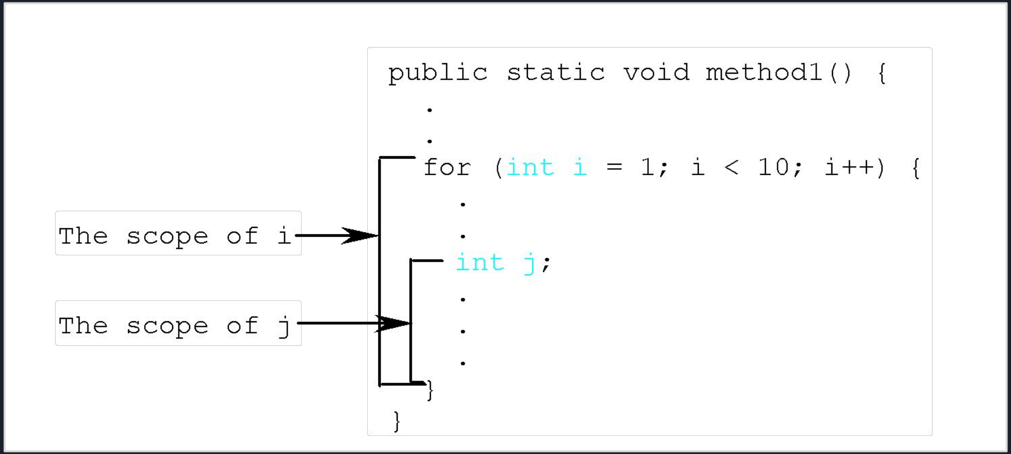
- A local variable: a variable defined inside a method.
- Scope: the part of the program where the variable can be referenced.
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared before it can be used.

Scope of Local Variables

- You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks.

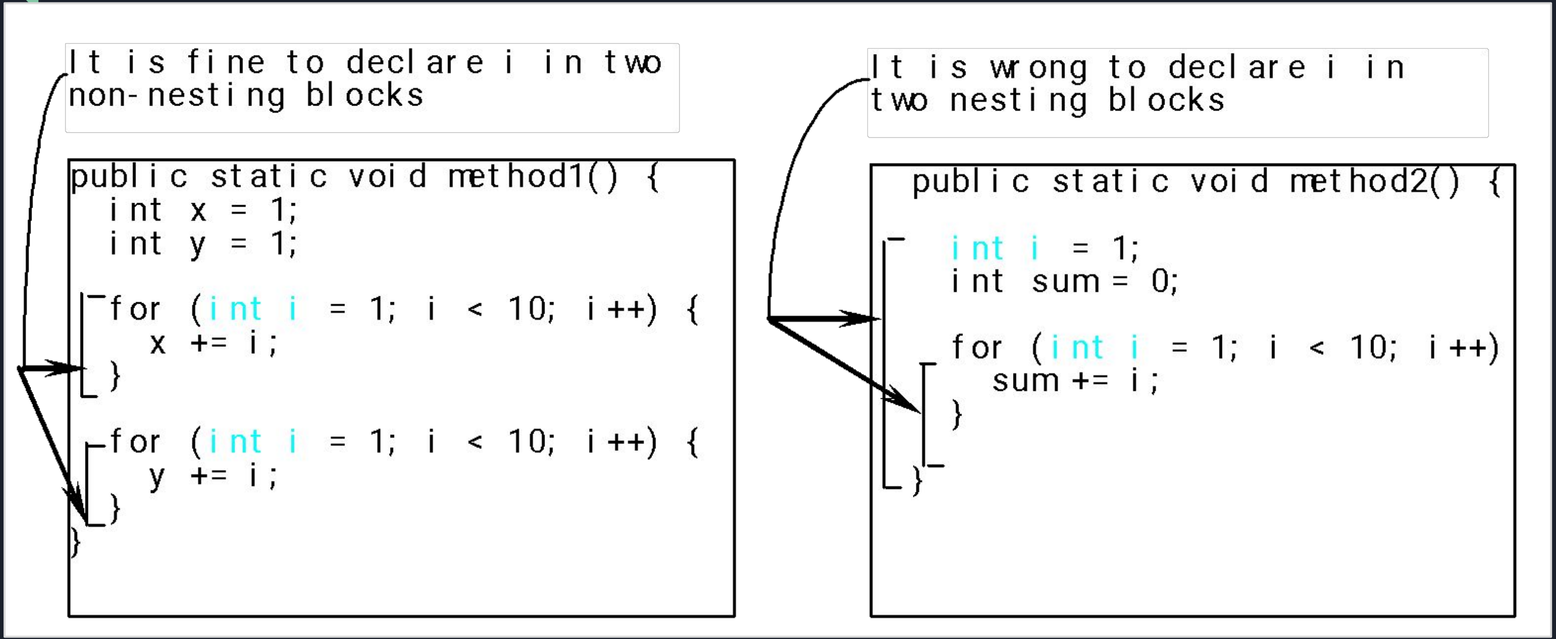
Scope of Local Variables

- A variable declared in the initial action part of a <u>for</u> loop header has its scope in the entire loop. But a variable declared inside a <u>for</u> loop body has its scope limited in the loop body from its declaration and to the end of the block that contains the variable.



```
public static void method1() {
   .
   .
   for (int i = 1; i < 10; i++) {
      .
      .
      int j;
      .
      .
      .
   }
}
```

Scope of Local Variables



It is fine to declare i in two non-nesting blocks

```
public static void method1() {
  int x = 1;
  int y = 1;

  for (int i = 1; i < 10; i++) {
    x += i;
  }

  for (int i = 1; i < 10; i++) {
    y += i;
  }
}
```

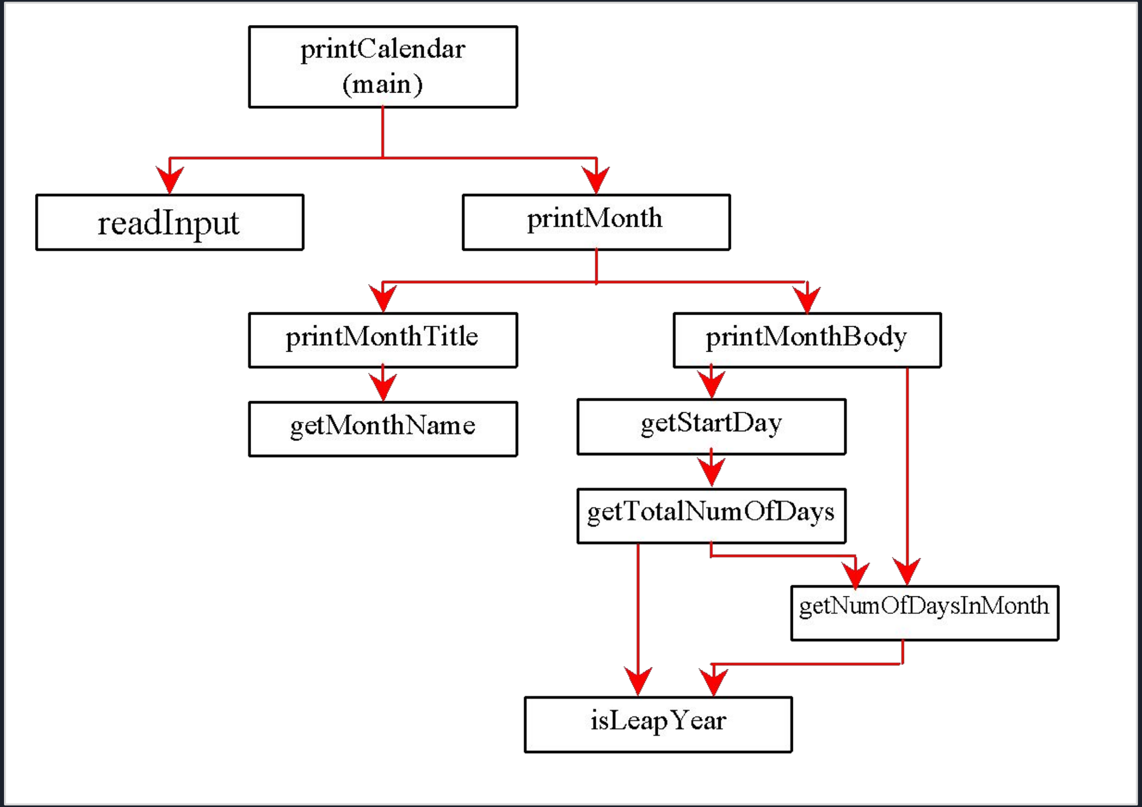
It is wrong to declare i in two nesting blocks

```
  public static void method2() {

    int i = 1;
    int sum = 0;

    for (int i = 1; i < 10; i++)
      sum += i;
    }

}
```

Stepwise Refinement (Optional)

- The concept of method abstraction can be applied to the process of developing programs. When writing a large program, you can use the “divide and conquer” strategy, also known as *stepwise refinement*, to decompose it into subproblems. The subproblems can be further decomposed into smaller, more manageable problems.

Design Diagram



Implementation: Top-Down

Top-down approach is to implement one method in the structure chart at a time from the top to the bottom. Stubs can be used for the methods waiting to be implemented. A stub is a simple but incomplete version of a method. The use of stubs enables you to test invoking the method from a caller. Implement the main method first and then use a stub for the printMonth method. For example, let printMonth display the year and the month in the stub. Thus, your program may begin like this:

Implementation: Bottom-Up

Bottom-up approach is to implement one method in the structure chart at a time from the bottom to the top. For each method implemented, write a test program to test it. Both top-down and bottom-up methods are fine. Both approaches implement the methods incrementally and help to isolate programming errors and makes debugging easy. Sometimes, they can be used together.

Mathematical Functions, Characters, and Strings

Mathematical Functions

Java provides many useful methods in the **Math** class for performing common mathematical functions.

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)`

```
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.rint(2.1) returns 2.0
```

```
Math.rint(2.0) returns 2.0
Math.rint(-2.0) returns -2.0
Math.rint(-2.1) returns -2.0
Math.rint(2.5) returns 2.0
Math.rint(-2.5) returns -2.0
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).

`(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.

Character Data Type

```
char letter = 'A'; (ASCII)
char numChar = '4'; (ASCII)

char letter = '\u0041'; (Unicode)

char numChar = '\u0034'; (Unicode)
```

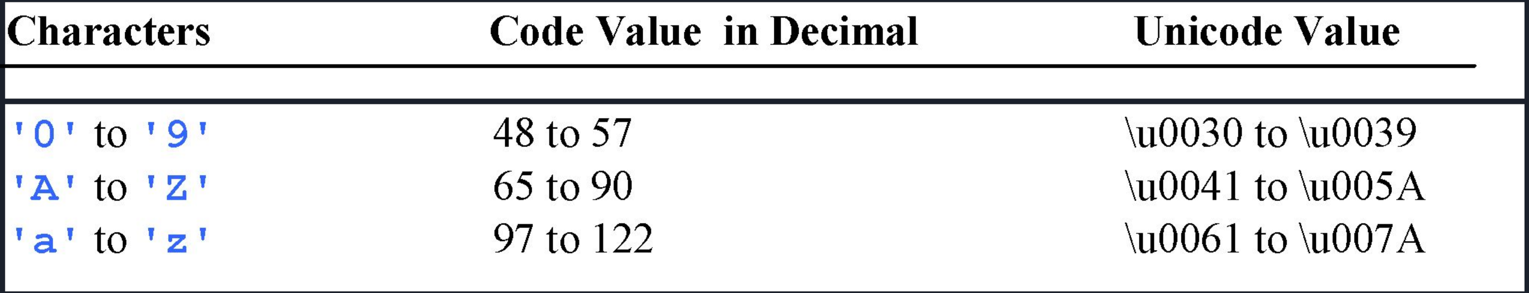
NOTE: The increment and decrement operators can also be used on char variables to get the next or preceding Unicode character. For example, the following statements display character b.

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

Unicode Format

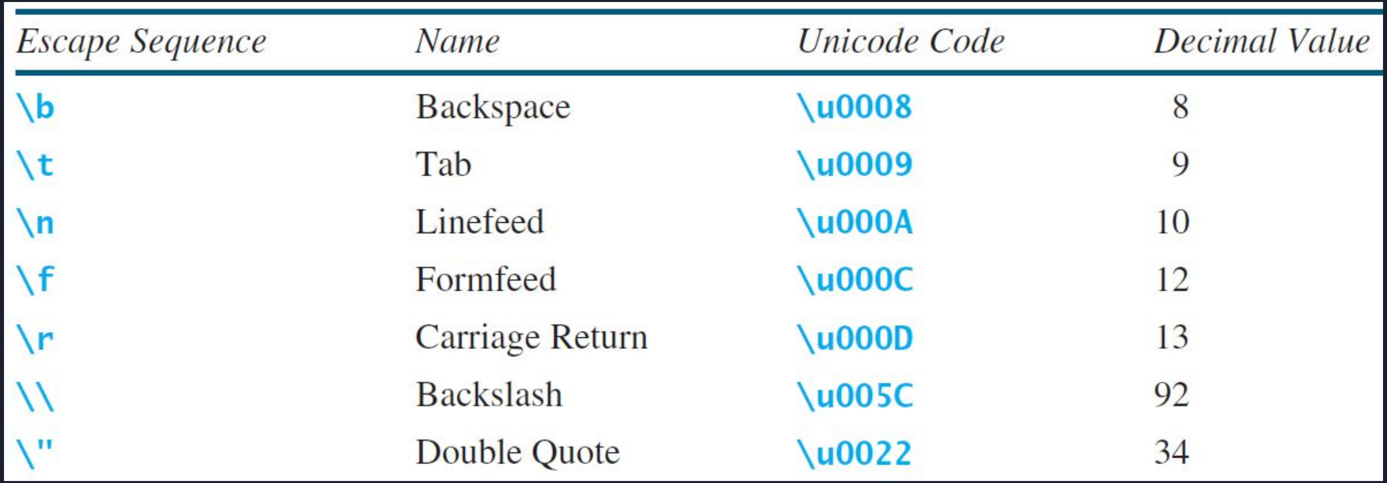
Java characters use *Unicode*, a 16-bit encoding scheme established by the Unicode Consortium to support the interchange, processing, and display of written texts in the world’s diverse languages. Unicode takes two bytes, preceded by \u, expressed in four hexadecimal numbers that run from '\u0000' to '\uFFFF'. So, Unicode can represent `65535 + 1 characters.`

ASCII Code for Commonly Used Characters



Characters	Code Value in Decimal	Unicode Value
'0' to '9'	48 to 57	\u0030 to \u0039
'A' to 'Z'	65 to 90	\u0041 to \u005A
'a' to 'z'	97 to 122	\u0061 to \u007A

Escape Sequences for Special Characters



Escape Sequence	*Name*	*Unicode Code*	*Decimal Value*
`\b`	Backspace	`\u0008`	8
`\t`	Tab	`\u0009`	9
`\n`	Linefeed	`\u000A`	10
`\f`	Formfeed	`\u000C`	12
`\r`	Carriage Return	`\u000D`	13
`\\`	Backslash	`\u005C`	92
`\"`	Double Quote	`\u0022`	34

Casting between char and Numeric Types

```
int i = 'a'; // Same as int i = (int)'a';


char c = 97; // Same as char c = (char)97;
```

Comparing and Testing Characters

```java
if (ch >= 'A' && ch <= 'Z')
   System.out.println(ch + " is an uppercase
  letter");
else if (ch >= 'a' && ch <= 'z')
   System.out.println(ch + " is a lowercase
  letter");
else if (ch >= '0' && ch <= '9')
   System.out.println(ch + " is a numeric
  character");
```

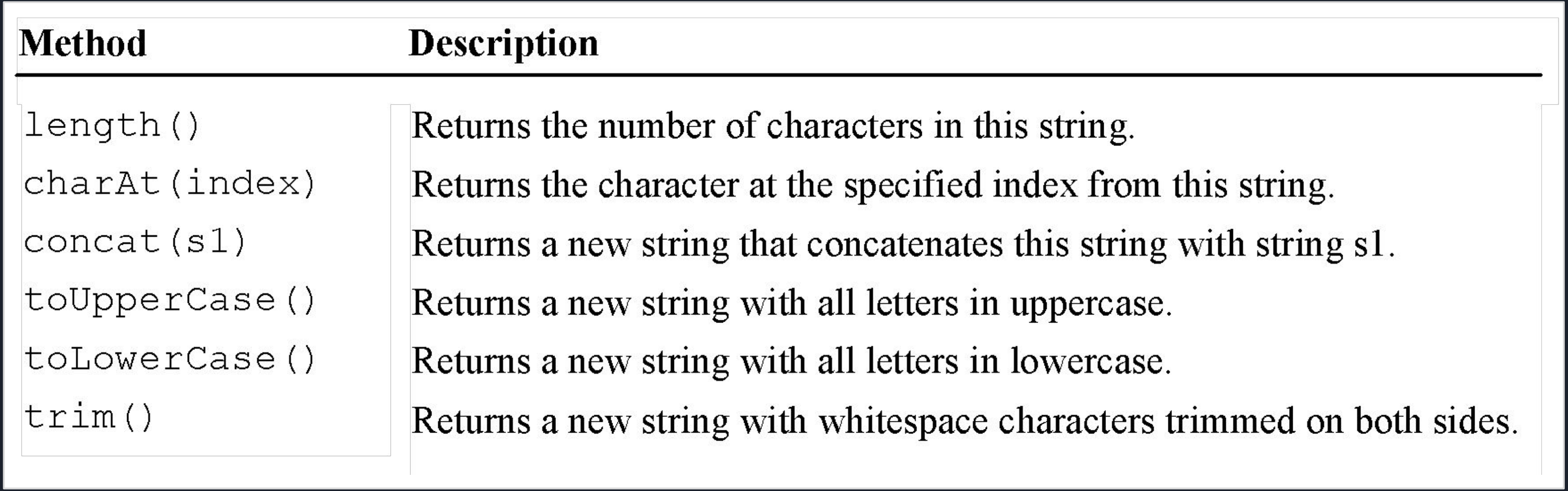
The String Type

The char type only represents one character. To represent a string of characters, use the data type called String. For example,

String message = "Welcome to Java";

String is actually a predefined class in the Java library just like the System class and Scanner class. The String type is not a primitive type. It is known as a *reference type*. Any Java class can be used as a reference type for a variable. Reference data types will be thoroughly discussed in later on. For the time being, you just need to know how to declare a String variable, how to assign a string to the variable, how to concatenate strings, and to perform simple operations for strings.

Simple Methods for **String** Objects



Method	Description
`length()`	Returns the number of characters in this string.
`charAt(index)`	Returns the character at the specified index from this string.
`concat(s1)`	Returns a new string that concatenates this string with string s1.
`toUpperCase()`	Returns a new string with all letters in uppercase.
`toLowerCase()`	Returns a new string with all letters in lowercase.
`trim()`	Returns a new string with whitespace characters trimmed on both sides.

Simple Methods for **String** Objects

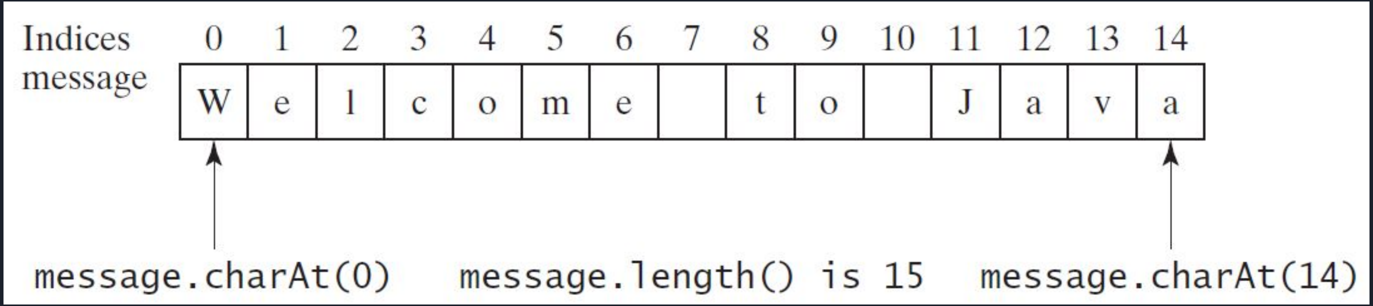
Strings are objects in Java. The methods in the preceding table can only be invoked from a specific string instance. For this reason, these methods are called *instance methods*. A non-instance method is called a *static method*. A static method can be invoked without using an object. All the methods defined in the **Math** class are static methods. They are not tied to a specific object instance. The syntax to invoke an instance method is

referenceVariable.methodName(arguments).

Getting String Length

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

System.out.println("The length of " + message + "
  is "

   + message.length());
```

Getting Characters from a String



```
String message = "Welcome to Java";
System.out.println("The first character in
message is "
    + message.charAt(0));
```

Converting Strings

```
"Welcome".toLowerCase() returns a new string,
welcome.

"Welcome".toUpperCase() returns a new string,
WELCOME.

"  Welcome  ".trim() returns a new string,
Welcome.
```

String Concatenation

```
String s3 = s1.concat(s2); or String s3 = s1 + s2;

// Three strings are concatenated
String message = "Welcome " + "to " + "Java";

// String Chapter is concatenated with number 2
String s = "Chapter" + 2; // s becomes Chapter2

// String Supplement is concatenated with
character B
String s1 = "Supplement" + 'B'; // s1 becomes
SupplementB
```

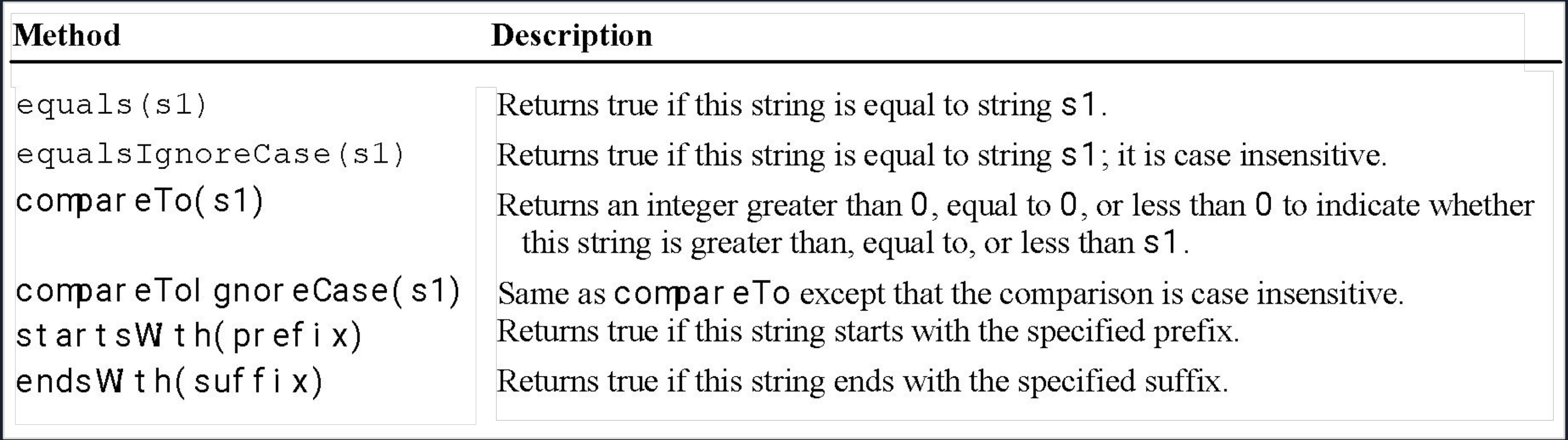
Reading a String from the Console

```
Scanner input = new Scanner(System.in);
System.out.print("Enter three words separated by spaces: ");
String s1 = input.next();
String s2 = input.next();
String s3 = input.next();
System.out.println("s1 is " + s1);
System.out.println("s2 is " + s2);
System.out.println("s3 is " + s3);
```

Reading a Character from the Console

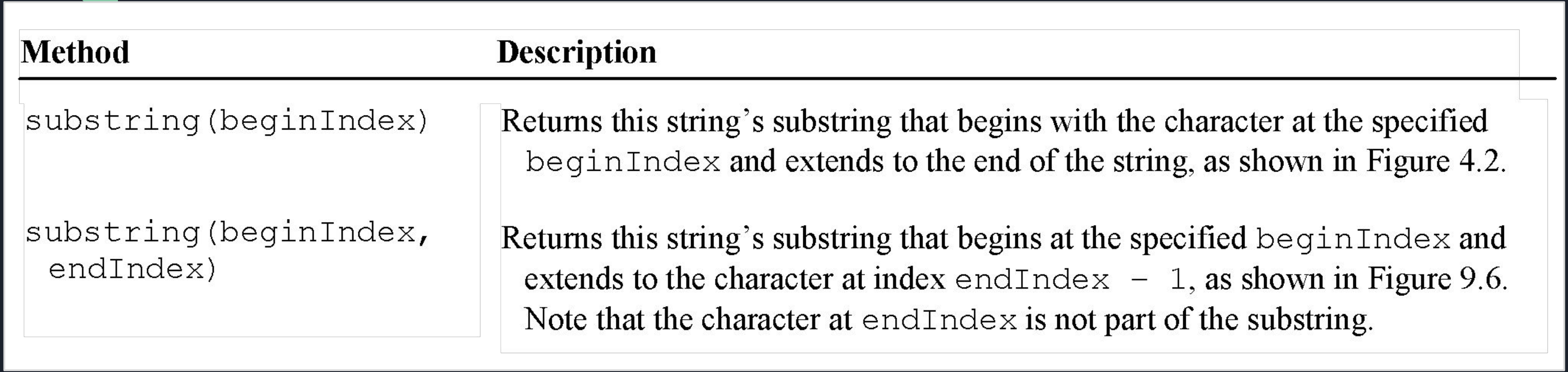
```
Scanner input = new Scanner(System.in);

System.out.print("Enter a character: ");

String s = input.nextLine();

char ch = s.charAt(0);

System.out.println("The character entered is " +
ch);
```

Comparing Strings

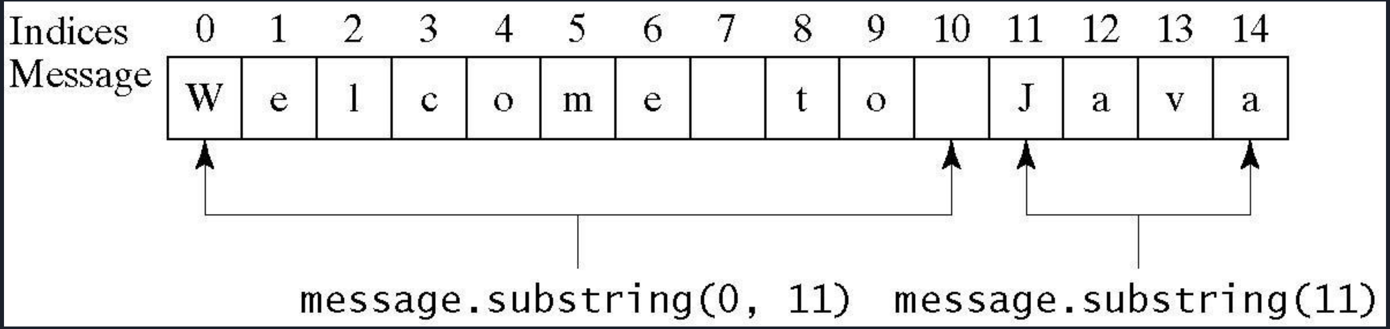


Method	Description
`equals(s1)`	Returns true if this string is equal to string `s1`.
`equalsIgnoreCase(s1)`	Returns true if this string is equal to string `s1`; it is case insensitive.
`compareTo(s1)`	Returns an integer greater than `0`, equal to `0`, or less than `0` to indicate whether this string is greater than, equal to, or less 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.

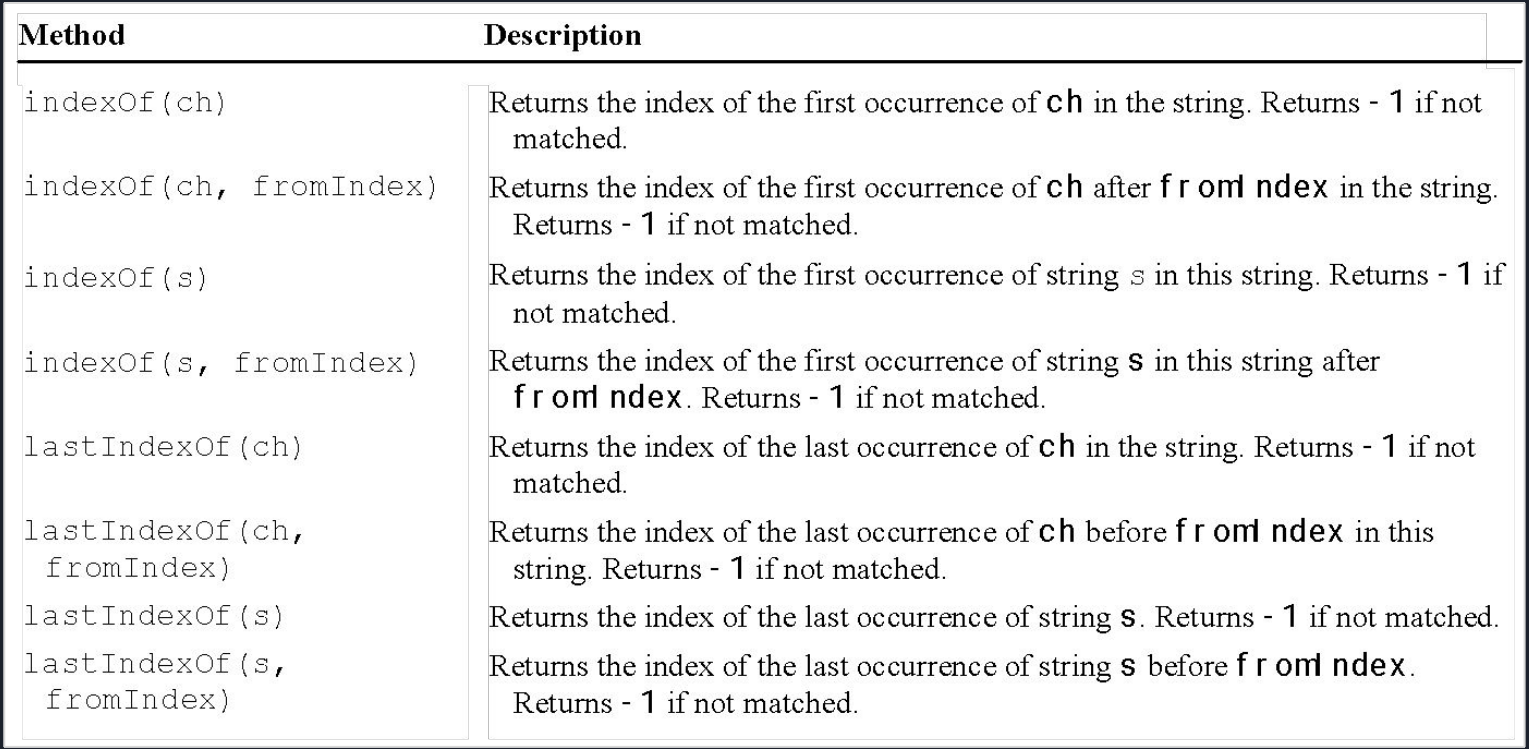
Obtaining Substrings



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.



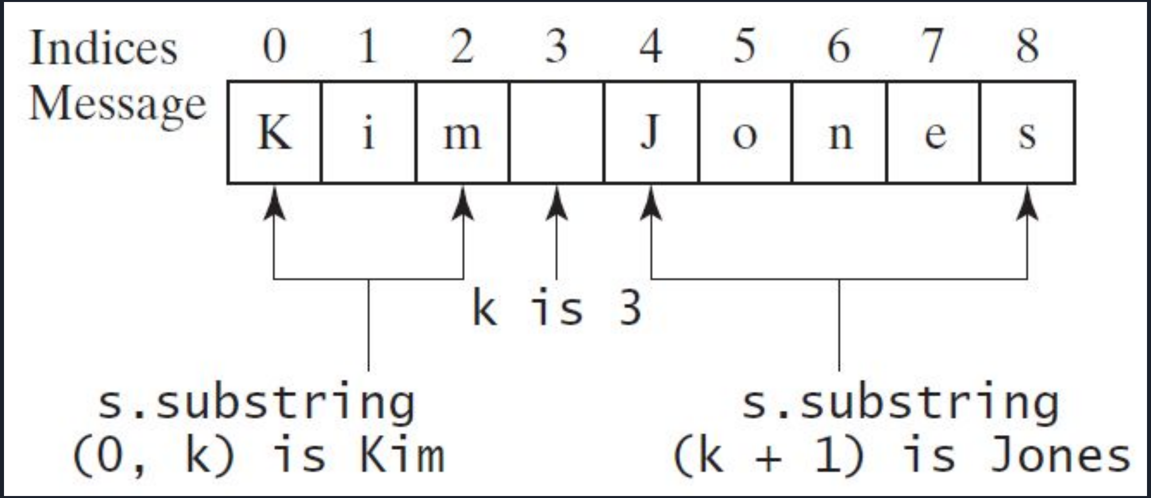
Finding a Character or a Substring in a String



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.

Finding a Character or a Substring in a String

```
int k = s.indexOf(' ');
String firstName = s.substring(0, k);
String lastName = s.substring(k + 1);
```



Conversion between Strings and Numbers

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

String s = number + "";
```

Formatting Output

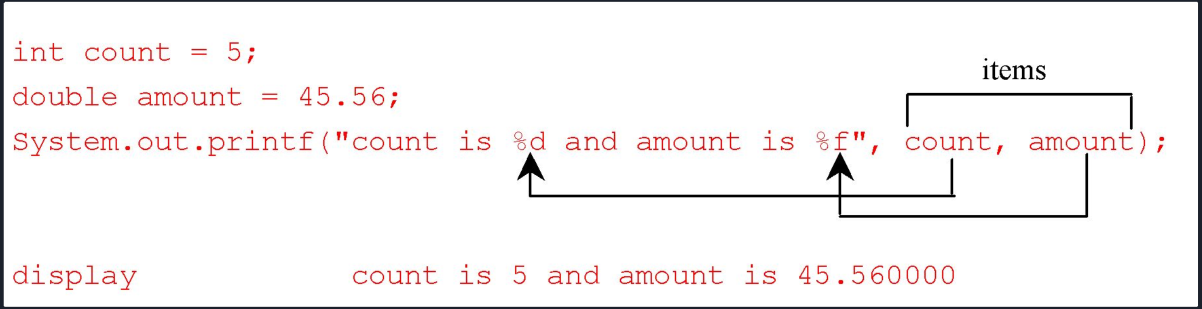
Use the printf statement.

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

Where format is a string that may consist of substrings and format specifiers. A format specifier specifies how an item should be displayed. An item may be a numeric value, character, boolean value, or a string. Each specifier begins with a percent sign.

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 = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);

display           count is 5 and amount is 45.560000
```

Problem: Number Guessing Game

Create a Program that will the user guess a number. The number will be a random integer from 1-100. You will initially show a hint whether the number is odd or even.

The user will have 5 chances.

If the guess is incorrect, tell the user if the number is higher or lower.

Otherwise, show that the user is correct.

Problem: Modified Fizz/Buzz

Create a Program that accepts an integer input n. The program will display all integers from 1 until n.

Conditions:

 if the number is divisible by 3; print out Fizz.

If the number is divisible by 5; print Buzz

If the number is divisible divisible by 3 and 5; skip, don’t print anything.

Otherwise print the number

Ex: Input: 16 Output: 1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 16

Assignments and Exercises

You have until September 19 (Monday) to finish them.

Submission will be closed on September 21 (Wednesday).

Email: you code to jv.lanticse@gmail.com

Please reply your new activities in the same thread as your previous submissions.

Zip it so that you can send them over.

If the code can’t be attached, upload them to your google drive and email the link for me to download it. Make sure to share it to me.