CEN 419
Introduction to Java Programming

Dr. H. Esin ÜNAL

FALL 2021

Slides are modified from original slides of Y. Daniel Liang

Mathematical Functions

- Java provides many useful methods in the Math class for performing common mathematical functions.
- There is no need to import the Math class because it is in the java.lang package and all the classes in the java.lang package are implicitly imported in a Java program.

The `Math` Class

- ***Class constants:***
  - `PI`
  - `E` : The base of natural algorithms
- ***Class methods:***
  - Trigonometric Methods
  - Exponent Methods
  - Rounding Methods
  - min, max, abs, and random Methods

Trigonometric Methods

Method	*Description*
`sin(radians)`	Returns the trigonometric sine of an angle in radians.
`cos(radians)`	Returns the trigonometric cosine of an angle in radians.
`tan(radians)`	Returns the trigonometric tangent of an angle in radians.
`toRadians(degree)`	Returns the angle in radians for the angle in degree.
`toDegree(radians)`	Returns the angle in degrees for the angle in radians.
`asin(a)`	Returns the angle in radians for the inverse of sine.
`acos(a)`	Returns the angle in radians for the inverse of cosine.
`atan(a)`	Returns the angle in radians for the inverse of tangent.

- The parameter for **sin**, **cos**, and **tan** is an angle in radians.
- The return value for **asin** and **atan** is a degree in radians in the range between $-\pi/2$ and $\pi/2$.
- The return value for **acos** is a degree in radians in the range between 0 and π
- One degree is equal to $\pi/180$ in radians, 90 degrees is equal to $\pi/2$ in radians, and 30 degrees is equal to $\pi/6$ in radians.

Trigonometric Methods

```
Math.toDegrees(Math.PI / 2) returns 90.0
Math.toRadians(30) returns 0.5236 (same as π/6)
Math.sin(0) returns 0.0
Math.sin(Math.toRadians(270)) returns -1.0
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
Math.asin(0.5) returns 0.523598333 (same as π/6)
Math.acos(0.5) returns 1.0472 (same as π/3)
Math.atan(1.0) returns 0.785398 (same as π/4)
```

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.

```
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).

```
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
```

The min, max, and abs Methods

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

  Returns the maximum of two parameters.

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

  Returns the minimum of two parameters.

- **`abs(a)`**

  Returns the absolute value of the parameter.

```
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).

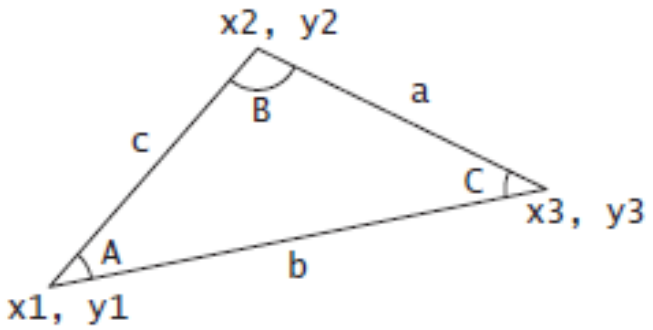
Examples:

```
(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.
```

Case Study: Computing Angles of a Triangle




```
A = acos((a * a - b * b - c * c) / (-2 * b * c))
B = acos((b * b - a * a - c * c) / (-2 * a * c))
C = acos((c * c - b * b - a * a) / (-2 * a * b))
```

Write a program that prompts the user to enter the x- and y-coordinates of the three corner points in a triangle and then displays the triangle's angles.

Compute Angles

Intro to Java Programming, Y. Daniel Liang - ComputeAngles.java (pearsoncmg.com)

Character Data Type

A character data type represents a single character.

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

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

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

```
char ch = (char)0XAB0041;  // The lower 16 bits hex code 0041 is
                           // assigned to ch
System.out.println(ch);    // ch is character A
```

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

```
char ch = (char)65.25;     // Decimal 65 is assigned to ch

System.out.println(ch); // ch is character A
```

Casting between `char` and Numeric Types

When a ***char is cast into a numeric type***, the character's Unicode is cast into the specified numeric type.

```
int i = (int)'A'; // The Unicode of character A is assigned to i

System.out.println(i); // i is 65
```

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

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

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

byte b = (byte)'\uFFF4'; // use explicit casting
```

Numeric Operators on Characters

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

Numeric Operators on Characters

```
int i = '2' + '3'; // (int)'2' is 50 and (int)'3' is 51
System.out.println("i is " + i); // i is 101
int j = 2 + 'a'; // (int)'a' is 97
System.out.println("j is " + j); // j is 99
System.out.println(j + " is the Unicode for character "
  + (char)j); // 99 is the Unicode for character c
System.out.println("Chapter " + '2');
```

display

```
i is 101
j is 99
99 is the Unicode for character c
Chapter 2
```

Comparing and Testing Characters

- Two characters can be compared using the relational operators just like comparing two numbers.
- This is done by comparing the Unicodes of the two characters.
  - **'a' < 'b'** is true because the Unicode for **'a'** (**97**) is less than the Unicode for **'b'** (**98**).
  - **'a' < 'A'** is false because the Unicode for **'a'** (**97**) is greater than the Unicode for **'A'** (**65**).
  - **'1' < '8'** is true because the Unicode for **'1'** (**49**) is less than the Unicode for **'8'** (**56**).

Comparing and Testing Characters

- How can we test whether a character is a number, a letter, an uppercase letter, or a lowercase letter???

```
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");
```

Methods in the Character Class

For convenience, Java provides the following methods in the **Character** class for testing characters

Method	*Description*
`isDigit(ch)`	Returns true if the specified character is a digit.
`isLetter(ch)`	Returns true if the specified character is a letter.
`isLetterOrDigit(ch)`	Returns true if the specified character is a letter or digit.
`isLowerCase(ch)`	Returns true if the specified character is a lowercase letter.
`isUpperCase(ch)`	Returns true if the specified character is an uppercase letter.
`toLowerCase(ch)`	Returns the lowercase of the specified character.
`toUpperCase(ch)`	Returns the uppercase of the specified character.

Methods in the Character Class

```
System.out.println("isDigit('a') is " + Character.isDigit('a'));
System.out.println("isLetter('a') is " + Character.isLetter('a'));
System.out.println("isLowerCase('a') is "
  + Character.isLowerCase('a'));
System.out.println("isUpperCase('a') is "
  + Character.isUpperCase('a'));
System.out.println("toLowerCase('T') is "
  + Character.toLowerCase('T'));
System.out.println("toUpperCase('q') is "
  + Character.toUpperCase('q'));
```

displays

```
isDigit('a') is false
isLetter('a') is true

isLowerCase('a') is true
isUpperCase('a') is false
toLowerCase('T') is t
toUpperCase('q') is Q
```

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.
- The variable declared by a reference type is known as a ***reference variable*** that references an object.
- Here, **message** is a reference variable that references a string object with contents **Welcome to Java**.

Simple Methods for **String** Objects

- The **String** methods for obtaining string length, for accessing characters in the string, for concatenating strings, for converting a string to upper or lowercases, and for trimming a string.

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.

Simple Methods for **String** Objects

- The syntax to invoke an instance method is:

`referenceVariable.methodName(arguments)`

```
String message = "Hello";
message.charAt(2);
```

- The syntax to invoke a static method is:

`ClassName.methodName(arguments)`

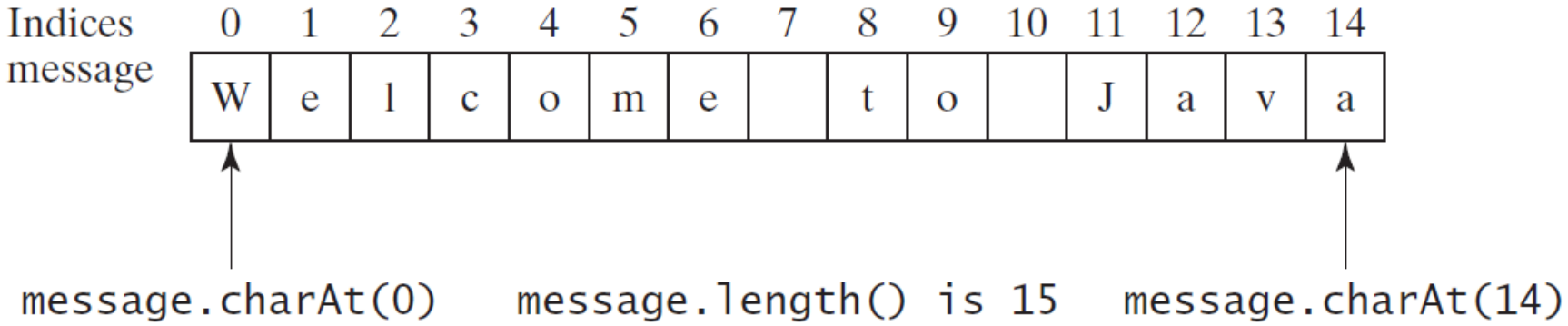
```
Math.pow(2,3);
```

Getting String Length

```
String message = "Welcome to Java";
System.out.println("The length of " + message + " is "
  + message.length());
```

```
System.out.println("The length of Welcome to Java is "
  + "Welcome to Java".length());
```

Getting Characters from a String




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

***NOTE**: Attempting to access characters in a string **s** out of bounds is a common programming error. To avoid it, make sure that you do not use an index beyond **s.length() – 1**. For example, **s.charAt(s.length())** would cause a **StringIndexOutOfBoundsException**.*

Converting Strings

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

- The **trim()** method returns a new string by eliminating whitespace (non-printed) characters from both ends of the string.
- The characters **' '**, **\t**, **\f**, **\r**, or **\n** are known as *whitespace characters*.

```
"   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 (casting)
String s = "Chapter" + 2; // s becomes Chapter2

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

//The augmented += operator can also be used
String message = "Welcome ";
message += "to " + "Java";
```

Reading a String from the Console

To read a string from the console, invoke the **next()** (ends with a whitespace) or **nextLine()** (ends with the Enter key pressed) methods on a **Scanner** object.

```
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);

System.out.println("Enter a line: ");
String s = input.nextLine();
System.out.println("The line entered is " + s);
```

Reading a Character from the Console

To read a character from the console, use the ***nextLine()*** method to read a string and then invoke the ***charAt(0)*** method on the string to return a character.

```
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

- Can we use the == operator to compare the contents of two strings?

```
if (string1 == string2)
  System.out.println("string1 and string2 are the same object");
else
  System.out.println("string1 and string2 are different objects");
```

- the == operator checks only whether string1 and string2 refer to the same object; it does not tell you whether they have the same content.

Comparing Strings

☞ The String class contains the following methods

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.
`contains(s1)`	Returns true if s1 is a substring in this string.

Comparing Strings

- The **equalsIgnoreCase** and **compareToIgnoreCase** methods ignore the case of the letters when comparing two strings.
- You can also use **str.startsWith(prefix)** to check whether string **str** starts with a specified prefix, **str.endsWith(suffix)** to check whether string **str** ends with a specified suffix, and **str.contains(s1)** to check whether string **str** contains string **s1** .

```
"Welcome to Java".startsWith("We") returns true.
"Welcome to Java".startsWith("we") returns false.
"Welcome to Java".endsWith("va") returns true.
"Welcome to Java".endsWith("v") returns false.
"Welcome to Java".contains("to") returns true.
"Welcome to Java".contains("To") returns false.
```

compareTo Method

- The **compareTo** method can be used to compare two strings.

```
s1.compareTo(s2)
```

- The method returns the value **0** if **s1** is equal to **s2**, a value less than **0** if **s1** is lexicographically (i.e., in terms of Unicode ordering) less than **s2**, and a value greater than **0** if **s1** is lexicographically greater than **s2**.

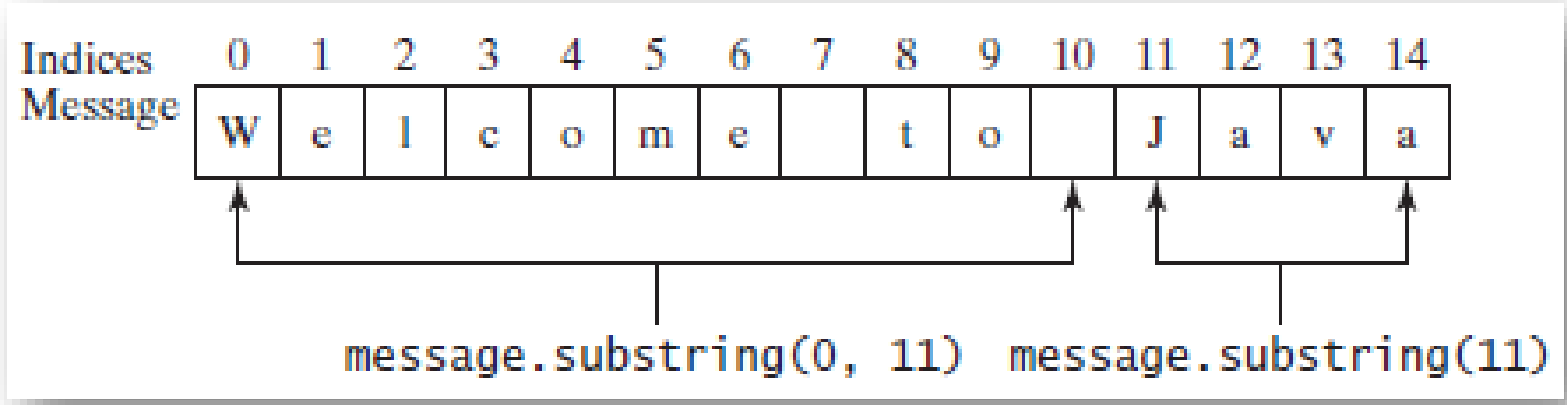
Order Two Cities

Intro to Java Programming, Y. Daniel Liang - OrderTwoCities.java (pearsoncmg.com)

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
`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 Note that the character at `endIndex` is not part of the substring.

```
String message = "Welcome to Java";
String message = message.substring(0, 11) + "HTML";
The string message now becomes Welcome to HTML.
```



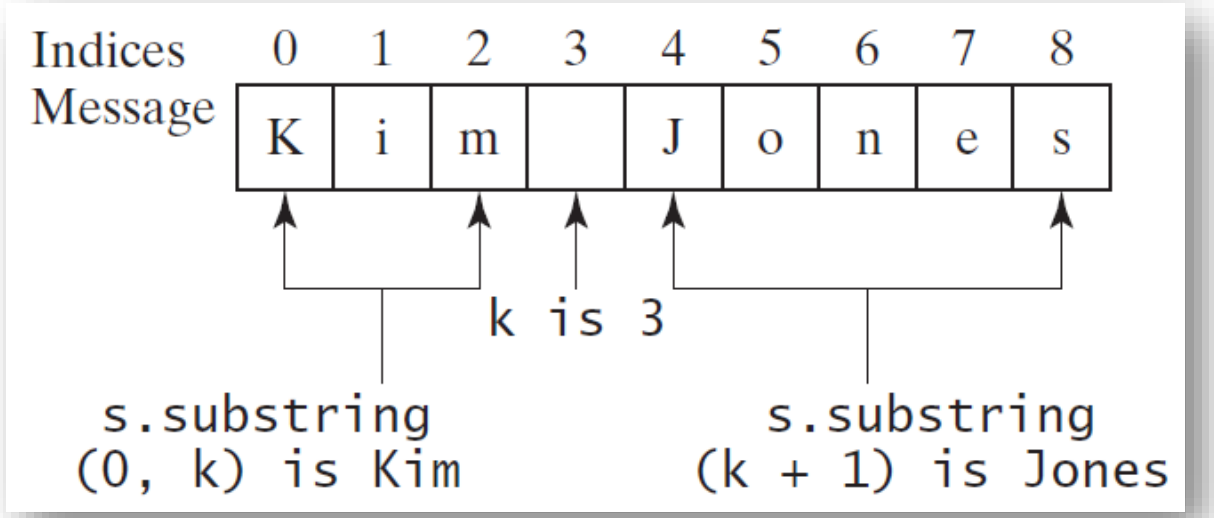
Finding a Character or a Substring in a String

The **String** class provides several versions of **indexOf** and **lastIndexOf** methods to find 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

Suppose a string **s** contains the first name and last name separated by a space. For example, if **s** is **Kim Jones:**




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

Conversion between Strings and Numbers

You can convert a numeric string into a number. To convert a string into an int value, use the Integer.parseInt method:

```
int intValue = Integer.parseInt(intString);
```

To convert a string into a double value, use the Double.parseDouble method:

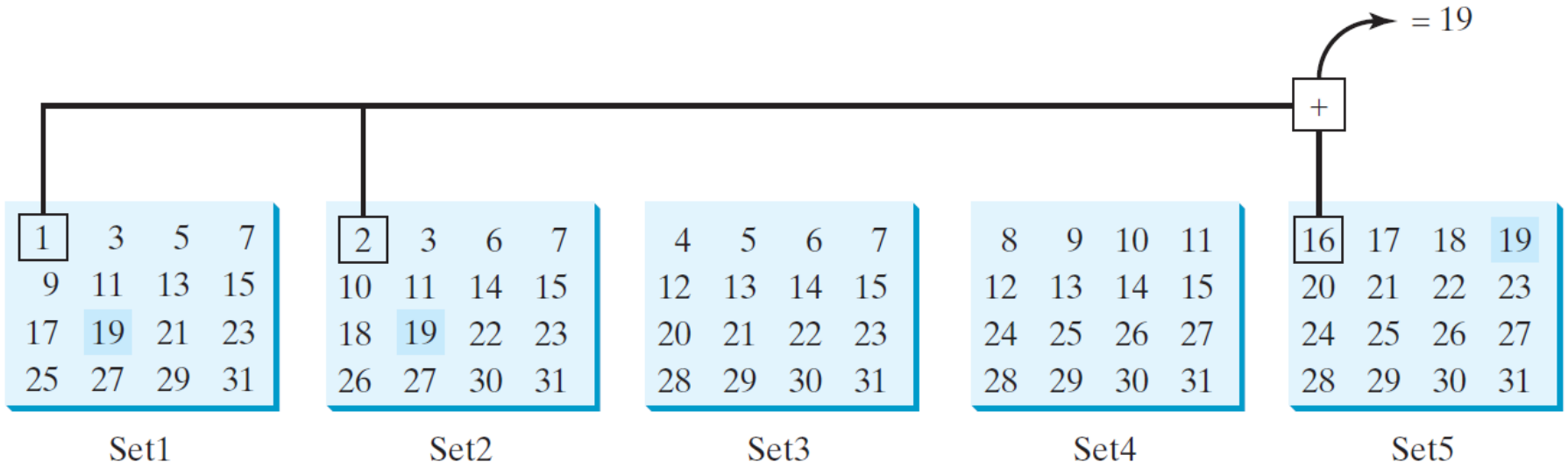
```
double doubleValue = Double.parseDouble(doubleString);
```

You can convert a number into a string, simply use the string concatenating operator

```
String s = number + "";
```

Problem: Guessing Birthday

The program can guess your birth date. Run to see how it works.



Guess Birthday

Intro to Java Programming, Y. Daniel Liang - GuessBirthday.java (pearsoncmg.com)

Mathematics Behind the Problem

1, 2, 4, 8, and 16, which correspond to 1, 10, 100, 1000, and 10000 in binary

Decimal	Binary
1	00001
2	00010
3	00011
...	
19	10011
...	
31	11111

(a)

$$
\begin{array}{r}
b_5\ 0\ 0\ 0\ 0 \\
b_4\ 0\ 0\ 0 \\
b_3\ 0\ 0 \\
b_2\ 0 \\
+\qquad b_1 \\
\hline
b_5\ b_4\ b_3\ b_2\ b_1
\end{array}
\qquad
\begin{array}{r}
10000 \\
10 \\
+\quad 1 \\
\hline
10011 \\
19
\end{array}
\qquad
\begin{array}{r}
10000 \\
1000 \\
100 \\
10 \\
+\quad 1 \\
\hline
11111 \\
31
\end{array}
$$

(b)

FIGURE 4.3 (a) A number between 1 and 31 can be represented using a five-digit binary number. (b) A five-digit binary number can be obtained by adding binary numbers 1, 10, 100, 1000, or 10000.

Problem: Converting a Hexadecimal Digit to a Decimal Value

- The hexadecimal number system has 16 digits: 0–9, A–F. The letters A, B, C, D, E, and F correspond to the decimal numbers 10, 11, 12, 13, 14, and 15.
- Write a program that converts a hexadecimal digit into a decimal value.

Hex Digit to Decimal

Intro to Java Programming, Y. Daniel Liang - HexDigit2Dec.java (pearsoncmg.com)

Case Study: Revising the Lottery Program Using Strings

Generates a random two-digit number, prompts the user to enter a two-digit number, and determines whether the user wins according to the following rule:

1. If the user input matches the lottery number in the exact order, the award is $10,000.
2. If all the digits in the user input match all the digits in the lottery number, the award is $3,000.
3. If one digit in the user input matches a digit in the lottery number, the award is $1,000.

Lottery

Intro to Java Programming, Y. Daniel Liang - LotteryUsingStrings.java (pearsoncmg.com)

Formatting Output

Use the printf statement:

```
System.out.printf(format, item1, item2, …, itemk);
```

Where format is a string that may consist of substrings and format specifiers.

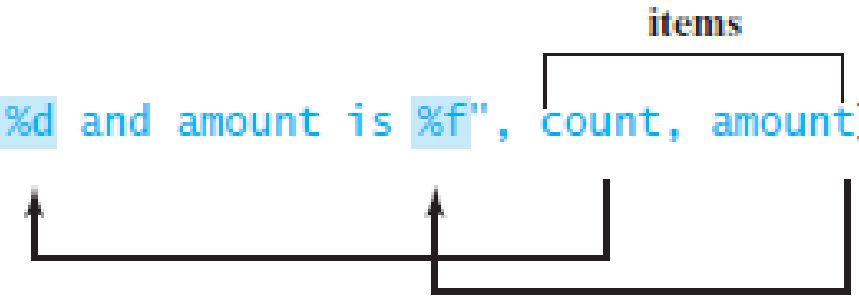
A format specifier specifies how an item should be displayed. An item may be a numeric value, character, boolean value, or a string.

Each specifier begins with a percent sign (%).

Frequently-Used Specifiers

Format 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
```

Formatting Example

The example gives a program that uses **printf** to display a table.

Format Demo

Intro to Java Programming, Y. Daniel Liang - FormatDemo.java (pearsoncmg.com)

LOOPS

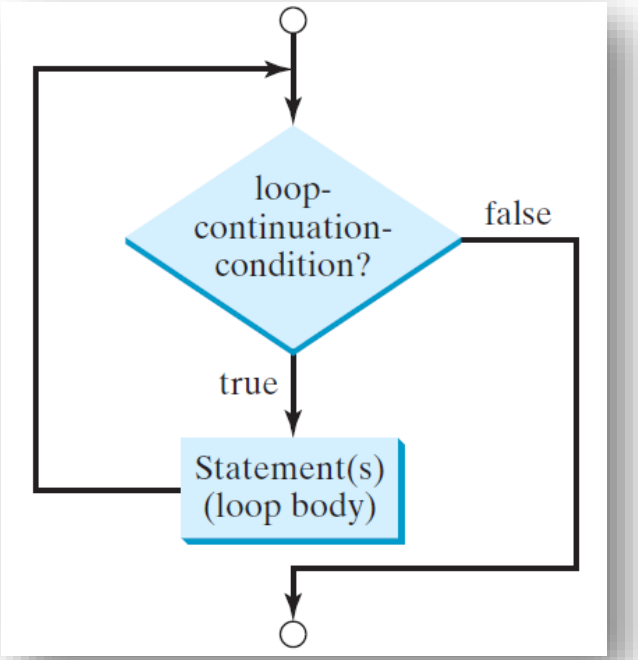
Introduction

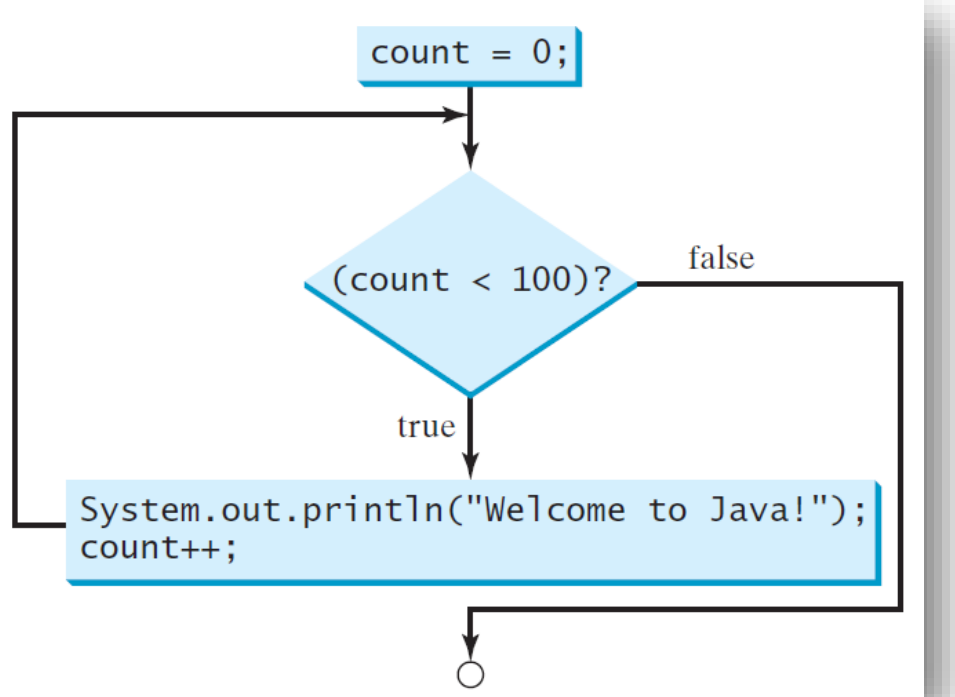
- Loop statements enable execution of the same statement (or statements) zero or more times according to a given condition expression.
- There are three kinds of loop statements in the Java programming language:
  - **while**
  - **do-while**
  - **for**

while Loop

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

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





Problem: Repeat Addition Until Correct

The program will work as follows:

1. Generate two numbers between 0 and 9, namely: **number1** and **number2**.
2. Prompt the student to answer, **"What is number1 + number2?"**
3. Check the student's answer and display whether the answer is correct
4. Let the user enter a new answer until it is correct.

Addition Quiz

Intro to Java Programming, Y. Daniel Liang - RepeatAdditionQuiz.java (pearsoncmg.com)

Loop Design Strategies

- Consider three steps when writing a loop:
  - ✓Step 1: Identify the statements that need to be repeated.
  - ✓Step 2: Wrap these statements in a loop like this:

```
while (true) {
      Statements;
}
```

  - ✓Step 3: Code the **loop-continuation-condition** and add appropriate statements for controlling the loop:

```
while (loop-continuation-condition) {
      Statements;
      Additional statements for controlling the loop;
}
```

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.

Sentinel Value

Intro to Java Programming, Y. Daniel Liang - SentinelValue.java (pearsoncmg.com)

Caution

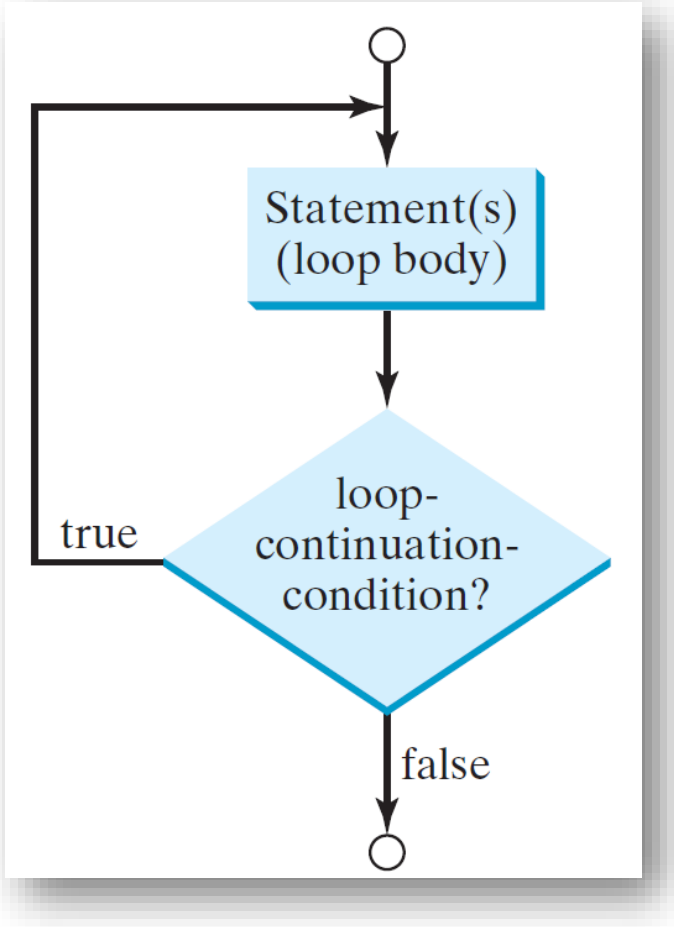
- Don’t use floating-point values for equality checking in a loop control. Since floating-point values are approximations for some values, using them could result in imprecise counter values and inaccurate results.
- Consider the following code for computing 1 + 0.9 + 0.8 + ... + 0.1:

```
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

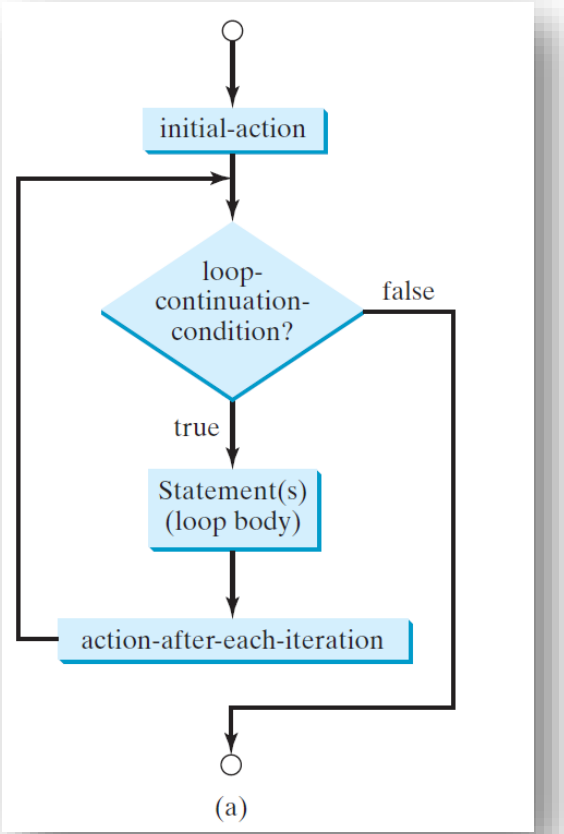
A **do-while** loop is the same as a **while** loop *except that it executes the loop body first and then checks* the loop continuation condition.

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



for Loop

```
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!");
}
```

If the loop control variable **is used only** in the loop, and not elsewhere, it is a good programming practice to declare it in the **initial-action** of the **for** loop.

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

`for` Loop Variations

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

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

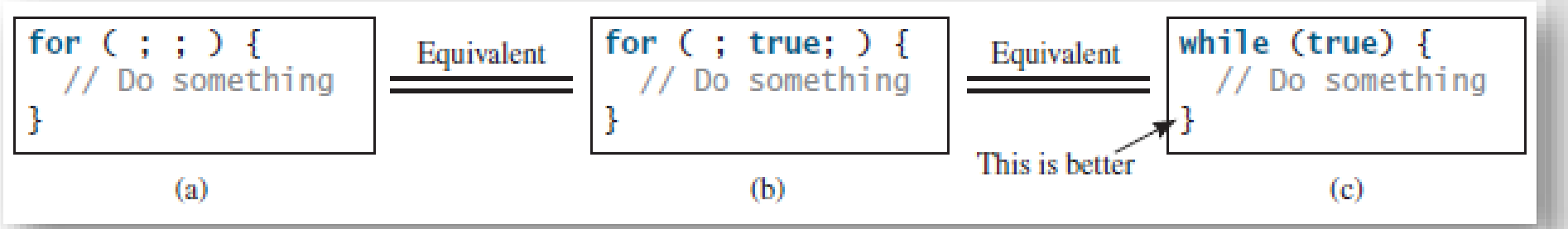
The **action-after-each-iteration** in a for loop can be a list of zero or more comma-separated statements.

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

However, they are rarely used in practice.

`for` Loop Variations

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 (c) to avoid confusion:



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);
}
```

Logic Error

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

Loop body is actually empty

Caution

Similarly, the following loop is also wrong:

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

Logic Error

In the case of the do loop, the following semicolon is needed to end the loop.

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

True

Which Loop to Use?

- Use the one that is most intuitive and comfortable for you. The three forms of loop statements expressively equivalent.
- 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.

Nested Loops

A loop can be nested inside another loop.

Write a program that uses nested for loops to print a multiplication table.

Multiplication Table

Intro to Java Programming, Y. Daniel Liang - MultiplicationTable.java (pearsoncmg.com)

Problem: Finding the Greatest Common Divisor

Write a program that prompts the user to enter two positive integers and finds their greatest common divisor. Suppose you enter two integers 4 and 2, their greatest common divisor is 2. Suppose you enter two integers 16 and 24, their greatest common divisor is 8. So, how do you find the greatest common divisor?

Solution:

- ✓ Let the two input integers be n1 and n2.
- ✓ You know number 1 is a common divisor, but it may not be the greatest common divisor.
- ✓ So you can check whether k (for k = 2, 3, 4, and so on) is a common divisor for n1 and n2, until k is greater than n1 or n2.

Greatest Common Divisor

Intro to Java Programming, Y. Daniel Liang - GreatestCommonDivisor.java (pearsoncmg.com)

Using `break`

The **break** *and* **continue** *keywords provide additional controls in a loop*:

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

Testing Break

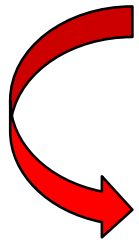
Intro to Java Programming, Y. Daniel Liang - TestBreak.java (pearsoncmg.com)

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);
  }
}
```



Guessing Number Problem with `break`

```
import java.util.Scanner;
public class GuessNumberUsingBreak {
  public static void main(String[] args) {
    // Generate a random number to be guessed
    int number = (int)(Math.random() * 101);

    Scanner input = new Scanner(System.in);
    System.out.println("Guess a magic number between 0 and 100");

    while (true) {
      // Prompt the user to guess the number
      System.out.print("\nEnter your guess: ");
      int guess = input.nextInt();

      if (guess == number) {
        System.out.println("Yes, the number is " + number);
        break;
      }
      else if (guess > number)
        System.out.println("Your guess is too high");
      else
        System.out.println("Your guess is too low");
    } // End of loop
  }
}
```

Using `continue`

- When **continue** is encountered, it ends the current iteration and program control goes to the end of the loop body.
- In other words, **continue** breaks out of the current iteration in the loop while the **break** keyword breaks out of a loop.

Testing Continue

Intro to Java Programming, Y. Daniel Liang - TestContinue.java (pearsoncmg.com)

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);
  }
}
```

Caution

- The **for** loop on the left is converted into the **while** loop on the right.

Are the outputs same?

```
int sum = 0;
for (int i = 0; i < 4; i++) {
  if (i % 3 == 0) continue;
  sum += i;
}
```

```
int i = 0, sum = 0;
while (i < 4) {
  if (i % 3 == 0) continue;
  sum += i;
  i++;
}
```

NO!!!

The **continue** statement is always inside a loop. In the **while** and **do-while** loops, the **loop-continuation-condition** is evaluated immediately after the **continue** statement. In the **for** loop, the **action-after-each-iteration** is performed, then the **loop-continuation-condition** is evaluated, immediately after the **continue** statement.

So how can we correct?...

Alternatives

```
int i = 0, sum = 0;
while (i < 4) {
if (i % 3 == 0) {
   i++;
   continue;
}
sum += i;
i++;
```

```
int i = 0, sum = 0;
while (i < 3) {
i++;
if (i % 3 == 0) continue;
sum += i;
```

Branching Statements

- **break**, and **continue** statements can control the execution of statements in **while**, **do-while**, and **for** blocks.
- There are two forms of these statements: plain and labeled.
  - Plain form is used to control execution of the current innermost statement block (classic use)
  - labeled form is used to control execution in nested statement blocks.
- A label can be given to a statement block as:

```
label:
control_flow_statement;
```

Branching Statements

break statement terminates the innermost block if used in the plain form, and terminates the specified block if used in labeled form.

```
...
for_loop:
for(int i=0; true; i++) {
        ...
        int j=0;
        while(true){
                j++;
                ...
                if(i==100) break for_loop;
                ...
                if(j>=200) break;
                ...
        }
}
```

causes outer for loop to terminate

causes inner while loop to terminate

Problem: Checking Palindromes

A string is a palindrome if it reads the same forward and backward. The words “mom,” “dad,” and “noon,” for instance, are all palindromes.

Solution:

- ✓ Check whether the first character in the string is the same as the last character. If so, check whether the second character is the same as the second-to-last character. And so on…
- ✓ This process continues until a mismatch is found or all the characters in the string are checked, except for the middle character if the string has an odd number of characters.

Palindrome

Intro to Java Programming, Y. Daniel Liang - Palindrome.java (pearsoncmg.com)

Problem: Displaying Prime Numbers

Write a program that displays the first 50 prime numbers in five lines, each of which contains 10 numbers. An integer greater than 1 is *prime* if its only positive divisor is 1 or itself. For example, 2, 3, 5, and 7 are prime numbers, but 4, 6, 8, and 9 are not.

Solution: The problem can be broken into the following tasks:

- ✓ For numbers 2, 3, 4, 5, 6, ..., test whether the number is prime.
- ✓ Determine whether a given number is prime.
- ✓ Count the prime numbers.
- ✓ Print each prime number, and print 10 numbers per line.

Prime Number

Intro to Java Programming, Y. Daniel Liang - PrimeNumber.java (pearsoncmg.com)

As a Summary…

- The **while** loop and the **do-while** loop often are used when the number of repetitions is not predetermined.
- The **for** loop generally is used to execute a loop body a fixed number of times.
- The **break** keyword immediately ends the innermost loop, which contains the break.
- The **continue** keyword only ends the current iteration.