Selections, and Mathematical Functions, Characters, and Strings

Introduction to Programming and Computational Problem Solving - 2

CSE 8B

Lecture 3

Announcements

- Assignment 1 is due Oct 6, 11:59 PM
- Quiz 1 is Oct 8
- Assignment 2 will be released Oct 6
  - Due Oct 13, 11:59 PM
- Educational research study
  - Oct 8, weekly survey
- Reading
  - Liang
    - Chapters 3 and 4

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`
- For example

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

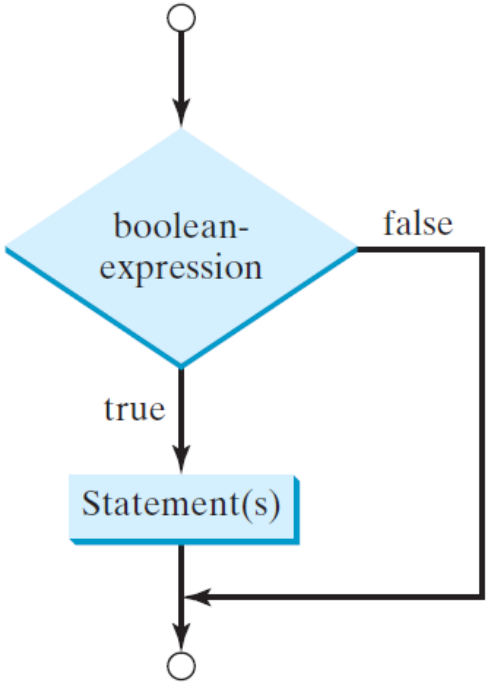
Relational operators

Java Operator	Mathematics Symbol	Name	Example (radius is 5)	Result
`<`	<	less than	`radius < 0`	`false`
`<=`	≤	less than or equal to	`radius <= 0`	`false`
`>`	>	greater than	`radius > 0`	`true`
`>=`	≥	greater than or equal to	`radius >= 0`	`true`
`==`	=	equal to	`radius == 0`	`false`
`!=`	≠	not equal to	`radius != 0`	`true`

`if` statements

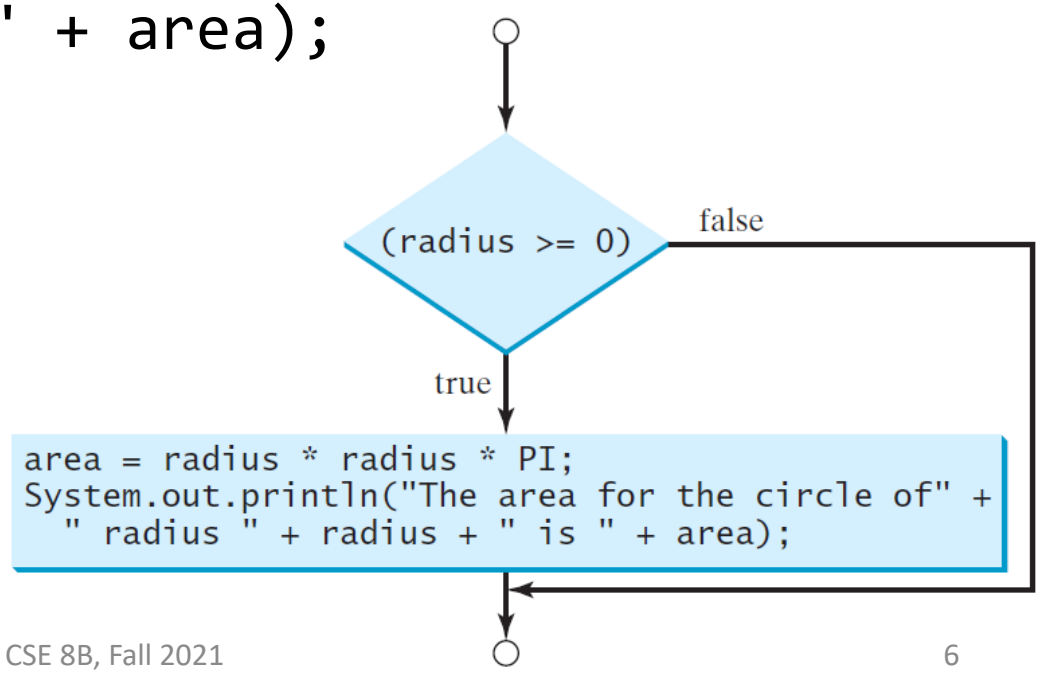
```
if (boolean-expression) {
  statement(s);
}
```

Braces are optional for a single statement; however, it is best practice (less error prone) to **always use braces**



`if` statements

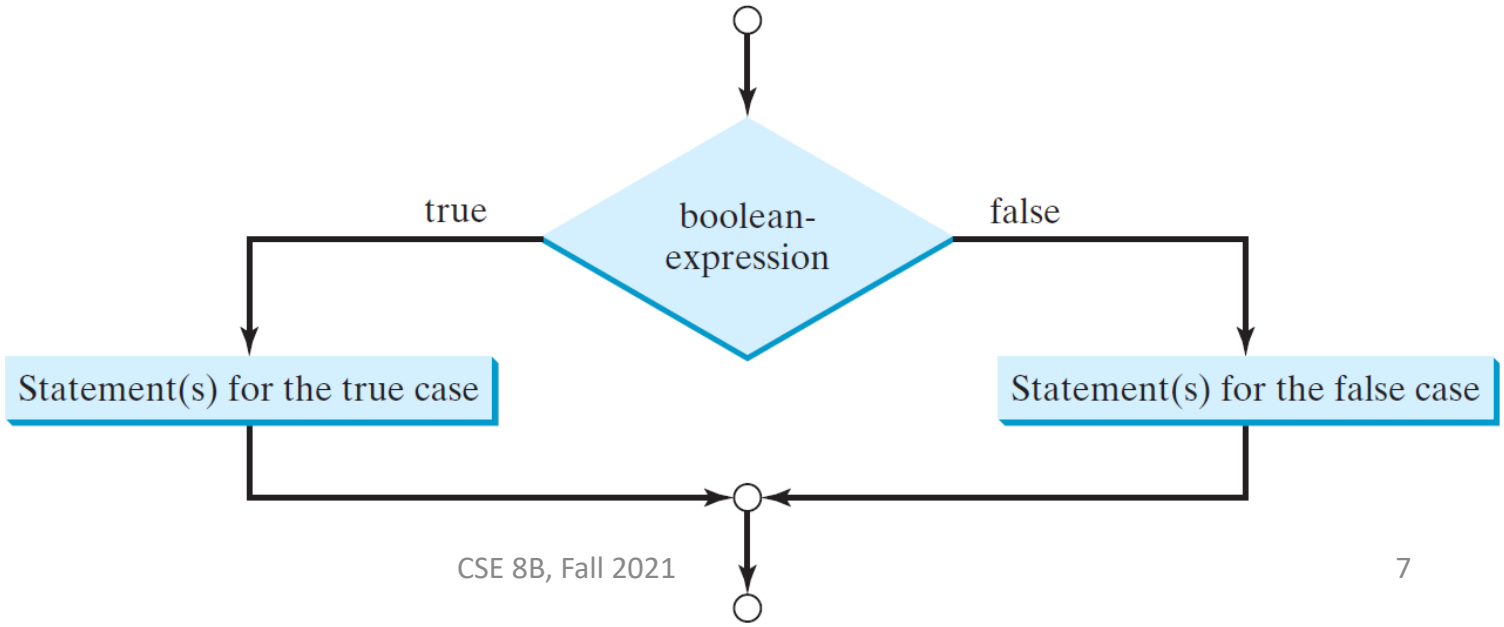
```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area for
  the circle of radius "
    + radius + " is " + area);
}
```



if-else statements

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

Braces are optional for a single statement; however, it is best practice (less error prone) to **always use braces**



if-else statements

```
if (radius >= 0) {
  area = radius * radius * 3.14159;

  System.out.println("The area for the "
    + "circle of radius " + radius +
    " is " + area);
}
else {
  System.out.println("Negative input");
}
```

Conditional operator

(boolean-expression) ? expression1 : expression2

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

is equivalent to

```
y = (x > 0) ? 1 : -1;
```

Multiple `if-else` 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)

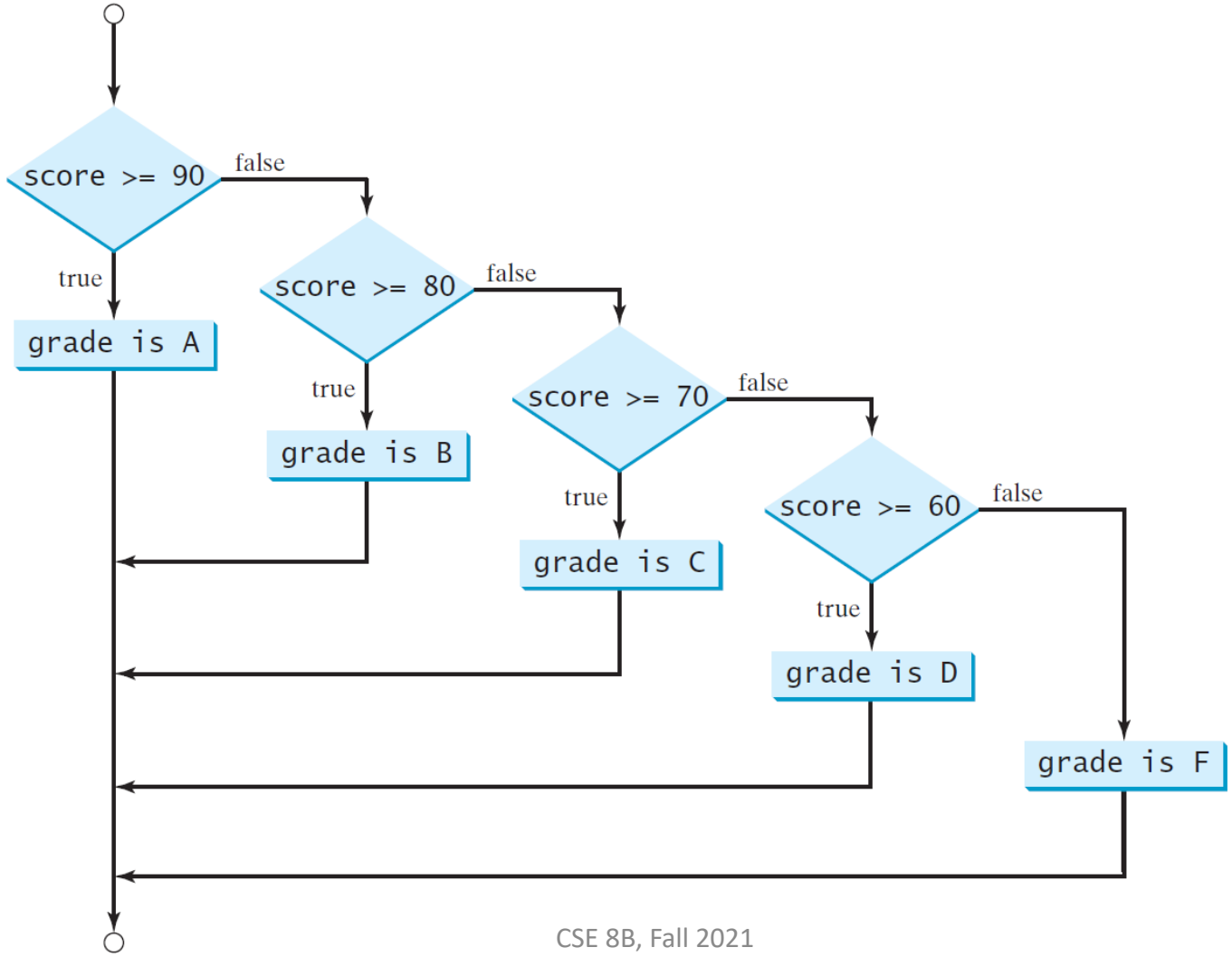
Equivalent

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");
```

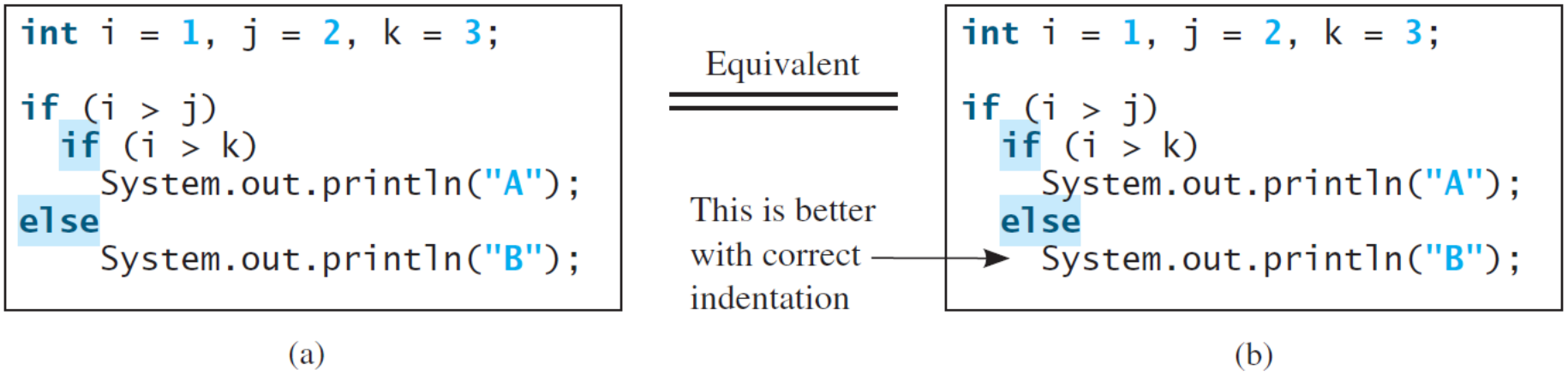
(b)

Multiple `if-else` statements



Nested statements

- The `else` clause matches the most recent `if` clause in the same block



Braces are optional for a single statement; however, it is best practice (less error prone) to **always use braces**

Nothing is printed

Nested statements

- 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");
```

Braces are optional for a single statement; however, it is best practice (less error prone) to **always use braces**

B is printed

Less error prone

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

(a)

Equivalent

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

(b)

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

(a)

Equivalent

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

(b)

Logical operators

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

Truth table for operator !

p	!p	Example: `age = 24` and `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:** `age = 24` **and** `weight = 140`
`false`	`false`	`false`	`(age <= 18) && (weight < 140)` is `false`, because both conditions are `false`
`false`	`true`	`false`	`(age <= 18) && (weight >= 140)` is `false`, because `(age <= 18)` is `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 conditions are `true`

Truth table for operator ||

p_1	p_2	p_1 || p_2	**Example:** `age = 24` **and** `weight = 140`
`false`	`false`	`false`	`(age > 34) || (weight >= 150)` is `false`, because both conditions are `false`
`false`	`true`	`true`	`(age > 34) || (weight <= 140)` is `true`, because `(weight <= 140)` is `true`
`true`	`false`	`true`	`(age > 14) || (weight >= 150)` is `false`, because `(age > 14)` is `true`
`true`	`true`	`true`	`(age > 14) || (weight <= 140)` is `true`, because both conditions are `true`

Truth table for operator ^

p_1	p_2	p_1 ^ p_2	**Example:** `age = 24` **and** `weight = 140`
`false`	`false`	`false`	`(age > 34) ^ (weight > 140)` is `false`, because both conditions are `false`
`false`	`true`	`true`	`(age > 34) ^ (weight >= 140)` is `true`, because `(age > 34)` is `false` and `(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`	`(age > 14) ^ (weight >= 140)` is `false`, because both conditions are `true`

switch statements

- When the value in a case statement matches the value of the switch-expression, the statements starting from this case are executed until either a break statement or the end of the switch statement is reached

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

`switch` statements

- The switch-expression must yield a value of `char`, `byte`, `short`, `int` or `String` type and must always be enclosed in parentheses
- The value1, ..., and valueN must have the *same data type* as the value of the switch-expression
- The resulting statements in the `case` statement are executed when the value in the `case` statement matches the value of the switch-expression
- Note that value1, ..., and valueN are *constant expressions* (i.e., they cannot contain variables in the expression, such as `1 + x`)

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

`switch` statements

- The keyword `break` is optional, but it should be used at the end of each case in order to terminate the remainder of the `switch` statement
  - If the `break` statement is not present, the next case statement will be executed
- The `default` case, which is optional, can be used to perform actions when none of the specified cases matches the switch-expression

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

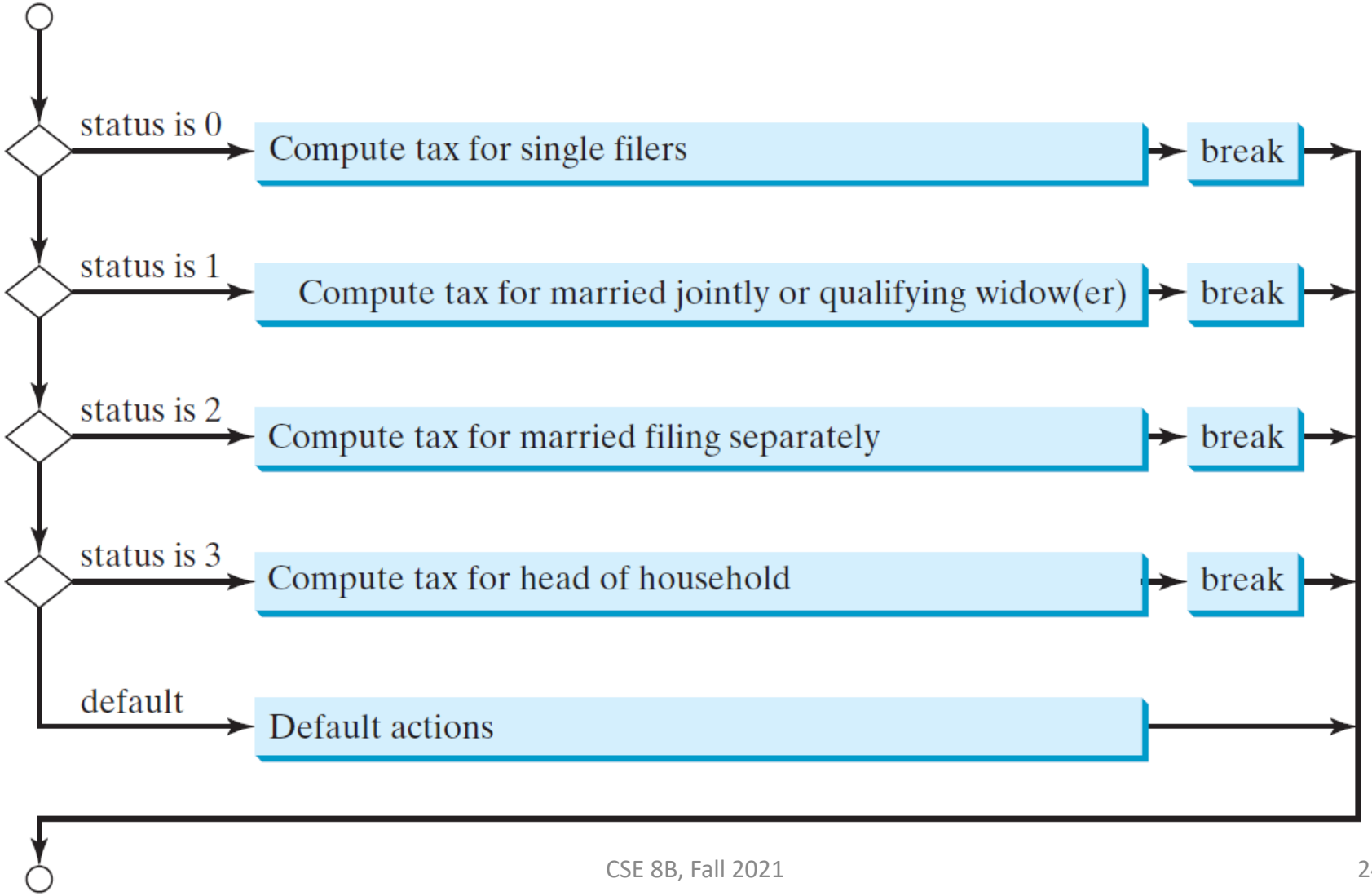
The default case is optional; however, it is best practice (less error prone) to **always have a default case**

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("Error: invalid status");
           System.exit(1);
}
```

The default case is optional; however, it is best practice (less error prone) to **always have a default case**

switch statements



switch statements

```
switch (day) {
  case 1:
  case 2:
  case 3:
  case 4:
  case 5:
System.out.println("Weekday");
break;
  case 0:
  case 6:
System.out.println("Weekend");
}
```

Fall-through

Fall-through

Operator precedence

- `()`, `var++`, `var--`
- `++var`, `--var`, `+`, `-` (unary plus and minus), `!` (not)
- (type) casting
- *, /, % (multiplication, division, and remainder)
- +, - (binary addition and subtraction)
- <, <=, >, >= (relational operators)
- ==, != (equality)
- ^ (exclusive OR)
- && (AND)
- || (OR)
- =, +=, -=, *=, /=, %= (assignment operators)

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

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

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

Mathematical functions

- Java provides many useful methods in the `Math` class for performing common mathematical functions
- `Math` class constants
  `PI`
  `E`
- Math class methods
  - Trigonometric methods
  - Exponent methods
  - Rounding methods
  - `min`, `max`, `abs`, and `random` methods

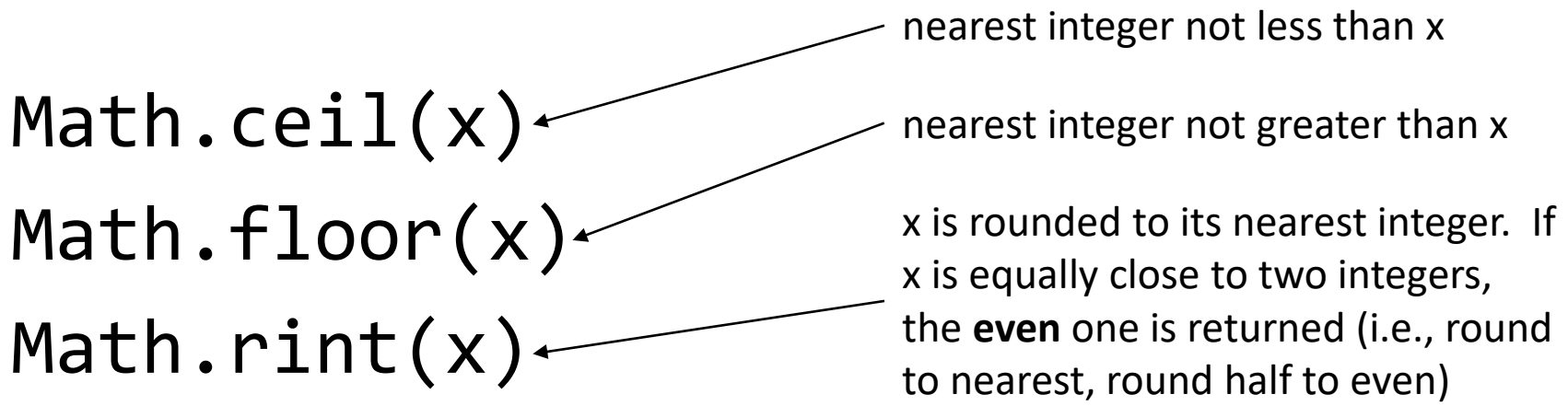
Trigonometric methods

```
Math.toDegrees(radians)
Math.toRadians(degrees)
Math.sin(radians)
Math.cos(radians)
Math.tan(radians)
Math.acos(a)
Math.asin(a)
Math.atan(a)
```

Exponent methods

`Math.exp(a)`	e^a
`Math.log(a)`	$\log_e(a)$
`Math.log10(a)`	$\log_{10}(a)$
`Math.pow(a, b)`	a^b
`Math.sqrt(a)`	

Rounding methods



- If you want to return an integer type, then

  `int Math.round(float x)`

  - Returns `(int)Math.floor(x + 0.5f)`

  `long Math.round(double x)`

  - Returns `(long)Math.floor(x + 0.5)`

`min`, `max`, `abs`, and `random` methods

```
Math.min(a, b)
Math.max(a, b)
Math.abs(a)
Math.random()
```

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

char data type

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

- 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`
  - Unicode can represent 65536 characters

Common and special 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 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 data types

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

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

Comparing and testing characters

```
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 `char` class

Method	Description
`isDigit(ch)`	Returns true if the specified character is a digit.
`isLetter(ch)`	Returns true if the specified character is a letter.
`isLetterOfDigit(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.

`String` type

- The `char` type only represents one character
- To represent a string of characters, use the `String` type
- `String` is a predefined class in the Java library (just like the System class and Scanner class)

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

- 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

Simple `String` methods

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.

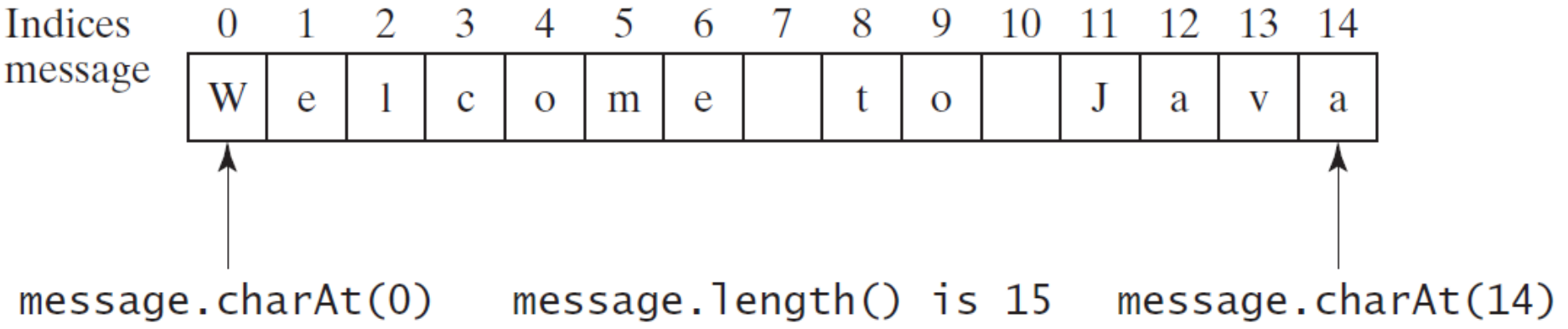
- These methods can only be invoked from a specific string instance
  - These methods are called instance methods

Instance methods vs static methods

- These methods can only be invoked from a specific string instance
  - These methods are called instance methods
  - The syntax to invoke an instance method is
    `referenceVariable.methodName(arguments)`
- A non-instance method is called a static method
  - **A static method can be invoked without using an object** (i.e., they are not tied to a specific object instance)
  - For example, all the methods defined in the `Math` class are static methods

Getting characters from a string

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



String concatenation

```
String s3 = s1.concat(s2); // These two are
String s3 = s1 + s2;       // equivalent

// 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

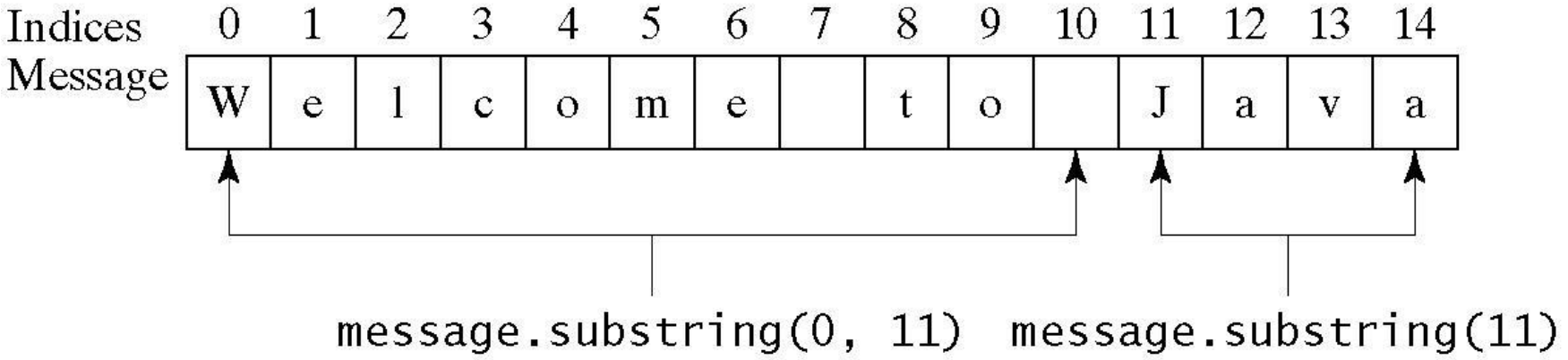
```
Scanner input = new Scanner(System.in);
System.out.print("Enter a character: ");
String s = input.nextLine();
char ch = s.charAt(0);
System.out.println("The character entered is " + ch);
```

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.

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.

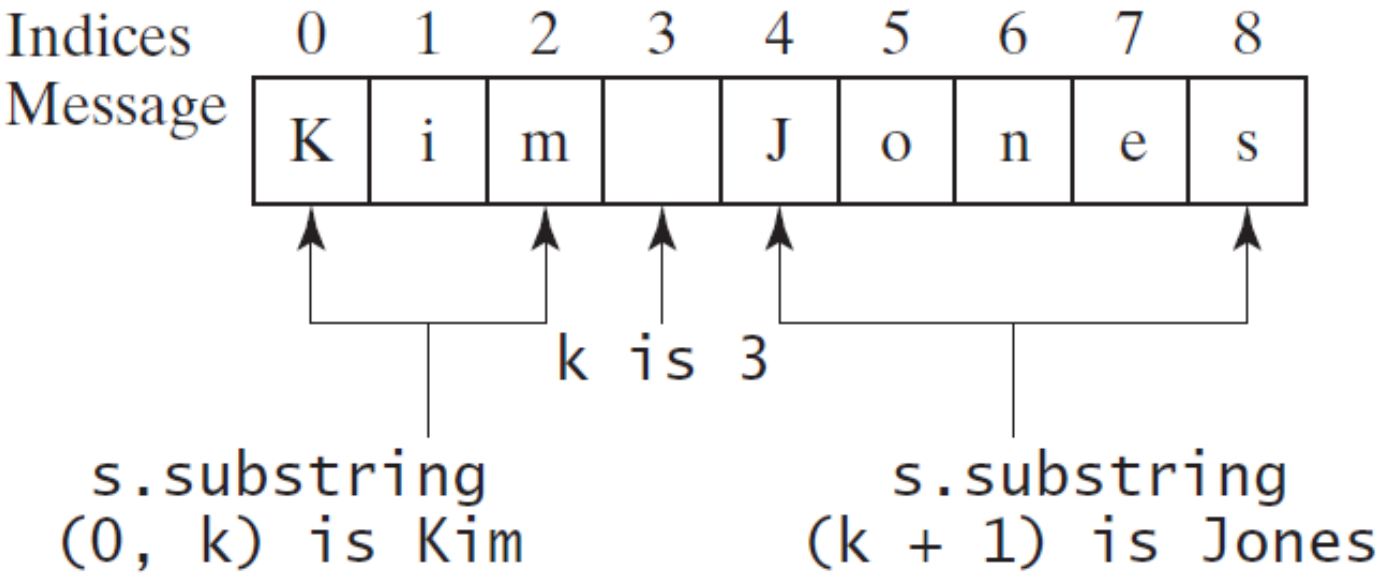


Finding a character or a substring in a string

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

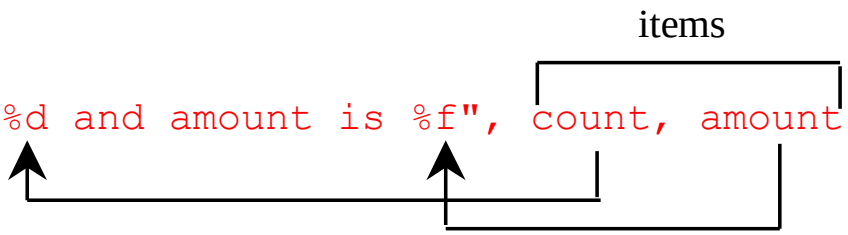
  ```
  System.out.printf(format, items);
  ```

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

Common 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
```

Next Lecture

- Loops
- Methods
- Reading
  - Liang
    - Chapters 5 and 6