# Selections

Introduction to Programming and Computational Problem Solving - 2

CSE 8B

Lecture 5

# Announcements

- Assignment 2 is due Apr 19, 11:59 PM
  - Upgrade beginning Apr 22, 12:01 AM
- Assignment 3 will be released Apr 19
  - Due Apr 26, 11:59 PM
- Educational research study
  - Apr 21, weekly survey

# Selections

- Relational operators (e.g., less than, equal to)
- Logical operators (e.g., not, and, or)
- `if` statements
- `if-else` statements
- `switch` statements

# 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);
```

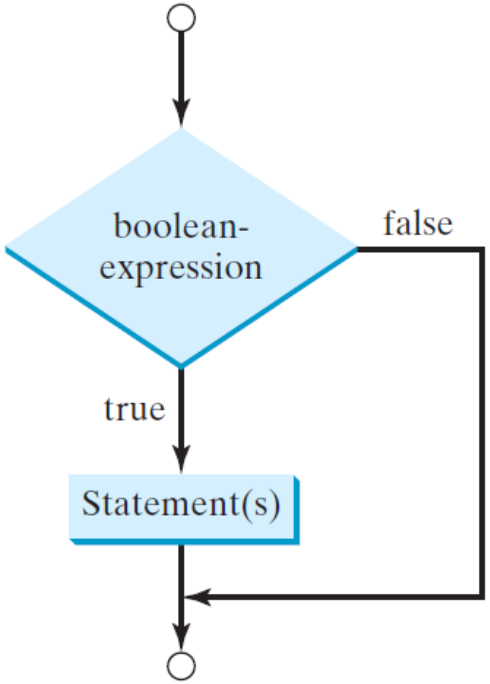
# Relational operators

| 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

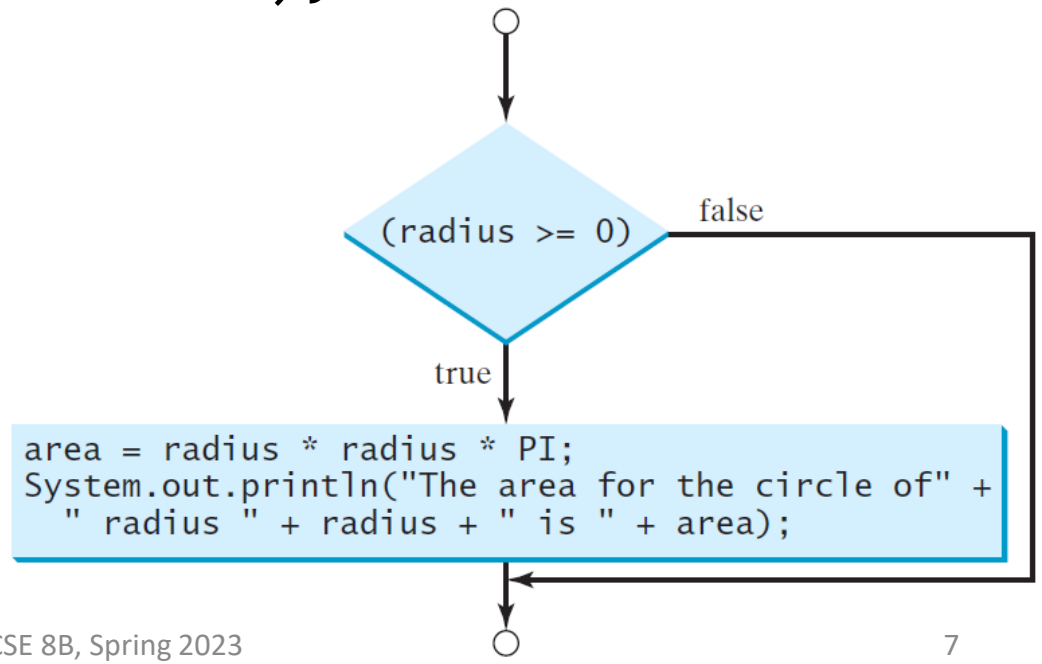
```
if (boolean-expression) {
  statement(s);
}
```

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



# `if` statements

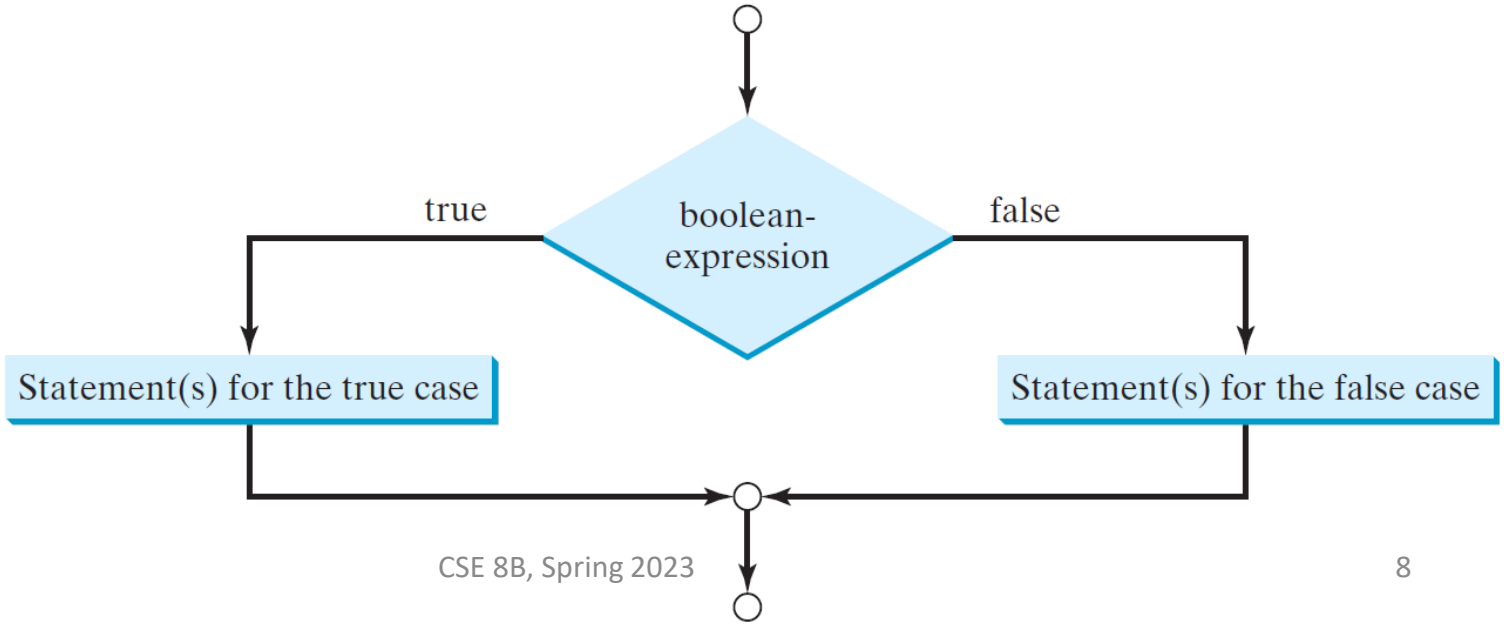
```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area for
  the circle of radius "
    + radius + " is " + area);
}
```



# 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

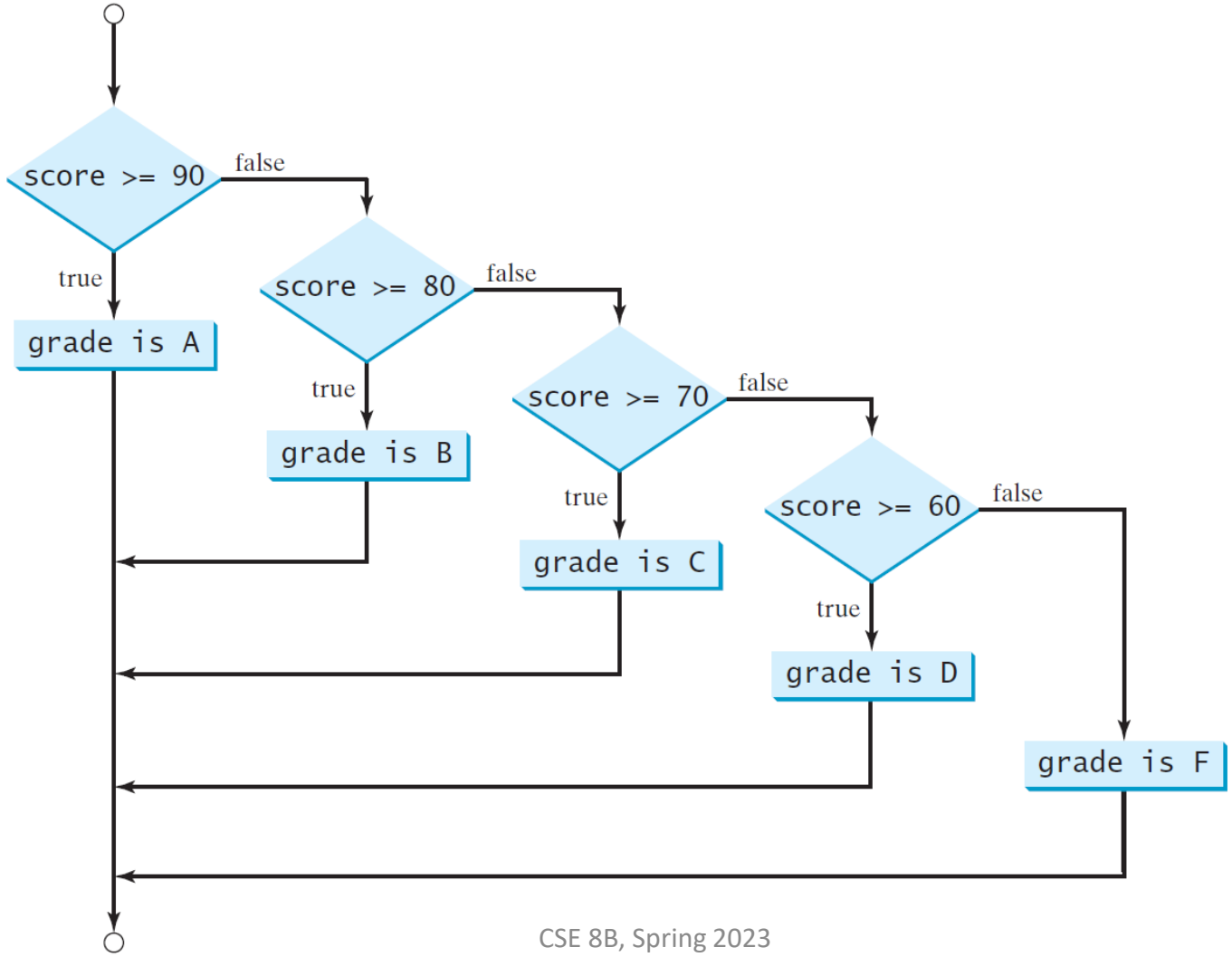
```
(boolean-expression) ? expression1 : expression2
```

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

is equivalent to

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

# Multiple `if-else` statements



# 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

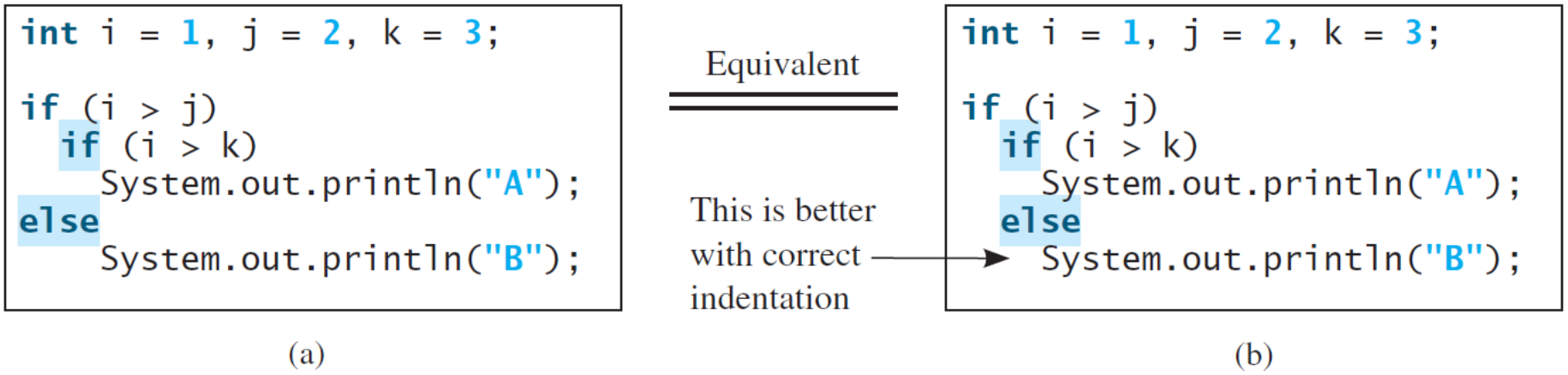
This is better

```
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)

# 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`,<br>because `(age > 18)` is `true` |
| `false` | `true` | `!(weight == 150)` is `true`,<br>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` |

# Short-circuit operators

- && and || are short-circuit operators
- `p1 && p2`
  - If `p1` or `p2` is `false`, then `p1 && p2` is `false`
  - `p1` is evaluated first
    - If `p1` is `true`, then `p2` **is** evaluated
    - If `p1` is `false`, then `p2` **is not** evaluated
- `p1 || p2`
  - If `p1` or `p2` is `true`, then `p1 || p2` is `true`
  - `p1` is evaluated first
    - If `p1` is `true`, then `p2` **is not** evaluated
    - If `p1` is `false`, then `p2` **is** evaluated

# `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;
}
```

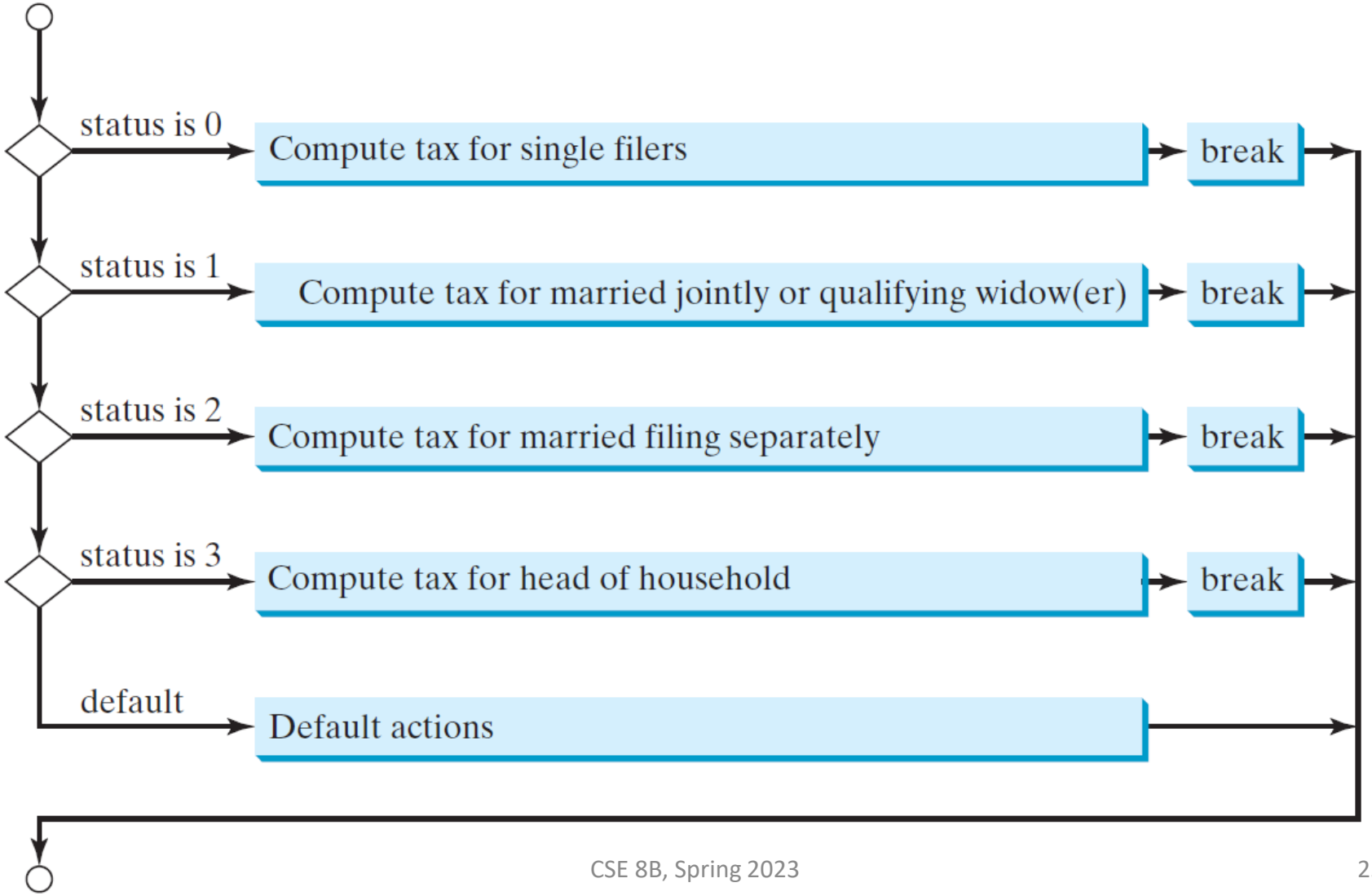
# `switch` statements

- 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 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 (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

# Next Lecture

- Methods