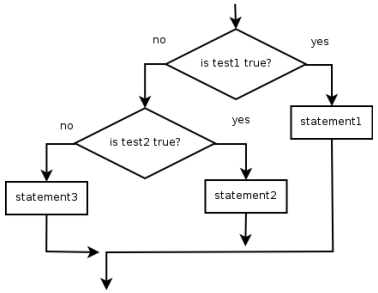
AP Computer Science

Booleans, Strings

Structures

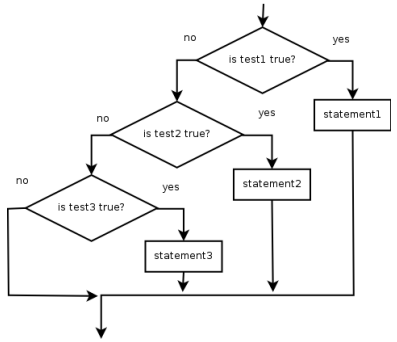
- Exactly 1 path: (mutually exclusive)

```
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else {
    statement(s);
}
```



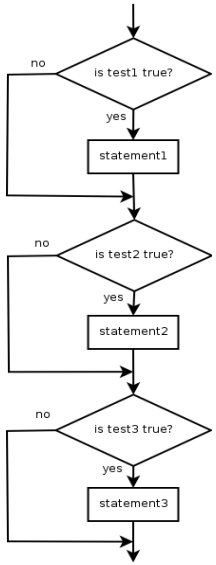
- 0 or 1 path:

```
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else if (test) {
    statement(s);
}
```



- 0, 1, or many paths: (independent tests, not exclusive)

```
if (test) {
    statement(s);
}
if (test) {
    statement(s);
}
if (test) {
    statement(s);
}
```



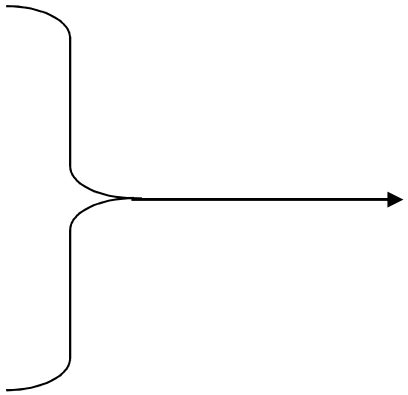
Which nested `if/else`?

- **(1) if/if/if (2) nested if/else (3) nested if/else/if**
  - Whether a user is lower, middle, or upper-class based on income.
    - **(2)** nested `if / else if / else`
  - Whether you made the dean's list (GPA ≥ 3.8) or honor roll (3.5-3.8).
    - **(3)** nested `if / else if`
  - Whether a number is divisible by 2, 3, and/or 5.
    - **(1)** sequential `if / if / if`
  - Computing a grade of A, B, C, D, or F based on a percentage.
    - **(2)** nested `if / else if / else if / else if / else`

Factoring `if/else` code

- **factoring**: extracting common/redundant code
  - Factoring `if/else` code can reduce the size of `if/else` statements or eliminate the need for `if/else` altogether.
- Example:

```
if (a == 1) {
    x = 3;
} else if (a == 2) {
    x = 6;
    y++;
} else {  // a == 3
    x = 9;
}
```



```
x = 3 * a;
if (a == 2) {
    y++;
}
```

Code in need of factoring

```
if (money < 500) {
    System.out.println("You have, $" + money + " left.");
    System.out.print("Caution!  Bet carefully.");
    System.out.print("How much do you want to bet? ");
    bet = console.nextInt();
} else if (money < 1000) {
    System.out.println("You have, $" + money + " left.");
    System.out.print("Consider betting moderately.");
    System.out.print("How much do you want to bet? ");
    bet = console.nextInt();
} else {
    System.out.println("You have, $" + money + " left.");
    System.out.print("You may bet liberally.");
    System.out.print("How much do you want to bet? ");
    bet = console.nextInt();
}
```

Code after factoring

```
System.out.println("You have, $" + money + " left.");
if (money < 500) {
    System.out.print("Caution!  Bet carefully.");
} else if (money < 1000) {
    System.out.print("Consider betting moderately.");
} else {
    System.out.print("You may bet liberally.");
}
System.out.print("How much do you want to bet? ");
bet = console.nextInt();
```

- If the start of each branch is the same, move it *before* the `if/else`.
- If the end of each branch is the same, move it *after* the `if/else`.
- If similar but code exists in each branch, look for patterns.

The "dangling `if`" problem

- What can be improved about the following code?

```
if (x < 0) {
    System.out.println("x is negative");
} else if (x >= 0) {
    System.out.println("x is non-negative");
}
```

- The second `if` test is unnecessary and can be removed:

```
if (x < 0) {
    System.out.println("x is negative");
} else {
    System.out.println("x is non-negative");
}
```

  - This is also relevant in methods that use `if` with `return`...

`if/else` with `return`

- Methods can return different values using `if/else`:

```
// Returns the largest of the three given integers.
public static int max3(int a, int b, int c) {
    if (a >= b && a >= c) {
        return a;
    } else if (b >= c && b >= a) {
        return b;
    } else {
        return c;
    }
}
```

  - Whichever path the code enters, it will return the appropriate value.
  - Returning a value causes a method to immediately exit.
  - All code paths must reach a `return` statement.
    - All paths must also return a value of the same type.

All paths must return

```
public static int max3(int a, int b, int c) {
    if (a >= b && a >= c) {
        return a;
    } else if (b >= c && b >= a) {
        return b;
    }
    // Error: not all paths return a value
}
```

- The following also does not compile:

```
public static int max3(int a, int b, int c) {
    if (a >= b && a >= c) {
        return a;
    } else if (b >= c && b >= a) {
        return b;
    } else if (c >= a && c >= b) {
        return c;
    }
}
```

  - The compiler thinks `if/else/if` code might skip all paths.

Logical operators: &&, ||, !

- Conditions can be combined using *logical operators*:

Operator	Description	Example	Result
`&&`	and	`(2 == 3) && (-1 < 5)`	`false`
`||`	or	`(2 == 3) || (-1 < 5)`	`true`
`!`	not	`!(2 == 3)`	`true`

- "Truth tables" for each, used with logical values *p* and *q*:

p	q	p && q	p || q
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

p	!p
`true`	`false`
`false`	`true`

Evaluating logic expressions

- Relational operators have lower precedence than math.

```
5 * 7 >= 3 + 5 * (7 - 1)
5 * 7 >= 3 + 5 * 6
35    >= 3 + 30
35    >= 33
true
```

- Relational operators cannot be "chained" as in algebra.

```
2 <= x <= 10          (assume that x is 15)
true   <= 10
   error!
```

  - Instead, combine multiple tests with && or | |

```
2 <= x && x <= 10     (assume that x is 15)
true   && false
   false
```

Logical questions

- What is the result of each of the following expressions?

```
int x = 42;
int y = 17;
int z = 25;
```

  - `y < x && y <= z`
  - `x % 2 == y % 2 || x % 2 == z % 2`
  - `x <= y + z && x >= y + z`
  - `!(x < y && x < z)`
  - `(x + y) % 2 == 0 || !((z - y) % 2 == 0)`

- Answers: `true`, `false`, `true`, `true`, `false`