Program Structures

Principles of straight-line code

- **Make dependences obvious:**

  (e.g., through passing arguments, return values)

```
firstResult = doThing1();
secondResult = doThingY(firstResult);
```

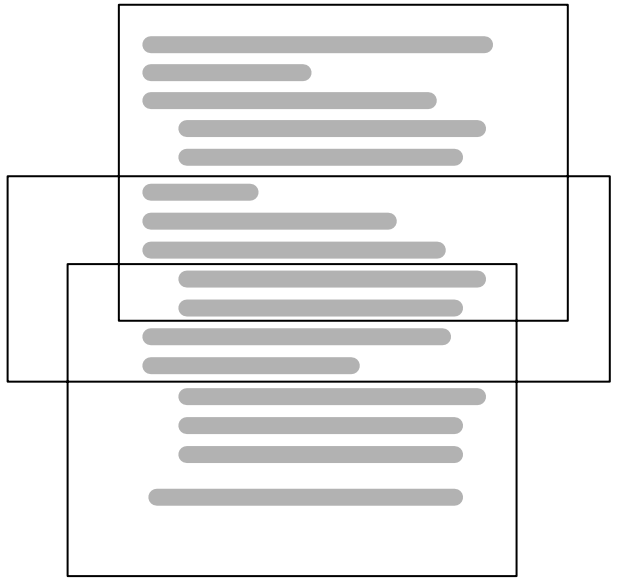
- **Vs.**

```
doThing1();
doThingY();
```

Principles of straight-line code, cont.

- **If no dependences, group related statements**
  - If you were to draw boxes around related statements

Good



Poor

Grouping related items (example)

- Ordering implicit, but emphasizes grouping

```
MarketingData marketingData = new MarketingData();
marketingData.ComputeQuarterly();
marketingData.ComputeAnnual();
marketingData.Print();

SalesData salesData = new SalesData();
salesData.ComputeQuarterly();
salesData.ComputeAnnual();
salesData.Print();
```

Which is better?

A

```
if (!done) {
   ...
}
```

B

```
if (done == false) {
   ...
}
```

C Control flow is fine for both

D Control flow is problematic for both

Which is better?

A

```
if (!task.isDone()) {
   task.restart();
} else {
   toDoList.markCompleted(task);
}
```

B

```
if (task.isDone()) {
   toDoList.markCompleted(task);
} else {
   task.restart();
}
```

C Control flow is fine for both

D Control flow is problematic for both

Which is best?

A

```
if (getAmountOfGasInTank() >= gasNeeded(destination)) {
  // avoid unnecessary stops; reduce wear on engine
} else {
  fillGasTank();
}
```

B

```
if (getAmountOfGasInTank() < gasNeeded(destination)) {
  fillGasTank();
} else {
  // avoid unnecessary stops; reduce wear on engine
}
```

C

```
if (gasNeeded(destination) < getAmountOfGasInTank()) {
  // avoid unnecessary stops; reduce wear on engine
} else {
  fillGasTank();
}
```

D

```
if (gasNeeded(destination) >= getAmountOfGasInTank()) {
  fillGasTank();
} else {
  // avoid unnecessary stops; reduce wear on engine
}
```

Principles of if/else

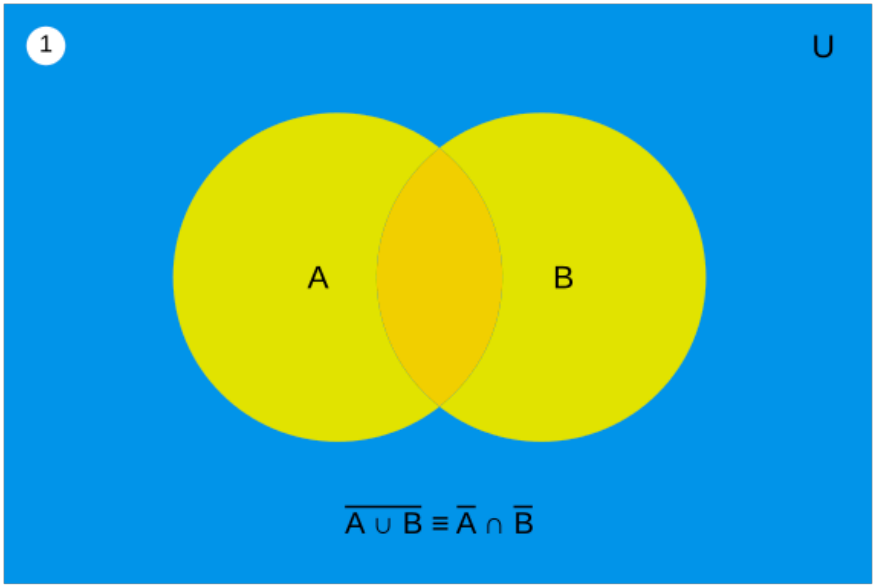
- **write the common case first; then write the unusual cases**
  - More precisely, cover first the case that will reduce the reader's cognitive load
- **Encode complex Boolean expressions in methods**
  - Naming documents the meaning of the expression
  - Even if the method is only called in one place
- **Use case/switch only when it applies**
  - Don't do nasty things with it

What is wrong with this code?

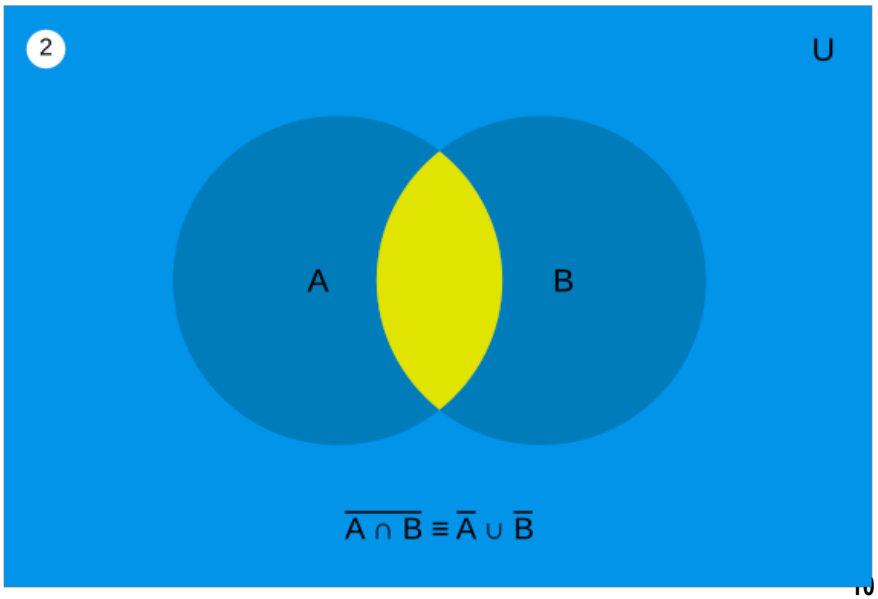
```
/**
 *
 * @param input a String
 * @return the input String with dashes and the characters
 *         following the dashes removed
 */
public String removeDashesAndFollowingChars(String input) {
    StringBuilder output = new StringBuilder();

    for (int i = 0; i < input.length(); i++) {
        char currentChar = input.charAt(i);
        if (currentChar != '-') {
            output.append(currentChar);
        } else {
            i++;
        }
    }
    return output.toString();
}
```

Power of De Morgan's Theorem

the complement of the union of two sets is the same as the intersection of their complements; and



the complement of the intersection of two sets is the same as the union of their complements.



De Morgan's Law in practice

- Simplify expression to avoid double negatives
- Instead of:

```
if (!(printer.hasPower() && !printer.hasPaper())) {
```

- Write:

```
if (!printer.hasPower() || printer.hasPaper()) {
```

Which is better?

A

```
public static Map<Integer, Integer> generateHistogram2(int[] data) {
    Map<Integer, Integer> histogram = new HashMap<Integer, Integer>();
    for (int value : data) {
        int count = 1 +
            (histogram.containsKey(value) ? histogram.get(value) : 0);
        histogram.put(value, count);
    }
    return histogram;
}
```

B

```
public static Map<Integer, Integer> generateHistogram3(int[] data) {
    Map<Integer, Integer> histogram = new HashMap<Integer, Integer>();
    for (int i = 0; i < data.length; i++) {
        int value = data[i];
        int count = 1 +
            (histogram.containsKey(value) ? histogram.get(value) : 0);
        histogram.put(value, count);
    }
    return histogram;
}
```

C Control flow is equivalent for both

D Control flow is problematic for both

Which is better?

A

```
public int[] copyIntArray(int[] input) {
    int [] copy = new int[input.length];
    int i = 0;
    for (int value: input) {
        copy[i++] = value;
    }
    return copy;
}
```

B

```
public int[] copyIntArray(int[] input) {
    int [] copy = new int[input.length];
    for (int i = 0; i < input.length; i++) {
        copy[i] = input[i];
    }
    return copy;
}
```

C Control flow is fine for both

D Control flow is problematic for both

Which is better?

This code takes an array of strings, it processes all of the strings before a dash one way and all of the remaining strings another way. Assume there is only one dash in the array of strings.

A

```
boolean dashFound = false;
for (String arg : args) {
    if (arg.equals("-")) {
        dashFound = true;
    } else if (!dashFound) {
        process1(arg);
    } else {
        process2(arg);
    }
}
```

B

```
int i = 0;
while(i < args.length && !args[i].equals("-")) {
    process1(args[i]);
    i++;
}

i++; // skip the dash
for( ; i < args.length ; i++) {
    process2(args[i]);
}
```

C Control flow is fine for both

D Control flow is problematic for both

Returns

- **use early returns to reduce nesting, eliminate cases**
  - guard clauses
- **minimize the number of returns in a routine**
  - all things being equal

Refactoring Examples

Auto Formatting

- **IntelliJ: Code->Reformat Code**

Why StringBuilder?