CIS 110: Introduction to Computer Programming

Lecture 14

Booleans and Program Assertions

(§ 5.3-5.5)

Outline

- The boolean primitive type
- Program assertions and invariants

Booleans

All hail the mighty boolean

- Booleans are primitives that have two possible values: `true` or `false`

```
boolean bTrue = true;
boolean bFalse = false;
```

- Whenever we needed a guard or test, we really needed a *boolean value*.

```
if (/* boolean */) { }
for (int i = 0; /* boolean! */; i++) { }
while (/* boolean! */) { }
```

Relational operators revisited

- Relational operators compare primitive values and return booleans!
  - Booleans are themselves primitive, so we can compare them with != and ==.

```
boolean b1 = 5 > 0;          // true
boolean b2 = 12.7 == 13.5;   // false
boolean b3 = 'c' != 'd';     // true
boolean b4 = b1 == b2;       // false
```

Logical operators revisited

- Logical operators take booleans as arguments and produce a boolean value.
  - && (logical AND): true iff both args are true
  - || (logical OR): true iff at least one arg is true
  - ! (logical NOT): opposite of the argument

```
boolean b1 = (5 > 0) && ('c' == 'd');  // false
boolean b2 = b1 || (5 <= 35);          // true
boolean b3 = !(3 < 0);                 // true
// boolean b4 = x == 1 || 2 || 3       // bad code!
```

Truth Tables

- Since booleans only have two possible values, we can summarize the results of logical operators using *truth tables*.

a && b	true	false
true	true	false
false	false	false

a || b	true	false
true	true	true
false	true	false

a	!a
true	false
false	true

Java operator precedence review

```
|  !  ++  --  +  -
|  *  /  %
|  +  -
|  <  >  <=  >=
| ==  !=
| &&
| ||
| =  +=  -=  *=  /=  %=  &&=  ||=
|
V (In order of decreasing precedence)
```

- e.g., these two are different:
  - `b1 && b2 || b3 && b4`
  - `b1 && (b2 || b3) && b4`

Short-circuiting && and ||

- && and || will not evaluate their second argument if it is unnecessary to do so
  - i.e., if the first argument is false (&&) or true (||).
  - Necessary behavior to write some guards cleanly!

```
Scanner in = new Scanner(System.in);
String line = in.nextLine();
int pos = 0;
while (pos < line.length() && line.charAt(pos) != 'X') {
  pos++;
}
String toX = line.substring(0, pos);
System.out.println(toX);
```

Without short-circuiting would cause an exception when pos == line.length()!

DeMorgan's Laws

- Identities concerning logical ops and negation.

```
   !(b1 || b2)
= !b1 && !b2
```

```
   !(b1 && b2)
= !b1 || !b2
```

- Useful for simplifying and reasoning about boolean expressions.

```
   while (!(s.equals("yes") || s.equals("no"))) { ... }
=  while (!s.equals("yes") && !s.equals("no")) { ... }
```

Boolean flags

- One use of boolean variables is a boolean *flag*.

```
Scanner in = new Scanner(System.in);
boolean seenPie = false;
for (int i = 1; i <= 5; i++) {
  System.out.print("Enter word " + i + ": ");
  String line = in.nextLine();
  if (line.equalsIgnoreCase("pie")) {
    seenPie = true;
  }
}
if (seenPie == true) {
  System.out.println("Haha.  You said pie.");
} else {
  System.out.println("Good words!");
}
```

This will be true if we ever see "Pie".

Boolean Zen

- *Boolean zen* is realizing the simplicity of the boolean expressions.

```
if (seenPie == true) {
  // ...
}
```

```
if (seenPie) {
  // ...
}
```

Much simpler!

If `seenPie` and `seenPie == true` have the same values, why don't I replace...?

seenPie	seenPie == true
true	true
false	false

Program Assertions

Imperative programming

- *Imperative programming*: being able to *mutate* (change) state/variables.
- Mutation makes program reasoning hard!
  - Need to keep track of both *control flow* and *state*.

```
int x = 0;
int y = 5;
int z = 25;
while (x + y < z) {
  System.out.printf("x = %d, y = %d, z = %d\n", x, y, z);
  x += y / 2;
  y += (int) (x * 1.5);
  z += x * 2;
}
System.out.printf("Final: x = %d, y = %d, z = %d\n", x, y, z);
```

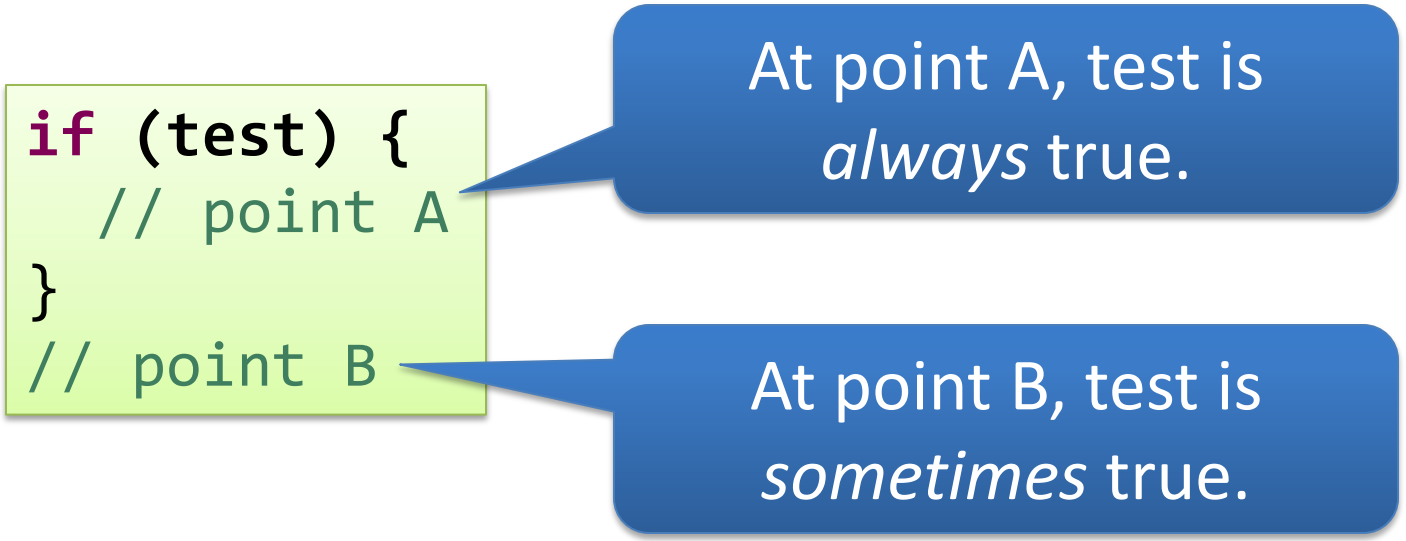
Assertions

- An assertion is a claim that is *true*, *false*, or *sometimes true and sometimes false*.
  - e.g., "2 + 2 = 4", "Cats bark", "The sky is blue"
- *Programming assertions* are such claims made about the state of a program.

```
while (i > 0) {
  // ...
}
// Is i > 0 always/never/sometimes true here?
```

Assertions implicitly made by programming constructs

- By design, certain language constructs enforce some assertions, e.g., if-statements.



Assertions and if-else

```
if (test) {
  // point A
} else {
  // point B
}
  // point C
```

test is *always* true.

test is *always* false.

test is *sometimes* true.

More assertions and if-else

```
if (test1) {
  // point A
} else if (test2) {
  // point B
} else {
  // point C
}
// point D
```

test1 is *always* true;
test2 is *sometimes true*.

test1 is *always* false;
test2 is *always* true.

test1 is *always* false;
test2 is *always* false.

test1 is *sometimes true*;
test2 is *sometimes true*.

Assertions and while loops

```
while(test) {
  // point A
}
// point B
```

test is *always* true

test is *always* false

Assertions and for loops

```
for (int i = 0; test; i++) {
  // Point A
}
  // Point B
```

test is *always* true

test is *always* false

Exercise that mental model

- In general, we must rely on our mental model of computation to reason about assertions.

```
public static String repeat(String msg, int n) {
  String ret = "";
  // Is ret.length() == 0 always/sometimes/never true?
  for (int i = 0; i < n; i++) {
    ret += msg;
  }
  // Is ret.length() == msg.length() * n
  // always/sometimes/never true?
  return ret;
}
```

Always true!

Always true!

An extended example: mystery

```
public static int mystery(int n) {
  int x = 0;
  // Point A
  if (n < 0) { return -1; }
  while (n != 0) {
    // Point B
    int d = n % 10;
    if (d % 2 == 1) {
      x += d;
    }
    // Point C
    n /= 10;
  }
  // Point D
  return x;
}
```

For each point, are the following always/sometimes/never true?

1) n < 0
2) x >= 0
3) d < 10
4) x < n

(See AssertionProblem.java.)