# 6 BOOLEANS

Terry Marris 12 April 2001

## 6.1 OBJECTIVES

By the end of this lesson the student should be able to

- use boolean variables, objects and methods
- use the relational operators in boolean expressions
- apply precedence rules
- identify boundary points
- perform boundary tests

## 6.2 PRE-REQUSITES

The student should be familiar with rules of precedence and associativity as they apply to arithmetic operations, div and mod operations on int variables and class diagrams.

## 6.3 PREVIEW

We have already met the *int* and *double* primitive (or non-object) data types. We now look at the non-number primitive type *boolean.*

We look at the boolean data type, the relational operators and the principles of boundary testing.

## 6.4 BOOLEANS

There are just two boolean values: *true* and *false*. A variable of type boolean is declared and initialised in a way that variables of type int and double are.

```
boolean isAStudent = false;
```

*boolean* is a Java keyword.

When choosing names for boolean variables (and methods and objects ) we consider including the word *is* or the word *has*, *isRich*, *isOk* and *hasNext* for example. Why? If the word *is* or *has* appears in an identifier then we know straight away that its is *boolean*.

## 6.5 A LIGHT CLASS

We model an electric light.

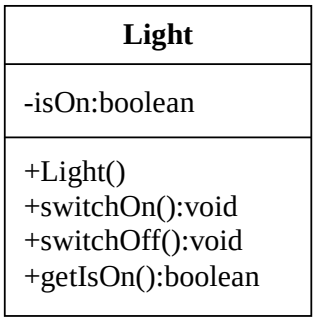



**Figure 6.1** *The Light class*

A *Light* object has just one private boolean attribute, *isOn*, which models the state of the light. If *isOn* is true, the light is on. If *isOn* is false, the light is off.

The public constructor, *Light()*, initialises *isOn* with the value *false*.

The public methods *switchOn()* and *switchOff()* set *isOn* to *true* and *false* respectively.

The public method *getIsOn()* returns the current state of the *isOn* attribute.

## 6.6 THE LIGHT CLASS IMPLEMENTATION

We show below the Java implementation of the *Light* class, a *main()* method to test it and the result of the program run.

A *Light* has one private *boolean* field, *isOn.*

```
public class Light {
  private boolean isOn;
```

The constructor initialises the *isOn* field with the value *false*.

```
public Light()
{
  isOn = false;
}
```

The public method *switchOn()* sets the private *boolean* field *isOn* to *true*.

```
public void switchOn()
{
  isOn = true;
}
```

The public method *switchOff()* sets the private *boolean* field *isOn* to *false*.

```
public void switchOff()
{
  isOn = false;
}
```

The public method *getIsOn()* retrieves the value stored in the private field, *isOn*.

```
public boolean getIsOn()
{
  return isOn;
}
```

The methods are tested in the *main()* method.

```
public static void main(String[] s)
{
```

We use the constructor create a new light that is initially off.

```
    Light aLight = new Light();
```

To show that it is off we write

```
    System.out.println("A new light is on: " +
                       aLight.getIsOn());
```

and expect the result

```
     A new light is on: false
```

to be displayed.

We send the message *switchOn()* to the *aLight* object.

```
    aLight.switchOn();
```

To show that is now on we write

```
  System.out.println("After switching on, the light is on: " +
                     aLight.getIsOn());
```

and expect the result

```
     After switching on, the light is on: true
```

to be displayed.

We send the message *switchOff()* to the *aLight* object.

```
aLight.switchOff();
```

To show that the light is now off we write

```
System.out.println("After switching off, the light is on: " +
                   aLight.getIsOn());
```

and expect to see the result

```
After switching off, the light is on: false
```

on the screen.

We show the complete program on the next page.

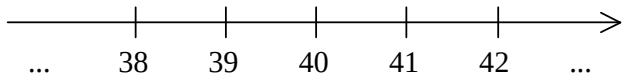
```
/* Light.java
   Terry Marris  12 April 2001
*/

public class Light {
  private boolean isOn;

  public Light()
  {
    isOn = false;
  }


  public void switchOn()
  {
    isOn = true;
  }


  public void switchOff()
  {
    isOn = false;
  }


  public boolean getIsOn()
  {
    return isOn;
  }


  public static void main(String[] s)
  {
    Light aLight = new Light();
    System.out.println("A new light is on: " +
                       aLight.getIsOn());
    aLight.switchOn();
    System.out.println("After switching on, the light is on: "
                       + aLight.getIsOn());
    aLight.switchOff();
    System.out.println("After switching off, the light is on:"
                       + aLight.getIsOn());
   }
}
```

Program run:

```
A new light is on: false
After switching on, the light is on: true
After switching off, the light is on: false
```

## 6.7 THE RELATIONAL OPERATORS

Look at the number line shown below.




39 comes before 40. We write 39 < 40 where < means *is less than.*

41 comes after 40. We write 41 > 40 where > means *is more than.*

The symbols are easy to remember. **L**ess than has its point on the **L**eft. Mo**R**e than has its point on the **R**ight.

40 is the same as 40. We write 40 == 40 where == means *is the same as.*

Symbols such as <, > and == are known as relational operators. The relational operators are shown in the following table.

| Symbol | Meaning | Example |
|---|---|---|
| < | is less than | 39 < 40 is true |
| > | is more than | 41 > 40 is true |
| == | is the same as (is equal to) | 40 == 40 is true |
| <= | is less than or equal to | 39 <= 40 is true, 40 <= 40 is true |
| >= | is more than or equal to | 41 >= 40 is true, 40 >= 40 is true |
| != | is not the same as | 39 != 40 is true |

**Figure 6.2** *The Relational Operators*

Notice that the symbol for *is less than or equal* to is written with the < first, followed by the = with no spaces in between them. Order and space (or the lack of space) is important.

Notice the difference between = and == . = means *assign to* or *copy from right to left.* == means *is the same as* or *is equal to.*

The relational operators fit into the rules of precedence like this.

| Operator | | | | Precedence | Associativity |
|---|---|---|---|---|---|
| ( ) | | | | highest | left to right |
| ! | ++ | -- | (cast) | | right to left |
| * | / | % | | | left to right |
| + | - | | | | left to right |
| < | <= | > | >= | | left to right |
| == | != | | | | left to right |
| = | | | | lowest | right to left |

**Figure 6.3** *The Precedence Table*

The relational operators are between the assignment operator, =, and the addition operators, + and -.

## 6.8 BOOLEAN EXPRESSIONS

I hope you will agree that 39 < 40 is true but 40 < 39 is not true; it is false. 39 < 40 is an example of a boolean expression.

A *boolean* expression has just one of two possible values: *true* and *false*.

Look at these declarations.

```
int n = 7;
boolean isEven = n % 2 == 0;
```

*n* is an *int* variable with an initial value of 7.

*isEven* is a *boolean* variable i.e. it can be either *true* or *false*.

*n % 2* is 1 (the remainder after division).

*n % 2 == 0* is a boolean expression. *n % 2 == 0* is false. And so *false* is assigned to (or copied into) *isEven.*

Look at the rules of precedence shown in Figure 8.3 above. The mod operator % has a higher precedence than both = and == and so is done first. The = has the lowest precedence and so is done last.

If it helps, put brackets in the expression to reflect the usual rules of precedence. For example

```
int n = 7;
boolean isEven = ((n % 2) == 0));
```

## 6.9 THE THERMOSTAT CLASS

A thermostat measures a given value against pre-set lower and upper limits. If the given value is outside these limits then the thermostat informs its controller; the controller then decides what action, if any, is to be taken.

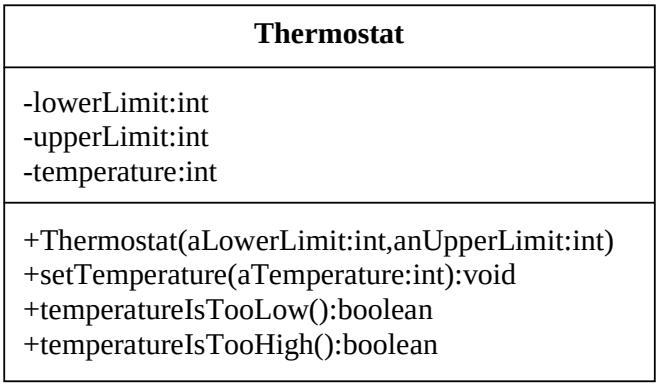



**Figure 6.4** *The Thermostat Class*

Our *Thermostat* class has three *private* attributes: *lowerLimit* and *upperLimit* whose values are set by the constructor, and *temperature* whose value is set by the *setTemperature()* method.

The *boolean* method *temperatureIsTooLow()* returns *true* if the *temperature* is less than the *lowerLimit* and *false* if the temperature is not less than the *lowerLimit*. In Java we write

```
public boolean temperatureIsTooLow()
{
  return temperature < lowerLimit;
}
```

The *boolean* method *temperatureIsTooHigh()* returns *true* if the temperature is more than the *upperLimit* and *false* otherwise. In Java we write

```
public boolean temperatureIsTooHigh()
{
  return temperature > upperLimit;
}
```

The complete implementation of the *Thermostat* class is shown on the next page.

```
/* Thermostat.java
   Terry Marris  15 April 2001
*/

public class Thermostat {
  private int lowerLimit;
  private int upperLimit;
  private int temperature;

  public Thermostat(int lower, int upper)
  {
    lowerLimit = lower;
    upperLimit = upper;
    temperature = 0;
  }


  public void setTemperature(int t)
  {
    temperature = t;
  }


  public int getTemperature()
  {
    return temperature;
  }


  public int getLowerLimit()
  {
    return lowerLimit;
  }


  public int getUpperLimit()
  {
    return upperLimit;
  }


  public boolean temperatureIsTooLow()
  {
    return temperature < lowerLimit;
  }


  public boolean temperatureIsTooHigh()
  {
    return temperature > upperLimit;
  }
```

```
public static void main(String[] s) {
    Thermostat aThermostat = new Thermostat(15, 25);
    System.out.println("Lower limit: " +
                       aThermostat.getLowerLimit());
    System.out.println("Upper limit: " +
                       aThermostat.getUpperLimit());
    System.out.println();

    aThermostat.setTemperature(14);
    System.out.println("Temperature: " +
                       aThermostat.getTemperature());
    System.out.println("Too cold: " +
                       aThermostat.temperatureIsTooLow());
    System.out.println();

    aThermostat.setTemperature(15);
    System.out.println("Temperature: " +
                       aThermostat.getTemperature());
    System.out.println("Too cold: " +
                       aThermostat.temperatureIsTooLow());
    System.out.println();

    aThermostat.setTemperature(16);
    System.out.println("Temperature: " +
                       aThermostat.getTemperature());
    System.out.println("Too cold: " +
                       aThermostat.temperatureIsTooLow());
    System.out.println();

    aThermostat.setTemperature(24);
    System.out.println("Temperature: " +
                       aThermostat.getTemperature());
    System.out.println("Too hot: " +
                       aThermostat.temperatureIsTooHigh());
    System.out.println();

    aThermostat.setTemperature(25);
    System.out.println("Temperature: " +
                       aThermostat.getTemperature());
    System.out.println("Too hot: " +
                       aThermostat.temperatureIsTooHigh());
    System.out.println();

    aThermostat.setTemperature(26);
    System.out.println("Temperature: " +
                       aThermostat.getTemperature());
    System.out.println("Too hot: " +
                       aThermostat.temperatureIsTooHigh());
  }
}
```

Program run:

```
Lower limit: 15
Upper limit: 25

Temperature: 14
Too cold: true

Temperature: 15
Too cold: false

Temperature: 16
Too cold: false

Temperature: 24
Too hot: false

Temperature: 25
Too hot: false

Temperature: 26
Too hot: true
```

## 6.10 BOUNDARY TESTING

Notice how the *boolean* methods of the *Thermostat* class were tested.

We (arbitrarily) set the lower and upper limits (i.e. boundaries) to 15 and 25 respectively.

Then for the lower limit of 15 we deliberately set the temperature three times with values 14 (just under) 15 (exactly on) and 16 (just over) the lower limit.

And for the upper limit of 25 we set the temperature three times with values 24 (just under) 25 (exactly on) and 26 (just over) the upper limit.

Boundaries are found wherever we have boolean expressions. In the boolean expression

```
temperature < lowerLimit
```

the boundary is *temperature == lowerLimit*.

For testing we fixed the *lowerLimit* and then choose values of *temperature*

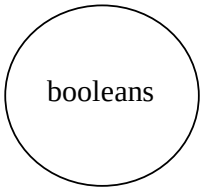
**(a)** just below
**(b)** exactly on
**(c)** just above

this lower limit.

And the same for the upper limit.

The purpose of testing is to discover errors. (If you were employed as a program tester your job would be to find errors in programs.) Errors are commonly found around boundary points. So in testing we always pay attention to these points.

## 6.11 REVIEW




values are true and false

e.g. isFinished, hasFlipped

relational operators

< less than e.g. 39 < 40 is true

> more than e.g. 41 > 40 is true

testing

purpose to detect errors

boundaries

occur with boolean expressions e.g. a < b

boundary testing

just under e.g. a = 2, b = 3

exactly equal e.g. a = 2, b = 2

just over e.g. a = 3, b = 2

## 6.12 EXERCISES

**1** Explain what is meant by the Java primitive type *boolean*.

**2** State the value stored in each variable for each of the Java code fragments shown below.

**(a)**
```
int m = 5;
int n = 5 / 2;
boolean b = n == 2;
```

**(b)**
```
int balance = 100;
int transaction = 99;
boolean isOk = balance - transaction >= 0;
```

**(c)**
```
int balance = 100;
int transaction = 101;
boolean isOk = balance - transaction >= 0;
```

**(d)**
```
int r = 3;
int A = 27;
boolean isACatastrophe = 2 * A > 3 * r * r;
```

**(e)**
```
double e = 3.0;
double m = 36.0;
double g = 9.0;
double h = (g + m / g) / 2.0;
boolean hasFinished1 = g - h < e;
boolean hasFinished2 = g + h < e;
```

**3** Look at the following Java code fragment then complete the table that follows.

```
int carbonMonoxideLevel;
boolean isDangerous = carbonMonoxideLevel > 80;
boolean isNormal = carbonMonoxideLevel < 40;
```

| # | **Test Data** | **Expected Result** | |
|---|---|---|---|
| | carbonMonoxideLevel | isDangerous | isNormal |
| 1 | 39 | | |
| 2 | 40 | | |
| 3 | 41 | | |
| 4 | 79 | | |
| 5 | 80 | | |
| 6 | 81 | | |

**4** Look at the Java code fragment shown below. By choosing appropriate values for *target* construct a test table with headings *test data* and *expected result* (just like the table in question 3 above).

```
int target;
int first = 1;
int last = 100;
int middle = (first + last) / 2;
boolean isFound = middle == target;
boolean goLeft = target < middle;
boolean goRight = target > middle;
```