If Statements and Booleans

For a program to do anything interesting, it needs if-statements and booleans to control which bits of code to execute. Here is a simple if-statement...

```
if (temperature > 100) {
  System.out.println("Dang, it's hot!");
}
```

The simplest if-statement has two parts – a boolean "test" within parentheses ( ) followed by "body" block of statements within curly braces { }. The test can be any expression that evaluates to a boolean value – true or false – value (boolean expressions are detailed below). The if-statement evaluates the test and then runs the body code only if the test is true. If the test is false, the body is skipped.

Another common form of if-statement adds an "else" clause such as with the code below which prints one message or the other...

```
if (temperature > 100) {
  System.out.println("Too darn hot");
}
else {
  System.out.println("At least it's not more than 100");
}
```

The if/else form is handy for either-or logic, where we want to choose one of two possible actions.

The if/else is like a fork in the road. Under the control of the boolean test, one or the other will be taken, but not both. For example, the famous Robert Frost poem is a thinly disguised comment on the importance of the if/else structure...

```
Two roads diverged in a wood, and I -
I took the one less traveled by,
And that has made all the difference.
```

Comparison Operators: <, <=, >, >=

The easiest way to get a boolean value (true or false) is using a comparison expression, such as (a < 10). The less-than operator, <, takes two values and evaluates to true if the first is less than the second. So for example, the expression `(var < 10)` evaluates to the value true if var is less than 10, and false otherwise. The < is an "operator" just like + and * – appearing between two values to compute something. Instead of "var" and "10", the comparison can use any int or double expressions, so for example we could write a comparison expression like `(score > (highScore+100))`. This comparison will

evaluate to true if the `score` value is greater than the `(highScore+100)` value, and will evaluate to false otherwise.

There are four less-than type operators...

<	less-than
>	greater-than
<=	less-or-equal (i.e. ≤)
>=	greater-or-equal (i.e. ≥)

There is overlap between these, since we could use less-than to write something like `(a<b)`, but that would work just the same as writing it with greater-than: `(b>a)`. It makes no difference to the computer. We will prefer the version which reads most naturally.

No: 10<a<20

It's tempting to write an expression like "10<a<20" to see if a is between 10 and 20. That does not work. Each "<" operator must get its own two values. So the correct way to write it is:(10<a && a<20).

Equality Operators ==, !=

The == operator tests if two values are exactly the same, so `(a == b)` is true if a and b have exactly the same value. The not-equal operator, `!=`, is the opposite, evaluating to true if the values are different. We use == and `!=` only with primitives such as int, never with objects like String and Color. With objects, we use the message equals() to test equality.

The similarity of == and equals() can be confusing, so we use a simple rule: Every value in Java is either a primitive or an object. We use ==, <, >=, etc.. with primitives like int and double. We use equals() with objects like String and Color. Since the dereference dot (.) only works with objects, you can remember it this way: if it can take a dot then use equals() (e.g. a String), otherwise use == (e.g. an int).

Doubles vs. ==

The one addition to this rule is that we never use == or `!=` with doubles, since the intrinsic little error term on every double value throws off the operation of ==. For example, summing up 1/10, 100 times may not exactly `== 10.0`. To compare two doubles, subtract one from the other, and then see if the difference is very small...

```
// suppose we have doubles d1 and d2
// We use the Math.abs() standard method that computes
// absolute value

// check if d1 and d2 are essentially equal
// (if the absolute value of their difference is less
//  than 1e-6 (i.e. 0.000001)
if (Math.abs(d1 - d2) <= 1e-6) {...
```

Boolean Type

The simplest and most common form of boolean expression is the use a < in an if-statement as shown above. However, **boolean** is a full primitive type in Java, just like int and double. In the boolean type, there are only two possible values: **true** and **false**. We can have variables and expressions of type boolean, just has we have variables and expressions of type int and double. A boolean variable is only capable of storing either the value true or the value false. The words true and false are built-in literals in Java that can be used right in the code. As with other types, Java checks the code to make sure that the right type of value goes into each variable...

```
int i = 6;           // ok
boolean a = false;   // ok
boolean b = "hello"; // NO, String and boolean are different
boolean c = "false"; // NO, "false" in quotes is a String!
String s = "false";  // ok
```

Boolean Operators

Just as we have + and * operators that work on int values, we have operators that work on boolean values. Suppose we have boolean expressions b1 and b2, which may be simple boolean variables, or may be boolean expressions such as (score < 100). The "and" operator && takes two boolean values and evaluates to true if both are true. The "or" operator || (two vertical bars) takes two boolean values and evaluates to true if one or the other or both are true.

Suppose we have int variables score and temperature, the code below prints "It's hot out and so am I" if the score is 10 or more and temperature is 100 or more. The && is true only if both of the two booleans it connects are true...

```
if ( (score >= 10) && (temperature >= 100) ) {
  System.out.println("It's hot out, and so am I!");
}
```

Suppose we are in a bad mood if our score is less than 5 or the temperature is 32 or below...

```
if ( (score < 5) || (temperature <= 32) ) {
  System.out.println("I'm in a bad mood");
}
```

Finally, the "not" operator ! (an exclamation mark) goes to the left of a boolean expression and inverts it, changing true to false and false to true.

Suppose we want to print something if it is not the case that the score is less than 5, we could write that as...

```
if (!(score < 5)) {
   System.out.println("Score is 5 or more)");
```

```
}
```

First, the expression (score < 5) evaluates to true or false, and then the ! inverts the boolean value. The result is that the body runs if the expression (score < 5) is false, which is to say it runs if (score >= 5) is true. (We could equivalently write the if-statement with (score >= 5), but the point was to demonstrate the !)

Restaurant Example

Suppose we want to get a table at a hip restaurant, but we are afraid that the Maitre'd is going to say something rude instead of seating us. The variable "style" represents the stylishness of our outfits and the variable "bribe" represents the bribe presented to the Maitre'd. The Maitre'd is satisfied if **style is 8 or more and bribe is 5 or more**. The Matire'd says "Je n'think so" if they are **not satisfied**. One way to code this up is to write the "satisfied" expression as a straight translation of the problem statement, and then put a ! in front of it...

```
// Say something mean if not satisfied
if (! ((style>=8) && (bribe>=5)) ) {
  System.out.println("Je n'think so");
}
```

With combinations of !, <, >=, etc. there can be a few different ways to code what amounts to the same thing. As a matter of style, we prefer the code that reads most naturally to express the goal of the code.

For example, the following is equivalent to the above, although I find it near impossible to decipher...

```
// Say something mean if not satisfied
if (!(!(style<8) && !(bribe<5))) {
```

Here is another equivalent form, which is not so bad...

```
// Say something mean if not satisfied
if ((style<8) || (bribe<5)) {
```

Java also has "bitwise" operators & and | (which we are not using) which are different from && and ||. If your boolean code will not compile, make sure you did not accidentally type a bitwise operator (&) instead of a boolean operator (&&).

Boolean Precedence

The above examples used parentheses to spell out the order of operations. The boolean not ! has a high precedence – in (!a && b) the ! is evaluated on the a, before the &&. We could add in parenthesis (!(a && b)) to force the && to evaluate before the !. The && operator is higher precedence than ||, so in (a || b && c), the && happens first. There are

parallels between the boolean operators and arithmetic: ! is like unary -, && is like *, || is like +.

The comparison operators (<, <=, ==, ...) all have higher precedence than && and ||, so without parentheses, comparisons always happen before the && and ||. This is very convenient, since we can use <= to get some booleans, and then && and || to combine the booleans. So our earlier example of...

```
if (! ((style>=8) && (bribe>=5))) {
```

works fine without parentheses around the comparisons...

```
if (! (style>=8 && bribe>=5)) {
```

The not ! has a high precedence (as do all the unary operators), so we need parentheses to force the ! to evaluate after the >= and &&.

Remember that all the unary operators, such as !, have a high precedence. To evaluate something before the !, we frequently need to put the something in parentheses like this: !(*something*). For example...

```
if (! i < 10 ) { ... // NO does not work
```

The precedence of the ! is too high, it wants to execute before the <. To fix this, we add parentheses around the (i < 10) so it goes first

```
if (!(i < 10 )) { ...   // Ok
```

Opposite of < ?

What is the opposite of the expression (score < 100)? That is, an expression that will be true when the first is false, and false when the first is true. The opposite expression is (score >= 100). You might think that the opposite of < is >, but it is not. The opposite of < is >=, and the opposite of > is <=.

Boolean Short Circuiting

The boolean expressions stop evaluating as soon as the result is clear. This is called "short circuiting". The `&&` short circuits at the first false – then the whole expression must be false. The `||` short circuits at the first true – then the whole expression must be true. Suppose we have an int i, and we want to check if i is 2 or 3...

```
if (i==2 || i==3) { ...
```

If i is 2, the boolean expression evaluates to true after the i==2 is true. It does not even look at the i==3.

Suppose we want to do something if 100.0/i is less than 10. We use short-circuiting to screen out the i==0 case, checking it before the division. If i is 0, the expression short-circuits out before the division

```
if (i!=0 && (100.0/i < 10.0)) { ...
```

A common use of short-circuiting is to call a method on an object, but only if the pointer to the object is not null. For example, suppose we have a bear pointer which may or may not be null...

```
if (bear!=null && bear.hasCubs() && bear.isMom()) { ...// run!
```

The Truth About Curly Braces { }

If the body of a while-loop or if-statement is made of many statements , then we use curly braces { } to group the statements in the body, as we have seen in many examples.

However, if there is only a single statement in such a body, then the curly braces are optional.

For example, suppose if score is greater than 100 then we want to set it to 100 and print something. This can be written with curly braces as we usually do...

```
if (score > 100) {
  score = 100;
  System.out.println("Score is 100");
}
```

However, if there is only one statement in the body (say just "score = 100;") then it can be written without the curly braces.

```
// One-statement body written without curly braces
if (score > 100)
  score = 100;
```

Or, in fact it can be written all on one line...

```
// One-statement body written without curly braces
if (score > 100) score = 100;
```

Digression – Why We Use Curly Braces

The no-brace style saves a little typing, and you can do it if you like. However, in our example code we will never do it. The problem is that it can lead to bugs if you later come to add a statement to the body. Suppose we want to add a println() statement to the above code, it's natural to just type it on the next line...

```
if (score > 100)
  score = 100;
  System.out.println("Score is 100");

// BUG, this code compiles fine, but it gives the appearance
// that the println() is controlled by the if, which it is not.
```

Unfortunately, without the curly braces, the code can have a deeply misleading appearance. Above, it appears (due to the misleading indentation) that the println() is under the control of the if-statement, but it is not. The "score = 100;" is controlled by the if-statement, and the println() is just the next statement that is outside of the control of the if-statement. An arrangement of space that more honestly shows how the code actually works is...

```
if (score > 100)
  score = 100;

System.out.println("Score is 100");
```

To avoid this situation where the code says one thing, but the indentation and spacing suggests something else, we prefer to write out our if-statements and loops with the curly braces for maximum clarity.

Readability

When writing comparisons, there are often different ways to say the same thing. For example, if we want to check if an int score is 100 or more, any of the following will do the same thing...

```
if (score >= 100) { ...    // version 1

if (100 <= score) { ...    // version 2

if (score > 99) { ...      // version 3
```

Though the three versions are effectively the same, as a matter of style we prefer the code that reads as the most natural translation of the goal of the code. In this case, the first is probably the best translation of the idea "score is 100 or more".

Nested if-statements

We can write an if-statement inside the body or else part of another, outer if-statement. The outer if-statement executes first, then proceeds to its body or else code normally. In this case, the code happens to be another if-statement.

Suppose we are writing the code for an alarm clock, and the int variable "day" represents the current day with Mon=1, Tue=2, ... Sun=7. On weekdays the clock tells us to get up, and weekends it tells us to snooze more. On Monday, the clock adds some extra

harassment to get us out of bed. The code uses an if-statement to distinguish weekends from weekdays, and inside the "weekday" case there is another if-statement for Mondays..

```
if (day <= 5) {   // 1..5 are weekdays
  System.out.println("Hi ho, hi ho, it's off to school we go...");
  if (day = 1) {  // Something extra for Mon
    System.out.println("It's not the weekend any more!");
  }
} else {          // 6..7, it's the weekend
  System.out.println("Hit the snooze button now!");
}
```

On Mon, it prints...

```
Hi ho, hi ho, it's off to school we go...
It's not the weekend any more!
```

On Tue, Wed, Thu, and Fri, it prints...

```
Hi ho, hi ho, it's off to school we go...
```

On Sat and Sun, it prints...,

```
Hit the snooze button now!
```

if-else Chain

If we have exactly two choices, we can use the basic if-else structure, but what if there are more than two? We can run a series of if-else together to make a chain, as shown below. Suppose we have a bowling score (300 is perfect, 200 is very good)...

```
if (score >= 300) {
  System.out.println("Perfect!");
}
else if (score >= 200) {
  System.out.println("Very good");
}
else if (score >= 50) {
  System.out.println("Good job");
}
else {
  System.out.println("Everyone is good at something.");
  System.out.println("You are good at something else!");
}
```

The if-else chain checks each test, working from top to bottom. The first test that is true runs the corresponding body, and then exits the whole chain. So for the above code, if

the score is 250, the program prints "Very good" and exits the chain. The result of the if-else chain is to choose one piece of code from among many. The optional final "else" in the chain does not have an "if" and plays a catch-all role in case all the others are false.

What does the above code print for a score of 250?

```
Very good
```

What does the above code print for a score of 150?

```
Good job
```

What does the above code print for a score of 20, or 10, or -123...

```
Everyone is good at something.
You are good at something else!
```

For the design of the bowling code above, notice that the tests must be done in the order shown, working from high to low. Each test relies on the fact that the tests above have been checked already, and were all false. For example, if the chain gets to the (score >= 50) line, we know that (score >= 200) must have been false, and so the score is 199 or less.

Boolean Methods

One useful sort of method for a class is one that tests if some condition is true or not about the receiver. One convention is that such methods begin with the word "is" or "has". For example, a Car object might implement an isOutOfGas() method which returns true if the car is out of gas or false otherwise. In general, the client can use boolean messages to check for various true/false conditions of the receiver. For example a Bear class might have an isMother() method which is true if the bear is a mother, and a hasCubs() method which is true if the bear has cubs with it. Sometimes the mommy bear has the cubs, and sometimes the daddy bear has the cubs...

```
public class Bear {
  ...
  // Returns true if the bear is a mother bear
  public boolean isMother() { ...

  // Returns true if the bear has cubs with it
  public boolean hasCubs() { ...
}
```

Now suppose we are writing the meetBear() code for a Backpacker, used when the Backpacker meets a bear in the woods...

```
// Code for when we meet a bear in the woods
void meetBear(Bear bear) {
  System.out.println("Oh look, a bear.");

  if (bear.isMother()) {
    System.out.println("Hi mom!");
  }

  if (bear.hasCubs()) {
    System.out.println("Hi cute little bears!");
  }


  if (bear.isMother() && bear.hasCubs()) {
    System.out.println("Oh @#$@#$!, Run!");
  }

}
```

Another way to write the mother if-statement uses "== true" like this...

```
  if (bear.isMother() == true) {
    System.out.println("Hi mom!");
  }
```

That code will work fine, but it is not the best style. The "== true" adds nothing, since the if-statement itself already checks if the test is true. Therefore, it is better to write it the direct way...

```
  if (bear.isMother()) {
    System.out.println("Hi mom!");
  }
```

Writing a Boolean Method

Writing a method that returns a boolean is just like writing any method with a return value, but ultimately we are just returning either **true** or **false**.

For example, suppose we are writing code for a hybrid-electric car. The car class as instance variables that track the amount of gasoline and battery charge: myGasoline (in the range 0..100), and myBatteries (also 0..100). Suppose the Car class implements an isLow() message that returns true if the car is low on fuels. It returns true if the sum of the gasoline level and the battery level is less than 10. The isLow() message is used by the dashboard to decide to light the "low fuel" light. What does the code for isLow() look like? Here is one way to write it...

```
public class Car {
  private int myGasoline;
  private int myBattery;

  ...
  // Returns true if we are low on fuels
  public boolean isLow() {
    if ((myGasoline+myBattery) < 10) {
      return true ;
    }
    else {
      return false;
    }
  }
}
```

The above code will work fine. It has a boolean test checking for low fuels, and depending on that it runs either "return true;" or "return false;". But there is a better way.

We can cut out the if-statement middle-man. Notice that the boolean in the if-test (true or false) happens to be the same as the value we want to return. If the test value is true, we return true. If the test value is false, we return false. So just return the test value directly!

In the short version below, we compute the desired boolean value with the expression "((myGasoline+myBattery) < 10)", and just return that value (which at runtime will either be the value true or the value false, depending on the gasoline and battery levels).

```
  public boolean isLow() {
    // Compute (true or false) if we are low,
    // and return that boolean value directly.
    return ((myGasoline+myBattery) < 10);
  }
```

Writing the method the long way is not terrible style (i.e. we won't mark off for it!), but it's nice to be comfortable enough with boolean values to write it the short way.

Boolean Logic Examples

```
/*
 * Do we go on a second date with someone?
 * The given chemistry is in the range 0..100 and isSchool
 * is true if it is during the school year. The answer is yes if chemistry is
 * 60 or more, or 40 or more not in the school year.
 */
public void secondDate(int chemistry, boolean isSchool) {
   if (chemistry>=60 || (!isSchool && chemistry>=40)) {
      System.out.println("Sure!");
   }
   else {
      System.out.println("I don't want to spoil our friendship");
   }
}
```

```
/*
 * Returns true if the person gets into Stanford.
 * Given gpa = 0...4.0, isGates = child of Bill Gates,
 * isDarth = associated with Darth Vader.
 * To get in: not associated with Darth Vader at all
 * Gpa must be over 3.95
 * Or gpa over 1.0 if child of Bill Gates
 */
public boolean getIntoStanford(double gpa, boolean isGates, boolean isVader) {
   if (!isVader && (gpa>=3.95 || (gpa>=1.0 && isGates))) {
      return true;
   }
   else {
      return false;
   }
}


// Variant, where we return the boolean directly
public boolean getIntoStanford2(double gpa, boolean isGates, boolean isVader) {
   return (!isVader && (gpa>=3.95 || (gpa>=1.0 && isGates)));
}


// Variant, where we use an if-return for the darth case.
// I think maybe this one is the best -- the one big expression is
// a bit hard to follow, this version makes it more obvious.
public boolean getIntoStanford3(double gpa, boolean isGates, boolean isVader) {
   if (isVader) {    // no way, it'a  deal killer!
      return false;
   }

   // Otherwise just use gpa and isGates
   return (gpa>=3.95 || (gpa>=1.0 && isGates));
}


/*
 Suppose we have a scratcher game where the player
 scratches off areas a, b, and c, each is either 0, 1, or 2.
 -If they are all 2's we win 5.
 -Otherwise if they are all the same, we win 2.
 -Otherwise so long as both b and c are different from a we win 1.
 -Otherwise we win nothing.
*/
public void scratcher(int a, int b, int c) {
   // Use a local var to build up our answer
   int pay = 0;

   if (a==2 && b==2 && c==2) {
      pay = 5;
   }
   else if (a==b && b==c) {
      pay = 2;
   }
   else if (a!=b && a!=c) {
      pay = 1;
```

```
   }

   // could write as !(a==b || a==c)

   System.out.println("pay:" + pay);
}
```

Boolean Puzzles

```
// For all values of num and bool. what does this print?
public void test(int num, boolean bool) {
   if (num >= 90 || !bool) {
      System.out.println("A");
   }
   else {
      System.out.println("B");
   }
}

/*
            bool
          true  false
num >=90   A     A

num < 90   B     A

*/


// For all values of num and bool, what does this print?
// Note: deciphering this is hard, harder than most
// code-writing problems, but it's a good way to
// exercise your logic skills. I can only figure it
// out by making a little chart of the possible values.
public void test2(int num, boolean bool) {
   if (num >= 90 && bool) {
      System.out.println("Tic");
   }
   else if (num >= 20) {
      System.out.println("Tac");
   }
   else {
      System.out.println("Toe");
   }
}

/*
            bool
          true  false
num >= 90    Tic    Tac

20<=num<90   Tac    Tac

num < 20     Toe    Toe

*/
```