Lesson: while loops

VOID METHODS

"void" means doesn't return a value.

We call a void method for its side effects.

Calls to void methods are statements.

BOOLEAN METHODS

Must return a boolean value.

We call a boolean method for its return value.

A good boolean method should have no side effects.

Calls to boolean methods are expressions.

What's wrong with this method:

```
public static void checkIfFrontIsBlocked()
{
    ...
}
```

The word "check" signals that we probably need to write a boolean method:

```
public static boolean frontIsBlocked()
```

The carpetRooms problem:

```
 1 2 3 4 5 6 7 8
X.X...X.X...X.X.X
E................
```

8 possible "rooms". Want to "carpet" (darken) each room with 2 walls: X.X

```
 1 2 3 4 5 6 7 8
X:X...X:X...X:X:X
................E
```

(Rooms 1, 4, 7, and 8 have been carpeted.)

```
//before: below left edge of possible room, facing east.
//after:  below right edge of possible room, facing east.
//        if room, center square has been darkened.
public static void carpetRoom()
{
    Robot.move(); Robot.turnLeft();
    Robot.move(); Robot.turnLeft();
    if (Robot.frontIsClear())
    { Robot.turnLeft(); }
    else
    {
        turnAround();
        if (!Robot.frontIsClear()) { Robot.makeDark(); }
        turnRight();
    }
    Robot.move(); Robot.turnLeft(); Robot.move();
}
```

We were thinking in Java!
How would we have thought about it in English?

```
public static void carpetRoom()
{
    enterRoom();

    if ( surrounded by walls )
    {
        Robot.makeDark();
    }

    exitRoom();
}
```

How do we do this?

```
public static boolean leftIsBlocked()
{
    Robot.turnLeft();
    if (Robot.frontIsClear())
    {
        turnRight();
        return false;
    }
    else
    {
        turnRight();
        return true;
    }
}
```

```
public static boolean rightIsBlocked()
{
    turnRight();
    if (Robot.frontIsClear())
    {
        Robot.turnLeft();
        return false;
    }
    else
    {
        Robot.turnLeft();
        return true;
    }
}
```

```
//returns true if both left and right side are walls.
//no side effects.
public static boolean surroundedByWalls()
{
    if (leftIsBlocked())
    {
        return rightIsBlocked();
    }
    else
    {
        return false;
    }
}
```

```
public static void carpetRoom()
{
    enterRoom();

    if (surroundedByWalls())
    {
        Robot.makeDark();
    }

    exitRoom();
}
```

A masterpiece!

"Missing Return Statement"

Every pathway through your code must reach a return.

```
//returns true if both left and right side are walls.
//no side effects.
public static boolean surroundedByWalls()
{
    if (leftIsBlocked())
    {
        if (rightIsBlocked())
        {
            return true;
        }
    }
    else
    {
        return false;
    }
}
```

No return value if left is blocked and right is not blocked.

Boolean Operators

!	is Java for	not	opposite
&&	is Java for	and	both true
||	is Java for	or	either true

Truth Tables

x	!x
true	false
false	true

x	y	x && y	x || y
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

```
//returns true if both left and right side are walls.
//no side effects.
public static boolean surroundedByWalls()
{
    return leftIsBlocked() && rightIsBlocked();
}
```

So elegant!

Suppose Java is evaluating the following expression

```
!(false || !true)
```

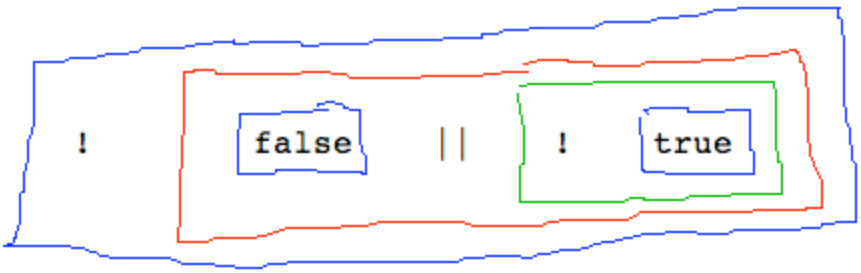
What part do you think it evaluates first? Then what?

```
The value of    false              is    false.
The value of    true               is    true.
The value of    !true              is    false.
The value of    false || false     is    false.
The value of    !false             is    true.
```

When you type:

```
!(false || !true)
```

Java sees:



And evaluates it from the inside out

What does this code do?

```
if (Robot.frontIsClear())
{
    Robot.move();
}
if (Robot.frontIsClear())
{
    Robot.move();
}
if (Robot.frontIsClear())
{
    Robot.move();
}
Robot.makeDark();
```

goes to wall and makes dark

(unless wall is more than 3 spaces away)

What does this code do?

```
if (Robot.frontIsClear())
{
    Robot.move();
}
if (Robot.frontIsClear())
{
    Robot.move();
}
if (Robot.frontIsClear())
{
    Robot.move();
}
Robot.makeDark();
```

What if we want to walk to a wall that's up to 5 squares away?

copy 2 more ifs

What does this code do?

```
if (Robot.frontIsClear())
{
    Robot.move();
}
if (Robot.frontIsClear())
{
    Robot.move();
}
if (Robot.frontIsClear())
{
    Robot.move();
}
Robot.makeDark();
```

What if we don't know how far the wall is?

We need to learn something new!

PROBLEMS

Walk to the next wall.

Light an arbitrary number of candles.

Clear an arbitrarily large rectangle of dark spots.

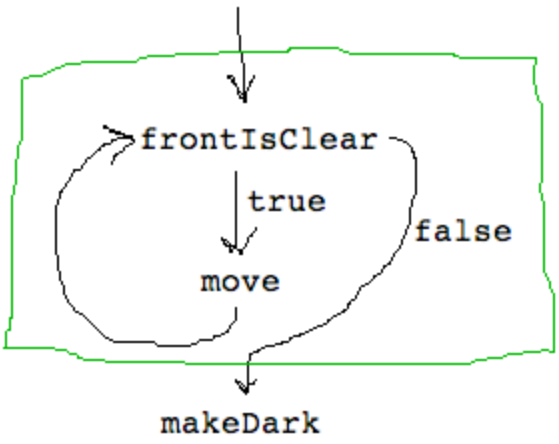
All require the ability to repeat something until some condition.

Introducing the very last topic on the course ...

(alphabetically)

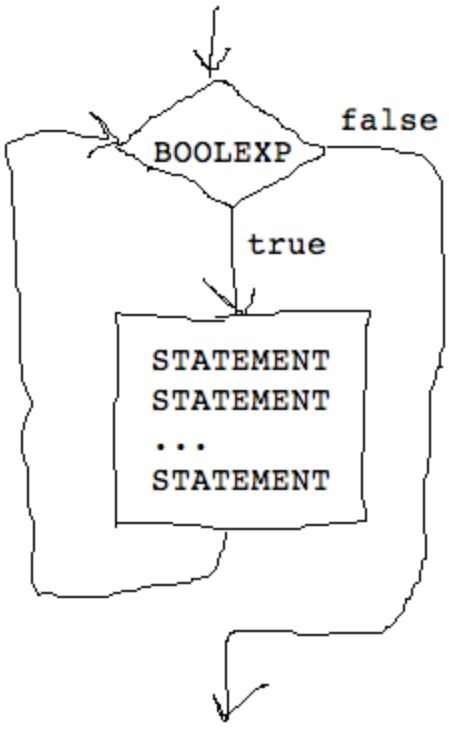
Introducing while loops:

```
while (Robot.frontIsClear())
{
    Robot.move();
}
Robot.makeDark();
```



In general:

```
while ( BOOLEXP )
{
  STATEMENT
  STATEMENT
  ...
  STATEMENT
}
```



Rule for while loops:

1. Find value of boolean expression.
2. If true, execute entire loop body, and go back to 1.
   If false, continue to next statement.

4 kinds
of
statements

```
FILENAME . METHODNAME ();

if ( BOOLEXP )
{
  STMTs
}
else
{
  STMTs
}

return BOOLEXP ;

while ( BOOLEXP )
{
  STMTS
}
```

WHEN TO USE

Use a while statement whenever you want something to repeat until some condition.

HOW TO USE

1. Determine what repeats, and write in the loop body.
2. Ask yourself: "When should my loop stop?"
3. Write a boolean expression for the opposite condition: ("When should my loop keep going?")

STUFF TO KEEP IN MIND

* when the loop condition will be tested
* edge cases (first and last time through loop)

```
E:..::...::..:::.::


public static void clearToWall()
{
    ...
}
```

```
//makes all squares light from here to wall
public static void clearToWall()
{
    while (Robot.frontIsClear())
    {                                   Never clears
        Robot.move();                   first square!
        clearSquare();
    }
}

public static void clearSquare()
{
    if (Robot.onDark())
    {
        Robot.makeLight();
    }
}
```

```
//makes all squares light from here to wall
public static void clearToWall()
{
    while (Robot.frontIsClear())
    {
        clearSquare();
        Robot.move();
    }
}

public static void clearSquare()
{
    if (Robot.onDark())
    {
        Robot.makeLight();
    }
}
```

Never clears last square!

```
//makes all squares light from here to wall
public static void clearToWall()
{
    clearSquare();
    while (Robot.frontIsClear())
    {
        Robot.move();
        clearSquare();
    }
}

public static void clearSquare()
{
    if (Robot.onDark())
    {
        Robot.makeLight();
    }
}
```

Handles first square as special case

```
//makes all squares light from here to wall
public static void clearToWall()
{
    while (Robot.frontIsClear())
    {
        clearSquare();
        Robot.move();
    }
    clearSquare();
}

public static void clearSquare()
{
    if (Robot.onDark())
    {
        Robot.makeLight();
    }
}
```

Handles last square as special case

```
//makes all squares light from here to wall
public static void clearToWall()
{
    while (Robot.frontIsClear())
    {
        clearSquare();
        Robot.move();
    }
    clearSquare();
}

//after: square robot is on is now light
public static void clearSquare()
{
    if (Robot.onDark())
    {
        Robot.makeLight();
    }
}
```

What MUST be true here?
front is clear

How about here?
no clue about front

What MUST be true here?
front is not clear

```
while (Robot.onDark())
{
    if (Robot.onDark()) ...
    ...
}
if (Robot.onDark()) ...
```

Silly, because we already know it's on dark.

Silly, because we already know it can't be on dark.

What do these loops do?

```
while (Robot.onDark())
{
    Robot.turnLeft();
}
```

If on dark, turns forever. Otherwise, does nothing.

```
while (true)
{
    Robot.turnLeft();
}
```

Always turns forever. "Infinite loop."

```
while (true)
{
}
```

Does nothing forever. Program "freezes."

clearRectangle, revisited ...

Robot is facing bottom of a rectangular region of dark squares of unknown length/width.

Task: make those squares light

(The immediately surrounding squares are light colored.)

In DrJava ...

```
//before:  robot is on light square, facing row of darks
//after:   robot in original position/direction.
//         darks have been cleared
public static void clearRow()
{
    Robot.move();
    while (Robot.onDark())
    {
        Robot.move();
    }
    backUp();
    while (Robot.onDark())
    {
        Robot.makeLight();
        backUp();
    }
}
```

```
public static void clearRectangle()
{
    while (frontIsDark())
    {
        clearRow();
        Robot.turnLeft();
        Robot.move();
        turnRight();
    }
}
```

Note: Now we've got loops within loops, since clearRow has its own loop.

```
public static boolean frontIsDark()
{
    Robot.move();
    if (Robot.onDark())
    {
        backUp();
        return true;
    }
    else
    {
        backUp();
        return false;
    }
}
```

```
public static void clearRectangle()
{
    Robot.move();
    while (Robot.onDark())
    {
        while (Robot.onDark())
        { Robot.move(); }
        backUp();
        while (Robot.onDark())
        {
            Robot.makeLight();
            backUp();
        }
        Robot.turnLeft();
        Robot.move();
        turnRight();
        Robot.move();
    }
}
```

DON'T WRITE CODE LIKE THIS!

no boolean method

nested while loops in same method, instead of putting each loop in its own method

This code is too hard to read, and therefore too hard to get right.

```
//before:  robot is on light square, facing row of darks
//after:   darks cleared.  on last cleared square.
public static void clearRow()
{
    while (frontIsDark())
    {
        Robot.move();
        Robot.makeLight();
    }
}

public static void clearRectangle()
{
    while (frontIsDark())
    {
        clearRow();
        Robot.turnLeft();
    }
}
```

An elegant solution.

Robot clears rectangle in spiral order.

The Racing Problem ...

(See Race.java)