Alfred Nobel’s Interactive Will

TOUCH TO EXPLORE

Lecture 11
Loops

“Life is just one damn thing after another.”
-Mark Twain

“Life isn’t just one damn thing after another… it’s the same damn thing over and over and over again.”
-Edna St. Vincent Millay
Before we see loops, we need some tools
- Turtle: will draw on the screen for us
  - Based on Seymour Papert’s Logo, a language for beginners
- PaneOrganizer: instructions for the turtle
  - Reminiscent of our first Robot example…

Turtles know where they are and what direction they are facing, can move and turn
Turtles can draw lines behind them as they move around the screen, or just move w/o drawing
Now let’s look at the Turtle class

public class Turtle {
    // instance variables elided
    /* constructor for Turtle
       instantiates a Polygon
       representing the Turtle graphically */
    public Turtle() {
        // some code here
    }
    /* reset turtle to center of pane */
    public void home() {
        // some code here
    }
    /* turn right a specified number of degrees */
    public void right(double degrees) {
        // some code here
    }
    /* turn left a specified number of degrees */
    public void left(double degrees) {
        // some code here
    }
    // continued

    public void forward(int distance) {
        // some code here
    }
    /* move backward a specified distance, drawing a
     line as the turtle moves */
    public void back(int distance) {
        // some code here
    }
    /* move turtle to a specified position without
     drawing a line */
    public void setLocation(Point2D loc) {
        // some code here
    }
    /* return turtle’s location */
    public Point2D getLocation() {
        // some code here
    }
    /* returns the Polygon contained in Turtle class
     so that we can graphically add it in the P.O. */
    public Node getNode() {
        // some code here
    }
}

TAs have written a Turtle class

Need class to tell Turtle how to draw some basic shapes
First, determine what shapes we want
  - this lecture: square, random walk
  - next lecture: recursive spiral, tree, fractal

How will we code it?
  - create PaneOrganizer class which defines methods for drawing each shape
  - the PaneOrganizer also instantiates the root Pane that the Turtle will draw on and contains the Turtle
  - Turtle is a wrapper class that contains a polygon (a triangle) and defines methods for how the Turtle will move; it can also return the shape as a node via getNode
Drawing with Turtle

- Time for some code!

```java
public class PaneOrganizer {
    private Turtle _turtle; // draws each pattern
    private Pane _root;

    public PaneOrganizer() {
        _root = new Pane();
        _turtle = new Turtle();
        _root.getChildren().add(_turtle.getNode());
    }
    // methods for each geometric pattern to follow...
}
```

A Repetitive Solution

- Let's write the `drawSquare` method in the `PaneOrganizer` class
- Brute force: write line of code for each side

```java
public void drawSquare(int sideLen) {
    _turtle.forward(sideLen);
    _turtle.right(90);
    _turtle.forward(sideLen);
    _turtle.right(90);
    _turtle.forward(sideLen);
    _turtle.right(90);
    _turtle.forward(sideLen);
    _turtle.right(90);
}
```

A Repetitive Solution

- What if we wanted to make a more general method that handles regular shapes such as pentagons or octagons?
  - need to call forward and right for each side
  - cannot fix how many sides we need in generic method
  - note that we're using the Turtle's primitive methods to generate higher-level shapes that are normally already defined in JavaFX
- There must be an easier way!

Looping

- Execute a section of code repeatedly
  - uses booleans (true and false) as loop conditions; when boolean is false, loop condition equals exit condition and loop is terminated
  - as with conditionals, code in the loop can be a single line or many lines enclosed in curly braces
  - section of code executed is called loop’s body
Looping

- Three loop structures in Java
  - `while` loop
  - `do while` loop
  - `for` loop
- Differ in relation between body and loop condition, as well as length of execution
  - `do while` loops execute body and then test condition; others test condition before executing body
  - `for` loops run for a predetermined number of iterations; others run for indefinite amount of time
- Let's look at `while` loop first

The while loop

- Execute `while` certain condition is true
  - tests loop condition before executing body
  - if loop condition is `false` first time through, body is not executed at all

```
while (<loop condition>) {
   <loop body>
}
```

The while loop

- Examples of loop conditions:
  ```
  numClasses < 6
  peopleStanding <= maxPeople
  this.checkAmount() <= acctBalance
  this.isSquare() //predicate, a method that returns a boolean
  ```
- Follow the same rules as conditions for `if-else` statements
- Multiple conditions can be combined using logical operators (`and`, `or`, `not`)
  ```
  (numClasses >= 3) && (numClasses <=5)
  (peopleStanding <= maxPeople) || (maxPeople < 50)
  ```

A while Flowchart

- `while` loops continue `while` loop condition is `true`
- Loop condition can be any boolean expression
A while Flowchart

- **while** loops continue while loop condition is true
- Loop condition can be any boolean expression

All Flow of Control Structures: 1-in, 1-out

- Benefits of predictable flow of control:
  - much easier debugging
  - compiler can optimize much better
- Contrast with “spaghetti” code produced by having **go to** construct which allows for jumping to another line of code
  - Go To Statement Considered Harmful letter by Edsger Dijkstra, 1968
  - “Go to”-less programming called “structured programming”, took a while to get traction

So, just how bad is **goto**?

Syntax: Random Walk Using **while**

- Method of PaneOrganizer class:
  - draws random lines while turtle is within its pane
  ```java
  public void randomWalk() {
      // while _turtle's position is inside its
      // pane, move _turtle randomly
      while (_root.contains(_turtle.getLocation())) {
          _turtle.forward ((int) (Math.random() * 15)); // cast to [0-14]
          _turtle.right ((int) (Math.random() * 360)); // cast to [0-359]
      }
  }
  ```
- On last step of walk, _turtle will move forward out of pane
  - the line is clipped by JavaFx since we don’t explicitly tell it to wrap around
  - no point in continuing to walk outside the pane
The **do while** Loop

- **do while** always executes loop body at least once by switching order of test and body
- **<loop condition>** is boolean expression

---

Example: Another Random Walk

- Method of PaneOrganizer class:
  - draws random lines while turtle is within pane
  - starts turtle in center of root pane, so first step guaranteed to be within pane

```java
public void centeredRandomWalk() {
    // moves turtle to Pane's center
    _turtle.home();
    // moves turtle randomly within pane
    do {
        _turtle.forward((int)(Math.random()*15));
        _turtle.right((int)(Math.random()*360));
    } while (_root.contains(_turtle.getLocation()));
}
```

Note the semicolon at the end of while statement

---

**do while vs. while**

- In both loops:
  - stops executing body if loop condition is false
  - must make sure loop condition becomes false by some computations to avoid an “infinite loop”
  - **infinite loop** means your loop condition will never turn false—i.e., exit condition never occurs (and your program “freezes up”!)

- **do while**
  - body always executed at least once
  - loop condition tested at bottom of loop

- **while**
  - may not execute at all
  - loop condition tested before body; loop condition variables must be set before loop entry
  - useful for screening bad data that might cause statements within loop to fail (e.g. **while (ref != null)**)

---

**do while vs. while**

What is the difference between these two?

```java
while (andyIsAway) {
    _tas.takeDayoff();
}
```

- **Condition tested before body**
- Body executes before condition is tested; TA’s take off at least one day
**for loops**

- Most specialized loop construct: typically used to execute loop body a *predetermined* number of times
  - *while* and *do while* loops can execute body for undetermined number of times; based on *boolean*
    ```java
    for (<init>expr>; <loop condition>; <update>) {
        <loop body>
    }
    ```
- `<init>expr`
  - expression for setting initial value of loop counter (traditionally use single char. identifier, e.g., `i`)
  - executed at start of loop code, i.e., only once, not for each time through the loop

**drawSquare Revisited**

- Better way of drawing square than explicitly drawing each side:
  ```java
  public void drawSquare(int sideLen) {
      /* start with integer i initialized to 0; execute as long as i < 4; each execution increments i by 1 */
      for (int i = 0; i < 4; i++) {
          _turtle.forward(sideLen);
          _turtle.right(90);
      }
  }
  ```

**for loops**

- Most specialized loop construct: typically used to execute loop body a *predetermined* number of times
  - *while* and *do while* loops can execute body for undetermined number of times; based on *boolean*
    ```java
    for (<init>expr>; <loop condition>; <update>) {
        <loop body>
    }
    ```
- `<loop condition>`
  - *true* or *false*
  - test involves loop counter to determine if loop should execute
  - checked at start of every loop (including the first)
- `<update>`
  - expression that modifies loop counter
  - run at end of every `<loop body>`, just before returning to the top of the loop
**for Flowchart**

- **for** loop has four parts
  - initialize value of counter
  - test loop condition
  - loop body
  - counter update

**for Flowchart**

- **for** loop has four parts
  - initialize value of counter
  - test loop condition
  - loop body
  - counter update

---

### Choosing the Right Loop (1/2)

- **for** loop is called a *definite* loop because you can typically predict how many times it will loop
- **while** and **do while** loops are *indefinite* loops, as you do not know when they will end
- **for** loop is typically used for math-related loops like counting finite sums

---

### Choosing the Right Loop (2/2)

- **while** loop is good for situations where boolean condition could turn false at any time
- **do while** loop is used in same type of situation as **while** loop, but when code should execute at least once
- *When more than one type of loop will solve problem, use the cleanest, simplest one*
Syntax: Nested Loops

- Loops, as with if statements, can be nested!

Example: `drawFilledSquare`

```java
public void drawFilledSquare(int sideLen) {
    // fill in concentric squares
    for (int i = 0; i < (sideLen / 2); i++) {
        // drawSquare contains a loop
        this.drawSquare(sideLen - (2 * i));
        /* note we can use loop counter in
           body but never reset it there! */
        // position turtle for next iteration
        _turtle.right(90);
        _turtle.forward(1);
        _turtle.left(90);
        _turtle.forward(1);
    }
}
```

- What does this do?
  - decrementing `sideLen` by 2 each iteration to guarantee that each “inner square” drawn by `drawSquare(…)` is exactly one unit away on either side from square immediately “outside” of it (hence, one+one = two)

- What is the outer loop doing?
  - first draws outer square

Decrementing Counter

- We can count backwards in our loop too
  - just change the counter update expression
  - in fact, we can update however we want

```java
public void countDownSeconds() {
    /* change counter to decrement, and change the loop condition accordingly */
    for (int i = 5; i > 0; i--) {
        timer.display(i);
    }
}
```

- for loops end in one of two ways
  - when counter value equals limit (for < or >)
  - when counter value “goes past” limit (for <= or >=)
  - thus, `countDownSeconds()` would display 6 seconds if used `i >= 0` (bad idea because movie should play on the 0th second, post-loop)
  - Beware of such “off-by-one” errors!

**break**

- `break` causes immediate exit from a flow-of-control structure (e.g. while, do while, for, switch)

- Example:

```java
for (int i = 0; i < 10; i++) {
    if (_cookieJar.getNumberOfCookies() == 0) {
        break; //If there are no cookies left, we should break out of the loop!
    }
    this.eatAcookie();
}
```

- Execution continues with first line of code after structure
- There are other ways to do this loop…
**continue**

- When used in `while`, `for`, or `do while` structures, `continue` skips remaining statements in body of that structure and proceeds with next iteration of loop
  - useful if there is list of data that you are looping over and you want to skip processing of data that is somehow “not allowed”
- In `while` and `do while` structures, execution continues by evaluating loop-continuation condition
- In `for` structure, execution continues by incrementing counter and then evaluating loop condition

---

**Boolean Flags**

- A boolean flag is a boolean variable that denotes a condition (e.g., `isDone`, `isWorking`, `isAvailable`)
  - set in one place, tested in another
  - similar to boolean methods, often starts with “is” or “has” by convention
- Boolean flags can also be used as loop condition
- Example (implementing a `for` loop, using `while`):
  ```java
  boolean isDone = false;
  int i = 0;
  while (isDone){
    if(i == 5){
      isDone = true;
    }
  }
  ```
  - Notice that boolean flag is set within loop
    - in previous slides, all checking was done through delegation (to methods that returned booleans)
    - here, we do it ourselves (not practical)

---

**Empty Intervals**

- Example scenario: we want to keep a running sum of a sequence of numbers
- What happens if we try to add integers in this loop?

```java
public int sum() {
    int tempSum = 0;
    for (int i = 1; i < 1; i++) {
        tempSum += i;
    }
    return tempSum;
}
```
- Answer: body of loop is not executed
- Why?
  - boolean is false for initial counter value
Correct Example

- What about this loop?

```java
/*This method sums all numbers from 1 up to and including 10 */
public int sum() {
    int tempSum = 0;
    for (int i = 1; i <= 10; i++) {
        tempSum += i;
    }
    return tempSum;
}
```

- It will work!

Off-by-one Errors

- Occur when loop executes one too many or one too few times
- Example: add even integers from 2 to some `number`, inclusive

```java
... count = 2;
result = 0;
while (count < number) {
    result += count;
    count += 2;
}
```

- Produces incorrect result if `number` is assigned an even value. Values from 2 to `number`-2 will be added (i.e. `number` is excluded)

- Should be:

```
while (count <= number) {
    ...
}
```

- Now, value of `number` is included in summation

Syntax: Other Loop Errors

- Make sure test variables have proper values before loop is entered

```
... product = 0;
do {
    product *=2;
} while (product < 100);
/* what will happen here */
```

- Make sure tests check proper conditions

```
... for (int i = 1; i <= 100; i += 2) {
    // do something here
} /* will we ever get here */
```

Syntax: Other Loop Errors

- ALWAYS HAND SIMULATE first, last, and typical cases through a loop to avoid off-by-one or infinite loop errors
  - the first and last cases of a loop's execution are called boundary conditions or edge cases
  - hand simulation doesn't just apply to loops – use it for everything! Trust us – it saves debugging time!
Which loop to use?

- You want to shoot 12 baskets
- Your job is to stand at the end of the bowling alley and pick up all the pins, one by one, that have been knocked over
- Ask a question and keep asking while the answer is incorrect
- Sleep until your clock reads 7:00AM or later

Announcements

- Cartoon Competition
  - Every year, we pick some of the most ambitious and creative cartoons to showcase in class. Winners get to grab lunch with Andy!
- Stricter Hours
  - Please read all the resources before coming to TA hours – else we will have to turn you away.
- Cartoon DQs due tomorrow @10pm
  - Cartoon early hand-in Thursday 10/22 @ 11:59pm
  - Cartoon regular hand-in Friday 10/23 @10pm