Collaboration Policy Reminder

- Brown’s policy is CS15’s policy: All written work must be your own
- We use MOSS to check for plagiarism
- MOSS is not fooled by renaming, resequencing, etc; it does a structural analysis
  - We carefully hand check high MOSS scores, not completely automated
- Regret Policy
- If ever in doubt, an NC and redo is far better than a directed NC on your transcript and parental notification

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

Outline

- **Turtle**
- **Looping**
- **while Loops**
  - Clicker Questions: Slides 34 and 39
- **for Loops**
- **Choosing the Right Loops**
  - Clicker Questions: Slides 56 and 61
Introduction to Turtle (1/2)
• Before we see loops, we need some tools:
  o We will use a Turtle ▲ to help us understand loops
  o Turtles are based on Seymour Papert’s Logo*, a language for beginners
• Turtles ▲ are imaginary pens that when given instructions can draw shapes for us


Introduction to Turtle (2/2)
● Turtles know where they are and what direction they are facing, and can move and turn
● Turtles can draw lines behind them as they move around the screen, or just move without drawing
● PaneOrganizer holds instructions for the turtle
  o Reminiscent of our first Robot example…

Turtle’s Methods (1 of 2)

```java
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
}
```

TAs have written a Turtle class
/* move forward a specified distance, drawing a line as the turtle moves */
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
}

Turtle's Methods
(2 of 2)

Drawing with Turtle
(1/2)

● Need class to tell Turtle how to draw some basic shapes
  o will contain a Pane and a Turtle
  o will have methods for each shape we want to draw

● First, determine what shapes we want
  o this lecture: square, random walk

Drawing with Turtle
(2/2)

● How will we code it?
  o create PaneOrganizer class which defines methods for drawing each shape
  o PaneOrganizer also instantiates the root Pane that the Turtle will draw on and contains the Turtle. The root is returned in getRoot()
  o 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 polygon as a node via getNode()
A Repetitive Solution (1/2)

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

```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 (2/2)

- 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 (1/2)

- Execute a section of code repeatedly
  - uses `booleans` (true and false) as loop conditions; continue looping as long as it is true, but 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 (2/2)

- 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
- Let's look at **while** loop first

The **while** loop (1/2)

- Executes **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 (2/2)

- Examples of loop conditions:
  - `numClasses < 6`
  - `peopleStanding <= maxPeople`
  - `this.checkAmount() <= acctBalance`
  - `this.isSquare()`  // predicate, a method that returns a boolean
- Follows 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)
```
while loop Flowchart (1/2)
- while loops continue while the loop condition is true
- <loop condition> can be any Boolean expression
  - <loop condition> true?
    - Yes: <loop body>
    - No: <rest of program>
  - <previous statement>

while loop Flowchart (2/2)
- while loops continue while the loop condition is true
- <loop condition> can be any Boolean expression
  - Is The Ratty open?
    - Yes: Get more food.
    - No: Go to Jo's.
  - Walk into the Ratty.
  - <previous statement>

All Flow of Control Structures: 1-in, 1-out
- Benefits of predictable flow of control:
  - much easier debugging
  - compiler can optimize much better
- Different from "spaghetti" code (unorganized and difficult to maintain code) by having a go to construct which allows the computer to jump to another line of code
  - Go To Statement Considered Harmful letter by Edsger Dijkstra, 1968
- One way in
  - Flow of Control Structure
  - One way out
  - <rest of program>
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
     _turtle's initial location set to (x,y)
    while (_root.contains(_turtle.getLocation()))
    {  // cast to [0-360]
         _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

TopHat Question 1
What is the value of `tempSum` after this `while` loop is terminated?

```java
int tempSum = 0;
while(tempSum < 10) {
    tempSum += 3;
}
```

A. 10  
B. 9  
C. 12  
D. The loop will never terminate
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
  - _turtle starts 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 (1/2)

- 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 vs. while (2/2)**

- **do while**
  - o Body always executes at least once
  - o Loop condition tested at bottom of loop body

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

---

**TopHat Question 2**

What's the difference between these two loops?

Loop 1:

```
while(andyIsAway()) {
  _tas.takeADayOff();
}
```

A. In the second loop, the condition is tested before the body
B. In the second loop, the TAs always take at least 1 day off
C. In the first loop, the body is executed before the condition is tested.
D. There is no difference between the two loops

---

**for loops (1/4)**

- Most specialized loop construct (and the first high-level, go-to-less loop in FORTRAN): typically used to execute loop body a predetermined number of times
  - o **while** and **do while** loops can execute body for undetermined number of times; based on boolean

- This is the syntax for a **for** loop:
  ```
  for (<init-expr>; <loop condition>; <update>) {
    <loop body>
  }
  ```
for loops (2/4)

\[
\text{for (init-expr}; \text{loop condition}; \text{update}) \{
\quad \text{loop body}\n\}
\]

- \text{init-expr}
  - expression for setting initial value of loop counter (traditionally use single char. identifier; e.g., \text{i})
  - executed at start of loop code, only once, not for each time through the loop

for loops (3/4)

\[
\text{for (init-expr}; \text{loop condition}; \text{update}) \{
\quad \text{loop body}\n\}
\]

- \text{loop condition}
  - true or false
  - test involves loop counter to determine if loop should execute
  - checked at start of every loop (including the first)

for loops (4/4)

\[
\text{for (init-expr}; \text{loop condition}; \text{update}) \{
\quad \text{loop body}\n\}
\]

- \text{update}
  - expression that modifies loop counter
  - run at end of every \text{loop body}, just before returning to the top of 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 Flowchart**

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

- *prevous statement* → *init-counter* → *Is <loop condition> true?* → *<loop body>* → *update-counter* → *<rest of program>*

---

**for Flowchart**

- We can use an example of a student reading books on different floors of the SciLi.

```
Student student = new Student("Creed");
student.goToSciLi();
for (int floor = 1; floor < 14; floor++) {
    student.readBook(); //read a new book
}
student.goHome();
```

- Note: For this example, we use the old SciLi, where every floor had books!
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**.

TopHat Question 3

What is the value of `sum` at the end of the following loop?

```
sum = 0;
for (int i = 0; i <= 10; i+=2) {
    sum+=;
}
```

A. 10   B. 11   C. 5   D. 6
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++) {
        for (int j = 0; j < 4; j++) {
            _turtle.forward(sideLen - (2*i));
            _turtle.right(90);
        }
        /* note we can use loop counter R/O (read-only)
           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 in the inner loop is
    exactly one unit away on either side from square
    immediately "outside" of it (hence, one + one = two)

Syntax for Nested Loops Explained

- Turtle is represented by ▲
- What is the outer loop doing?
  - first draws outer square

  - Turtle starts upright!
  - Rotate 90 degrees right!
  - Move forward 1 unit!
  - Rotate 90 degrees left!
  - Move forward 1 unit!
  - Draw inner square

drawFilledSquare draws concentric squares; each individual square is
drawn using the nested loop.

Looping to Make a Filled-in Square

- 3D Printing a Pizza
  https://www.youtube.com/watch?v=ISXqC-YPnpC
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--){
      System.out.print(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`

- beware of such “off-by-one” errors! → hand simulation really helps!

break

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

  ```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 legal”

- 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
continue Example

```java
// We'd like to try on shirts that hang on a rack
for (int i = 0; i < 20; i++) {
  if(!rack.isShirtOnHanger(i)) {
    // If there's no shirt on the current hanger,
    // skip to the next iteration
    continue;
  }
  // Only do this if there's a shirt on the hanger
  this.tryOnShirt(rack.getShirtOnHanger(i)); // Get the shirt and try on shirt
}
```

// more code here

---

### 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) {
  i++;
  if (i == 5) {
    isDone = true;
  }
}
```

*Note: Here, the Boolean flag is set within loop, which, though legal, is not practical. In previous slides, all checking was done through delegation (to methods that returned booleans) — these are called predicates, e.g., `isShirtOnHanger(i)`."

---

**TopHat Question 4**

In the loop to the right, what is the value of `i` upon exit?

A. 4  
B. 5  
C. 6  
D. Infinite loop
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?
  - loop condition 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

- These 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;
    }
    ```
  - Should be:
    ```java
    while (count <= number) {
    ```
  - Corrected result:
    ```java
    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)
- Now, value of number is included in summation
Syntax: Other Loop Errors (1/2)

- Make sure test variables have proper values before loop is entered
  ```
  ... 
  product = 4;
  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? */
  ```

TopHat Question 5

Given the following code:
```java
num = 2019;
do {
  num--;
} while (num < 2019);
```
What do you expect will happen?
A. Loop will never end
B. Loop will run 2019 times (until `num` is 0), then end
C. Loop will run only once

Syntax: Other Loop Errors (2/2)

- 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 or corner 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 stack 17 sandwiches
- 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
- Sleep until your clock reads 7:51AM or later

Announcements

- Reminder that Cartoon check-ins are happening this week; email your section TAs if you have not heard from them yet!
- Cartoon help slides going out today
- Cartoon is the first project without support code, as projects get longer, remember: “Start early, start today, start yesterday!”