REPETITION

CITS1001
Scope of this lecture

• Repetition
• For loops
• While loops
Repetition

• Computers are good at repetition
• We have already seen the for each loop
• The for loop is a more general loop form
• It is often used to iterate (repeat an activity) a fixed number of times.
For loop pseudo-code

General form of the for loop

```plaintext
for(initialization; condition; post-body action) {
    statements to be repeated
}
```

Equivalent in while-loop form

```plaintext
initialization;
while(condition) {
    statements to be repeated
    post-body action
}
```
A Java example

for loop version

```java
for(int count = 0; count < 100; count++) {
    System.out.println("I can count to: " + count);
}
```

while loop version

```java
int count = 0;
while(count < 100) {
    System.out.println("I can count to: " + count);
    count++;
}
```
FOR LOOP
The **for** loop

The for loop is the most important looping structure, although it takes some getting used to. It has the following syntax:

```java
for (<initialization>; <boolean-expression>; <post-body update>)
{
    <statement-1>
    <statement-2>
    ...
    <statement-n>
}
```

The *body* of the loop is the collection of statements between the curly brackets.
The initialization part

- The initialization part consists of any Java statement
- It is performed *once only* when execution first reaches the *for* loop
- It is normally used to initialize a *counter* variable
The boolean-expression part

- The boolean expression controls whether or not the body of the loop is executed
- The expression is evaluated immediately after the initialization has occurred
- If its value is `true`, then all of the statements in the body of the loop are executed in turn
- If its value is `false`, then the loop has finished, none of the statements in the body are executed, and execution continues at the first statement after the `for` loop
The post-body update

• The post-body update is a Java statement that is executed once *each time* the body of the *for* loop is executed
• It is executed immediately after the last statement of the body has been executed
• It is usually used to *update* the counter variable
The for loop

- Before loop
- Initialization
- Is the boolean expression true?
  - Yes
    - Perform Body
    - Post-body update
  - No
    - After loop
The **for** loop idiom

- **for**-loops are most often used when we want to do something a specific number of times

```java
for (int i=0; i<5; i=i+1)
{
    System.out.println(i);
}
```

`System.out.println` is a library method that just prints out its argument to the terminal window.
How did this work?

Initialization creates $i$ and sets it equal to 0
Check if $i < 5$, yes
Print out $i$ – causes 0 to appear on terminal window
Update $i$ from 0 to 1
Check if $i < 5$, yes
Print out $i$ – causes 1 to appear on terminal window
Update $i$ from 1 to 2
Check if $i < 5$, yes
Print out $i$ – causes 2 to appear on terminal window
Update $i$ from 2 to 3
Check if $i < 5$, yes
Print out $i$ – causes 3 to appear on terminal window
Update $i$ from 3 to 4
Check if $i < 5$, yes
Print out $i$ – causes 4 to appear on terminal window
Update $i$ from 4 to 5
Check if $i < 5$, NO
The increment operator

- The post-body update so often consists of
  \[ i = i+1; \]
  that there is a short-hand notation for this operation
- The statement \( i=i+1 \) may be replaced simply by \( i++ \)
  ```java
  for (int i=0; i<5; i++) {
    System.out.println(i);
  }
  ```
- In fact \( i++ \) is both a statement and an expression
  - as an expression it has the value of \( i \) after the incrementing

**Warning:** Either use \( i=i+1; \) or \( i++ \); but never mix them together and use \( i=i++; \)
A one-statement body

- If the body consists of only one statement, then you can leave out the braces BUT it is good style to ALWAYS include them

```java
for (int i=0; i<10; i++) {
    System.out.println(i*i);
}
```

is the same as

```java
for (int i=0; i<10; i++)
    System.out.println(i*i);
```

You have been warned! Many hard to trace bugs arise from leaving out the brackets in a statement block
Common mistake

• A surplus semicolon! (this mistake is hard to track down)
  ```java
  int i;
  for (i=0; i<10; i++) ;
  {
    System.out.println(i*i);
  }
  ```
• The output from this code is just
  100
• Why? What went wrong with the loop?
  • There was a problem with the body
Common mistake

```java
int i;
for (i=0; i<10; i++) {
    System.out.println(i*i);
}
```

```java
int i;
for (i=0; i<10; i++)
{
    System.out.println(i*i);
}
```

The first `for` loop has an *empty body* (just a single semicolon!), while the second shows the "desired" body.
Making tables

- Another common use of `for` loops is to produce *tables*
- For example, suppose you are asked to produce a temperature conversion table listing the Fahrenheit equivalents of 0°C - 100°C in increments of 5 degrees

<table>
<thead>
<tr>
<th>Temperature (°C)</th>
<th>Fahrenheit (°F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>41</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>15</td>
<td>59</td>
</tr>
<tr>
<td>20</td>
<td>68</td>
</tr>
</tbody>
</table>
A for loop is the solution

```java
int celsius;
int fahrenheit;

for (celsius=0; celsius <= 100; celsius = celsius + 5) {
    fahrenheit = 32 + celsius*9/5;
    System.out.print(celsius);
    System.out.print(" ");
    System.out.println(fahrenheit);
}
```

Notice: the easy-to-read variable names
Notice: the use of `System.out.print()` to print things without starting a new line
A numerical example

• It is well known that

\[ \pi = 4 (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots ) \]

• Suppose that we wish to approximate \( \pi \) using this formula

• There are two ways that we can do it
  • approximate by adding up a *fixed number* of terms
  • approximate to with a *certain accuracy*

• The first way is best done using a `for` loop
**π to a fixed number of terms**

```java
public double pi(int n) {
    double approx = 0;
    double mult = 4;
    for (int i=0; i<n; i++) {
        approx = approx + mult/((2*i)+1);
        mult = -mult;
    }
    return approx;
}
```
Situation at top of loop

<table>
<thead>
<tr>
<th>i</th>
<th>mult</th>
<th>(2*i+1)</th>
<th>approx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4.0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>-4.0</td>
<td>3</td>
<td>4.0</td>
</tr>
<tr>
<td>2</td>
<td>4.0</td>
<td>5</td>
<td>2.666666667</td>
</tr>
<tr>
<td>3</td>
<td>-4.0</td>
<td>7</td>
<td>3.466666667</td>
</tr>
<tr>
<td>4</td>
<td>4.0</td>
<td>9</td>
<td>2.895238095</td>
</tr>
<tr>
<td>5</td>
<td>-4.0</td>
<td>11</td>
<td>3.339682540</td>
</tr>
<tr>
<td>6</td>
<td>4.0</td>
<td>13</td>
<td>2.976046176</td>
</tr>
</tbody>
</table>

Stops when i reaches the requested value
Practice

Make a circle object and move it 20 places to the right

Circle c = new Circle();
c.makeVisible();

for ...

c.moveRight();
Practice

Print the first 10 even numbers greater than 0

2 4 6 8 10 12 14 16 18 20 ...

for ...
for loop with bigger steps

// Print multiples of 3 that are below 40.
for(int num = 3; num < 40; num = num + 3) {
    System.out.println(num);
}
For loops are used when an index variable is required
For loops are used when the number of repetitions is known
Used with a regular step size

For each loops are less error-prone than for loop
So use a for each loop unless you need access to indices or step size
WHILE LOOP
Search tasks are indefinite

- We cannot predict, *in advance*, how many places we will have to look.
- Although, there may well be an absolute limit – i.e., checking every possible location.
- ‘Infinite loops’ are also possible, either through error or the nature of the task.
The while loop

- A for-each loop repeats the loop body for each object in a collection.
- Sometimes we require more variation than this.
- We use a boolean condition to decide whether or not to keep going.
- A while loop provides this control.
While loop pseudo code

General form of a while loop

while keyword

while(loop condition) {
  loop body
}

boolean test

Statements to be repeated

Pseudo-code expression of the actions of a while loop

while we wish to continue, do the things in the loop body
Looking for your keys

```
while (the keys are missing) {
    look in the next place;
}
```

Or:

```
while (not (the keys have been found)) {
    look in the next place;
}
```
Looking for your keys

```java
boolean searching = true;
while(searching) {
    if(they are in the next place) {
        searching = false;
    }
}
```

Suppose we don’t find them?
/**
 * List all file names in the organizer.
 */

public void listAllFiles()
{
    int index = 0;
    while(index < files.size()) {
        String filename = files.get(index);
        System.out.println(filename);
        index++;
    }
}
Elements of the loop

- We have declared an index variable.
- The condition must be expressed correctly.
- We have to fetch each element.
- The index variable must be incremented explicitly.
Continuing a search

• With a while loop we need to state the condition for continuing:

• So the loop’s condition will be the *opposite* of that for finishing:
  
  \[
  \text{index} < \text{files.size()} \land \neg \text{found} \\
  \text{index} < \text{files.size()} \land \text{searching}
  \]

• **NB:** ‘or’ becomes ‘and’ when inverting everything.
Searching a collection

```java
int index = 0;
boolean found = false;
while(index < files.size() && !found) {
    String file = files.get(index);
    if(file.contains(searchString)) {
        // We don't need to keep looking.
        found = true;
    } else {
        index++;
    }
}
// Either we found it at index,
// or we searched the whole collection.
```
Indefinite iteration

• Does the search still work if the collection is empty?
  Yes! The loop’s body won’t be entered in that case.
• Important feature of while:
  • The body will be executed zero or more times.
for-each versus while

• for-each:
  • easier to write.
  • safer: it is guaranteed to stop.

• while:
  • we don’t have to process the whole collection.
  • doesn’t even have to be used with a collection.
  • take care: could be an infinite loop.
Searching a collection

- A fundamental activity.
- Applicable beyond collections.
- Necessarily indefinite.
- We must code for both success and failure – exhausted search.
- Both must make the loop’s condition *false*.
- The collection might be empty.
While without a collection

// Print all even numbers from 2 to 30.
int index = 2;
while(index <= 30) {
    System.out.println(index);
    index = index + 2;
}
Finishing a search

• How do we finish a search?
• *Either* there are no more items to check:
  
  \[
  \text{index} \geq \text{files.size}() 
  \]

• Or the item has been found:

  \[
  \text{found} \equiv \text{true} \\
  \text{found} \equiv !\text{searching} 
  \]
Review

- Loop statements allow a block of statements to be repeated.
- The for-each loop allows iteration over a whole collection.
- The for-loop allows bounded iteration to iterate a fixed number of times.
- The while-loop allows indefinite iteration when you don’t know how many times a loop will be repeated.
Summary of for loop rule

1. Execute initialiser statement (i=0)
2. Test exit condition expression (i<10)
3. If exit condition is false then execute loop body (in { }) and then execute loop increment statement (i++)
4. If exit condition is true then stop

You are responsible for making sure the exit condition is eventually true

```java
for (int i=0; i<10; i++) {
    System.out.println(i*i);
}
```
Summary of while loop rule

1. Evaluate the boolean expression of the while
2. If true then execute the loop body (in { })
3. Then repeat from step 1
4. If expression is not true then exit

You are responsible for ensuring that the condition does eventually become true (or the program loops forever)

```java
while(index <= 30) {
    System.out.println(index);
    index = index + 2;
}
```