Iteration: Intro

- **Iteration/repetition/looping**: Executing a sequence of instructions multiple times

- Loop components:
  - Condition: determines whether loop should continue
  - Body: instructions being repeated
  - Loop control variable (LCV): Variable whose value determines when loop stops

- Basic steps of loop control:

  Initialize Process  
  Initialize LCV

  BEGIN LOOP  
  Update Process  
  Update LCV  
  END LOOP

- Two types of loops:

  1. Pretest  
     - Condition precedes body  
     - Iterates 0+ times

![Flowchart showing the logic of a pretest loop](image-url)
Iteration: Intro (2)

2. Posttest
   - Condition follows body
   - Iterates 1+ times
Iteration: *While* Loops

- Pretest loop
- Most general loop construct
  - Only loop construct needed
- Syntax:

  ```
  while (boolean expr) 
  stmt
  ```

  - Schematically:
    ```
    while (boolean expr)
    stmt
    ```
  - Note the similarity to the *if* statement

- Semantics:
Iteration: Loop Control

• Main issue with loops is *loop control*

  – Determining
    1. How many times the loop body should execute, and
    2. The methods for controlling this

• 2 general types of loop control:

  1. Definite iteration:
     – Loop executes a fixed number of times
  2. Indefinite iteration:
     – Loop executes an indeterminate number of times
Iteration: Definite Iteration

- Is a variation of *count-controlled* iteration
- *Body* will execute a fixed number of times
- *LCV* is an example of a variable called a *counter*
  - It counts how many times something occurs
- Example

```java
// count-controlled while loop: Average <limit> integers

int limit, counter, sum, number;

limit = 10;
counter = 0;
sum = 0;
while (counter < limit) {
    number = (int) (Math.random() * 100);
    counter++;
    sum += number;
}
System.out.println("The average = " + (float) sum / limit);
```

- *counter* is the *LCV*
- *counter* is a *counter* variable
- *Sum* is an *accumulator*
  - Keeps a total
Iteration: Indefinite Iteration

- Used when don’t know ahead of time how many times loop should execute

- 3 types of control:
  1. Count controlled
  2. Sentinel
  3. Flag

- Count controlled
  - Iterations determined by occurrence of some event a specified number of times
  - Loop may need to iterate more than this number
  - Example:
    ```java
    // count-controlled while loop, indefinite iteration:
    // Average <limit> positive integers
    
    int limit, counter, sum, number;
    limit = 10;
    counter = 0;
    sum = 0;
    while (counter < limit) {
        number = (int) (Math.random() * 100);
        // Make 1 out of 5 numbers negative
        if (((int)(Math.random() * 5) == 0)
            number = -number;
        if (number >= 0) {
            counter++;
            sum += number;
        }
    }
    System.out.println("The average = " + (float) sum / limit);
    ```
Iteration: Indefinite Iteration (2)

- Sentinel control
  - *Sentinel* is one or more values that signal end of iteration
  - Usually outside of range of acceptable values used for processing
  - Example:

    // Sentinel-controlled while loop: Average non-negative
    // integers. Terminate when a negative is input.

    int counter, sum, number;
    counter = 0;
    sum = 0;
    number = (int) (Math.random() * 100);
    if ((int)(Math.random() * 5) == 0)
      number = -number;
    while (number >= 0) {
      counter++;
      sum += number;
      number = (int) (Math.random() * 100);
      if ((int)(Math.random() * 5) == 0)
        number = -number;
    }
    if (counter == 0)
      System.out.println("no integers were averaged");
    else
      System.out.println("The average = " + (float) sum / counter);

- Requires *priming read*
  - Must get initial value for LCV prior to entering loop
Flag control

- *Flag* is *LCV* that keeps track of whether a situation has occurred
- Usually Boolean
- Can be thought of as a specialized sentinel
- Often used when multiple situations are used to terminate loop
- Example:

```java
// Flag-controlled while loop: Determine if an int is prime.
// Terminate when divisor == number or if is prime

import java.util.Scanner;

public class PrimeNumber
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);

        System.out.print("Enter an integer: ");
        int x = keyboard.nextInt();
        boolean done, isPrime = true;

        int div = 2;
        while (!done) {
            int y = x % div;
            if (y == 0) {
                done = true;
            }
            else {
                div += 1;
            }
        }

        System.out.print(x);
        if (isPrime)
            System.out.println(" is a prime number");
        else
            System.out.println(" is not a prime number");
    }
}
```
1. Infinite loop
   - Loop that never ends
   - Can result due to many reasons:
     - Failure to update $LCV$
     - Failure to update $LCV$ properly
     - Poor choice of termination condition
     - Failure to update process

2. Off-by-one error
   - Count-controlled loop that iterates one too many or one too few times
Iteration: Nested Loops

- Can place one loop inside of another
- Must make sure that initializations and updates occur for each loop and in the appropriate places
- Example:

  // nested WHILE loops: Print times tables.

  int n, counter1, counter2;

  n = 5;
  counter1 = 1;
  while (counter1 <= n) {
    System.out.println(counter1 + "'s times table: ");
    counter2 = 1;
    while (counter2 <= 9) {
      System.out.print(counter1 + " x " + counter2 + " = ");
      System.out.println(counter1 * counter2);
      counter2++;
    }
    counter1++;
  }
Iteration: *Loop-and-a-half* Construct

- In some situations, makes sense to halt loop execution in middle of loop
- This referred to as *loop-and-a-half* control
- No specific loop construct that implements this
- Implemented via a *break* statement
  
  - Syntax: \textit{break};
  - Semantics: Control passes to statement following containing control statement
Iteration: Loop-and-a-half Construct (2)

- Example:

```java
// Loop-and-a-half while loop: Determine if an int is prime.
// Terminate when divisor == number or if is prime

int x, y, div;
boolean isPrime;

System.out.print("Enter an integer: ");
x = keyboard.nextInt();
done = false;
div = 2;
is_prime = true;
while (true) {
    y = x % div;
    if (y == 0) {
        isPrime = false;
        break;
    }
    div += 1;
    if (div == x)
        break;
}
System.out.print(x);
if (isPrime)
    System.out.println(" is a prime number");
else:
    System.out.println(" is not a prime number");
```

- Problems:

1. Potential infinite loop
2. More difficult to debug as now have multiple exit points
Iteration: *Do-while* Loops

- Post test loop
- Syntax:

```
  do stmt while (Boolean expr);
```

- Schematically:

```
do
  stmt
while (Boolean expr);
```

- Semantics:

```
Statement

true
```

- Body must execute at least once
- Example:

```
// Average <limit> integers

int limit, counter, sum, number;

limit = 10;
counter = 0;
sum = 0;
do {
    number = (int) (Math.random() * 100);
counter++;
sum += number;
} while (counter < limit);
System.out.println("The average = " + (float) sum / limit);
```
Iteration: *For* Loops

- Pre test loop
- Works same as *while*
- Intended for count-controlled looping
- Loop control handled automatically in header
- Syntax:

```plaintext
for (initializers; continuation condition; updates) statement
where
  * initializers initializes variables (including the LCV)
    Schematically it looks like
    
    \[ \text{var1} = \text{val1}, \text{var2} = \text{val2}, \ldots \]

  * The *continuation expr* halts the iteration when false
  * updates modifies the values of the variables in initializers
    Schematically it looks like
    
    \[ \text{var1} = \text{expr1}, \text{var2} = \text{expr2}, \ldots \]
```
• Semantics:
  
  – var1 = val1;  //variables in <initializers>
  var2 = val2;
  ...
  while (continuation condition) {
    statement
    var1 = expr1;
    var2 = expr2;
    ...
  }
  – Usually only a single variable (LCV) appears in initializers and updates
  – If continuation condition omitted, defaults to true

• Example: Basic count control

  //Basic count control: Average n integers.

  int limit, counter, sum;

  limit = 10;
  sum = 0;
  for (counter = 0; counter < limit; counter++) {
    number = (int) (Math.random() * 100);
    sum += number;
  }
  System.out.println("The average = " + (float) sum / limit);

• Example: Basic count control

  //Basic count control: Counting backwards by 2’s

  int max, counter;

  max = (int) (Math.random() * 100);
  for (counter = max; counter >= 0; counter = counter - 2)
    System.out.println("Counter = " + counter);

• Example: Multiple variables in header

  int i, j;

  for (i = 0, j = 10; i < 5; i++, j++)
    System.out.println("i = " + i + ", j = " + j);
Iteration: *For* Loops (3)

- **Example: Nested *for* loops**

// Nested for loops: Times tables

```java
int n, counter1, counter2;

n = 5;
for (counter1 = 1; counter1 <= n; counter1++) {
    System.out.println(counter1 + "'s times table: ");
    for (counter2 = 1; counter2 <= 9; counter2++) {
        System.out.print(counter1 + " x " + counter2 + " = ");
        System.out.println(counter1 * counter2);
    }
}
```

- **Example: Indefinite iteration**

// Sentinel-controlled for loop: Average non-negative // integers. Terminate when a negative is input.

```java
int counter, sum = 0, number;

for (counter = 0; ; counter++) {
    number = (int) (Math.random() * 100);
    if ((int)(Math.random() * 5) == 0)
        number = -number;
    if (number < 0)
        break;
    sum += number;
}
if (counter == 0)
    System.out.println("No integers were averaged");
else
    System.out.println("The average = " + (float) sum / counter);
```
• **continue** statement

  – Syntax: `continue;`
  – Semantics: Control jumps to end of loop without exiting loop

  

  ![Diagram of continue statement]

  

• Example: Indefinite iteration

  ```java
  // Count-controlled while loop, indefinite iteration:
  // Average <limit> positive integers

  int limit, counter, sum, number;
  limit = 10;
  counter = 0;
  sum = 0;
  for (counter = 0; counter < 10; ) {
    number = (int) (Math.random() * 100);
    if ((int)(Math.random() * 5) == 0)
      number = -number;
    if (number < 0)
      continue;
    sum += number;
    counter++;
  }
  System.out.println("The average = " + (float) sum / counter);
  ```
Iteration: Loop Invariants

- Loop invariant is a statement that specifies a condition that holds at all times with respect to a loop

- For example:

```plaintext
//Precondition: num >= 1
sum = 0;
counter = 1;
while (counter <= num) {
    sum += counter;
    counter++;
}
//Postcondition: sum = 1 + 2 + ... + (num - 1) + num

- Invariant: sum = 1 + 2 + ... + (counter - 1)
- When loop terminates, counter == num + 1
- Hence,
  
  sum = 1 + 2 + ... + (counter - 1)
  = 1 + 2 + ... + ((num + 1) - 1)
  = 1 + 2 + ... + num

- Invariant usually stated immediately preceding loop condition

- Invariants can be used to help design loops:
  1. Id loop invariant
  2. Determine initializations that insure invariant holds prior to loop entry
  3. Select loop condition that insures postcondition when condition is false
  4. Encode loop body so that invariant holds on every iteration
Example:

```java
//Precondition: 0 <= n <= 12
//Initiallization
while ////invariant
    //condition
{
    //body
}
//Postcondition: factorial = n!
```

1. Id invariant
   - factorial = counter!
     ```java
     //Precondition: 0 <= n <= 12
     //Initiallization
     while //factorial = counter!
         //condition
         {
             //body
         }
     //Postcondition: factorial = n!
     ```

2. Determine initializations
   - counter = 0
   - factorial = 1
   - Above satisfy postcondition of loop if n == 0
     ```java
     //Precondition: 0 <= n <= 12
counter = 0;
factorial = 1;
while //factorial = counter!
    //condition
    {
        //body
    }
//Postcondition: factorial = n!
```
3. Select loop condition

- counter < n
- Thus loop terminates when counter == n

//Precondition: 0 <= n <= 12
counter = 0;
factorial = 1;
while //factorial = counter!
    (counter < n)
{
    //body
}
//Postcondition: factorial = n!

4. Code body

- counter++
- factorial = factorial * counter
- This guarantees that postcondition is met

//Precondition: 0 <= n <= 12
counter = 0;
factorial = 1;
while //factorial = counter!
    (counter < n)
{
    counter++;
    factorial = factorial + counter;
}
//Postcondition: factorial = n!