Repetition gives you the opportunity to repeat a set of statements without having to write those same statements multiple times over. So, it cleans up code, and gives you an added level of control. Realize also, that you will have a number of ways to express your repetition, using the control structures provided by Java. However, not all ways you come up with will be appropriate for any given task. You must select the most appropriate control structure that is required to accomplish the task.

This chapter will show you how.
Java Loop Statements

repeat...repeat...repeat
Repetition simplifies coding
How would you sum up 1 through 100?

\[
\text{sum} = 1 + 2 + 3 + \ldots \ldots 100
\]

\[
\text{sum} = 1 \\
\text{sum} += 2 \\
\ldots \\
\text{sum} += 100
\]

Notice here that adding a large number of values creates a great deal of unnecessary effort, if you have to manually include each term.

Furthermore, this approach does not scale up easily, since you will have to make major changes if you want to add a larger data set, not to mention the additional number of lines it will take.
How about 1 through 100,00?

We need a more efficient way to accomplish this task.

One that can scale up easily!
Repetition of course helps in such instances, and the for statement, is the most suitable for such tasks, i.e. counting. The for is thus best suited if you know how many times you would like to do something, or put another way, the number of iterations is fixed. Once the iteration begins, you know how many times it will repeat.
int sum = 0;

for (int num = 1; num <= 100; num++) {
    sum += num;
}

Is this solution scalable?

int sum = 0;

for (int num = 1; num <= 10000; num++)
{
    sum += num;
}
package sumnumbers;
import java.util.Scanner;

public class SumNumbers {
    public static void main(String[] args) {
        int upperBound;
        int sumOfNumbers = 0;
        Scanner keyboard = new Scanner(System.in);

        System.out.println("I will add numbers from 1 to n.");
        System.out.print("Please enter n: ");
        upperBound = keyboard.nextInt();
    }
}
for (int num = 1; num <= upperBound; num++) {
    sumOfNumbers += num;
} //end for
System.out.println("\n");
System.out.println("The sum of 1 to " +
    upperBound + " is " + sumOfNumbers);
} //end main()
} //end SumNumbers

Output - SumNumbers (run)
run:
I will add numbers from 1 to n.
Please enter n: 100

The sum of 1 to 100 is 5050
BUILD SUCCESSFUL (total time: 6 seconds)
Recap: for

- Use to repeat a fixed number of times
- Excellent choice for counting problems
- Very concise - all steps on one line
Sometimes you may not want to repeat an action at all.

There are times where the number of iterations is not known in advance. For example if you are given a number and asked to count how many times you have to double the number to reach 100, then your answer will only be determined after the value is given, not before hand. So, if the number you are given is 100 let’s say, then the count is 0, but if the number was instead 50, then the count would be 1.

So, to accommodate such variations, Java provides two additional repetition structures for such purpose, the while and do-while. We cover the while first and do-while second.
Summing a list of numbers

First decide what ends the list
Let's say that -1 will signal the end of the input
Now, could the list be empty?
It could, if user enters -1 as the first number
The condition in a while is at the top or before the actual body. This structure of course is what allows for the possibility of zero iteration or multiple iterations. Since the condition is evaluated first, it could evaluate to false and the while body skipped.

This is a major criterion in selecting to use the while loop. Do you foresee that the repetition could occur zero times? If you answer yes, then this is the choice iteration structure to use. Also note that the total number of iteration is unknown before hand.
while Deconstructed

<Sum of numeric list>

```java
int num;
int sumOfNumericList = 0;

// initial step
System.out.print("Enter number of -1 to quit: ");
num = keyboard.nextInt();

// condition step
while(num != -1) {
    sumOfNumericList += num;
    System.out.print("Enter number of -1 to quit: ");
    num = keyboard.nextInt();
}
```

package sumnumericlist;
import java.util.Scanner;

public class SumNumericList {
    public static void main(String[] args) {
        int num;
        int sumOfNumericList = 0;
        Scanner keyboard = new Scanner(System.in);

        System.out.print("Enter a number or -1 to quit: ");
        num = keyboard.nextInt();

        while (num != -1) {
            sumOfNumericList += num;

            System.out.print("Enter a number or -1 to quit: ");
            num = keyboard.nextInt();
        } // end while
Application Deconstructed

<SumNumericList.java>

```java
System.out.println("The sum of the numbers is " + sumOfNumericList);
}
```

```java
//end main()
}
//end SumNumericList
```
Recap: while

- Use to repeat 0 or more times
- Number of iterations is unknown before hand
- Test before execution
And sometimes you may want to repeat an action at least once.
Sum numbers until you reach 100

Add next number if sum is less than 100
Stop when sum becomes greater than or equal to 100
Notice how the condition in this case comes after the body has been executed. This gives rise to the 1-time guarantee, i.e. the body will execute at least one time. After that the execution of the body depends on the condition's evaluation.

Recall, the number of iterations will not be known beforehand.
do-while Deconstructed
<Sum to reach 100>

```java
int num;
int sumToReach = 0;

do {
    System.out.print("Enter number: ");
    num = keyboard.nextInt();
    sumToReach += num;
} while (sumToReach < 100);
```

initial step

update step

condition step
package sumtoreach;
import java.util.Scanner;

public class SumToReach {
    public static void main(String[] args) {
        int num;
        int sumToReach = 0;
        Scanner keyboard = new Scanner(System.in);

        do {
            System.out.print("Enter a number: ");
            num = keyboard.nextInt();

            sumToReach += num;
        } while(sumToReach < 100);
Application Deconstructed
<SumToReach.java>

```java
    System.out.println("The sum is " + sumToReach);
} //end main()
} //end SumToReach
```

Output - SumToReach (run)
- Enter a number: 100
  The sum is 100
- BUILD SUCCESSFUL (total time: 11 seconds)

Output - SumToReach (run)
- Enter a number: 3
- Enter a number: 30
- Enter a number: 45
  The sum is 108
- BUILD SUCCESSFUL (total time: 9 seconds)
Recap: do-while

- Use to repeat 1 or more times
- Number of iterations is unknown before hand
- Test after execution
The for-each Statement

Allows you to cycle through an enumeration or collection, in its entirety
for-each Deconstructed
<Cycle through an enumeration>

```java
class Suit {
    CLUBS, DIAMONDS, HEARTS, SPADES

    for (Suit nextSuit : Suit.values()) {
        System.out.print(nextSuit + " ");
    }
```

package foreachenum;

public class ForEachEnum {
    enum Suit {
        CLUBS,
        DIAMONDS,
        HEARTS,
        SPADES
    }

    public static void main(String[] args) {
        for (Suit nextSuit : Suit.values()) {
            System.out.print(nextSuit + " ");
        } //end for
        System.out.println(" ");
    } //end main()
} //end ForEachEnum
Programming with Loops
In Class Examples

for: Count down
User enters upper bound, lower bound and step down.
Program outputs list.

while: Number reversal
User enters integer to reverse.
Program displays reversed number.
In Class Examples

while: Structure violation (implement in class)

```java
int num = 0;
int count = 0;

while (num != -1) {
    read number
    count number
}
```
Recap: Repetition

- Strive to always choose the most appropriate loop for the task
- Each loop has a purpose and a structure
- Do not violate either one