while loops
Definite loops

- **definite loop**: A loop that executes a known number of times.
  - The *for* loops we have seen so far are definite loops.

We often use language like
- "Repeat these statements $N$ times."
- "For each of these 10 things, …"

Examples:
- Print "hello" 10 times.
- Find all the prime numbers up to an integer $n$. 
Indefinite loops

- **indefinite loop**: A loop where it is not obvious in advance how many times it will execute.

- We often use language like
  - "Keep looping as long as or while this condition is still true."
  - "Don't stop repeating until the following happens."

- **Examples**:
  - Print random numbers until a prime number is printed.
  - Continue looping while the user has not typed "n" to quit.
while loop: A control structure that repeatedly performs a test and executes a group of statements if the test evaluates to true.

while loop, general syntax:
while (<test>) {
   <statement(s)>;
}

Example:
int number = 1;
while (number <= 200) {
   System.out.print(number + " ");
   number *= 2;
}

Output:
1 2 4 8 16 32 64 128
while loop flow chart

- Is the test true?
  - No: (continue)
  - Yes: Execute the controlled statement(s)

- Execute statement after while loop
Example

- Finds and prints a number's first factor other than 1:

```java
Scanner console = new Scanner(System.in);
System.out.print("Type a number: ");
int number = console.nextInt();
int factor = 2;
while (number % factor != 0) {
    factor++;
}
System.out.println("First factor: " + factor);
```

**Sample run:**
Type a number: 91
First factor: 7
for vs. while

- Any for loop of the following form:

```cpp
for (<initialization>; <test>; <update>) {
    <statement(s)>;
}
```

is equivalent to a while loop of the following form:

```cpp
<initialization>;
while (<test>) {
    <statement(s)>;
    <update>;
}
```
for vs. while: Example

What while loop is equivalent to the following for loop?

```java
for (int i = 1; i <= 10; i++) {
    System.out.println(i + " squared = " + (i * i));
}
```

Solution:

```java
int i = 1;
while (i <= 10) {
    System.out.println(i + " squared = " + (i * i));
    i++;
}
```
Exercise

- Write a program that will repeatedly prompt the user to type a number until the user types a non-negative number, then square it.

Example log:
Type a non-negative integer: $-5$
Invalid number, try again: $-1$
Invalid number, try again: $-235$
Invalid number, try again: $-87$
Invalid number, try again: $11$
11 squared is 121
Solution

System.out.print("Type a non-negative integer: ");
int number = console.nextInt();

while (number < 0) {
    System.out.print("Invalid number, try again: ");
    number = console.nextInt();
}

int square = number * number;
System.out.println(number + " squared is " + square);

- Notice that the number variable had to be declared outside the while loop in order to remain in scope.
Exercise: digitSum

- Write a class named `DigitSum` that reads an integer from the user and prints the sum of the digits of that number. You may assume that the number is non-negative.

**Example:**
Enter a nonnegative number: 29107
prints 2+9+1+0+7 or 19

- Hint: Use the `%` operator to extract the last digit of a number. If we do this repeatedly, when should we stop?
import java.util.Scanner;
public class DigitSum {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        int n = keyboard.nextInt();
        int sum = 0;
        while (n > 0) {
            sum += n % 10;   // add last digit to sum
            n = n / 10;      // remove last digit
        }
        System.out.println("sum = " + sum);
    }
}
Cumulative sum
Consider the following code:

```java
Scanner console = new Scanner(System.in);
System.out.print("Type a number: ");
int num1 = console.nextInt();

System.out.print("Type a number: ");
int num2 = console.nextInt();

System.out.print("Type a number: ");
int num3 = console.nextInt();

int sum = num1 + num2 + num3;
System.out.println("The sum is " + sum);
```

Any ideas to improve the code?
Cumulative sum

- The variables `num1`, `num2`, and `num3` are unnecessary:

```java
Scanner console = new Scanner(System.in);
System.out.print("Type a number: ");
int sum = console.nextInt();

System.out.print("Type a number: ");
sum += console.nextInt();

System.out.print("Type a number: ");
sum += console.nextInt();

System.out.println("The sum is " + sum);
```

- **cumulative sum**: A variable that keeps a sum-in-progress and is updated many times until the task of summing is finished.
  - The variable `sum` in the above code represents a cumulative sum.
How could we modify the code to sum 100 numbers?
- Creating 100 copies of the same code would be redundant.

An incorrect solution:

```java
Scanner console = new Scanner(System.in);
for (int i = 1; i <= 100; i++) {
    int sum = 0;
    System.out.print("Type a number: ");
    sum += console.nextInt();
}

System.out.println("The sum is "+sum);
```
Cumulative sum loop

- A correct version:

```java
Scanner console = new Scanner(System.in);
int sum = 0;
for (int i = 1; i <= 100; i++) {
    System.out.print("Type a number: ");
    sum += console.nextInt();
}
System.out.println("The sum is " + sum);
```

- Key idea: Cumulative sum variables must always be declared outside the loops that update them, so that they will continue to live after the loop is finished.
The user's input can control the number of times the loop repeats:

```java
Scanner console = new Scanner(System.in);
System.out.print("How many numbers to add? ");
int count = console.nextInt();
int i = 1;
int sum = 0;
while(i <= count) {
    System.out.print("Type a number: ");
    sum += console.nextInt();
    i++;
}
System.out.println("The sum is "+ sum);
```

Sample Run:
How many numbers to add? 3
Type a number: 2
Type a number: 6
Type a number: 3
The sum is 11
Write a program that reads input of the number of hours an employee has worked and displays the employee's total and average (per day) number of hours.
- The company doesn't pay overtime, so cap any day at 8 hours.

Sample Run:
How many days? 3
Hours? 6
Hours? 12
Hours? 5
Employee's total paid hours = 19
Employee’s average paid hours = 6.3333333
Random numbers
The Random class

- Objects of the Random class generate pseudo-random numbers.
  - Class Random is found in the java.util package.
    ```java
    import java.util.*;
    ```

- The methods of a Random object

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nextInt()</code></td>
<td>returns a random integer</td>
</tr>
<tr>
<td><code>nextInt(max)</code></td>
<td>returns a random integer in the range [0, max) in other words, from 0 to one less than max</td>
</tr>
<tr>
<td><code>nextDouble()</code></td>
<td>returns a random real number in the range [0.0, 1.0)</td>
</tr>
</tbody>
</table>
Generating random numbers

Random rand = new Random();
int randomNum = rand.nextInt(10);
// randomNum has a random value between 0 and 9

- What if we wanted a number from 1 to 10?
  int randomNum = rand.nextInt(10) + 1;

- What if we wanted a number from min to max (i.e. an arbitrary range)?
  int randomNum = rand.nextInt(<size of the range>) + <min>

where <size of the range> equals (<max> - <min> + 1)
Random questions

Given the following declaration, how would you get:

```java
Random rand = new Random();
```

- A random number between 0 and 100 inclusive?
- A random number between 1 and 100 inclusive?
- A random number between 4 and 17 inclusive?
Random solutions

■ Given the following declaration, how would you get:
  Random rand = new Random();

  □ A random number between 0 and 100 inclusive?
    int random1 = rand.nextInt(101);

  □ A random number between 1 and 100 inclusive?
    int random1 = rand.nextInt(100) + 1;

  □ A random number between 4 and 17 inclusive?
    int random1 = rand.nextInt(14) + 4;
Exercise: Die-rolling

- Write a program that simulates the rolling of two six-sided dice until their combined result comes up as 7.

Sample run:
Roll: 2 + 4 = 6
Roll: 3 + 5 = 8
Roll: 5 + 6 = 11
Roll: 1 + 1 = 2
Roll: 4 + 3 = 7
You won after 5 tries!
import java.util.*;

public class Roll {
    public static void main(String[] args) {
        Random rand = new Random();

        int sum = 0;
        int tries = 0;
        while (sum != 7) {
            int roll1 = rand.nextInt(6) + 1;
            int roll2 = rand.nextInt(6) + 1;
            sum = roll1 + roll2;
            System.out.println("Roll: " + roll1 + " + " + roll2 + " = " + sum);
            tries++;
        }

        System.out.println("You won after " + tries + " tries!");
    }
}
Indefinite loop variations
**Variant 1: do/while**

- **do/while loop**: A control structure that executes statements repeatedly while a condition is true, testing the condition at the end of each repetition.

- **do/while loop, general syntax:**
  ```java
do {
    <statement(s)>;
}
while (<test>);
```

- **Example:**
  ```java
  // roll until we get a number other than 3
  Random rand = new Random();
  int die;
  do {
    die = rand.nextInt();
  } while (die == 3);
  ```
do/while loop flow chart

- How does this differ from the **while** loop?
- The controlled `<statement(s)>` will always execute the first time, regardless of whether the `<test>` is true or false.
Variant 2: "Forever" loops

- Loops that go on... forever

```java
while (true) {
  <statement(s)>;
}
```

- If it goes on forever, how do you stop?
**breaking the cycle**

- **break statement**: Immediately exits a loop (**for**, **while**, **do/while**).

  - Example:
    ```java
    while (true) {
        <statement(s)>;
        if (<test>) {
            break;
        }
        <statement(s)>;
    }
    ```

- Why is the **break** statement in an **if** statement?
Revisiting the sentinel problem

- Sentinel loop using `break`:

```java
Scanner console = new Scanner(System.in);
int sum = 0;
while (true) {
    System.out.print("Enter a number (-1 to quit): ");
    int inputNumber = console.nextInt();
    if (inputNumber == -1) { // don't add -1 to sum
        break;
    }
    sum += inputNumber;   // inputNumber != -1 here
}
System.out.println("The total was " + sum);
```