CSEN202 – Introduction to Computer Programming

Topics:

Decisions: Conditional Statements
Iteration: Loops

Prof. Dr. Slim Abdennadher

13.3.2008

Control Structures

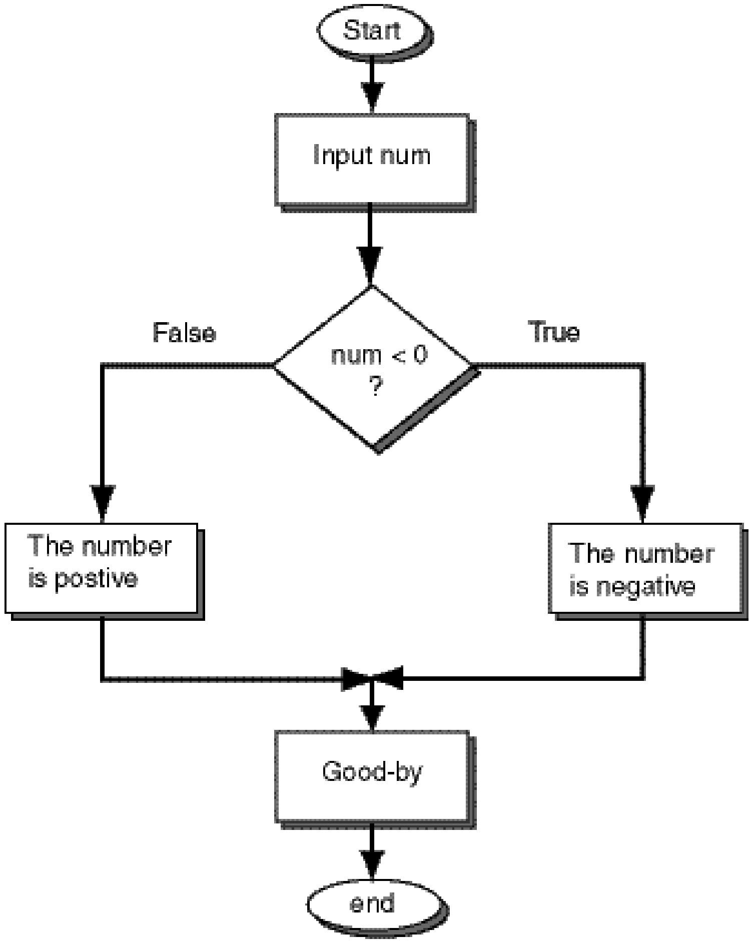
- **Control structures** influence the execution of statements
- **Two basic types**
  - **Conditional statements** are executed only if a condition is met
    - **If-Statement**
    - **Conditional Expression**
    - **Switch-Statement**
  - **Loop statements** are executed more than once
    - **While-Loop**
    - **Do-Loop**
    - **For-loop**

Conditional: If-Statement

```
if ( Expression )
   Statement 1
else
   Statement 2
```

- `Expression` must be of type `boolean`
- The first statement `Statement 1` is executed only if `Expression` evaluates to true.
- Otherwise, the (optional) `else` branch is executed
- `Statement 1` and `Statement 2` can be replaced with a **statement block**, i.e. a sequence of statements
- A **block** is a group of zero or more statements enclosed with curly braces, `{...}`, and can be used anywhere a single statement is allowed.

If-Statement: Example(I)



```
if ( num < 0 )
   System.out.println("The number " + num + " is negative");
else
   System.out.println("The number " + num + " is positive");
System.out.println("Good-bye for now");
```

If-Statement: Example (II)

```
System.out.print(studentsNr);
if (studentsNr == 1) {
   System.out.print("student");
   }
else {
   System.out.print("students");
}
System.out.println(" registered.");
```

- if `studentsNr == 1` ⇒ `1 student registered`
- otherwise ⇒ e.g., `5 students registerd`

Blocks and Braces

- **Do you believe that the following section of a program is correct?**

```
if ( num < 0 )
    System.out.println("The number " + num + " is negative");
else
    System.out.println("The number " + num + " is positive");
    System.out.print  ("positive numbers are greater ");
    System.out.println("or equal to zero ");
System.out.println("Good-bye for now");
```

- **No**. The programmer probably wants the three statements after the else to be part of a false block, but has not used **braces** to show this.

```
if ( num < 0 )
    System.out.println("The number " + num + " is negative");//true-branch
else
{
    System.out.println("The number " + num + " is positive");//false-branch
    System.out.print  ("positive numbers are greater ");     //false-branch
    System.out.println("or equal to zero ");                 //false-branch
}
System.out.println("Good-bye for now");                   //always executed
```

If-Statement and Boolean Expressions

- **Boolean Conjunction**: `&&`

```
// check that there are enough of both ingredients
if ( flour >= 4 && sugar >= 2 )
      System.out.println("Enough for cookies!" );
else
      System.out.println("sorry...." );
```

- **Boolean Disjunction**: `||`

```
// check that at least one qualification is met
if ( cash >= 25000  ||  credit >= 25000 )
      System.out.println("Enough to buy this car!" );
else
      System.out.println("What about a Yugo?" );
```

- **Boolean Negation**: `!`

```
if ( !(  speed > 2000   &&   memory > 512 ) )
  System.out.println("Reject this computer");
else
  System.out.println("Acceptable computer");
```

Nested if-Statements

- `if (condition 1) if (condition 2) Statements 1 else Statements 2`
- The ambigious `else` is called a **dangling else**.
- **Java rule**: An `else` **belongs to the closest** `if`
- To avoid having to think about the pairing of the `else`, it is recommended that you always use a set of braces when a body of an `if` contains another `if`
-

```
if (condition 1)
    if (condition 2)
        Statements 1
    else
        Statements 2
```

```
if (condition 1)
{
    if (condition 2)
      Statement 1
    else
      Statement 2
 }
```

Simplifying Conditional Statements

Refactoring	**Before**	**After**	**Equivalence**

Swap branches	`if (!condition) {`
	` Statements 1`
	`} else {`
	` Statements 2`
	`}`
		`if (condition) {`
		` Statements 2`
		`} else {`
		` Statements 1`
		`}`
			are these always equivalent?
			YES

Remove redundant tests	`if (condition) {`
	` Statements 1`
	`}`
	`if (condition) {`
	` Statements 2`
	`}`
		`if (condition) {`
		` Statements 1`
		` Statements 2`
		`}`
			are these really equivalent under all circumstances?
			NO

Simplifying Conditional Statements

Refactoring	Before	After	Equivalence

Extract to front	`if (condition) {`
	`Statements 1`
	`Statements 2`
	`} else {`
	`Statements 1`
	`Statements 3`
	`}`
		`Statements 1`
		`if (condition) {`
		`Statements 2`
		`} else {`
		`Statements 3`
		`}`
			are these always equivalent?
			NO

Extract to back	`if (condition) {`
	`Statements 1`
	`Statements 3`
	`} else {`
	`Statements 2`
	`Statements 3`
	`}`
		`if (condition) {`
		`Statements 1`
		`} else {`
		`Statements 2`
		`}`
		`Statements 3`
			are these always equivalent?
			YES

Simplifying Conditional Statements

However, care is needed . . .

- Some statements may alter the state which is tested for.

  ⇒ Test is performed with the wrong state

- In this case, **refactoring is not legal**

- **Example:**

```
if (i%7 != 0) {
   i++;
   j = i;
} else {
  i++;
}
```

- **is not equivalent to**

```
i++;
if (i%7 != 0)
   j = i;
```

Conditional Operator

- `condition ? expression1 : expression2`
- **Operand types**:
  - `condition`: **boolean**
  - `expression1` and `expression2`: **can be any type**
- works like `if` statement but for expressions
  - if `condition` is true, the value of the whole expression is `expression1`, otherwise it is `expression2`
- **Example:**

```
if (num < 0)
   x = -num;
else                          equivalent       x = num < 0 ? -num : num;
   x = num;
```

Conditional Operator: Example

```
// print the number of books found

public class Books
{
int num = 4;
   public static void main(String[] args) {
      System.out.println("Number of hits:" + num + " " +
                         ((num == 1) ? "book" : "books")
      );
   }
}
```

- Useful when **duplication** of code or the introduction of a variable can be avoided.

Switch Statement

- Instead of using multiple **if-then-else** branches which test a single value against several constants, the **switch** statement can be used.

-
  ```
  switch (Expression)
  {
    case Literal : statement; break;
    case Literal : statement; break;
    case Literal : statement; break;
    ...
    default: statement;
  }
  ```
- If one case branch matches, all statements after it will be executed ⇒ use **break** to avoid this
- otherwise, the statements after the (optional) `default:` are executed.

Switch Statement: Incorrect Example

```
switch(studentsNr)
{
case 0:
  System.out.print("no one");
case 1:
  System.out.print("1 student");
default:
  System.out.print(studentsNr);
  System.out.print(" students");
}
System.out.println(" registered");
```

Why doesn't this work as expected

Switch Statement: Correct Example

```
switch(studentsNr)
{
case 0:
  System.out.print("no one");
  break;
case 1:
  System.out.print("1 student");
  break;
default:
  System.out.print(studentsNr);
  System.out.print(" students");
}
System.out.println(" registered");
```

Switch Statement: Example

Problem: Display the name of the month, based on the value of month, using the `switch` statement:

```
int month = 8;
switch (month) {
    case 1:  System.out.println("January"); break;
    case 2:  System.out.println("February"); break;
    case 3:  System.out.println("March"); break;
    case 4:  System.out.println("April"); break;
    case 5:  System.out.println("May"); break;
    case 6:  System.out.println("June"); break;
    case 7:  System.out.println("July"); break;
    case 8:  System.out.println("August"); break;
    case 9:  System.out.println("September"); break;
    case 10: System.out.println("October"); break;
    case 11: System.out.println("November"); break;
    case 12: System.out.println("December"); break;
    default: System.out.println("Hey, that's not a valid month!");
}
```

Switch Statement and If-Statements

```
switch(studentsNr)
{
case 0:
 System.out.print("no one");
 break;
case 1:
System.out.print("1 student");
 break;
default:
 System.out.print(studentsNr);
 System.out.print(" students");
}
System.out.println(" registered");
```

```
int studentsNr;
if (studentsNr == 0)
   System.out.print("no one");
 else if (studentsNr == 1)
   System.out.print("1 student");
 else { System.out.print(studentsNr);
    System.out.print(" students")
   }
 System.out.println(" registered");
```

Switch Statement

- **Advantage:** All branches test the same value, namely `studentsNr`
- The **test cases** must be integers or characters. You cannot use a `switch` to branch on floating-point or string values. The following fragement of code is an error:

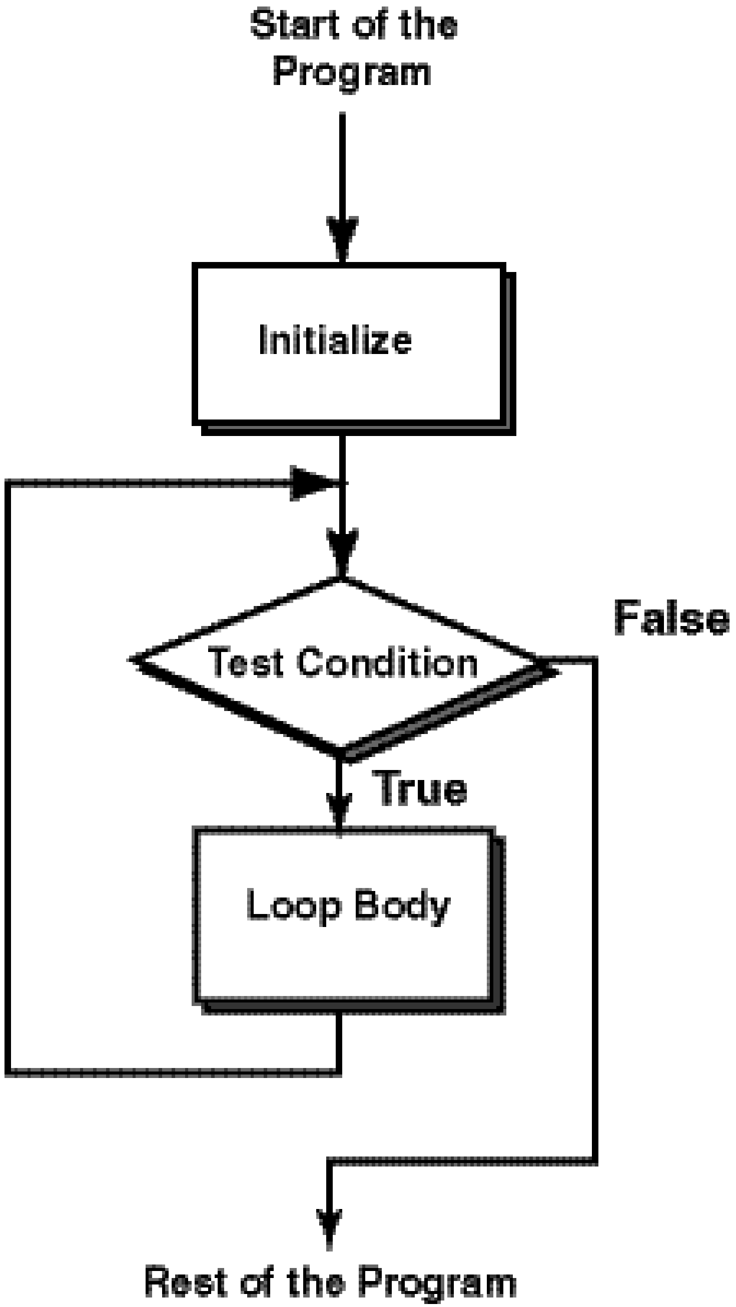
```
switch(studentName) {
case "Nora" : System.out.println("female"); break;
case "Ahmad" : System.out.println("male"); break;
case "Sarah" : System.out.println("female"); break;
....
}
```

Loops: while

```
while (Expression) {
   loop body
}
```

- A **while-loop** executes statements (`loop body`) as long as the loop condition (`Expression`) is true.
- `Expression` must be of type `boolean`
- Before the first and before any following execution of the loop body, the loop condition is evaluated.
- As soon as the condition evaluates to `false`, the loop **terminates**.
- **The loop body may not be executed at all.**

while Flowchart



How to write a while-Loop?

1. **Formulate the test** which tells you whether the loop needs to be run again
   - `count <= 3`
2. **Formulate the actions** for the loop body which take you one step closer to termination

   -
     ```
     {
             System.out.println( "count is:" + count );
             count = count + 1;     // add one to count
     }
     ```
3. In general, **initialization** is required before the loop and some **postprocessing** after the loop
   - `int count = 1`

While-Loop: Example I

```
class LoopExample
{
  public static void main (String[] args )
  {
    int count = 1;              // start count out at one
    while ( count <= 3 )    // loop while count is <= 3
    {
      System.out.println( "count is: " + count );
      count = count + 1;    // add one to count
    }
    System.out.println( "Done with the loop" );
  }
}
```

While-Loop: Example II

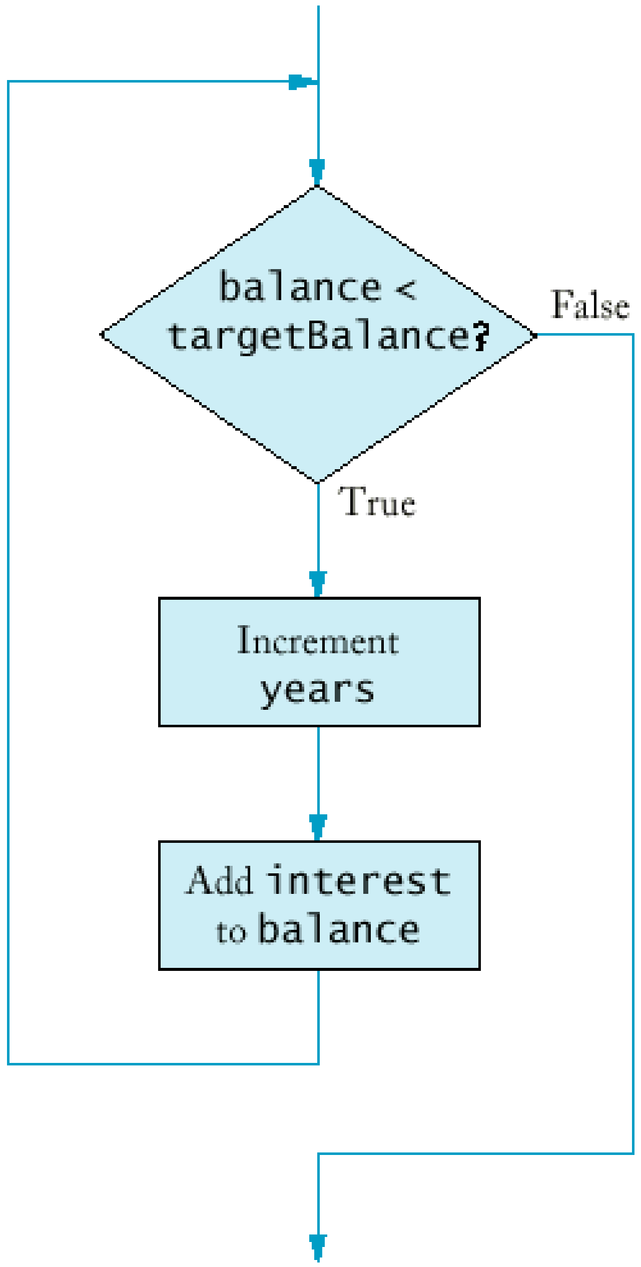
Investment with Compound Interest:

Invest 10000 Euro with 5% interest compounded annually:

Year	**Balance**
0	10 000
1	10 500
2	11025
3	11 576.25
4	12 155.06
5	12 762.82

Question: When will the balance be at least 20000 Euro?

Flowchart for Example II



Java Code for Example II

```
class InvestmentTest
{
  public static void main (String[] args )
  {
     double balance = 10000;
     double rate = 5;
     double targetBalance = 20000;
     int year = 0;
     while (balance < targetBalance)
     {
        year++;
        double interest = balance * rate / 100;
        balance = balance + interest;
     }
     System.out.println("The investment doubled after"+
                                year +"years");
  }
}
```

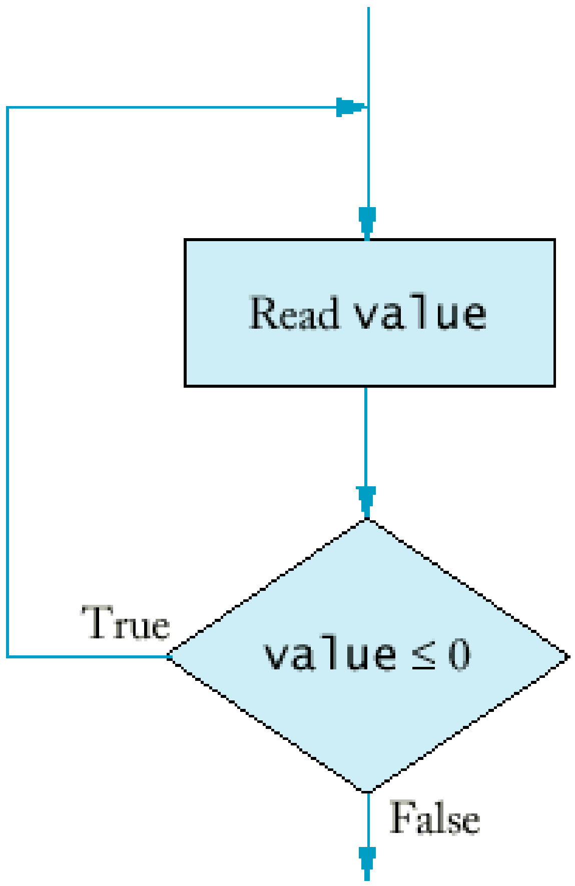
Loops: do-while

```
do {
   loop body
   }
while (Expression)
```

- `Expression` must be of type `boolean`.
- A **do-while** loop checks the condition after execution of the loop body.
- When `Expression` evaluates to `false` – evaluation takes place after execution of the loop body – the loop terminates.
- One can think of the **do-while** loop as a **repeat until** loop where the condition has to be negated.
- **Loop body is executed at least once.**

do-while Flowchart

Example: Validate input (Accept only a positive integer)



do-while Loop: Example

```
class ValidateInput
{
  public static void main (String[] args ) throws IOException
  {
    BufferedReader userin = new BufferedReader
        (new InputStreamReader(System.in));
    String inputData;
    int value;             // data entered by the user
    do
    {
      System.out.println( "Please enter a  positive number: " );
      inputData  = userin.readLine();
      value      = Integer.parseInt( inputData ); //
    }
    while (value <=0);
    System.out.println( "Entered postive number: " + value );
  }
}
```

Loops: while versus do-while (I)

- **In both loops**
  - stops executing body if loop condition is false
  - you must make sure loop condition becomes false by some computations
  - **Infinite loop** means your loop condition is such that it will never turn false, i.e. the exit condition never occurs
- **do-while**
  - body always executed at least once
  - loop condition tested at bottom of loop
- **while**
  - may not execute at all
  - loop condition tested before body; loop condition variables must be set before loop entry

Loops: while versus do-while (II)

while-loops and do-while loops can be transformed to each other

- **do-while ⇒ while:** must copy loop body to front of loop

```
do {
   statement1;
   statement2;
   } while (condition);
```

```
statement1;
statement2;
while (condition) {
   statement1;
   statement2;
}
```

- **while ⇒ do-while:** guard loop with condition

```
while (condition) {
  statement1;
  statement2;
}
```

```
if (condition)
 do {
     statement1;
     statement2;
 } while (condition);
```

Loops: for

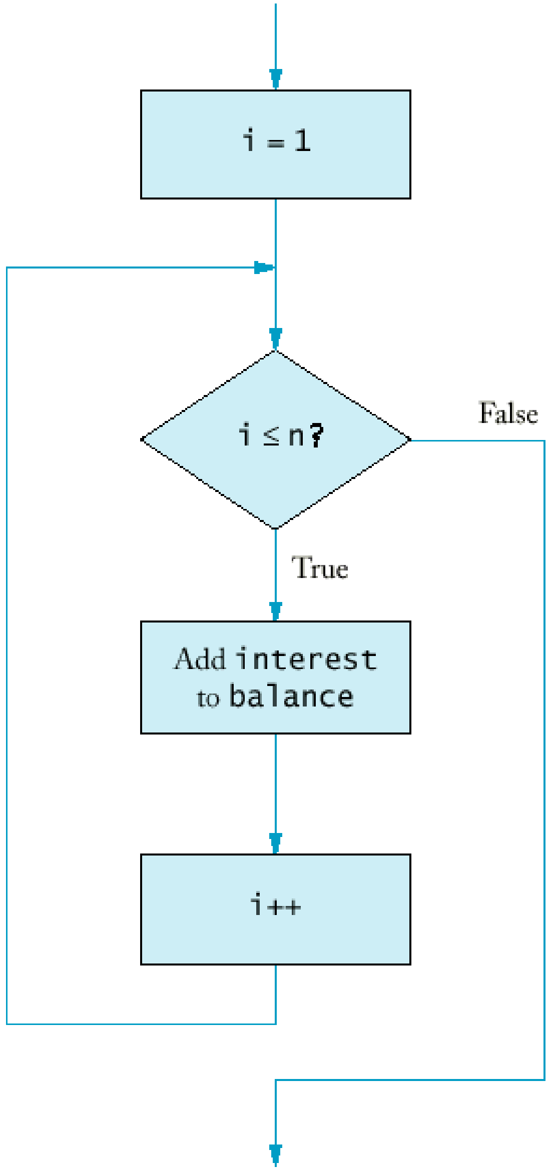
```
for (initialization; condition; update)
  {
    loop body
  }
```

- Most flexible loop construct: **just repeats a statement for a fixed number of times** (counting loop)
- `initialization`: expression for setting initial value of loop counter.
- `condition` must be of type `boolean`
- `update` expression that modifies loop counter
- **Purpose:** To execute an initialization, then keep executing and updating an expression while a condition is true.

A Flowchart for for-Loop

Example: Invest 10000 Euro with 5% interest compounded annually.

Question: What will be the balance after `n` years?



A Java Program with a `for`-loop (I)

```
class Balance
{
  public static void main (String[] args )
  {
     double balance = 10000;
     double rate = 5;
     int year = 15;
     for (int i = 1; i <= year; i++)
     { double interest = balance * rate / 100;
       balance = balance + interest;
     }
     System.out.println("The investment after"+ year +
                              "will be" + balance);
  }
}
```

Strings

- `String` is **NOT** a primitive data type: It is an **Object**.
- Predefined class `String` has special support in Java.
- A string literal is surrounded by double quotes.

  ```
  String hamlet = "to be or not to be"
  ```

- Once a string has been created, we can use the **dot operator** to invoke its methods:

  ```
  hamlet.length();
  ```

- The `String` class has several methods to manipulate strings
  - `char charAt(int index)`: returns the character at the specified index
  - `String toLowerCase()`: Converts all of the characters in this String to lower case.
  - `String replace(char oldChar, char newChar)`: Returns a new string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.

Comparing Strings

- Use `compareTo()` method of the `String` class to perform the comparisons.
- The `compareTo()` method returns different integer values depending on the **lexicographical** (that is, alphabetical) ordering of the Strings.

`s1.compareTo(s2)`	**Return Value**
`s1 < s2`	`< 0`
`s1 equals s2`	`0`
`s1 > s2`	`> 0`

A Java Program with a for-loop (II)

Program to reverse a String

```
public class reverse {
    public static void main(String[] args)  {
    String word = "Slim";
    int max;
    if (word == null) {
        return;
            }
    max = word.length();
    for (int i=max-1; i >=0; i--)
    {
           System.out.print(word.charAt(i));
    }
    System.out.println("");
  }
}
```

Loops: while versus for

- In general a **while** loop has the form

```
initialization
  while (condition) {
     core loop body;
     loop advancement;
  }
```

- This is exactly matched by the **for** loop

```
for (initialization; condition; loop advancement)
  core loop body
```

Choosing the Right Loop

- **for** loop is called **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 a priori when they will end.
- **for** loop is typically used for math-related loops like counting finite sums.
- **while** loop is good for situations where boolean condition could turn false at any time.
- **do while** is used in same type of situation as **while** loop, but when the body of the loop should execute at least once.
- **When more than one type of loop will solve problem, use cleanest, simplest one**