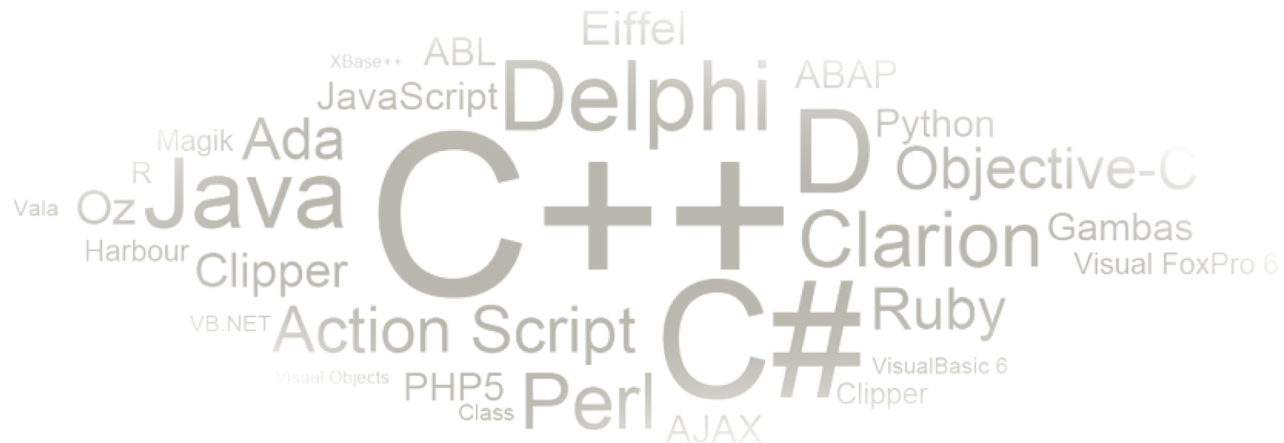


CIS 351-Data Structure-String-Conditional

Jan 21, 2020

Dr. Farzana Rahman

Syracuse University



String

- *String* is actually a class, not a **basic data type**
 - Java string is a sequence of characters
 - *String* variables are objects
- Example of *String* literals:

```
"Hello world"  
"The value of x is"
```

String Concatenation Operator (+)

- Combines *String* literals with other data types for printing

- **Example:**

```
String hello = "Hello";  
String there = "there";  
String greeting = hello + ' ' + there;  
System.out.println( greeting );
```

Output is:

```
Hello there
```

- For string concatenation the Java compiler converts an operand to a *String* whenever the other operand of the + is a *String* object.

Creating Strings

- The default constructor creates an empty string:

```
String s = new String();
```

- Initializing the String:

```
s = "Hello";
```

Or

```
String s = new String("Hello");
```

- Construct a string object by passing another string object:

```
String str2 = new String(str);
```

String Operations

- **charAt(int index)** - returns character at the index position

```
char ch = "abc".charAt(1); // ch = "b"
```

- **equals()** - Compares if two object are equal

```
public boolean equals(Object anObject)
```

- **length()** – returns the length of the string object

```
int len = "ab\\c\t".length();
```

len = 5

String Operations

- **equalsIgnoreCase()**- Compares this String to another String, ignoring case considerations. Two strings are considered equal ignoring case if they are of the same length, and corresponding characters in the two strings are equal ignoring case.

```
public boolean equalsIgnoreCase(String anotherString)
```

- **startsWith()** – Tests if this string starts with the specified prefix.

```
public boolean startsWith(String prefix)
```

```
"Figure".startsWith("Fig"); // true
```

- **endsWith()** - Tests if this string ends with the specified suffix.

```
public boolean endsWith(String suffix)
```

```
"Figure".endsWith("re"); // true
```

- **indexOf(int ch, int fromIndex)**– Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

- **indexOf(String str, int fromIndex)** - Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

Variable Scope

- A variable's scope consists of all code blocks in which it is visible.
- **Java Code block:** One or more lines of code contained between braces {}
- Code blocks can be nested

```
{ Statement1;  
  {  
    Statement2;  
  }  
}
```

- Starts at the declaration statement
- Ends at the end of the block it was declared in

Variable Scope

Examp

```
public class ScopeTest
{
    public static void main(String[] args)
    {
        int num1 = 10;

        if (num1 < 100)
        {
            System.out.print("value of num1 is : " + num1);
        }
    }
}
```

10

```
public class ScopeTest
{
    public static void main(String[] args)
    {
        int num1 = 10;

        if (num1 < 100)
        {
            int num1 = 99;
            System.out.print("value of num1 is : " + num1);
        }
    }
}
```

99

Scope



Relational Operators

Logical operator connect two or more relational expressions into one or reverse the logic of an expression

Relational Operator	Meaning
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to
==	is equal to
!=	is not equal to

Boolean/Logical Operators

- Logical “and” (conjunction)

- true only when both expressions are true

```
(A<B) && (C<D) // (3<4) && (10>5) ---- True
```

- Logical inclusive “or” (disjunction)

- true when either or both expressions are true

```
(A<B) || (C<D) // (3<4) || (10<5) ---- True
```

- Logical “not” (negation)- reverses the truth of a Boolean expression

```
(MINIMUM_WAGE != wages)
```

Boolean/Logical Operator Truth Table

A	B	A && B	A B	!A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Expression evaluation

	Expression	Data Type of result	Result
1	<code>(!(true == false)) (6.0 < 4.0)</code>	boolean	true
2	<code>"CS 139" == "CS " + "139"</code>	boolean	false
3	<code>"Your guess is " + (5 == 5)</code>	String	Your guess is true
4	<code>(20 - 10) > 5 > (6 - 8)</code>	Boolean	True
5	<code>(true false) && (!(false false))</code>	Boolean	True

if Statement

- The *if* statement allows the programmer to make decisions on whether a section of code executes or not.
- Ensures that a statement is executed only when a condition is true

```
if (boolean expression is true)  
    execute next statement.
```

- **Example:**

```
if (age >= 18)  
    System.out.println("Youare eligible to vote.");
```

Decisions (if)

```
if (boolean_expression)  
{  
    statement 1;  
}
```

```
int x = 10;  
if (x > 0)  
{  
    System.out.println("x is positive");  
}
```

Note: else is not mandatory !!

Block if Statements

- Conditionally executed statements can be grouped into a block by using curly braces `{}` to enclose them.
- If curly braces are used to group conditionally executed statements, the if statement is ended by the closing curly brace.

```
if(expression)
```

```
{
```

```
    statement1;
```

```
    statement2;
```

```
}
```

 **Curly brace ends the statement.**

Block if Statements

- Remember that if the curly braces are not used, then only the next statement after the if condition will be executed conditionally.

```
if(expression)
```

```
    statement1;
```



Only this statement is conditionally executed.

```
    statement2;
```

```
    statement3;
```


Decisions (if-else)

```
if (boolean_expression)  
{  
    statement 1;  
}  
else  
{  
    statement 2;  
}
```

```
int x = 10;  
if (x >= 0)  
{  
    System.out.println("x  
is positive");  
}  
else  
{  
    System.out.println("x  
is negative");  
}
```

Note: if and else are exclusive or branch !

if-else Statements

- The if-else statement adds the ability to conditionally execute code based if the expression of the if statement is false.

```
if (expression)
```

```
    statementOrBlockIfTrue;
```

```
else
```

```
    statementOrBlockIfFalse;
```

if-else Matching

This else
matches
with this
if.

This else
matches
with this
if.

```
if (employed == 'y')
{
    if (recentGrad == 'y')
    {
        System.out.println("You qualify for the special
rate.");
    }
    else
    {
        System.out.println("You must be a recent college
graduate");
    }
}
else
{
    System.out.println("You must be employed to qualify.");
}
```

if-else-if Statements

- if-else-if statements can become very complex.
- Imagine the following decision set.

```
if it is very cold, wear a heavy coat,  
else, if it is chilly, wear a light jacket,  
else, if it is windy wear a windbreaker,  
else, if it is hot, wear no jacket.
```

if-else-if Statements

```
if (expression)
    statement or block
else
    if (expression)
        statement or block
        // Put as many else ifs as needed here
    else
        statement or block
```

- Care must be used since else statements match up with the immediately preceding unmatched if statement.

Decisions (if-else if-else)

```
if (boolean_expression)  
{  
    statement 1;  
}  
else if  
{  
    statement 2;  
}  
else  
{  
    statement 3;  
}
```

```
int x = 10;  
if (x > 0)  
{  
    System.out.println("x is  
    positive");  
}  
else if (x < 0)  
{  
    System.out.println("x  
    is negative");  
}  
else  
    System.out.println("x  
    is zero");
```

**Note: testing more than 2
exclusive or conditions**

switch Statement

- Used to accomplish multi-way branching based on the value of an integer selector variable
- Example:

```
switch (numberOfPassengers) {  
    case 0: out.println("The Harley");  
            break;  
    case 1: out.println("The Dune Buggy");  
            break;  
    default: out.println("The Humvee");  
}
```

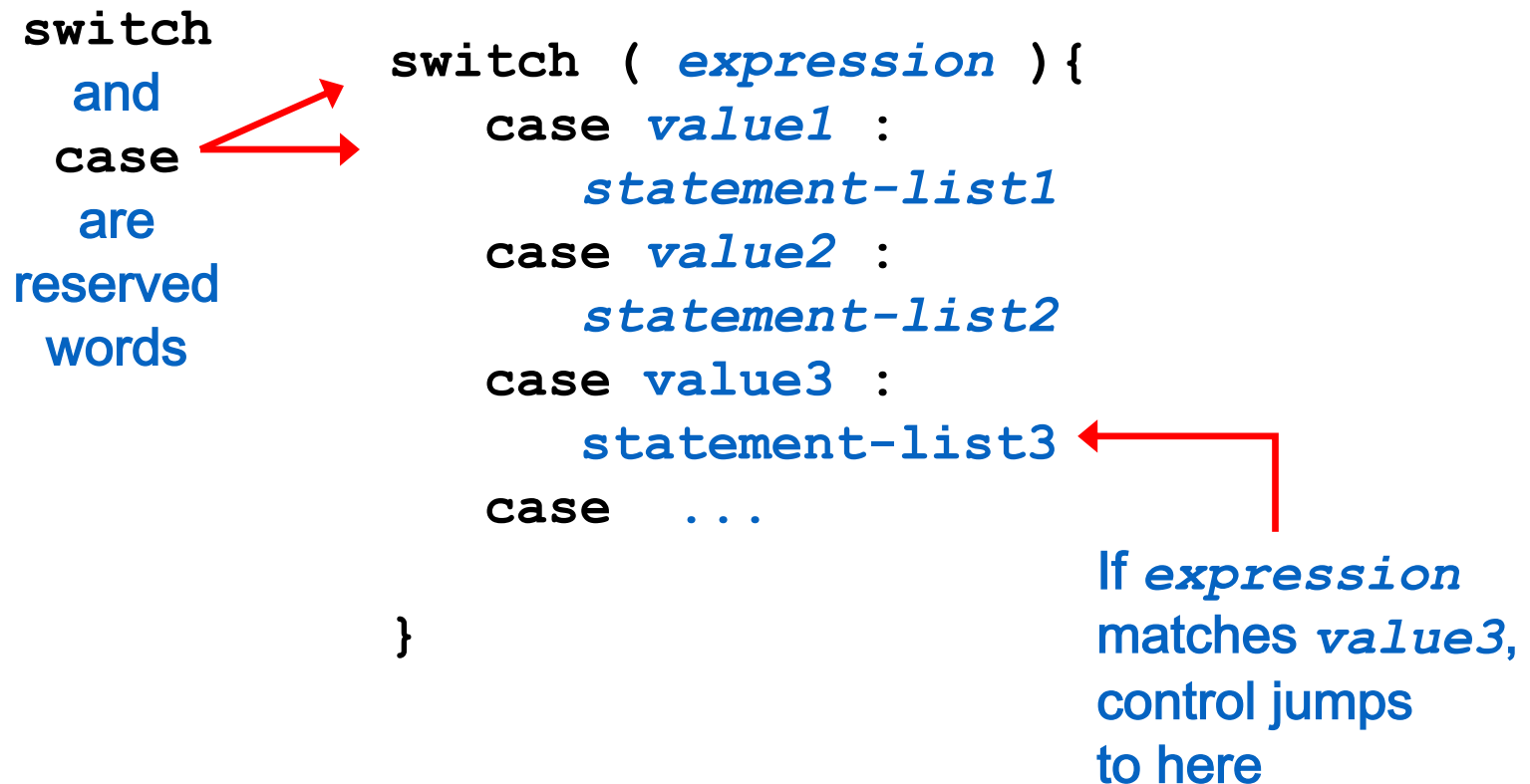
Switch - syntax

- The general syntax of a `switch` statement is:

`switch`
`and`
`case`
`are`
reserved
words

```
switch ( expression ) {  
    case value1 :  
        statement-list1  
    case value2 :  
        statement-list2  
    case value3 :  
        statement-list3  
    case ...  
}
```

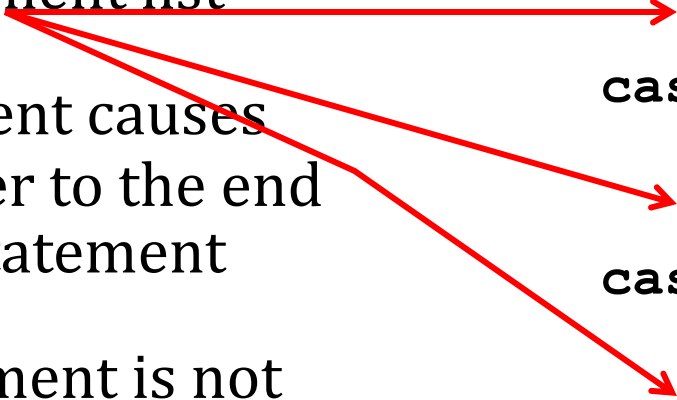
If *expression*
matches *value3*,
control jumps
to here



The Switch Statement

- The *break statement* can be used as the last statement in each case's statement list
- A break statement causes control to transfer to the end of the switch statement
- If a break statement is not used, the flow of control will continue into the next case

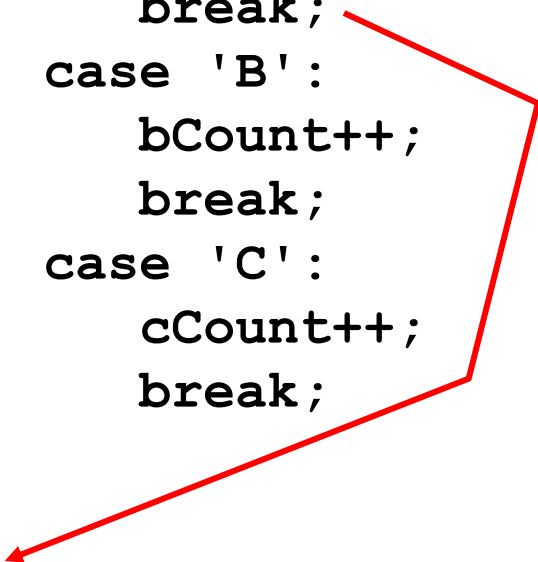
```
switch ( expression ) {  
    case value1 :  
        statement-list1  
        break;  
    case value2 :  
        statement-list2  
        break;  
    case value3 :  
        statement-list3  
        break;  
    case ...  
}
```

A diagram consisting of three red arrows originates from the 'break;' statements in the first three cases of the switch statement. The first arrow points from the 'break;' of the first case to the closing curly brace '}' of the switch statement. The second arrow points from the 'break;' of the second case to the closing curly brace '}' of the switch statement. The third arrow points from the 'break;' of the third case to the closing curly brace '}' of the switch statement. This illustrates that the 'break;' statement causes control to exit the switch statement.


Switch – no breaks!!!

- Another Example:

```
switch (option){  
    case 'A':  
        aCount++;  
        break;  
    case 'B':  
        bCount++;  
        break;  
    case 'C':  
        cCount++;  
        break;  
}
```



```
switch (option){  
    case 'A':  
        aCount++;  
    case 'B':  
        bCount++;  
    case 'C':  
        cCount++;  
}
```



Switch - default

- A `switch` statement can have an optional *default case*
- The default case has no associated value and simply uses the reserved word `default`
- If the default case is present, control will transfer to it if no other case value matches
- If there is no default case, and no other value matches, control falls through to the statement after the switch

Why is break used in switch statements?

- Consider the code fragment below

```
int i = 1;
switch (i) {
    case 0: out.println("0");
    case 1: out.println("1");
    case 2: out.println("2");
    case 3: out.println("3");
}
out.println( );
```

- Without breaks the output is: 123

The switch Statement

- Switch with default case:

```
switch (option) {  
    case 'A':  
        aCount++;  
        break;  
    case 'B':  
        bCount++;  
        break;  
    case 'C':  
        cCount++;  
        break;  
    default:  
        otherCount++;  
        break;  
}
```