07 Character Strings and Booleans

Instructor: Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP112/18/

Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

September 17, 2018

Outline

Character Data Type

- In addition to processing numeric values, you can process *characters* in Java.
- We use the character data type char to represent a single character.
- A character literal is enclosed in single quotation marks.

```
char letter = 'A';
char numChar = '4';
```

- A character is stored in a computer as a binary number.
- Mapping a character to its binary representation is called *encoding*.
- There are different ways to encode a character. How characters are encoded is defined by an *encoding scheme*.
- A character in Java is represented as a 16-bit number, encoded in *Unicode*.
- A 16-bit Unicode character literal is expressed in 4 hexadecimal digits, proceeded by \u.

```
char han = '\u6F22';
```

Escape Characters

- Some characters cannot be written directly in a character or string literal, such as the *newline* (*linefeed*) character, the *tab* character and the *double quotation* character.
- Java uses a special notation to represent special characters, called escape characters.
- An escape character consists of a backslash (\) followed by a character or a character sequence. For example,
  - `\t` is an escape character for the tab character, and
  - an escape character such as `\u03b1` is used to represent a Unicode.
- The following is a list of escape characters.

`\b`	backspace	`\t`	tab	`\n`	linefeed	`\f`	formfeed
`\r`	carriage return	`\\`	backslash	`\"`	double quote	`\'`	single quote

Casting between char and Numeric Types

- A char can be cast into any numeric type, and vice versa.
- When an integer is cast into a char, only the lower 16 bits are used, the rest is ignored.

```
char ch = (char)0XAB0041;   // The lower 16 bits hex code 0041 is assigned to ch.
System.out.println(ch);     // ch is character 'A'.
```

- A floating-point value is first cast into an `int`, then cast into a `char`.

```
char ch = (char)65.25; // Decimal 65 is assigned to ch.
```

- When a `char` is cast into a numeric type, it evaluates to the character's Unicode.

```
int i = (int)'A';
```

- Implicit casting can be used if the result of a casting fits into the target variable.
- All numeric operators can be applied to `char` operands.

```
int i = '2' + '3';                   // addition of 50 and 51
System.out.println("i␣is␣" + i);     // concatenation of "i is " and "101"
```

The *String* Type

- The `char` type stores only one character. To store a string of characters, use *String*.

```
String message = "Welcome␣to␣Java";
```

- A string literal is a sequence of characters, including escape characters, quoted by double quotation marks (").
- Two strings can be concatenated. The plus sign (+) is the concatenation operator if one of the operands is a string.

```
String message = "Welcome␣to␣Java" + "\nJava\u6B61\u8FCE\u4F60";
```

- Since the plus sign is *overloaded*, the meaning of the operation depends on the evaluation order.

```
int i = 100, j = 200;
System.out.println("i␣+␣j␣is␣" + i + j); // What will be the output?
```

✍ If you want to output the result of the addition of *i* and *j*, how to do it?

Selected String Processing Methods

We need to call the methods of the *String* class to process strings, such as to extract a substring from a string. Suppose *s* is a string of value `"␣aBcD␣eFgH␣"`,

- `s.length()` returns the number of characters in *s* — `s.length()` returns 11,
- `s.charAt(i)` returns the character at position *i*, the position starts from 0 — `s.charAt(4)` returns `'D'`, `s.codePointAt(i)` returns the Unicode of the character at position *i*,
- `s.substring(i,j)` returns the substring containing the characters from position *i* to position $j-1$ — `s.substring(3,7)` returns `"cD␣e"`, `s.substring(i)` returns the substring containing all the remaining characters from position *i* — `s.substring(5)` returns `"␣eFgH␣"`,
- `s.trim()` returns the substring without leading and trailing blanks — `s.trim()` returns `"aBcD␣eFgH"`,
- `s.toLowerCase()` returns the string with all uppercase letters converted to lowercase — `s.toLowerCase()` returns `"␣abcd␣efgh␣"`, `s.toUpperCase()` returns the string with all lowercase letters converted to uppercase.

Converting Strings to Numbers

- Numbers are automatically converted to strings in the concatenation operation.

```
int n = 256;
String s = ""+n;  // s becomes "256".
```

- We can also explicitly convert numbers to strings.

```
int n = 256;
String s = Integer.toString(n);         // s becomes "256".
double d = 100.256;
String t = Double.toString(d);          // t becomes "100.256".
```

- On the other hand, we can convert a string back to the number it represents.

```
int n = Integer.parseInt("512");        // n becomes 512.
String s = "1024E-3";
double d = Double.parseDouble(s);       // d becomes 1.024.
```

boolean Data Type

- Often in a program you need to compare two values, such as whether i is greater than j.
- Java provides six comparison operators (also known as *relational operators*) that can be used to compare two values.

Java Operator	Mathematics Symbol	Name	Example (*radius* is 5)	Result
`<`	$<$	less than	*radius* `< 0`	`false`
`>`	$>$	greater than	*radius* `> 0`	`true`
`<=`	$\leq$	less than or equal to	*radius* `<= 0`	`false`
`>=`	$\geq$	greater than or equal to	*radius* `>= 0`	`true`
`==`	$=$	equal to	*radius* `== 0`	`false`
`!=`	$\neq$	not equal to	*radius* `!= 0`	`true`

- The result of the comparison is a `boolean` value: `true` or `false`.

```
boolean b = 1 > 2; // b becomes false.
```

An Example: Addition Quiz

```java
import java.util.Scanner;
public class AdditionQuiz {
    public static void main(String[] args) {
        int n1 = (int)(System.currentTimeMillis() % 10); // get two random numbers
        int n2 = (int)(System.currentTimeMillis() / 7 % 10);

        Scanner scanner = new Scanner(System.in);
        System.out.print("What␣is␣" + n1 + "␣+␣" + n2 + "?␣");
        int answer = scanner.nextInt();
        scanner.close();

        System.out.println(n1 + "␣+␣" + n2 + "␣=␣" + answer
            + "␣is␣" + (n1 + n2 == answer));
    }
}
```

Boolean Operations

A boolean operation takes some boolean values as the operands, and yields a boolean result. Boolean operations are also called logical operations, to combine logical results.

- **Negation (not)**. Negation returns the opposite of its operand. Negation is a unary operation. The negation operator in Java is `!`.

  ```
  boolean b = !(1 < 5); // b becomes false.
  ```

- **Conjunction (and)**. Conjunction returns `true` only if both the operands are `true`. Conjunction is a binary operation. The conjunction operator in Java is `&&`.

  ```
  boolean b = '0' <= c && c <= '9'; // b becomes true if c is a decimal digit.
  ```

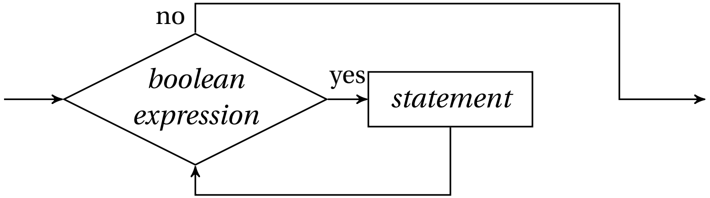
- **Disjunction (or)**. Disjunction returns `false` only if both the operands are `false`. Disjunction is a binary operation. The disjunction operator in Java is `||`.

  ```
  boolean b = 100 == 80 || 70 < 100; // b becomes true.
  ```

- Without parentheses, negations are evaluated first, then conjunctions, last disjunctions.

Condition-Controlled Loops: `while` Statement

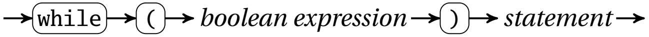
- A loop is a block of statements which is written once but may be repeated several times in succession.
- A `while` loop consists of two parts: a boolean expression as the *loop condition*, and a block of statement as the *loop body*.
- The loop condition is evaluated first, if true, the loop body is executed, then the control goes back to the loop condition; otherwise the loop body is skip entirely.



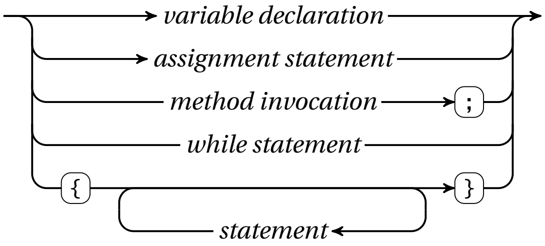
```
while ( n >= 13 ) { n -= 13; ++q; }
```

Syntax Diagram of `while` Statement

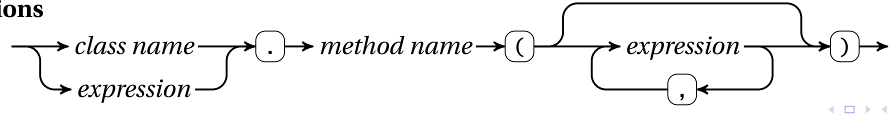
`while` statement



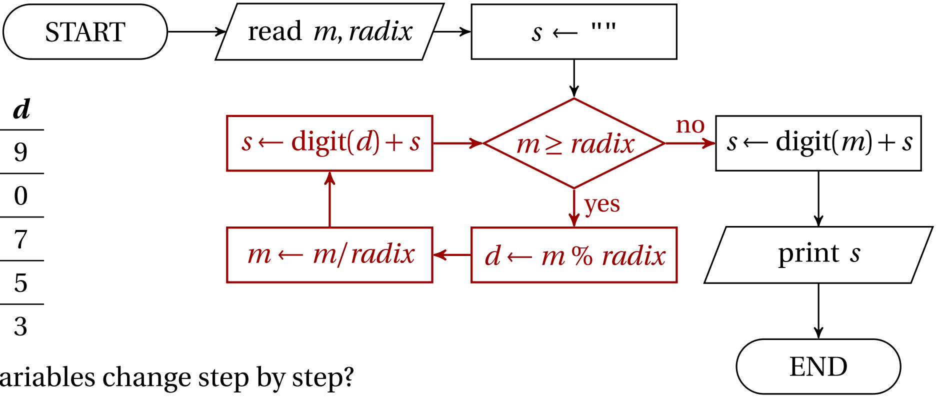
Statements



Method invocations



Print a Number in Radix *R*



START
read $m, radix$
$s \leftarrow$ ""
$m \geq radix$
no
yes
$s \leftarrow \text{digit}(m) + s$
print s
END
$d \leftarrow m \,\%\, radix$
$m \leftarrow m / radix$
$s \leftarrow \text{digit}(d) + s$

m	$radix$	d
35709	10	9
3570	10	0
357	10	7
35	10	5
3	10	3

- How do the variables change step by step?
- In what condition should we exit the loop?
- Will the condition be met?
- How can we collect the digits obtained during the loop in a *correct* order?

Reading Homework

Textbook

- Section 3.1 – 3.2, 3.10.
- 4.1 – 4.6.
- 5.1 – 5.2.

Internet

- String (`http://en.wikipedia.org/wiki/String_(computer_science)`).
- Boolean algebra (`http://en.wikipedia.org/wiki/Boolean_algebra`).
- While loop (`http://en.wikipedia.org/wiki/While_loop`).

Self-test

- 2.58 – 2.72 (`http://tiger.armstrong.edu/selftest/selftest9e?chapter=2`).
- 3.1 – 3.5, 3.38 – 3.44 (`http://tiger.armstrong.edu/selftest/selftest9e?chapter=3`).
- 4.1 – 4.5 (`http://tiger.armstrong.edu/selftest/selftest9e?chapter=4`).