ความรู้เพิ่มเติม : Reserve Keyword User Input, Regular Expression File Handling

ENGCE174 การเขียนโปรแกรมเชิงวัตถุ (Object-oriented programming)

อ.กิตตินันท์ น้อยมณี (mr.kittinan@rmutl.ac.th)

Java Reserved Keywords

Java has a set of keywords that are reserved words that cannot be used as variables, methods, classes, or any other identifiers:

abstract	A non-access modifier. Used for classes and methods: An abstract class cannot be used to create objects (to access it, it must be inherited from another class). An abstract method can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from)
assert	For debugging
boolean	A data type that can only store true and false values
break	Breaks out of a loop or a switch block
byte	A data type that can store whole numbers from -128 and 127
case	Marks a block of code in switch statements
catch	Catches exceptions generated by try statements
char	A data type that is used to store a single character

Java Reserved Keywords

class	Defines a class
continue	Continues to the next iteration of a loop
const	Defines a constant. Not in use - use final instead
default	Specifies the default block of code in a switch statement
do	Used together with while to create a do-while loop
double	A data type that can store whole numbers from 1.7e−308 to 1.7e+308
else	Used in conditional statements
enum	Declares an enumerated (unchangeable) type
exports	Exports a package with a module. New in Java 9
extends	Extends a class (indicates that a class is inherited from another class)
final	A non-access modifier used for classes, attributes and methods, which makes them non-changeable (impossible to inherit or override)
finally	Used with exceptions, a block of code that will be executed no matter if there is an exception or not

Java Reserved Keywords

float	A data type that can store whole numbers from 3.4e−038 to 3.4e+038
for	Create a for loop
goto	Not in use, and has no function
if	Makes a conditional statement
implements	Implements an interface
import	Used to import a package, class or interface
instanceof	Checks whether an object is an instance of a specific class or an interface
int	A data type that can store whole numbers from -2147483648 to 2147483647
interface	Used to declare a special type of class that only contains abstract methods
long	A data type that can store whole numbers from -9223372036854775808 to 9223372036854775808
module	Declares a module. New in Java 9
native	Specifies that a method is not implemented in the same Java source file (but in another language)

Java Reserved Keywords

new	Creates new objects
package	Declares a package
private	An access modifier used for attributes, methods and constructors, making them only accessible within the declared class
protected	An access modifier used for attributes, methods and constructors, making them accessible in the same package and subclasses
public	An access modifier used for classes, attributes, methods and constructors, making them accessible by any other class
requires	Specifies required libraries inside a module. New in Java 9
return	Finished the execution of a method, and can be used to return a value from a method
short	A data type that can store whole numbers from -32768 to 32767
static	A non-access modifier used for methods and attributes. Static methods/attributes can be accessed without creating an object of a class
strictfp	Restrict the precision and rounding of floating point calculations
super	Refers to superclass (parent) objects
switch	Selects one of many code blocks to be executed

Java Reserved Keywords

synchronized	A non-access modifier, which specifies that methods can only be accessed by one thread at a time
this	Refers to the current object in a method or constructor
throw	Creates a custom error
throws	Indicates what exceptions may be thrown by a method
transient	A non-accesss modifier, which specifies that an attribute is not part of an object's persistent state
try	Creates a try...catch statement
var	Declares a variable. New in Java 10
void	Specifies that a method should not have a return value
volatile	Indicates that an attribute is not cached thread-locally, and is always read from the "main memory"
while	Creates a while loop

Java User Input

The `Scanner` class is used to get user input, and it is found in the `java.util` package.

To use the `Scanner` class, create an object of the class and use any of the available methods found in the `Scanner` class documentation. In our example, we will use the `nextLine()` method, which is used to read Strings:

```
import java.util.Scanner;  // Import the Scanner class

class Main {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);  // Create a Scanner object
    System.out.println("Enter username");

    String userName = myObj.nextLine();  // Read user input
    System.out.println("Username is: " + userName);  // Output user input
  }
}
```

Input Types

In the example above, we used the `nextLine()` method, which is used to read Strings. To read other types, look at the table below:

Method	Description
`nextBoolean()`	Reads a `boolean` value from the user
`nextByte()`	Reads a `byte` value from the user
`nextDouble()`	Reads a `double` value from the user
`nextFloat()`	Reads a `float` value from the user
`nextInt()`	Reads a `int` value from the user
`nextLine()`	Reads a `String` value from the user
`nextLong()`	Reads a `long` value from the user
`nextShort()`	Reads a `short` value from the user

Input Types

In the example below, we use different methods to read data of various types:

Note: If you enter wrong input (e.g. text in a numerical input), you will get an exception/error message (like "InputMismatchException").

You can read more about exceptions and how to handle errors in the Exceptions chapter.

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);

    System.out.println("Enter name, age and salary:");

    // String input
    String name = myObj.nextLine();

    // Numerical input
    int age = myObj.nextInt();
    double salary = myObj.nextDouble();

    // Output input by user
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
    System.out.println("Salary: " + salary);
  }
}
```

What is a Regular Expression?

A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.

A regular expression can be a single character, or a more complicated pattern.

Regular expressions can be used to perform all types of text search and text replace operations.

Java does not have a built-in Regular Expression class, but we can import the `java.util.regex` package to work with regular expressions. The package includes the following classes:

- `Pattern` Class - Defines a pattern (to be used in a search)
- `Matcher` Class - Used to search for the pattern
- `PatternSyntaxException` Class - Indicates syntax error in a regular expression pattern

What is a Regular Expression?

Find out if there are any occurrences of the word "w3schools" in a sentence:

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("w3schools", Pattern.CASE_INSENSITIVE);
    Matcher matcher = pattern.matcher("Visit W3Schools!");
    boolean matchFound = matcher.find();
    if(matchFound) {
      System.out.println("Match found");
    } else {
      System.out.println("Match not found");
    }
  }
}
// Outputs Match found
```

What is a Regular Expression?

Example Explained

In this example, The word "w3schools" is being searched for in a sentence.

First, the pattern is created using the `Pattern.compile()` method. The first parameter indicates which pattern is being searched for and the second parameter has a flag to indicates that the search should be case-insensitive. The second parameter is optional.

The `matcher()` method is used to search for the pattern in a string. It returns a Matcher object which contains information about the search that was performed.

The `find()` method returns true if the pattern was found in the string and false if it was not found.

Flags

Flags in the `compile()` method change how the search is performed. Here are a few of them:

- `Pattern.CASE_INSENSITIVE` - The case of letters will be ignored when performing a search.
- `Pattern.LITERAL` - Special characters in the pattern will not have any special meaning and will be treated as ordinary characters when performing a search.
- `Pattern.UNICODE_CASE` - Use it together with the `CASE_INSENSITIVE` flag to also ignore the case of letters outside of the English alphabet

Regular Expression Patterns

The first parameter of the `Pattern.compile()` method is the pattern. It describes what is being searched for.

Brackets are used to find a range of characters:

Expression	Description
[abc]	Find one character from the options between the brackets
[^abc]	Find one character NOT between the brackets
[0-9]	Find one character from the range 0 to 9

Metacharacters

Metacharacters are characters with a special meaning:

Metacharacter	Description
|	Find a match for any one of the patterns separated by | as in: cat|dog|fish
.	Find just one instance of any character
^	Finds a match as the beginning of a string as in: ^Hello
$	Finds a match at the end of the string as in: World$
\d	Find a digit
\s	Find a whitespace character
\b	Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b
\uxxxx	Find the Unicode character specified by the hexadecimal number xxxx

Quantifiers

Quantifiers define quantities:

Quantifier	Description
n+	Matches any string that contains at least one *n*
n*	Matches any string that contains zero or more occurrences of *n*
n?	Matches any string that contains zero or one occurrences of *n*
n{x}	Matches any string that contains a sequence of *X* *n*'s
n{x,y}	Matches any string that contains a sequence of X to Y *n*'s
n{x,}	Matches any string that contains a sequence of at least X *n*'s

Java Files

File handling is an important part of any application.

Java has several methods for creating, reading, updating, and deleting files.

Java File Handling

The `File` class from the `java.io` package, allows us to work with files.

To use the `File` class, create an object of the class, and specify the filename or directory name:

```
import java.io.File;  // Import the File class

File myObj = new File("filename.txt"); // Specify the filename
```

Java File

Handling

The `File` class has many useful methods for creating and getting information about files. For example:

Method	Type	Description
`canRead()`	Boolean	Tests whether the file is readable or not
`canWrite()`	Boolean	Tests whether the file is writable or not
`createNewFile()`	Boolean	Creates an empty file
`delete()`	Boolean	Deletes a file
`exists()`	Boolean	Tests whether the file exists
`getName()`	String	Returns the name of the file
`getAbsolutePath()`	String	Returns the absolute pathname of the file
`length()`	Long	Returns the size of the file in bytes
`list()`	String[]	Returns an array of the files in the directory
`mkdir()`	Boolean	Creates a directory

Create a File

To create a file in Java, you can use the `createNewFile()` method. This method returns a boolean value: `true` if the file was successfully created, and `false` if the file already exists. Note that the method is enclosed in a `try...catch` block. This is necessary because it throws an `IOException` if an error occurs (if the file cannot be created for some reason):

```java
import java.io.File;  // Import the File class
import java.io.IOException;  // Import the IOException class to handle errors

public class CreateFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      if (myObj.createNewFile()) {
        System.out.println("File created: " + myObj.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

```
File created: filename.txt
```

Create a File

To create a file in a specific directory (requires permission), specify the path of the file and use double backslashes to escape the "\" character (for Windows). On Mac and Linux you can just write the path, like: /Users/name/filename.txt

```
File myObj = new File("C:\\Users\\MyName\\filename.txt");
```

Create a File

```java
import java.io.File;
import java.io.IOException;

public class CreateFileDir {
  public static void main(String[] args) {
    try {
      File myObj = new File("C:\\Users\\MyName\\filename.txt");
      if (myObj.createNewFile()) {
        System.out.println("File created: " + myObj.getName());
        System.out.println("Absolute path: " + myObj.getAbsolutePath());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

```
File created: filename.txt
Absolute path: C:\Users\MyName\filename.txt
```

Write To a File

In the following example, we use the `FileWriter` class together with its `write()` method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the `close()` method:

```java
import java.io.FileWriter;   // Import the FileWriter class
import java.io.IOException;  // Import the IOException class to handle errors

public class WriteToFile {
  public static void main(String[] args) {
    try {
      FileWriter myWriter = new FileWriter("filename.txt");
      myWriter.write("Files in Java might be tricky, but it is fun enough!");
      myWriter.close();
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

```
Successfully wrote to the file.
```

Read a File

In the previous chapter, you learned how to create and write to a file.

In the following example, we use the `Scanner` class to read the contents of the text file we created in the previous chapter:

```
import java.io.File;  // Import the File class
import java.io.FileNotFoundException;  // Import this class to handle errors
import java.util.Scanner; // Import the Scanner class to read text files

public class ReadFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      Scanner myReader = new Scanner(myObj);
      while (myReader.hasNextLine()) {
        String data = myReader.nextLine();
        System.out.println(data);
      }
      myReader.close();
    } catch (FileNotFoundException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

```
Files in Java might be tricky, but it is fun enough!
```

Get File Information

To get more information about a file, use any of the `File` methods:

Note: There are many available classes in the Java API that can be used to read and write files in Java: `FileReader`, `BufferedReader`, `Files`, `Scanner`, `FileInputStream`, `FileWriter`, `BufferedWriter`, `FileOutputStream`, etc. Which one to use depends on the Java version you're working with and whether you need to read bytes or characters, and the size of the file/lines etc.

Tip: To delete a file, read our Java Delete Files chapter.

```
import java.io.File;  // Import the File class

public class GetFileInfo {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.exists()) {
      System.out.println("File name: " + myObj.getName());
      System.out.println("Absolute path: " + myObj.getAbsolutePath());
      System.out.println("Writeable: " + myObj.canWrite());
      System.out.println("Readable " + myObj.canRead());
      System.out.println("File size in bytes " + myObj.length());
    } else {
      System.out.println("The file does not exist.");
    }
  }
}
```

```
File name: filename.txt
Absolute path: C:\Users\MyName\filename.txt
Writeable: true
Readable: true
File size in bytes: 0
```

Delete a File

To delete a file in Java, use the `delete()` method:

```
import java.io.File;  // Import the File class

public class DeleteFile {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.delete()) {
      System.out.println("Deleted the file: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the file.");
    }
  }
}
```

```
Deleted the file: filename.txt
```

Delete a Folder

You can also delete a folder. However, it must be empty:

```java
import java.io.File;

public class DeleteFolder {
  public static void main(String[] args) {
    File myObj = new File("C:\\Users\\MyName\\Test");
    if (myObj.delete()) {
      System.out.println("Deleted the folder: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the folder.");
    }
  }
}
```

```
Deleted the folder: Test
```