Files

Table of Contents

Input: Command line arguments

```java
void main(String[] args)
```

- `args`: variable storing command line arguments as array of `String`s
- Guide to configuring IntelliJ for command-line args

Write a program that creates a `Person` object from 3 command line arguments (age, height, name), and then outputs the object as a string

```java
class Program {
    static void main(String[] args) {
        int age = Integer.parseInt(args[0]);
        double height = Double.parseDouble(args[1]);
        String name = args[2];
        Person person = new Person(age, height, name);
        System.out.println(person);
    }
}
```

Input: Scanner

- Documentation

- `import java.util.Scanner`
- create scanner: `Scanner scanner = new Scanner(System.in);`
- `System.in`: object representing standard input stream
- only ever create **one** `Scanner` for each program
- `nextLine()`: reads a single line of text up until a newline character
  - this is the only method that **eats** newline characters
  - in some instances you need to follow `nextXXX` with `nextLine` if input is on multiple lines
- `next()`: returns next complete token from the scanner (i.e. up to next delimiter)

Read in various data types

Scanner reads in a single value matching the method name

```
boolean b = scanner.nextBoolean();
int i = scanner.nextInt();
double d = scanner.nextDouble();
```

- `Scanner` does not automatically downcast (e.g. `float` to `int`)
- when using `nextXXX` it is up to programmer to ensure input matches what code expects
- `hasNext()`: returns `true` if there is any input to be read
- `hasNextXXX()`: returns `true` if the next *token* matches *XXX*

Scanner example

Write a program that accepts three user inputs, creates an IMDB entry for an `Actor` and prints the object: - `String name`: name of character - `double rating` - `String review`

```
import java.util.Scanner;

public class Program {
    public static void main(String[] args) {
        String name = scanner.nextLine();
        double rating = scanner.nextDouble();
        scanner.nextLine();
        String review = scanner.nextLine();
        Actor actor = new Actor(name, rating, review);
        System.out.println(actor);
    }
}

public class Actor {
    public static final int MAX_RATING = 10;
```

```
16      public String name;
17      public double rating;
18      public String review;
19
20      public Actor(String name, double rating, String review) {
21          this.name = name;
22          this.rating = rating;
23          this.review = review;
24      }
25
26      public String toString() {
27          return String.format("You gave %s a rating of %f%d\n",
28              name, rating, MAX_RATING) +
29              String.format("Your review: '%s'", review);
30      }
31  }
```

Boilerplate: reading plaintext with `Scanner`

- can also use `Scanner`, allowing you to parse lines into tokens, read as integers, …

```
1  import java.util.scanner;
2  try (Scanner scanner = new Scanner(new FileReader("test.txt"))) {
3      while (scanner.hasNextLine()) {
4          // do stuff
5      }
6  }
```

Reading files

Boilerplate for reading plaintext files

```
1  import java.io.FileReader;      // low level file for simple character
       reading
2  import java.io.BufferedReader;  // higher level file object that reads
       Strings
3  import java.io.IOException;     // handle exceptions
4
5  public class ReadFile {
6      public static void main(String[] args) {
7          try (BufferedReader br = new BufferedReader(new FileReader("
               test.txt"))) {
8              String text;
9              while ((text = br.readLine()) != null) {
10                 // do stuff with text
11             }
```

```
12          } catch (Exception e) {
13              e.printStackTrace();
14          }
15      }
16  }
```

- `BufferedReader` is a wrapper that encompasses `FileReader`, allowing you to manipulate files
  - well suited to large files and fast processing
- can use `Scanner` to read files, allowing you to parse text as you read it
  - smaller buffer size
  - slower than `BufferedReader`
  - works well for small files

Reading CSV files

```
1  String[] columns = text.split(",");
```

Writing files

Boilerplate for writing plaintext files

```
1  import java.io.FileWriter;
2  import java.io.PrintWriter;
3  import java.io.IOException;
4
5  public class Program {
6      public static void main(String[] args) {
7          try (PrintWriter pw = new FileWriter("test.txt")) {
8              pw.println("Hello World");
9              pw.format("Test a %s and an integer %d", "string", 10);
10         } catch (Exception e) {
11             e.printStackTrace();
12         }
13     }
14 }
```