Building Java Applications

Prof. Michele Loreti

Programmazione Avanzata
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Building Java applications. . .

The basic tool we can use for building Java applications is javac.

Building Java applications. . .

The basic tool we can use for building Java applications is javac.

This is the default Java compiler distributed with the Java SDK .

Building Java applications. . .

The basic tool we can use for building Java applications is javac.

This is the default Java compiler distributed with the Java SDK .

The javac tool reads class and interface definitions, written in the Java programming language, and compiles them into bytecode class files.

Building Java applications. . .

We can specify options and input files when javac is executed:

```
javac [options] [source-files]
```

Building Java applications...

We can specify options and input files when javac is executed:

```
javac [options] [source-files]
```

Input files can be specified as arguments to the javac command or kept in a argument files (see later).

Building Java applications. . .

We can specify options and input files when javac is executed:

```
javac [options] [source-files]
```

Input files can be specified as arguments to the javac command or kept in a argument files (see later).

Source files should be arranged in a directory hierarchy corresponding to the fully qualified names of the types they contain.

Example. . .

```
package it.unicam.cs.pa.battleship19;

public interface Player {

   ...

}
```

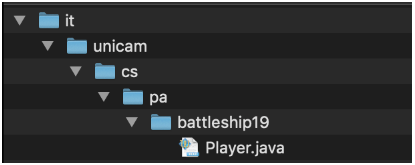
Example. . .

```
package it.unicam.cs.pa.battleship19;

public interface Player {

    ...

}
```



Structuring you Java project

The specific structure of a Java project depends on the tool you use to develop your application.

Structuring you Java project

The specific structure of a Java project depends on the tool you use to develop your application.

Basic rules:

- different folders for *sources* and *generated classes*;
- a separated folder for tests.

Structuring you Java project

The specific structure of a Java project depends on the tool you use to develop your application.

Basic rules:

- different folders for *sources* and *generated classes*;
- a separated folder for tests.

Eclipse Java Projects respect the rules above.

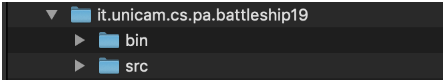
Structuring you Java project

The specific structure of a Java project depends on the tool you use to develop your application.

Basic rules:

- different folders for *sources* and *generated classes*;
- a separated folder for tests.

Eclipse Java Projects respect the rules above.



Standard options. . .

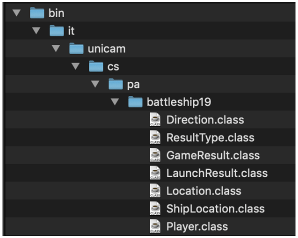
One of the most standard options of javac is `-d` that can be used to specify the destination directory for the generated classes.

```
javac -d ../bin/ it/unicam/cs/pa/battleship19/Player.java
```

Standard options. . .

One of the most standard options of javac is `-d` that can be used to specify the destination directory for the generated classes.

```
javac -d ../bin/ it/unicam/cs/pa/battleship19/Player.java
```

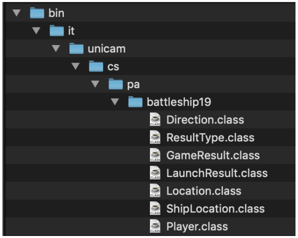


```
bin
  it
    unicam
      cs
        pa
          battleship19
            Direction.class
            ResultType.class
            GameResult.class
            LaunchResult.class
            Location.class
            ShipLocation.class
            Player.class
```

Standard options. . .

One of the most standard options of javac is `-d` that can be used to specify the destination directory for the generated classes.

```
javac -d ../bin/ it/unicam/cs/pa/battleship19/Player.java
```



```
▼ bin
    ▼ it
        ▼ unicam
            ▼ cs
                ▼ pa
                    ▼ battleship19
                          Direction.class
                          ResultType.class
                          GameResult.class
                          LaunchResult.class
                          Location.class
                          ShipLocation.class
                          Player.class
```

To see the details of the building process, the option `-verbose` can be used.

Standard options. . .

Standard options. . .

- `-cp` (or `-classpath`, `class-path`): specifies where types required to compile our source files can be found. . .

Standard options. . .

- `-cp` (or `-classpath`, `class-path`): specifies where types required to compile our source files can be found. . .
  - . . . if this option is missing and the `CLASSPATH` environment variable isnt set, the current working directory is used instead.

Standard options. . .

- `-cp` (or `-classpath`, `class-path`): specifies where types required to compile our source files can be found. . .
  - . . . if this option is missing and the `CLASSPATH` environment variable isnt set, the current working directory is used instead.
- `-p` (or `module-path`): indicates the location of necessary application modules. . .

Standard options. . .

- `-cp` (or `-classpath`, `class-path`): specifies where types required to compile our source files can be found. . .
  - . . . if this option is missing and the `CLASSPATH` environment variable isnt set, the current working directory is used instead.
- `-p` (or `module-path`): indicates the location of necessary application modules. . .
  - . . . this option is only applicable to Java 9 and above.

Standard options...

- `-cp` (or `-classpath`, `class-path`): specifies where types required to compile our source files can be found...
  - ... if this option is missing and the `CLASSPATH` environment variable isnt set, the current working directory is used instead.
- `-p` (or `module-path`): indicates the location of necessary application modules...
  - ... this option is only applicable to Java 9 and above.
- `-g`: Generate all debugging info.

Standard options. . .

- `-cp` (or `-classpath`, `class-path`): specifies where types required to compile our source files can be found. . .
  - . . . if this option is missing and the `CLASSPATH` environment variable isnt set, the current working directory is used instead.
- `-p` (or `module-path`): indicates the location of necessary application modules. . .
  - . . . this option is only applicable to Java 9 and above.
- `-g`: Generate all debugging info.
- `--help`: can be used to obtain a full list of the available options.

Compilation arguments and file...

Instead of passing arguments directly to the javac tool, we can store them in argument files. The names of those files, prefixed with the `@` character, are then used as command arguments.

Compilation arguments and file...

Instead of passing arguments directly to the javac tool, we can store them in argument files. The names of those files, prefixed with the `@` character, are then used as command arguments.

When the javac command encounters an argument starting with `@`, it interprets the following characters as the path to a file and expands the file's content into an argument list. Spaces and newline characters can be used to separate arguments included in such an argument file.

Compilation arguments and file...

Instead of passing arguments directly to the javac tool, we can store them in argument files. The names of those files, prefixed with the `@` character, are then used as command arguments.

When the javac command encounters an argument starting with `@`, it interprets the following characters as the path to a file and expands the file's content into an argument list. Spaces and newline characters can be used to separate arguments included in such an argument file.

Example:

```
-d ../bin/
-verbatim
it/unicam/cs/pa/battleship19/Player.java
```

To be continued. . .