# Simple Commands

BaseIO simplifies command creation with the class `SimpleCommand`. Instead of implementing `ICommand` you extend `SimpleCommand`.

> If you don't like the simplifications made by this feature just override the original method from ICommand.

This is what SimpleCommand can do for you:

- ## Simple command names

  `getCommandName()` and `getCommandAliases()` are combined in `commandNames()` where the first command name is the actual command name and the others are aliases. Example:

- ```
  public List<String> commandNames() {
      //return Collections.singletonList("name1");
      return Arrays.asList("name1", "name2", "name3");
  }
  ```

- ## localized/default command usage

  Provide a localization key in `commandUsage()` and it will be localized as described in Localization. Provide an invalid or no (*null*) localization key and SimpleCommand creates a default command usage for you based on the arguments.

- ## redundancy removal

  `isUsernameIndex` and `compareTo` are removed.

- ## Options

  You can define command options in `getOptions()`. Options are arguments starting with a hyphen that can be used at any position of the arguments. Additionally options might have arguments attached to themselves. When implemented, getOptions() must return a map with the option string as the key and a string array of potential option arguments, the elements describing the option argument at its index. If found, the options will be cut out of the argument array and added to the option set that will be passed to the run method you have to implement in your command.

  **Example:**
  ```

- ```
  @Override
  public void run(MinecraftServer server, ICommandSender sender, String[] args, Map<String, String[]> option
      System.out.println("Args: " + String.join(" ", args));
      for (Map.Entry<String, String[]> option: options.entrySet()) {
          System.out.println("Option " + option.getKey() + ": " + (option.getValue == null ? "null" : String.join(" ",
      }
  }

  @NotNull
  @Override
  protected Map<String, String[]> getOptions() {
      //return Utils.mapKeys("opt1", "opt2", "opt3");
      return Utils.map("opt1", new String[]{"oa1", "oa2"}, "opt2", null, "opt3", null);
  }
  ```

  If now a player uses the command `/mycommand arg1 arg2 -opt1 A B arg3 -opt3` the output would be:

  > Args: arg1 arg2 arg3
  > Option opt1: A B
  > Option opt3: null

- ## Tab Completion

  Tab completions can be defined by the return string in `tabCompletions()`. The syntax allowes highly configurable tab completions:
  1. The completions for each argument are seperated by a space.
  2. The different completions for an argument are seperated by a pipe ("|").
  3. Conditions for a completion are written in braces directly before it.
  4. If you have multiple completions for a condition you can seperate them by a comma instead of a pipe.

  Conditions are arguments that have to be typed anywhere before the completion. With that you are able to implement completely different tab completions based on the arguments the user used. See the example for usage.

  Examples:

- ```
  "comp1|comp2|comp3" => The completions "comp1","comp2" and "comp3" for the first argument

  "compA compB1|compB2 compC" =>
   "compA" for the first, "compB1" and "compB2" for the second and "compC" for the third argument

  "compA1|compA2 (compA1)compB1,compB2|compB3" =>
   "compA1" and "compA2" for the first argument, "compB3" for the second argument, "compB1" and "compB2
  ```

  Variables are available for tab completions. They are defined by a leading "%" and are also customizable. Build-in variables are:

- `%d`: Dimensions (can be resolved to the dimension id by WorldUtils.
- `%p`: Players
- `%null`: No completions (explicitly)

Additionally you can define your own variable by implementing `customTabCompletions`. This method will be called if the tab completion contains an unknown variable and requests a list of completions for this variable. The currently requested variable is found in the parameter `tag` without the leading `%`. Other parameters are the command sender, the currently typed arguments and the completions up to that point (**read-only**).

Example:

```
@Override
protected String tabCompletions() {
  return "compA|%p|%var1";
}

@Override
protected Collection<String> customTabCompletions(String tag, List<String> current, ICommandSender sen
    if (tag.equals("var1")) {
        return Collections.singletonList("result1");
    }
    return null;
}
```

---

Revision #13
Created 31 May 2017 11:00:39 by deregges
Updated 15 October 2017 21:44:22 by deregges