IDEs

Introduction

What

★☆☆☆ 🏆 Can explain IDEs

Professional software engineers often write code using *Integrated Development Environments (IDEs)*. IDEs support most development-related work within the same tool (hence, the term *integrated*).

An IDE generally consists of:

- A source code editor that includes features such as syntax coloring, auto-completion, easy code navigation, error highlighting, and code-snippet generation.
- A compiler and/or an interpreter (together with other build automation support) that facilitates the compilation/linking/running/deployment of a program.
- A debugger that allows the developer to execute the program one step at a time to observe the run-time behavior in order to locate bugs.
- Other tools that aid various aspects of coding e.g. support for automated testing, drag-and-drop construction of UI components, version management support, simulation of the target runtime platform, and modeling support.

Examples of popular IDEs:

- Java: Eclipse, Intellij IDEA, NetBeans
- C#, C++: Visual Studio
- Swift: XCode
- Python: PyCharm

Some web-based IDEs have appeared in recent times too e.g., Amazon's Cloud9 IDE.

Some experienced developers, in particular those with a UNIX background, prefer lightweight yet powerful text editors with scripting capabilities (e.g. Emacs) over heavier IDEs.

Exercises

Debugging

What

★★★☆ 🏆 Can explain debugging

***Debugging* is the process of discovering defects in the program.** Here are some approaches to debugging:

- **Bad** -- **By inserting temporary print statements**: This is an ad-hoc approach in which print statements are inserted in the program to print information relevant to debugging, such as variable values. e.g. `Exiting process() method, x is 5.347`. This approach is not recommended due to these reasons:
    - Incurs extra effort when inserting and removing the print statements.
    - These extraneous program modifications increase the risk of introducing errors into the program.
    - These print statements, if not removed promptly after the debugging, may even appear unexpectedly in the production version.

- 👎 Bad -- **By manually tracing through the code**: Otherwise known as 'eye-balling', this approach doesn't have the cons of the previous approach, but it too is not recommended (other than as a 'quick try') due to these reasons:
  - It is a difficult, time consuming, and error-prone technique.
  - If you didn't spot the error while writing the code, you might not spot the error when reading the code either.
- 👍 Good -- **Using a debugger**: A debugger tool allows you to pause the execution, then step through the code one statement at a time while examining the internal state if necessary. Most IDEs come with an inbuilt debugger. **This is the recommended approach for debugging**.