Reserved words in ruby

The following are predefined words that have been reserved to execute specific tasks, which means that we cannot use them for any other purpose, such as naming variables, objects or constants.

- **__ENCODING__**

  The script encoding of the current file.
- **__LINE__**

  The line number of this keyword in the current file.
- **__FILE__**

  The path to the current file.
- **alias**

  Creates an alias between two methods (and other things).
- **and**

  Short-circuit Boolean and with lower precedence than &&
- **begin**

  Starts an exception handling block.
- **BEGIN**

  Runs before any other code in the current file.
- **break**

  Leaves a block early.
- **case**

  Starts a case expression.
- **class**

  Creates or opens a class.
- **def**

  Defines a method.
- **defined?**

  Returns a string describing its argument.
- **do**

Starts a block.

- **else**

  The unhandled condition in case, if and unless expressions.

- **elsif**

  An alternate condition for an if expression.

- **end**

  The end of a syntax block. Used by classes, modules, methods, exception handling and control expressions.

- **END**

  Runs after any other code in the current file.

- **ensure**

  Starts a section of code that is always run when an exception is raised.

- **false**

  Boolean false.

- **for**

  A loop that is similar to using the each method.

- **if**

  Used for if and modifier if expressions.

- **in**

  Used to separate the iterable object and iterator variable in a for loop.

- **module**

  Creates or opens a module.

- **next**

  Skips the rest of the block.

- **nil**

  A false value usually indicating “no value” or “unknown”.

- **not**

  Inverts the following boolean expression. Has a lower precedence than !

- **or**

  Boolean or with lower precedence than ||

- **redo**

  Restarts execution in the current block.
- **rescue**

  Starts an exception section of code in a begin block.
- **retry**

  Retries an exception block.
- **return**

  Exits a method.
- **self**

  The object the current method is attached to.
- **super**

  Calls the current method in a superclass.
- **then**

  Indicates the end of conditional blocks in control structures.
- **true**

  Boolean true.
- **undef**

  Prevents a class or module from responding to a method call.
- **unless**

  Used for unless and modifier unless expressions.
- **until**

  Creates a loop that executes until the condition is true.
- **when**

  A condition in a case expression.
- **while**

  Creates a loop that executes while the condition is true.
- **yield**

  Starts execution of the block sent to the current method.

Software Design Patterns

- ### What are design patterns?

  Software design patterns are reusable solutions to common problems in software development. It is not a finished design that can be transformed directly into code, rather they are guides to help developers to create products following design patterns.

  Sometimes they are confused with algorithms since both concepts describe typical solutions to known problems, however algorithms define a clear set of actions that can achieve a goal and a pattern is a more high-level description of a solution, it acts like a blueprint that we can customize to solve a problem.

- ### Why use design patterns?

  Because they offer a best practice approach to **support object-oriented** software design. They leverage a common language, which makes it easier to **communicate** about problems and to improve code **readability** and **architecture** in early stages of the planning. When used well, design patterns can both **speed up** the development process and generally **reduce** the chance of errors.

  Some benefits:

  - Solutions already exist, so it speeds up the development process.
  - Standard way to solve a problem (better communication inside the team)
  - They teach how to solve all sorts of problems using principles of object-oriented design.

- **Types of design patterns**
  - **Creational Design Patterns**

    These patterns deal with object creation and initialization, they also provide object creation mechanisms that increase flexibility and reuse of existing code.

    - **Factory Method**: Creates objects with a common interface and lets a class defer instantiation to subclasses.
    - **Abstract Factory**: Creates a family of related objects.
    - **Builder**: A step-by-step pattern for creating complex objects, separating construction and representation.
    - **Prototype**: Supports the copying of existing objects without code becoming dependent on classes.
    - **Singleton**: Restricts object creation for a class to only one instance.

  - **Structural Design Patterns**

    Explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

    - **Adapter**: How to change or adapt an interface to that of another existing class to allow incompatible interfaces to work together.
    - **Bridge**: A method to decouple an interface from its implementation.
    - **Composite**: Leverages a tree structure to support manipulation as one object.
    - **Decorator**: Dynamically extends (adds or overrides) functionality.
    - **Façade**: Defines a high-level interface to simplify the use of a large body of code.
    - **Flyweight**: Minimize memory use by sharing data with similar objects.

- **Proxy**: How to represent an object with another object to enable access control, reduce cost and reduce complexity.

- **Behavioral Design Patterns**

  A behavioral design pattern is concerned with communication between objects and how responsibilities are assigned between objects.

- **Chain of Responsibility**: A method for commands to be delegated to a chain of processing objects.
- **Command**: Encapsulates a command request in an object.
- **Interpreter**: Supports the use of language elements within an application.
- **Iterator**: Supports iterative (sequential) access to collection elements.
- **Mediator**: Articulates simple communication between classes.
- **Memento**: A process to save and restore the internal/original state of an object.
- **Observer**: Defines how to notify objects of changes to other object(s).
- **State**: How to alter the behavior of an object when its stage changes.
- **Strategy**: Encapsulates an algorithm inside a class.
- **Visitor**: Defines a new operation on a class without making changes to the class.
- **Template Method**: Defines the skeleton of an operation while allowing subclasses to refine certain steps.

- **Why are they criticized?**

  Inefficient solutions

  Patterns try to systematize approaches that are already widely used. This unification is viewed by many as a dogma, and they implement patterns “to the letter”, without adapting them to the context of their project.

  Anti-Patterns

  There is not always a straightforward solution to implementing each pattern, with the risk of creating an **anti-pattern** (an ineffective or counterproductive solution) if the wrong method is chosen.

- **Conclusion**

  There are advantages and disadvantages to using software design patterns, so it’s important to know when to use them and when not and how best to implement each pattern.

References

- https://www.netsolutions.com/insights/software-design-pattern/
- https://sourcemaking.com/design_patterns
- https://refactoring.guru/design-patterns
- https://blog.devgenius.io/software-design-patterns-1b41de14ab8b
- https://docs.ruby-lang.org/en/2.2.0/keywords_rdoc.html