Duck Typing Across Languages: Python, JavaScript, Java, and C#

Introduction

The concept of duck typing is commonly introduced through its well-known slogan: "If it looks like a duck and quacks like a duck, it must be a duck". While this phrase provides an intuitive entry point, it conceals more subtle and complex aspects of this typing system. Superficially, it suggests that a duck typed programming language should verify the correct usage of objects by observing their behaviour, rather than relying on their statical properties. However, it's important to stress the distinctions between duck typing, structural typing, and nominal typing, as well as understanding the central role played by the runtime in dynamically typed languages.

To deepen this initial understanding, I asked what duck typing fundamentally means. The AI[1] answered that “duck typing is an implicit structural typing with runtime checking,” which means that the notion of type is not tied to class names or explicit type declarations, but rather to the set of operations the object supports at the moment it is used. More precisely, it stated that “duck typing is a structural type, not a nominal one.” This characterization is only partially correct, as it lacks of a crucial aspect of duck typing, that is its dynamic nature. Similar definitions are provided in many sites or blog, as well as in the Python documentation.

So I further asked for a precise distinction between nominal typed languages, structural typed languages and duck typed languages. In nominally typed languages, the type of an object depends on its declared type, and static checks are performed to verify that the type of the object is valid with respect to the expected type, possibly relying on features such as inheritance or interface implementation. Static structural typed languages, by contrast, base compatibility on compile-time knowledge of an object’s structure, requiring that the set of supported operations be fixed and known. Duck typing departs from both approaches by eliminating any mandatory declarations and verifying type compatibility purely at runtime, checking if the object has the requested methods.

Before moving on to a more detailed analysis, I also questioned why duck typing was introduced and what its main advantages and disadvantages are. As expected, duck typing emphasizes flexibility, in contrast to more rigid statically typed languages, facilitating interoperability between components and different languages, since objects are not required to satisfy an explicit and precise language-related contract before being used. Moreover, the need to interact with highly dynamic data source (such as JSON, XML documents or external code in web-oriented environments) has contributed significantly to the adoption and spread of duck typing.

Among the main advantages highlighted there are a much simpler form of ad-hoc polymorphism, since no interfaces or similar concept are needed, and a faster development and prototyping. The drawbacks, however, are equally evident: the absence of static checks forces programmers to infer expected behaviour informal conventions; responsibility for correctness is shifted from the language to the programmer; and scalability becomes extremely difficult.

Dynamically typed languages

I began exploring duck typing by asking a broad question: "why do dynamically typed programming languages rely on duck typing?". The answer was reasonable, as the AI stated that dynamic typing provides no mechanism to verify type compatibility before execution, since such checks are deferred to runtime. As a result, it is impossible for these languages to enforce static type checks like those mentioned before. Duck typing therefore fits naturally with the design goals of dynamic languages such as Python and JavaScript, which tends to prioritize flexibility, expressiveness, and rapid development over compile-time type safety.
To understand duck typing in these languages, the runtime is the central aspect that must be analysed.

Python

Upon asking how the Python runtime implements duck typing, the AI answered by highlighting how objects are implemented in Python. In Python every value is represented as an object, instance of some class. Each object is characterized by three components: the identity (i.e. the memory address where it's stored), its representation or value, and its type, "which describes what kind of object it is and how it behaves at runtime". That is a valid definition of type, as it matches the one provided in the Data model section of the Python documentation.

When discussing about duck typing, the notion of type is the particularly important, as it is itself an object that stores important information such as attributes names, method definition and possibly inheritance relation with other classes. However the analysis becomes more subtle when we consider the distinction between the high-level and low-level views of classes and objects in Python.

At high level, the definition of a class creates a new namespace, represented as a class object with the same name. This namespace contains all the statements specified in the class body (attributes and method definition). An object instantiation simply creates a nested namespace inside the class' one, allowing implicit attribute referencing with the visibility rules of Python. The official documentation states that namespaces are implemented as dictionaries. As a result, each instance object maintains its own dictionary, representing its instance namespace, while being able to access the class dictionary.

This last information is particularly important because it's directly related to how attribute and method lookup is performed at runtime.

From a low-level perspective, in CPython, the main Python interpreter written in C, the class or type object is actually represented as a C struct called PyTypeObject. On the other hand, every "normal" Python object is represented as a PyObject struct, which always maintains a reference to its corresponding PyTypeObject. Again the Python documentation confirms these affirmations. More specifically, the core structure of a PyObject is defined as follows:

```
typedef struct _object {
Py_ssize_t ob_refcnt; /* reference count */
struct _typeobject *ob_type; /* pointer to type object */
} PyObject;
```

All additional information of each specific object are stored in extended structs, as stated by the AI and the documentation as well.

At this point, all the necessary elements to explain the attribute and method lookup process are in place. The lookup scheme can be summarized as follows:

1. First, the requested name is searched inside the namespace of the object on which the access is performed. This means that the lookup starts in the instance dictionary `x.__dict__`.
2. If the previous search fails, it continues in the class namespace. If the class inherits from one or more classes, then the Method Resolution Order (MRO) is used to linearize the set of inherited classes, to further expand the search.
3. If the attribute or method is eventually found, the corresponding operation is performed; otherwise, an `AttributeError` is raised.

This description, however, only captures the abstract behaviour of attribute and method access. The actual operations performed by CPython are more complex.

When an attribute is accessed, the `PyObject_GetAttr` function is called. This function performs the lookup of the requested attribute or method name (represented as a `PyObject`) on a given `PyObject`. The value returned is either a reference to a `PyObject` (the requested attribute or method) or `NULL` if the search fails.

More precisely, `PyObject_GetAttr` inspects the `PyTypeObject` of the `PyObject` associated with the target instance and searches for the `tp_getattro` slot, which contains the handler responsible for attribute access. By default, this slots points to the `PyObject_GenericGetAttr` function, that delegates the actual work to the

`PyObject_GenericGetAttrWithDict` function. This is the core of the attribute lookup mechanism that can be divided into three main phases.

The first phase consists of initializing the variables required for the lookup:

```c
PyTypeObject *tp = Py_TYPE(obj);
PyObject *descr = NULL;
PyObject *res = NULL;
descrgetfunc f;
```

Here, `tp` is the pointer to the `PyTypeObject` associated with the instance `obj`, which is needed for class-level lookup. The variable `descr` store the result of the `PyType_Lookup` function, which scans the class namespace and all the inherited classes (following the MRO) for the requested name. This variable represents the object corresponding to the requested attribute or method. Finally, `f` is used to store the data descriptor associated to `descr`, if present.

The lookup begins at the class level:

```c
descr = _PyType_Lookup(tp, name);
f = NULL;
if (descr != NULL) {
    Py_INCREF(descr);
    f = Py_TYPE(descr)->tp_descr_get;
    if (f != NULL && PyDescr_IsData(descr)) {
        // Data descriptor found
        res = f(descr, obj, (PyObject *)tp);
        goto done;
    }
}
```

If `_PyType_Lookup` succeeds, the result is inspected to determine whether it is a data descriptor or not. If the object defines a `tp_descr_get` slot and satisfies the data descriptor condition, then the descriptor takes precedence over instance attributes.

If no data descriptor is found, precedence is given to the instance namespace:

```c
if (dict != NULL) {
    res = PyDict_GetItemWithError(dict, name);
    if (res != NULL) {
        Py_INCREF(res);
        goto done;
    }
}
```

Finally, if the instance dictionary lookup fails, the class-level result is reconsidered:

```c
if (f != NULL) {
    res = f(descr, obj, (PyObject *)Py_TYPE(obj));
    goto done;
}
if (descr != NULL) {
    res = descr;
    descr = NULL;
    goto done;
}
```

If `descr` defines a `tp_descr_get` slot but is not a data descriptor, it is treated as a *non-data descriptor*, and its `__get__` method is invoked. Otherwise, if the object is not a descriptor at all, the raw object found in the class namespace is returned directly.

If none of the above cases succeed, the lookup fails and an `AttributeError` exception is raised.

The code presented here closely resembles the actual CPython implementation, although it is intentionally simplified for explanatory purposes. The full and optimized version can be found in the CPython source code.

At first glance, this implementation may suggest that methods and attributes defined in classes are prioritized over instances ones. This is not entirely true. When I asked the AI to clarify what data descriptors are, it replied that they are Python objects that implement `__set__`, `__delete__` or both methods. They are used to design and control attribute access and update directly at class level for specific attribute names.

```
class C:
    @property
    def value(self):
        return 42

c = C()
c.value = 10  # triggers property's __set__ or raises
```

This AI generated code is an hallucination, as it won't run correctly, but throws an exception complaining that no setter is defined. The `@property` decorator is used to generate the getter for the value attribute, but assigning to `c.value` attempts to invoke a setter that doesn't exist.

Nevertheless, the underlying idea is correct: since data descriptors are meant to perform some specific operations when accessing or updating some attribute. For this reason, Python gives them precedence over instance attributes during attribute lookup.

After requesting a concrete use case for data descriptors, the following example effectively illustrates their behaviour:

```
class NonNegative:
    def __init__(self):
        self.values = {}  # store per-instance values internally
    def __get__(self, obj, objtype=None):
        return self.values.get(id(obj), 0)  # fallback default
    def __set__(self, obj, value):
        if not isinstance(value, int) or value < 0:
            raise ValueError("must be non-negative")
        self.values[id(obj)] = value
class Account:
    balance = NonNegative()
a = Account()
a.balance = 50
print(a.balance)  # 50
# directly setting in __dict__ does not affect descriptor
a.__dict__['balance'] = -999
print(a.balance)  # still 50
```

Although this isn't an advanced use of data descriptors, it correctly captures the semantics described earlier.

In summary, instance attributes take precedence over non-data descriptors, while data descriptors are deliberately prioritized to preserve the semantics explicitly defined at the class level.

Returning to the method lookup analysis, if the requested name refers to an attribute, the `PyObject_GenericGetAttrWithDict` function checks whether the requested name is a data descriptor. Methods aren't data descriptor, so in their case the first effective lookup step is performed in the **instance namespace**, potentially allowing an instance field to shadow a class-defined one.

As a final remark, if the requested name refers to a method, the result of the lookup is a bound method. This is a function object in which the self parameter is already bound to the current instance. Therefore, a method invocation in Python conceptually consists of two distinct steps: first, the method is searched using the lookup procedure described above; second, if the method is found, the resulting bound method object is invoked like a function.

This entire lookup process is what enables and concretely implements duck typing in Python. When an object is used in a given context, Python does not perform any static type checking to verify if the object belongs to a specific class or implements some interface. Instead, it simply try to access the required attribute or method **at runtime**, relying on this dynamic lookup mechanism. If the lookup succeeds, the operation proceeds as expected; if it fails, an `AttributeError` is raised. As a result, Python encourages the EAFP (Easier to Ask Forgiveness than Permission) programming style. In this programming style, operations are attempted and potential runtime errors, such as `AttributeError`, are handled using `try...except` blocks, rather than checking whether an object provides a given attribute or method.

As a result, an object is considered suitable for a certain operation solely based on its runtime behaviour, specifically, whether it provides the required attributes or methods. Hence duck typing in Python emerges naturally from the object model and attribute lookup mechanism.

To conclude, I asked for some code example demonstrating duck typed behaviour. One particularly illustrative is the following:

```
def log_message(logger, level, msg):
    logger.log(level, msg)
# Two implementations:
class ConsoleLogger:
    def log(self, level, msg): print(f"{level}: {msg}")
class FileLogger:
    def __init__(self, filename): self.file = open(filename, "a")
    def log(self, level, msg): self.file.write(f"{level}: {msg}\n")
log_message(ConsoleLogger(), "INFO", "Started")
log_message(FileLogger("app.log"), "WARN", "Low disk")
```

In this example, the two classes `ConsoleLogger` and `FileLogger` are completely unrelated: they don't share a common base class or even explicitly implement a logger-like interface. Nevertheless, the function `log_message` operates correctly on both objects because it only relies on the presence of a `log` method at runtime.

Another examples involves iterators:

```
class CountUpTo:
    def __init__(self, limit): self.limit = limit
    def __iter__(self): return self
    def __next__(self):
        if self.limit <= 0:
            raise StopIteration
        self.limit -= 1
        return self.limit
print(list(CountUpTo(5)))  # [4,3,2,1,0]
```

Here, the list constructor implicitly requires that its argument implements the `__iter__` and `__next__` methods. The class `CountUpTo` does not implements any iterator interface, or inherits the required methods from another class. Statically, it is therefore impossible to determine whether the constructor call is well-defined. Dynamically, the invocation succeed because `CountUpTo` provides those methods at runtime.

Finally:

```
def call_if_callable(obj):
    try:
        obj()
    except TypeError:
        print("Not callable")
    else:
        print("Called successfully!")
call_if_callable(lambda: print("Hi"))
call_if_callable(42)
```

In this case, the function `call_if_callable` simply attempts to invoke the passed object without performing any prior checks. No static guarantees are enforced when the call is made; instead, potential runtime errors are handled using a `try` / `except` block following the EAFP principle. If the object is callable, the invocation succeeds; otherwise, a `TypeError` is raised and caught.

JavaScript

As before, I began analysing duck typing in JavaScript by asking the AI how it is implemented and how objects are represented in the language. The AI explained that, in JavaScript, every object maintains a reference to another object called its prototype. Since prototypes are themselves objects, they also reference their own prototypes. This creates a *prototype chain* that always terminates with a `null` reference, since `null` is the prototype of `Object`.

Attribute and method lookup in JavaScript relies entirely on the prototype chain. When accessing a property of a JavaScript object, the interpreter first checks whether the object itself defines the requested property. If it doesn't, the search continues by traversing the prototype chain, checking each prototype until either the property is found or the end of the chain is reached. This mechanism is also described in the official JavaScript documentation.

This approach provides a simple yet effective way to implement both inheritance and method lookup, and it naturally supports duck typing. The underlying idea is conceptually similar to Python's attribute resolution strategy: first inspect the object itself, and then consult a sequence of related objects (the class hierarchy in Python, the prototype chain in JavaScript). However, these two languages differ in how objects are modelled and how they are related to their classes.

According to both the AI and the JavaScript documentation, JavaScript objects can be viewed as dynamic key–value mappings, where one of the entries is `__proto__`, the reference to the object's prototype. Objects can be created in two main ways. If an object is declared using an object literal (e.g., `let obj = { ... }`), its default prototype is `Object.prototype`. If an object is created using the `new` keyword (e.g., `let obj = new Obj(...)`), its prototype is set to `Obj.prototype`.

Starting with the second object-creation mechanism, `Obj` may appear to refer to a class, as in most object oriented programming languages. However, a peculiarity of JavaScript is that classes are syntactic sugar. While in Python classes are objects with their own identity and behaviour, in JavaScript the `class` syntax is essentially a more readable abstraction over constructor functions and prototypes, as confirmed in the JavaScript documentation.

More precisely, a class definition such as:

```
class Obj {
    constructor(args) { ... }
    method() { ... }
}
```

Is conceptually equivalent to defining a constructor function:

```
function Obj(args) { ... }
```

Plus an explicitly attachment of the specified methods to `Obj.prototype`:

```
Obj.prototype.method = function() {...}
```

The `new` keyword is used to invoke the constructor function, but it also performs several additional operations, as specified in the JavaScript documentation:

1. A new empty object is created.
2. The internal `[[Prototype]]` of this object is set to the constructor function's `prototype`.
3. The reserved keyword `this` is bound to the newly created object, and the constructor function body is executed. Any assignment to `this` inside the constructor therefore creates an own property of the new object.
4. If the constructor explicitly returns an object, that object becomes the result of the `new` expression; otherwise, the newly created object bound to `this` is returned.

The `class` syntax encapsulates the constructor function but also the definition of the constructor's prototype. All methods declared inside the class are automatically added to the prototype of the constructor, so they aren't own properties of class instances. Without this syntactic sugar, these prototype assignments would need to be written explicitly, making the code more verbose and error-prone.

This design represents JavaScript's prototype-based nature: rather than associating objects with classes in the traditional sense, JavaScript links objects directly to other objects, their prototype. Thus, duck typing emerges naturally from the dynamic traversal of the prototype chain at runtime, used to determine type compatibility checking the availability of properties and methods at runtime, not on explicit type declarations or interface conformance

With respect to duck typing, the second step of the `new` operation, the linking of the newly created object to its prototype, is the crucial one, since it enables method lookup through the prototype chain. The lack of an explicit representation for classes in JavaScript introduces an even higher degree of dynamism compared to class-oriented languages. As in Python, object can be modified dynamically at runtime, either by updating their own properties or by modifying their prototypes (or their corresponding classes). In addition, JavaScript allows the prototype of an existing object to be changed dynamically via the `Object.setPrototypeOf` method.

Conceptually, this means that in JavaScript it is possible to dynamically modify the type of an object, thus altering is behaviour, as it is determined by its prototype chain. As a consequence, any attempt to enforce static type checking based is fundamentally unreliable. Instead, duck typing naturally emerges from the prototype-based lookup mechanism: when accessing an attribute or method, JavaScript searches first among the object's own properties and then along its prototype chain. If no matching name is found, a runtime error is raised. If the name resolves to a function, the function is invoked with the `this` keyword bound to the original object, even if the method itself is defined in one of the object's prototypes.

Focusing on object creation via literals, the expression:

`let obj = {};`

is equivalent to:

`let obj = Object.create(Object.prototype);`

This operation creates an empty object whose prototype is `Object.prototype`. Unlike the more explicitly object-oriented approach that rely on constructors and the `new` keyword, no constructor invocation is involved. However, the importance of the prototype concept in the language is such that the prototype chain is correctly established, ensuring that attribute and method lookup behaves consistently.

As with Python, I also asked for a lower-level view on these mechanisms. The AI explained that, according to the ECMAScript specification, core operations such as object creation, property access, and method invocation are defined in terms of internal methods that are not directly exposed to the programmer. These operations correspond, respectively, to `[[Construct]]`, `[[Get]]`, and `[[Call]]`. Although these internal methods are abstract specification rather than concrete implementation details, they provide a formal foundation for understanding how JavaScript realizes dynamic behaviour.

Starting with `[[Get]]`, this internal method is invoked whenever a property access `O.P` is performed. The `[[Get]]` operation takes as input the property key `P` and a *Receiver* value (which, in this case, is bound to `O`), and returns the result of calling `OrdinaryGet(O, P, Receiver)`.

`OrdinaryGet` is the core operation that performs property lookup, and it proceeds according to the following steps:

```
1. Let desc = O.[[GetOwnProperty]](P).
2. If desc is undefined, then
     1. Let parent = O.[[GetPrototypeOf]]().
     2. If parent is null, return undefined.
     3. Return parent.[[Get]](P, Receiver).
3. If IsDataDescriptor(desc) is true, return desc.[[Value]].
4. // Otherwise, it is an accessor property:
5. Let getter be desc.[[Get]].
6. If getter is undefined, return undefined.
7. Return Call(getter, Receiver).
```

As presented before, the first step consists in verifying if P is an own property of O. It this is not the case, the lookup is performed in the prototype of O by calling again `[[Get]]` on the prototype. If the end of the prototype chain is reached (that corresponds to a `null` prototype), the lookup fails and `undefined` is returned.

If the property is found directly on `O`, the algorithm distinguishes between data properties and accessor properties. In the case of a data property (a property associated with a concrete value) that value is returned immediately. If, instead, the property is an accessor property, the associated getter function is retrieved and invoked via the `[[Call]]` internal method, with the *Receiver* bound as the `this` value.

When performing a method invocation `O.P()`, the same lookup procedure is applied. In this case, `OrdinaryGet(O, P, Receiver)` returns a function object, which is then invoked through `[[Call]]`. Regardless of where the object function was retrieved, the `this` keyword is bound to `O`, even if the function was found in one of the prototypes along the prototype chain.

The pseudo-code presented above closely resembles the one defined in the ECMAScript specification.

With regard to object creation, the AI produced the following pseudocode for the `[[Construct]]` internal method:

```
F.[[Construct]](argumentsList, newTarget):
    1. Let obj be a newly created object.
    2. Set obj.[[Prototype]] to newTarget.[[GetPrototypeOf]]().
    3. Let result be F.[[Call]](obj, argumentsList).
    4. If Type(result) is Object, return result.
    5. Return obj.
```

As affirmed by the AI itself, this is an highly simplified description of the actual specification. Nevertheless, it correctly captures the essential steps involved in object construction: the creation of a new object, the linkage of its prototype, the invocation of the constructor function and the conditional return of either the explicitly returned object or the newly created instance.

The full ECMAScript specification defines the `[[Construct]]` operation with an higher detail level:

```
1. Let callerContext be the [running execution context].
2. Let kind be F.[[ConstructorKind]].
3. If kind is base, then
    1. Let thisArgument be [OrdinaryCreateFromConstructor](newTarget, "%Object.prototype%").
4. Let calleeContext be [PrepareForOrdinaryCall] (F, newTarget).
5. [Assert]: calleeContext is now the [running execution context].
6. If kind is base, then
    1. Perform [OrdinaryCallBindThis](F, calleeContext, thisArgument).
    2. Let initializeResult be [Completion]([InitializeInstanceElements](thisArgument, F)).
    3. If initializeResult is an [abrupt completion], then
        1. Remove calleeContext from the [execution context stack] and
restore callerContext as the [running execution context].
        2. Return initializeResult.
7. Let constructorEnv be the LexicalEnvironment of calleeContext.
8. Let result be [Completion]([OrdinaryCallEvaluateBody](F, argumentsList)).
9. Remove calleeContext from the [execution context stack] and restore callerContext as
the [running execution context].
10. If result is a [throw completion], then
    1. Return result.
11. [Assert]: result is a [return completion].
12. If result.[[Value]] [is an Object], return result.[[Value]].
13. If kind is base, return thisArgument.
14. If result.[[Value]] is not undefined, throw a TypeError exception.
15. Let thisBinding be constructorEnv.GetThisBinding().
16. [Assert]: thisBinding [is an Object].
17. Return thisBinding.
```

The most important operations are the creation of the empty object and the prototype linkage, performed at step 3.1 through the invocation of `OrdinaryCreateFromConstructor`; the binding of the `this` keyword at step 6.1 via `OrdinaryCallBindThis`; the execution of the constructor body at step 8 through `OrdinaryCallEvaluateBody`; and, finally, steps 12–17, which determine the value returned by the `new` expression and consequently the returned value of the whole procedure.

In conclusion, in JavaScript duck typing arises directly from the way property access is resolved through objects and their prototype chains. When an object is used in a specific context, the language doesn't and couldn't verify whether that object was built from some specific constructor or implements an explicit interface. Instead, the runtime attempts to resolve the requested property by searching first among the object’s own properties and, if necessary, by traversing its prototype chain.

If a matching property is found, execution continues using the retrieved value, possibly invoking the associated getter. In case of method invocation, the retrieved value is a function that is automatically called with the `this` keyword bound to the original object. If no match is found anywhere along the chain, the lookup yields `undefined`, so invoking a non-existing method results in a runtime exception. This behaviour-driven approach, based on the prototype-based lookup, is the core of duck typing in JavaScript.

As before, I asked the AI to generate some duck-typed code examples in JavaScript, possibly reusing those previously discussed for Python.

```
function firstItem(collection) {
  for (const item of collection) {
    return item;
  }
  return null;
}

console.log(firstItem([1, 2, 3]));        // 1
console.log(firstItem("hello world"));    // "h"
console.log(firstItem(new Set([4, 5])));  // 4
```

The first example illustrates the use of the `for...of` construct, which requires an iterable collection as operand. This requirement is not statically enforced, instead, it is verified dynamically by checking whether the object implements the iteration protocol. Consequently, passing arrays, strings, or sets, not related types, works correctly, because all of them provide the required iteration behaviour.

The logger example discussed earlier can also be naturally translated into JavaScript:

```
function logMessage(logger, level, message) {
  if (typeof logger.log !== "function") {
    throw new Error("Logger must implement log()");
  }
  logger.log(level, message);
}

const consoleLogger = { log: (l, m) => console.log(`${l}: ${m}`) };
const fileLogger = {
  log(level, message) {
    /* imagine writing to a file */
    console.log(`(file) ${level}: ${message}`);
  }
};

logMessage(consoleLogger, "INFO", "Started");
logMessage(fileLogger, "WARN", "Low disk space");
```

Once again, the two logger objects are not related by inheritance and don't implement a shared interface. Nevertheless, the code is correct because, at runtime, both objects provide a `log` method. The AI also introduced an explicit runtime check, to capture incorrect use of the `logMessage` function. This checks follows the LBYL (Look Before You Leap) programming style, which aims to prevent runtime errors before they occur, in contrast with the EAFP programming style, which relies on handling errors after they occur.

The following example further highlights the strong relationship between duck typing and JavaScript's highly dynamic object model:

```
const httpResponse = { status: 200, body: "OK" };
// Later we need it to behave like a file reader:
httpResponse.read = function() {
  return `Status: ${this.status}, Body: ${this.body}`;
};

function processReader(reader) {
  console.log(reader.read());
}

processReader(httpResponse); // Works — no formal type required
```

At creation time, the `httpResponse` object does not support a `read` operation. However, JavaScript allows properties and methods to be added dynamically at runtime. Once the `read` method is introduced, the object becomes compatible with the expectations of `processReader`. The function call succeeds because the object satisfies the required behaviour at the moment it is used.

Before proceeding to the next part, I want to present another interesting concept proposed by the AI: Proxy objects. JavaScript allows the creation of Proxy objects, which are wrapper objects around target objects and can redefine the semantic of the internal method used to perform object operations.

A Proxy object is created providing the target object and the handler object. The handler specifies which internal methods are intercepted and how they are implemented.

The following example shows how a proxy can intercept property access:

```
const target = { name: "Alice", age: 30 };

const handler = {
  get(target, prop, receiver) {
    console.log(`Accessing ${String(prop)}`);
    return Reflect.get(target, prop, receiver); // default behavior
  }
};

const proxy = new Proxy(target, handler);

console.log(proxy.name); // Logs: "Accessing name" + "Alice"
```

In this example, the handler defines a `get` trap, which intercepts the `[[Get]]` internal method. As a result, whenever a property of `proxy` is accessed, the custom `get` function is invoked. The Reflect object is used to explicitly delegate to the default lookup behaviour.

One practical use case of proxies is the definition of default properties:

```
const defaults = { x: 0, y: 0 };

function withDefaults(obj) {
  return new Proxy(obj, {
    get(target, prop) {
      return prop in target ? target[prop] : defaults[prop];
    }
  });
}

const p = withDefaults({ y: 5 });
console.log(p.x, p.y); // 0 5
```

In this example, the proxy dynamically supplies default values for missing properties, without modifying the original object or its prototype chain.

Another common application of proxies is the enforcement of runtime constraints. In the following example, the set trap intercepts property assignments and dynamically enforces a type constraint:

```
const userValidator = {
  set(target, prop, value) {
    if (prop === "age" && typeof value !== "number") {
      throw new TypeError("Age must be a number");
    }
    return Reflect.set(target, prop, value);
  }
};

const user = new Proxy({}, userValidator);
user.age = 30;     // OK
// user.age = "old"; // TypeError
```

Proxy objects are interesting because they allow to redefine the internal methods that determines object behaviour at the specification level. This enable objects to adapt their behaviour dynamically, greatly increasing the expressiveness and flexibility of the language. When combined with the Reflect API, which exposes the default implementations of these internal operations, proxies constitute a powerful metaprogramming feature, as they allow to modify the semantics of fundamental object operations at runtime, further reinforcing JavaScript's highly dynamic, duck-typed nature.

A similar result can be achieved in Python by through the definition of specific methods within a class, namely `__getattribute__`, `__getattr__`, `__setattr__`, and `__delattr__`. These methods allow programmers to intercept and customize attribute access, assignment, and deletion at runtime:

- `__getattribute__` intercepts all attribute access operations.
- `__getattr__` is invoked only when the standard lookup process fails.
- `__setattr__` is called whenever an attribute assignment is performed.
- `__delattr__` is invoked when an attribute deletion is requested.

Unlike data descriptors, these methods intercept all attribute accesses, independently of the specific attribute name, as they modify the global runtime semantics of attribute access and update for the object.

Statically typed languages

After analysing how duck typing naturally emerges in dynamic languages through their object models and runtime lookup mechanisms, it is necessary to shift the focus to statically and nominally typed languages, in particular Java and C#. These languages don't natively support duck typing, since type compatibility is determined explicitly at compile time based on the declared type of value rather than on their runtime behaviour. Nevertheless, both languages provide mechanisms that enable more or less limited forms of duck-typed behaviour.

As before, I began by asking a broad question: "why do statically typed languages rely on explicit type systems and compile-time checks?". The answer was again satisfying. In such languages, the type associated with each variable is explicitly declared by the programmer or inferred by the language. The compiler verifies correctness before execution, leading to early error detection, improved performance (due to the absence of runtime type checks) and stronger safety guarantees, including the possibility of proving program invariants.

The main drawbacks of this approach are increased verbosity code and, most importantly, reduced flexibility. For this reason, even statically typed languages such as Java and C# introduce features that partially resemble dynamic typing, in order to regain some of the expressiveness typically associated with dynamically typed languages.

Java

Java is a common example of a statically and nominally typed language. In Java, every value and every expression is associated with a static type, which is explicitly declared by the programmer or inferred by the language. These static types are used by the Java compiler to enforce type compatibility across all program construct, such as object creation, parameter passing, attribute access or method invocation. Type compatibility is verified by advanced mechanism such as subtyping relations and inheritance hierarchies.

It is important to distinguish between the static type of a variable and its runtime type. The AI in fact produced an example of statically invalid code:

```
Object obj = new Person();
obj.age = 10; // compile-time error: Object has no field age
```

This code does not compile because the static type of `obj` is `Object`, which does not declare any field named `age`, even though the runtime type of `obj` would be `Person`. Attribute access in Java is subject to strict compile-time checks, which can be summarised as follows:

1. The compiler determines the static type of the object being accessed.
2. It verifies that this static type (or one of its supertypes) declares the requested attribute in an unambiguous manner.
3. The type of the field access expression matches the type of the member field.

These rules are formally specified in the Java Language Specification.

Method invocation in Java involves more elaborate checks, mainly due to the presence of method overloading. At compiler time, the Java compiler performs the following steps:

1. It identifies the static type of the expression on which the method is invoked.
2. It determine the arity of the method invocation and the static types of the actual arguments.
3. It search for all methods declared in the static type of the object (as well as in its superclasses and implemented interfaces) that share the same method name and arity, and whose formal parameter types are compatible with the static types of the actual arguments.
4. If multiple candidate methods are found, the compiler applies the most specific method rule to select the appropriate one.

The method selected by this procedure is called the compile-time declaration for the method invocation. Its definition is important because it allows the compiler to generate the corresponding bytecode instruction.

At bytecode level, a virtual method invocation on a class instance has the form `invokevirtual #id`, where the operand `#id` refers to an entry in the constant pool, a table storing symbolic metadata for each **.class** file. Constant-pool entries related to methods have type `CONSTANT_Methodref_info` (for class methods) or `CONSTANT_InterfaceMethodref_info` (for interface methods), and both encodes information such as the class or interface in which the compile-time declaration of the method is found, the method name, and its descriptor (i.e., parameter and return types). Crucially, these entries don't contain a direct reference to the method executable's code. This is intentional, since the method that will actually be executed may differ from the compile-time declaration due to method overriding.

At runtime, the dynamic type of the object may be a subtype of its static type, and that subtype may provide its own implementation of the method. For this reason, the Java Virtual Machine must resolve the symbolic reference through a process known as method resolution. Resolution is the mechanism that translates a symbolic reference into a concrete method to be invoked. Method resolution proceeds by starting from the dynamic type of the object and searches for a method with the same name and descriptor as specified in the symbolic reference. If no matching method is found in the current class, the search continues recursively through the superclass hierarchy; if necessary, it also examines the relevant implemented interfaces.

Although this runtime lookup resembles the dynamic lookup mechanism of dynamically typed languages, the similarities are limited. In java, method resolution occurs only after strict compile-time checks have already ensured that the method exists and that it is applicable with respect to the static type of the arguments. Consequently, many errors that would surface at runtime in a duck-typed languages are instead detected at compile time in Java. The dynamic aspect of method invocation in Java is therefore not a mechanism for establishing type compatibility, but rather a necessary runtime step to support polymorphism and method overriding. In contrast, in dynamically typed languages, dynamic lookup is essential to determining whether an operation is valid at all, since no static guarantees are provided before execution.

Interestingly, Java provides to the programmer the interface feature, which can be interpreted as behavioural contracts. If a class explicitly implements an interface, it must provide implementations for all the methods declared by the interface. This mechanism introduces a limited structural perspective in the language, as interfaces describe a set of required behaviours rather than concrete implementations. However, this structural aspect remains embedded within a nominal typed system, since the relationship between a class and an interface must be declared explicitly. As a result, interfaces cannot fully replicate the behaviour of dynamically typed languages. Duck typing relies on implicit structural compatibility that is verified dynamically at runtime, while Java interfaces enforce explicit and static compatibility checks at compile time. The following example shows this distinction:

```java
public interface Quacker {
    void quack();
}

class Duck implements Quacker {
    public void quack() {
        System.out.println("Quack!");
    }
}

class RobotDuck implements Quacker {
    public void quack() {
        System.out.println("Electronic quack!");
    }
}

public class Main {
    public static void makeItQuack(Quacker q) {
        q.quack();  // only calls quack() defined in Quacker
    }

    public static void main(String[] args) {
        makeItQuack(new Duck());
        makeItQuack(new RobotDuck());
    }
}
```

This program is statically well-typed because both `Duck` and `RobotDuck` explicitly implement the `Quacker` interface, which is the required static type of the formal parameter of the `makeItQuack` method. During compilation, the Java compiler performs several checks:

1. It verifies that both `Duck` and `RobotDuck` provide concrete implementations of all methods declared in the `Quacker` interface.
2. It checks that the method invocation inside `makeItQuack` is valid, which is guaranteed because the parameter `q` is statically declared with type `Quacker`.
3. It verifies the calls to `makeItQuack`, ensuring that the actual arguments are type compatible with the formal parameter.
   All these checks are performed at compile time, guaranteeing that no type errors can occur at runtime.

If `RobotDuck` did not implement the `Quacker` interface, the compiler would complain, since `RobotDuck` and `Quacker` would not be related by a subtyping relationship (instead, `RobotDuck <: Quacker` holds in the example). This would happen even if `RobotDuck` correctly defined a `quack` method with the correct signature.

This behaviour highlights a fundamental difference with duck-typed languages: in Java, type compatibility must be declared explicitly and verified statically, whereas in duck-typed languages compatibility is determined implicitly and at runtime. In the latter case, a method invocation succeeds just if the object passed at runtime provides the required method, without any explicit type declaration.

As a workaround for this limitation, the AI proposed the use of the Adapter pattern:

```
class LegacyDuck {
    public void makeNoise() {
        System.out.println("Quack!");
    }
}
// Adapter
class LegacyDuckAdapter implements Quacker {
    private final LegacyDuck legacy;
    LegacyDuckAdapter(LegacyDuck legacy) { this.legacy = legacy; }
    public void quack() { legacy.makeNoise(); }
}
```

With this approach, an instance of `LegacyDuckAdapter` can be passed to the `makeItQuack` function, since it explicitly implements the `Quacker` interface. Conceptually, this allows `LegacyDuck` to be treated as if it implemented the interface, but only through an explicit and statically defined wrapper. Type compatibility isn't inferred from the structure of `LegacyDuck`, but is instead established by an additional type that is statically correct. While effective, this solution remans a static and explicit. It requires an explicit adapter definition and lacks the flexibility of duck typing, where behavioural compatibility is inferred implicitly at runtime. In contrast to dynamically typed languages, Java enforces compatibility through declared types and explicit design decisions, even when the underlying behaviours are already compatible.

Greater flexibility can be achieved in Java through the use of the Reflection API, which enables runtime introspection and intercession. This means that through reflections a program can inspects classes, methods and field at runtime, but also to create new classes instances and invoke method dynamically without static type guarantees.

These capabilities allow Java to approximate duck-typed behaviour, since it becomes possible to verify at runtime whether an object provides a desired method and to invoke it dynamically, without relying on static type checks. However, this increased flexibility comes at a cost. The use of reflection potentially leads to runtime errors, introduce security concerns and performance overhead.

The core abstraction of the Java Reflection API is the `Class` object, which represents the runtime metadata of a class. A `Class` instance can be obtained either from an existing object (using `obj.getClass()`) or by providing the class name (via `Class.forName(String)`). In both cases, the resulting `Class` object contains all relevant information about the target class. Through this object, it is possible to inspect the presence of specific methods and, if needed, invoke them dynamically.

A simple example illustrating this mechanism is the following:

```
import java.lang.reflect.Method;

public class DuckTypingReflection {

    public static void tryQuack(Object obj) {
        try {
            Method quackMethod = obj.getClass().getMethod("quack");
            quackMethod.invoke(obj);
        } catch (NoSuchMethodException e) {
            System.out.println(obj.getClass().getSimpleName() + " cannot quack!");
```

```java
        } catch (Exception ex) {
            throw new RuntimeException(ex);
        }
    }

    public static void main(String[] args) {
        class Duck { public void quack() { System.out.println("Quack!"); } }
        class RobotDuck { public void quack() { System.out.println("Electronic quack!"); } }
        class Dog { public void bark() { System.out.println("Woof!"); } }

        tryQuack(new Duck());       // Quack!
        tryQuack(new RobotDuck());  // Electronic quack!
        tryQuack(new Dog());        // Dog cannot quack!
    }
}
```

This program resembles more the examples previously discussed for Python and JavaScript. The most relevant aspect is that the `tryQuack` method requires a parameter of static type `Object`. Since in Java all reference types are subtypes of `Object`, any object can be passed to `tryQuack` without violating any compile-time type constraints. However, at runtime, the actual class of the passed object is inspected through the Reflection API, in order to determine whether it provides a method named `quack`. The presence of this method is not verified statically. Instead, all checks are deferred to runtime. For this reason, the use of the `try` – `catch` block is essential. If the requested method doesn't exists, a `NoSuchMethodException` is thrown so it must be handled explicitly. This approach closely mirrors the EAFP programming style described for Python.

Another example is the following:

```java
import java.lang.reflect.Method;

public class DynamicMethodInvoker {

    public static Object callIfExists(Object target, String methodName, Object... args) {
        try {
            Class<?>[] paramTypes = new Class<?>[args.length];
            for (int i = 0; i < args.length; i++) {
                paramTypes[i] = args[i].getClass();
            }
            Method method = target.getClass().getMethod(methodName, paramTypes);
            return method.invoke(target, args);
        } catch (NoSuchMethodException e) {
            System.out.println("Method " + methodName + " not found in " +
                               target.getClass().getSimpleName());
        } catch (Exception ex) {
            throw new RuntimeException(ex);
        }
        return null;
    }

    public static void main(String[] args) {
        class Printer {
            public void print(String s) { System.out.println(s); }
        }

        Printer p = new Printer();
        callIfExists(p, "print", "Hello via reflection!"); // Hello via reflection!
        callIfExists(p, "noSuch", "ignored"); // Method noSuch not found…
    }
}
```

Once again, the static type of the `target` parameter in the `callIfExists` method is `Object`, which means that any object can be passed to the without violating any type constraint. Again, the Reflection API is exploited to determine if the object possesses the required method. Furthermore, to perform the method invocation, not only the method name but also the runtime types of all actual parameters must be provided explicitly, as they are declared as `Object` instances. These parameter types are also retrieved dynamically via reflection.

These example illustrate how Java reflections can be used to approximate duck-typed behaviour: method compatibility is determined dynamically based on the runtime structure of the object, rather than on its static type. However, it also highlights the limitations of this approach. Compared to the examples presented for dynamically typed languages, this solution is more verbose and error-prone. The dynamic behaviour that emerges naturally in languages such as Python or JavaScript must be encoded explicitly in Java through the Reflection API, resulting in increased complexity, less readable and more error-prone code.

C#

Like Java, C# relies on a nominal and statically typed type system, in which every variable and expression is associated with a type that is known (or inferred) at compile time. These static types are used by the compiler to perform type checking, ensuring that operations such as method invocation and attribute access are valid with respect to the declared type of the calling object. In particular, when a member is accessed, the compiler verifies that the static type of the expression declares (or inherits) the requested member.

As in Java, the static type of a variable may differ from the runtime type of the object it references. While the compiler relies on static types to establish type correctness, the runtime type is used to resolve operations that require dynamic behaviours, as method invocations. This distinction between static and dynamic typing contexts is explicitly described in the C# documentation.

In C#, to allow a method to be overridden in subclasses, it must be declared using the `virtual` keyword. A subclass that provides a new implementation of a virtual method must explicitly use the `override` modifier. This design choice makes dynamic dispatch explicit in contrast to Java, where most instance methods are virtual by default. At compile time, the C# compiler checks the validity of method invocations based on the static type of the calling object, possibly resolving overloading. At runtime, however, the Common Language Runtime (CLR) performs dynamic dispatch to determine which overridden method implementation should actually be executed.

According to the C# documentation at the Intermediate Language (IL) level, virtual method calls are typically compiled into the `callvirt` instruction, which requires a method metadata token as a parameter. The method metadata token stores the name, class and signature of the requested method. Importantly, this metadata token doesn't directly point to the concrete method to execute. Instead, the `callvirt` instruction performs a runtime lookup that selects the appropriate method implementation based on the runtime type of the object, rather than on the static type recorded in the metadata. The method resolution process proceeds as follows: starting from the runtime type of the object, the CLR searches for a method whose name and signature match those specified by the metadata token. If no matching method is found in the current class, the search continues up the base class hierarchy until a suitable implementation is located. This lookup is performed through the object's method table, which contains pointers to the actual method implementations associated with the object's runtime type.

Again, this last dynamic lookup mirrors the steps performed in Python and JavaScript, but it is performed only passing static type checks the ensures that the expected behaviour exists for the current object. This dynamic aspect of method invocation isn't used at all for type compatibility, but to correctly support the overriding mechanism.

Again, this final dynamic lookup step closely resembles the behaviour of dynamically typed languages. However, in C#, as in Java, dynamic lookup is performed only after static checks have already ensured that the operation is well typed. The compiler guarantees that the method exists, at runtime the dynamic dispatch selects which implementation must be executed. Therefore, the dynamic aspect of method invocation in C# is not used to establish type compatibility, as it is in duck-typed languages, but rather to support polymorphism and method overriding.

Duck-typed–like behaviours can be partially emulated in C# through the use of interfaces and reflection. The following example illustrates the usage of interfaces:

```
using System;

public interface IQuacker
{
    void Quack();
}

public class Duck : IQuacker
{
    public void Quack() => Console.WriteLine("Quack!");
}

public class RobotDuck : IQuacker
{
    public void Quack() => Console.WriteLine("Electronic quack!");
}

public class Toy
{
    public void Quack() => Console.WriteLine("Squeak!");
}

static class Program
{
    static void MakeItQuack(IQuacker q)
    {
        q.Quack();  // Compile-time safety & polymorphism via interface
    }

    static void Main()
    {
        MakeItQuack(new Duck());
        MakeItQuack(new RobotDuck());
        // MakeItQuack(new Toy()); // Compile error: Toy does not implement IQuacker
    }
}
```

As in Java, interfaces in C# define explicit behavioural contracts that implementing classes are required to satisfy. Since the `makeItQuack` function expects a parameter of an interface type, the compiler statically verifies that any argument passed to the function is an instance of a is a subtype of `IQuacker`. The compiler is also able to statically ensure the correctness of the `Quack` method invocation, as the static type of the formal parameter guarantees the existence of that method.

However, interfaces remains firmly inserted in a nominal typing system. If a class does not explicitly declare that it implements a given interface, it cannot be used where that interface is expected, even if it defines all the required methods. In the example above, the `Toy` class provides a `Quack` method but does not implement `IQuacker`, so passing a `Toy` instance to `MakeItQuack` results in a compile-time error.

As a more flexible use of interfaces, the AI presented an example based on the Strategy pattern:

```
using System;

public interface IValidator
{
    bool Validate(string input);
}

public class NotEmptyValidator : IValidator
{
    public bool Validate(string input) => !string.IsNullOrEmpty(input);
}

public class LengthValidator : IValidator
{
    public int MinLength { get; }
    public LengthValidator(int min) => MinLength = min;
    public bool Validate(string input) => input?.Length >= MinLength;
}

public class FormProcessor
{
    private readonly IValidator _validator;
    public FormProcessor(IValidator validator) => _validator = validator;
    public void Process(string data)
    {
        if (_validator.Validate(data))
            Console.WriteLine("Valid");
        else
            Console.WriteLine("Invalid");
    }
}

static void Main()
{
    var processor1 = new FormProcessor(new NotEmptyValidator());
    var processor2 = new FormProcessor(new LengthValidator(5));

    processor1.Process("Hi");
    processor2.Process("Hi");  // invalid if < 5 chars
}
```

In this example, both `NotEmptyValidator` and `LengthValidator` explicitly implement the `IValidator` interface, and therefore their instances can be safely passed to the `FormProcessor` constructor. Since the compile-time type of the `_validator` field is `IValidator`, the invocation of the `Validate` method inside `Process` can be fully type-checked at compile time.

This pattern allows to change dynamically the behaviour of `FormProcessor` instances by supplying different validator, while still preserving static type safety. The compiler ensures that all validators conform to the expected interface, and at runtime the current implementation of `Validate` method is invoked.

C# Reflection API is extremely similar to Java's reflection mechanism. It introduces additional flexibility by allowing program inspections and dynamic operations, such method invocations, to be performed while deferring checks at runtime.

```
using System;
using System.Reflection;
```

```
static double CalculateArea(object shape) {
    // Look for a method named GetArea
    MethodInfo getArea = shape.GetType().GetMethod("GetArea");
    if (getArea == null) {
        throw new ArgumentException("Shape does not have GetArea()");
    }
    return (double)getArea.Invoke(shape, null);
}

class Circle {
    public double Radius { get; set; }
    public double GetArea() => Math.PI * Radius * Radius;
}

class Rectangle {
    public double Width, Height;
    public double GetArea() => Width * Height;
}

var circle = new Circle { Radius = 3 };
var rect   = new Rectangle { Width = 4, Height = 5 };

Console.WriteLine(CalculateArea(circle)); // 28.27...
Console.WriteLine(CalculateArea(rect));   // 20
```

In this example, no interface is used to statically enforce that `Circle` and `Rectangle` define a common `GetArea` method. From the compiler's perspective, the parameter of `CalculateArea` has type `object`, and therefore no assumptions can be made about the presence of any specific members. Instead, all the checks are deferred to runtime using reflection. The program dynamically inspects the runtime type of the object to determine whether a method named `GetArea` exists and invokes it only if it is found. If the method is missing, an exception is raised explicitly.

Finally, C# introduces the `dynamic` type, which provides the most direct approximation of duck typing within the language. The `dynamic` keyword is a special static type that informs the compiler to bypass all compile-time type checks for the associated variable.

At first glance, `dynamic` may appear similar to declaring a variable of type `object`, but there is an important distinction. When a variable is declared as `object`, the compiler still enforces static type checks based on the `object` type, requiring explicit downcasts before invoking members not defined on `System.Object`. In contrast, when a variable is declared as `dynamic`, no static checks are performed at all, even on member access or method invocation.

A simple example illustrates this behaviour:

```
dynamic x = 10;
x++;            // OK - resolved at runtime
x.NonExistent(); // Compiles, but throws RuntimeBinderException at runtime
```

Here, the compiler doesn't report any error for the invocation of `NonExistent`, because the presence of the `dynamic` type disables compile-time verification. At runtime, however, the actual type of `x` (`int`) does not define such a method, and the operation fails with a `RuntimeBinderException`. As a result, method invocation on `dynamic` values relies entirely on runtime resolution, closely resembling the behaviour of dynamically typed languages.

The use of `dynamic` also affects method overloading. When an argument is declared as `dynamic`, overload resolution is deferred to runtime rather than being performed at compile time:

```
void Foo(int i) { }
void Foo(string s) { }

dynamic d = "hello";
Foo(d); // resolved at runtime to Foo(string)
```

In this case, the compiler does not select an overload during compilation. Instead, the appropriate method is determined at runtime based on the runtime type of `d`. Overall, the `dynamic` keyword is an highly flexible programming feature that introduces very close ducked typed behaviours in the language, without introducing the verbosity typical of the Reflection API. However, the flexibility of this feature comes at the cost of completely losing compile time guarantees, thus introducing potential runtime errors, and incurring additional overhead. The capability of this feature are showed in the following example:

```
using System;

class Duck {
    public void Quack() => Console.WriteLine("Quack!");
}

class Person {
    public void Quack() => Console.WriteLine("The person imitates a duck!");
}

static void MakeItQuack(dynamic obj) {
    try {
        obj.Quack();  // resolved at runtime
    } catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException) {
        Console.WriteLine("This object does not quack.");
    }
}

MakeItQuack(new Duck());   // Quack!
MakeItQuack(new Person()); // The person imitates a duck!
MakeItQuack(42);           // This object does not quack.
```

In this example, the parameter of `MakeItQuack` is declared as `dynamic` and as a result, the invocation of `Quack` is not verified at compile time and is instead resolved entirely at runtime based on the actual type of the object passed as argument.

If the runtime type of the object provides a compatible `Quack` method, the invocation succeeds; otherwise, a `RuntimeBinderException` is thrown and handled explicitly. Once again, this control flow closely mirrors the EAFP programming style.

---

1. All the information presented derives from questions posed to ChatGPT 5.2, cross checked by online resource. Here is the link to the chat ↩