**Programming paradigm**

1. **Imperative programming**

* The program describes a sequence of steps that **change the state** of the computer.
* Tells the computer “**how**” to accomplish a task.
* Programs are always **compiled** to binary executables.

(Note: binary executables run more efficiently since all CPU instructions are themselves imperative statements.)

* **Procedural programming**: grouping code into a compound block (a procedure), then it can be used as a single imperative statement, abstracting the control flow of a program and allowing the developer to express programming ideas more naturally.

1. **Object-oriented Programming**

* Object rather than actions; data rather than logic.
* Advantages:

1. **Inheritance**: enforces thorough data analysis, reduces development time, ensures accurate coding.
2. **Encapsulation**: greater system security and avoids data corruption.
3. **Reusable**: can be easily distributed.
4. **Flexible**: allows new data types to be created.

* **Prototype-based OOP in JavaScript**
  + JS has only 1 construct: objects.
  + **someObject.Prototype** property refers to another object called its **prototype**.

Object.getPrototypeOf()

Object.setPrototypeOf()

* + **null** has no prototype; null is the final link in the prototype chain.
  + **Inheriting properties**:

1. Search for property down the prototype chain, until either a property with a matching name is found or the end of the prototype chain is reached.
2. When an inheriting function is executed, the value of **this** points to the inheriting object.
3. **The prototype property is used primarily for inheritance:** you add methods and properties on a function’s prototype property to make those methods and properties available to instances of that function.
   * **Ways to create objects**
4. Syntax constructs
5. Constructor: a function happens to be called with the **new** operator.
6. Object.create(prototypeObj)
7. class keyword
   * **performance** **issue:** hasOwnProperty() inherits from Object.prototype checks whether an object has a property defined on *itself* and not somewhere on its prototype chain.
8. **Functional Programming**

* Composing pure functions, avoiding shared state, mutable data, and side-effects.
* Pure functions
  + Idempotent function
    - Given the same inputs, will always return the same output.
    - Don’t have any dependency on time or sample resolution, it’s possible to treat continuous data as unbounded data streams.
  + Does not effect external state.
    - Stronger encapsulation.
    - Independence from outer code → portablility.
* Function composition
* Closure
  + Created whenever a function accesses a variable defined outside the immediate function scope.
  + Can be used to create data privacy.

**Questions**

1. Difference between the identity (===) operator and the equality (==) operator.

* ===
  + no type conversion
  + two values of different type → false
* ==
  + compare for equality only after doing necessary type conversions.
* Both are equally quick.
* Both checks for reference equality of objects, value equality of primitive types, including string literal.
* Compare string literal and object created by String() function

var a = new String("123");

var b = "123";

alert(a === b); // false

2. Truth, false, and Equality

* False
  + undefined
  + null
  + 0
  + NaN
  + empty string
* True
  + Numbers that are not NaN or 0
  + Non-empty string
  + Objects are ALWAYS TRUE
* The equality (==) operator: X==Y
  + X, Y same type
    - return X===Y
  + X String, Y Number
    - return toNumber(X)==Y
  + X Boolean, Y any type
    - return toNumber(X)==Y
  + X String/Number, Y Object
    - return toPrimitive(X)==Y
  + Otherwise
    - return false
* ToNumber
  + undefined → NaN
  + null → 0
  + true → 1
  + false → 0
  + Number → the input argument
  + String → Number(String)
  + Object → ToNumber(ToPrimitive(…))
* ToPrimitive
  + Object
    - if valueOf returns a primitive, return it
    - else if toString returns a primitive, return it
    - else throw an error
  + Otherwise → return the input argument
* valueOf
  + Returns the primitive value of the specified object.
  + If an object has no primitive value, returns the object itself.
* Case 1

[0] == true