Object Oriented principles and design.

I am in pain.

Solid:

Single Responsibility Principle (SRP):
A class should only have a single responsibility, and only one reason to change.

Open-Closed Principle (OCP):
Objects should be open to extension but closed for modification.

Liskov Substitution Principle (LSP):
Every subclass or derieved class should be able to substitue for their base or super class.

Interface Segregation Principle (ISP):
A client should never be forced to implement an interface that it doesn't use, nor should clients be forced to depend on methods they do not use. Many client-specific interfaces are better then one general-purpose interface.

Dependency Inversion Principle (DIP):
One should depend on abstractions, rather than on concretions.

Other:

Seperation of Concern (SoC):
Concerns of the software should be split up into individual parts. One should not write their software as one giant block and should instead split the code into smaller more distinctive parts.

Command-Query Separation Principle (CQSP):
A method should either return data or modify data(void in most cases), but not both.
Something that breaks this principle is for example the `pop()` method, that exists for lists in many languages, because it both modifies the list by removing the last index and then it returns that data. This is however allowed because there simply is no better solution.

High Cohesion, Low Coupling (HCLC):
Low Coupling: Modules and classes should be as independent from each other as possible.
High Cohesion: Related code should be close together. For example all methods in a module should be related to the modules distinct task.

Composition over Inheritance (CoI):
Classes should preferably contain other classes that implement needed behavior rather then inheriting behavior.

Law of Demeter (LoD):
To avoid loose coupling classes should only talk to their immediate "friends".
For example: instead of writing `a.b.c.d()` one should do `a.tellBtoTellCtoDoD()`, as one should avoid relying on other classes implementation. This also ties into information hiding, as in for example if something doesn't need to be `public` it shouldn't be `public`. Package- or class-private should always be preferred.

Defensive Programming (DP):
Software and libraries should be written as if the user **will** use it wrongly. For example, if data should only gettable instead of settable, a getter method should be implemented and the data should be hidden.

Concepts

Class and object

"The class is the template used to create objects." It can also contain its own static methods and fields.

Static and dynamic type

Static typed languages are those in which type checking is done at compile-time, whereas dynamic typed languages are those in which type checking is done at run-time. Left side is static, and dynamic is right side.

Encapsulation

Encapsulation means wrapping data together as a single unit. Specifically data should also be hidden and getters should be supplied.

Specification and Implementation inheritance

Implementing an interface or abstract class or inherting code from classes/abstract classes.

Static method and instance method

Static methods are a single copy of a method and can not access instance variables but can be used without an instance, complete opposite of an instance method which can access instance variables and requires an instance.

Variables and attributes

Attributes are instance variables while variables are declared in methods.

Abstract class and interface

An interface is a form of specification inheritance where what methods a class needs to have is defined, similarly an abstract class can do this while also having methods and attributes that are inherited. But can be used as a type but neither can be instanced.

Dynamic Binding

Also called late binding, and this is done when you are overriding inherited public methods as the method used is decided during runtime.

The opposite of this is static binding which is done with static, private and final methods as they cannot be overridden.

Primitive and referencetypes

Primitive types are types such as booleans, integers and floats, and they are stored on the stack.

Reference types are types such as Classes, which are stored on the heap, and when these are passed around you are actually passing around copies of the reference to the object in question.

Alias

Alias is when two variables point to the same object. (They have a copy of the same reference.)

```
A a = new A();B b = A;
```

Super and subclass

When inheriting or implementing another class/interface onto a class, the implementation is a super class and the class implementing said class/interface is the subclass.

Overloading and overriding

Overloading means having multiple methods (or constructors) with the same name, but with different arguments.

Overriding means replacing an inherited method or constructor with a new implementation (or even using the old one in the new one), also known as dynamic or late binding.

Constructor

A specialized method used for initialization of new Objects of a Class.

Initialization

This means assigning a value to an attribute or variable, used in constructors for example.

Mechanisms and techniques

Mutability and immutability
Values that can be modified and or not modified(const/final).

Refactoring
Altering internal code without changing its external behavior.

Defensive Copying
To ensure immutability, getter methods should return a copy instead of a reference the value in question.

Immutable Adapters
An implementation of the adapter pattern, where an interface is created to only expose immutable and safe methods of the underlying class.

Design by contract
This is a design philosophy where a methods specifications are seen as a contract between the client and the method.
"If you promise to give me two positive integers, I promise to give you their biggest common divisor."

- Precondition(Förvilkor): Things that must be fullfil for the method to be called.
- Postcondition(Eftervilkor): If the precondition is not fulfilled there is no guarantee on the return.
  Otherwise the return should be able to be guaranteed.
- Invariant: The needed requirements should be in a well formatted state.

Method Cascading
Calling multiple methods on the same object. For example:
`a.b().c();` instead of `a.b();a.c();`

Mutate-by-copy
Instead of mutating an object's state it should be copied with the new state.

Lambdas and function interfaces
Short-form replacement for anonymous classes for interfaces with single methods:

```
interface k { int p(double g); }
List<k> l = new List<>();
l.push((double g) -> return (int)g;);
```

Exceptions
An error that is thrown whenever something that should not happen happens.
These should be handled using try catches(varies between languages).

Polymorphism and code reuse

Polymorphism

Polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

Subtype-polymorphism

Being able to take the superclass or any of its subclasses and still work.

`functionThatTakesA(new BThatInheritsA())`

Parametric polymorphism

The use of generics, ie `T extends A`.

Ad hoc-polymorphism

Different types should be able to give definitions of the same symbol/name without knowing of each other.

(See operator overloading in C# and type classes in Haskell)

Something like this:

`<T implements +> T add(T x, T y) = x + y;`

This is however not possible in Java.

Co-,contra-, and invariant

Covariant: ? extends T
Contravariant: ? super T
Invariant: T (No sub or supertypes)

Delegation

Delegation means handing over responsibilities for a task to another class or method.

Inheritance

Inheriting means making a class a subclass of another, and inheriting methods and such from the super.

Type constructor and type parameter

`class Box<T>{}`: The T here is a type parameter. Type constructor is a ?.

Type argument and Wildcards

`Box<Integer> b = new Box<>();`: The Integer is in this case a type argument.

Wildcards are used for creating generic declarations and method calls but not for defining generic type, for example: `List<? extends A>` but not `class A<? extends B>`!

Upper and lower bounds

Upper bound wildcard menads `? extends Type` and lower bound means `? super Type`.

Get-Put principle

Use `List<? super T>` to only be able to put data in the list and `List<? extends T>` to be able to get data out of the list.