

# Reading User Input

```
Scanner myObj = new Scanner(System.in); // Create a Scanner object
System.out.println("Enter username");

String userName = myObj.nextLine(); // Read user input
System.out.println("Username is: " + userName); // Output user input
```

```
// Scanner Methods
String myString = myObj.next();
int myInt = myObj.nextInt();
double myDouble = myObj.nextDouble();
```

# Interface Structure

Interfaces are used as a blueprint for classes. Defines methods and constant variables that must be used by the class which implements the interface.

```
interface Printing {
    int portNumber = 1234;
    // Implicitly `public static final`
    void printSingleSided();
    // Implementing Class Must Override
    default void printDoubleSided() {
        // Method body
    } // Implementing Class Can Override
}

class LaserPrinter implements Printing {
    public LaserPrinter() {}
    @Override
    public void printSingleSided() {
        System.out.println("Laser Printer Printing Single Sided");
    } // Necessary to override
    @Override
    public void printDoubleSided() {
        System.out.println("Laser Printer Printing Double Sided");
    } // Not necessary
}
```

- Note that:
- 1| Anything defined in an interface is implicitly public and static
  - 2| An interface can inherit from another interface using the keyword 'extends'

# Comparing two objects

Both Comparable and Comparator interfaces in Java are used for sorting objects, but they have key differences:

Comparable	Comparator
Part of java.lang package	Part of java.util package
Contains only one method: compareTo()	Contains multiple methods, with compare() being the main one
Provides the natural ordering of objects	Provides custom ordering that may be different from natural ordering
Class itself must implement the interface	External class implements the interface
Only one sorting sequence per class	Multiple sorting sequences possible
Used with Arrays.sort(array) or Collections.sort(list)	Used with Arrays.sort(array, comparator) or Collections.sort(list, comparator)

## When to use Comparable:

- When there is a single, obvious natural ordering for objects
- When control over the class source code is available
- When the ordering is intrinsic to the class

## When to use Comparator:

- When multiple ways to sort objects are needed
- When sorting objects that cannot be modified (like classes from libraries)
- When the sorting logic changes based on context or user preference

# Iterators

An Iterator is an interface that provides a way to traverse through elements of a collection sequentially.

```
public interface Iterator<E>
```

Modifier and Return Type	Method	Description
default void	forEachRemaining(Consumer<? super E> action)	Performs the given action for each remaining element until all elements have been processed or the action throws an exception.

boolean	hasNext()	Returns true if the iteration has more elements.
E	next()	Returns the next element in the iteration.
default void	remove()	Removes from the underlying collection the last element returned by this iterator (optional operation).

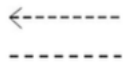
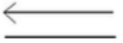
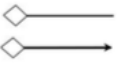
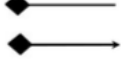
## Public, Default, Protected, and Private

Modifier	Symbol	Class	Package	Subclass	World
public	+	✔	✔	✔	✔
protected	#	✔	✔	✔	✗
no modifier (default)	~	✔	✔	✗	✗
private	-	✔	✗	✗	✗

Note that, for **Nested Classes**, to specify where a variable is coming from in a nested class you can use ``nameOfClass`.this.myVar`



## Modelling Relationships

Association Types

Type	Direction	Notation	Examples	Scope	Ownership	Sharing	Lifetime
Dependency / Weak Association	Uni Bi		CustomerView – Customer	Method	No	Yes	Independent of each other
Simple Association	Uni Bi		Student – Faculty	Object / Class	No	Yes	Independent of each other
Aggregation / Weak Composition	Bi Uni		Car – Tyre Team – Player	Object / Class	Usually, yes	Yes	Mostly dependent
Composition	Bi Uni		Polygon – Side Person – Heart	Object / Class	Always, yes	Yes	Strictly dependent

Many	Exactly One	Zero or More	Specified range
*	1	0..*	2..4

Arrow Type

Type	Notation	Context	Example
Generalisation		Concrete, super and sub class. Note that, if the super class is Abstract, its name will be italicized	Rectangle – Square Vehicle – Car
Realisation		Super entity is an interface	List<T> – ArrayList<T>

Distinction among dependency, association, aggregation and composition is decided by four factors.

Scope	The scope defines the part of class body where an instance of an outside class maybe accessed.
Ownership	An object objA of class A owns an object objB of class B, if objB is "created" inside objA
Sharing	An object objB of class B can be accessed/shared by two or more different objects
Lifetime	The lifetime of an object objA of class A is defined by its existence in memory, i.e. the moment it is created in computer memory until it is killed.

## SOLID Principles

Single Responsibility	Each piece of code (like a module or class) should have only one specific job or one reason to change.
Open/Closed Principle	You should be able to add new features to your code without changing its existing core logic.
Liskov Substitution Principle	If you have different versions of something (like a "dog" and a "cat" both being "animals"), you should be able to use any of those versions interchangeably without causing problems. New versions should behave as expected by anything that uses the original version.

Interface Segregation Principle	Don't force users of your code (clients) to see or use methods they don't actually need. Give them only the tools they require.
Dependency Inversion Principle	Your most important parts of the code (high-level) shouldn't directly rely on the nitty-gritty details (low-level). Instead, both should rely on general ideas or agreements (abstractions). The details should be built to fit those general ideas.
Just to clarify the dependency inversion principle: <i>Don't let your important, central parts of the code get "stuck" using specific, detailed implementations. Instead, design your code so that both the central parts and the specific implementations agree on a common interface or contract. This way, you can easily swap out different implementations without breaking your core system.</i> <i>It's about making your code more "plug-and-play" and less "hardwired."</i>	

SOLID is a guideline for how to structure code, but sometimes it is not applicable for every use-case, like in ...

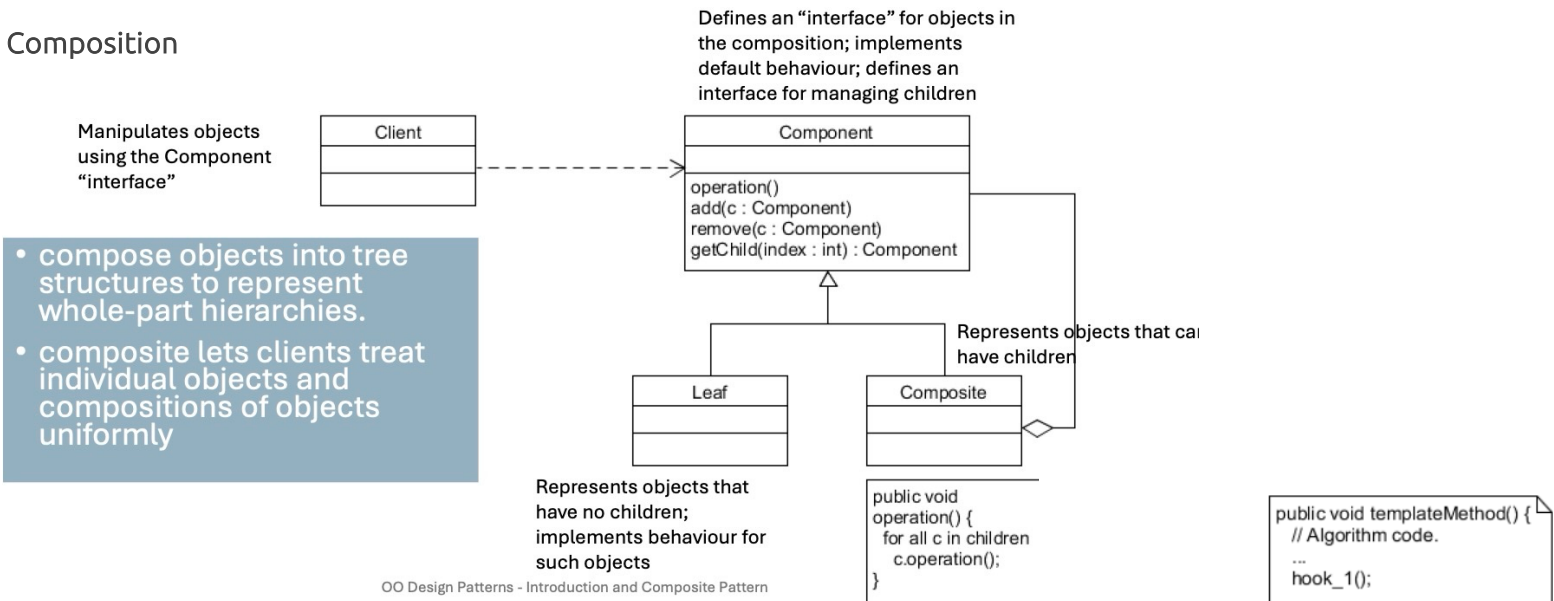
- **SRP (ISP):** Are small classes always easier to understand?
- **DIP (and OCP):** Should every direct dependency result in at least one additional interface?
- **LSP:** does not sound practical enough

## Patterns (Composite, Template, Observer, Singleton, and Factory)

Use-cases for different patterns

Singleton Pattern	How to ensure that a particular class has only one instance which has a global point of access
Template Method Pattern	How to define the skeleton of an algorithm in a superclass and allow subclasses to redefine certain steps without changing the algorithm's structure
Composite Pattern	How to compose objects into tree structures represent whole part hierarchies, where leaf and composite nodes are treated uniformly
Adapter Pattern	How to make objects with incompatible interfaces work together
Factory Pattern	How to deal with the situation where you know that you need to create an object – but you don't know which particular class of object to insatiate

### Composition



### Template

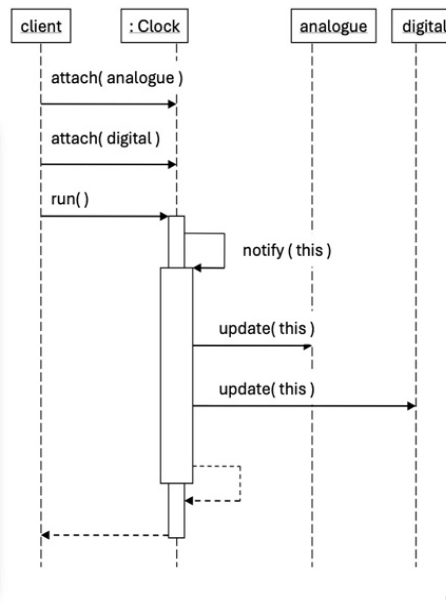
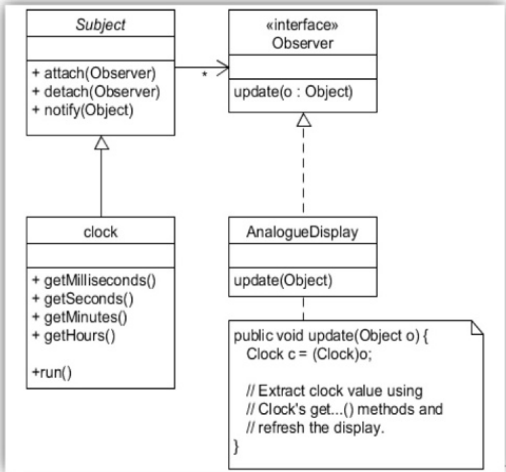
- Template Method is a technique that defines the **skeleton of an algorithm** in a super class method
- The Template Method allows subclasses to **redefine certain steps** of an algorithm without changing the algorithm's structure

### Observer

The Observer pattern is a behavioral design pattern that defines a one-to-many dependency between objects. This means that when one object (the "subject" or "publisher") changes its state, all its dependents (the "observers" or "subscribers") are automatically notified and updated.

## The Observer pattern

### Application to clock App



## Factory

```

enum EnemyType { GOBLIN, ORC, DRAGON; }
public class EnemyFactory {
    public Enemy createEnemy(EnemyType type) {
        return switch (type) {
            case GOBLIN -> new Goblin();
            case ORC -> new Orc();
            case DRAGON -> new Dragon();
            default -> null;
        };
    }
}

```

## Singleton

```

class MySingleton {
    private static MySingleton instance;
    private MySingleton() {}

    public static MySingleton getInstance() {
        if (instance == null) instance = new
        MySingleton();
        return instance;
    }
}

```

## Multithreading

### METHOD 1

```

class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
        ...
    }
}

```

```

PrimeThread p = new PrimeThread(143);
p.start();
try { p.join(); } // makes main thread wait
catch (InterruptedException e) { //... }

```

### METHOD 2

```

class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
        ...
    }
}

```

```

PrimeRun p = new PrimeRun(143);
new Thread(p).start();

```

## Thread Lifecycle

Note that, the runnable state is managed by the OS

Do not call .run() directly, because it will be treated as a normal method call. Use .start() instead

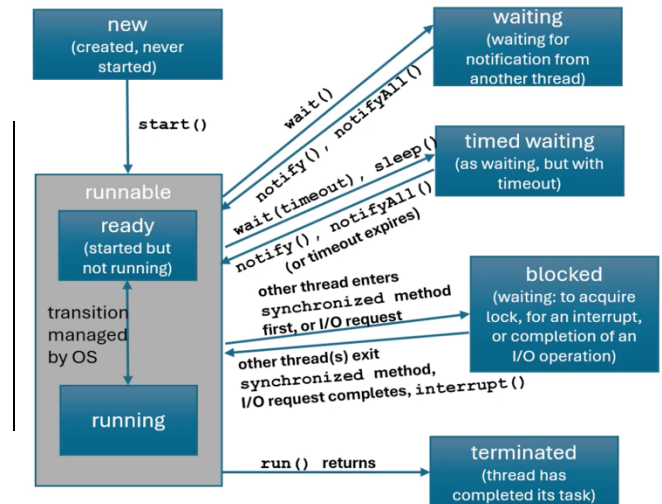
Thread.getState() // returns state

Thread.sleep(milliseconds) // 'pauses' execution of current thread

Defining a function with the 'synchronized' keyword means that an interrupt will be dealt with after the function has completed execution. And it means that only one thread can execute the function at a time

Example syntax:

```
public synchronized int increment() { count++; }
```



## Swing Framework

SwingWorker simplifies the development of Swing applications that need to do background tasks

The swing framework handles coordination of ED (Event Dispatch) and worker threads

