Fakultas Informatika
School of Computing
Telkom University

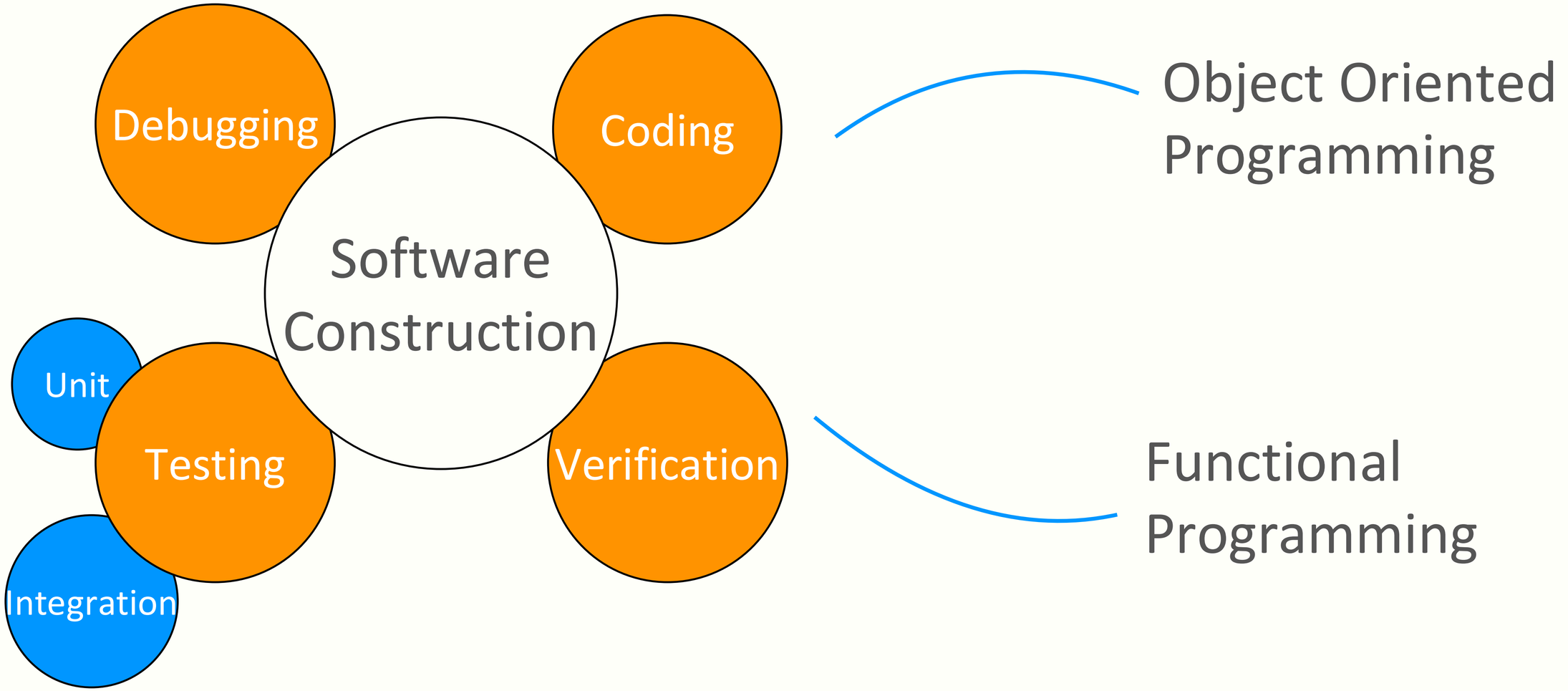
Yayasan Pendidikan Telkom

# Object Oriented Programming
# Functional Programming

## Perancangan Aplikasi Sains Data
## (Week 5 – 6)

**PROGRAM STUDI SAINS DATA**
**TELKOM UNIVERSITY**

# Introduction to OOP and FP



# Brief History of OOP and FP

## Tracing the Development of Programming Paradigms

**Origins of OOP:** Developed in the 1960s with Simula; popularized by Smalltalk, C++, and Java

**Origins of FP:** Rooted in lambda calculus (1930s); adopted in Lisp, Haskell, and modern Python.

**Modern Trends:** Hybrid approaches integrate OOP and FP (e.g., Python, Scala, and JavaScript)

# Understanding The Paradigm

## OOP vs. FP: Understanding the Distinctions

**OOP; State & Objects:** OOP focuses on objects that store state and behavior, making it ideal for modular applications

**FP; Pure Functions & Immutability:** FP emphasizes stateless functions and immutable data, enabling parallelism and predictability

**Common Misconceptions:** OOP is not always slower than FP, and FP can be used for complex applications, not just small scripts

# Object Oriented in Data Science

## Building Reusable and Scalable Code

**Core OOP concepts**: Encapsulation, inheritance, polymorphism

OOP in Data Science Enhances **code modularity**, **reusability**, and **maintainability** in machine learning pipelines.

Example: Scikit-learn; Scikit-learn models use OOP principles (e.g., fit), predict).

# Functional in Data Science

## A Declarative Approach to Data Processing

**Core FP concepts:** Referential Transperency, immutability, higher-order functions.

Facilitates **parallelism**, **reduces side effects**, and **simplifies complex transformations**.

Example: Pandas & PySpark ; Data manipulation using map()', 'filter()', 'apply()' for scalable processing.

# Functional

## Principles

**Immutability**

**Referential Transperency**

**Higher Order Functions**

# Functional

## Immutability

In functional programming, everything is immutable. This means that **once a variable is set, its value cannot be changed**. If `x=3` at the beginning of a program, x will always have the value of 3 for the rest of the program.

```
x = -5
y = x + 1
```

This code have immutable concept . Why?. Give another example!

## So...

How do we get any work done if we can't change any variable?

# Functional

## Referential Transperency

In functional programming, referentially transparent if we can replace it with its value anywhere in the code. If you can replace the call of a function with its actual value, then the function is referentially transparent

```
def add(a,b):
    return a + b
Print(add(2,3))
```

This code have referential transperency concept . Why?. Give another example!

# Functional

## Higher Order Function

In Functional Programming, functions are first-class citizens, meaning they can be passed as arguments, returned from other functions, and stored in variables. A higher order function is a function that takes another function as an argument or returns a function. Higher order functions also allow us to minimise duplicate code.

```
def apply_twice(func, x):
    return func(func(x))

def increment(num):
    return num + 1

print(apply_twice(increment, 3))
```

This code have referential Higher Order Function concept . Why?. Give the results and another example!

# Functional Task

Analyze this code and explain the concept of functional programming we discussed before (Immutability, referential transparency, and higer-order functions)!

```
def increment_list(numbers):
    return [num + 1 for num in numbers]

def square(x):
    return x * x

def apply_function(func, data):
    return [func(x) for x in data]

# Sample data
numbers = [1, 2, 3, 4, 5]

# Apply functions
incremented_numbers = increment_list(numbers)
squared_numbers = apply_function(square, numbers)

# Print results
print("Original:", numbers)
print("Incremented:", incremented_numbers)
print("Squared:", squared_numbers)
```

# Home Work

Make the source code that demonstrates Immutability, Referential Transparency and Higer Order Function in single running program.

# End Of Week 5

# Obejct-Oriented

## Principles

Inheritance

Encapsulation

polymorphism

# Object Oriented

## Inheritance

Inheritance means that you can extend a class by creating another class that builds on it. This helps reduce repetition, because if you need a new class that's closely related to one you have already written, you don't need to duplicate that class to make a minor change.

## So...

Explain the inheritance method in that code!. Give another example!

```
# Parent class (Base class)
class Hewan:
    def __init__(self, name):
        self.name = name

    def bunyi(self):
        return ”bunyi hewan"
# Child class
class Bebek(Hewan):
    def bunyi(self):
        return "Kwek-kwek!"
# Child class
class Tikus(Hewan):
    def bunyi(self):
        return "Gludug-Gludug!"

Bebek = Bebek("Donald")
Tikus = Tikus("Mickey")

# Same method from parent, different result
depend on clid
print(Bebek.name, ”Bunyi:", Bebek.bunyi())
print(Tikus.name, ”Bunyi:", Tikus.bunyi())
```

# Object Oriented

## Encapsulation

In Encapsulation means that your class hides its details from the outside. You can see only the interface to the class, not the internal details of what's going on. The interface is made up of the methods and attributes that you design. It's not so common in Python, but in other programming languages classes are often designed with hidden or private methods or attributes that can't be changed from the outside.

## So...

Which gives error? ❶ or ❷ . Why?

```
class BankAccount:
    def __init__(self, saldo):
        self.__saldo = saldo

    def deposit(self, jumlah):
        """Increase balance"""
        if jumlah > 0:
            self.__saldo += jumlah
        return self.__saldo

    def cek_saldo(self):
        """Return the current balance"""
        return self.__saldo

account = BankAccount(1000)
account.deposit(500)
#print("Saldo:", account.cek_saldo ❶
#print(account.__saldo) ❷
```

# Object Oriented

## Polymorphism

Polymorphism means that you can have the same interface for different classes, which simplifies your code and reduces repetition. That is, two classes can have a method with the same name that produces a similar result, but the internal workings are different. The two classes can be a parent and child class, or they can be unrelated.

## So...

Where the polymorphism concept in that code?, and Add several shapes with polymorphism concept!

```python
class Bentuk:
    def luas(self):
        return 0

class Persegi(Bentuk):
    def __init__(self, sisi):
        self.sisi = sisi

    def luas(self):
        return self.sisi ** 2

class Lingkaran(Bentuk):
    def __init__(self, jejari):
        self.jejari = jejari

    def luas(self):
        return 3.14 * self.jejari ** 2

bentuk = [Persegi(4), Lingkaran(3)]

for bentuk in bentuk:
    print("Luas:", bentuk.luas())
```

# Object Oriented

## Simple Example

```python
class DataProcessor:
    def __init__(self, data):
        self.data = data  # Store data inside the object

    def clean_data(self):
        """Remove negative values from data"""
        self.data = [x for x in self.data if x >= 0]
        return self.data  # Return processed data

# Example usage
processor = DataProcessor([10, -5, 20, -2, 30])
print(processor.clean_data())  # Output: [10, 20, 30].
```

### Why OOP?

- Ideal Groups related functions (clean_data) **inside a class**.
- Encapsulation: The data is stored **inside the DataProcessor** object.

# Functional

## Simple Example

```
data = [10, -5, 20, -2, 30]

# Functional transformations
clean_data = list(filter(lambda x: x >= 0, data))  # Remove negatives
squared_data = list(map(lambda x: x ** 2, clean_data))  # Square the values

print(squared_data)  # Output: [100, 400, 900]
```

### Why FP?

- Uses **pure functions (map(), filter())** without modifying the original list.
- No side effects, making it **more predictable and reusable**.

# Object Oriented - Functional

## Choosing the Right Paradigm

### OOP Strengths

Best for structured, reusable, and large-scale applications (e.g., machine learning pipelines).

### FP Strengths

Ideal for data transformations, parallel computing, and immutable workflows.

### Hybrid

Many real-world applications mix both paradigms for flexibility and performance.

# Why it's Matter?

## The Role of OOP & FP in Data Science

**Code Organization:** Programming paradigms define how code is structured, improving readability and maintainability.

**Performance Optimization:** Programming paradigms define how code is structured, improving readability and maintainability.

**Scalability in Data Science:** OOP and FP enable scalable solutions for data science, from preprocessing to machine learning.

# When We Use It?

## Choosing the Right Paradigm for Your Use Case

**Use OOP When:** Building large, structured applications like ML model deployment & API'S.

**Use FP When:** Processing large datasets with transformations, parallel computing, and immutability.

**Hybrid Approach:** Many data science projects combine OOP for structure and FP for transformations.

# Case Study

```python
import pandas as pd
from IPython.display import display

class CekHargaBeras:
    def __init__(self, df):
        self.df = df.copy()
        # Mengisi missing value
        self.df['Harga_Beras'].fillna(self.df['Harga_Beras'].mean(), inplace=True)

    def get_data(self):
        return self.df

# Contoh Data
df = pd.DataFrame({
    'Provinsi': ['Jakarta', 'Jawa Barat', 'Jawa Tengah', 'Bali'],
    'Harga_Beras': [12000, None, 11000, 11500]
})

# Process Data
processor = CekHargaBeras(df)
processed_data = processor.get_data()

# Display
display(processed_data)
```

# Case Study

Identify, why this source code adopt OOP concept?. Make the program in FP!

```python
from sklearn.linear_model import LinearRegression

class ModelWrapper:
    def __init__(self):
        self.model = LinearRegression()

    def train(self, X, y):
        """Train the model on the given data"""
        self.model.fit(X, y)
        return self

    def predict(self, X):
        """Make predictions using the trained model"""
        return self.model.predict(X)

# Example usage
X = processed_data[['A']]  # Feature
y = [5, 10, 15, 20]  # Target variable

# Train model
model = ModelWrapper().train(X, y)

# Make predictions
predictions = model.predict(X)
print(predictions)
```

# Home Work

Identify and analyze this code. What concept of OOP/FP used in that code and where?

```python
class MachineLearningPipeline:
    def __init__(self, df, target_column):
        self.df = df
        self.target_column = target_column
        self.processor = DataProcessor(df)
        self.model = ModelWrapper()

    def run(self):
        """Complete ML workflow: preprocess data, train model, predict"""
        processed_df = self.processor.clean_data().scale_features(['A']).get_data()
        X, y = processed_df[['A']], self.df[self.target_column]
        self.model.train(X, y)
        return self.model.predict(X)

# Example usage
df['Target'] = [5, 10, 15, 20]  # Add target column
pipeline = MachineLearningPipeline(df, 'Target')
predictions = pipeline.run()
print("Final Predictions:", predictions)
```

# Home Work

Identify and analyze this code. What concept of OOP/FP used in that code and where?

# End Of Week 6