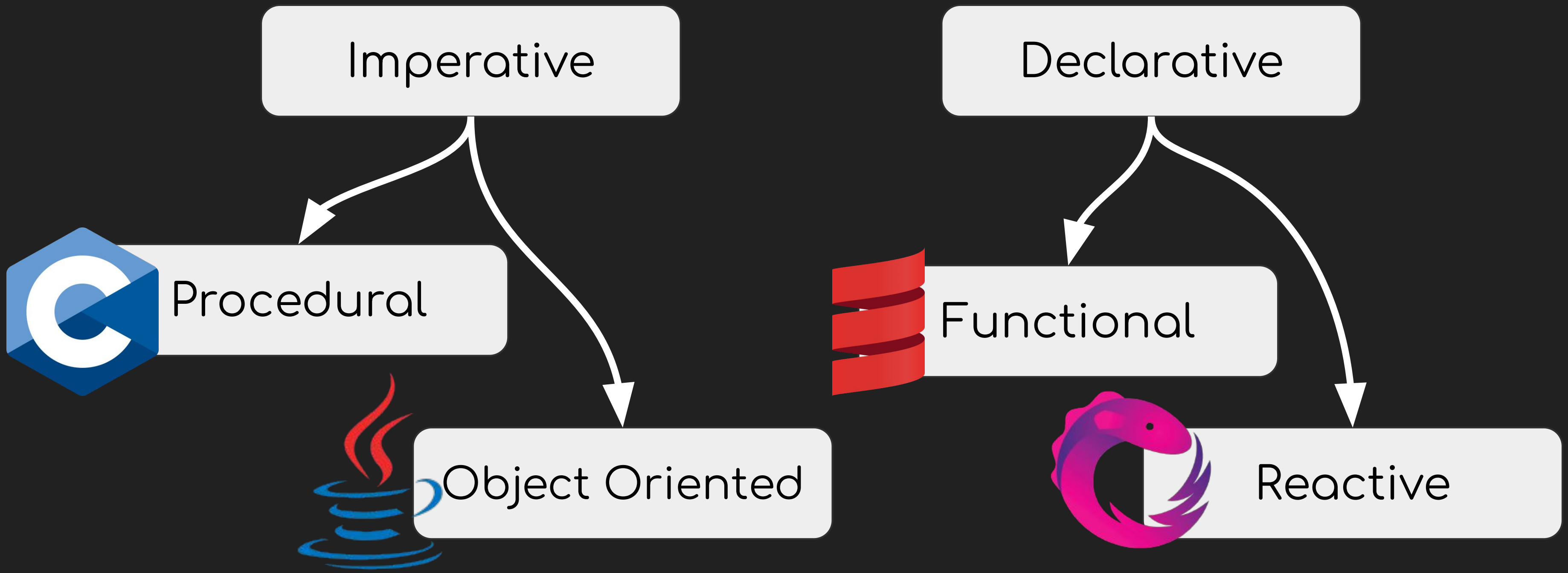
Programming paradigms

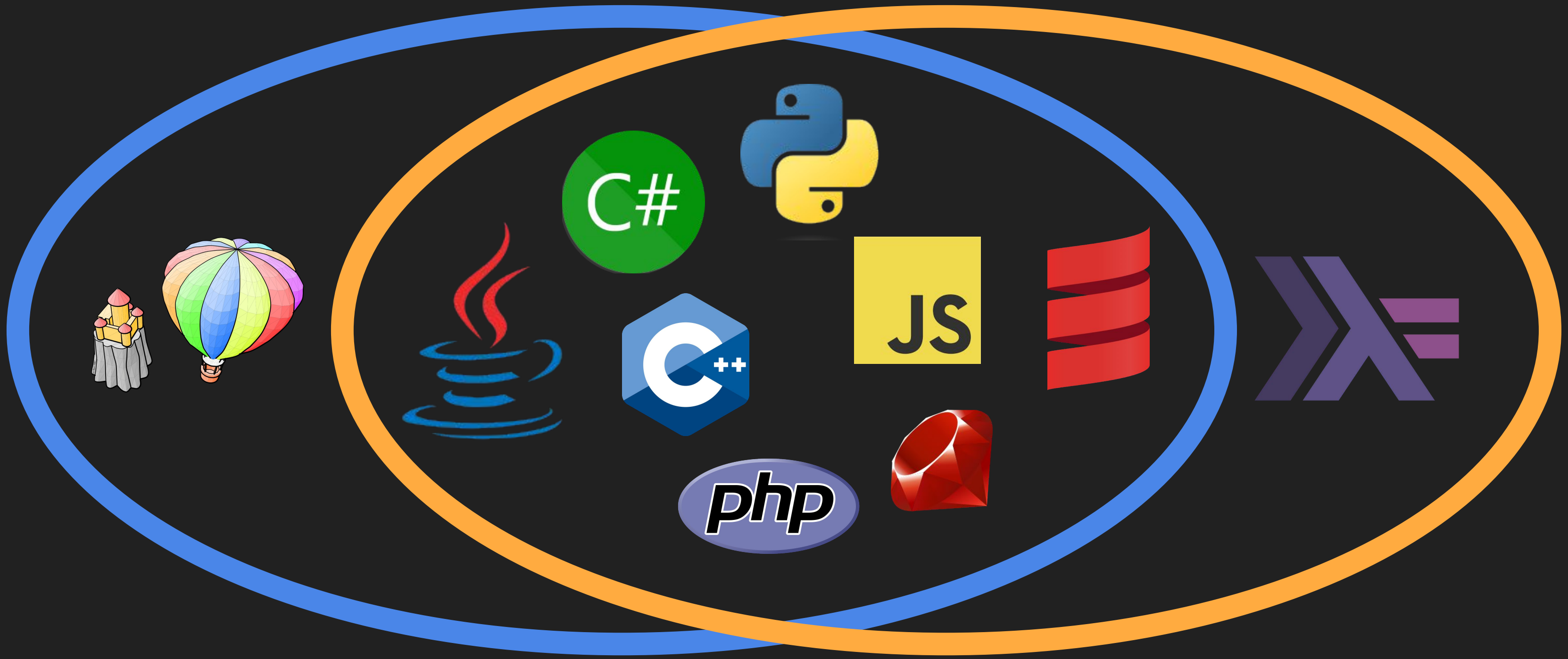
Procedural, OO, Functional, Reactive

Hany ahmed
namozag.com

Common programming paradigms



Languages landscape



Object oriented

Functional

Imperative programming

Programming paradigm that uses statements to change a program's state or commands for the computer to perform which are executed linearly allowing side effects

early (non structured programming)

Control flow using unstructured jumps to labels or instruction addresses (goto statements)

Assembly, windows batch files, early versions of BASIC, Fortran, COBOL

```
@echo off
SET day=Mon

FOR %%D in (Mon Wed Fri)
  do if "%%D" == "%day%" goto SHOP%%D
echo Today, %day%, is not a shopping day.
goto end

:SHOPMon
echo buy pizza for lunch - Monday is Pizza day.
goto end

:SHOPWed
echo buy Calzone to take home - today is
Wednesday.
goto end

:SHOPFri
echo buy Seltzer in case somebody wants ...

:end
```

Structured programming

~~goto~~

Variables
Assignment
Conditionals
Loops
subroutines (functions)

```
int marks[5] = { 12, 32, 45, 13, 19 }
int sum = 0;
float average = 0.0;
for (int i = 0; i < 5; i++) {
    sum = sum + marks[i];
}
average = sum / 5;
```

C COBOL FORTRAN
BASIC

goto → loop

goto	loop

goto

```
int main() {
  int sum = 0;
  int counter = 0;
  int max = 0;
  scanf("%d", &max);
  L:
  sum += ++counter;
  if (counter < max) {
    goto L;
  }
  printf("Sum: %d", sum);
}
```

loop

```
int main() {
  int sum = 0;
  int counter = 0;
  int max = 0;
  scanf("%d", &max);
  while (counter < max) {
    sum += ++counter;
  }
  printf("Sum: %d", sum);
}
```

goto → function

goto	function

goto

```
if (ch == '+) {
  goto addition;
} else if (ch == '-') {
  goto subtraction;
}
goto end;
addition:
sum = a + b;
printf("Result: %d", sum);
goto end;
subtraction:
sum = a - b;
printf("Result: %d", sum);
end:
```

function

```
int main()
{
  if (ch == '+) {
    sum = add(a, b);
  } else if (ch == '-') {
    sum = subtract(a, b);
  } else {
    return 0;
  }
  printf("Result: %d", sum);
  return 0;
}
function add(int x, int y) {
  return x + y;
}
function subtract(int x, int y) {
  return x - y;
}
```

Procedural Programming

procedure routine

subroutine function

```c
int add(int a, int b);

int main() {
  int sum = add(3, 4);
  printf("Sum = %d", sum);
  return 0;
}

int add(int a, int b) {
  return a + b;
}
```

record struct structure

```c
struct Person {
  int id;
  float salary;
};

int main() {
  struct Person person1;
  person1.id = 11;
  person1.salary = 2500;
  printf("Salary: %.2f", person1.salary);
  return 0;
}
```

Object Oriented Programming

Inheritance

Polymorphism

Abstraction

Reusability

```
public class Car {
  public void run() {
     System.out.println(“beep”);
  }
}

public class ModernCar extends Car {
  public void run() {
     System.out.println(“beep beep”);
  }
}
```

Encapsulation

Security

```
public class Employe {
  private String name;
  private double salary;


  public double getSalary() {
    return salary;
  }
  public double getNetSalary() {
    return salary * 0.9;
  }
}
```

Declarative programming

a programming paradigm
that expresses the logic of a computation
without describing its control flow

describing **what** the program must accomplish
rather than describing **how** to accomplish it

Declarative language

Imperative language	Declarative language

```java
double sum = 0;

for (Customer c : customers) {

    if (c.isActive()) {

        sum += c.getBalance();

    }

}
```

```sql
SELECT    SUM(balance)

FROM      accounts

WHERE     active = 1
```

Declarative API

Imperative style

```
List<Coupon> coupons;

public void addCoupons(int amount) {
  for (int i=0; i<amount; i++)
    coupons.add(Util.createCoupon());
}

public void removeCoupons(int amount) {
  int s = coupons.size();
  for (int i=0; i<amount; i++)
    coupons.removeRange(s - amount, s);
}
```

Declarative style

```
List<Coupon> coupons;

public void setCount(int desired) {
  int actual = coupons.size();
  if (desired < actual)
    addCoupons(desired - actual);
  if (desired > actual)
    removeCoupons(desired - actual);
}
```

Declarative API: Java streams

Imperative style	Declarative/Functional style

Imperative style

```
double sum = 0;

for (Customer c : customers) {

    if (c.isActive()) {

        sum += c.getBalance();

    }

}
```

Declarative/Functional style

```
double sum =

    customers.stream()

    .filter(c -> c.isActive())

    .mapToDouble(c -> c.getBalance())

    .sum();
```

Functional programming

a programming paradigm

where programs are constructed

by applying and composing functions

FP Principles

Function as first-class citizen

Pure function

Referential Transparency

Immutable variables

Recursion

Function as first-class citizen

(First class function)

1. Assigning functions to variables
2. Functions as arguments
3. Function as a return type

Assigning functions to variables (JS)

```
//Assign function
const add = (a, b) => a + b;

//Use function
console.log(add(3, 4));
```

Function as arguments (JS)

```
//Receive function as argument
function register(name, callback) {
    // some logic here
    callback(name + '@domain.com');
}

function sendWelcomeMail(mail) {
    console.log('MAILING ' + mail);
}

//Send function as argument
register('hany', sendWelcomeMail);
```

Function as a return type (JS)

```
function multiplyBy(x) {
  //Return function
  return function(y) {
    return x * y;
  };
}
```

```
doubleNumber = multiplyBy(2);
tripleNumber = multiplyBy(3);

console.log(doubleNumber(4));
console.log(tripleNumber(4));
console.log(multiplyBy(5)(6));
```

Currying

Transforms a function that takes multiple arguments into a chain of functions each with a single argument

```
function multiply(x, y) {
     return x * y;
}
```

```
function multiplyBy(x) {
  //Return function
  return function(y) {
    return x * y;
  };
}

multiplyBy(5)(6)
```

Pure function

a function that

produces the same output for the same input
without side effects

Pure function

pure	impure

```
int add(int a, int b) {
    return a + b;
}

add(3, 4);        // 7
add(3, 4);        // 7
```

```
int value = 0;
int accumulate(int amount) {
    // depending on state
    return value + amount;
}
accumulate(2);    // 2
accumulate(2);    // 4
```

Side effect

a function reads/writes something outside its own arguments

```
User createUser(String name) {
  User u = new User();
  u.setName(name);
  u.setActivationCode(UUID.create());     // non predictive output
  u.setCreationDate(new Date());          // non predictive output
  u.setCreator(this.loggedInUser);        // Read variable outside arguments
  userRepo.save(u);                       // Write into database
  log.debug(“User created {}”, u);        // Write into console or file
  return u;
}
```

Purity & Idempotency

pure

```
int add(int a, int b) {

    return a + b;

}


add(3, 4);              // 7

add(3, 4);              // 7
```

idempotent

```
int add(int a, int b) {
    int sum = a + b;
    // side effect
    cache(a + “+” + b, sum);
    return sum
}
                  output    cache
                            null
add(3, 4);        // 7      cache 7
add(3, 4);        // 7      cache 7
```

not idempotent

```
int add(int a, int b) {
    int sum = a + b;
    // side effect
    save(a + “+” + b, sum);
    return sum
}
                  output    db
                            null
add(3, 4);        // 7      1 rec
add(3, 4);        // 7      2 recs
```

Referential Transparency

Ability to replace a function call with its output value without changing the program's behavior

referentially transparent

```
double calculateTax(Order o, int ratio)
{
        return o.amount * ratio;
}

double tax = calculateTax(order, 0.1);
```

referentially opaque

```
double ratio = 0.1;

double calculateTax(Order o) {
     return o.amount * ratio;
}

double tax = calculateTax(order);
```

```
double calculateTax(Order o) {
     return o.amount * getRationFromDb();
}

double tax = calculateTax(order);
```

Immutability

immutable	mutable

immutable

```
toCaps(List<String> col) {
    capList = new ArrayList<>();
    for (String s : col) {
        capList.add(s.toUpper());
    }
    return capList;
}

in = Arrays.asList("red", "green");
// protecting the input object
result = toCaps(ImmutableList.copyOf(in));
// input list       [red, green]
// result list      [RED, GREEN]
```

mutable

```
toCaps(List<String> col) {
  for (int i=0; i<col.size(); i++) {
    // mutating the input object
    col.set(i, col.get(i).toUpper());
  }
  return col;
}

in = Arrays.asList("red", "green");
result = toCaps(in);
// input list      [RED, GREEN]
// result list     [RED, GREEN]
```

Functional

Imperative	Functional

```
const nums = [1, 2, 3, 4, 5];
let result = 0;
for (let i = 0; i < nums.length; i++) {
    if (nums[i] % 2 === 0) {
        result += numList[i] * 10;
    }
}
```

```
const result = [1, 2, 3, 4, 5]
    .filter(n => n % 2 === 0)
    .map(a => a * 10)
    .reduce((a, b) => a + b);
```

Composing functions

imperative	functional

```
const numbers = [1,2,3,4,5,6,7,8,9,10]
sum = 0;

for (i=0; i<numbers.length; i++) {

    n = numbers[i];

    if (n % 2 == 0) {

        d = n * 2;

        sum += d;

    }

}
```

```
isEven = n => n % 2 == 0;
doubleNumber = n => n * 2;
sumNumbers = (a, b) => a + b;

[1,2,3,4,5,6,7,8,9,10]
      .filter(n => isEven(n))
      .map(n => doubleNumber(n))
      .reduce((a, b) =>
            sumNumbers(a, b))
```

```
[1,2,3,4,5,6,7,8,9,10]
     .filter(n => n % 2 == 0)
     .map(n => n * 2)
     .reduce((a, b) =>
            a + b)
```

Benefits

Easy to read/understand

Easy to remove/replace

Increases reusability

Easy to test

Easy to troubleshoot in most cases

More to trust

Safe to run in parallel

Reactive programming

a declarative programming paradigm
concerned with data streams
and the propagation of change

Issue with the imperative style

```
Product product = productRepo.findProuct(productId);                  // 1s


List<Review> reviews = reviewApiClient.findReviews(productId);        // 1s


for (Review r : reviews) {

    Customer c = customerApiClient.findCustomer(r.customerId);        // 1s

}
```

Going async

```
CompletableFuture<String> productFuture =
  CompletableFuture.supplyAsync(() ->
    productRepo.findProduct(productId));               // 1s



CompletableFuture<List<Review>> reviewsFuture =
  CompletableFuture.supplyAsync(() ->
    reviewApiClient.findReviews(productId));           // 1s parallel
```