Int. to SP. Note Summary

Yucheng Wu (吴禹承) 25.8.4

Int. to System Programming Notes Directory

1. Compilation & Execution

- **C Compilation Flow**
  - Preprocessing (.i via cpp)
  - Compilation (.s via cc1)
  - Assembly (.o via as)
  - Linking (executable via ld, using libraries like stdlib.so)
- **Java Compilation & JVM**
  - Compilation with javac (.java → .class bytecode)
  - JVM execution with dynamic class loading and type checking
- **Python Interpreted Flow**
  - Sequential interpretation of code
  - Function definitions and executions
  - Variable assignments and outputs
- **Error Handling in Compiled vs. Interpreted Languages**

---

2. Memory Management

- **Memory Regions**
  - Heap
  - Stack
  - Static data segment
  - Shared memory
- **Memory Management Models**
  - Manual:
    - C (malloc / free)
    - C++ (new / delete)
  - Garbage Collection:
    - Java (mark-and-sweep)
    - Python (reference counting + cycle detection)

---

3. Type Systems

- **Static vs Dynamic Typing**

- **General Primitive Types**
  - Basic types such as int, char, double, bool
- **Type Checking**
  - Compile-time (Java, C++) vs runtime (Python)

---

4. Functions & Parameter Passing

- **Cross-language Parameter Passing Models**
  - Pass-by-value
  - Pass-by-reference
  - Pass-by-assignment

---

5. Collections & Abstract Data Types (ADTs)

- **Python Collections**
  - list: ordered, mutable
  - tuple: ordered, immutable
  - set: unordered, no duplicates
  - dict: ordered, changeable, no duplicates
- **General ADTs**
  - Lists (linear sequences)
  - Sets (unordered unique elements)
  - Stacks (FILO)
  - Queues (FIFO)
- **Modularity and Encapsulation**
  - Java: Packages and Classes
  - C++: Header files (.hpp) and source files (.cpp)
  - Python: Modules (files) and namespaces

---

6. Object-Oriented Programming (OOP)

- **Introduction to Object-Oriented Programming**
- **Classes and Objects**
- **Inheritance, Encapsulation, and Abstract Classes**
- **The** this **/** self **Keyword**

---

7. Generic Programming

- **C++ Templates**
  - Compile-time type-specific code generation

    - Designing Generic Functions with Templates
- **Java Generics**
    - Static type-checked generics (reference types only)
- **Python Duck Typing**
    - Dynamic typing with no enforced generic restrictions

---

8. Language-Specific Features

- **Java Features**
    - JVM, JIT compilation, file structure (one public class per file)
- **Python Features**
    - Interpreted language
    - Uses if **name** == "**main**" to differentiate module vs script execution
- **C++ Features**
    - Multi-paradigm (procedural + OOP)
    - Operator overloading
    - Manual memory management (new / delete)
- Summary

1.Compilation and Execution

1.1. C Compilation Flow

Definition

The process of converting C source code into an executable binary file, involving preprocessing, compilation, assembly, and linking.

Feature

- Converts human-readable `.c` files into machine-executable files.
- Produces intermediate files: `.i`, `.s`, `.o`.
- Uses external libraries (e.g. `stdlib.so`) during linking.

Syntax

```
gcc hello.c -o hello
```

Example

Step-by-step compilation of `hello.c`:

1. **Preprocessing**
   Command: `cpp hello.c > hello.i`
   Action: Handles `#include`, `#define`, and macro expansions.

   Macro Expansion Example:

   ```
   #define PI 3.14
   float area = PI * r * r;
   ```

   After macro expansion:

   ```
   float area = 3.14 * r * r;
   ```

   Explanation:
   Macros are replaced with their defined values before compilation to simplify and reuse code efficiently.
2. **Compilation**
   Command: `cc1 hello.i -o hello.s`
   Action: Converts preprocessed code into **assembly code**.

   Assembly Code Explanation:
   - Assembly is a low-level language close to machine code but still readable by humans familiar with CPU instructions.
   - Example:

     ```
     mov eax, 5
     add eax, 2
     ```
3. **Assembly**
   Command: `as hello.s -o hello.o`
   Action: Converts assembly code into **machine code object files** (`.o`).
4. **Linking**
   Command: `ld hello.o -o hello`
   Action: Links object files and libraries to produce the final executable.

Explanation

- Each stage outputs a file used by the next stage.
- `gcc hello.c -o hello` runs all stages automatically.
- Linking errors such as “undefined reference” occur if required libraries or function definitions are missing.

Remark

- `.i`: preprocessed C code with macros expanded and headers included.
- `.s`: assembly code generated from `.i`.
- `.o`: machine code, not human-readable.
- Knowing these stages helps debug compilation issues precisely.

1.2.Java Compilation & JVM

Definition

Java source code is compiled into bytecode, which is executed by the Java Virtual Machine (JVM).

Feature

- `javac` compiles `.java` files to `.class` bytecode files.
- JVM loads and executes bytecode.
- Supports **dynamic class loading** and runtime type checking.

Syntax

```
javac HelloWorld.java
java HelloWorld
```

Example

```
// HelloWorld.java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Steps:

1. Compile: `javac HelloWorld.java` → generates `HelloWorld.class` bytecode.
2. Run: `java HelloWorld` → JVM executes bytecode.

Explanation

- **Bytecode Definition:**
  Intermediate Java code between source code and machine code.
- **Why Bytecode is Platform-Independent:**
  - Bytecode is not tied to any CPU or OS.
  - JVM is platform-specific and translates bytecode to the native machine code for that system.
  - This enables Java's **"write once, run anywhere"** feature.
- **Dynamic Class Loading Explanation:**
  The JVM loads classes into memory only when needed during execution. For example:

```
MyClass obj = new MyClass();
```

If `MyClass` has not been loaded yet, the JVM loads it at this point.

Remark

- Java's portability is due to the JVM executing bytecode on any platform.
- Compile-time checks are performed by `javac`; runtime checks by the JVM.
- JVM uses **Just-In-Time (JIT) compilation** to convert frequently used bytecode into machine code at runtime for better performance.

1.3.Python Interpreted Flow

Definition

Python code is executed line by line by the interpreter at runtime.

Feature

- No need to compile before execution.
- Interpreter executes statements sequentially.
- Supports REPL (Read-Eval-Print Loop) for interactive coding.

Syntax

```
python hello.py
```

Example

```
# hello.py
print("Hello, World!")
```

Run:

```
$ python hello.py
Hello, World!
```

Explanation

- Python reads and executes each statement immediately.
- **Function definitions** are read and stored but only executed when called.
- **Bytecode in Python:**
  Although Python is interpreted, it compiles code into bytecode (`.pyc`) before executing.
  `.pyc` files are stored in `__pycache__` to speed up subsequent executions.

Remark

- Python bytecode is specific to CPython's interpreter and not platform-independent like Java's bytecode.
- Understanding the interpreted nature helps optimize code execution order and performance.

1.4. Error Handling in Compiled vs. Interpreted Languages

Definition

Error handling in compiled vs. interpreted languages refers to how programming languages detect and manage errors, such as undefined functions, during the compilation or execution process. Compiled languages (e.g., C++) check for errors at compile time, while interpreted languages (e.g., Python) detect errors at runtime.

Feature

- Compile-time error detection in compiled languages prevents execution if errors like undefined functions are found.
- Runtime error detection in interpreted languages allows execution until the erroneous code is encountered.
- Common errors include undefined function calls, type mismatches, and syntax errors.

Syntax

```cpp
// C++ (Compiled Language)
int bar(); // Declaration needed to avoid compile-time error
int foo(int a) {
    if (a > 50) {
        return bar(); // Compile-time error if bar() is undefined
    }
    return 42;
}
```

```python
# Python (Interpreted Language)
def foo(a):
    if a > 50:
        return bar() # No error until runtime if bar() is undefined
    return 42
```

Example

- C++ Example:

```cpp
#include <iostream>
int bar() { return 100; }
int foo(int a) {
    if (a > 50) {
        return bar();
    }
    return 42;
}
int main() {
    std::cout << foo(63) << std::endl; // Outputs 100
    std::cout << foo(1) << std::endl;  // Outputs 42
}
```

- Python Example:

```python
def bar():
    return 100
def foo(a):
    if a > 50:
        return bar()
    return 42
print(foo(63)) # Outputs 100
print(foo(1))  # Outputs 42
print(foo(63)) # Runtime error if bar() is undefined
```

Explanation

- In C++, the compiler checks for function definitions or declarations before generating the executable. If bar() is undefined, a compile-time error occurs, preventing execution.
- In Python, the interpreter executes code line-by-line. An undefined bar() raises a NameError only when the function is called (e.g., when a > 50).
- This difference affects debugging: compiled languages catch errors early, while interpreted languages require runtime testing to identify issues.

Remark

- Compiled languages like C++ are stricter, ensuring robust code before execution but requiring complete definitions.
- Interpreted languages like Python offer flexibility but may delay error detection, necessitating thorough testing.
- Understanding these differences is crucial for cross-language development.

2.Memory Management

2.1.Memory Regions

Definition

Different parts of memory are used for different purposes when a program runs.

Feature

- Each region has its own role in storing data.
- Helps organize program execution efficiently.

Syntax

(No direct syntax – these are runtime concepts.)

Example

The main memory regions are:

1. **Heap**
   - Used for **dynamic memory allocation** (e.g. objects created with `new` in Java, `malloc` in C).
   - Grows upward (towards higher addresses).
   - Managed by the programmer (C/C++) or garbage collector (Java/Python).
2. **Stack**
   - Stores **function call frames** (local variables, function parameters, return addresses).
   - Grows downward (towards lower addresses).
   - Automatically managed: when a function exits, its stack frame is removed.
3. **Static Data Segment**
   - Stores **global variables** and **static variables**.
   - Exists throughout the program's lifetime.
4. **Shared Memory**
   - A memory region accessible by **multiple processes**.
   - Used in inter-process communication (IPC) to share data efficiently.

Explanation

- **Heap vs Stack:**
  Heap is for dynamic memory with manual or garbage collection management. Stack is for function-related data with automatic allocation and deallocation.
- **Static Data Segment:**
  Holds variables that keep their value across function calls or the whole program run.
- **Shared Memory:**
  Used when different programs need to access the same data without using files.

Remark

- Stack overflow occurs if too many nested function calls exceed stack size.
- Memory leaks happen if dynamically allocated heap memory is not freed in C/C++.
- Java and Python handle heap memory with garbage collection to avoid leaks.

2.2.Memory Management Models

Definition

Different programming languages use different ways to manage memory allocation and deallocation.

Feature

- **Manual Memory Management:** Programmer controls allocation and freeing of memory.
- **Garbage Collection:** The language runtime automatically finds unused memory and frees it.

Syntax

C (Manual)

```c
int *p = malloc(sizeof(int)); // allocate memory
*p = 10;
free(p); // free memory
```

C++ (Manual)

```cpp
int *p = new int; // allocate memory
*p = 10;
delete p; // free memory
```

Java (Garbage Collection)

```java
Integer x = new Integer(10); // allocate memory
// No explicit free; GC reclaims unused memory
```

Python (Garbage Collection)

```python
x = 10 # allocate memory
# Unused objects are automatically deleted
```

Example

Language	Allocation	Deallocation	Management Model
C	`malloc`	`free`	Manual
C++	`new`	`delete`	Manual
Java	`new`	Automatic GC	Garbage Collection

Language	Allocation	Deallocation	Management Model
Python	Assignment	Automatic GC	Garbage Collection

Explanation

- **Manual (C/C++):**
  You must allocate and free memory yourself. If you forget to free, it causes memory leaks. If you free twice or use after free, it causes errors.
- **Garbage Collection (Java/Python):**
  The runtime automatically finds objects no longer used and reclaims their memory.
- **Java GC:**
  Uses **mark-and-sweep** to find objects not reachable and deletes them.
- **Python GC:**
  Uses **reference counting** (counts how many references point to an object; deletes when zero) plus **cycle detection** to handle objects referring to each other.

Remark

- Manual memory management gives **more control** but higher risk of bugs.
- Garbage collection **reduces programmer burden** but may cause unpredictable pauses (e.g. during GC in Java).
- Understanding memory models helps write efficient and safe programs.

3.Type Systems

3.1.Static vs Dynamic Typing

Definition

A type system defines how a language handles data types. Static typing checks types at compile-time, while dynamic typing checks types when the program runs.

Feature

- Static typing detects type errors before running the program.
- Dynamic typing is flexible, but type errors only show up at runtime.

Syntax

```
// Java (Static Typing)
int x = 10;
x = "hello"; // Error at compile-time
```

```
// C++ (Static Typing)
double y = 5.5;
y = "test"; // Error at compile-time
```

```
# Python (Dynamic Typing)
z = 100
z = "now a string" # No error until z is used in an incompatible way
```

Example

- Java:

```
int a = 5;
a = "string"; // Compile-time error
```

- C++:

```
char c = 'A';
c = 10; // Allowed, implicit conversion from int to char
```

- Python:

```
b = 3.14
b = True # Allowed, type changes dynamically
```

Explanation

- In **static typing**, the compiler checks variable types before execution, preventing type-related bugs early.
- In **dynamic typing**, types are determined when the program runs, making coding faster but riskier for large systems.

Remark

- Static typing (Java, C++) is safer for large projects.
- Dynamic typing (Python) makes code shorter and easier to write for small scripts.

3.2.General Primitive Types

Definition

Primitive types are the most basic data types that store simple values like numbers or characters.

Feature

- They directly represent data in memory.
- No complex structure or methods attached.

Syntax

```java
int age = 20;
char grade = 'A';
double weight = 55.8;
boolean passed = true;
```

```cpp
int n = 100;
char letter = 'B';
double pi = 3.14159;
bool isReady = false;
```

```python
num = 42    # int
char = 'C'  # str of length 1
pi = 3.14   # float
flag = True # bool
```

Example

- Java:

```java
boolean isOpen = false;
int count = 15;
```

- C++:

```cpp
double temp = -5.5;
char symbol = '#';
```

- Python:

```python
is_valid = True
height = 170.2
```

Explanation

- Primitive types store simple, non-divisible values.
- They use minimal memory and are built into the language.

Remark

- In Java and C++, primitives are different from objects.
- In Python, everything is an object, including primitive-looking types.

3.3.Type Checking

Definition

Type checking is how a language ensures variables and operations use compatible types. It can happen at compile-time (static) or at runtime (dynamic).

Feature

- Compile-time checking prevents many errors early.
- Runtime checking allows more flexible coding.

Syntax

```
// Compile-time type checking (Java)
String s = "hello";
s = 10; // Error
```

```
// Compile-time type checking (C++)
int x = 3.5; // Warning: truncates to 3
```

```
# Runtime type checking (Python)
x = "hi" + 5 # Error only when executed
```

Example

- Java:

```
double d = "test"; // Compile-time error
```

- C++:

```
int i = 3.7; // Compiles with warning, value becomes 3
```

- Python:

```
def add(x, y):
    return x + y

print(add(5, "hi")) # Runtime TypeError
```

Explanation

- **Java and C++** check types during compilation; programs with type errors do not compile.
- **Python** checks types when the line runs; errors appear only then.

Remark

- Static type checking improves safety and readability in big projects.
- Dynamic type checking is simpler for scripting and rapid development.

4.Functions & Parameter Passing

4.1.Functions

Definition

Functions are reusable blocks of code that perform specific tasks and can return a value.

Feature

- Can take inputs (parameters)
- May return outputs (return value)
- Improve code reusability and readability

Syntax

```
// Java function
returnType functionName(parameterType parameterName) {
    // code
    return value;
}
```

```
// C++ function
returnType functionName(parameterType parameterName) {
    // code
    return value;
}
```

```
# Python function
def function_name(parameter):
    # code
    return value
```

Example

- **Java:**

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

- **C++:**

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

- **Python:**

```
def add(a, b):
    return a + b
```

Explanation

- All three define a function `add` that takes two parameters and returns their sum.

Remark

- Functions reduce repeated code.
- Function names should describe their purpose clearly.

4.2.Cross-language Parameter Passing Models

Definition

Parameter passing determines how arguments are sent to functions.

4.2.1.Pass-by-Value

Feature

- A copy of the argument is passed.
- Changes in the function do not affect the original variable.

Syntax

```
void foo(int x) {
    x = 10;
}
```

```
void foo(int x) {
    x = 10;
}
```

```
def foo(x):
    x = 10
```

Example

- **Java:**

```
int a = 5;
foo(a);
// a is still 5
```

- **C++:**

```
int a = 5;
foo(a);
// a is still 5
```

- **Python:**

```
a = 5
foo(a)
# a is still 5 (for immutable types like int)
```

Explanation

- Java and C++ pass primitive types by value.
- Python immutable types (int, str, tuple) act like pass-by-value as reassignment inside the function does not affect the original.

Remark

- Pass-by-value avoids unintended changes to the original variable.

4.2.2.Pass-by-Reference

Feature

- The reference (address) of the argument is passed.
- Changes in the function affect the original variable.

Syntax

```cpp
// C++ pass-by-reference
void foo(int &x) {
    x = 10;
}
```

```java
// Java uses references for objects
void updateArray(int[] arr) {
    arr[0] = 100;
}
```

```python
# Python mutable objects
def update(lst):
    lst[0] = 100
```

Example

- **C++:**

```cpp
int a = 5;
foo(a);
// a is now 10
```

- **Java:**

```java
int[] nums = {1,2,3};
updateArray(nums);
// nums[0] is now 100
```

- **Python:**

```python
nums = [1,2,3]
update(nums)
# nums[0] is now 100
```

Explanation

- C++ explicitly uses `&` to pass by reference.
- Java passes object references by value, but the object itself can be modified inside the function.
- Python function parameters for mutable types like lists behave similarly.

Remark

- Useful for modifying data structures within functions.

4.2.3.Pass-by-Assignment

Feature

- Python uses pass-by-assignment:
  - The function parameter becomes a new reference to the same object.
  - Rebinding does not change the original, but modifying the object does if it is mutable.

Syntax

```
def foo(x):
    x = [10,20]
```

Example

- **Python:**

```
a = [1,2,3]
foo(a)
# a is still [1,2,3] because x was rebound to a new list

def modify(x):
    x[0] = 99

nums = [1,2,3]
modify(nums)
# nums is now [99,2,3] because the list object itself was modified
```

Explanation

- When `x[0] = 99` is used, it modifies the original list.
- When `x = [10,20]` is used, it only rebinds `x` locally without affecting the original.

Remark

- Understanding Python's assignment model avoids unexpected side effects in functions.

5.Collections & Abstract Data Types (ADTs)

5.1.Python Collections

Definition

Python collections are built-in data structures to store multiple items, like lists, tuples, sets, and dictionaries, each with specific properties.

Feature

- `list` : Ordered, can be changed, allows duplicates.
- `tuple` : Ordered, cannot be changed, allows duplicates.
- `set` : Unordered, only unique items.
- `dict` : Ordered (since Python 3.7), key-value pairs, no duplicate keys.

Syntax

```
# List
my_list = [1, 2, 3]

# Tuple
my_tuple = (4, 5, 6)

# Set
my_set = {7, 8, 9}

# Dict
my_dict = {"name": "Alice", "age": 25}
```

Example

- List:

```
fruits = ["apple", "banana"]
fruits.append("orange")  # Add item
print(fruits)  # ["apple", "banana", "orange"]
```

- Tuple:

```
point = (10, 20)
print(point[0])  # 10
```

- Set:

```
numbers = {1, 2, 2}  # Duplicate 2 ignored
print(numbers)  # {1, 2}
```

- Dict:

```
person = {"name": "Bob", "age": 30}
print(person["name"])  # Bob
```

Explanation

- **List**: Stores items in order, can add or remove items, accessed by index (e.g., `fruits[0]`).
- **Tuple**: Like a list but fixed; cannot change after creation, good for constant data.
- **Set**: Stores unique items, no order, useful for removing duplicates or checking membership.
- **Dict**: Maps keys to values, like a phonebook, for quick lookups by key.

Remark

- Lists and dictionaries are flexible but can be slower for large data.
- Tuples are faster than lists because they cannot change.
- Sets are great for unique items but don't keep order.

5.2.General ADTs

Definition

Abstract Data Types (ADTs) describe data structures and their operations without specifying how they're built, like lists, sets, stacks, and queues.

Feature

- **Lists**: Ordered sequence, allows duplicates, supports adding/removing.
- **Sets**: Unordered, only unique items, supports operations like union.
- **Stacks**: First-In-Last-Out (FILO), like a stack of plates.
- **Queues**: First-In-First-Out (FIFO), like a line at a store.

Syntax

```
// Java: List and Stack
import java.util.ArrayList;
import java.util.Stack;

ArrayList<Integer> list = new ArrayList<>();
Stack<Integer> stack = new Stack<>();
```

```
// C++: Vector and Queue
#include <vector>
#include <queue>
```

```cpp
std::vector<int> vec;
std::queue<int> q;
```

```python
# Python: List as Stack or Queue
stack = []  # Use as stack
queue = []  # Use as queue
```

Example

- List (Java):

```java
ArrayList<Integer> nums = new ArrayList<>();
nums.add(5);  // Add 5
System.out.println(nums.get(0));  // 5
```

- Set (C++):

```cpp
#include <set>
std::set<int> s;
s.insert(3);  // Add 3
s.insert(3);  // Ignored
std::cout << s.size();  // 1
```

- Stack (Python):

```python
stack = []
stack.append(1)  # Push 1
stack.pop()      # Remove 1
print(stack)     # []
```

- Queue (Python):

```python
queue = []
queue.append(2)  # Add 2
queue.pop(0)     # Remove 2
print(queue)     # []
```

Explanation

- **Lists**: Store items in order, like a shopping list, can grow or shrink.
- **Sets**: Store only unique items, like a set of IDs, no duplicates allowed.
- **Stacks**: Add and remove from the top, used in undo features.
- **Queues**: Add at the end, remove from the front, used in task scheduling.

Remark

- Lists are versatile but may use more memory than arrays.

- Sets are efficient for checking if an item exists.
- Stacks and queues can be built using lists in Python, but Java/C++ have specific classes.

5.3.Modularity and Encapsulation

Definition

Modularity organizes code into separate units (e.g., files or packages) to make it reusable. Encapsulation hides data details to protect them from misuse.

Feature

- **Java**: Uses packages to group classes, hides data with `private` keyword.
- **C++**: Uses header (`.hpp`) and source (`.cpp`) files to separate code.
- **Python**: Uses modules (`.py` files) to organize code, namespaces avoid conflicts.

Syntax

```java
// Java: Package and class
package mypackage;
public class MyClass {
    private int data = 10;
    public int getData() { return data; }
}
```

```cpp
// C++: Header file (myclass.hpp)
#ifndef MYCLASS_H
#define MYCLASS_H
class MyClass {
private:
    int data;
public:
    MyClass();
    int getData();
};
#endif

// Source file (myclass.cpp)
#include "myclass.hpp"
MyClass::MyClass() { data = 10; }
int MyClass::getData() { return data; }
```

```python
# Python: Module (mymodule.py)
data = 10
def get_data():
    return data
```

Example

- Java:

```java
package example;
public class Test {
    private String name = "Alice";
    public String getName() { return name; }
}
```

- C++:

```cpp
// test.hpp
#ifndef TEST_H
#define TEST_H
class Test {
private:
    int value;
public:
    Test();
    int getValue();
};
#endif

// test.cpp
#include "test.hpp"
Test::Test() { value = 5; }
Test::getValue() { return value; }
```

- Python:

```python
# mymodule.py
value = 20
def get_value():
    return value

# main.py
import mymodule
print(mymodule.get_value())  # 20
```

Explanation

- **Java**: Packages group related classes (e.g., `mypackage`). Encapsulation uses `private` to hide data, accessed via public methods.
- **C++**: Header files declare what a class does, source files define how. `#ifndef` prevents duplicate declarations.
- **Python**: Modules are `.py` files. Importing them creates a namespace (e.g., `mymodule.value`) to avoid name conflicts.

Remark

- Java's packages are strict, requiring one public class per file.
- C++ headers need careful management to avoid errors like redefinition.
- Python modules are simple to create but less strict, which can lead to naming issues.

6.Object-Oriented Programming (OOP)

6.1.Introduction to Object-Oriented Programming (OOP)

Definition

OOP is a programming approach where programs are designed using objects that combine data and methods.

Feature

- Uses concepts like class, object, inheritance, and polymorphism.
- Helps organize complex programs by modelling real-world entities.
- Java, Python and C++ can support OOP

Language	Supports OOP	Keywords/Concepts Used
Java	Yes	class, object, package, inheritance
Python	Yes	class, object, self, **init**
C++	Yes	class, object, public/private, constructor

Explanation

- Class defines a blueprint.
- Object is an instance of a class.
- OOP groups related data and functions together for better structure.

Remark

- Java is primarily an OOP language.
- Understanding OOP helps write clearer, reusable code.

6.1.1.Python OOP

Syntax

```
class Dog:
    def __init__(self, name):
```

```
        self.name = name

    def bark(self):
        print(self.name + " barks")

d = Dog("Buddy")
d.bark()
```

Example

```
class Student:
    def __init__(self, name, id):
        self.name = name
        self.id = id

    def introduce(self):
        print("My name is", self.name, "and my id is", self.id)

s = Student("Alice", "A001")
s.introduce()
# Output: My name is Alice and my id is A001
```

6.1.2.C++ OOP

Syntax

```
#include <iostream>
using namespace std;

class Dog {
public:
    string name;

    void bark() {
        cout << name << " barks" << endl;
    }
};

int main() {
    Dog d;
    d.name = "Buddy";
    d.bark();
    return 0;
}
```

Example

```
#include <iostream>
using namespace std;
```

```cpp
class Student {
public:
    string name;
    int id;

    void introduce() {
        cout << "My name is " << name << " and my id is " << id << endl;
    }
};

int main() {
    Student s;
    s.name = "Alice";
    s.id = 1001;
    s.introduce();
    // Output: My name is Alice and my id is 1001
    return 0;
}
```

6.1.3.Java OOP

Syntax

```java
class Dog {
    String name;
    void bark() {
        System.out.println(name + " barks");
    }
}
```

Example

```java
Dog d = new Dog();
d.name = "Buddy";
d.bark(); // prints "Buddy barks"
```

6.2.Classes and Objects

Definition

A class is a blueprint for creating objects, which are instances of the class. Objects hold data (attributes) and behavior (methods) defined by the class.

Feature

- Classes define the structure and behavior of objects.
- Objects are created from classes and store specific data.
- Classes support code reuse and organization.

Syntax

```java
// Java
class Car {
    String model;
    int speed;

    void drive() {
        System.out.println(model + " is driving at " + speed + " mph");
    }
}

Car myCar = new Car(); // Creating an object
myCar.model = "Toyota";
myCar.speed = 60;
myCar.drive();
```

```cpp
// C++
class Car {
public:
    string model;
    int speed;

    void drive() {
        cout << model << " is driving at " << speed << " mph" << endl;
    }
};

Car myCar; // Creating an object
myCar.model = "Honda";
myCar.speed = 70;
myCar.drive();
```

```python
# Python
class Car:
    def __init__(self, model, speed):
        self.model = model
        self.speed = speed

    def drive(self):
        print(f"{self.model} is driving at {self.speed} mph")

my_car = Car("Ford", 50) # Creating an object
my_car.drive()
```

Example

- Java:

```java
class Student {
    String name;
    int age;

    void introduce() {
        System.out.println("I am " + name + ", age " + age);
    }
}

Student s1 = new Student();
s1.name = "Alice";
s1.age = 20;
s1.introduce();
```

- C++:

```cpp
class Student {
public:
    string name;
    int age;

    void introduce() {
        cout << "I am " << name << ", age " << age << endl;
    }
};

Student s1;
s1.name = "Bob";
s1.age = 22;
s1.introduce();
```

- Python:

```python
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        print(f"I am {self.name}, age {self.age}")

s1 = Student("Charlie", 21)
s1.introduce()
```

Explanation

- A **class** is like a template that defines what data an object can hold (attributes) and what it can do (methods).
- An **object** is a specific instance of a class with actual values for its attributes.

- In Java and C++, you explicitly create objects (with `new` in Java). In Python, objects are created by calling the class like a function.

Remark

- Classes make code organized and reusable by grouping related data and behavior.
- Python uses `__init__` to initialize objects, while Java and C++ rely on constructors (same name as the class).

6.3.Inheritance, Encapsulation, and Abstract Classes

Definition

Inheritance allows a class to inherit attributes and methods from another class. Encapsulation hides data and exposes only necessary parts. Abstract classes are classes that cannot be instantiated and are meant to be inherited.

Feature

- **Inheritance**: A class (child) can reuse code from another class (parent).
- **Encapsulation**: Protects data by making attributes private and using methods to access them.
- **Abstract Classes**: Define common behavior for subclasses but cannot create objects directly.

Syntax

```
// Java
abstract class Animal {
    protected String name; // Encapsulation
    abstract void makeSound(); // Abstract method
}

class Dog extends Animal { // Inheritance
    void makeSound() {
        System.out.println(name + " says Woof");
    }
}

Dog dog = new Dog();
dog.name = "Rex";
dog.makeSound();
```

```
// C++
class Animal {
protected:
    string name; // Encapsulation
```

```
public:
    virtual void makeSound() = 0; // Pure virtual function (abstract)
};

class Dog : public Animal {
public:
    void makeSound() {
        cout << name << " says Woof" << endl;
    }
};

Dog dog;
dog.name = "Max";
dog.makeSound();
```

```
# Python
from abc import ABC, abstractmethod

class Animal(ABC): # Abstract class
    def __init__(self, name):
        self._name = name # Encapsulation with convention (_name)

    @abstractmethod
    def make_sound(self):
        pass

class Dog(Animal): # Inheritance
    def make_sound(self):
        print(f"{self._name} says Woof")

dog = Dog("Buddy")
dog.make_sound()
```

Example

- Java:

```
abstract class Vehicle {
    protected String brand;
    abstract void start();
}

class Car extends Vehicle {
    void start() {
        System.out.println(brand + " car starts with a key");
    }
}

Car car = new Car();
car.brand = "BMW";
car.start();
```

- C++:

```cpp
class Vehicle {
protected:
    string brand;
public:
    virtual void start() = 0;
};

class Car : public Vehicle {
public:
    void start() {
        cout << brand << " car starts with a key" << endl;
    }
};

Car car;
car.brand = "Toyota";
car.start();
```

- Python:

```python
from abc import ABC, abstractmethod

class Vehicle(ABC):
    def __init__(self, brand):
        self._brand = brand

    @abstractmethod
    def start(self):
        pass

class Car(Vehicle):
    def start(self):
        print(f"{self._brand} car starts with a key")

car = Car("Honda")
car.start()
```

Explanation

- **Inheritance**: A child class (e.g., `Dog`) inherits from a parent class (e.g., `Animal`), reusing its attributes and methods.
- **Encapsulation**: Data is hidden (e.g., `protected` in Java/C++, `_name` in Python) to control access and ensure safety.
- **Abstract Classes**: Define methods that must be implemented by child classes, ensuring a common structure.

Remark

- Java uses `abstract` and `protected` for abstraction and encapsulation.
- C++ uses `virtual` and `= 0` for abstract methods and `protected` for encapsulation.
- Python uses `ABC` and `_` prefix for abstraction and encapsulation (by convention).

6.4.The `this` / `self` Keyword

Definition

The `this` (Java, C++) or `self` (Python) keyword refers to the current object, allowing access to its attributes and methods within the class.

Feature

- Helps distinguish between class attributes and local variables or parameters.
- Used to call methods or access data of the current object.

Syntax

```
// Java
class Person {
    String name;

    void setName(String name) {
        this.name = name; // this distinguishes class attribute from parameter
    }

    void introduce() {
        System.out.println("My name is " + this.name);
    }
}

Person p = new Person();
p.setName("Alice");
p.introduce();
```

```
// C++
class Person {
public:
    string name;

    void setName(string name) {
        this->name = name; // this-> distinguishes class attribute from parameter
    }

    void introduce() {
        cout << "My name is " << this->name << endl;
    }
};

Person p;
p.setName("Bob");
p.introduce();
```

```python
# Python
class Person:
    def __init__(self, name):
        self.name = name # self is required to access class attribute

    def introduce(self):
        print(f"My name is {self.name}")

p = Person("Charlie")
p.introduce()
```

Example

- Java:

```java
class Book {
    String title;

    void setTitle(String title) {
        this.title = title;
    }

    String getTitle() {
        return this.title;
    }
}

Book book = new Book();
book.setTitle("Java Basics");
System.out.println(book.getTitle());
```

- C++:

```cpp
class Book {
public:
    string title;

    void setTitle(string title) {
        this->title = title;
    }

    string getTitle() {
        return this->title;
    }
};

Book book;
book.setTitle("C++ Guide");
cout << book.getTitle() << endl;
```

- Python:

```python
class Book:
    def __init__(self, title):
        self.title = title

    def get_title(self):
        return self.title

book = Book("Python 101")
print(book.get_title())
```

Explanation

- In Java and C++, `this` is used to refer to the current object's attributes or methods, especially when parameter names match attribute names.
- In Python, `self` is explicitly required as the first parameter in instance methods to refer to the current object.
- Using `this` or `self` ensures clarity when accessing object-specific data.

Remark

- Java and C++ implicitly pass `this`, so it's optional unless needed for clarity.
- Python requires `self` explicitly in method definitions and calls.
- Misusing `this` or `self` can lead to errors, like accessing the wrong variable.

7.Generic Programming

7.1.C++ Templates

Definition

Templates in C++ allow writing code that works with any data type, generating type-specific code at compile-time.

Feature

- Templates enable reusable code for different types without rewriting.
- Type checking happens at compile-time.
- Templates support both functions and classes.

Syntax

```cpp
// C++ Function Template
template <typename T>
T add(T a, T b) {
    return a + b;
```

```
}

int x = add(5, 10);       // Works with int
double y = add(3.5, 4.2); // Works with double
```

```
// C++ Class Template
template <typename T>
class Box {
public:
    T value;
    Box(T v) : value(v) {}
    T getValue() { return value; }
};

Box<int> intBox(42);
Box<double> doubleBox(3.14);
```

Example

- Function Template:

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << max(10, 20) << endl;      // Outputs 20
    cout << max(2.5, 1.5) << endl;    // Outputs 2.5
    return 0;
}
```

- Class Template:

```
template <typename T>
class Pair {
public:
    T first, second;
    Pair(T f, T s) : first(f), second(s) {}
    T getSum() { return first + second; }
};

Pair<int> intPair(3, 4);
Pair<double> doublePair(1.1, 2.2);
```

Explanation

- Templates allow writing one piece of code that works for multiple types (e.g., `int`, `double`).
- The compiler generates specific versions of the code for each type used.
- Templates are resolved at compile-time, ensuring no runtime overhead.

Remark

- Templates are powerful for creating flexible, reusable code in C++.
- They can make code harder to read if overused or poorly designed.
- Errors in templates are caught at compile-time, but error messages can be complex.

7.2.Java Generics

Definition

Java generics allow classes, interfaces, and methods to work with any reference type, with type checking at compile-time.

Feature

- Generics ensure type safety without casting.
- Only reference types (not primitives like `int`) can be used.
- Generics are erased at runtime (type information is removed).

Syntax

```
// Java Generic Method
public <T> T getFirst(T[] array) {
    return array[0];
}

Integer[] intArray = {1, 2, 3};
String[] strArray = {"a", "b", "c"};
Integer firstInt = getFirst(intArray); // Returns 1
String firstStr = getFirst(strArray);  // Returns "a"
```

```
// Java Generic Class
public class Container<T> {
    private T item;
    public Container(T item) { this.item = item; }
    public T getItem() { return item; }
}

Container<Integer> intContainer = new Container<>(42);
Container<String> strContainer = new Container<>("hello");
```

Example

- Generic Method:

```
public <T> void printArray(T[] array) {
    for (T element : array) {
```

```
        System.out.println(element);
    }
}

Double[] doubles = {1.1, 2.2, 3.3};
printArray(doubles); // Outputs 1.1, 2.2, 3.3
```

- Generic Class:

```
public class Pair<T> {
    private T first, second;
    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }
    public T getFirst() { return first; }
}

Pair<String> strPair = new Pair<>("left", "right");
System.out.println(strPair.getFirst()); // Outputs "left"
```

Explanation

- Generics allow writing type-safe code that works with any reference type (e.g., `Integer`, `String`).
- The compiler checks types at compile-time, preventing errors like assigning a `String` to an `Integer` container.
- At runtime, generic type information is erased, so the JVM sees only raw types.

Remark

- Generics make Java code safer and eliminate the need for explicit casting.
- They are limited to reference types, so primitives must be wrapped (e.g., `int` to `Integer`).
- Generic code is easier to maintain but can be verbose.

7.3.Python Duck Typing

Definition

Duck typing in Python allows objects to be used based on their behavior (methods and attributes) rather than their explicit type, with no enforced generic restrictions.

Feature

- If an object has the required methods or attributes, it can be used, regardless of its type.
- No need to declare types explicitly; Python checks at runtime.
- Highly flexible but can lead to runtime errors if assumptions about behavior are wrong.

Syntax

```
# Python Duck Typing
class Dog:
    def speak(self):
        return "Woof"

class Cat:
    def speak(self):
        return "Meow"

def make_animal_speak(animal):
    print(animal.speak()) # Works if animal has a speak() method

dog = Dog()
cat = Cat()
make_animal_speak(dog) # Outputs "Woof"
make_animal_speak(cat) # Outputs "Meow"
```

Example

- Duck Typing with Different Objects:

```
class Duck:
    def quack(self):
        return "Quack"

class Person:
    def quack(self):
        return "I'm quacking!"

def make_it_quack(obj):
    print(obj.quack()) # Works if obj has quack() method

duck = Duck()
person = Person()
make_it_quack(duck)   # Outputs "Quack"
make_it_quack(person) # Outputs "I'm quacking!"
```

Explanation

- Duck typing follows the principle: "If it walks like a duck and quacks like a duck, it’s a duck."
- Python doesn’t care about the object’s type, only whether it has the expected methods or attributes.
- Errors occur at runtime if the object lacks the required behavior.

Remark

- Duck typing makes Python code flexible and concise, ideal for rapid development.
- It can lead to runtime errors if objects don’t have the expected methods.

- Unlike C++ templates or Java generics, duck typing doesn’t enforce type constraints, relying on programmer discipline.

8.Language-Specific Features

8.1.Java Features

8.1.1 Java Virtual Machine (JVM) and Just-In-Time (JIT) Compilation

Definition

JVM is the runtime that executes Java bytecode, enabling platform-independent execution.

JIT is part of JVM that improves performance by compiling frequently used bytecode to native machine code at runtime.

Feature

- **JVM**: Provides platform independence by running bytecode, manages memory with garbage collection, and ensures portability.
- **JIT**: Compiles "hot" (frequently executed) bytecode to native code for faster execution after initial interpretation.

Syntax

No direct syntax; usage involves compiling Java code to bytecode, then executing via JVM:

```
javac Hello.java
java Hello
```

Example

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

When you run `java Hello`, JVM loads bytecode and JIT may compile `main` to native code for faster execution.

Explanation

- JVM enables "Write Once, Run Anywhere".
- JIT improves performance over pure interpretation by compiling during runtime.

8.1.2.File Structure

Definition

Each public class must be in its own `.java` file with the same name as the class.

Feature

- Ensures clear structure and avoids naming conflicts.
- Allows compiler to locate public classes easily.

Syntax

```
// File: MyClass.java
public class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello from MyClass");
    }
}
```

Example

```
// MyClass.java
public class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello from MyClass");
    }
}

class Helper {
    void help() {
        System.out.println("Helping...");
    }
}
```

Here, `MyClass` is public (file must be named MyClass.java), while `Helper` is package-private.

Explanation

- Improves code organization.
- Different from C++, where multiple public classes can be in one file.

8.2. Python Features

8.2.1 Interpreted Language

Definition

Python code is executed line by line by the interpreter at runtime.

Feature

- No compilation step before execution.
- Bytecode (.pyc) is generated for faster reruns.
- Highly portable across platforms with a Python interpreter.

Syntax

```
# hello.py
print("Hello, World!")
```

Run with:

```
python hello.py
```

Example

```
def add(a, b):
    return a + b

print(add(2, 3))
```

Interpreter compiles to bytecode and executes, printing `5`.

Explanation

- Easier debugging and rapid development.
- Slower than compiled languages like C++ for performance-critical tasks.

8.2.2 `if __name__ == "__main__":`

Definition

Ensures code runs only when the script is executed directly, not when imported as a module.

Feature

- Prevents unintended execution when importing modules.
- Separates script code from reusable functions.

Syntax

```python
if __name__ == "__main__":
    print("Running directly")
```

Example

```python
# module.py
def greet(name):
    print("Hello,", name)

if __name__ == "__main__":
    greet("World")
```

When run directly, prints "Hello, World". When imported, `greet` is available without executing the block.

Explanation

- `__name__` is set to `"__main__"` if run directly, otherwise to the module name.

8.3. C++ Features

8.3.1. Multi-Paradigm Programming

Definition

Supports procedural, object-oriented, generic, and functional programming styles.

Feature

- Procedural: function-based code
- OOP: classes, inheritance, encapsulation
- Generic: templates
- Functional: lambdas, std::function

Syntax

Procedural:

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

OOP:

```
class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }
};
```

Example

```
#include <iostream>

class Person {
public:
    std::string name;
    Person(std::string n) : name(n) {}
    void greet() {
        std::cout << "Hello, " << name << std::endl;
    }
};

int main() {
    Person p("Alice");
    p.greet();
    return 0;
}
```

Explanation

- Flexibility to choose paradigm based on task complexity and requirements.

8.3.2 Operator Overloading

Definition

Allows custom behavior for operators when used with user-defined types.

Feature

- Improves code readability for operations on custom classes.

Syntax

Member function:

```
class Complex {
public:
    int real, imag;
    Complex operator+(const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }
};
```

Example

```
#include <iostream>

class Complex {
public:
    int real, imag;
    Complex(int r=0, int i=0) : real(r), imag(i) {}
    Complex operator+(const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }
};

int main() {
    Complex c1(1,2), c2(3,4);
    Complex c3 = c1 + c2;
    std::cout << c3.real << " + " << c3.imag << "i" << std::endl;
    return 0;
}
```

Explanation

- Makes mathematical operations with classes intuitive.

8.3.3. Manual Memory Management

Definition

Uses `new` to allocate and `delete` to free memory manually.

Feature

- Precise control over memory allocation and deallocation.
- Risk of memory leaks if `delete` is forgotten.

Syntax

Single object:

```cpp
int* ptr = new int(5);
delete ptr;
```

Array:

```cpp
int* arr = new int[3];
// use arr
delete[] arr;
```

Example

```cpp
#include <iostream>

int main() {
    int* num = new int(10);
    std::cout << *num << std::endl;
    delete num;
    return 0;
}
```

Explanation

- `new` allocates on heap; `delete` frees it.
- Modern C++ recommends smart pointers to avoid manual errors.

8.4. Summary

Language	Feature	Key Benefit	Drawback
Java	JVM + JIT	Portability + Runtime speed	Initial JIT overhead
Java	File Structure	Organized code management	Restrictive in small projects
Python	Interpreted Language	Quick development + Portability	Slower performance
Python	if **name** == "**main**"	Prevents unintended execution	Requires understanding structure
C++	Multi-Paradigm	Flexible programming styles	Can increase complexity
C++	Operator Overloading	Intuitive syntax for classes	Misuse can cause confusion
C++	Manual Memory Management	Precise control over memory	Risk of leaks / undefined behavior