Advanced Programming

Course Structure:

1. Programming Paradigms & evaluation models (lazy vs eager)
2. Runtime Architectures (LLVM, JVM, CLR)
3. Memory Management (stack vs heap, GC, RAII, ownership)
4. Type systems & Abstractions (polymorphism, inference, traits)
5. Functions, Closures & Control flow
6. Object Oriented & Component Programming (reflections, attributes, decorators)
7. Structured Data querying (LINQ, list comprehension, streams, ORMs)
8. Asynchronous programming (async/await, GIL, promises)
9. Toolchain & AI-assisted development (GIT, IDEs, Copilot, refactoring)
10. **EXAM: Individual Project using AI, and to explain, check AI Code + Oral exam.**

✅ Lecture 0 – Introduction

1.1 Course Overview and Objectives

The **Advanced Programming** course explores programming not only as writing code but, more importantly, as the ability to **understand, analyze, and evaluate code written by others**—including by **artificial intelligence (AI)** systems. The main challenge for modern programmers is not just to code, but to **read and reason about code** that is often created by automated tools .

This reflects the new reality in software engineering: in the near future, programmers will need to **supervise and verify code** generated by AI models such as GitHub Copilot, ChatGPT, or Google Gemini. The goal is to prepare

students to check **correctness, efficiency, and security** of this AI-produced code.

1.2 The Role of Artificial Intelligence in Programming

In the past, programming languages were created as **human-readable layers** over machine code.

Today, **AI can generate full programs** that are syntactically and semantically correct. However, even if the code compiles and runs, its **behavior may not be correct**. This creates risks such as **hidden bugs, unsafe code, or security backdoors** .

Two main problems appear:

- **Probabilistic Generation:** Large Language Models (LLMs) generate text by predicting likely continuations. This allows creativity but also causes "hallucinations" — wrong or misleading outputs.
- **Security Issues:** AI-generated code can accidentally (or intentionally, if manipulated) include vulnerabilities. Past examples of backdoors in open-source software show how serious this can be.

Therefore, modern programmers must learn not only to **write code**, but also to **inspect and reason about AI-generated code**.

1.3 Historical and Conceptual Context

The course introduction referred to **Alan Turing's** early thoughts on machine intelligence. Turing imagined that machines might one day create their own "tables" — an early idea of programs. This is now reality: **AI systems can generate their own code**, but humans must still **critically evaluate** it .

This also raises questions of **responsibility**. Similar to debates about **self-driving cars** (who is responsible — the programmer, the company, or the AI?), programmers must ask: **who is responsible for errors in AI-generated software?**

This ethical question is an important theme of the course.

1.4 Teaching Methodology

The **teaching style** of Advanced Programming combines **theory and practice**:

- **Theoretical Foundations:** topics such as syntax, semantics, type systems, closures, and runtime environments.

- **Practical Sessions:** using AI tools (GPT, Copilot, Claude, Gemini) to **generate and analyze code**.
- **Critical Exercises:** testing AI output, finding weak points, and applying **unit testing and cross-model comparison** to check correctness .

Students are also part of an **experimental version** of the course, so flexibility and adaptation are expected.

1.5 Illustrative Examples

The professor showed several demos to explain both the strengths and weaknesses of AI programming:

- **Mandelbrot Viewer:** The AI created a Mandelbrot set visualizer in seconds, including zoom and graphics. This showed how quickly AI can work, but also how **prompt clarity** affects results. (Available online: *Mandelbrot Viewer*.)
- **3D Flight Simulator:** The AI generated a simple 3D airplane game with balloon-popping. It was less refined, showing that **vague prompts lead to weaker programs**. (Available online: *3D Flight Simulator*.)
- **Security Case Study (Eligere Voting System):** Using OpenAI Codex, the AI analyzed a real electronic voting system. It could detect **encryption structures** and **possible vulnerabilities**, proving that AI can act as a **code reviewer**.

These examples show why **reviewing and testing AI-generated code** is now an essential skill.

1.6 Probabilistic Outputs and LLM Behavior

Large Language Models do not always produce the same output, even with the same prompt. Unlike compilers, they work through **probabilistic text generation**.

At each step, the model predicts the next word (or token) from a list of possibilities. To control randomness, a **temperature parameter** is used:

- **Low temperature (~0):** the model becomes predictable and consistent — good for precision but less creative.
- **High temperature (>1):** the model becomes more creative but also more error-prone.

In programming, this trade-off matters.

A **low temperature** gives safer, more stable code, while a **higher temperature** can produce new ideas but may introduce mistakes. Developers must **balance safety and creativity** when using AI tools for coding or refactoring.

1.7 Key Concepts Preview

The lecture introduced several key concepts:

- **Lexical Closures:** functions that keep access to variables from the environment where they were created, even after that environment ends.
- **Scoping and Semantics:** code should behave exactly as its text describes — this is a core rule of programming language design.
- **Programming Language Families:** languages evolve in groups (e.g., C → C++ → Java; Lisp → Scheme → Haskell). Understanding these links helps in learning new languages.
- **Multiplicity of Languages:** As Douglas Hofstadter wrote in *Gödel, Escher, Bach*, programming languages are like musical keys. Different keys (or languages) fit different problems. Using only one language for everything would be as unreasonable as writing all music in one key .

1.8 Conclusion

The first lecture introduced the main theme of the course: In the **age of AI**, programming is less about typing code and more about **understanding, testing, and taking responsibility** for what AI systems produce. This new approach requires strong theory, practical knowledge of AI tools, and above all, a **critical mindset** — knowing when to trust AI and when to question it.

✅ Lecture 1 – Programming Language Stack

2.1 Introduction

In the second lecture, Professors **Cisternino and Corradini** explained the **architecture of programming languages**, focusing on the difference between **compiled and interpreted languages**, the **role of types**, and how **language semantics** evolved over time.

They emphasized that programming languages are not just technical systems but also **cultural products**, shaped by design debates, historical events, and the effort to balance **human readability** with **machine execution**.

2.2 Historical Perspectives

Every programming language symbol and syntax rule has a history. For example, different languages use different symbols for assignment:

- **C** uses `=`,
- **Pascal** uses `:=`, which reflects the mathematical idea of equality as a relation, not an action.

These design choices show how language creators have long debated **clarity, formality, and usability**.

One story from class was about the early 2000s, when **Microsoft Research** developed **C# generics** (a type of polymorphism). Teams spent days deciding where to put angle brackets and commas to keep the grammar consistent. This shows that even small syntax details can reflect **deep theoretical discussions**.

2.3 Compiled vs. Interpreted Languages

The lecture then discussed the difference between **compiled** and **interpreted** languages. In simple terms:

- **Compiled languages** translate **source code** into **machine code** before running.

  Source code is code that we see, and human can understand, machine code is code that computer readable code (binaries 0s and 1s).

- **Interpreted languages** execute code **line by line, in fly (at a time of running)** through an interpreter program.

However, this difference is not absolute:

- A **CPU** itself works as a basic interpreter at the hardware level (fetch-execute cycle).
- Many modern languages use **hybrid systems**. For example, **Java compiles to bytecode**, which runs on the **Java Virtual Machine (JVM)**.
- Other tools, like **transpilers**, translate code from one high-level language to another (e.g., TypeScript → JavaScript).

So, **compilation and interpretation are not opposites**; they exist on a **continuum**.

```
  A[Source Code] -->|Compilation| B[Machine Code]

A -->|Transpilation| C[Intermediate Language e.g., JavaScript]
C -->|Execution|    D[Interpreter or VM]
B -->|Execution|    E[CPU Fetch-Execute Cycle]
```

Figure 2.1 – Simplified view of compilation and interpretation.

2.4 Formal Specifications and Machines

A **machine** (physical or virtual) has its own **instruction language**.

Compilers translate a **source language** (like C or Java) into the **target language** that the machine can execute, keeping the meaning (semantics) the same.

Interpreters, instead, read and execute code directly or through an intermediate form. In practice, **no language is purely compiled or interpreted**:

- Compiled languages depend on **runtime libraries** (e.g., `printf` in C), which are not compiled by the programmer.
- Interpreters often include **preprocessing** or **optimization** steps before running the code.

2.5 The Example of printf in C

A good example is the `printf` function in C. Many students believe C is a fully compiled language, but this is not completely true. When you compile:

```
printf("Hello, world!\n");
```

1. The **C compiler** translates your code into **assembly**.
2. The **runtime library** provides the actual code for `printf`, which is **linked at runtime**, not compiled by your compiler.
3. During execution, your program **pushes the arguments** onto the **stack** and calls the runtime's `printf` function.

The **stack** is essential for managing function calls. In C, the **caller** cleans up the stack after a function call (unlike Pascal, where the callee does). This rule is important because `printf` takes a **variable number of arguments**, so only the caller knows how many items to clean up.

In class, a **GPT-based tool** was used to show how the compiler turns this code into **assembly instructions**. The example showed the `CALL` to the runtime's `printf`, making it clear that **even compiled languages depend on runtimes**.

```
graph TD
  A[Source Program with printf] —>|Compilation| B[Assembly Code]

  B —>|Linking|     C[Executable Binary]
  C —>|Execution|   D[Runtime Library: printf]
  D —>|System Call| E[Operating System Kernel]
```

Figure 2.2 – Flow of control when using printf in a C program.

2.6 Runtime Systems

Runtime systems is the layer that makes your program actually work when it is running. It manages memory, handles input/output, manages threads, and communicates with the operating system. They include:

- Memory management (e.g., garbage collection)
- Input/output operations
- Networking and concurrency support

Examples:

- The **C runtime**
- The **Java Virtual Machine (JVM)**
- Microsoft's **.NET Universal Runtime (URT)**, created to modernize C's runtime by adding garbage collection, networking, and graphics.

Runtimes blur the boundary between compiled and interpreted execution, because they include **interpreters and management tools** even for compiled languages.

2.7 Types and Type Systems

Types organize and control data in programming.

A **type system** is the set of types and typing rules that each programming language uses to help the programmer avoid certain kinds of errors called type errors.

A **type** defines:

- What **values** exist (e.g., integers, strings)
- What **operations** can be done with them

Languages differ in how strictly they enforce type rules:

- **Strongly typed languages** (like Java, Haskell) prevent invalid operations between types.
- **Weakly typed languages** (like C, C++) allow unsafe actions (e.g., pointer casts), leaving responsibility to the programmer.

Static vs. Dynamic Typing

- **Statically typed** languages (C, Java) check types **at compile time**.
- **Dynamically typed** languages (Python, JavaScript) check types **while running**.

Some languages mix both. For example, **Java** checks types at compile time but still performs **runtime checks** (like downcasting). Java's **generics** reduce these cases, but due to **type erasure**, some checks still happen during execution.

Duck Typing

In dynamic languages, **duck typing** means that if an object **behaves like** another (has the same methods), it can be used the same way — no formal type declaration is required. This is common in **Python** and **JavaScript**. This means if an object "looks like a duck and quacks like a duck," it can be treated as a duck for the purpose of the program.

2.8 Type Constructors and Language Expressivity

A **type constructor** is a feature that allows creating new types from existing ones. Languages differ in support for this:

- **Java and C++**: allow creating new types (classes, structs, templates).
- **JavaScript, Lua, and early Lisp**: lack formal type constructors and rely on **objects or dictionaries** instead.

In **JavaScript**, for example, the `class` keyword is only **syntactic sugar**—internally, objects are just key-value dictionaries (`a.b` is the same as `a["b"]`). The meaning of `this` also changes depending on the **calling context**, making the semantics more complex.

2.9 Language Classification

Programming languages can be grouped by several criteria:

- **Compiled vs. Interpreted**
- **Statically vs. Dynamically typed**
- **Strongly vs. Weakly typed**
- **With or without type constructors**

Examples:

- **Rust:** compiled, statically typed, strongly typed, with type constructors.
- **Python:** interpreted, dynamically typed, weakly typed, no traditional type constructors.
- **JavaScript:** interpreted, dynamically typed, limited type construction via objects.

These are not strict categories—**most languages combine features** from multiple groups.

2.10 Conclusion

The lecture showed that:

- **Compilation and interpretation** form a **continuum**, not a binary choice.
- **Types** are central to understanding program behavior and safety.
- Programming languages are also **historical and cultural** creations, not just technical inventions.

To analyze a language, ask:

1. Is it compiled or interpreted?
2. Is it statically or dynamically typed?
3. Is it strongly or weakly typed?
4. Does it have type constructors?

These questions give a **structured way** to understand any programming language—classical or modern.

✅ Lecture 2 – PL Transformation Architecture

3.1 Introduction

The third lecture started by explaining the **difference between programming languages and markup languages**.

For example, **HTML** is not a programming language but a **markup language**, used to **structure documents** rather than control computations.

HTML -Markup Language, CSS - Stylesheet Language, JS - Programming Language

Newer and simpler formats such as

Markdown and **Mermaid** were also discussed. Mermaid, for instance, allows you to **write diagrams as text** and then automatically **render them visually**. This shows how text can be turned into structured, meaningful content.

This topic introduced the main idea of the lecture: **programming languages transform text into structured forms** through a **series of steps (a pipeline)**. Understanding this process is important for anyone who wants to **build or analyze compilers, interpreters, or AI code systems**.

3.2 Metaprogramming

One of the key concepts discussed was **metaprogramming** — writing programs that **generate or modify other programs**.

A program is not always the final product to be executed by a machine; sometimes, it is **input for another program** that transforms it. This idea is fundamental in modern computing. Examples include:

- **Compilers and interpreters:** programs that read and transform other programs.
- **Frameworks** like **TypeScript** or **React:** they translate higher-level syntax into JavaScript and HTML.
- **ORMs (Object-Relational Mappers):** they transform code into SQL queries.

Metaprogramming often comes from a desire to automate repetitive tasks.
After writing the same code patterns many times, programmers build tools or generators to do it automatically.

AI tools today do exactly this — they perform **metaprogramming at scale**.

3.3 The Compiler Pipeline

A **compiler** or **interpreter** processes a program in several stages, known as the **compiler pipeline**:

```
graph TD
  A[Source Code] --> B[Lexical Analysis]

  B --> C[Syntax Analysis]
  C --> D[Semantic Analysis]
  D --> E[Intermediate Representation]
  E --> F[Optimizations]
  F --> G[Target Code Generation]
```

Figure 3.1 – General structure of a programming language processor.

3.3.1 Lexical Analysis

- **Input:** a stream of characters.
- **Task:** Convert **Source code** (text) into **tokens** (such as keywords, identifiers, numbers, or symbols).
- Removes unimportant elements like spaces and comments.
- Often implemented using **finite automata**, which are based on **regular languages**.

3.3.2 Syntax Analysis (Parsing)

- **Input:** a stream of tokens.
- **Task:** take **tokens, pushes to** parser and build a **tree representation** of the program (called a **parse tree** or **abstract syntax tree – AST**).
- Example: the expression `a + b` becomes a tree where `+` is the root and `a` and `b` are its children.

3.3.3 Semantic Analysis

- **Input:** the **abstract syntax tree -** AST.

- **Task:** Checks of the code is logically correct, enforce the **rules of meaning** (semantics), like checking that variable types are used correctly.
- Supports **type inference**, meaning the compiler can deduce a variable's type (e.g., `let a = 2` implies `a` is an integer).

3.3.4 Intermediate Representation and Optimization

- The compiler converts the program into an **Intermediate Representation (IR)** for easier processing. And after we got IR we focus on **optimization analysis**.
- **Goals:**
  1. Map high-level code to lower-level instructions.
  2. Optimize the program for speed and efficiency.

3.3.5 Target Code Generation

- Converts the IR into **machine code**, **assembly**, or **bytecode**. At this point program is fully compiled and ready to run at CPU.
- Examples:
  - **GCC** uses **RTL** as its IR.
  - **Java** compiles into **bytecode**.
  - **.NET** compiles into **CIL (Common Intermediate Language)**.

3.4 Determinism and Safety in Language Processing

A key property of programming languages is **determinism**: given the same input, the compiler or interpreter must always produce the same output.

Non-determinism (inconsistent behavior) is unacceptable in safety-critical systems like aerospace.

A famous example is the **Ariane 5 rocket explosion (1996)**, caused by an **integer overflow** in software reused from Ariane 4. This disaster showed the importance of deterministic and well-tested software.

3.5 Intermediate Representations and Virtual Machines

The lecture also discussed the importance of **Intermediate Representations (IR) — a form of code between source code and machine code.** Examples:

- **P-code** in Pascal: an early portable intermediate format.

- **Java bytecode:** simpler than Java but richer than raw machine code. It includes metadata about classes and methods, allowing **reflection** and **portability**.
- **.NET's CLR (Common Language Runtime):** allows multiple languages (C#, VB.NET, F#) to interoperate by compiling into the same IR.

LLVM: a modern system for creating and optimizing intermediate representations across many platforms. Shows high level Source Code in IR code

```
graph TD
  A[Java Source Code] ⟶ B[Java Compiler]
  B ⟶ C[Bytecode]
  C ⟶ D[JVM - Interpreter/JIT]
  D ⟶ E[Machine Code]
```

Figure 3.2 – Java compilation to bytecode and execution on the JVM.

3.6 Automata and Regular Languages

Automata - An **automaton** (plural: **automata**) is a **machine or model that follows a set of rules to process input and produce output** — automatically, without human intervention. It reads some input (like a string of letters or numbers), follows specific rules, and decides what to do next (for example, accept or reject the input).

Main types of automata:

1. **Finite Automata (FA)** – simplest one, has a limited number of states.
2. **Pushdown Automata (PDA)** – can use a "stack" memory.
3. **Turing Machine** – most powerful, can simulate any computer program.

Lexical analysis depends on **finite automata**, mathematical models that recognize specific patterns in text. However, automata have **limitations**. For example, they cannot verify if parentheses are properly balanced — for that, **context-free grammars** are required.

Regular expressions (regex) are practical tools based on automata theory. They are used to find or manipulate text. **Tool that helps you to check if string matches a certain pattern**. They are widely used for validation of forms.

For Example: If you enter your email address somewhere, you can use regex to check if its a actually valid email address. But its very hard to use, and understand, read them, if you use them wrong they can cause lots of problems.

Key concepts:

- **Operators:** concatenation, alternation (`|`), repetition (, `+`, `{n,m}`).
- **Shorthands:** `\d` (digit), `\w` (word character), `[a-z]` (range).
- **Greedy vs. lazy matching:** matches as much as possible; `?` matches as little as possible.

In practice, modern regex engines (like Perl-style regex) go beyond strict regular languages because they allow **backtracking** and are even **Turing-complete**.

```
graph LR
  Q0((Start)) -->|F| Q1((q1))
  Q1 -->      |O| Q2((q2))
  Q2 -->      |R| Q3((Accept))
  Q0 -->      |Other| QDead((Dead State))
```

Figure 3.3 – Automaton recognizing the keyword “for”.

3.7 Tokenizers and Practical Examples

A **tokenizer** (or **lexer**) **converts raw text into tokens for later stages of compilation**. In class, a **GPT-assisted example** showed how to build a simple tokenizer that recognizes:

- Keywords (`for`, `while`),
- Identifiers,
- Numeric literals.

This demonstrated how **regular expressions** and **state machines** work in real code. Example:

```
if (char.IsLetter(c) || c == '_') {
    // Identifier or keyword
    while (char.IsLetterOrDigit(c)) advance();
    if (lexeme == "for") return Token.For;
    if (lexeme == "while") return Token.While;
```

```
    return Token.Identifier;
}
else if (char.IsDigit(c)) {
    while (char.IsDigit(c)) advance();
    return Token.Number;
}
```

Figure 3.4 – Simplified tokenizer logic.

3.8 Conclusion

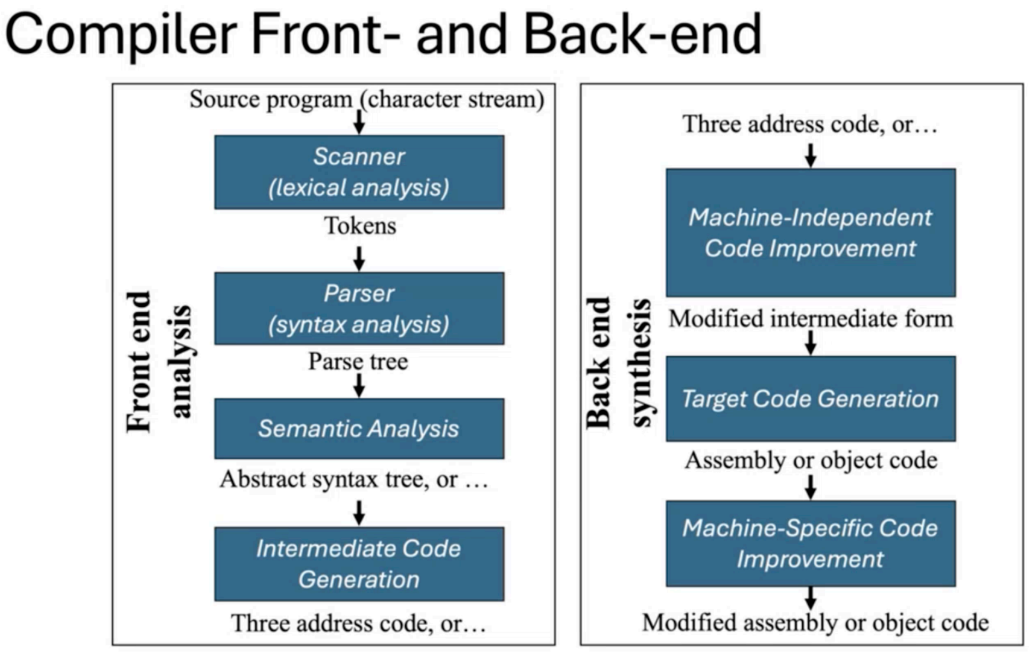
This lecture covered the **two main parts** of programming language processing:

1. **Analysis** – lexical, syntax, and semantic analysis.
2. **Synthesis** – code transformation, optimization, and code generation.

The professors emphasized that **programming languages evolve slowly** because of their complexity and the need for reliability. Understanding the **transformation pipeline**—from plain text to tokens, from syntax trees to machine code—helps programmers think clearly about how both **traditional compilers** and **AI-based code generators** work.

✅ Lecture 3 – Context Free Grammars

4.1 Introduction



In this lecture, **Professor Andrea Corradini** explained the theory and practice of **grammars**, which define the **syntax** of programming languages. The focus was on **top-down parsing** — a way of checking whether a program follows the correct structure.

Unlike previous lectures that studied the whole compiler, this one concentrated on:

- how grammars **generate languages**,
- how **derivations** and **parse trees** work, and
- how **parsers** decide if a program is **syntactically correct** .

4.2 Syntax, Semantics, and Pragmatics

To fully describe a programming language, we need three main parts:

1. **Syntax** – **the formal set of rules that define, what a valid program looks like in a given language** (expressed through grammars). Syntax is combination of regular expressions, characters and terminal symbols. Syntax refers to the form of code that comes before, semantic processing. The error generated by syntax is syntactic error.
2. **Semantics** – **what those valid programs actually mean. MEANINGFUL sentence, which is formed by using correctly well structured syntax.** Semantics is study of meaning of a programming language. It does this by evaluating meaning of syntactically valid string. **Invalid syntax doesn't proceed for Semantics**
3. **Pragmatics** – conventions that make code easier to read and use (for example, programming styles or naming rules). It refers to practical aspects of how different properties, features may be used to further improvement of code.

This lecture mainly focused on **syntax**, but also mentioned that **semantics** are checked later (during compilation), and **pragmatics** help make code clear and maintainable.

4.3 Grammars and the Chomsky Hierarchy

A **grammar** is a formal system that defines how strings in a language can be built. It is made of four parts:

- A set of **terminals symbols** (tokens). **Terminals** are the **basic symbols** of the language — they cannot be broken down any further.

`if`, `while`, `+`, `-`, `(`, `)`, numbers, variable names (identifiers).

- A set of **non-terminal symbols** are **syntactic categories** — they represent groups or structures made of terminals and other non-terminals. They are like placeholders for language constructs (functions) (expressions, statements, programs, etc.).
- A set of **production rules** (rewriting rules). is a **definition that tells how a non-terminal can be replaced** by a combination of terminals and/or other non-terminals.
- A **start symbol** is the **top-level non-terminal** — it represents the *whole program* or *complete input* the grammar can generate. The parser starts from this symbol and tries to apply grammar rules until it matches the input tokens.

Chomsky hierarchy is classification of grammars and languages, it provides a framework for understanding the power of different types of formal languages.

The **Chomsky hierarchy** classifies grammars **(Languages)** based on their **expressive power**:

1. **Regular grammars (Type 3)** – can be recognized by **finite automata, least powerful in hierarchy.**
2. **Context-free grammars (Type 2)** – can be recognized by **pushdown automata** (machines with a stack). Used to define nested structures. Commonly used in PL, can describe balanced parenthesis.
3. **Context-sensitive grammars (Type 1)** – can be recognized by **linear bounded automata**. Context of symbols matters, they can describe copy languages. More powerful and require complex machines.
4. **Unrestricted grammars (Type 0)** – as powerful as **Turing machines**. Recursively enumerable and can describe undecidable problems.

Each type includes the one below it. For example:

- Finite languages are **regular**.
- The language $\{a^n b^n\}$ is **context-free**, but **not regular**.
- The language $\{a^n b^n c^n\}$ is **context-sensitive**, but **not context-free**.

```
graph TD
  A["Unrestricted Languages (Type 0)"] --> B["Context-Sensitive Language
```

```
s (Type 1)"]
  B --> C["Context-Free Languages (Type 2)"]
  C --> D["Regular Languages (Type 3)"]
```

Figure 4.1 – The Chomsky hierarchy of grammars.

4.4 Derivations and Parse Trees

A **derivation** is a step-by-step process of applying production rules to generate a string, starting from the **start symbol** of the grammar and applies the **production rules** one by one until we get a string made only of **terminal symbols** (tokens like `if`, `+`, `(`, `)` etc.).

- In a **leftmost derivation**, the parser always expands the **leftmost non-terminal** first.
- In a **rightmost derivation**, it expands the **rightmost non-terminal** first.

A **parse tree** shows the structure of the derivation:

- **Root:** the start symbol.
- **Internal nodes:** non-terminals, show which production rules were applied.
- **Leaves:** terminal symbols (tokens).

A parse tree gives a **clear visual structure** of the program, independent of the derivation order.

However, **a grammar can be ambiguous**, meaning that the same string can produce more than one valid parse tree. Ambiguity causes problems because it can lead to **multiple interpretations** of a program.

Example: Ambiguity in arithmetic

For the string `9 - 5 + 2`, two different parses are possible:

- `(9 - 5) + 2 = 6`
- `9 - (5 + 2) = 2`

To remove ambiguity, programming languages use:

- **Operator precedence:** decides **which operator should be done first** when there are several in the same expression.(for example, `*` has higher precedence than `+`), and

- **Associativity rules:** decides **which direction** operators of the same precedence are applied — **left-to-right** or **right-to-left**.

  Both `+` and `-` have the same precedence. So we need **associativity** to decide the order.

  If they are **left-associative**, we group from **left to right**: `(9 - 5) + 2 = 6`

  If they were **right-associative**, we would group from **right to left**: `9 - (5 + 2) = 2`

  Most arithmetic operators (`+`, `-`, `*`, `/`) are **left-associative**. Some others, like **assignment (=)** and **exponentiation ()**, are **right-associative**.

Another classic ambiguity is the **dangling else** problem in nested `if` statements, where it's unclear which `if` an `else` belongs to. Most languages solve this by connecting the `else` to the **closest unmatched** `if` .

4.5 Lexical and Syntax Grammars

Programming languages usually have **two separate grammars**:

- **Lexical grammar** (regular): defines how **characters** forms **tokens** such as keywords, numbers, or identifiers. It is usually implemented with **regular expressions** and **finite automata**. One for recognizing words and symbols.

  So when the compiler sees: `sum = 10 + 5;` The **lexical analyzer (tokenizer)** converts it into: `IDENTIFIER(sum) ASSIGN(=) NUMBER(10) PLUS(+) NUMBER(5) SEMICOLON(;)`

- **Syntax grammar** (context-free): Once the program is broken into **tokens**, the **syntax grammar** decides **how these tokens combine** to form **valid statements or expressions**. It Deals with **structure**, not characters.

Things Grammars *Cannot* Check: Some language rules cannot be described using context-free grammars, such as:

- Variables must be declared before they are used,
- Functions must have the correct number of parameters.

These are checked later during **semantic analysis**.

4.6 Parsing Techniques

Parsing is the process of checking if a sequence of **tokens** follows the **grammar rules** of programming languages.

Some **general parsing algorithms** can handle *any* grammar, but they are **too slow** — around **$O(n^3)$** time (where n = number of tokens). This is fine for small inputs but too slow for large programs.

So, compilers use **restricted types of grammars** (grammars with certain properties) that allow **faster parsing** — ideally **linear time (O(n))**.

4.6.1 Top-Down Parsing

- Builds the **parse tree from the top (root)** — starting from the **start symbol** of the grammar — and goes **down to the leaves** (tokens). It tries to expand the grammar rules step by step to match the input tokens.
- Easy to understand and implement: each **non-terminal** can be represented by a **procedure** that tries to match input tokens.
- If the parser makes a wrong guess (chooses the wrong rule), it might have to go **back** and try a different rule — this is called **backtracking**. In the worst case, it might try **many combinations**, making it **exponential time** — very slow.

4.6.2 Predictive Parsing (LL Parsing)

- A **more efficient faster, smarter form of top-down parsing**.
- Avoids backtracking by using **lookahead tokens** (checking the next token before deciding which rule to use).
- If a grammar works with only one lookahead token, it's called **LL(1)** — and this type is ideal for compilers because it can be parsed in **linear time**.

  Why "LL(1)"

  L = Left-to-right scanning of input. **L** = Leftmost derivation. **(1)** = Uses **1 lookahead token** to decide which rule to use.

```
graph TD
  A[Start Symbol] --> B[Predictive Parser]
  B --> C[Lookahead Token]
  C --> D[Select Production]
  D --> E[Expand Non-Terminals]
```

Figure 4.2 – Simplified predictive parsing strategy.

First and Follow Sets

Predictive parsers use two important sets:

- **First(α):** all possible tokens that can appear at the start of strings derived from α.
- **Follow(A):** all possible tokens that can appear **right after** the non-terminal A.

These sets help determine whether a grammar is suitable for **predictive parsing (LL(1))**.

Eliminating Left Recursion

Predictive parsers **cannot handle left-recursive grammars**, where a non-terminal can produce itself as the first symbol (e.g., `A → A + B`).

Such rules must be **rewritten** into a **right-recursive form** so the parser can handle them.

4.7 Practical Examples

During the lecture, small examples were used to show how **predictive parsing** works.

A **ChatGPT demonstration** showed how to translate a grammar into **recursive functions**, and how **lookahead tokens** help the parser choose the correct rule **without backtracking** . Example:

```
void Expr() {
    Term();
    while (lookahead == '+' || lookahead == '-') {
        Token op = lookahead;
        match(op);
        Term();
    }
}
```

Figure 4.3 – Example of a recursive descent parsing function for expressions.

Let's go through it:

1. `Term();`

   → The parser first parses a **Term** — e.g., a number like `3`.

2. `while (lookahead == '+' || lookahead == '-') { ... }`

→ After reading a term, it checks if the next token is `+` or `-`.

→ If yes, that means the expression continues (`3 + 5`).

3. `Token op = lookahead;`

   → Store the operator (`+` or `-`).

4. `match(op);`

   → Consume (or "eat") that token from the input.

   Example: after matching `+`, `lookahead` will move to the next token (like the next number).

5. `Term();`

   → Parse the next term (for example, the `5` in `3 + 5`).

6. The `while` loop repeats for multiple additions or subtractions, like: `3 + 5 - 2 + 4`

When there are **no more `+` or `-` tokens**, the parser **stops** — meaning it successfully parsed the entire expression.

4.8 Conclusion

This lecture showed how **grammars define the syntax** of programming languages, how **parse trees** give structure, and how **ambiguity** must be resolved to ensure clear interpretation. It also explained **top-down parsing**, especially **predictive (LL) parsing**, as an efficient and widely used method in compilers.

✅ Lecture 4 – Parsing & Whitespace Compiler

5.1 Introduction

The lecture on September 29 had two parts.

In the first hour, Professor Corradini finished explaining **predictive parsing**, focusing on **First** and **Follow** sets and how to build **LL(1) parsers**.

In the second hour, Professor Cisternino showed a practical example — the **Whitespace compiler**, a fun but educational project that demonstrates how the theory of compilers is applied in real programs.

5.2 Predictive Parsing: First and Follow Sets

Predictive parsing allows a parser to decide which grammar rule to use by looking only **one token ahead** in the input. To do this, two sets must be computed: They help the parser make decisions **without guessing or backtracking**, so parsing is fast and deterministic.

First(α): the set of tokens that can appear at the start of any string derived from α.

- If α begins with a terminal (like `id` or `+`), then that terminal is in First(α).
- If α begins with a non-terminal, First(α) includes the First sets of all that non-terminal's productions.
- If α can produce ε (the empty string), then ε is also in First(α).

Follow(A): the set of **tokens** that can appear immediately **after non-terminal A** in a valid sentence.

- If there is a production like A β, then everything in First(β) (except ε) is added to Follow(A).
- If β can produce ε, then everything in the Follow set of the left-hand side non-terminal is added to Follow(A).
- For the start symbol S, the end-of-input marker `$` is always in Follow(S).

A grammar is **LL(1)** if, for each non-terminal, the lookahead sets for all its productions do not overlap. This ensures the parser can decide deterministically (without guessing or backtracking) which production to use.

Recursive Descent vs. Table-Driven Parsing

There are two common LL(1) parsing methods:

- **Recursive descent:** each non-terminal has its own function. The lookahead token decides which production to call.
- **Table-driven parsing:** uses a **parsing table** indexed by (non-terminal, lookahead). The parser uses a **stack** and the table to guide parsing steps.

Both methods work the same for LL(1) grammars.

```
graph TD
  A[Grammar] --> B[Compute First/Follow]
  B --> C["Check LL(1) Conditions"]
  C --> D[Build Parsing Table]
```

```
D --> E[Table-Driven Parser]
C --> F[Recursive Descent Parser]
```

Figure 5.1 – Steps to build LL(1) parsers

5.3 Error Handling in Parsing

When a compiler parses (reads) your code, it sometimes finds **ERRORS** — for example, missing a semicolon or writing a wrong expression. A good compiler must handle errors smoothly instead of just stopping at the first error. Good compiler: Detect the problem, report it cleanly, and try to keep going.

Common strategies include:

- **Panic mode:** The parser **skips tokens** until it reaches a "safe point" (like `;` or `}`) is found. My skip large parts of code.
- **Phrase-level recovery:** The parser tries to **fix small mistakes** immediately — like inserting or deleting one token. *Ex:* If the parser expects `;` and doesn't find it, it can assume it was just forgotten and **pretend to insert it**. Sometimes makes wrong assumptions.
- **Error productions:** We **add extra grammar rules** that describe *common errors* students or programmers make, customizable and precise. But makes grammar complicated
- **Global correction:** The parser tries to find the **smallest set of edits** (insertions, deletions, replacements) to turn the code into a valid program. This is like an *auto-fix* that finds the minimal way to make input correct. But slow and complex - rare in real compilers.

LL(1) parsers have the **Viable prefix property =** The parser can detect an error **as soon as** it reaches a point where the current prefix (the part of the input read so far) **cannot be completed into a valid program**.

5.4 The Whitespace Language

Whitespace is a *funny* or *esoteric* programming language — it was made as a joke to show that even "invisible" code can still be a real program.

In **Whitespace**:

- It only uses **spaces**, **tabs**, and **line feeds** (Enter).
- Every **visible character** (like letters, numbers, symbols) is **ignored**.

- So when you look at a Whitespace program, it looks empty — but it actually contains instructions made of invisible characters.

Internally, those spaces and tabs represent **commands**.

For example: A certain pattern of spaces and tabs means **"push a number to the stack."** Another pattern means **"add two numbers."** So the program looks blank, but it's actually telling the computer what to do.

Even though it started as a joke, **Whitespace is Turing complete** - meaning it's powerful enough to perform any computation — just like C, Python, or Java.

It can compute anything that any programming language can.

- **Stack manipulation:** push, pop, duplicate, swap
- **Arithmetic:** add, subtract, multiply, divide
- **Heap access:** read/write memory
- **Flow control:** labels, jumps, subroutines
- **Input/Output:** read and print integers or characters .

```
graph TD
  A[Whitespace Program] --> B[Tokenizer]
  B --> C[Parser]
  C --> D["Intermediate Representation (Instructions)"]
  D --> E[.NET Bytecode Generator]
  E --> F[Executable Program]
```

Figure 5.2 – Architecture of the Whitespace compiler in C#

5.5 Architecture of the Whitespace Compiler

Whitespace compiler is a real program that can **read**, **understand**, and **run** code written in the Whitespace language. It was originally created from 2003.

Main components:

1. **Tokenizer:** Reads the input program and **translates spaces, tabs, and line feeds into tokens** (symbols the compiler can understand). All other visible characters (like letters, numbers, etc.) are **ignored**.
2. **Parser:** The parser checks if the sequence of tokens **follows the Whitespace grammar** (rules of the language). It uses a **recursive descent**

parser, meaning each grammar rule is handled by a small function that calls others (like a chain).The parser then builds a list (or tree) of **instructions** — like a plan for what the program will do.

3. **Intermediate Representation:** The parsed instructions are stored as **objects** in memory. For example, one object represents "PUSH 6," another represents "ADD." For control flow (like jumps or loops), a **table of labels** is used so the program knows where to go next.
4. **Code Generation:** This stage **translates the IR into real .NET bytecode.** It uses **reflection** — a C# feature that allows generating code while the program is running. Whitespace uses its **own stack** for computations, so a custom `Stack` class is implemented to manage pushes, pops, etc.
5. **Executable Output:** generates `.dll` and `.exe` files that run the Whitespace code .

Example Program:

A simple program that pushes 6, pushes 1, adds them, and prints the result (7) appears blank, but internally it contains:

- **Invisible source:** `[space][number 6][LF][space][number 1][LF][tab][space][LF][tab][LF]`
- **Readable version:**

```
PUSH 6
PUSH 1
ADD
PRINT_INT
END
```

- **Execution output:** 7. This shows how invisible characters can perform real computation.

5.6 Stack Machines and Compilation

A **stack machine** is a type of computer model (or virtual machine) that **uses a stack** to store temporary data — instead of using CPU registers. Think of a **stack** like a pile of plates:

- You can **push** a new value on top,
- Or **pop** (remove) the top value when you need it.

Languages like **Java** and **.NET** use **stack-based virtual machines**. This makes compiler design simpler, because there's no need to manage a limited number of registers.

The Whitespace compiler uses this idea:

- **PUSH n** → Means: "put the number n on top of the stack." Compiler translates it to:

  `ldc.i4 n` // load integer n
  `Stack.Push` // push it to the Whitespace stack

- **ADD** → pops two values, adds them, and pushes the result.

The compiler also **keeps track of how many values are currently on the stack**. This helps it find **errors before running the program**. This process — analyzing what the program *would* do **without actually running it** — is called **abstract interpretation**.

5.7 Modernization with AI Assistance

The original Whitespace compiler was written in **C# 1.0**. To update it for **.NET 9**, Professor Cisternino used **AI tools** like GitHub Copilot and GPT models to:

- Suggest replacements for old code.
- Refactor and simplify code (for example, using `var` instead of explicit types).
- Automatically create pull requests for modernization.

The final version is a **working modern compiler**, partly rewritten with AI support. This demonstrates how software development is changing — programmers increasingly guide AI tools rather than write everything manually.

✅ Lecture 5 – Names, Scopes, Lexical Closures & Memory Management

1. Names

Definition: A **name** is an identifier that refers to a program entity.

What names can refer to

- Variables
- Functions / methods
- Types
- Constants

- Modules

Purpose

- Give meaning to memory locations and computations
- Improve readability and abstraction

Example

```
int x =5;
```

`x` is a **name** referring to a memory location storing `5`.

---

2. Scope

Definition: The **scope** of a name is the region of the program where that name is **visible and usable**.

Types of scope

a) **Lexical (static) scope** — most modern languages

- Scope determined by **program structure**
- Known at **compile time**

Example (Python / Java / C)

```
x =10

deff():
print(x)

f()
```

`x` refers to the global variable.

---

b) **Dynamic scope** (rare today)

- Scope depends on **call stack at runtime**
- Used historically (e.g., early Lisp)

Block scope

A name is visible only inside a block `{}` or function.

```
if (true) {
inty=3;
}
// y is not visible here
```

3. Lexical Closures

Definition: A **closure** is a function together with the **environment** (bindings of variables) in which it was defined. Key idea:

> A function “remembers” variables from its defining scope.

Why closures exist

- Functions are **first-class values**
- Functions can be returned or passed as arguments

Example (JavaScript / Python style)

```
defmake_adder(n):
defadd(x):
return x + n
return add

add5 = make_adder(5)
add5(3)# returns 8
```

- `add` uses `n`
- `n` is **not local** to `add`
- `n` is kept alive by the closure

Key properties

- Lexically scoped
- Variables captured by **reference or value** (language-dependent)
- Require heap allocation of captured variables

4. Memory Management

Memory management is how a language:

- Allocates memory
- Deallocates memory
- Prevents memory errors

Main memory regions

Stack

- Function calls
- Local variables
- Fast
- Automatic deallocation

```
voidf() {
int x =3;// stack
}
```

Heap

- Dynamically allocated objects
- Shared across functions
- Managed manually or automatically

```
int* p =malloc(sizeof(int));// heap
```

---

5. Memory Management Strategies

a) Manual memory management (C, C++)

- Programmer allocates and frees memory
- Errors possible:
  - Memory leaks
  - Dangling pointers
  - Double free

---

b) Garbage collection (Java, Python)

- Runtime automatically frees unused objects
- Safer, but with runtime overhead

---

c) Ownership-based management (Rust)

- Each value has one owner
- Compiler enforces safe usage
- No garbage collector

---

6. Closures and Memory Management (Important for exams)

Closures require:

- Captured variables to live **longer than stack frame**
- Therefore stored on the **heap**

Example problem

```
deff():
    x =10
returnlambda y: x + y
```

- `x` cannot be on stack
- Closure keeps it alive on heap

✅ Lecture 6 – Stack vs Heap, Data Lifetimes, Activation Records, Reference Counting, Garbage Collection

1. Stack vs Heap

Stack. The **stack** is a memory region used for:

- Function calls
- Local variables
- Parameters
- Return addresses

Properties

- Fast allocation/deallocation
- Automatic (LIFO: last in, first out)
- Limited size

Example

```
voidf() {
int x =10;// stored on stack
}
```

- `x` exists only while `f()` is running
- Removed automatically when `f()` returns

Heap. The **heap** is used for:

- Dynamically allocated objects
- Data that must outlive a function call

Properties

- Slower than stack
- Managed manually or by GC
- Large and flexible

Example

```
int* p =malloc(sizeof(int));// heap
```

Stack vs Heap (Summary)

Stack	Heap
Automatic	Manual or GC
Fast	Slower
Short-lived	Long-lived
Local variables	Objects, closures

---

2. Data Lifetimes

Data lifetime is how long a value exists in memory.

Types of lifetimes

a) Automatic (stack) lifetime

- Exists during function execution

```
voidf() {
int x =5;
}
```

`x` dies when `f()` ends.

---

b) Dynamic (heap) lifetime

- Exists until explicitly freed or garbage collected

```
Objecto=newObject();// heap
```

---

c) Static lifetime

- Exists for the entire program execution

```
staticint counter =0;
```

---

3. Activation Records (Stack Frames)

An **activation record** (or **stack frame**) stores all information about a function call.

What an activation record contains

- Parameters
- Local variables
- Return address
- Saved registers
- Control information

Example

```
intadd(int a,int b) {
int sum = a + b;
return sum;
}
```

When `add(2, 3)` is called, the stack frame contains:

- `a = 2`
- `b = 3`
- `sum = 5`
- return address

Why activation records matter

- Enable recursion
- Manage nested function calls
- Provide isolation between calls

---

4. Reference Counting

Reference counting is a memory management technique where:

- Each object keeps a count of how many references point to it
- When the count reaches zero, the object is freed

Example

```
a = []
b = a
```

- Reference count of `[]` is 2
- If both `a` and `b` are deleted → count = 0 → object freed

Advantages

- Simple
- Objects freed immediately

Problem: Cycles

```
a = {}
b = {}
a["b"] = b
b["a"] = a
```

- Reference count never reaches zero
- Memory leak

---

5. Garbage Collection (GC)

Garbage collection automatically finds and frees memory that is no longer reachable.

Key idea

> If no reference can reach an object, it is garbage.

Common GC approaches

a) Mark-and-sweep

1. Mark reachable objects
2. Free unmarked ones

b) Tracing GC

- Follows references from root objects (stack, globals)

Languages using GC: Java, Python, JavaScript, C#

Example

```
Objecto=newObject();
o =null;
```

- Object becomes unreachable
- GC will eventually free it

---

6. Reference Counting vs Garbage Collection

Reference Counting	Garbage Collection
Immediate deallocation	Periodic cleanup
Simple	More complex
Fails on cycles	Handles cycles
Low pause times	Possible pauses

---

7. Closures and Heap Allocation (Exam-Relevant)

Closures require:

- Captured variables to outlive function call
- Heap allocation instead of stack

```
deff():
    x =10
returnlambda y: x + y
```

`x` must be on the heap.

✅ Lecture 7 – Introduction to RUST, Ownership

7.1. Brief History

- Rust was created at **Mozilla** around **2010**.
- Main idea: build a systems language (like C/C++) **without memory bugs**.
- Rust became **1.0** in 2015.
- Used today by Microsoft, Amazon, Dropbox, Cloudflare, Linux kernel.

Rust's slogan:

👉 *"Fast, Safe, Concurrent — Pick three."*

7.2. Rust Goals and Syntax

Goals

- **Memory safety** without garbage collector (no Java-like GC).
- **High performance** (similar to C++).
- **Safe concurrency** (no data races).

Syntax style

- Looks like C/C++ but cleaner.

Example:

```
let x = 5;
println!("{}", x);
```

- `let` = variable
- `{}` = placeholder for printing

7.3. Rust Overview

- Rust is a **compiled**, **statically typed**, **systems-level** language.It has:
  - **Ownership system**
  - **Borrowing**
  - **Lifetimes**
  - **Zero-cost abstractions**
  - **Pattern matching (`match`)**
  - **Enums, structs, generics, traits**

Compared to other languages:

- Faster than Java/Python.
- Safer than C/C++.

7.4. Memory Safety

Memory safety means:

👉 **No crashes or corrupted memory because of wrong pointers.**

In C/C++, bad things happen:

- Dangling pointers
- Null pointer dereference
- Double free
- Race conditions

Rust **prevents** these at compile time using:

- Ownership
- Borrowing
- Lifetimes
- Immutability by default

This is Rust's biggest strength.

7.5. Null Pointers and Avoiding Null in Rust

C/C++ have **null pointers**, which often cause crashes. Rust has **no null**. Instead it uses:

Using `Option`

`Option<T>` expresses "there might be a value or not".

It has two cases:

- `Some(value)`
- `None`

Example:

```
let name: Option<String> = Some("Alice".to_string());
let empty: Option<String> = None;
```

Using `if let`:

```
if let Some(n) = name {
    println!("{}", n);
}
```

Rust forces you to **handle missing values safely**.

7.6. Digression: Primitive Types in Rust

Rust basic types:

- Integer: `i32`, `i64`
- Unsigned: `u32`, `u64`
- Float: `f32`, `f64`

- Boolean: `bool`
- Character: `char`
- String: `String` or `&str`

Rust is strict and safe:

👉 You must choose the type, cannot mix them accidentally.

7.7. Dangling Pointer Example (in C++)

A dangling pointer = pointer to memory that was freed.

```
int* p = new int(10);
delete p;
std::cout << *p;  // BAD: dangling pointer
```

Rust example (safe):

Rust refuses to compile any code that might create a dangling pointer.

Double Free Example (in C++) Double free = freeing memory twice.

```
int* p = new int(5);
delete p;
delete p; // crash
```

Rust: Impossible. Ownership rules prevent multiple frees.

Race Condition Example (in C++): Multiple threads write to same variable → unpredictable bug.

```
int x = 0;
thread a writes x++
thread b writes x++
```

Rust prevents this by:

- No shared mutable data without explicit locking.
- Compiler errors if unsafe sharing is detected.

7.8. Memory Management

Rust does **not** use a garbage collector. Rust uses:

👉 **Ownership system**

👉 **Moves**

👉 **Borrowing rules**

Memory is freed automatically when the **owner variable goes out of scope**.
Example:

```
{
    let s = String::from("hello");
} // here s is dropped automatically
```

7.9. Immutability By Default: Variables are **immutable** unless you add `mut`.

```
let x = 5;      // cannot change
let mut y = 10; // can change
```

Why?

👉 Prevents accidental modification (safer code).

7.10. RAII (Resource Acquisition Is Initialization)

Concept from C++. Meaning:

- When a variable is created → it "owns" a resource (memory, file, etc.)
- When the variable goes out of scope → resource is freed

Rust applies RAII to everything:

- Memory
- Files
- Sockets

Example:

```
{
    let file = File::open("test.txt");
} // file automatically closed here
```

7.11. Ownership System (Core of Rust)

Every value in Rust has: 👉 **one and only one owner**

When the owner goes out of scope: → The value is automatically freed.

Example:

```
let s = String::from("hello"); // s owns the string
```

Move Semantics: When you assign one variable to another, **ownership moves**.

```
let a = String::from("hi");
let b = a; // a is moved into b
```

After this:

- `b` is owner
- `a` is invalid (cannot be used)

Rust does this to prevent double free.

Same for function arguments:

```
fn foo(s: String) {}  // takes ownership

let x = String::from("hello");
foo(x);  // x is moved
```

Ownership: Unique Owner: Key rule:

👉 At any time, a value has exactly **one unique owner**. Why?

- Prevents memory bugs
- Prevents double free
- Prevents race conditions

✅ Lecture 8 – Rust: Borrowing, Lifetimes, Traits

1. Borrowing

Borrowing means: 👉 You can *use* a value **without owning it**.

Rust has **references** like `&T` (shared reference) and `&mut T` (mutable reference).

Why Borrowing?

- Avoids copying large data.
- Lets multiple parts of program read data safely.
- Ensures **no two owners** exist at the same time.

Borrowing Rules (VERY IMPORTANT FOR EXAM)

✔️ Rule 1 — **Any number of immutable (`&T`) references**

You can read the data many times safely.

✔️ Rule 2 — **Only ONE mutable (`&mut T`) reference at a time**

Because multiple writers cause bugs.

✔️ Rule 3 — **Immutable and mutable references cannot coexist**

This prevents "read while modifying," i.e., race condition.

✔️ Rule 4 — **References must ALWAYS be valid (compiler checks this with lifetimes)**

Examples

Immutable borrow

```
let s = String::from("hello");
let r1 = &s;   // borrow immutable
let r2 = &s;   // allowed
println!("{}", r1);
```

Mutable borrow

```
let mut x = 10;
let m = &mut x;   // allowed
*m += 1;
```

❌ **Error: two mutable borrows**

```
let mut x = 10;
let a = &mut x;
let b = &mut x;   // ERROR
```

❌ **Error: mutable + immutable**

```
let mut s = String::from("hi");
let r1 = &s;
let r2 = &s;
let r3 = &mut s;   // ERROR
```

Borrowing protects memory and prevents data races at compile time.

---

2. Strings in Rust

Rust has two important string types:

(1) `&str` **(string slice)**

- Immutable view of a string.
- Usually written like: `"hello"`.

(2) `String` **(owned string)**

- Growable, heap-allocated.
- Allows modifications.

Example:

```
let s1 = "hello";        // &str
let s2 = String::from("hi"); // String
```

Why Rust strings are complicated?

- UTF-8 encoding → each character has variable size.
- Rust ensures indexing is safe (no partial characters).

---

3. Lifetimes

Lifetimes = the **time during which a reference is valid**.

Rust uses lifetime annotations `'a` to ensure references never outlive the values they point to.

Why Lifetimes exist?

To avoid **dangling references** (using memory that no longer exists).

Lifetime Example

❌ Code with an invalid reference:

```
let r;
{
    let x = 5;
    r = &x; // x goes away after block ends
}
println!("{}", r); // ERROR
```

Compiler prevents the reference from living longer than the value.

Lifetimes in Functions

Sometimes Rust needs help to understand how long references will live.Example:

```
fn longest<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() > b.len() { a } else { b }
}
```

Meaning of `'a`:

👉 Returned reference must live at least as long as both `a` and `b`.

You don't manage memory manually — Rust just checks validity.

---

4. Enums: Algebraic Data Types (ADTs)

Enums in Rust can store different **kinds** of data. This makes Rust enums extremely powerful.

Simple Enum

```
enum Color {
    Red,
    Green,
```

```
    Blue,
}
```

Enum with Data (Algebraic Data Types)

```
enum Shape {
    Circle(f64),
    Rectangle(f64, f64),
}
```

You can store **values inside variants** — this is why Rust enums are called **Algebraic Data Types (ADTs)**.

Trees as ADT Example

A binary tree:

```
enum Tree<T> {
    Empty,
    Node(T, Box<Tree<T>>, Box<Tree<T>>)
}
```

This shows:

- ADTs can represent recursive structures.
- They can be **generic** (`<T>`).

Pattern Matching

`match` in Rust is a control-flow construct that lets you compare a value against patterns and run code based on which pattern matches. Rust uses `match` to work with enums:

```
match shape {
    Shape::Circle(r) ⇒ println!("Radius: {}", r),
    Shape::Rectangle(w, h) ⇒ println!("{}x{}", w, h),
}
```

Pattern matching is essential in Rust, especially for enums like `Option` or `Result`.

---

5. Classes in Rust = Struct + Impl

Rust doesn't have *classes* like in languages such as Java or C++, but the same idea is achieved using two:

- `struct` = data (fields) → only holds data. It has **no functions** attached by default.
- `impl` methods (functions for that struct) → block adds **methods** to the struct.

Example:

```
struct Person {
    name: String,
    age: u32,
}

impl Person {
    fn new(name: String, age: u32) -> Self {
        Self { name, age }
    }

    fn say_hi(&self) {
        println!("Hi, I'm {}", self.name);
    }
}
```

Usage:

```
let p = Person::new("Alice".into(), 30);
p.say_hi();
```

This is Rust's version of OOP.

---

6. Traits & System Traits

Traits = Rust's version of **interfaces,** a collection of method signatures that types can implement.

They define **what methods a type must have**, but not how they're implemented.

Example:

```
trait Speak {
    fn speak(&self);
}
```

A type implementing the trait:

```
impl Speak for Person {
    fn speak(&self) {
        println!("Hello!");
    }
}
```

Useful System Traits

Common built-in traits:

- `Debug` → allow printing values with `{:?}`
- `Clone` → make copies
- `Copy` → simple copy (for small types)
- `Eq` / `PartialEq` → comparisons
- `Ord` / `PartialOrd` → sorting
- `Iterator` → defines iterators
- `From` / `Into` → conversions
- `Display` → `println!("{}", value)`

Example: Debug

```
#[derive(Debug)]
struct Point { x: i32, y: i32 }
println!("{:?}", Point { x: 1, y: 2 });
```

✅ Lecture 9 – Rust: Smart Pointers, Unsafe

1. Smart Pointers. A **smart pointer** is like a normal pointer, **but with extra behavior**, such as:

- automatic memory management
- counting references
- runtime borrow checking
- interior mutability

Rust includes several smart pointer types.

---

2. `Box<T>`

What it is:

- `Box<T>` stores data **on the heap**.
- Simple, safe, owns exactly **one value**.
- No reference counting.

Why use it?

- When you need data on the heap.
- For recursive types (trees, linked lists).

Example

```
let b = Box::new(5);
println!("{}", b);
```

---

3. `Rc<T>` — Reference Counted Pointer

What it is: `Rc<T>` lets you have **multiple owners** of the same heap data.

Key idea

- Only for **read-only** shared data.
- Not thread-safe (single-thread only).
- Keeps a **reference count**.

Example

```
use std::rc::Rc;

let a = Rc::new(10);
```

```
let b = Rc::clone(&a);
println!("{}", a); // two owners
```

---

4. `RefCell<T>` — Interior Mutability

What it is: A smart pointer that allows you to **mutate data even if it's not mutably owned**.

It performs **borrow checking at runtime**, not at compile time.

Why needed: Sometimes you want shared mutable data, but Rust's rules forbid it at compile time.

Example

```
use std::cell::RefCell;
let x = RefCell::new(5);
*x.borrow_mut() = 10;
```

Rc + RefCell (very common pattern)

Allows:

- **multiple owners** (`Rc`)
- **with mutable interior** (`RefCell`)

Used for trees, graphs, GUIs.

---

5. Closures

Closures are **anonymous functions** that can capture variables from their environment.

Example

```
let x = 5;
let add = |y| x + y;
println!("{}", add(3)); // prints 8
```

Closures can capture:

- by copy (`Fn`)
- by mutable borrow (`FnMut`)

- by taking ownership (`FnOnce`)

Rust infers automatically.

---

6. Iterators. Iterators allow processing collections in a **functional style**.

Example

```
let v = vec![1, 2, 3];
let sum: i32 = v.iter().map(|x| x * 2).sum();
println!("{}", sum); // 12
```

Common iterator methods:

- `.map()` → transform elements
- `.filter()` → keep some elements
- `.sum()` → sum values
- `.collect()` → build vectors, strings, etc.

---

7. Race Conditions & How Rust Avoids Them

A **race condition** happens when:

- two threads access the same memory
- at least one writes
- no synchronization

Rust **prevents race conditions at compile time** using:

✔️ Unique mutable access. Only **one mutable reference** allowed → no concurrent writes.

✔️ Ownership system. Data cannot be used after being moved.

✔️ "Send" and "Sync" traits. These ensure safe sharing and safe sending of data across threads.

Rust's type system guarantees safety **before the program runs**, avoiding the bugs C++ or Java can have.

---

8. Traits `Send` and `Sync` (marker traits)

Send: A type is `Send` if it is safe to **transfer ownership** to another thread. (Owned types: integers, Strings, Vec, etc.)

Sync: A type is `Sync` if it is safe for **multiple threads** to hold immutable references simultaneously.

Important: You **do not implement these yourself**—Rust does it automatically unless unsafe.

These two traits are the foundation of Rust's safe concurrency.

---

9. "What if mutable sharing is necessary?"

Sometimes you NEED: multiple owners, with mutable data, across threads

Rust **normally forbids** this because it's dangerous. So what's the solution?

---

10. Rust's Solution: Raw Pointers + `unsafe`

Rust provides:

- `mut T` = raw mutable pointer
- `const T` = raw constant pointer

They **do not follow** Rust's safety rules. You must handle everything manually.

When do we use `unsafe`?

- Call C code
- Implement smart pointers
- Create data structures the compiler can't reason about
- Mutably share memory across threads (with locks)

Unsafe superpowers: Inside an `unsafe{}` block, you can do things Rust normally forbids:

- Dereference raw pointers
- Call unsafe functions
- Modify static mut variables
- Implement unsafe traits

Example: Raw pointers = dangerous but powerful → used for low-level systems.

```
let mut x = 10;
let raw = &mut x as *mut i32;
```

```
unsafe {
   *raw = 20;
}
```

---

11. RustBelt — Formal Proof of Rust's Correctness

RustBelt is an academic project proving that Rust's key safety guarantees are **mathematically correct**. **What it proves:**

- Ownership model is sound
- Borrowing is correct
- Unsafe abstractions (like RefCell, Rc) still behave safely
- Traits and type system prevent memory bugs

Why important: It gives confidence that Rust's "memory safety without garbage collector" is not just "probably safe," but **proven safe by formal logic**. RustBelt validates the core of Rust's design.

✅ Lecture 10 – Python: Reference counters & GIL + Syntactic Sugar in F#, C#

1. Garbage Collection in Python

Python uses **automatic memory management**, mainly based on:

1. Reference Counting (main system)

Every object has a **reference counter**. When something points to an object, counter **+1**. When a reference disappears, counter **–1**. When counter becomes **0**, memory is freed immediately.

Example

```
a = [1,2,3]
b = a    # counter = 2
del a    # counter = 1
del b    # counter = 0 → memory freed
```

2. Cycle Garbage Collector (backup system)

Handles *reference cycles* like:

```
a = {}
a['self'] = a
```

Reference counter never reaches 0, so Python has a separate cycle detector that removes these.

---

2. Memory Safety in Python

Python is **memory-safe**, because:

- No pointer arithmetic
- No manual malloc/free
- Buffer overflows are rare (protected by runtime)
- Type safety is enforced at runtime

This removes common C/C++ errors:

- use-after-free
- double free
- dangling pointers
- accessing invalid memory

But Python is NOT "fully safe"—you can get:

- memory leaks from global references
- reference cycles that GC does not collect automatically

However, Python avoids *dangerous* memory bugs.

---

3. Race Conditions in Python?

Yes, Python code **can** have race conditions:

- when **multiple threads** modify shared data
- without proper synchronization

Example:

```
import threading
```

```
x = 0

def add():
    global x
    for _ in range(100000):
        x += 1

t1 = threading.Thread(target=add)
t2 = threading.Thread(target=add)
t1.start(); t2.start()
t1.join(); t2.join()

print(x)  # often NOT 200000
```

Even though Python has the GIL (explained below), **operations on Python objects are not atomic**. But Python does avoid **memory corruption**, because the GIL ensures only one thread runs Python bytecode at a time.

---

4. Handling Reference Counters

Reference counts are updated **atomically** because of the **GIL**. This prevents:

- double free
- freeing objects from one thread while another uses them
- concurrent modification of reference counters

This is one reason the **GIL exists**: to make CPython simple and memory-safe.

---

5. Concurrency in Python. Python supports:

Threads

- Good for I/O-bound code (networking, file I/O)
- Not good for CPU-bound tasks because of the GIL

Multiprocessing

- Runs separate processes
- True parallelism
- Recommended for CPU-heavy tasks

Asyncio

- Lightweight concurrency with coroutines
- Ideal for many simultaneous I/O tasks (servers, APIs)

---

6. The Global Interpreter Lock (GIL)

Definition: The **GIL** is a **mutex (lock)** that allows **only one thread** to execute Python bytecode at a time.

Why does it exist?

- Simplifies memory management
- Makes reference counting safe
- Avoids race conditions inside CPython interpreter
- Makes implementation easier

Important: The GIL **does NOT prevent race conditions on Python variables**, it only prevents *multi-threaded parallel CPU execution* of Python code.

Example Demonstration

```
import threading

def work():
    for i in range(5_000_000):
        pass   # CPU work but NOT parallel

threads = [threading.Thread(target=work) for _ in range(4)]
for t in threads: t.start()
for t in threads: t.join()
```

It uses **one CPU core**, even with 4 threads.

---

7. More on the GIL — Important Properties

Threads are useful only for I/O

Because threads release the GIL during:

- waiting for network
- waiting for disk
- waiting for external libraries (C extensions)

CPU work stays single-threaded. C extensions can bypass the GIL

NumPy, Pandas, TensorFlow run parallel C code.

Example:

- NumPy loops run in C
- They release GIL
- They use multiple CPU cores

This is why NumPy is fast.

---

8. Alternatives to the GIL. Several ideas exist:

1. No GIL (experimental CPython project): Python 3.13 and later includes an experimental **free-threaded** build.

2. Use Multiprocessing: True parallelism by starting multiple processes:

```
from multiprocessing import Pool

def square(x): return x*x

with Pool() as p:
    print(p.map(square, range(10)))
```

3. Use external libraries that release the GIL: NumPy, Numba (JIT compiler), Cython, Rust extensions, C/C++ extensions

4. Use other Python implementations

- **PyPy STM** (Software Transactional Memory) – experimental
- **Jython** (runs on JVM, no GIL)
- **IronPython** (.NET version, no GIL)

5. Use async/await for concurrency: Great for servers and networking.

1. What is Syntactic Sugar

Syntactic sugar is language syntax designed to make code **shorter, clearer, or more expressive** without adding new computational power.

- It does **not change semantics**

- It is **translated by the compiler** into more verbose core constructs
- It improves **readability and developer productivity**

---

2. Syntactic Sugar in F#

F# is intentionally **rich in syntactic sugar** to support functional programming.

2.1 Function Definition. **Sugar**

```
let add x y = x + y
```

Desugared (conceptually)

```
let add = fun x → fun y → x + y
```

➡️ Functions are **curried by default**.

3. Syntactic Sugar in C#

C# adds sugar progressively while keeping a **nominal, OO core**.

3.1 Properties

Sugar

```
publicint Age {get;set; }
```

Desugared

```
privateint _age;
publicintget_Age() ⇒ _age;
publicvoidset_Age(intvalue) ⇒ _age =value;
```

✅ Lecture 11 – First-class functions, lambdas, higher-order functions in F#

Functional programming is a style of programming that emphasizes the use of functions and immutable data. Typed functional programming is when functional programming is combined with static types, such as with F#. In general, the following concepts are emphasized in functional programming:

- Functions as the primary constructs you use
- Expressions instead of statements
- Immutable values over variables
- Declarative programming over imperative programming

Throughout this series, you'll explore concepts and patterns in functional programming using F#. Along the way, you'll learn some F# too.

Terminology

Functional programming, like other programming paradigms, comes with a vocabulary that you will eventually need to learn. Here are some common terms you'll see all of the time:

- **Function** - A function is a construct that will produce an output when given an input. More formally, it *maps* an item from one set to another set. This formalism is lifted into the concrete in many ways, especially when using functions that operate on collections of data. It is the most basic (and important) concept in functional programming.
- **Expression** - An expression is a construct in code that produces a value. In F#, this value must be bound or explicitly ignored. An expression can be trivially replaced by a function call.
- **Purity** - Purity is a property of a function such that its return value is always the same for the same arguments, and that its evaluation has no side effects. A pure function depends entirely on its arguments.
- **Referential Transparency** - Referential Transparency is a property of expressions such that they can be replaced with their output without affecting a program's behavior.
- **Immutability** - Immutability means that a value cannot be changed in-place. This is in contrast with variables, which can change in place.

1. First-Class Functions

A programming language supports **first-class functions** if functions are treated as values. This means a function can:

- Be assigned to a variable
- Be passed as an argument
- Be returned from another function

- Be stored in data structures

F# fully supports first-class functions.

Example (F#)

```
let add x y = x + y

let f = add        // function assigned to a value
let result = f 2 3 // result = 5
```

Here, `add` is not executed when assigned to `f`; it is treated as a value.

---

2. Lambda Expressions (Anonymous Functions)

A **lambda expression** is a function defined **without a name**. It is typically used for short, local behavior passed directly as an argument. In F#, lambdas use the syntax:

```
fun parameters → expression
```

Example (F#)

```
let square = fun x → x * x
let result = square 4  // result = 16
```

Lambda used inline:

```
List.map (fun x → x * 2) [1; 2; 3]
// result: [2; 4; 6]
```

---

3. Higher-Order Functions

A **higher-order function** is a function that:

- Takes one or more functions as parameters, **and/or**
- Returns a function as a result

Higher-order functions rely on first-class functions.

Example 1: Function as Argument

```
let applyTwice f x =
    f (f x)

let result = applyTwice (fun x -> x + 1) 3
// result = 5
```

Here, `applyTwice` takes a function `f` as an argument.

Example 2: Function as Return Value

```
let makeAdder x =
    fun y -> x + y

let add5 = makeAdder 5
let result = add5 10
// result = 15
```

This example also demonstrates **closures**: the returned function remembers `x`.

---

4. Relationship Between the Concepts

Concept	Role
First-class functions	Enables functions to be treated as values
Lambda expressions	Provide a concise way to create functions
Higher-order functions	Operate on functions as inputs or outputs

In F#, all three are foundational and used extensively in:

- `List.map`, `List.filter`, `List.fold`
- Function composition
- Pipelines (`|>`)

---

5. Canonical F# Example Combining All Three

```
let process f =
    List.map f [1; 2; 3]
```

```
process (fun x -> x * x)
// result: [1; 4; 9]
```

- `process` is a higher-order function
- `f` is a first-class function
- `(fun x -> x * x)` is a lambda expression

❌ Lecture 12 – Computational Expressions in F#, Monads

Computation expressions in F# provide a convenient syntax for writing computations that can be sequenced and combined using control flow constructs and bindings. Depending on the kind of computation expression, they can be thought of as a way to express monads, monoids, monad transformers, and applicative functors. However, unlike other languages (such as *do-notation* in Haskell), they are not tied to a single abstraction, and do not rely on macros or other forms of metaprogramming to accomplish a convenient and context-sensitive syntax.

Overview

Computations can take many forms. The most common form of computation is single-threaded execution, which is easy to understand and modify. However, not all forms of computation are as straightforward as single-threaded execution. Some examples include:

- Non-deterministic computations
- Asynchronous computations
- Effectful computations
- Generative computations

More generally, there are *context-sensitive* computations that you must perform in certain parts of an application. Writing context-sensitive code can be challenging, as it is easy to "leak" computations outside of a given context without abstractions to prevent you from doing so. These abstractions are often challenging to write by yourself, which is why F# has a generalized way to do so called **computation expressions**.

Computation expressions offer a uniform syntax and abstraction model for encoding context-sensitive computations.

Every computation expression is backed by a *builder* type. The builder type defines the operations that are available for the computation expression. See Creating a New Type of Computation Expression, which shows how to create a custom computation expression.

Syntax overview

All computation expressions have the following form:

F#**Copy**

```
builder-expr { cexper }
```

In this form, `builder-expr` is the name of a builder type that defines the computation expression, and `cexper` is the expression body of the computation expression. For example, `async` computation expression code can look like this:

F#**Copy**

```
let fetchAndDownload url =
    async {
        let! data = downloadData url

        let processedData = processData data

        return processedData
    }
```

There is a special, additional syntax available within a computation expression, as shown in the previous example. The following expression forms are possible with computation expressions:

F#**Copy**

```
expr { let! ... }
expr { and! ... }
expr { do! ... }
expr { yield ... }
expr { yield! ... }
```

```
expr { return ... }
expr { return! ... }
expr { match! ... }
```

Each of these keywords, and other standard F# keywords are only available in a computation expression if they have been defined in the backing builder type. The only exception to this is `match!`, which is itself syntactic sugar for the use of `let!` followed by a pattern match on the result.

The builder type is an object that defines special methods that govern the way the fragments of the computation expression are combined; that is, its methods control how the computation expression behaves. Another way to describe a builder class is to say that it enables you to customize the operation of many F# constructs, such as loops and bindings.

`let!`

The `let!` keyword binds the result of a call to another computation expression to a name:

F#**Copy**

```
let doThingsAsync url =
    async {
        let! data = getDataAsync url
        ...
    }
```

If you bind the call to a computation expression with `let`, you will not get the result of the computation expression. Instead, you will have bound the value of the *unrealized* call to that computation expression. Use `let!` to bind to the result.

`let!` is defined by the `Bind(x, f)` member on the builder type.

1. Computation Expressions in F#

A **computation expression (CE)** in F# is a syntactic construct that allows you to write code in a structured, imperative-like style while being translated by the compiler into **method calls on a builder object**.

They are **syntax sugar** over function composition and method chaining.
General form:

```
builder {
    ...
}
```

Examples of built-in computation expressions:

- `option { ... }`
- `async { ... }`
- `task { ... }`
- `seq { ... }`

---

2. Why Computation Expressions Exist

Problem:

- Functional composition becomes hard to read when chaining many computations.
- Error handling, async flow, state, or context passing causes deeply nested code.

Solution:

- Computation expressions provide **readable control flow** while preserving functional semantics.

---

3. Monads (Programming Languages Perspective)

A **monad** is an abstraction that:

1. Wraps a value in a context
2. Provides a way to sequence computations that operate on wrapped values
3. Handles the context implicitly

Formally, a monad consists of:

- `return` (also called `unit`): `a -> M<a>`
- `bind` (`>>=`): `M<a> -> (a -> M<b>) -> M<b>`

---

6. Meaning of `let!`, `return`, and `do!`

Keyword	Meaning
`let! x = m`	Bind monadic value `m` and extract `x`
`return x`	Wrap value into the monadic context
`do! m`	Execute monadic computation, ignore result

✅ Lecture 13 – Quotations and Meta-programming

1. Meta-programming

Meta-programming is the technique of writing programs that **analyze, generate, or transform other programs (or themselves) as data**.

Core goals:

- Program inspection
- Program generation
- Program transformation

In F#, meta-programming is primarily supported via **quotations**.

2. Quotations in F#

A **quotation** is a typed representation of F# code as an **abstract syntax tree (AST)** that is available **at runtime**. Instead of executing code, quotations **capture its structure**.

Syntax:

```
<@ expression @>
```

Type:

```
Expr<'T>
```

3. Simple Quotation Example

```
open Microsoft.FSharp.Quotations
```

```
let expr = <@ 1 + 2 @>
```

- The expression is **not evaluated**
- `expr` contains the syntax tree of `1 + 2`
- Type: `Expr<int>`

---

4. Quotation vs Normal Evaluation

```
let x = 1 + 2      // x = 3
let q = <@ 1 + 2 @> // q = AST
```

Key difference:

- Normal code → value
- Quotation → structure

---

5. Typed Quotations

F# quotations are **strongly typed**.

```
let addExpr : Expr<int> = <@ 3 + 4 @>
```

This guarantees:

- Type safety
- No invalid ASTs at runtime

---

6. Inspecting Quotations (Pattern Matching on AST)

Quotations can be deconstructed using pattern matching.

```
open Microsoft.FSharp.Quotations.Patterns

let analyze expr =
    match expr with
    | Value(v, _) → printfn "Value: %A" v
    | Call(_, methodInfo, args) →
        printfn "Method call: %s" methodInfo.Name
    | _ → printfn "Other expression"
```

Usage:

```
analyze <@ 10 + 20 @>
```

---

7. Splicing and Anti-Quotations

Splicing allows inserting an existing quotation into another quotation.

Syntax:

```
%expr
```

Example

```
let x = <@ 1 + 2 @>

let y = <@ %x * 3 @>
```

Resulting AST represents:

```
(1 + 2) * 3
```

---

8. Quotation Evaluation (Limited)

Quotations are **not normally executed**, but they can be compiled or evaluated using reflection or libraries.

Example (simple evaluation):

```
open Microsoft.FSharp.Linq.RuntimeHelpers

let value =
    LeafExpressionConverter.EvaluateQuotation <@ 5 + 6 @>
```

Result:

```
value = 11
```

Use cases:

- Dynamic execution

- Expression compilation

---

11. Limitations of Quotations

- Cannot capture all .NET constructs
- Limited support for mutable state
- Performance overhead
- Mostly used in advanced scenarios

✅ Lecture 14 – Reflection and Meta-programming

2. Reflection

Reflection is a runtime mechanism that allows a program to:

- Inspect types (проверять)
- Inspect members (methods, fields, properties)
- Discover attributes
- Dynamically invoke code

Reflection operates on **metadata emitted by the compiler** and stored in assemblies. On .NET, reflection lives in:

```
System.Reflection
```

---

3. What Reflection Can Inspect

Reflection can inspect:

- Assemblies
- Types (classes, records, unions)
- Methods and functions
- Properties and fields
- Attributes
- Generic type information

It **cannot** inspect source code or expressions.

---

4. Basic Reflection Example (F#)

Inspecting a Type

```
open System

type Person = {
    Name : string
    Age : int
}

let t = typeof<Person>

printfn "%s" t.FullName
```

---

7. Attributes and Annotations

Attributes attach metadata to program elements, consumable via reflection.

Example

```
[<Obsolete("Use NewFunction")>]
let oldFunction x = x + 1
```

Reading attribute:

```
let attrs =
    typeof<Calculator>.GetCustomAttributes(true)
```

---

8. Reflection vs Quotations

Aspect	Reflection	Quotations
Time	Runtime	Compile-time capture
Data	Metadata	Typed AST
Type safety	Low	High
Use cases	Frameworks, serializers	DSLs, compilers

Reflection:

- Answers **“what exists?”**

Quotations:

- Answer **“what code does?”**

---

9. Reflection Use Cases

- Dependency injection containers
- ORMs and serializers
- Plugin systems
- Test frameworks
- Runtime discovery of types

---

10. Costs and Limitations

Costs

- Slower than direct calls
- Requires boxing/unboxing
- Harder to reason about

Limitations

- No source-level information
- Weak compile-time guarantees
- Runtime failures possible

---

11. Reflection and Safety

- Reflection bypasses encapsulation
- Can break invariants
- Should be isolated and controlled

In F#, reflection is often wrapped in:

- Active patterns
- Generic helpers
- Compile-time checks where possible

✅ Lecture 15 – Type systems, Polymorphism - ad hoc vs universal

1. Types in Programming Languages

A **type** describes:

- what kind of data a value is, and
- what operations are allowed on it.

Examples

- `int` → numbers
- `bool` → true / false
- `string` → text

Why types exist

- prevent errors (e.g. adding a number to a string),
- help the compiler understand programs,
- improve readability and correctness.

---

2. Type Systems and Type Checking

A **type system** is a set of rules that:

- assigns types to expressions,
- checks whether operations are used correctly.

Type checking means verifying that:

- variables are used consistently,
- operations match operand types.

Example

```
x = 5
x = "hello"   # allowed in Python
```

In some languages this is an error, in others it is allowed.

---

3. Strongly Typed, Dynamically Typed, Statically Typed

Statically Typed: Types are checked **before the program runs**. **Example**

```
int x = 5;
x = "hello";   // error at compile time
```

Dynamically Typed: Types are checked **while the program runs**. **Example**

```
x = 5
x = "hello"    # allowed
```

Strongly Typed: The language **does not allow unsafe type conversions**.

Example: `"5" + 2   # error`

Weakly Typed: The language allows implicit, possibly unsafe conversions.

Example (C): `int x = 'A';   // allowed (ASCII value)`

Important note

- Static vs Dynamic → *when* types are checked
- Strong vs Weak → *how strict* the rules are

They are **independent concepts**.

---

4. Extensible Type Systems: Three Views of Types

Types can be seen in three ways:

1. **Set of values (Denotational)**
   - `int` = all integers
2. **Set of operations (Constructive)**
   - `int` allows `+`,
   - `bool` allows logical operations
3. **Abstraction-based / Interface**
   - a type defines *what you can do*, not *how it works*

Extensible type systems allow **new types** to be added without breaking existing code.

---

5. Primitive, Composite, and Subroutine Types

Primitive Types: Basic built-in types.

- `int`
- `float`
- `bool`
- `char`

Composite Types: Built from other types.

- arrays
- lists
- structs
- tuples

Example: `point = (3, 4)`

Subroutine Types: Types of functions.

- input types
- output type

Example: `int → int`

A function that takes an integer and returns an integer.

Subroutine types can be:

- **First-class:** can be passed as parameter, returned, assigned
- **Second-class:** passed as parameter only
- **Third class:** not even passed as parameter

---

1. Polymorphism

Polymorphism is the ability of a single program element (function, method, or operator) to operate on **values of different types**.

The key idea:

> one interface, multiple types, correct behavior for each type.

---

2. Classification of Polymorphism

In programming languages theory, polymorphism is typically divided into:

1. **Ad hoc polymorphism**

2. **Universal polymorphism**

---

3. Ad Hoc Polymorphism

Ad hoc polymorphism occurs when a function or operator works on different types, but **each type has its own separate implementation**. There is **no single uniform definition** of the function.

Characteristics

- Type-specific behavior
- Resolved at compile time
- Implementations are unrelated internally

Common Mechanisms

- Function overloading
- Operator overloading

Example: Function Overloading (C#-style)

```
intAdd(int x,int y)
doubleAdd(double x,double y)
```

The name `Add` refers to different implementations.

Example: Operator Overloading (F#)

```
let inline add x y = x + y

add 1 2       // integer addition
add 1.5 2.5   // floating-point addition
```

Here:

- `+` is overloaded
- Each type defines its own implementation

Key Point

> The behavior depends on the specific types involved.

---

4. Universal Polymorphism

Universal polymorphism occurs when a **single implementation** works uniformly for **all types**.

The behavior does **not depend on the concrete type**, only on its structure or abstraction.

7. Binding Time. **Binding time** = when a name is associated with something. Examples:

- variable to type
- function name to code
- operator to implementation

Common binding times

- compile time
- run time

Example: `x = 5` . The type of `x` is bound at **runtime**.

---

8. Overloading (Ad Hoc Polymorphism)

Ad hoc polymorphism means:

- same name,
- different implementations,
- chosen by argument types.

Example:

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

Overloading in Haskell. Haskell uses **type classes**.

Example: `(+) :: Num a ⇒ a → a → a`

The same `+` works for `Int`, `Float`, etc., but each has its own implementation.

---

9. Coercion (Universal Polymorphism)

Coercion means: automatic conversion from one type to another. **Example**

```
int x = 3;
double y = x;   // int → double
```

This is **universal polymorphism** because:

- one function works for many types,
- conversion happens automatically.

---

10. Inclusion Polymorphism

Used in **object-oriented languages**. A value of a **subtype** can be used where a **supertype** is expected.

Example: `Animal a = new Dog();`

- `Dog` is an `Animal`
- works without conversion

This is also called **subtyping polymorphism**.

---

11. Overriding: Overloading + Overriding

Overriding: A subclass **redefines** a method of its parent class. **Example:**

```
class Animal {
   void speak() { }
}

class Dog extends Animal {
   void speak() { System.out.println("Bark"); }
}
```

The method is chosen at **runtime** based on the object.

Difference

Concept	Happens when	Based on
Overloading	Compile time	Parameter types
Overriding	Runtime	Object type

✅ Lecture 16 – Parametric polymorphism: Java generics

1. Java Generics: Purpose, Classic Example

Java Generics allow you to:

- write **type-safe code**,
- avoid casting,
- write **reusable algorithms** that work for many types.

This is **parametric polymorphism**: the same code works for **any type**, chosen as a parameter.

Classic Example (Without Generics)

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0); // cast needed
```

Problems: unsafe, runtime errors possible.

With Generics

```
List<String> list = new ArrayList<>();
String s = list.get(0); // no cast
```

---

2. Generic Method

A **generic method** declares its own type parameter. **Example**

```
public static <T> T identity(T x) {
    return x;
}
```

Usage:

```
int a = identity(5);
String s = identity("hi");
```

The method works for **any type T**.

---

3. Bounded Type Parameters

Sometimes we want **generic types with restrictions**.

Example: `<T extends Number>`

Means: `T` must be `Number` or a subclass of `Number`.

---

4. Type Bounds

Type bounds restrict what types are allowed.

Single Bound: `<T extends Number>`

Multiple Bounds: `<T extends Number & Comparable<T>>`

Meaning:

- `T` must be a `Number`
- and implement `Comparable`.

---

5. A Generic Algorithm with Type Bounds

Example: max function

```
public static <T extends Comparable<T>> T max(T a, T b) {
   return (a.compareTo(b) > 0) ? a : b;
}
```

Why bounds? because `compareTo()` must exist.

---

6. Generics and Other Language Features

Generics work with:

- classes,
- interfaces,
- methods,
- collections.

But **not with primitives**.

❌ `List<int>`

✅ `List<Integer>`

---

7. Generics and Subtyping

Important rule in Java:

> Generics are invariant. Even if:

`Integer extends Number`

this is **false**: `List<Integer> ≠ List<Number>`

Why? to preserve type safety.

---

8. Java Rules: List<Number> and List<Integer> (Contravariance)

What is NOT allowed: `List<Number> ln = new ArrayList<Integer>(); // error`

Why? If it were allowed: `ln.add(3.14); // double`

This would break `List<Integer>`.

Contravariance

Handled using **wildcards** (explained later).

---

9. Generics and Subtypes in C#

C# supports:

- **covariant generics** (`out`)
- **contravariant generics** (`in`)

Example: `IEnumerable<out T>`

Java does **not** support this directly → uses wildcards instead.

---

10. Arrays and Generics

Java arrays are **covariant**.

```
Integer[] ints = new Integer[10];
Number[] nums = ints; // allowed
```

Generics are **invariant**.

```
List<Integer> li;
List<Number> ln = li; // not allowed
```

---

11. Problems with Array Covariance

Example:

```
Number[] nums = new Integer[2];
nums[0] = 3.14; // runtime error!
```

Arrays check types **at runtime**, not compile time.

---

12. Type Erasure (Recall)

Java generics use **type erasure**:

- generic type information is removed at runtime,
- all `T` become `Object` (or bound).

Example: `List<String> → List`

This is why:

- no `new T()`,
- no `instanceof T`.

---

13. Array Covariance and Generics

Because:

- arrays know their type at runtime,
- generics do not,

Java **forbids generic arrays**.

❌ `new List<String>[10];`

---

14. Wildcards for Covariance & Contravariance

Wildcards allow **flexibility**.

Covariance: `List<? extends Number>`

- can read `Number`
- cannot add elements

Contravariance: `List<? super Integer>`

- can add `Integer`
- reading gives `Object`

---

15. PECS Principle

PECS = Producer Extends, Consumer Super

- If you **read** from a structure → `extends`
- If you **write** to a structure → `super`

Examples

```
List<? extends Number> producer;
List<? super Integer> consumer;
```

---

16. Price to Pay with Wildcards: Type Safety

Wildcards increase flexibility but:

- reduce precision,
- sometimes prevent operations.

Example:

```
List<? extends Number> list;
list.add(3); // error
```

Compiler blocks unsafe operations.

---

17. Limitations of Java Generics

- No primitives (`int` , `double`)
- Type erasure (no runtime info)
- No generic arrays
- No `instanceof T`
- Cannot create `new T()`

✅ Lecture 17 – Type classes in Haskell

1. Core Haskell

- Haskell is a **purely functional language**: no side effects, all functions return values.
- Everything is **immutable**: once you define a variable, it cannot change.

- Functions are **first-class**: they can be passed as arguments, returned, stored in data structures.
- Example: `square x = x * x`

---

2. Polymorphism in Haskell

- **Polymorphism** means **one function can work with multiple types**.
- Haskell has **two main kinds of polymorphism**:
  1. **Parametric polymorphism**: function works for **any type**.

     ```
     identity :: a → a
     identity x = x
     ```

     Here, `a` can be **Int, String, Bool**, etc.
  2. **Ad hoc polymorphism**: function works for **some types with specific behavior**.

     Implemented via **type classes**.

---

3. Ad Hoc Polymorphism

- **Ad hoc polymorphism** same function name works differently for different types.
- Haskell uses **type classes** for this.
- Examples: `+`, `==`, `show`.

Haskell Type Classes — what you must understand for the exam

What a type class is (core idea)

A *type class* defines a **set of operations (functions)** that a type must implement.

It is **not** a class like in OOP. Think of it as an **interface / contract for behavior**.

> "If a type belongs to this type class, it promises to provide these functions."

Why type classes exist

They enable **ad-hoc polymorphism**: the same function name works for different types, but **each type can have its own implementation**.

Example idea (no syntax focus):

- `+` works for `Int`, `Float`, etc.
- Equality works for many types, but not all.

Type class vs instance (must be clear)

- **Type class** → *what functions are required*
- **Instance** → *how a specific type implements them*

Example conceptually:

- Type class: "Things that can be compared for equality"
- Instance: "Integers can be compared like this"

Important built-in type classes to know

- `Eq` → equality (`==`, `/=`)
- `Ord` → ordering (`<`, `>`, `compare`)
- `Show` → convert to string
- `Read` → parse from string
- `Num` → numeric operations
- `Functor` → mapping over a structure (`fmap`)

You don't need all methods—**just know what behavior each represents**.

Type class constraints (very important)

Functions can **require** a type class: Meaning in words:

> "This function works for any type as long as that type supports these operations."

Example idea:

- A function that uses `==` requires `Eq`.

Difference from generics (exam favorite)

- Generics → same code for all types

- Type classes → same function name, **different behavior per type**

This is **ad-hoc polymorphism**, not parametric.

✅ Lecture 19 – .NET Runtime

What is the .NET Runtime

The **.NET Runtime** is the **core execution environment** of the .NET platform.

It is responsible for **running .NET applications**, managing their **execution**, and providing **low-level services** such as memory management, type safety, and exception handling.

The main .NET runtime is called the **CLR (Common Language Runtime)**.

How .NET code runs

1. You write code in a .NET language (C#, F#, VB)
2. The compiler translates the code into **Intermediate Language (IL)**
3. The **.NET Runtime**:
    - Loads the IL
    - Verifies it for safety
    - Compiles it to native machine code using **JIT (Just-In-Time) compilation**
    - Executes it on the operating system

```
SourceCode → IL →.NET Runtime → NativeCode → OS / Hardware
```

Main responsibilities of the .NET Runtime

1. Code execution

- Converts IL to native code (JIT)
- Manages method calls and execution flow

2. Memory management

- Allocates memory for objects
- Automatically frees unused memory via **Garbage Collection (GC)**

3. Type safety

- Ensures correct use of types at runtime
- Prevents invalid memory access

4. Exception handling

- Provides a unified mechanism for runtime errors
- Ensures consistent stack unwinding

5. Threading and concurrency

- Manages threads and thread pools
- Supports async and parallel execution

6. Interoperability

- Allows .NET code to call native libraries (C/C++)
- Handles **marshaling** between managed and unmanaged code

Why the .NET Runtime matters

Without the runtime, .NET would be just a language specification. The runtime:

- Makes applications **portable**
- Makes execution **safe and predictable**
- Removes the need for manual memory management
- Enables multiple languages to run on the same platform

❌ Lecture 21 – Laziness, Constructor Classes and Monads in Haskell

1. Laziness: pros & cons

What is laziness? Laziness means:

> Expressions are not evaluated until their value is actually needed.

Pros

- Better performance (avoid unnecessary computation)
- Can work with **infinite data structures**
- Clear separation between **what** to compute and **when**

Cons

- Harder to predict performance
- Memory leaks if values are kept unevaluated too long
- Debugging can be harder

---

2. Laziness in programming languages

Languages can evaluate expressions in different ways:

- **Eager (strict)**: evaluate arguments first. Example: C, Java, Python
- **Lazy (non-strict)**: evaluate only when needed. Example: Haskell

Most languages are eager. Haskell is **lazy by default**.

---

3. Applicative and Normal Order evaluation (λ-calculus)

In theory (lambda calculus):

- **Applicative order** (eager): Evaluate function arguments first
- **Normal order** (lazy): Replace expressions first, evaluate only if needed

Important result:

> Normal order finds a result if one exists

This is the theoretical foundation of laziness.

---

4. Laziness in Haskell

Haskell uses **lazy evaluation everywhere**. Example:

```
x = 1 / 0
y = 5
```

This program does **not crash**, because `x` is never used.

Another example: `take 5 [1..]`

Works, even though `[1..]` is infinite.

---

5. Binding variables

In Haskell: `x = expensiveComputation`

This does **not run** the computation immediately.

It creates a **binding** to a future computation (a thunk).

The computation happens **only when x is needed**.

---

6. Type Constructor Classes – Example: `map`

map

```
map :: (a → b) → [a] → [b]
```

- `map` does not care about concrete types
- Works on lists of **any type**

Key idea: map works on a type constructor ([]), not a concrete type

---

7. Constructor Classes

A **type constructor**: Takes a type and produces a new type

Examples:

- `[]` → `[Int]`, `[String]`
- `Maybe` → `Maybe Int`, `Maybe String`

A **constructor class**: A type class whose instances are **type constructors**

---

8. The Functor constructor class and instances

Functor

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

Meaning:

> If something can be “mapped over”, it is a Functor

Instances

```
instance Functor [] where
  fmap = map
```

```
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing  = Nothing
```

Intuition:

- List → apply function to each element
- Maybe → apply function if value exists

---

9. Constructor Classes (summary)

Constructor classes:

- Abstract over **shapes of data**
- Let us write **generic code**
- Prepare the ground for **monads**

---

10. Towards Monads

Problem:

- How to **chain computations**
- When computations may **fail**, **produce effects**, or **depend on context**

Monads solve: How to sequence computations in a controlled way

---

11. The Maybe type constructor

```
data Maybe a = Just a | Nothing
```

Used for:

- Partial functions
- Failure handling
- Avoiding null pointers

Example:

```
safeDiv x 0 = Nothing
safeDiv x y = Just (x / y)
```

---

12. Composing partial functions

Problem: `safeDiv 10 2 >>= ?`

If one step fails (`Nothing`), everything should fail.

Manual checks are ugly and repetitive.

---

13. Bind (`>>=`) and the Maybe monad

Bind

```
(>>=) :: Maybe a → (a → Maybe b) → Maybe b
```

Meaning:

- If `Nothing` → stop
- If `Just x` → apply function

Example:

```
safeDiv 10 2 >>= (\x → safeDiv x 5)
```

If any step fails, result is `Nothing`. This gives **safe composition**.

---

14. Imperative-style syntax: `do`

Same example using `do`:

```
do
  x ← safeDiv 10 2
  safeDiv x 5
```

Looks like:

- Imperative code
- But still purely functional

---

15. Understanding Monads as containers

Think of monads as **containers**:

- `Maybe a` → value or no value
- `[]` → many values

- `IO a` → value with effects

Monad controls **how values are taken out and combined**.

---

16. Understanding Monads as computations

Better view:

> A monad represents a computation with context

Examples:

- `Maybe` → computation may fail
- `IO` → computation interacts with world
- `[]` → computation produces multiple results

Bind (`>>=`) defines:

- How computations are chained
- How context is preserved

17. Pros of Functional Programming

Functional Programming (FP) is based on:

- Pure functions
- Immutability
- No hidden side effects

Main advantages

1. **Easier reasoning**
    - Same input → same output
    - Programs are easier to understand and prove correct
2. **Safer code**
    - No unexpected side effects
    - Fewer bugs (especially concurrency bugs)
3. **Better modularity**
    - Programs are built by composing small functions
4. **Parallelism**

- No shared mutable state → easier parallel execution

5. **Strong mathematical foundation**
   - Based on lambda calculus

---

18. Problems of Functional Programming

Despite advantages, FP has challenges:

1. **I/O is difficult**
   - Real programs must read input, write output
   - Pure functions cannot change the world
2. **Performance concerns**
   - Laziness can cause memory issues if not understood
3. **Learning curve**
   - Concepts like monads are hard for beginners
4. **State and sequencing**
   - Order of execution is not explicit

This raises a question:

> How can we do I/O and state changes without breaking purity?

---

19. The Direct Approach

A **direct (imperative) approach** would be:

- Allow functions to perform I/O directly
- Allow mutation of variables

Example (imperative idea):

```
print "Hello"
read input
```

Problem:

- Breaks referential transparency
- Makes reasoning and optimization harder

Haskell **rejects this approach**.

---

20. But what if we are “lazy”?

Haskell is lazy:

- Evaluation order is not fixed
- Side effects cannot be executed “whenever”

So this does **not work**:

```
print "Hello"
print "World"
```

Why?

- Order is not guaranteed
- Lazy evaluation breaks sequencing

We need a **safe way to sequence actions**.

---

21. Fundamental question

Question:

> Is it possible to add imperative features without changing the meaning of pure Haskell expressions?

Answer: Yes — using monads.

Key idea:

- Separate **pure values** from **effectful computations**
- I/O is represented as **data**, not executed immediately

---

22. Monadic I/O: The Key Ideas

Haskell uses the **IO monad**. Important ideas:

- I/O actions are **values**
- They describe what to do
- The runtime system executes them in order

This preserves:

- Purity
- Laziness
- Referential transparency

---

23. A Helpful Picture

A value of type: `IO t`

Means:

> An action that, when performed:
>
> - may do input/output
> - returns a value of type `t`

Examples:

- `IO ()` → action with no meaningful result
- `IO String` → action that returns a string

Important:

> Creating an IO value does not perform I/O
>
> Only **running** it does

---

24. Implementation of the IO monad

Internally (simplified idea):

- An `IO` value is a function
- It takes the **world state**
- Returns a new world state and a value

Conceptually: `World → (World, Value)`

Programmers never see this directly. The runtime system handles it safely.

---

25. Simple I/O actions

Examples: `putStrLn :: String → IO ()`

- Action that prints a string: `getLine :: IO String`
- Action that reads input

Using `do` notation:

```
do
  putStrLn "Enter your name:"
```

```
  name ← getLine
  putStrLn name
```

This:

- Looks imperative
- Is still pure and functional

---

26. The Bind Combinator (>>=)

Definition: `(>>=) :: IO a → (a → IO b) → IO b`

Meaning

- Take an action
- Get its result
- Use it to build the next action

Example without `do`

```
getLine >>= \name →
putStrLn name
```

With `do`

```
do
  name ← getLine
  putStrLn name
```

`do` is just syntax sugar for `>>=`.

✅ Lecture 22 – Marshaling and Markup

1. Marshaling

Marshaling is the process of **transforming data into a format suitable for transmission or storage**, typically so it can be sent **across process boundaries, networks, or language runtimes**, and later reconstructed (unmarshaled).

Why it exists

Different systems may have:

- Different memory layouts
- Different data representations
- Different programming languages or runtimes

Marshaling provides a **common, transferable representation**.

Typical use cases

- Remote Procedure Calls (RPC)
- Inter-process communication (IPC)
- Client–server communication
- Calling native code from managed languages (and vice versa)

Examples

Conceptual example

- Object in memory → byte stream → sent over network → reconstructed object

Python (JSON marshaling)

```
import json

data = {"id":1,"name":"Alice"}
serialized = json.dumps(data)# marshal
restored = json.loads(serialized)# unmarshal
```

C# (.NET interop)

- Marshaling is used when calling native C/C++ code from C#
- The runtime converts managed objects into unmanaged representations

Key properties

- Data-focused (not presentation)
- Often binary or structured text (JSON, XML, Protobuf)
- Concerned with **correctness and compatibility**

---

2. Markup

Markup is the use of **tags or annotations embedded in text** to describe its **structure, semantics, or presentation**.

Plain text lacks structure. Markup adds meaning such as:

- What is a title
- What is a paragraph
- What is emphasized
- How content should be displayed or interpreted

Typical use cases

- Web documents
- Documentation
- UI structure
- Data description formats

Examples

HTML

```
<h1>Title</h1>
<p>This is a paragraph.</p>
```

XML

```
<user>
<id>1</id>
<name>Alice</name>
</user>
```

Key properties

- Text-based
- Human-readable
- Describes **structure or presentation**, not behavior

---

3. Marshaling vs Markup (Direct Comparison)

Aspect	Marshaling	Markup
Purpose	Data transfer and reconstruction	Structure and meaning of text
Focus	Data representation	Content annotation
Typical formats	Binary, JSON, Protobuf	HTML, XML, Markdown
Used in	IPC, RPC, networking	Web, documents, UI
Human readability	Optional	Essential

One-line intuition

- **Marshaling**: "How do I safely move this data between systems?"
- **Markup**: "How do I describe what this text *is* or *looks like*?"

✅ Lecture 24 – Lambda Expressions and Stream API in Java

1. Java 8: language extensions – main new features Java 8 introduced **functional-style programming**.

1. What a Lambda Expression Is

A **lambda expression** is an anonymous function: a block of code that can be passed as a value.

Purpose

- Treat behavior as data
- Enable functional-style programming in Java
- Reduce boilerplate compared to anonymous classes

Syntax

```
(parameters) → expression
(parameters) → { statements }
```

Example:

```
(x, y) → x + y
```

2. What You Must Understand Before Lambdas

2.1 Functional Interfaces (CRITICAL)

A functional interface has **exactly one abstract method**.

Examples:

```
Runnable// void run()
Callable<V>// V call()
Comparator<T>// int compare(T a, T b)
Predicate<T>// boolean test(T t)
```

Custom example:

```
@FunctionalInterface
interfaceAdder {
intadd(int a,int b);
}
```

Key point for exam

> A lambda does not define a type by itself.
>
> Its type is inferred from the **target functional interface**.

2.2 Type Inference

Java infers parameter types from context.

```
Comparator<Integer> c = (a, b) → a - b;
```

Equivalent:

```
Comparator<Integer> c = (Integer a, Integer b) → a - b;
```

Important

- Types can be omitted
- Return type is inferred

---

3. Lambda vs Anonymous Class

Anonymous class:

```
Runnabler=newRunnable() {
publicvoidrun() {
        System.out.println("Hello");
    }
};
```

Lambda:

```
Runnabler= () → System.out.println("Hello");
```

Key Differences (Exam Favorite)

Aspect	Lambda	Anonymous Class
`this`	Refers to enclosing object	Refers to inner class
State	Stateless by design	Can have fields
Boilerplate	Minimal	Verbose
Inheritance	No	Yes

4. Lambda Semantics (VERY IMPORTANT)

4.1 Effectively Final Variables

Lambdas can capture **local variables** only if they are **final or effectively final**.

```
intx=10;
Runnabler= () → System.out.println(x);// OK
x =20;// ERROR
```

Reason

- Lambdas capture values, not variables
- Ensures thread safety and predictable behavior

4.2 Scope Rules

- No new scope is created for variables
- Cannot redeclare a variable from the enclosing scope

```
intx=5;
(x) → x +1// INVALID
```

5. Common Standard Functional Interfaces

Know these **by heart** for oral exam:

Interface	Method	Meaning
`Predicate<T>`	`boolean test(T)`	Condition
`Function<T,R>`	`R apply(T)`	Transform
`Consumer<T>`	`void accept(T)`	Side effect
`Supplier<T>`	`T get()`	Produce value
`Comparator<T>`	`int compare(T,T)`	Ordering

Example:

```
Predicate<Integer> isEven = x → x %2 ==0;
```

6. Lambdas and Streams (High-Level)

Lambdas are heavily used in **Stream API**.

```
list.stream()
    .filter(x → x >10)
    .map(x → x *2)
    .forEach(System.out::println);
```

Key idea:

- Lambdas define **behavior**
- Streams define **data flow**

7. Method References (Lambda Shortcut)

```
x → System.out.println(x)
```

Becomes:

```
System.out::println
```

Types:

- Static method reference
- Instance method reference
- Constructor reference

---

8. Runtime and Compilation Model

Compilation

- Lambdas are **not compiled into anonymous classes**
- Implemented using `invokedynamic`

Runtime

- JVM creates function objects dynamically
- Enables better optimization

Exam phrase to use

> "Java lambdas are implemented via invokedynamic, not as syntactic sugar for anonymous classes."

---

9. Typical Oral Exam Questions (With Short Answers)

Q: Can a lambda exist without a functional interface?

No. It always targets a functional interface.

Q: Why must captured variables be effectively final?

To avoid inconsistent state and allow safe value capture.

Q: Difference between lambda and method reference?

Method reference is a more concise form when an existing method matches the lambda signature.

Q: Is lambda an object?

Yes, an instance of a functional interface.

Q: Can lambdas have state?

No internal mutable state; they rely on captured context.

---

10. One-Sentence Summary (Exam Ready)

> A Java lambda is an anonymous function that implements a functional interface, enabling behavior to be passed as data, with strict rules on scope, variable capture, and type inference, implemented efficiently via invokedynamic.

STREAM API

The **Stream API** is a high-level abstraction for **processing sequences of data** in a **functional, declarative style**.

Key idea

- Describe **what** to do, not **how** to loop
- Operations are applied to a **data source**, not stored

```
list.stream()
    .filter(x → x >10)
    .map(x → x *2)
    .forEach(System.out::println);
```

2. What a Stream Is (Important)

A **Stream** is:

- **Not a data structure**
- A **view over a data source** (collection, array, I/O, generator)

Properties

- No storage
- One-time use
- Lazily evaluated

3. Stream Pipeline Model (CORE EXAM TOPIC)

A stream program has **three stages**:

3.1 Source

Creates the stream.

```
list.stream()
Stream.of(1,2,3)
Arrays.stream(arr)
```

3.2 Intermediate Operations

- Transform the stream
- **Lazy** (not executed immediately)
- Return a new stream

Examples:

```
filter(Predicate)
map(Function)
flatMap(Function)
sorted()
distinct()
limit()
```

```
stream.filter(x → x >0)
      .map(x → x *2);
```

3.3 Terminal Operation

- Triggers execution
- Produces a result or side effect
- Consumes the stream

Examples:

```
forEach()
collect()
reduce()
count()
findFirst()
anyMatch()
```

```
stream.count();
```

Key phrase for exam

> Without a terminal operation, nothing is executed.

---

4. Lazy Evaluation (VERY IMPORTANT)

Intermediate operations are **deferred** until a terminal operation is called.

```
stream.filter(x → {
   System.out.println(x);
return x >5;
});
```

Nothing prints. Execution happens only after:

```
stream.forEach(System.out::println);
```

Why

- Performance optimization
- Short-circuiting

---

5. Stateless vs Stateful Operations

Stateless

Do not depend on previous elements:

```
map, filter
```

Stateful

Depend on stream history:

```
distinct, sorted, limit
```

Stateful operations may reduce parallel performance.

---

6. Internal Iteration vs External Iteration

Traditional Loop (External)

```
for (int x : list) {
if (x >10) process(x);
}
```

Stream (Internal)

```
list.stream()
    .filter(x → x >10)
    .forEach(this::process);
```

Key idea

- Stream controls iteration
- Enables optimizations and parallelism

---

7. Streams vs Collections (Exam Table)

Aspect	Collection	Stream
Stores data	Yes	No
Reusable	Yes	No
Evaluation	Eager	Lazy
Purpose	Data storage	Data processing

---

8. Reduction Operations

8.1 reduce()

Combines elements into one result.

```
intsum= list.stream()
            .reduce(0, Integer::sum);
```

Concept

- Associative operation
- Identity element

8.2 collect()

Used to build collections or complex results.

```
List<Integer> result =
    list.stream()
        .filter(x → x >0)
        .collect(Collectors.toList());
```

Common collectors:

- `toList()`
- `toSet()`
- `groupingBy()`
- `partitioningBy()`

---

9. Parallel Streams

```
list.parallelStream()
    .map(x → x *2)
    .forEach(System.out::println);
```

Characteristics

- Uses Fork/Join framework
- Automatic parallelism
- Order not guaranteed unless enforced

Caution

- Not always faster
- Avoid with:
    - I/O
    - Side effects
    - Stateful lambdas

---

10. Side Effects and Purity

Recommended

- Lambdas should be **pure functions**

Avoid

```
stream.forEach(x → sharedList.add(x));// unsafe
```

Reason

- Breaks parallel safety
- Unpredictable behavior

---

11. Stream Ordering

- Streams may be **ordered** or **unordered**
- Operations like `findFirst()` respect order
- `findAny()` may return any element (faster in parallel)

---

12. Typical Oral Exam Questions (With Short Answers)

Q: Is a Stream a collection?

No. It processes data but does not store it.

Q: Why are streams lazy?

To optimize execution and avoid unnecessary computation.

Q: What triggers stream execution?

A terminal operation.

Q: Can a stream be reused?

No. After a terminal operation, it is consumed.

Q: Difference between `map` and `flatMap`?

`map` transforms elements; `flatMap` flattens nested streams.

✅ Lecture 25 – Reflection and Annotations in Java

1. Reflection & kinds of reflection

Reflection is the ability of a program to:

- Inspect itself
- Modify its behavior at runtime

Kinds of reflection:

- **Structural reflection** → structure of classes
- **Behavioral reflection** → behavior (methods, execution)

Java supports **runtime reflection**.

---

2. Structural and behavioral reflection

Structural reflection

Inspect:

- Classes
- Fields
- Methods
- Constructors
- Interfaces

Example:

> "What fields does this class have?"

Behavioral reflection

Invoke or change behavior:

- Call methods dynamically
- Modify field values

Example:

> "Call a method whose name is only known at runtime"

---

3. LISP-style reflection & metaprogramming (metacircularity)

In **LISP-like languages**:

- Programs and data have the same structure
- Code can manipulate itself easily

Metaprogramming:

- Writing programs that write or modify programs

Metacircularity:

- Language implemented in itself

Java is **not metacircular**, but supports limited reflection.

---

4. Uses of Reflection

Reflection is used for:

- Frameworks (Spring, Hibernate)
- Dependency Injection
- Serialization / Deserialization
- Unit testing
- IDE tools
- Debuggers

Key idea:

> Reflection enables generic and flexible frameworks

---

5. Drawbacks of Reflection

Main problems:

- Slower performance
- Breaks encapsulation
- Harder to read and maintain
- Errors appear at runtime, not compile time
- Security risks

Reflection should be used **only when needed**.

---

6. Reflection in Java

Java reflection is in package: `java.lang.reflect`

Main classes:

- `Class`
- `Field`
- `Method`

- `Constructor`

---

7. Reflection logical hierarchy in Java

Hierarchy:

```
Class
 ├── Field
 ├──Method
 └── Constructor
```

Everything starts with a **Class object**.

---

8. Retrieving Class objects

Three common ways:

```
MyClass.class
obj.getClass()
Class.forName("MyClass")
```

Used to get metadata about a class.

---

9. Class file structure

A `.class` file contains:

- Class name
- Fields
- Methods
- Constructors
- Bytecode
- Constant pool
- Annotations

Reflection reads this metadata at runtime.

---

10. Inspecting a Class

Using `Class` methods:

```
Class<?> c = MyClass.class;
```

You can inspect:

- Name
- Package
- Superclass
- Interfaces
- Modifiers

---

11. Discovering class members

Members include:

- Fields
- Methods
- Constructors

Reflection allows:

- Listing members
- Reading types and modifiers

---

12. Class methods for locating members

Main methods:

Method	Meaning
`getFields()`	public fields (incl. inherited)
`getDeclaredFields()`	all fields (only this class)
`getMethods()`	public methods
`getDeclaredMethods()`	all methods

---

13. Example: locating methods

```
Method[] methods = c.getMethods();
```

Finds:

- All public methods
- Including inherited ones

---

14. Working with class members

After finding members, you can:

- Read their name
- Read type
- Invoke methods
- Read or write fields

---

20. Generic methods: effects of erasure

Java generics use **type erasure**:

- Type parameters removed at runtime
- Reflection cannot see generic types fully

Example: `List<String> → List`

Loss of type info at runtime.

---

21. Using reflection for program manipulation

Reflection allows:

- Creating objects dynamically
- Setting fields
- Calling methods by name

Used in:

- Testing frameworks
- Dependency injection

---

22. Creating new objects

```
Objecto= c.getConstructor().newInstance();
```

Creates an object **without knowing class at compile time**.

---

23. Accessing fields

```
Fieldf= c.getDeclaredField("x");
Objectvalue= f.get(obj);
```

Used to read data dynamically.

---

24. Invoking methods

```
Methodm= c.getMethod("foo");
m.invoke(obj);
```

Method name can be computed at runtime.

---

25. Accessible objects

`AccessibleObject`:

- Superclass of `Field`, `Method`, `Constructor`

Method:

```
setAccessible(true)
```

Disables access checks.

---

26. Accessing private fields

```
field.setAccessible(true);
```

Allows:

- Reading private fields
- Writing private fields

⚠ Breaks encapsulation.

---

27. Exploiting reflection: Unit testing

Used by testing frameworks:

- JUnit
- TestNG

Reflection helps:

- Discover test methods
- Invoke them automatically
- Access private methods if needed

Annotations:

1. From Modifiers to Annotations

Modifiers (background). In early Java, **modifiers** were the main way to attach extra meaning to code elements.

Examples:

- `public`, `private`, `protected`
- `static`, `final`, `abstract`
- `synchronized`, `volatile`

They:

- Are part of the Java language syntax
- Have fixed meaning
- Are interpreted directly by the compiler or JVM

Why modifiers were not enough. Modifiers are:

- Limited in number
- Not extensible by developers
- Cannot carry custom metadata

Example limitation: You cannot express things like:

- “This method is deprecated since version 2.1”
- “This class is a REST controller”
- “This field must not be null”

Annotations

Annotations were introduced in **Java 5** to solve this.

Definition: An annotation is a **form of metadata** that provides information about code **without changing its logic**. Annotations:

- Start with `@`
- Can be processed by:
  - The compiler
  - Tools (e.g., build tools, frameworks)
  - The JVM at runtime (via reflection)

Example:

```
@Override
public StringtoString() {
return"Example";
}
```

`@Override` tells the compiler:

- This method must override a superclass method

---

2. Structure of Annotations

Basic syntax: `@AnnotationName`

Example:

```
@Deprecated
publicvoidoldMethod() {}
```

Annotation with elements (parameters)

Annotations can have **elements**, similar to methods.

`@AnnotationName(element1 = value1, element2 = value2)`

Example: `@SuppressWarnings(value = "unchecked")`

Annotation type definition:

Annotations are defined using `@interface`.

```
public@interface MyAnnotation {
    Stringvalue();
intversion()default1;
}
```

Key points:

- `@interface` defines an annotation type
- Elements look like methods
- No method body
- `default` provides optional values

Allowed element types

Annotation elements can only be:

- Primitive types (`int`, `boolean`, etc.)
- `String`
- `Class`
- `enum`
- Other annotations
- Arrays of the above

Invalid:

- Objects
- Collections
- Arbitrary classes

Example:

```
public@interface Example {
    Stringname();
int[] values();
    Class<?> targetClass();
}
```

---

3. Which Elements Can Be Annotated?

Annotations can be applied to many Java elements.

Common targets

- Classes, interfaces, Enums, Methods, Fields, Constructors, Parameters, Local variables, Packages, Other annotations.

Example:

```
@Deprecated
publicclassOldClass {

@Deprecated
publicvoidoldMethod(@Deprecatedint x) {}
}
```

Controlling where annotations can be used

This is done using `@Target`. Example:

```
import java.lang.annotation.*;

@Target(ElementType.METHOD)
public@interface MethodOnlyAnnotation {}
```

Common `ElementType` values:

- `TYPE` → class, interface, enum
- `METHOD, FIELD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, ANNOTATION_TYPE, PACKAGE`

---

4. Some Predefined Annotations

`@Override`

- Applied to methods
- Ensures the method overrides a superclass method
- Compiler checks correctness

Example:

```
@Override
public StringtoString() {
return"Hello";
}
```

Why important:

- Prevents errors caused by wrong method signatures

`@Deprecated`

- Marks code as obsolete
- Compiler generates a warning when used

Example:

```
@Deprecated
publicvoidoldApi() {}
```

`@SuppressWarnings` : Tells compiler to ignore specific warnings. Example:

5. Define and Use Your Own Annotations

Why define custom annotations: Custom annotations allow developers to:

- Attach **domain-specific metadata**
- Guide frameworks, tools, or runtime behavior
- Avoid hard-coded rules in code

Annotations themselves **do nothing automatically**. They become useful only when:

- Processed by the compiler
- Read by tools
- Read at runtime via reflection

Defining a custom annotation. Basic syntax: `public@interface MyAnnotation { }`

Annotation with elements:

```
public@interface MyAnnotation {
    Stringauthor();
intversion();
}
```

Using default values:

```
public@interface MyAnnotation {
    Stringauthor()default"unknown";
```

```
intversion()default1;
}
```

Key rules:

- No method bodies
- Elements act like configuration fields
- `default` makes elements optional

✅ Lecture 26 – Functions, Decorators and OOP in Python

A **function** is a reusable piece of code that performs a specific task.

- To avoid repeating code
- To organize code
- To make programs easier to understand

Functions in Python:

- Functions are first-class objects
- All functions return some value
- Function call creates a new scope
- Parameters passed by object reference
- Functions can have **optional keyword arguments**
- Functions can take a **variable number** pf args and kwargs
- **No overloading**
- Higher order functions are supported

Basic idea

```
def greet():
    print("Hello")
```

- `def` = define a function

- `greet` = function name
- When you call `greet()`, it runs the code inside

Functions with parameters: You can pass values into a function:

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

- `a` and `b` are inputs
- `return` gives a result back

Higher-order function is simply:

A function that **takes another function as an argument** OR **returns a function**.

Examples in Python:

- `map()`
- `filter()`
- `sorted()` with key functions

Simple Example:

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

Here:

- `apply_twice` is a higher-order function
- it receives the function `f` as input

Python allows functions to be treated like data, same as numbers or strings.

(Map) - Higher order Functions

- Functions can be passed as argument and returned as result
- Main combinators **(map, filter)** predefined: allow standard functional programming style in Python
- Heavy use of iterators, which support laziness

- Lambdas supported for use with combinators
    - `lambda arguments: expression` - The body only can be a single expression

`map()` applies a function to every element of a list.

Example:

```
numbers = [1, 2, 3]
result = map(lambda x: x * 2, numbers)
```

This means:

- Take each number
- Multiply it by 2
- Create a new list-like object

You can convert it to a list:

```
list(result)  # [2, 4, 6]
```

Simple Explanation:

map() = apply a function to each item in a list.

List Comprehension can replace uses of **Map**

This is a **shorter, cleaner way** to create lists. Instead of:

```
result = []
for x in numbers:
    result.append(x * 2)
```

You write:

```
result = [x * 2 for x in numbers]
```

Simple Explanation: List comprehension = a compact way to build lists using a loop in one line. Example with a condition:

```
evens = [x for x in numbers if x % 2 == 0]
```

This keeps only even numbers.

Decorators: Decorators are **functions that modify other functions**.

Simple explanation:

Imagine you have a function, and you want to **add extra behavior** without changing its code.

A decorator "wraps" the original function and adds something new.

Example (conceptual):

- You have a function `login()`
- You want to **automatically check authentication** before the function runs
- A decorator can do that

Small example:

```
def decorator(func):
    def wrapper():
        print("Before function")
        func()
        print("After function")
    return wrapper

@decorator
def hello():
    print("Hello")
```

When you call `hello()`, it actually runs `wrapper()` around it.

Simple explanation:

A decorator adds extra code before or after a function runs — without changing the original function.

- A **decorator** is any callable Python object that is used to modify a **function**, method or class definition.
- A decorator is passed the original object being defined and returns a modified object, which is then bound to the name in the definition.
- (Function) Decorators exploit higher order features:

- Passing functions as argument
- nested definition of functions
- Returning

Usage:

```
@my_decorator
defsay_hello():
print("Hello")
```

Calling `say_hello()`:

- Executes `wrapper`
- Adds behavior before and after the function

---

4. Namespaces and Scopes

A **namespace** is a mapping between:

- Names
- Objects

Example: `x =10`

`x` exists in a namespace and refers to `10`.

Types of namespaces: Local, Global, Built-in

---

5. Scoping Rules: Python uses **LEGB** rule:

1. **L**ocal
2. **E**nclosing
3. **G**lobal
4. **B**uilt-in

Example:

```
x =5
deffunc():
    x =10
print(x)
```

Output: `10`

Local `x` overrides global `x`.

`global` **keyword:** Used to modify global variables.

```
x =5
deffunc():
global x
    x =10
```

`nonlocal` **keyword:** Used for enclosing scopes.

```
defouter():
    x =5
definner():
nonlocal x
        x =10
```

✅ Lecture 27 – Concurrent and Asynchronous Programming

1. Concurrent Programming

Concurrent programming is about **structuring a program so that multiple tasks make progress during the same time period**.

The tasks may:

- Run in parallel (on multiple CPU cores), or
- Interleave execution on a single core

Concurrency is a **program design concept**, not a guarantee of parallel execution.

Purpose

- Improve responsiveness
- Handle multiple tasks or users
- Structure complex workflows

Example (conceptual)

- A web server handling multiple client requests
- An application processing input while updating UI

Simple example (Python threads)

```
import threading

deftask():
print("Task running")

threading.Thread(target=task).start()
```

Key characteristics

- Multiple tasks in progress
- Requires synchronization (locks, semaphores)
- Risk of race conditions and deadlocks

---

2. Asynchronous Programming

Asynchronous programming is about **not blocking while waiting for long-running operations** (I/O, network, disk).

Instead of waiting:

- The task is started
- The program continues
- A callback or continuation runs when the result is ready

Asynchrony is about **waiting efficiently**, not about running multiple tasks at once.

Purpose

- Avoid blocking threads
- Improve scalability for I/O-bound work
- Keep applications responsive

Example (async/await)

```
import asyncio

asyncdeffetch_data():
await asyncio.sleep(1)
return"done"
```

Key characteristics

- Non-blocking execution
- Usually single-threaded
- Common in I/O-heavy systems

4. Relationship between the two

- Asynchronous programming **can be concurrent**
- Concurrent programming **does not have to be asynchronous**
- They often work together in modern systems

Example: A web server:

- **Concurrent**: handles many requests
- **Asynchronous**: does not block threads while waiting for database or network responses

---

One-line summaries

- **Concurrency**: *Doing multiple things in overlapping time*
- **Asynchrony**: *Not blocking while waiting*

✅ Lecture 28 – CPS & Asynch

Continuation-passing style (CPS) is a programming style where **functions do not return values directly**.

Instead, they **receive another function (a continuation)** that represents *"what to do next"* with the result.

The function **calls the continuation** with its result rather than returning it.

Simple example

Direct style

```
defadd(a, b):
return a + b

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

CPS style

```
defadd_cps(a, b, cont):
    cont(a + b)

add_cps(2,3,print)
```

Here:

- `cont` is the continuation
- `print` defines what happens after `add_cps` finishes

Why CPS is used

1. Explicit control flow

- The program never relies on the call stack to decide what happens next
- All future steps are explicit functions

2. Asynchronous programming

- Callbacks are CPS in practice
- Common in event-driven systems

3. Compiler and runtime design

- Used in functional language compilers
- Makes control structures (loops, exceptions) explicit

4. Non-standard control flow

- Early exits
- Backtracking
- Coroutines

CPS and asynchronous code

This callback-based async code is CPS:

```
readFile("data.txt",function(result) {
process(result);
});
```

Modern `async/await` hides CPS, but internally:

- The compiler rewrites async code into continuation-based state machines

---

Key characteristics

- No direct returns
- Control flow passed as functions
- Call stack replaced by explicit continuations
- Often harder to read manually

✅ Lecture 29 – Lifecycle managements, software distribution and debugging

DevOps + Docker