COM1013
INTRODUCTION TO COMPUTER SCIENCE

Lecturer: Begüm MUTLU BİLGE, PhD

begummutlubilge+com1013@gmail.com (recommended)
bmbilge@ankara.edu.tr

CHAPTER 6

Programming Languages

In this chapter we study programming languages. Our purpose is not to focus on a particular language, although we will continue to use examples from Python where appropriate. Rather it is to learn *about* programming languages. We want to appreciate the commonality as well as the diversity among programming languages and their associated methodologies.

Historical Perspective: Early Generations

Writing programs in a machine language is a tedious task

- that often leads to errors

In the 1940s, researchers simplified the programming process by developing notational systems by which instructions could be represented in **mnemonic** rather than numeric form.

For example, the instruction

Move the contents of register 5 to register 6

would be expressed as

4056 (using the machine language),

in a mnemonic system it might appear as

MOV R5, R6

Historical Perspective: Early Generations

Once such a mnemonic system was established, programs called **assemblers** were developed

- to convert mnemonic expressions into machine language instructions.

A mnemonic system for representing programs is collectively called an **assembly language**.

Historical Perspective: Early Generations

a program written in an assembly language is inherently machine dependent

- a program written in assembly language cannot be easily transported to another computer design because it must be rewritten to conform to the new computer’s register configuration and instruction set.

The programmer is still forced to think in terms of the small, incremental steps of the machine’s language.

Historical Perspective: Early Generations

The design process is better suited to the use of high-level primitives,

- each representing a concept associated with a major feature of the product.

Once the design is complete, these primitives can be translated to lower-level concepts relating to the details of implementation.

Historical Perspective: Early Generations

Following this philosophy, computer scientists began developing **programming languages**

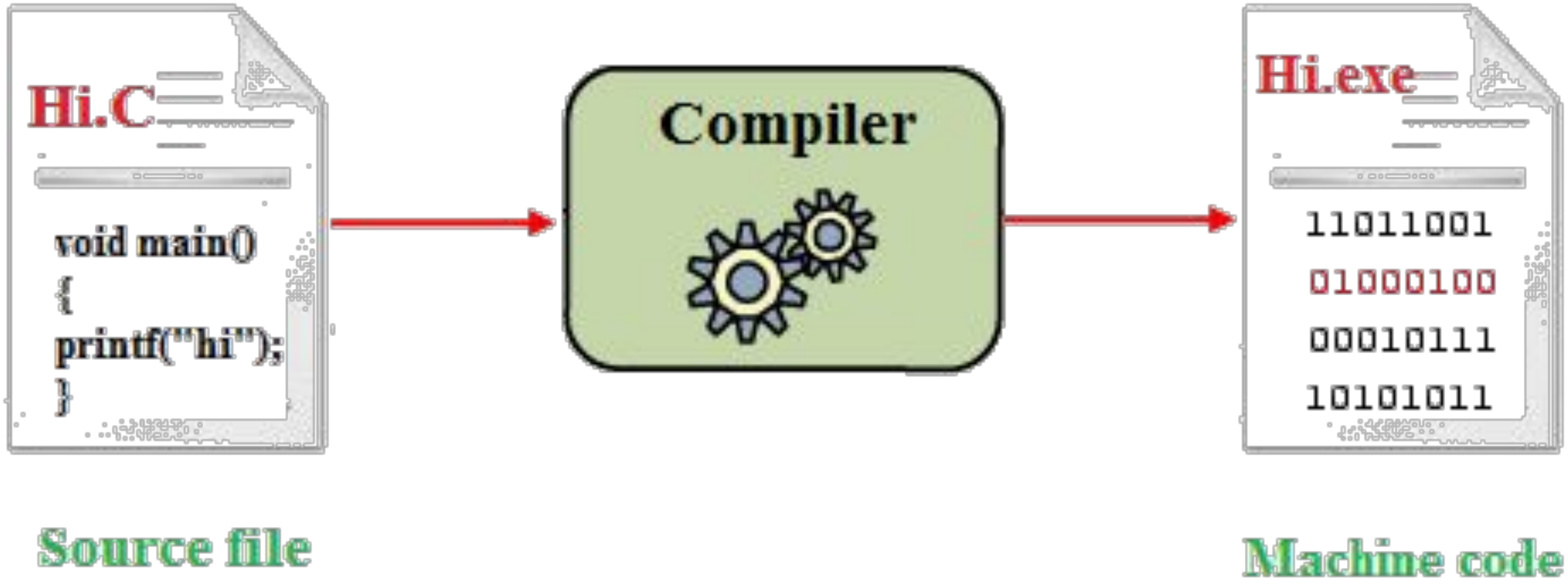
- that were more conducive to software development than were the low-level assembly languages.

High level & **machine independent**

The best-known early examples are

- FORTRAN (FORmula TRANslator), which was developed for scientific and engineering applications, and
- COBOL (COmmon Business-Oriented Language), which was developed by the U.S. Navy for business applications.

Historical Perspective: Early Generations

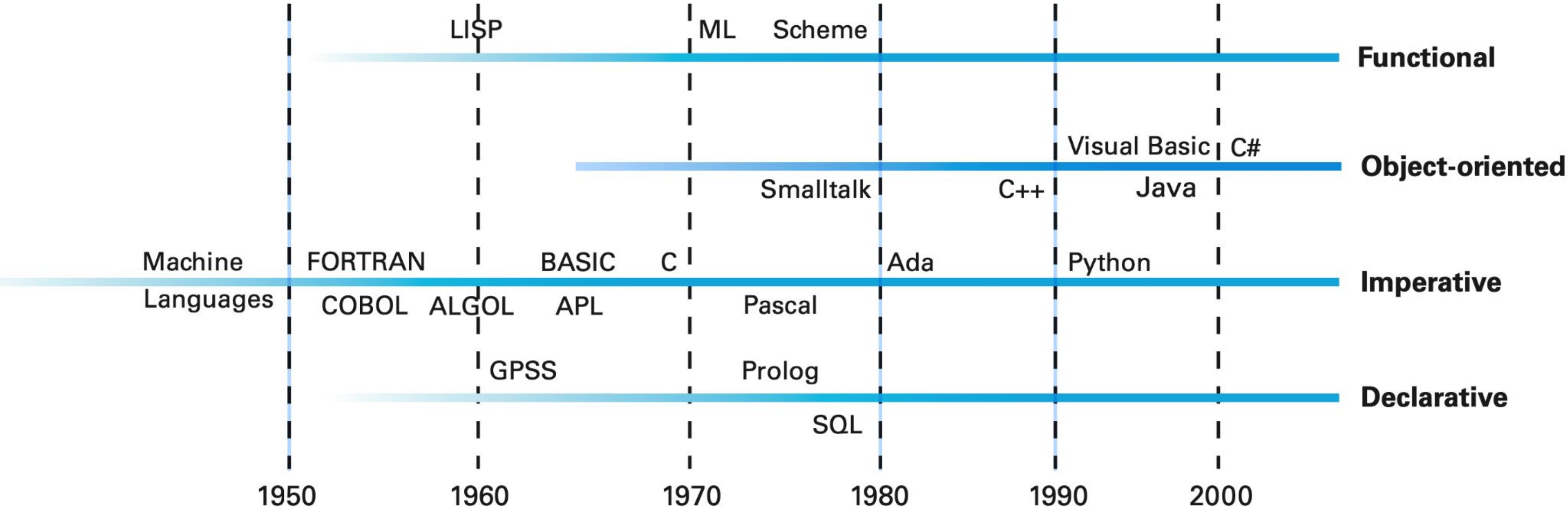


Programming Paradigms

The development of programming languages has developed along different paths

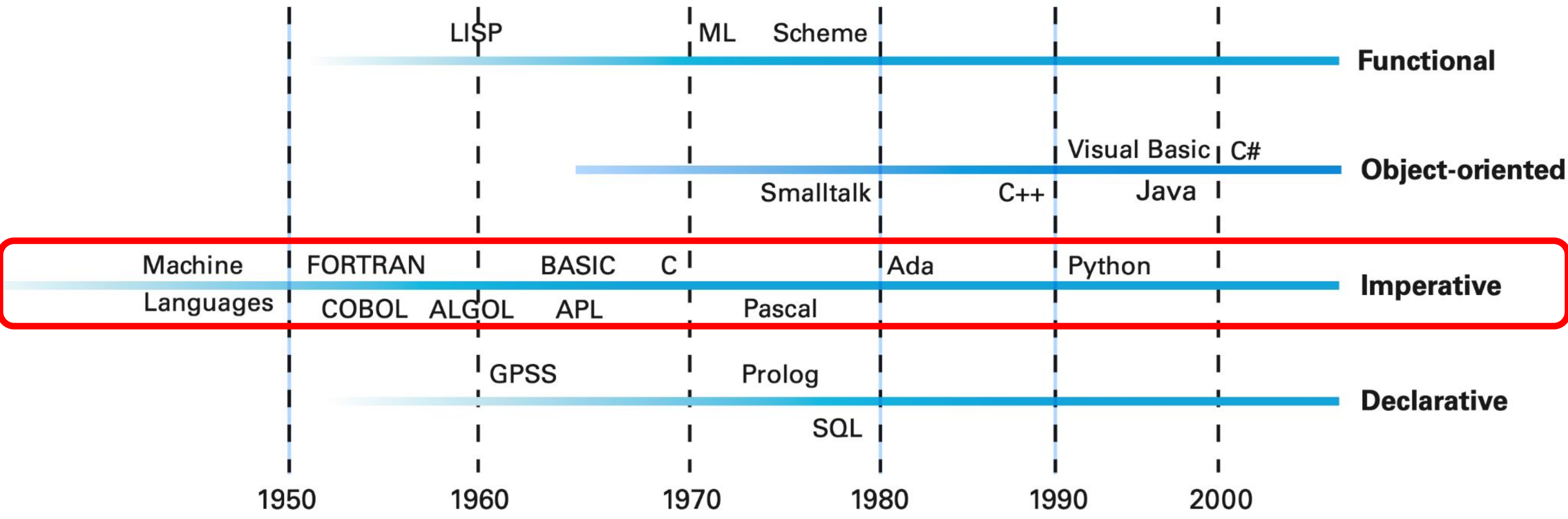
- as alternative approaches to the programming process
- (called programming paradigms)
    - software development paradigm

Programming Paradigms



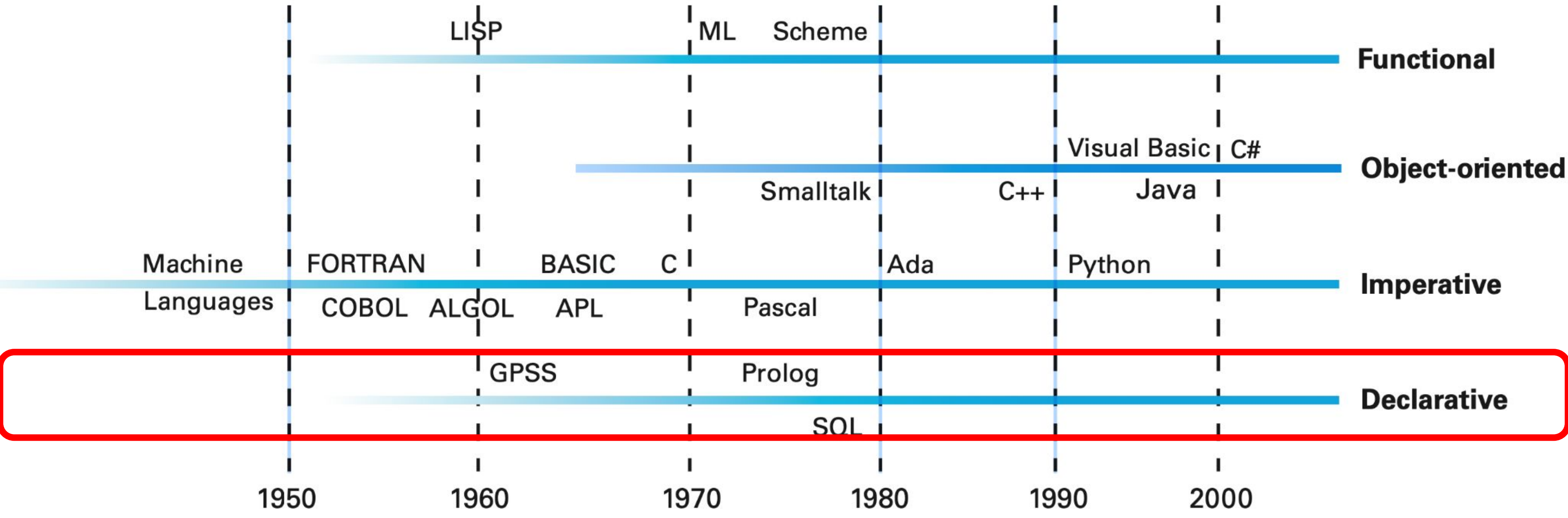
A better representation for the development of programming languages.

Programming Paradigms : Imperative (procedural) paradigm



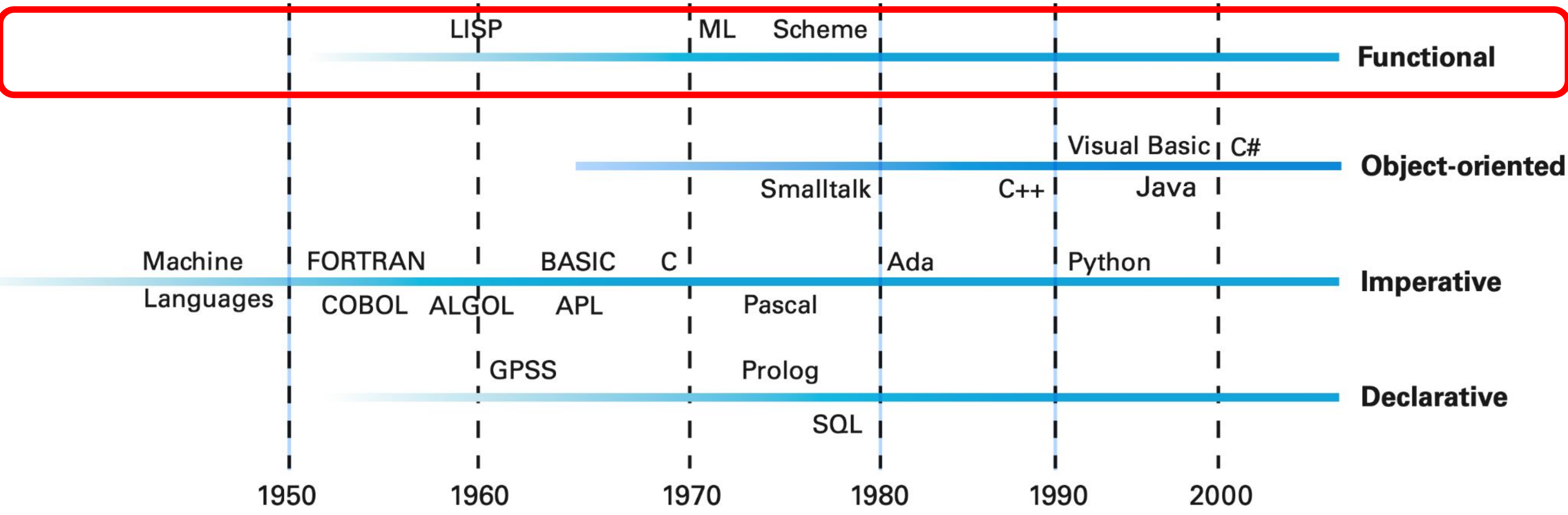
- Traditional approach to the programming process.
- The development of a sequence of commands that manipulate data to produce the desired result.
- Tells us to approach the programming process by
    - finding an algorithm to solve the problem at hand and then
    - expressing that algorithm as a sequence of commands

Programming Paradigms : Declarative paradigm



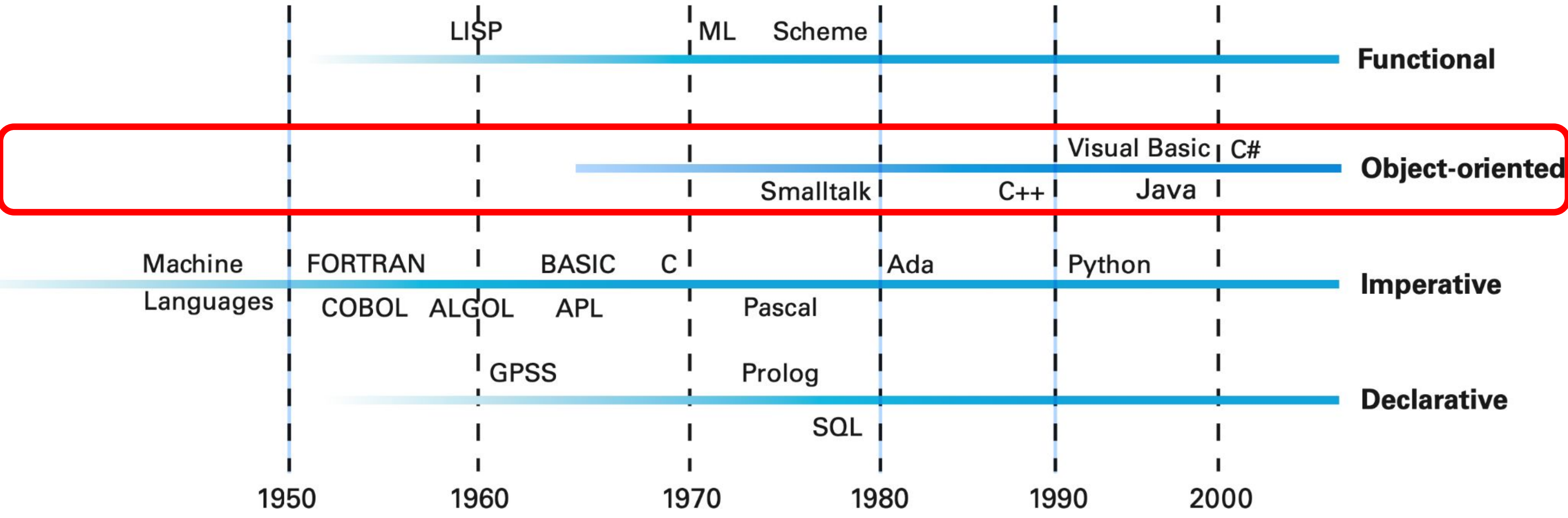
- Applies a pre-established general-purpose problem-solving algorithm to solve problems

Programming Paradigms : Functional paradigm



- A program is viewed as an entity that accepts inputs and produces outputs.

Programming Paradigms : Object oriented paradigm

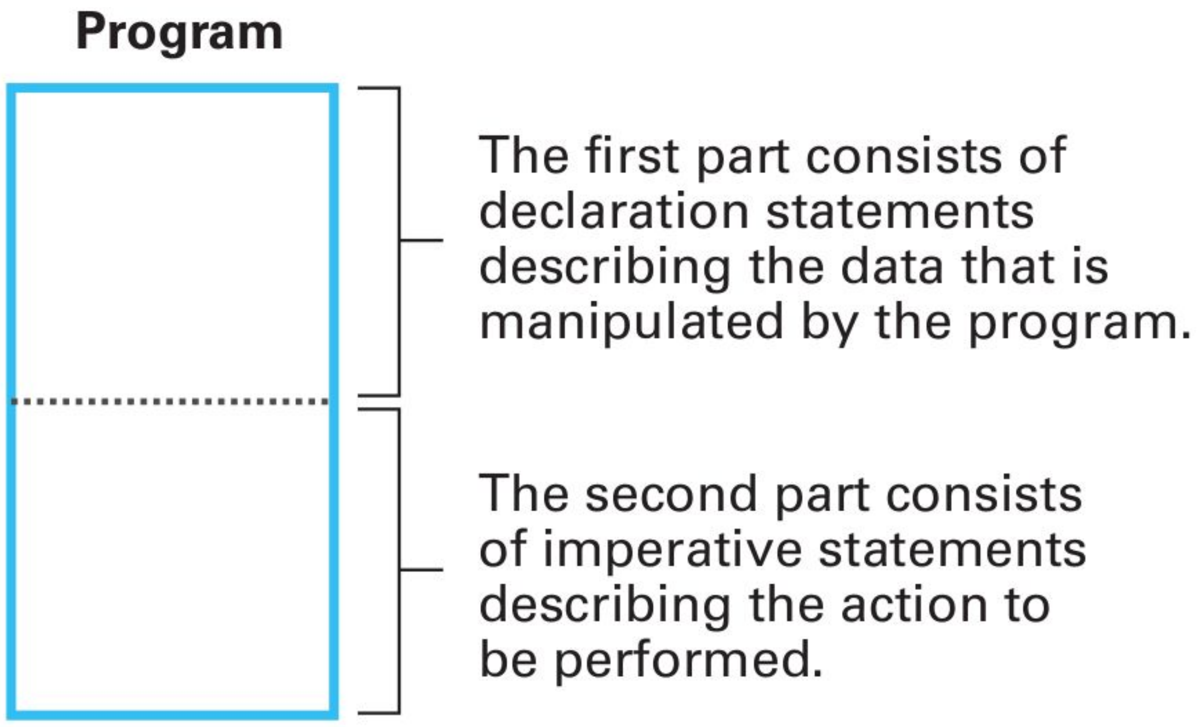


- A software system is viewed as a collection of units, called objects,
    - each one is capable of performing the actions that are immediately related to itself as well as requesting actions of other objects.
- Together, these objects interact to solve the problem at hand.

Traditional Programming Concepts

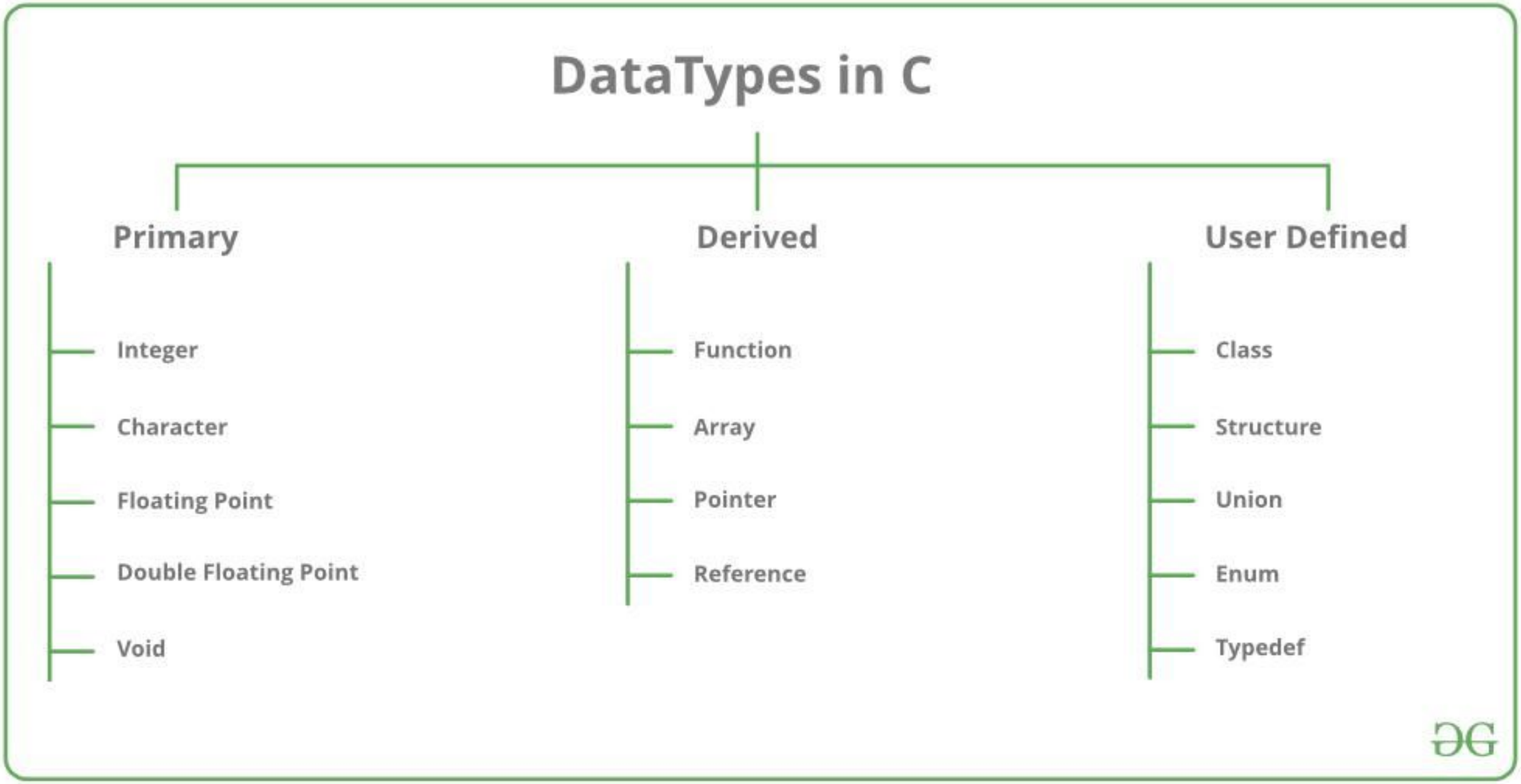
A program consists of a collection of statements that tend to fall into three categories:

- declarative statements,
- imperative statements, and
- comments.


Program
The first part consists of declaration statements describing the data that is manipulated by the program.
The second part consists of imperative statements describing the action to be performed.

Traditional Programming Concepts

Data types



Traditional Programming Concepts

Data Structure

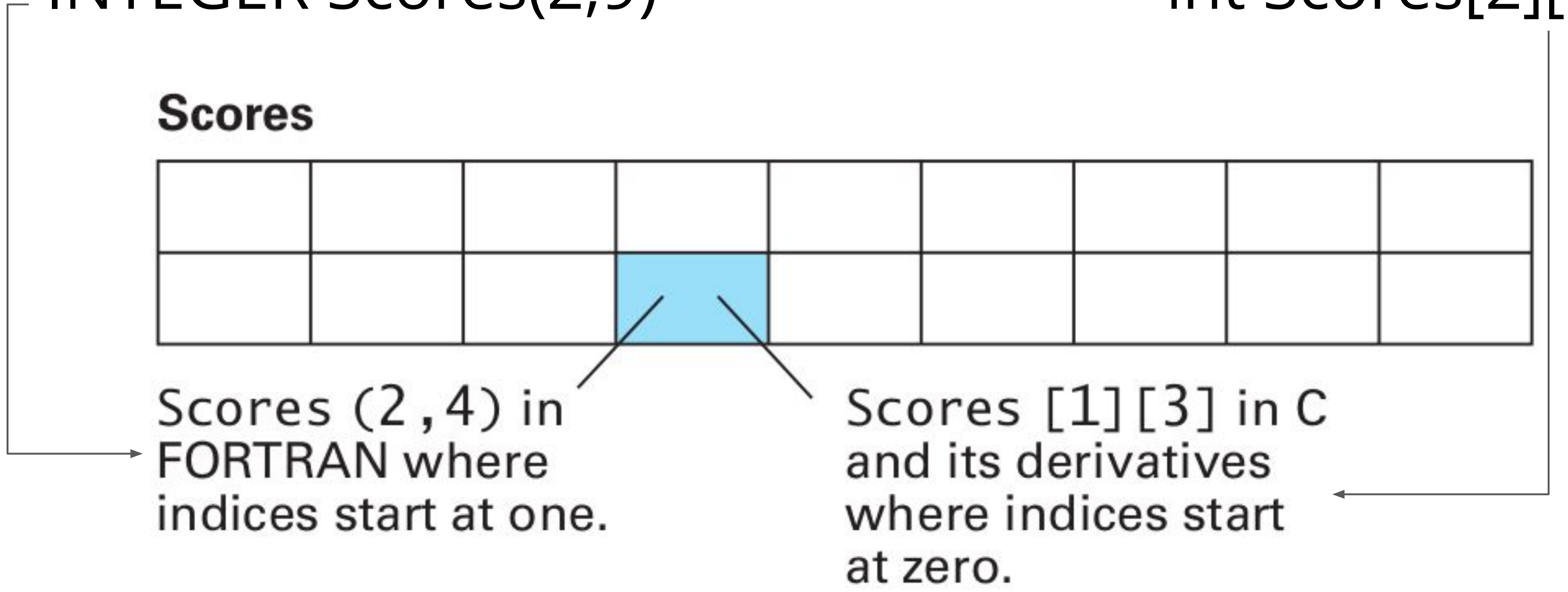
In addition to data type, variables in a program are often associated with data structure, which is **the conceptual shape or arrangement of data**.

Traditional Programming Concepts

Data Structure

One common data structure is the array, which is a block of elements of the same type such as a one-dimensional list, a two-dimensional table with rows and columns, or tables with higher dimensions.

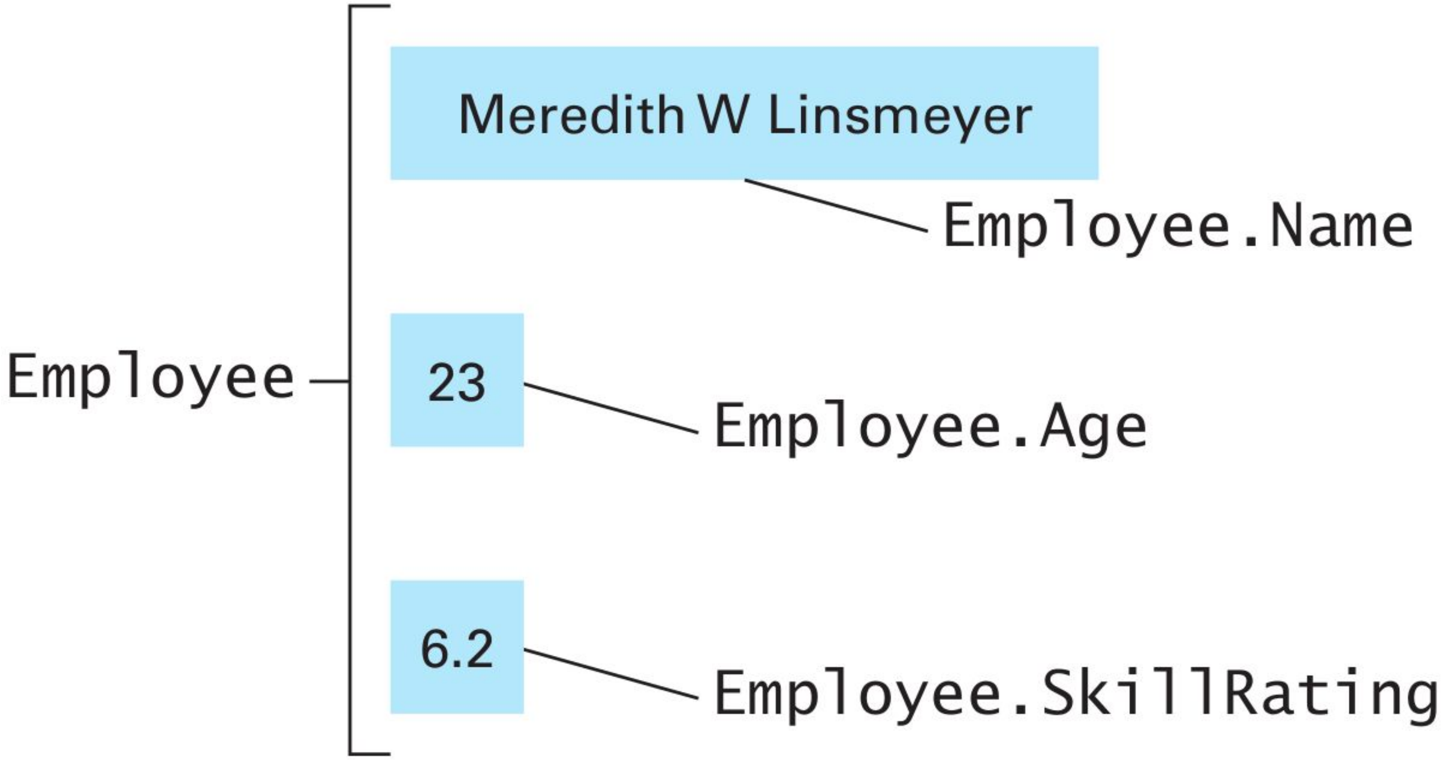
INTEGER Scores(2,9) int Scores[2][9];


Scores
Scores (2,4) in FORTRAN where indices start at one.
Scores [1][3] in C and its derivatives where indices start at zero.

Traditional Programming Concepts

Data Structure

```
struct { char  Name[25];
         int Age;
         float SkillRating;
       } Employee;
```



Traditional Programming Concepts

Assignment Statements

The most basic imperative statement is the assignment statement

- requests that a value be assigned to a variable

Z = X + Y;

Z := X + Y;

Z ← X + Y

Traditional Programming Concepts

Control Statements

A control statement alters the execution sequence of the program.

```
if (condition):
    statementA
else:
    statementB
```

and

```
while (condition):
    body
```

```
IF condition THEN
    statementA;
ELSE
    statementB;
END IF;
```

and

```
WHILE condition LOOP
    body
END LOOP;
```

```
if (condition) statementA; else statementB;
```

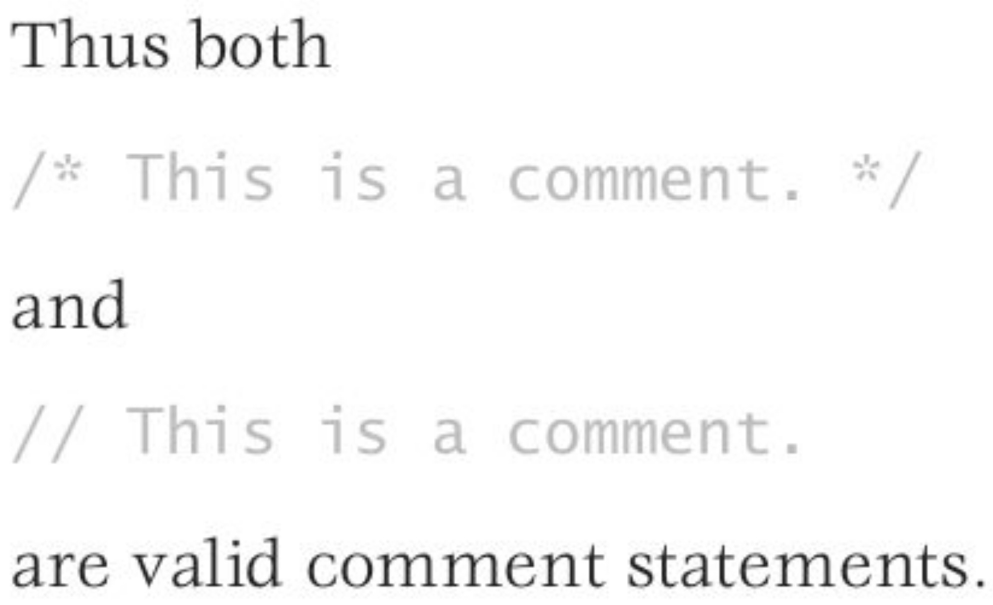
and

```
while (condition) { body }
```

Traditional Programming Concepts

Comments

- explanatory statements
- These statements are ignored by a translator,
    - therefore their presence or absence does not affect the program from a machine’s point of view.



Thus both

```
/* This is a comment. */
```

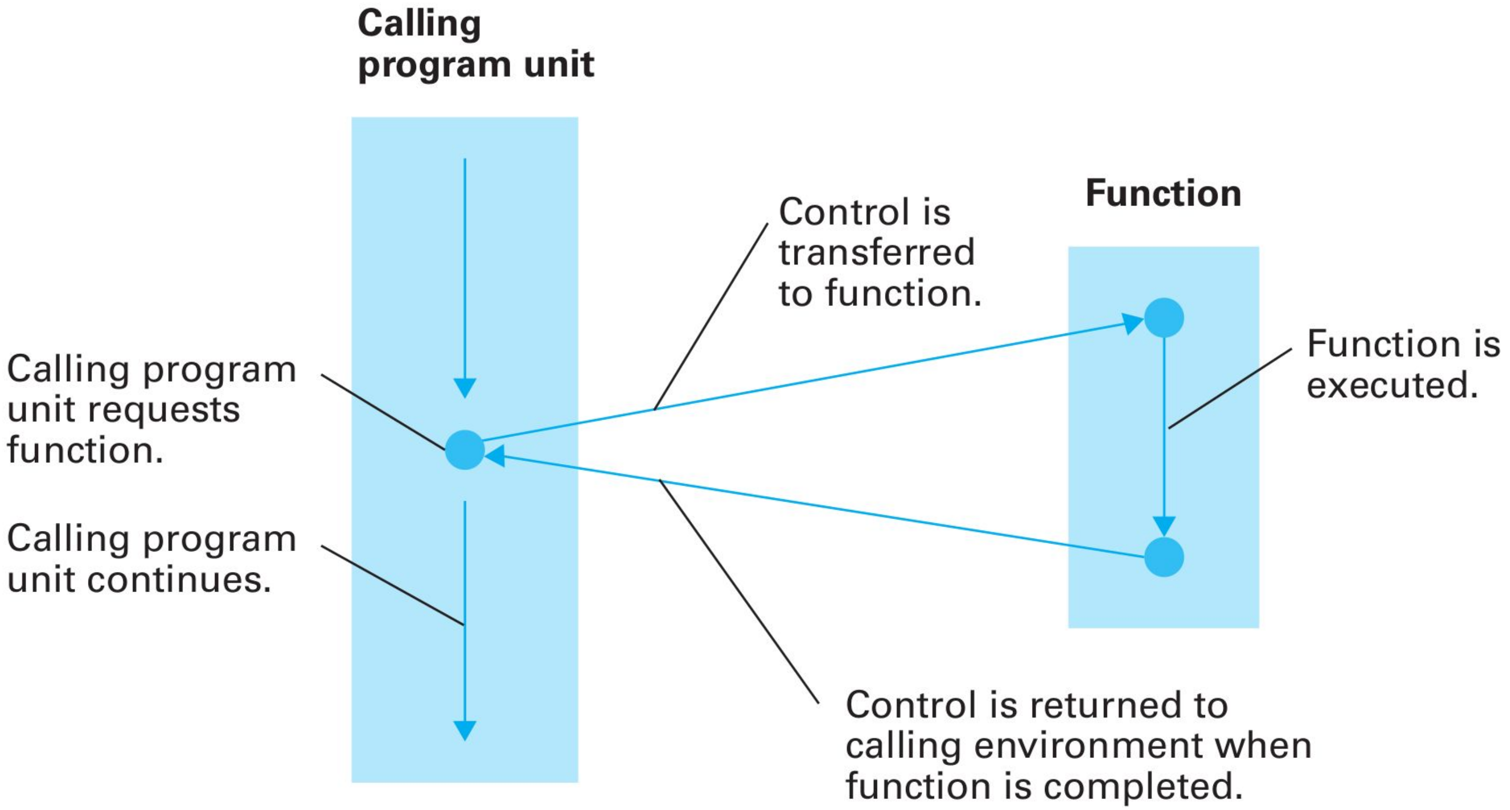
and

```
// This is a comment.
```

are valid comment statements.

Procedural Units

A **function**, in its generic sense, is a set of instructions for performing a task that can be used as an abstract tool by other program units.

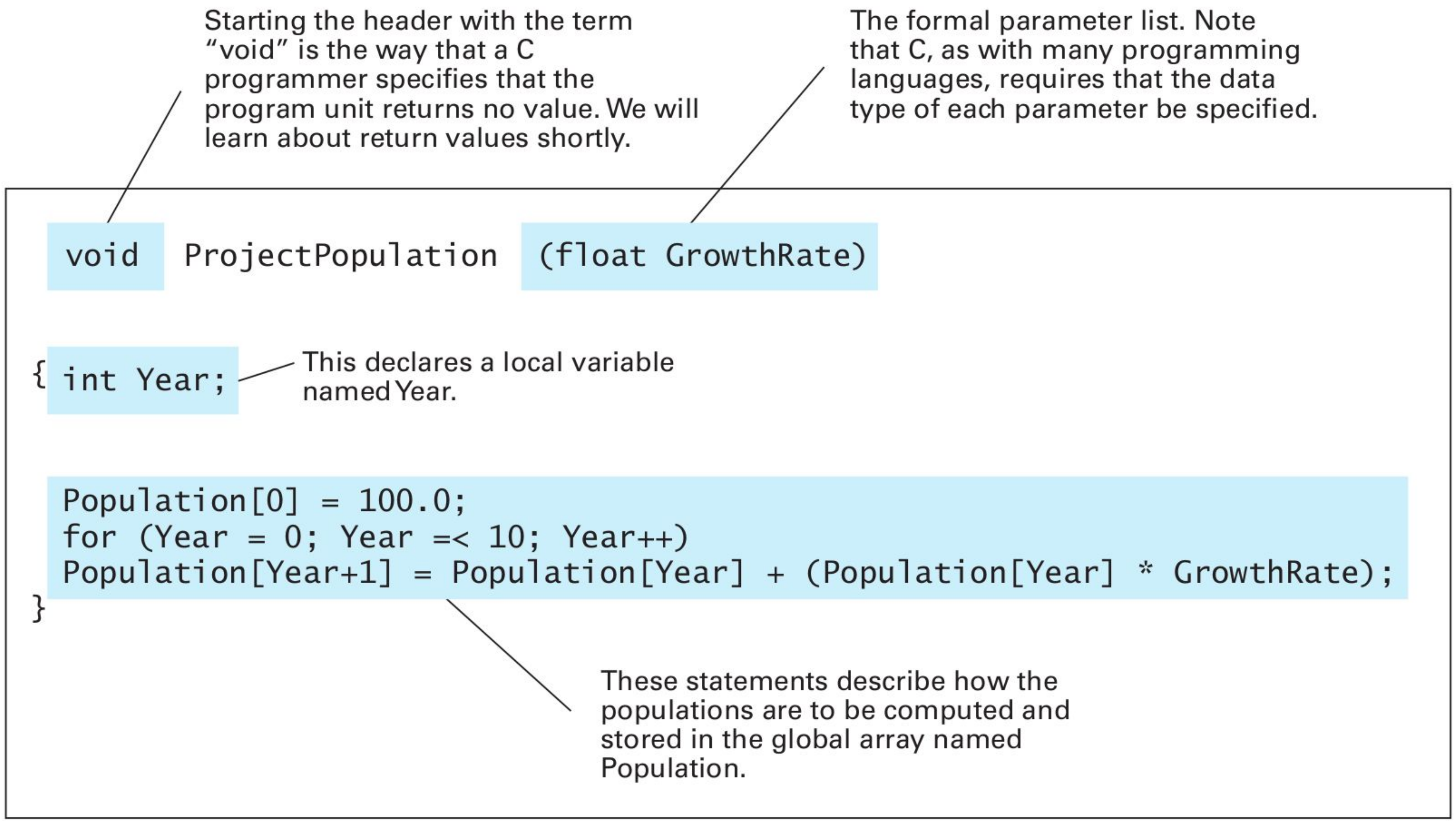

Calling program unit
Function
Control is transferred to function.
Function is executed.
Calling program unit requests function.
Calling program unit continues.
Control is returned to calling environment when function is completed.

Procedural Units

Functions are often written using generic terms that are made specific when the function is applied.

- Such generic terms within functions are called **parameters (formal parameters)**
- The precise meanings assigned to these formal parameters when the function is applied are called **actual parameters.**
    - The task of transferring data between actual and formal parameters is handled in a variety of ways
        - passed by VALUE
        - passed by REFERENCE

Procedural Units



Starting the header with the term "void" is the way that a C programmer specifies that the program unit returns no value. We will learn about return values shortly.

The formal parameter list. Note that C, as with many programming languages, requires that the data type of each parameter be specified.

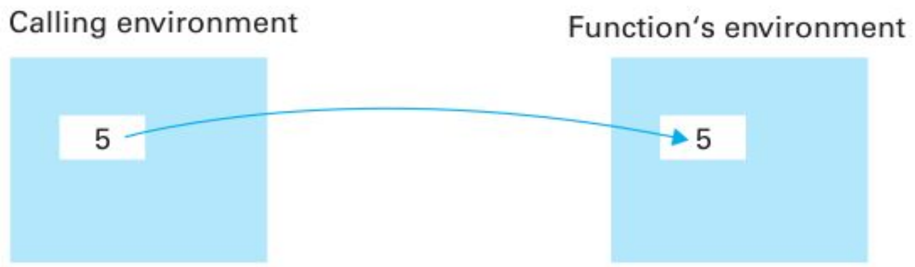
```
void ProjectPopulation (float GrowthRate)

{ int Year;

  Population[0] = 100.0;
  for (Year = 0; Year =< 10; Year++)
  Population[Year+1] = Population[Year] + (Population[Year] * GrowthRate);
}
```

This declares a local variable named Year.

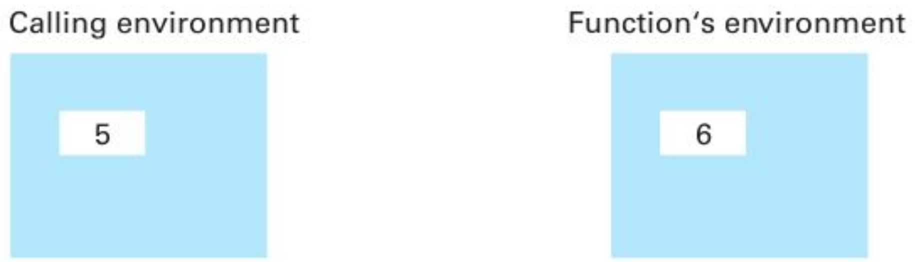
These statements describe how the populations are to be computed and stored in the global array named Population.

Procedural Units

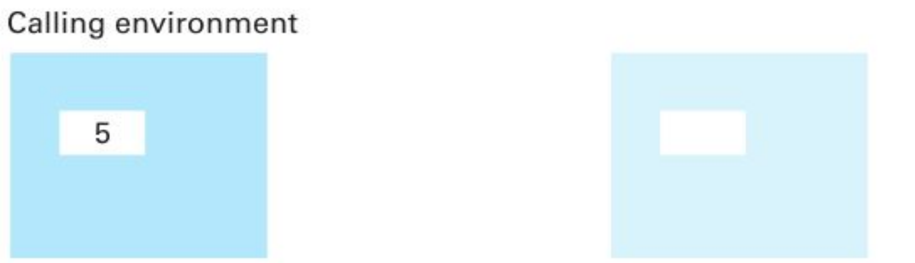
a. When the function is called, a copy of the data is given to the function



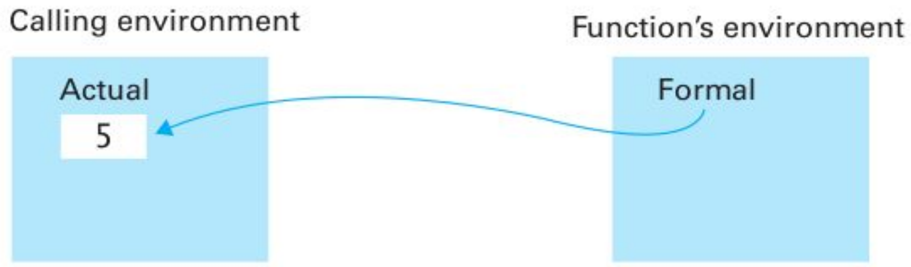
b. and the function manipulates its copy.



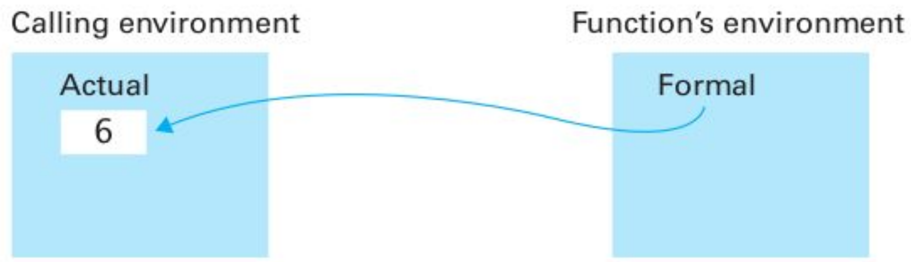
c. Thus, when the function has terminated, the calling environment has not been changed.



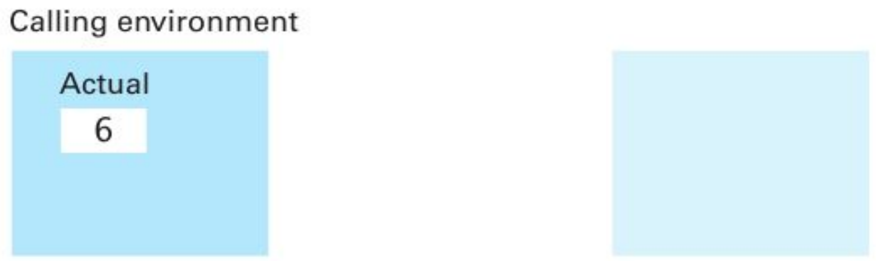
a. When the function is called, the formal parameter becomes a reference to the actual parameter.



b. Thus, changes directed by the function are made to the actual parameter

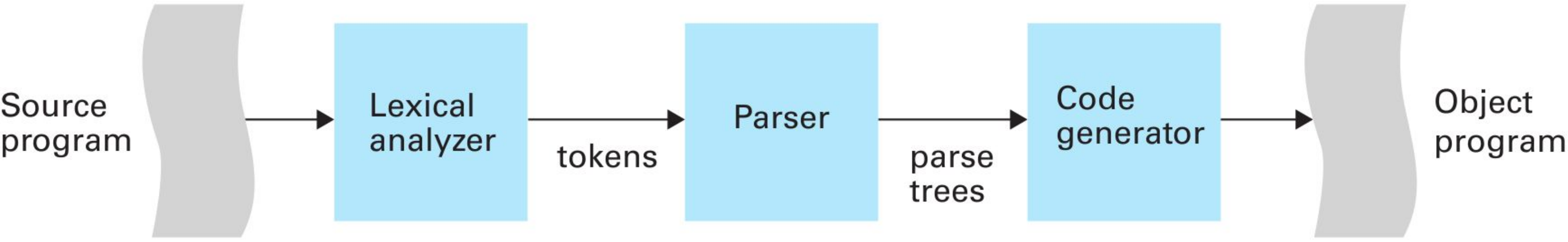


c. and are, therefore, preserved after the function has terminated.



Language Implementation

Translation: The process of converting a program from one language to another



The Translation Process

Lexical analysis is the process of recognizing which strings of symbols from the source program represent a single entity, or token.

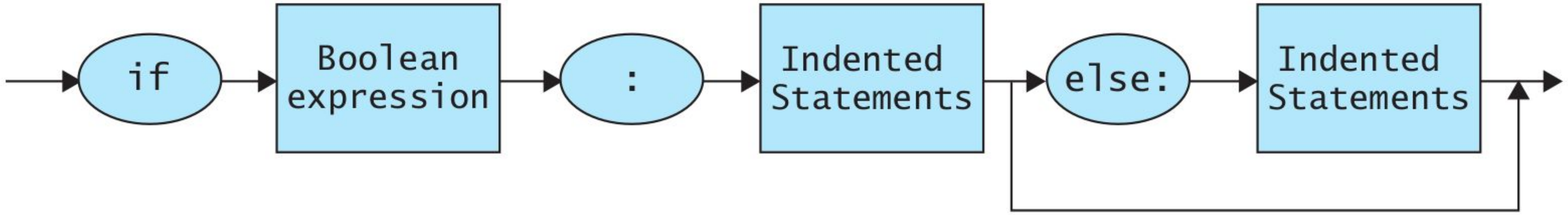
Lexical analyzer reads the source program symbol by symbol, identifying which groups of symbols represent tokens, and classifying those tokens according to whether they are numeric values, words, arithmetic operators, and so on.

The Translation Process

The **parsing process** is based on a set of rules that define the syntax of the programming language.

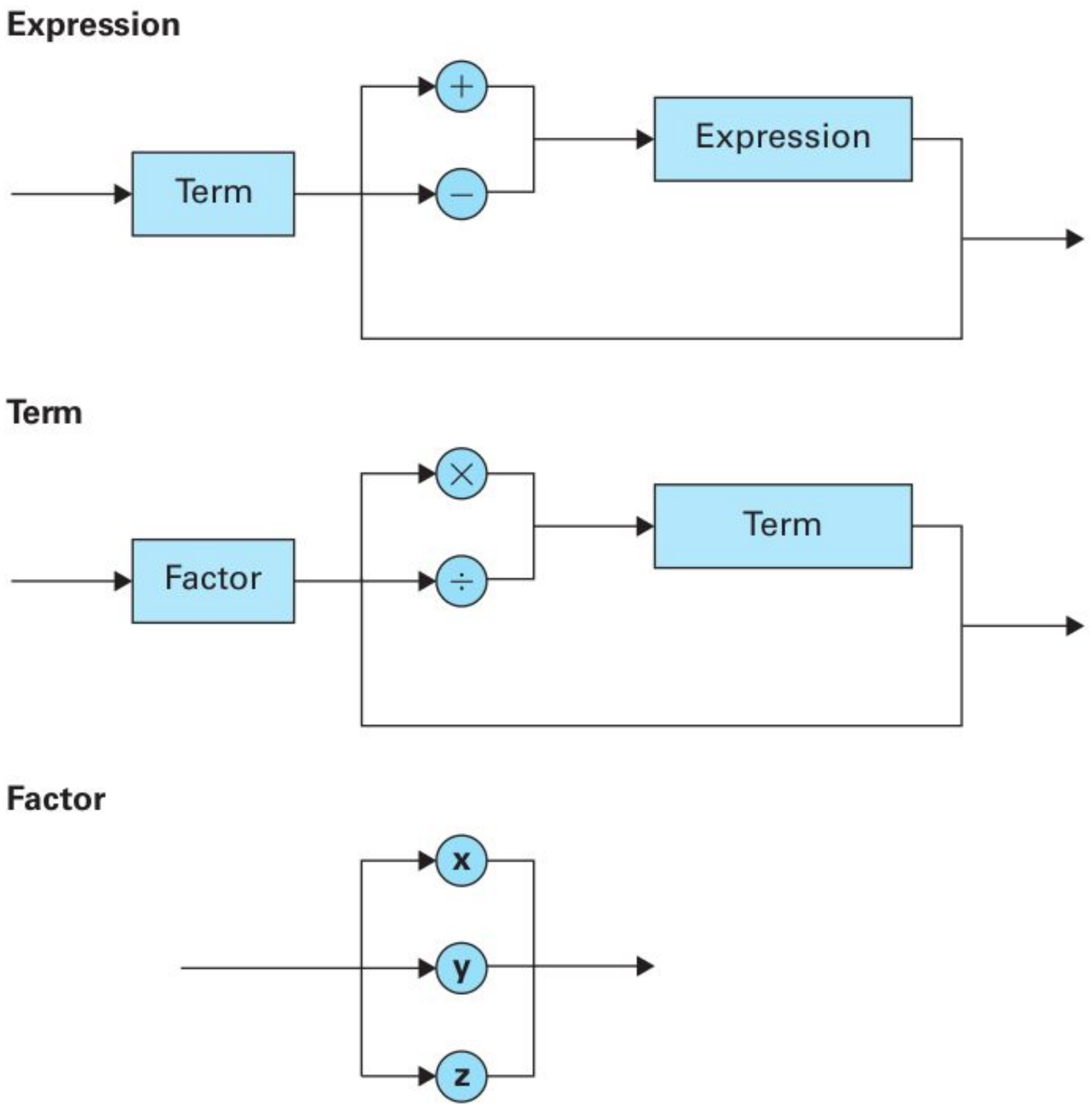
Collectively, these rules are called a grammar.

One way of expressing these rules is by means of syntax diagrams, which are pictorial representations of a language’s grammatical structure.



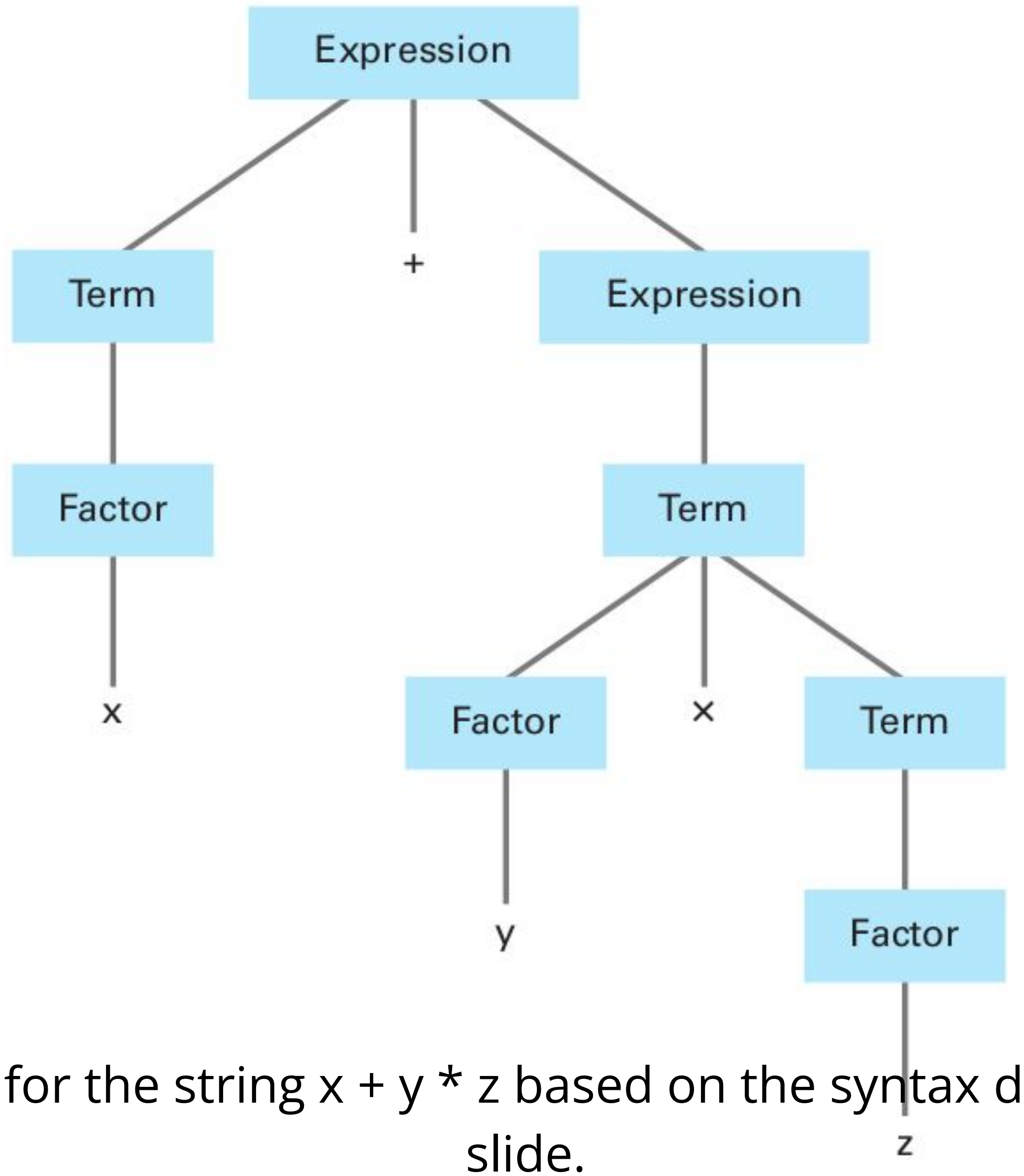
A syntax diagram of Python's if-else statement

The Translation Process



Syntax diagrams describing the structure of a simple algebraic expression

The Translation Process



The parse tree for the string x + y * z based on the syntax diagrams in prev. slide.

The Translation Process

The final activity in the translation process is **code generation**, which is the process of constructing the machine-language instructions to implement the statements recognized by the parser.

$$\left.\begin{array}{l} x = y + z \\ w = x + z \end{array}\right\} \text{Requires code optimization}$$

Object-Oriented Programming

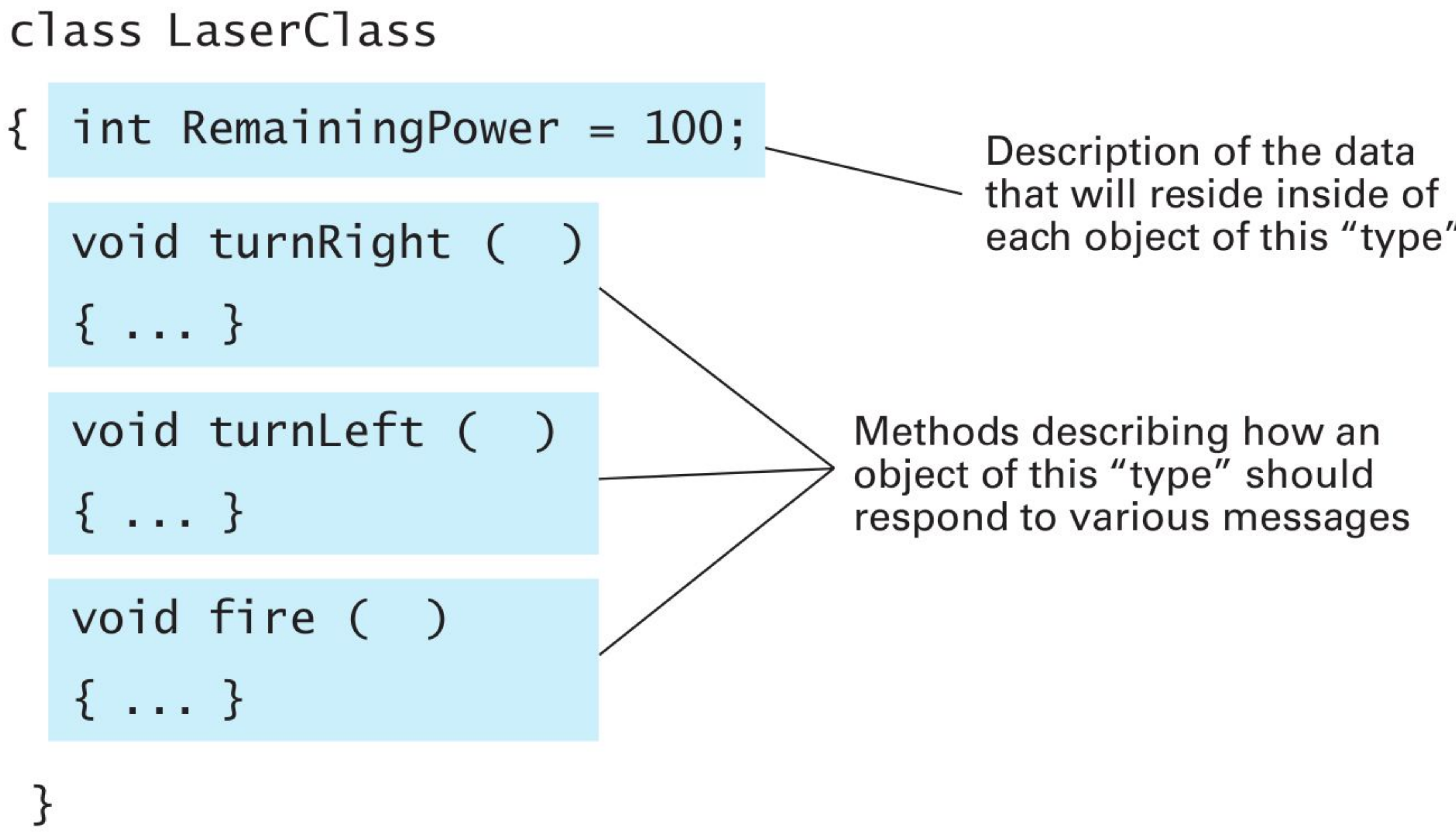
object-oriented programming languages provide statements for describing objects and their behavior.

C++, Java, and C#, which are three of the more prominent object-oriented languages used today

In the object-oriented paradigm a template for a collection of objects is called a **class**.

```
class Name
{
      .
      .
      .
}
```

Object-Oriented Programming



```
class LaserClass
{ int RemainingPower = 100;
  void turnRight (  )
  { ... }
  void turnLeft (  )
  { ... }
  void fire (  )
  { ... }
}
```

Description of the data that will reside inside of each object of this “type”
Methods describing how an object of this “type” should respond to various messages

Object-Oriented Programming

Once we have described the class LaserClass in our game program, we can declare three variables Laser1, Laser2, and Laser3 to be of “type” LaserClass

by a statement of the form

```
LaserClass Laser1, Laser2, Laser3;
```

Note that this is the same format as the statement

```
int x, y, z;
```

Object-Oriented Programming

Once we have declared the variables Laser1, Laser2, and Laser3 to be of “type” LaserClass, we can assign them values.

- In this case the values must be objects that conform to the “type” LaserClass.

LaserClass Laser1, Laser2, Laser3;

- not only establishes the variables Laser1, Laser2, and Laser3, but also creates three objects of “type” LaserClass, one as the value of each variable.

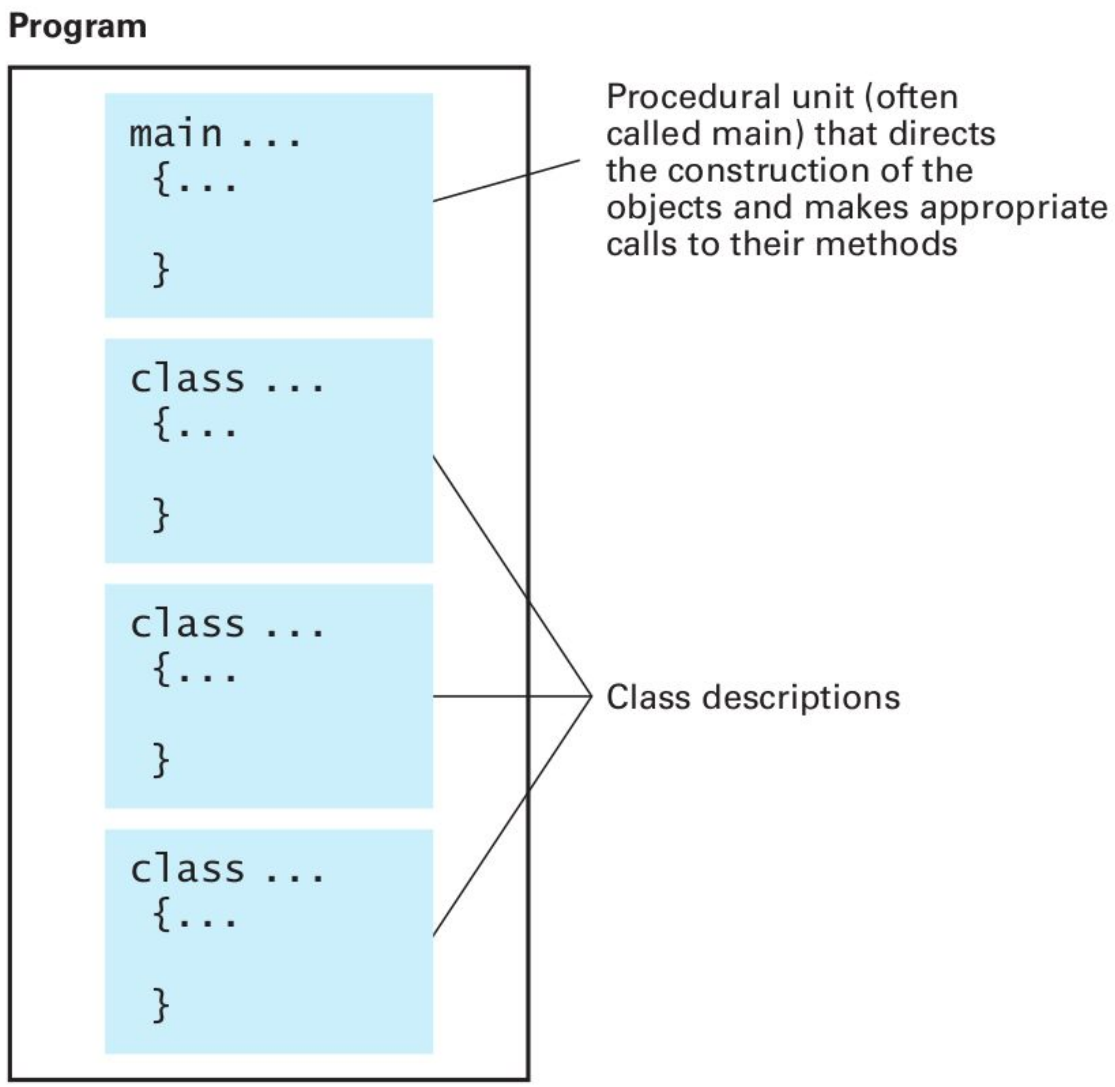
Object-Oriented Programming

```
LaserClass Laser1 = new LaserClass();
```

declares the variable Laser1 to be of “type” LaserClass and also creates a new object using the LaserClass template and assigns that object as the starting value of Laser1.

```
Laser1.fire();

Laser2.turnLeft();
```

Object-Oriented Programming

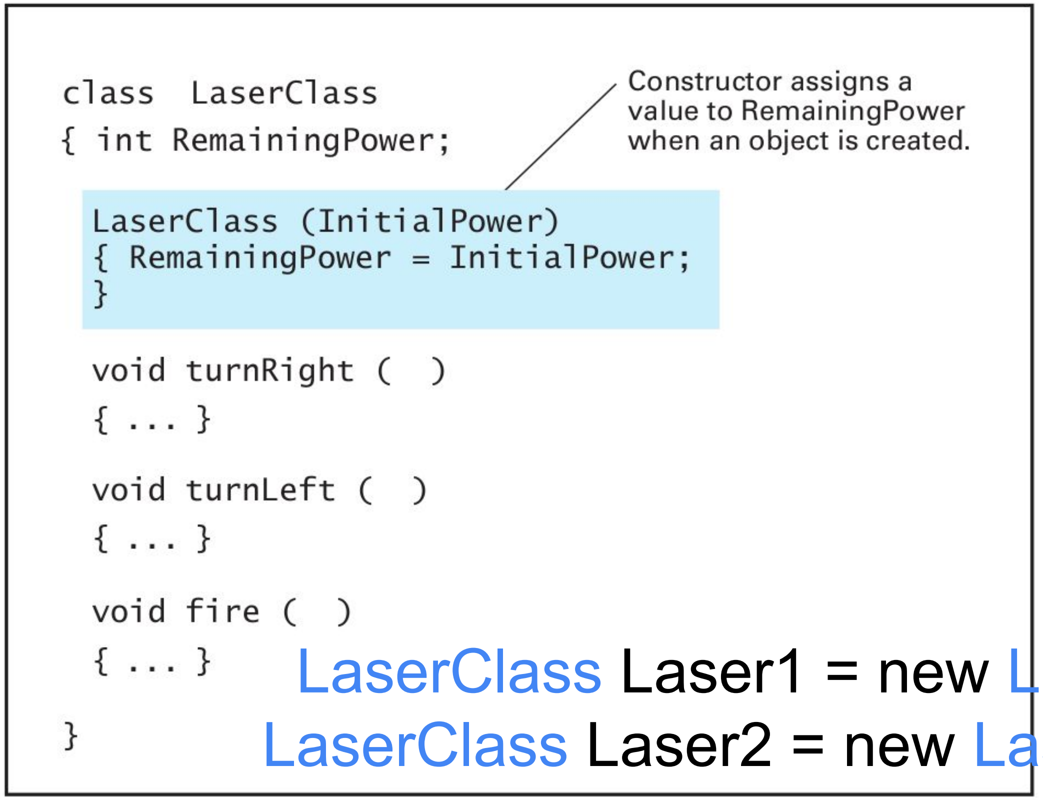

Program
main ...
{...
}
class ...
{...
}
class ...
{...
}
class ...
{...
}
Procedural unit (often called main) that directs the construction of the objects and makes appropriate calls to their methods
Class descriptions

The structure of a typical object-oriented program

Constructors

Constructors are executed automatically when an object is constructed from the class.

A constructor is identified within a class definition by the fact that it is a method with the same name as the class.



```
class  LaserClass
{ int RemainingPower;

  LaserClass (InitialPower)
  { RemainingPower = InitialPower;
  }

  void turnRight (  )
  { ... }

  void turnLeft (  )
  { ... }

  void fire (  )
  { ... }
}
```

Constructor assigns a value to RemainingPower when an object is created.

```
LaserClass Laser1 = new LaserClass(50);
LaserClass Laser2 = new LaserClass(100);
```

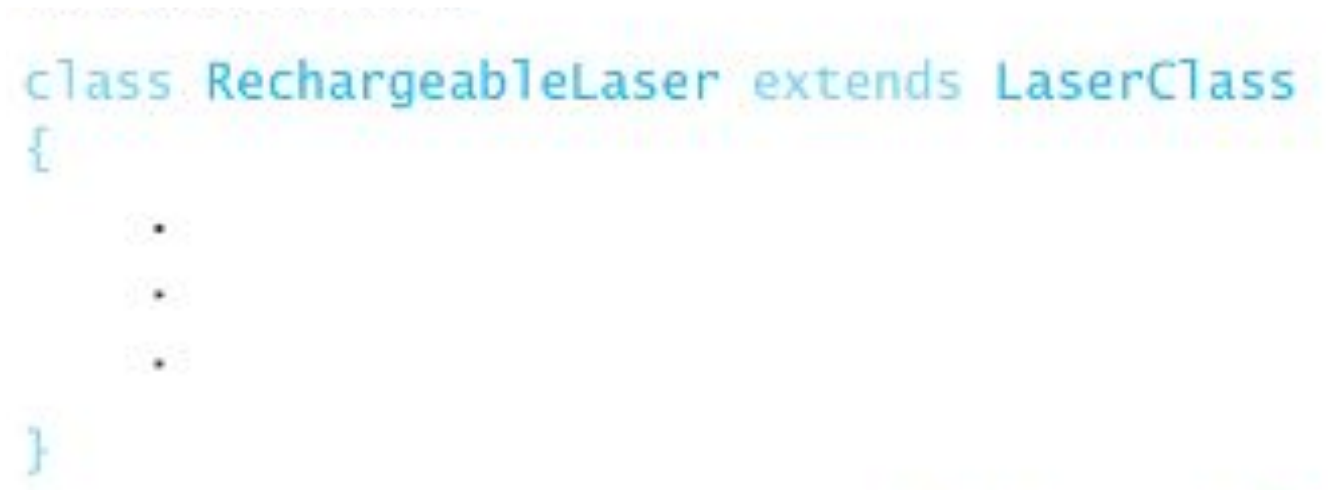
Object-Oriented Programming

The four basic concepts of OOP are

- Inheritance,
- Abstraction,
- Polymorphism and
- Encapsulation

Inheretance

Object-oriented languages allow one class to encompass the properties of another through a technique known as **inheritance**.



```
class RechargeableLaser extends LaserClass
{
      .
      .
      .
}
```

LaserClass Laser1, Laser2;

RechargeableLaser Laser3, Laser4;

Abstraction

Abstraction is the process of hiding the internal details of an application from the outer world.

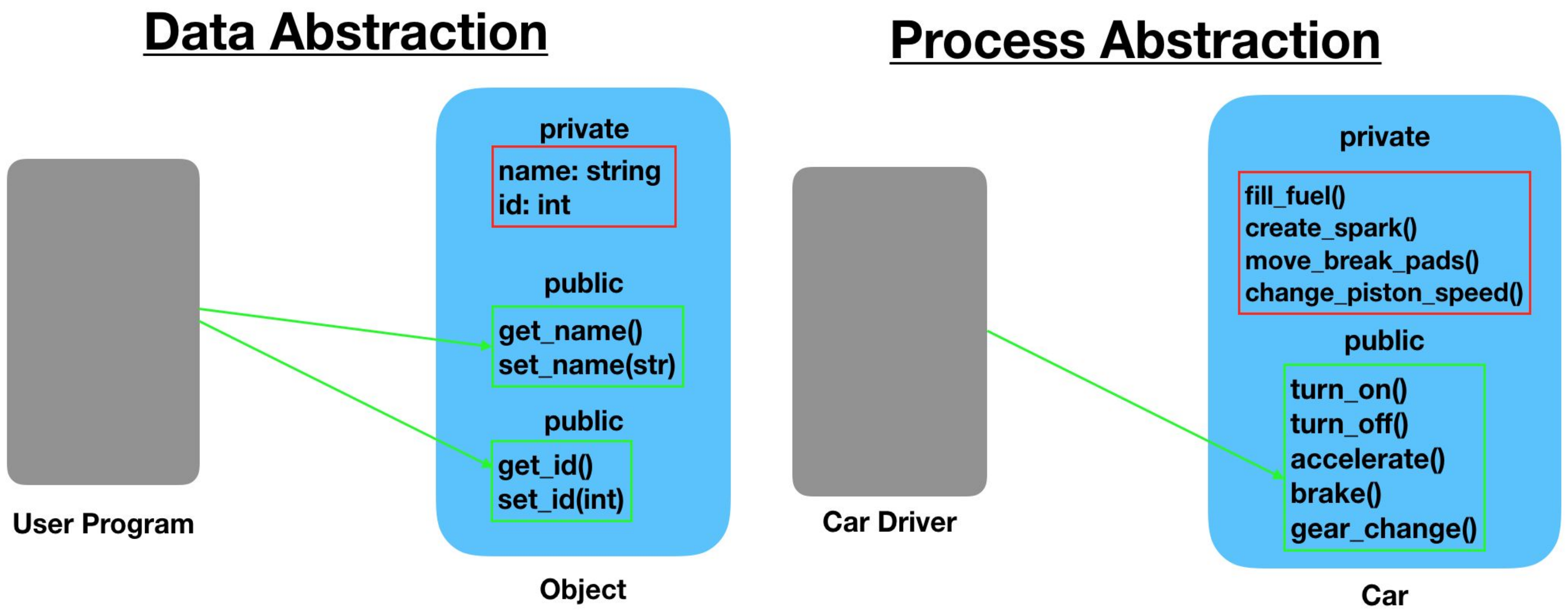
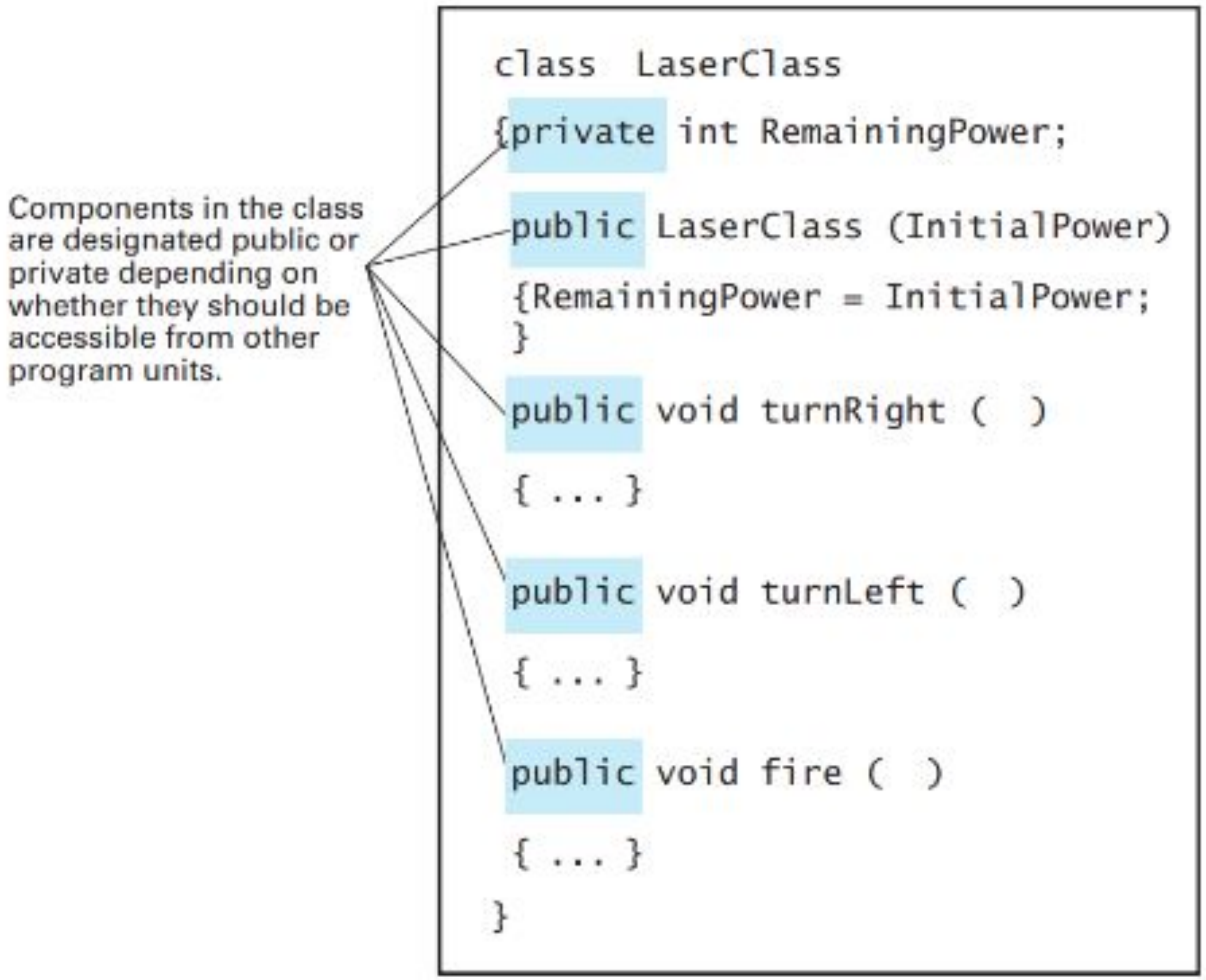


Figure from: https://www.digitalocean.com/community/tutorials/what-is-abstraction-in-oops

Polymorphism

"Now Speak!"
"Quack"
"Woof"
"Meow"
arbabwaseer@gmail.com

Encapsulation


Components in the class are designated public or private depending on whether they should be accessible from other program units.

```
class  LaserClass
{private int RemainingPower;
 public LaserClass (InitialPower)
 {RemainingPower = InitialPower;
 }
 public void turnRight (  )
 { ... }
 public void turnLeft (  )
 { ... }
 public void fire (  )
 { ... }
}
```

Encapsulation

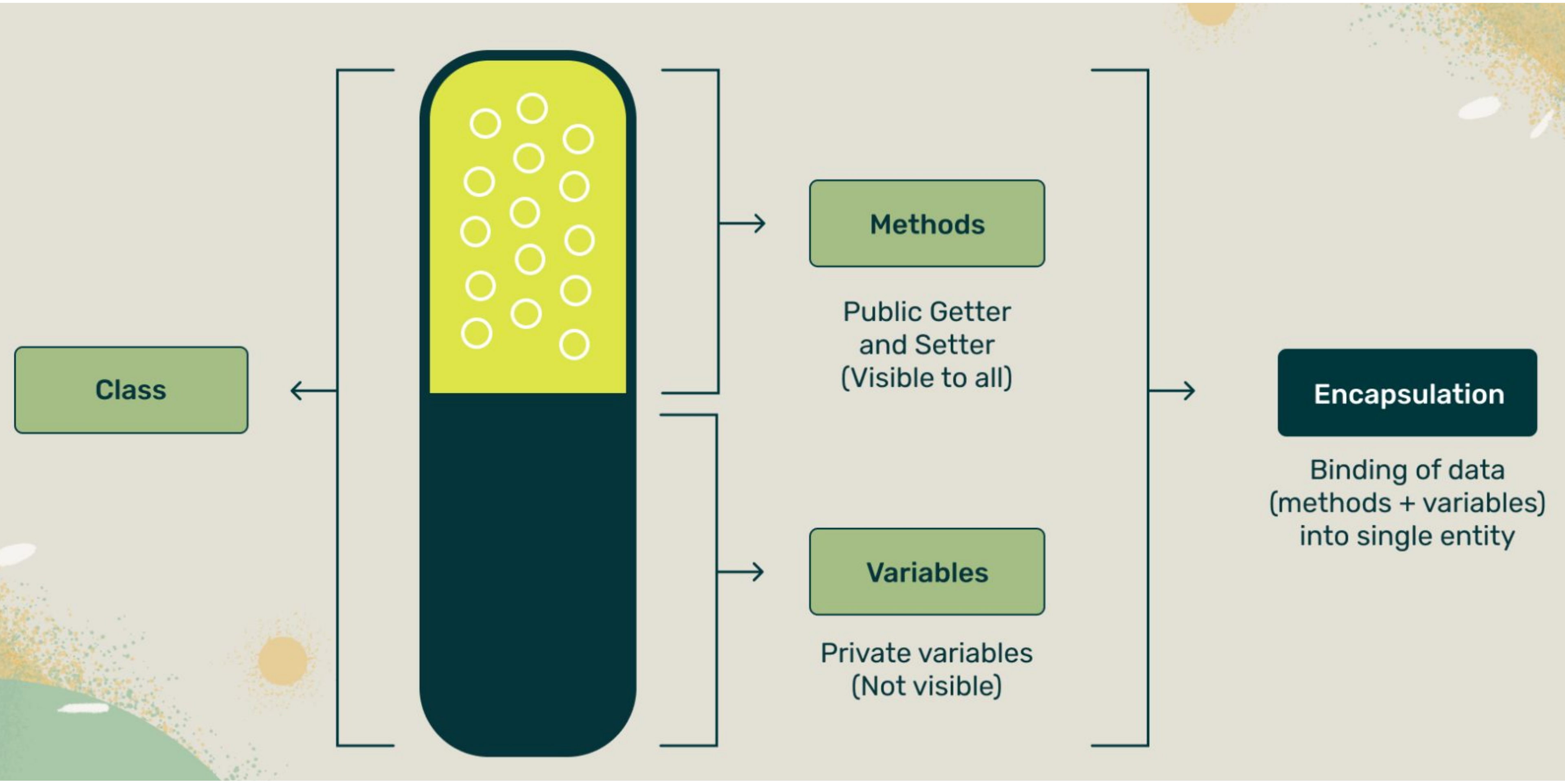


Figure from https://www.crio.do/blog/encapsulation-in-java/

Research Themes

1. Debugging.
2. IDEs and Coding Environments.
3. Probabilistic programming

COM1013
INTRODUCTION TO COMPUTER SCIENCE

Lecturer: Begüm MUTLU BİLGE, PhD

begummutlubilge+com1013@gmail.com (recommended)
bmbilge@ankara.edu.tr