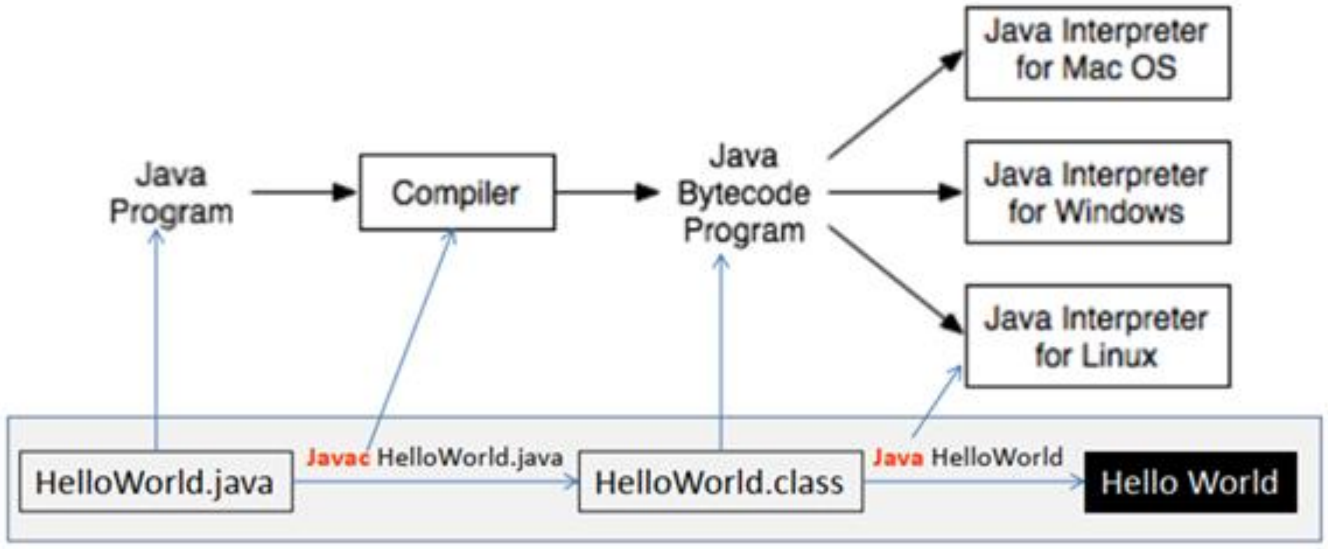
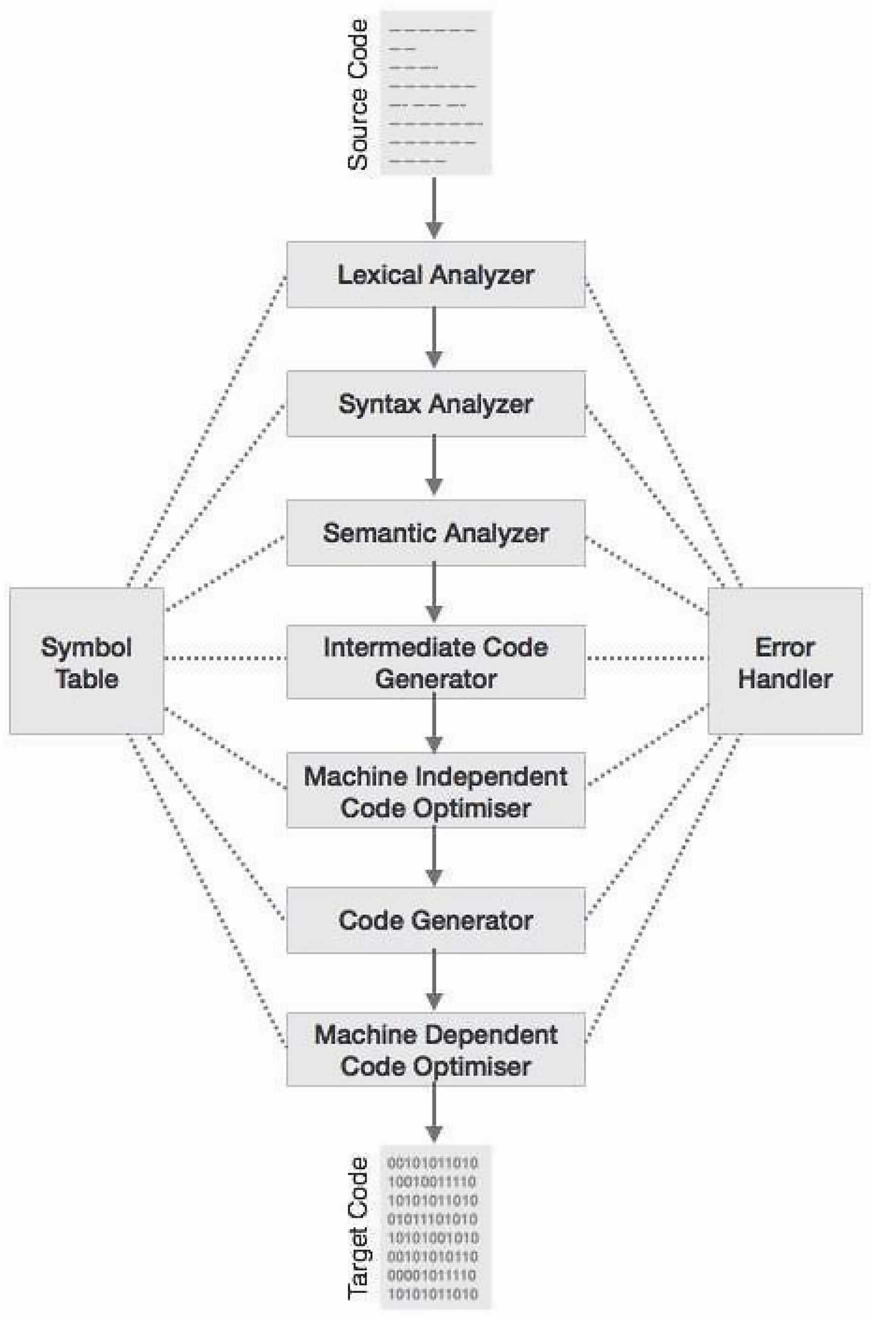
# Lecture 2

## Introduction to Compiler, Virtual Machine and Basic OOP

## Introduction to Compiler

- Unlike other programming languages, Java is designed to compile source code into bytecode instead of directly into machine code
- The bytecode is then interpreted by the execution environment and translated into machine code before running.





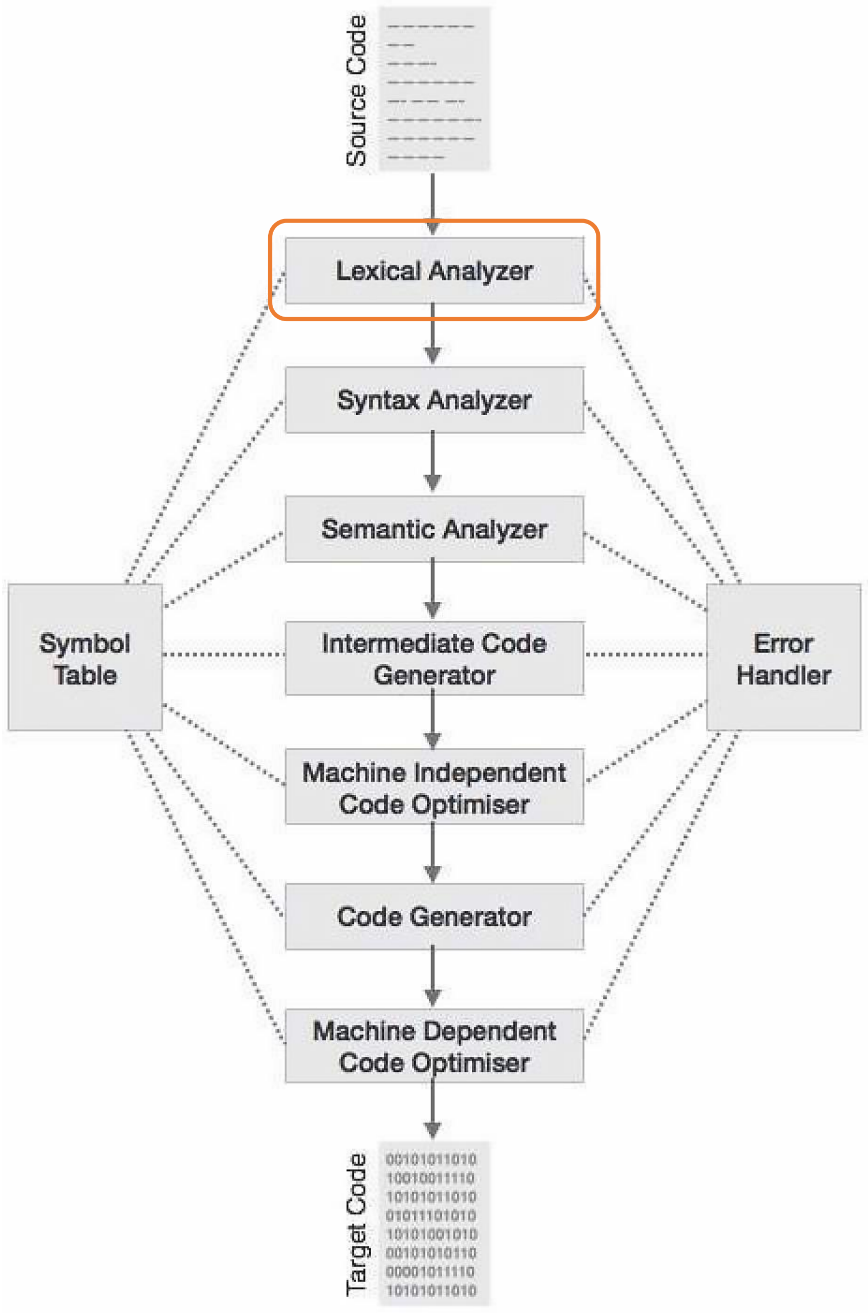
# Compiler operations

Operation is also called a phase

Operations are organized into two groups: front end, back end

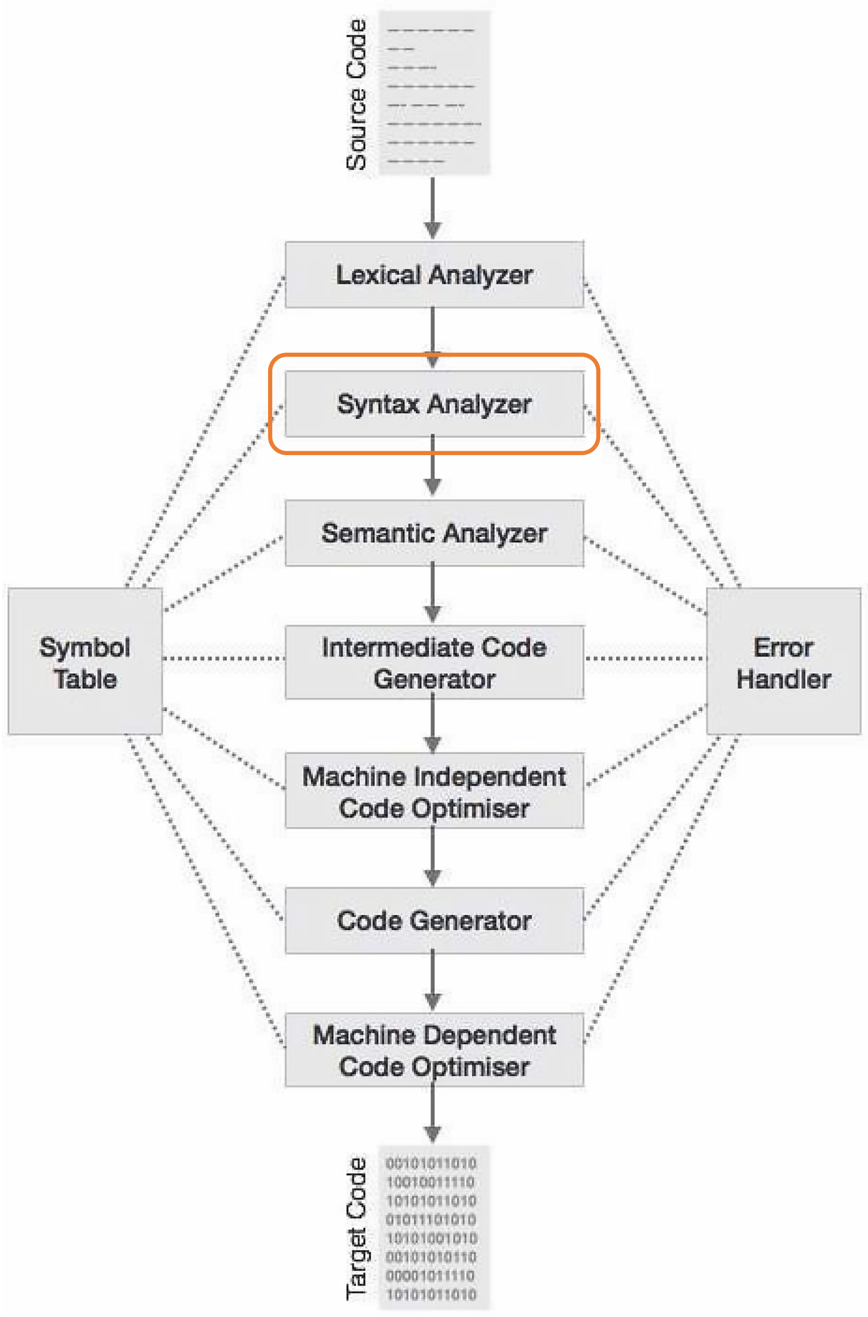
**Front end:**
- Lexical and Syntax Analysis
- Semantic Analysis

**Back end:**
- Target code generation
- Code improvement
- A pass is a sequence of one or more phases



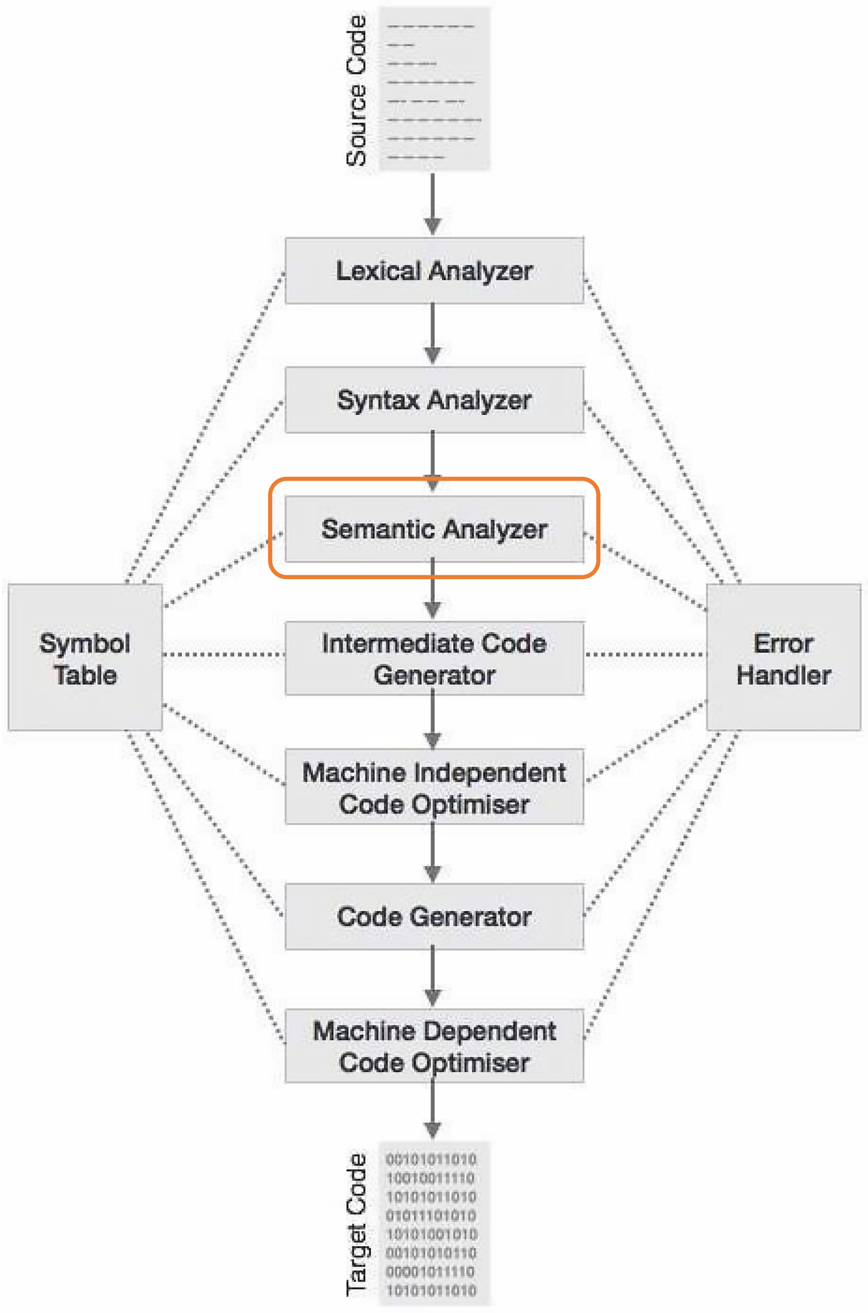
# Lexical Analysis (Scanner)

- Analyzes the source code into tokens.
- Removes whitespaces and comments.



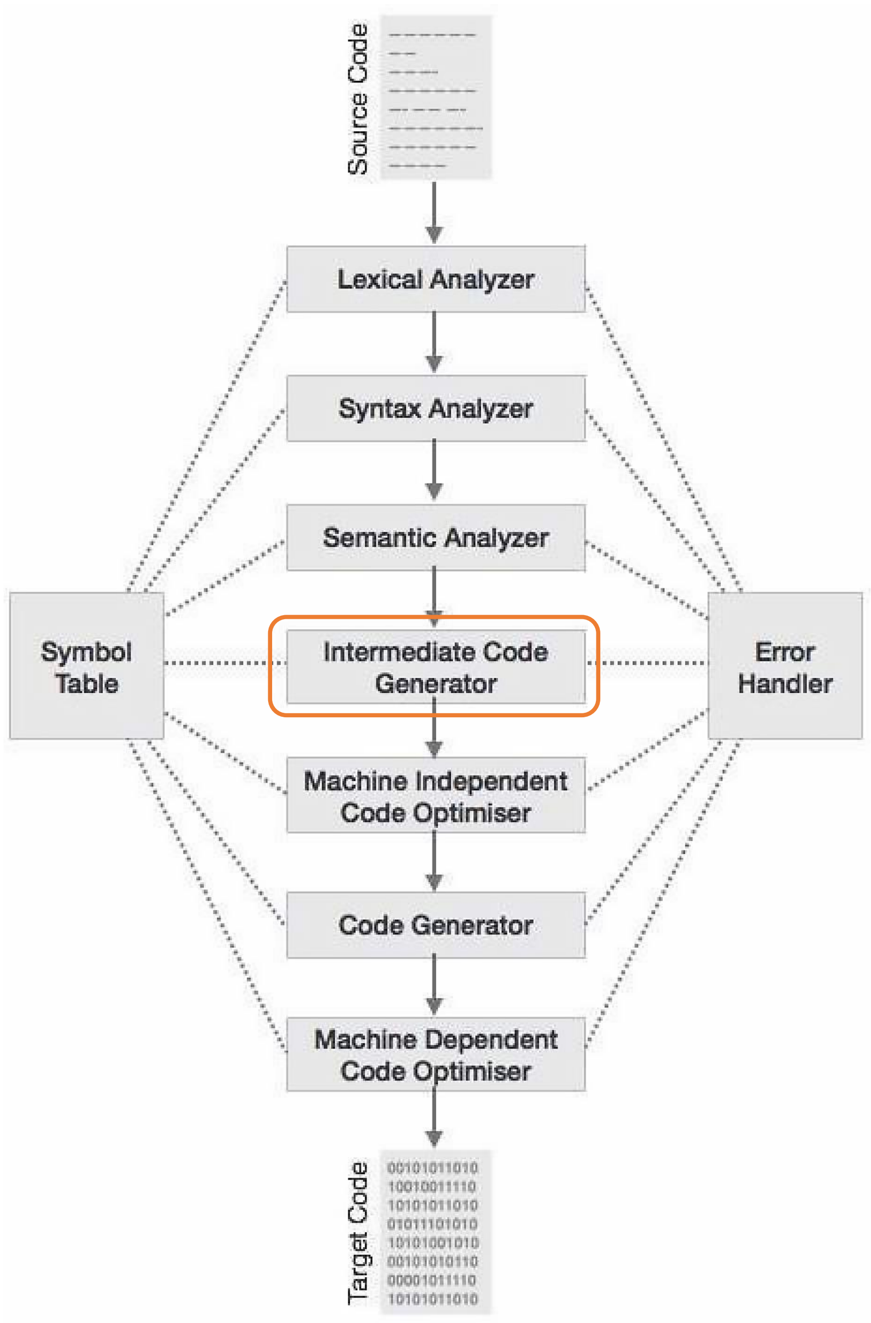
## Syntax Analysis (Parser)

- Checks for syntactic correctness.
- Builds a syntax tree from tokens.



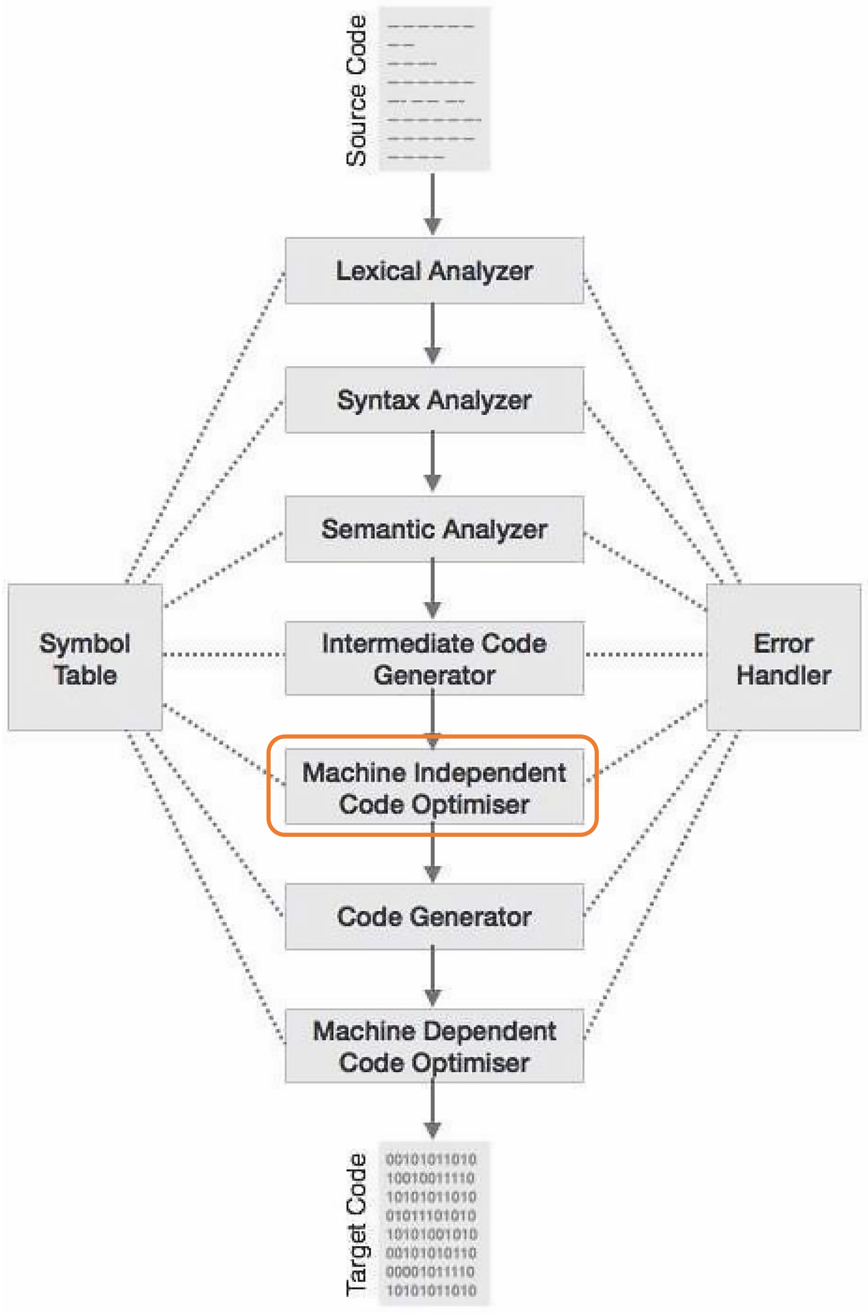
# Semantic Analysis

- Checks the logic and correctness of the program.
- Determines the meaning of expressions and statements.



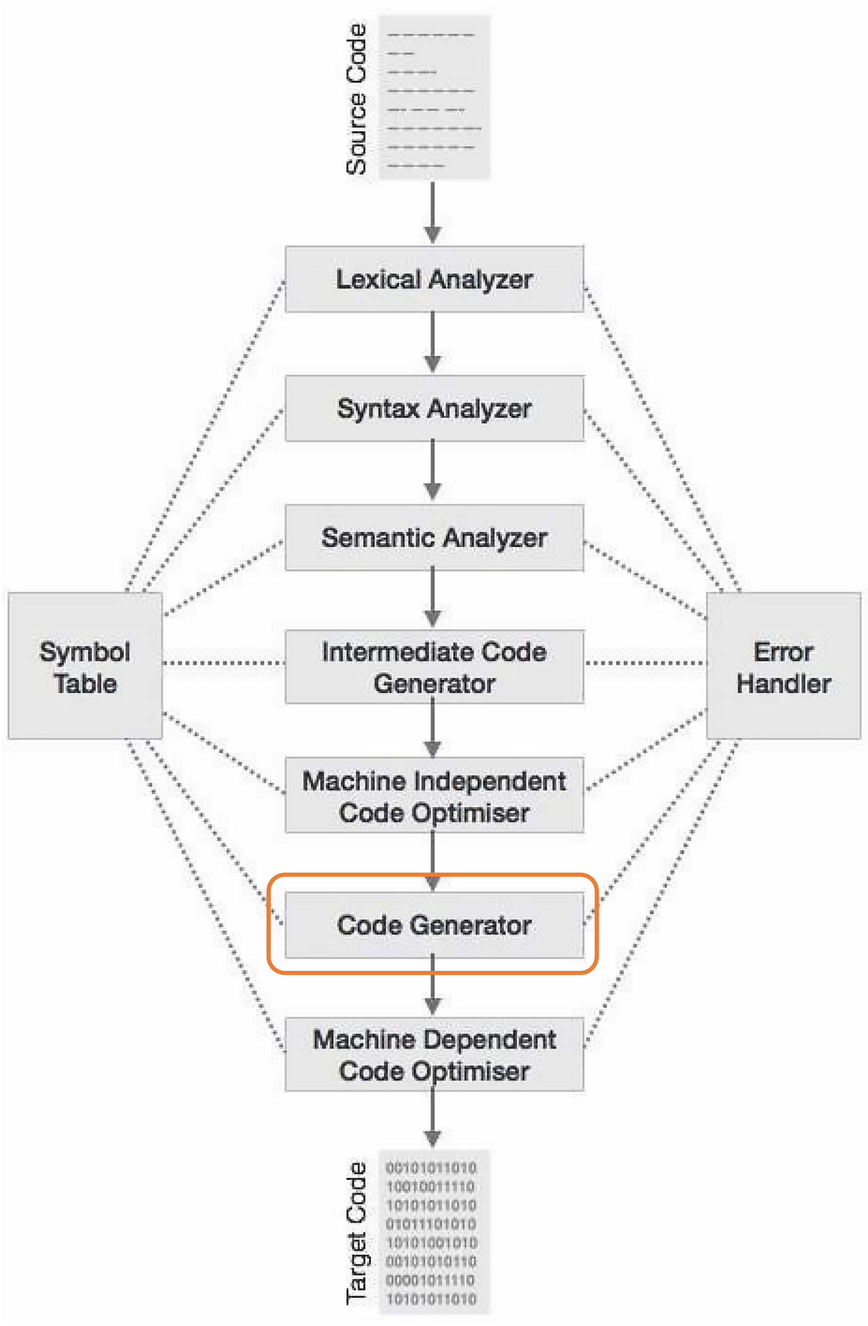
# Intermediate Code Generation

- Generates an intermediate code between source code and machine code.



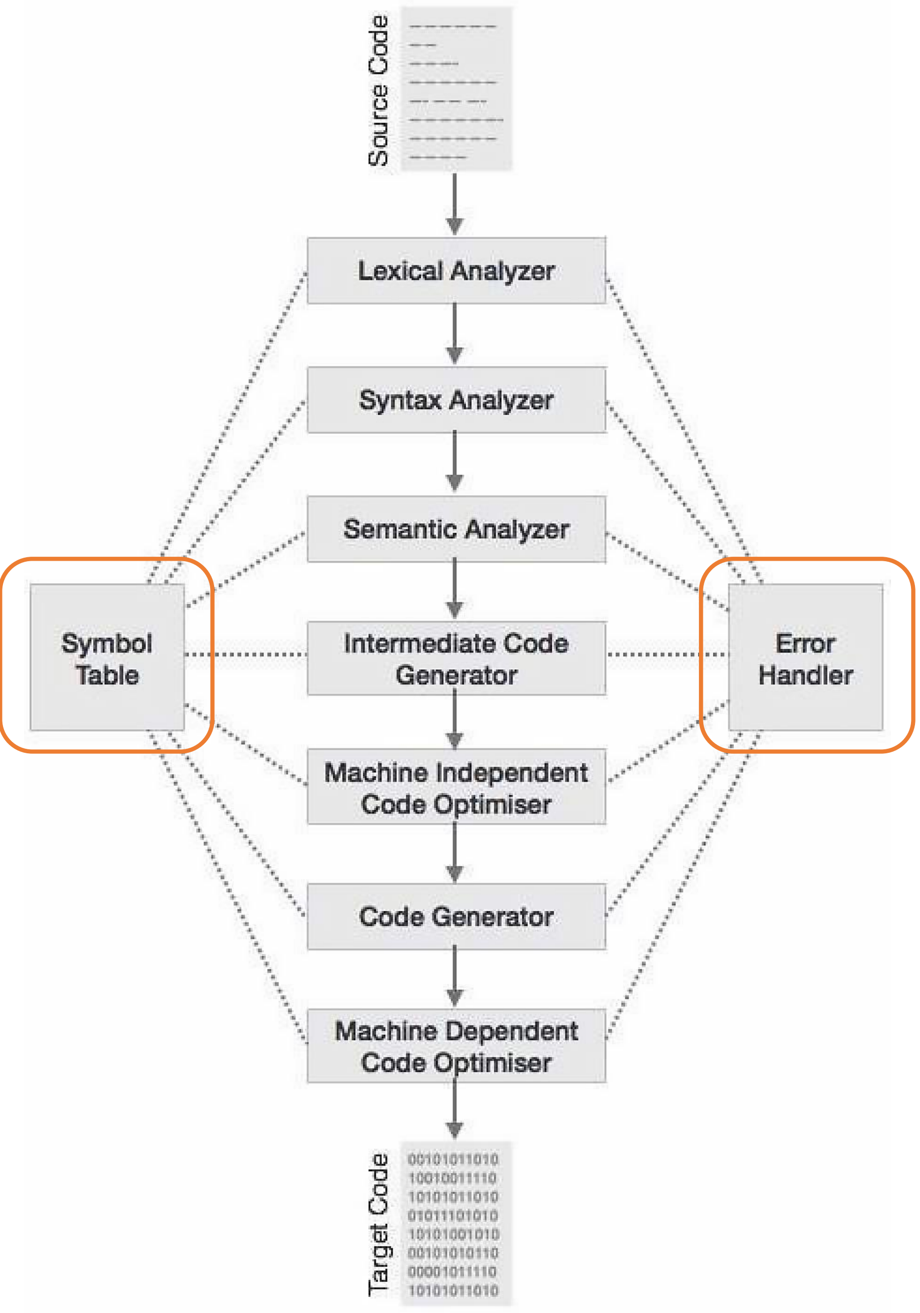
# Code Optimization

- Optimizes the intermediate code to improve performance.



## Code Generation

- Generates machine code from the intermediate code.

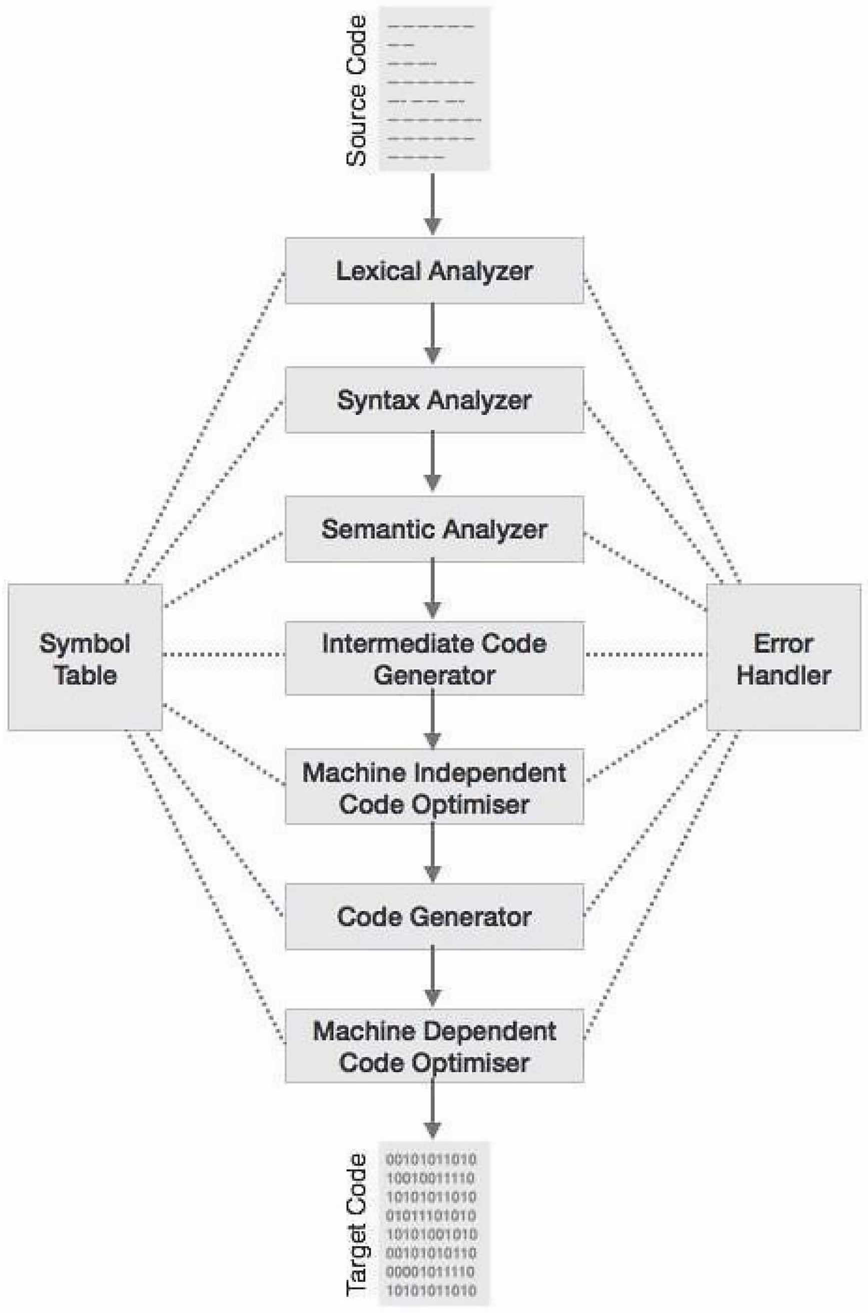


## Symbol Table Management

- Manages a symbol table to track information about variables, functions, etc.

## Error Handling

- Handles errors that may occur during compilation.



# Executable File Creation

- Combines machine code into an executable file.
- Links libraries and creates a standalone executable file.

## Introduction to Virtual Machine

- A software emulation of computer hardware
  Includes:
  - an ISA (Instruction Set Architecture) to express programs
  - shared services: e.g. I/O, scheduling, memory management
- Two types of VM:
  - System VM: emulation to support a full-featured OS
  - Process VM: emulation to support a user-level process
    - e.g. Java Virtual Machine

## Advantages of Virtual Machines

1. **Resource Optimization:**

   Virtual machines consolidate multiple workloads on a single server, optimizing resource usage and improving overall efficiency.

2. **Isolation and Security:**

   VMs provide a layer of isolation, preventing issues in one virtual machine from affecting others, enhancing system security and stability.

3. **Flexibility and Scalability**

   Virtualization allows for easy creation and scaling of virtual machines, providing flexibility to adapt to changing workloads and ensuring efficient resource allocation.

## Compiler vs. Virtual Machine

**Key Differences:**

- Compiler converts the entire source code into machine code.
- Virtual Machine executes the source code directly.
    - Like an interpreter

## Trade-offs: Compiler vs. Virtual Machine

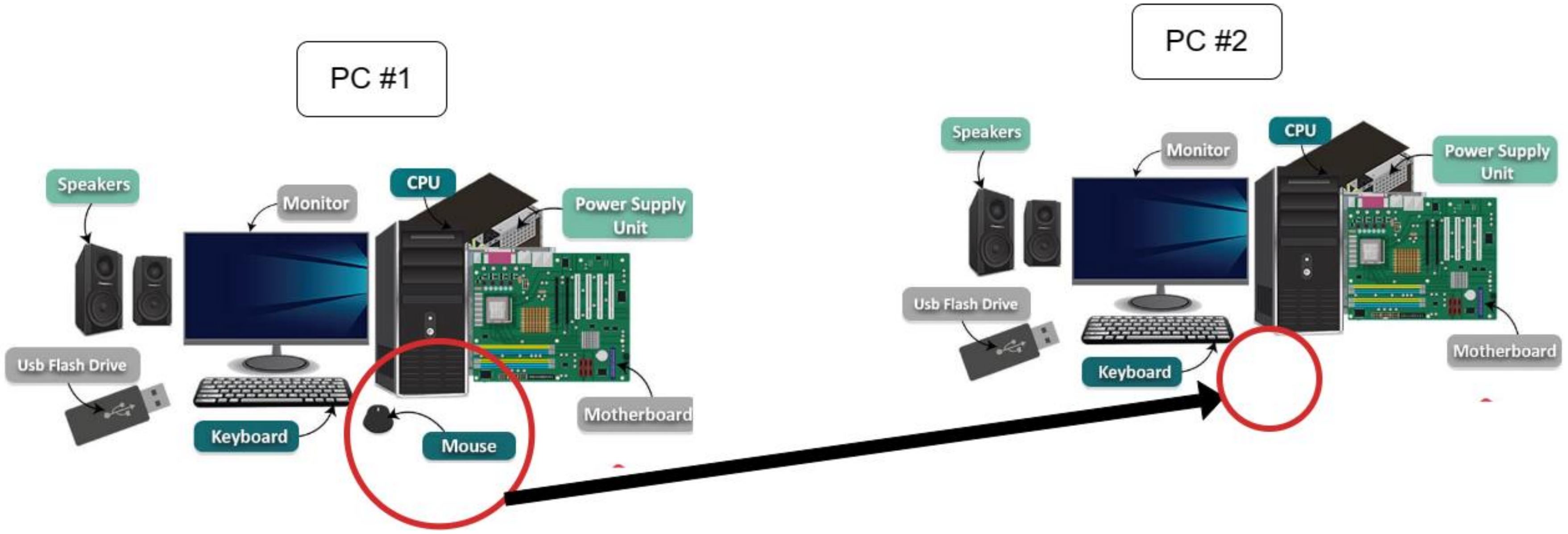
- Virtual Machine reduces compilation time but introduces overhead during execution.
- Compiler enhances performance but may increase compilation time.

## Getting Started with Object-Oriented Programming (OOP)

# Why use OOP?

**Hardware Assembly Analogy**

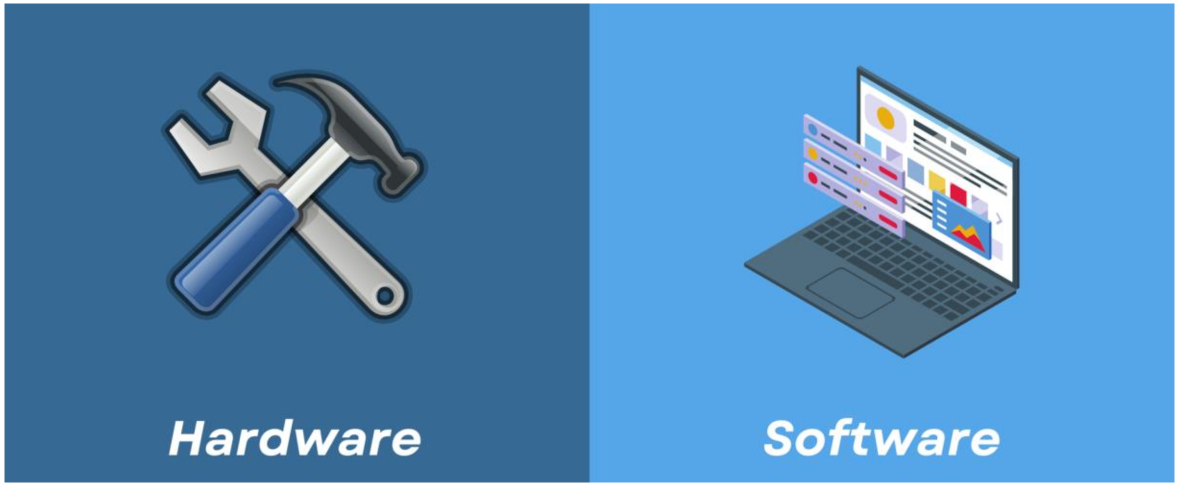
- Assembling a PC involves combining reusable components with standardized interfaces.
- Hardware components, like mouse or motherboards, can be easily reused in different systems.



## Why use OOP?

**Software Challenge**

- Unlike hardware, software development traditionally required writing code from scratch for each new application.
- Difficulty in reusing software components led to reinventing code for similar functionalities in different projects.



## Why use OOP?

1. Object-Oriented Programming (OOP) addresses the challenge of code reuse.
2. OOP promotes the creation of modular and reusable code components modeled as objects.
3. Objects encapsulate data and behavior with standardized interfaces, facilitating easy integration into various applications.

## Introduction to Object

- Object: models a
  - Real world object (e.g. computer, book, box…)
  - Concept (e.g. meeting, interview…)
  - Process (e.g. sorting a stack of papers, comparing two computers’ performance)
- Each object is characterized by its own attributes and behaviors.

# Attributes and Behavior

**Car Object**

- Attributes
  - Color
  - Model
  - Type
  - Cylinder
- Behaviors (what can a car do?)
  - Start
  - Stop
  - Reverse
  - Accelerate

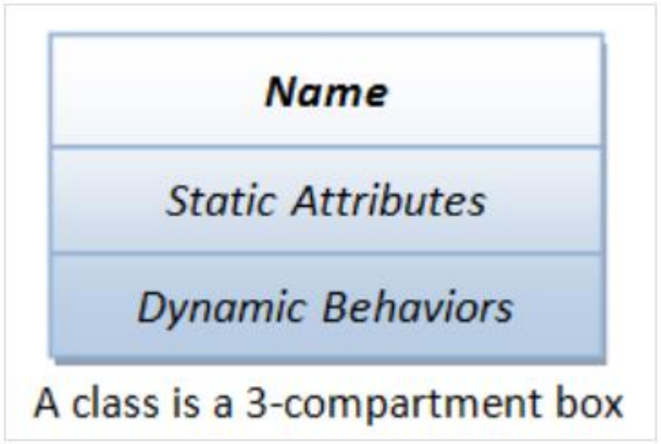
## What is Class?

- A `class` in Java serves as a blueprint or template for objects of the same type.
- It defines both static attributes and dynamic behaviors common to all objects of that class.

## Instances in Java

- An instance is a realization of a specific item defined by a class. It represents a unique instantiation of the class, sharing properties described in the class definition.
- Example:
  - If you have a class named `Student`, you can create instances for individuals like Alice, Bob and Ali.
  - The term "object" is often used interchangeably with "instance".
  - In a broader context, "object" may refer to both a class or an instance, although it commonly denotes instances in Java.

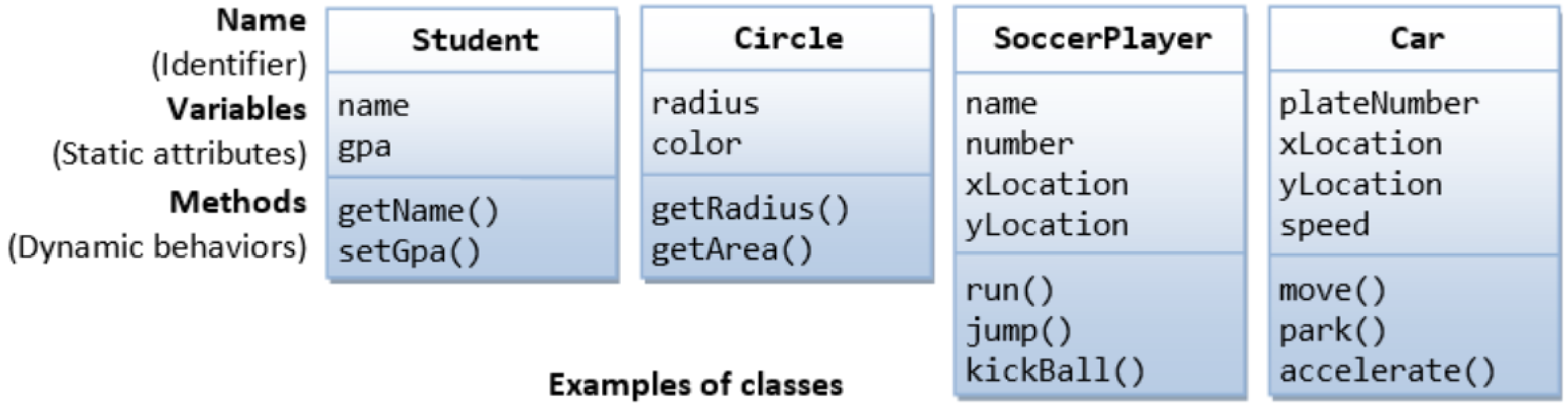
## 3-Compartment Box



A class is a 3-compartment box

- A class can be visualized as a three-compartment box, as illustrated:
    1. Name (or identity): identifies the class.
    2. Variables (or attribute, state, field): contains the static attributes of the class.
    3. Methods (or behaviors, function, operation): contains the dynamic behaviors of the class.

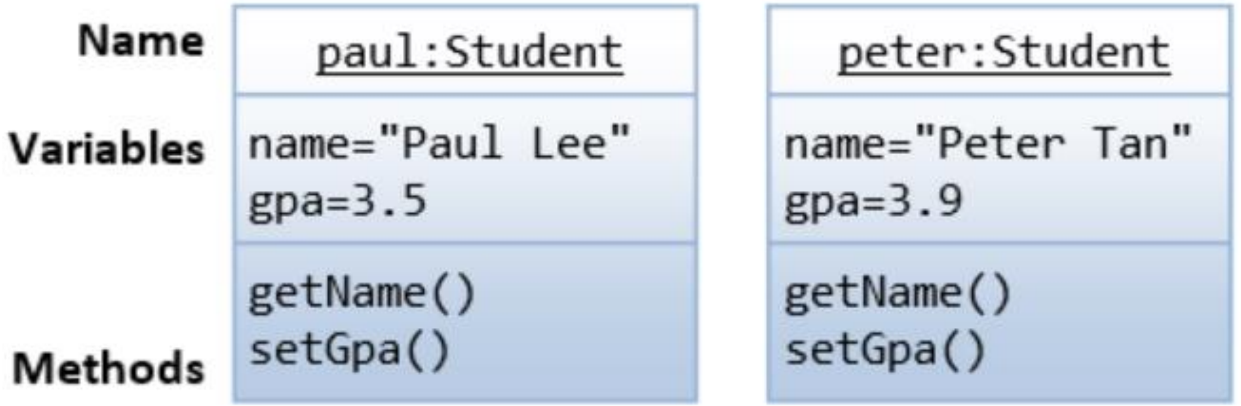
## Examples of classes



Examples of classes

# Examples of classes

- The following figure shows two instances of the class `Student`, identified as `paul` and `peter`.



**Two instances - paul and peter - of the class Student**

- Extended information: Unified Modeling Language (UML) Class and Instance Diagrams: The above class diagrams are drawn according to the UML notations.

## Class Definition in Java

- The syntax for class definition in Java is:

```
[AccessControlModifier] class ClassName {
   // Class body contains members (variables and methods)
   ......
}
```

## Class Definition in Java

- Example (1):

```
public class Circle {           // class name
   double radius;               // variables
   String color;

   double getRadius() { ...... }  // methods
   double getArea() { ...... }
}
```

## Class Definition in Java

- Example (2):

```
public class SoccerPlayer {  // class name
   int number;               // variables
   String name;
   int x, y;

   void run() { ...... }     // methods
   void kickBall() { ...... }
}
```

## Creating Instances of a Class

- To create an instance of a class, you have to:
  - Declare an instance identifier (instance name) of a particular class.
  - Construct the instance (i.e., allocate storage for the instance and initialize the instance) using the `new` operator.

## Creating Instances of a Class

- For example: suppose that we have a class called `Circle`, we can create instances of `Circle` as follows:

```
// Declare 3 instances of the class Circle, c1, c2, and c3
Circle c1, c2, c3;  // They hold a special value called null
// Construct the instances via new operator
c1 = new Circle();
c2 = new Circle(2.0);
c3 = new Circle(3.0, "red");

// You can Declare and Construct in the same statement
Circle c4 = new Circle();
```

# Dot（.）Operator

- Use the dot operator（.）to reference the desired member variable or method.

```
// Suppose that the class Circle has variables radius and color,
// and methods getArea() and getRadius().
// Declare and construct instances c1 and c2 of the class Circle
Circle c1 = new Circle ();
Circle c2 = new Circle ();
// Invoke member methods for the instance c1 via dot operator
System.out.println(c1.getArea());
System.out.println(c1.getRadius());
// Reference member variables for instance c2 via dot operator
c2.radius = 5.0;
c2.color = "blue";
```

## Constructors

```
public class Baby {
    public Baby () {
        // nothing
    }
    public Baby(String name, boolean isMale) {
        // some codes
    }
}

Baby b1 = new Baby();
Baby b2 = new Baby("Alex", true);
```

A constructor is invoked when an instance is created.

## Constructors

- A constructor is like a method
- Constructor name is the same as class name
  - No return type – never returns anything
- Usually initialize fields
- All classes need at least one constructor
  - There can be more than one constructor.
  - There is a default constructor even if you don’t write one.

```
public ClassName () {
    // default constructor
}
```

## Example: Baby class' constructor

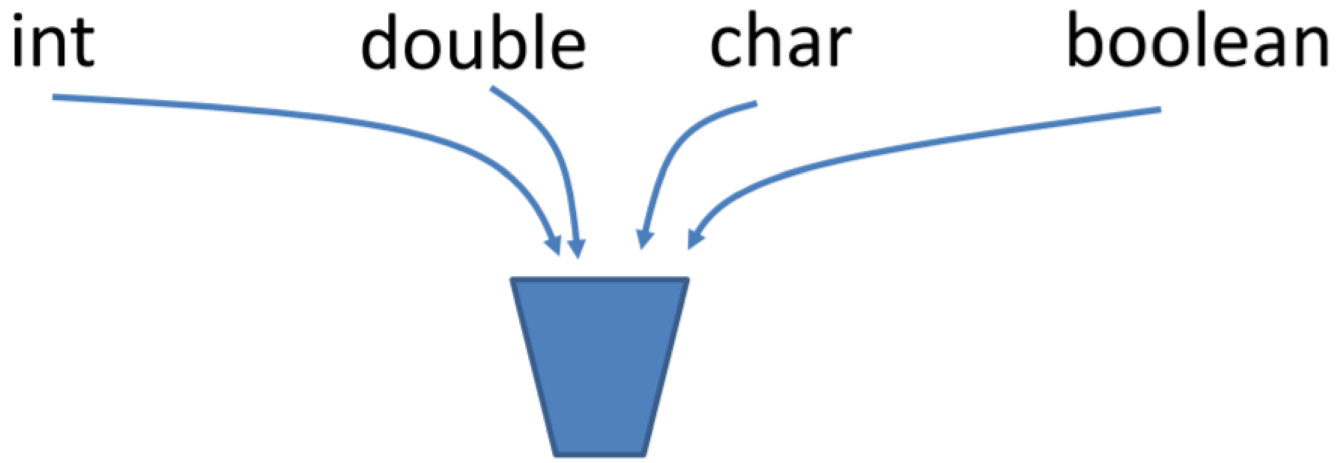
```
public class Baby {
    String name;
    boolean isMale;
    public Baby(String n, boolean m) {
        name = n;
        isMale = m;
    }
}
```

## Primitives vs References

- Primitive types are basic java types
  - `int, long, double, boolean, char, short, byte, float`
  - The actual values are stored in the variable
- Reference types are arrays and objects
  - `String, int[], Circle, …`
  - The variable only stores object's memory address

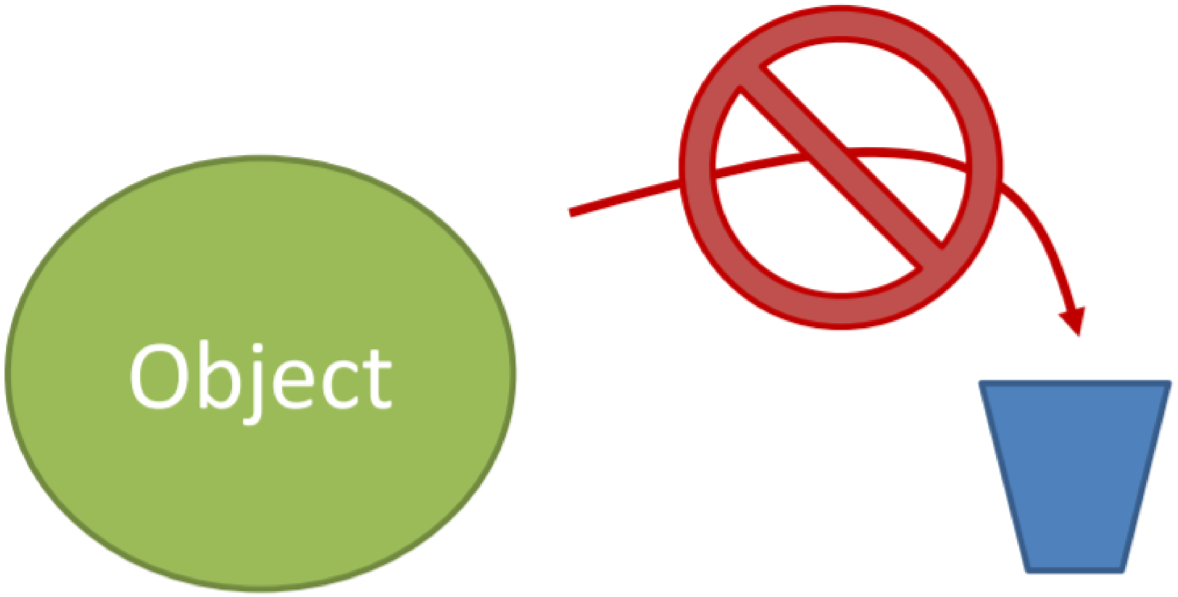
## How Java stores primitives

- Variables are like fixed size cups
- Primitives are small enough that they just fit into the cup



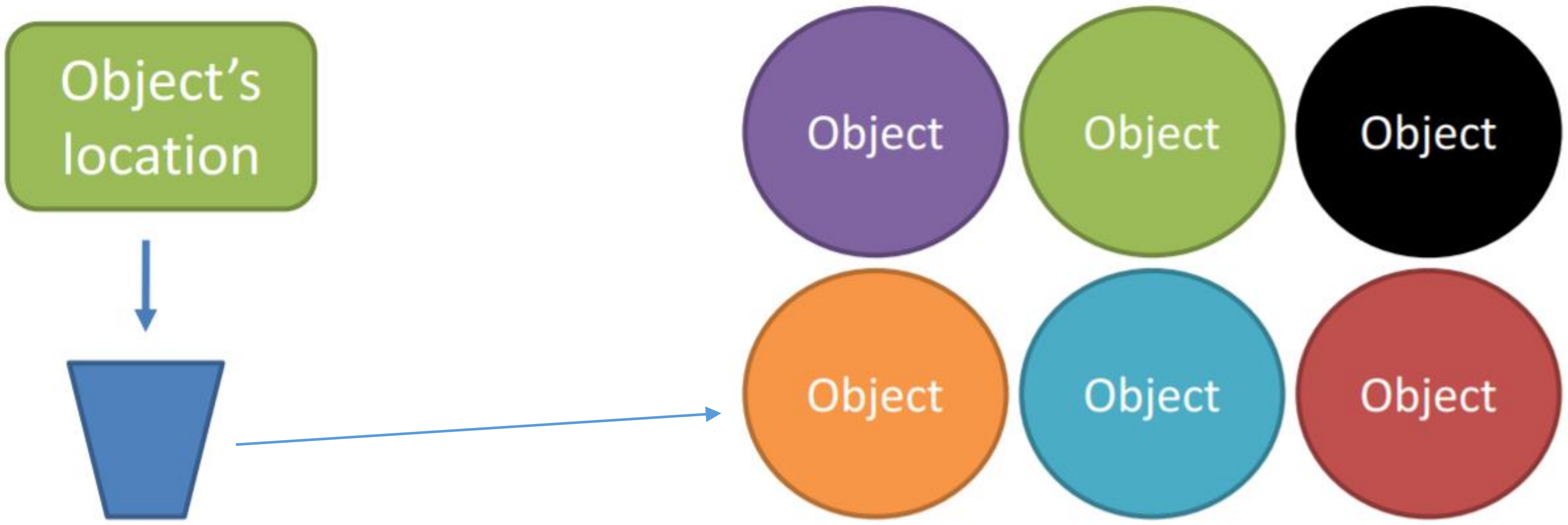
## How Java stores objects

- Objects are too big to fit in a variable
  - Stored somewhere else
  - Variable stores a number that locates the object



## How Java stores class instances

- Class instances (objects) are too big to fit in a variable
  - Stored somewhere else
  - Variable stores a number that locates the object



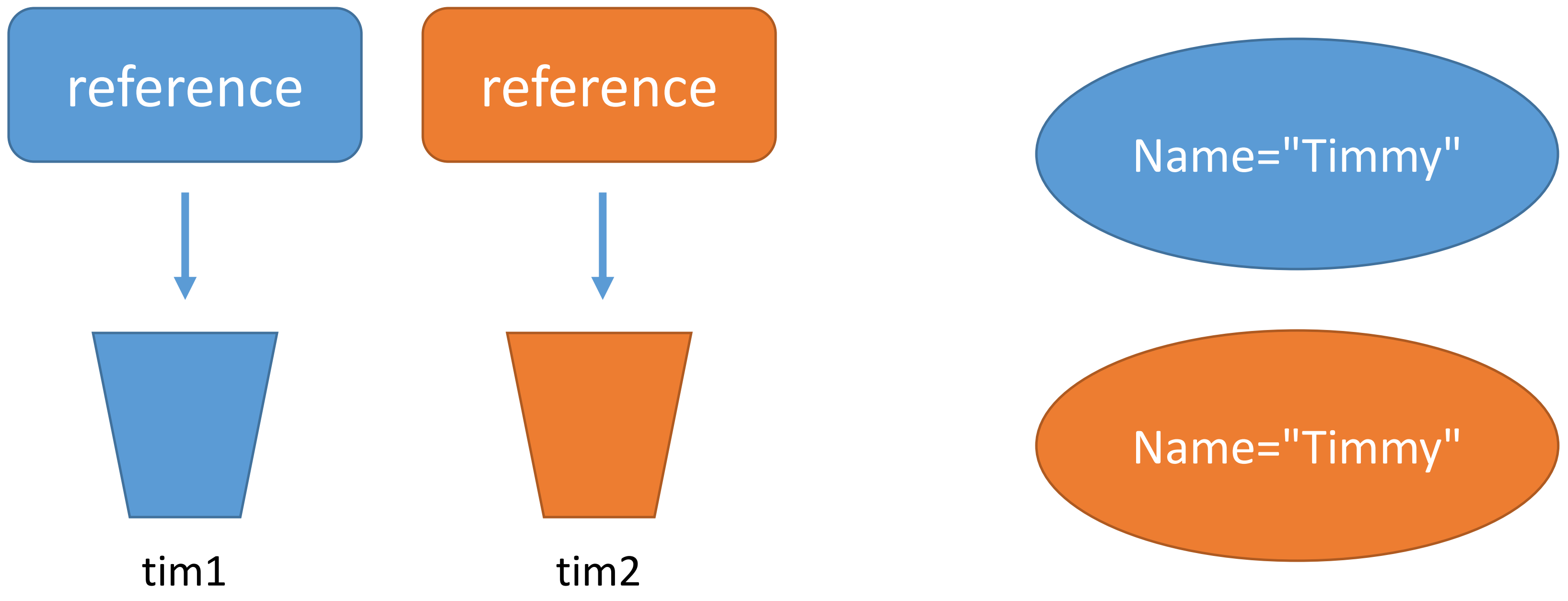
## References

- The object’s location is called a reference
  - All object variables are references, as opposed to primitive variables
- `==` only compares the references

```java
Baby tim1 = new Baby("Timmy");
Baby tim2 = new Baby("Timmy");
// true or false?
boolean b = (tim1 == tim2);
```

## References

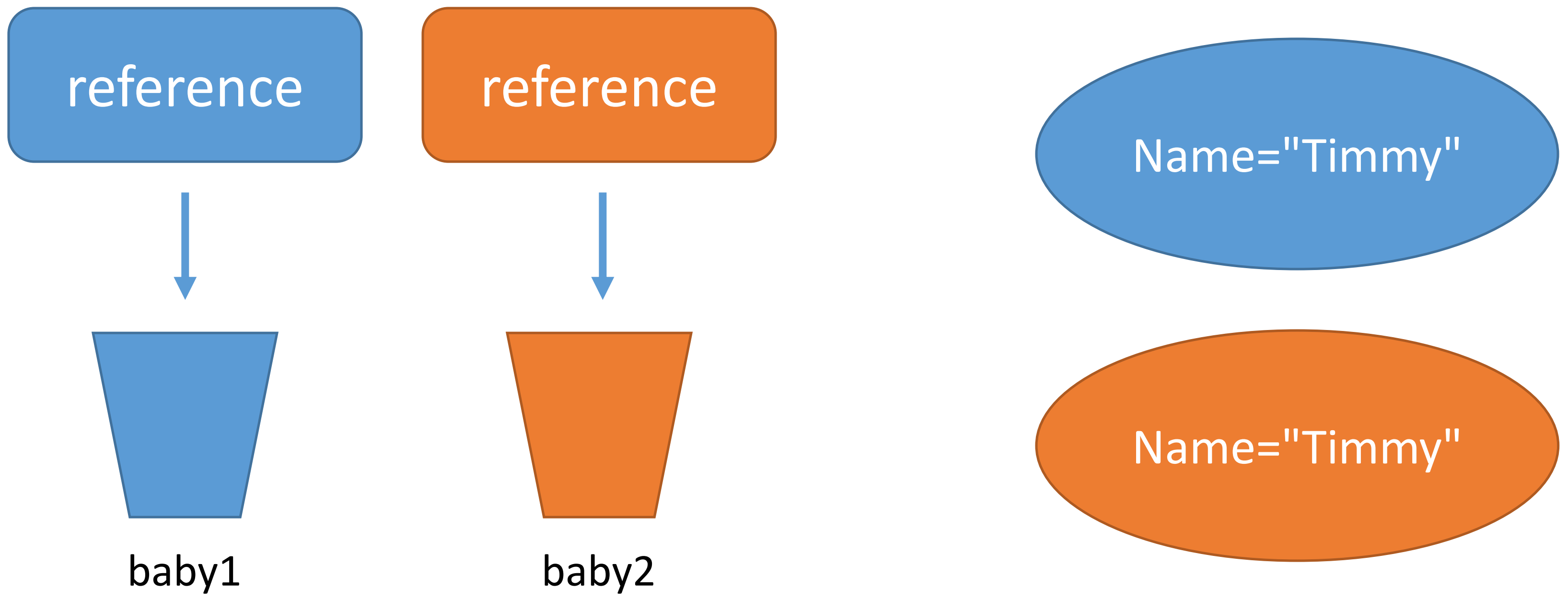
```
Baby tim1 = new Baby("Timmy");
Baby tim2 = new Baby("Timmy");
// true or false?
boolean b = (tim1 == tim2);
```



# References

- Using = updates the reference.

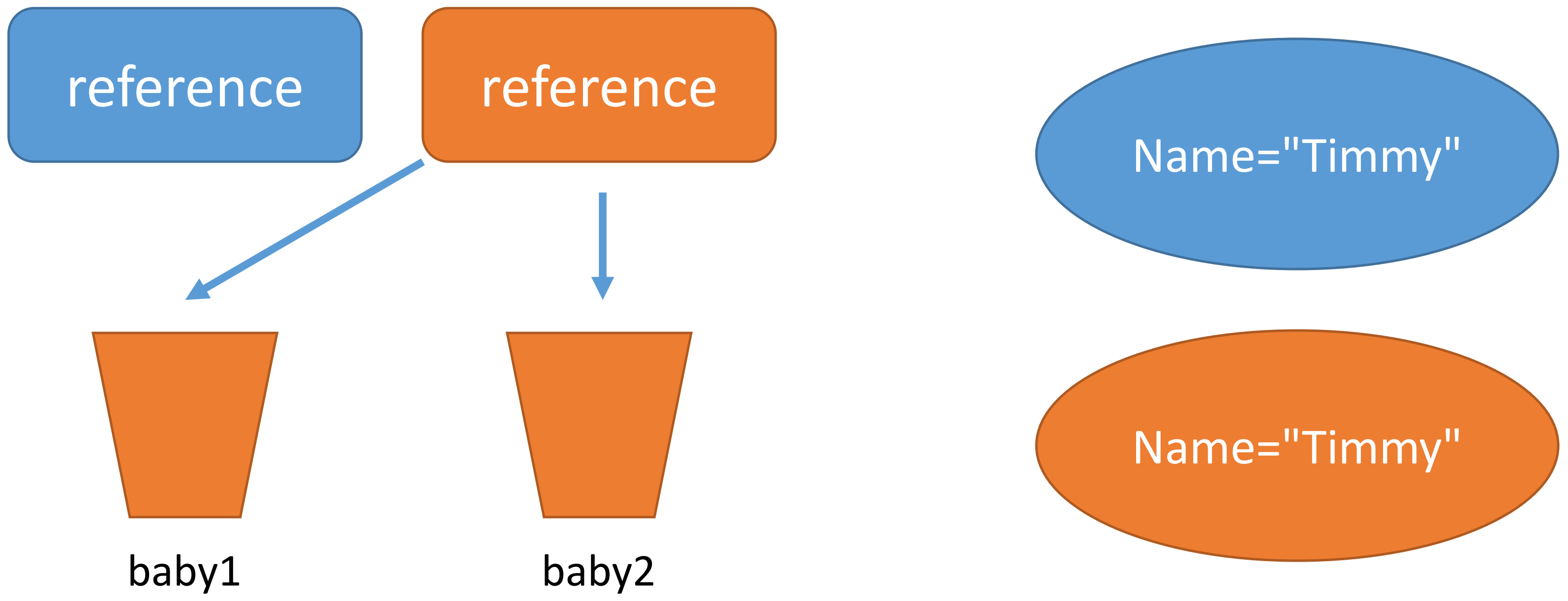
```
baby1 = baby2;
```



## References

- Using = updates the reference.

```
baby1 = baby2;
```



## static keyword

- Applies to fields and methods
- `static` means the field/method
    - Belongs to the static context
    - Is defined for the entire class
    - Is not unique to each instance

## static example

- Keep track of the number of babies that have been made

```java
public class Baby {
    // fields
    static int babiesMade = 0;

    // constructor(s)
    public Baby(String n, boolean m) {
        name = n;
        isMale = m;
        babiesMade++;
    }

    // methods
}
```

## BankAccount example

```java
public class BankAccount {
    double balance;
    int transactions;
    public BankAccount(double initial) {
        this.balance = initial;
        this.transactions = 1;
    }

    public void deposit(double amount) {
        balance += amount;
        transactions++;
    }

    public void withdraw(double amount) {
        balance -= amount;
        transactions++;
    }

    public void monthlyFee() {
        this.withdraw(10);
    }
}
```

# this keyword

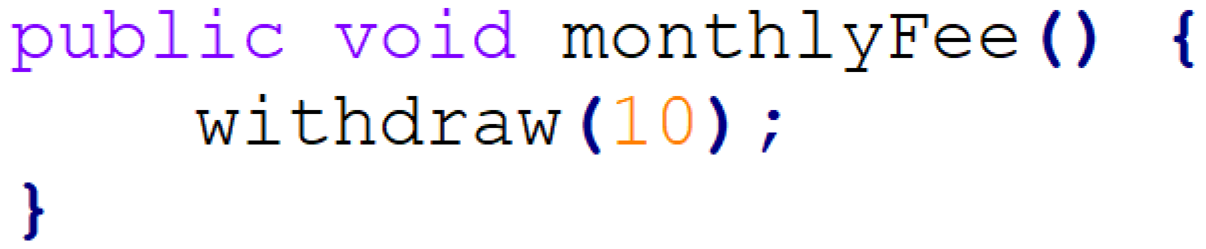
- You create a bank account:

```
BankAccount myAcc = new BankAccount(500);
```

- You withdraw $50:

```
myAcc.withdraw(50);
```

- Here, you specify the method to call and the instance to which the method belongs.



```
public void monthlyFee() {
    withdraw(10);
}
```

- What’s the instance? → The current instance

## this keyword

```
public void monthlyFee() {
    this.withdraw(10);
}
```

this keyword refers to the current instance.

## Packages

- All of the Java classes are organized into <u>packages</u>.
  - A <u>package</u> is a group of related classes.
- A class has a path that shows which package it belongs to:
  - `java.lang.String, java.lang.Math, java.util.Scanner…`
- Classes in `java.lang` package are available for use without importing. Others need <u>importing</u>.
  - `import java.util.Scanner;`
- All classes in a package can be imported using wildcard:
  - `import java.util.*;`

## Summary

- Compiler is a program translation program, operates in phases
- Virtual machine is an abstraction of computer hardware implemented in software
- Why Use OOP?
    - Addresses the challenge of code reuse.
    - Enhances code organization, maintainability, scalability.
    - Facilitates modular and reusable code.
- Basic OOP
    - Object models real-world entities with attributes and behaviors.
    - Class is the blueprint or template for objects. It defines static attributes, dynamic behaviors.
    - Creating Instances of a Class.
    - Dot （.） Operator