Introduction to Object-Oriented Programming

At the heart of Java's design philosophy is Object-Oriented Programming. OOP is a paradigm that uses "**objects**" — **entities that combine data and behavior** — to design applications and computer programs. It's a way of organizing code that helps developers manage and use data efficiently and securely. **The core concept of the object-oriented approach is to break complex problems into smaller objects.**

The **four fundamental principles of OOP** — **encapsulation**, **abstraction**, **inheritance**, and **polymorphism** — are not just abstract concepts but practical tools that Java offers to solve real-world problems in software design.

Encapsulation ensures that the internal representation of an object is hidden from the outside view.

Abstraction simplifies complex reality by modeling classes appropriate to the problem.

Inheritance allows one class to inherit the properties and methods of another. **Polymorphism** enables a single interface to represent different underlying forms (data types).

These principles are not unique to Java. However, Java's implementation of OOP principles is acclaimed for its clarity and consistency, making it an ideal language for those looking to master OOP.

The Java Environment: Setup and IDEs

To begin coding in Java, you need a Java Development Kit (JDK), which includes the Java Runtime Environment (JRE) and an interpreter/loader (Java). An Integrated Development Environment (IDE) like **Eclipse**, **IntelliJ IDEA**, or **NetBeans** can significantly simplify coding in Java. These IDEs provide a user-friendly interface for coding, debugging, and testing Java applications.

Java Syntax: The Building Blocks

Java syntax is the set of rules that defines how a Java program is written and interpreted. **The basics include**:

-**Variables**: Variables in Java are containers that hold data values during the execution of a program. Each variable must be declared with a data type, whether it's a **primitive type** like **int, float, double, or an object type**.
-**Data Types**: Java is **strongly typed**, meaning every variable and expression type is known at compile time. **Data types** in Java are categorized into **primitive types** (such as **int, char, double**) and **non-primitive** types (such as **String, Arrays, and Classes**).
-**Operators**: Java provides a rich set of operators to manipulate variables. These include **arithmetic operators** (+, -, *, /), **relational operators** (==, !=, >, <), and **logical operators** (&&, ||, !).
-**Control Structures**: Java's flow of execution is controlled by **conditional statements** (**if-else, switch-case**) and **loops** (**for, while, do-while**), allowing the program to make decisions and perform repetitions of tasks.

Java Functions and Methods

Methods in Java are **blocks** of code that perform a specific task. They're used to create reusable code and to **divide a complex problem into smaller**, manageable pieces. Understanding how to write and use methods is essential for effective Java programming.

Basics of Java Programming Language

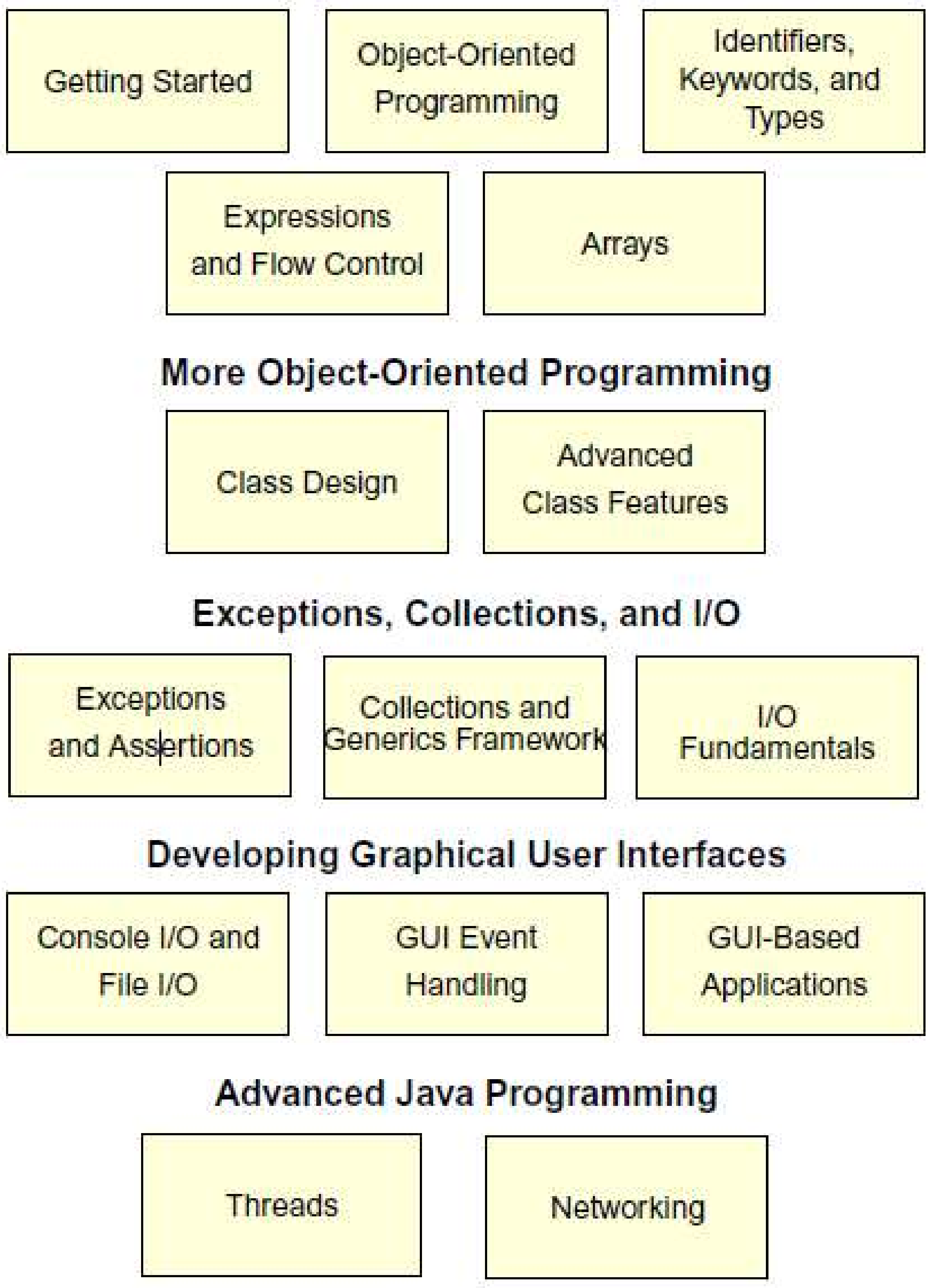

Getting Started
Object-Oriented Programming
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
More Object-Oriented Programming
Class Design
Advanced Class Features
Exceptions, Collections, and I/O
Exceptions and Assertions
Collections and Generics Framework
I/O Fundamentals
Developing Graphical User Interfaces
Console I/O and File I/O
GUI Event Handling
GUI-Based Applications
Advanced Java Programming
Threads
Networking

Fig 1: Java programming Basics

Java Runtime Environment

The Java application environment performs as follows:

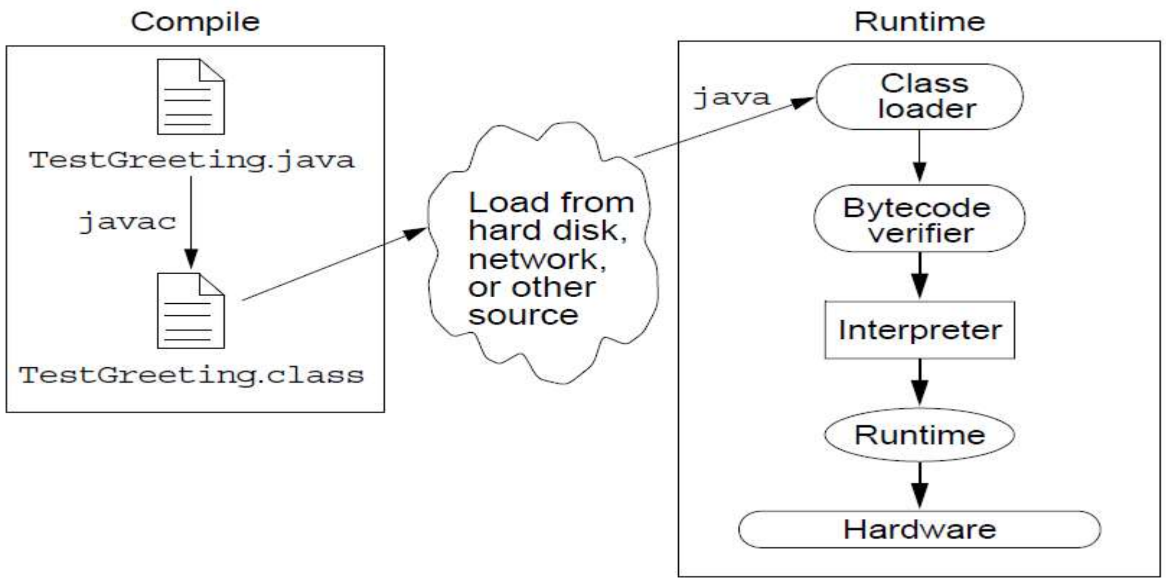


Fig 2: Java Runtime Environment (JRE)

In some Java technology runtime environments, apportion of the verified bytecode is compiled to native machine code and executed directly on the hardware platform. What is the difference to use that?

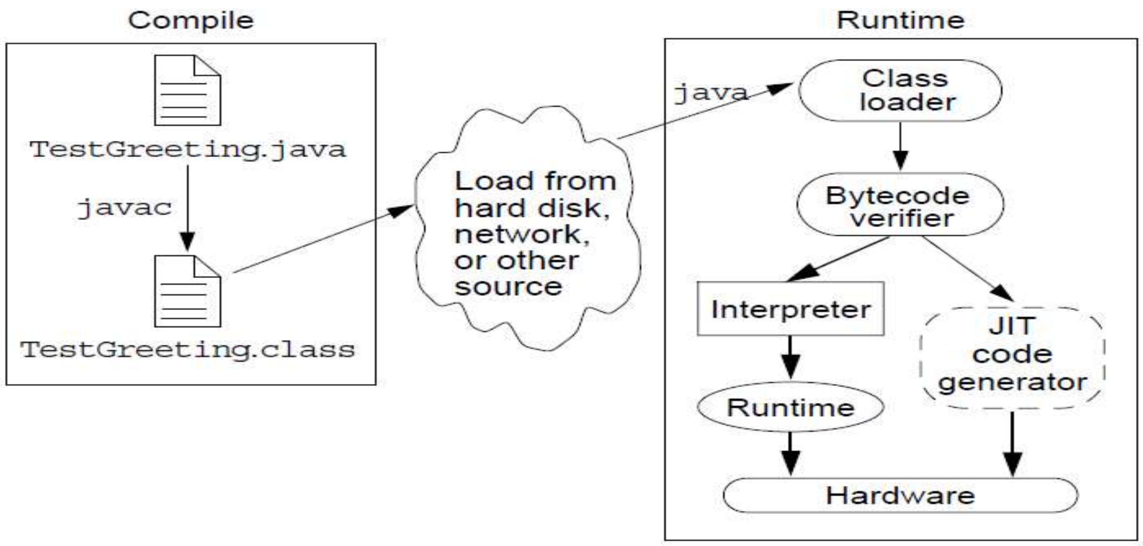


Fig 3: Java Runtime Environment (JRE) with Just in Time (JIT) compiler

Java Classes and Objects

Even though classes and objects belong to the realm of OOP, it's important to introduce them in the basics. In Java, everything revolves around classes and objects. **A class is a blueprint for objects**, **and an object is an instance of a class**. Grasping this relationship is pivotal for understanding Java's OOP nature.

Understanding Classes

A class in Java can be thought of as a blueprint or a template for creating objects. It defines a datatype by bundling data and methods that operate on the data into a single unit. Classes contain:

- **Fields:** Variables that hold the state of an object.
- **Methods:** Blocks of code that define the behavior of the object.

Think of a **class** as a blueprint for a **house**. It contains the **design details** but is not a house itself. Similarly, a **class defines the structure and capabilities of what its objects will be**, but it is not the object itself.

The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the instance variables that it contains and the methods that operate on them. Although very simple classes might contain only methods or only instance variables, most real-world classes contain both.

A **class is created** by using the **keyword class**. The general form of a **class** definition is shown here:

```
class classname {
// declare instance variables
type var1;
type var2;
// ...
type varN;
// declare methods
type method1(parameters) {
// body of method
}
type method2(parameters) {
// body of method
}
// ...
type methodN(parameters) {
// body of method
}
}
```

Defining a Class

To illustrate classes, we will develop a **class** that encapsulates information about **vehicles**, such as **cars**, **vans**, and **trucks**. This **class** is called **Vehicle**, and it will **store three items of information** about a vehicle: the **number of passengers** that it can carry, its **fuel capacity**, and its **average fuel consumption** (in **miles per gallon**). The **first version** of **Vehicle** is shown next. It **defines three instance variables**: **passengers**, **fuelcap**, and **mpg**. Notice that **Vehicle** does **not contain any methods**. Thus, it is currently a **data-only class**. (Subsequent sections will add methods to it.)

```
class Vehicle {
int passengers; // number of passengers
int fuelcap; // fuel capacity in gallons
int mpg; // fuel consumption in miles per gallon
}
```

A **class** definition **creates a new data type**. In this case, the new data type is called **Vehicle**. You will use this name to declare objects of type **Vehicle**. Remember that a **class declaration** is only a type of description; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Vehicle** to come into existence.

To **actually create** a **Vehicle object**, you will use a statement like the following:

Vehicle minivan = new Vehicle(); // create a Vehicle object called minivan

How Objects Are Created

In the preceding programs, the following line was used to **declare an object of type Vehicle**:

Vehicle minivan = new Vehicle( );

This **declaration performs two functions**.
First, it declares a variable called **minivan** of the class type **Vehicle**. This variable does not define an object. Instead, it is simply a variable that can **refer to an object**.
Second, the declaration creates a **physical copy** of the object and assigns to **minivan** a reference to that object. This is done by using the **new** operator.

The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in a variable. Thus, in Java, all class objects must be dynamically allocated.

The **two steps combined in the preceding statement can be rewritten** like this to show each step individually:

Vehicle minivan; // declare reference to object
minivan = new Vehicle (); // allocate a Vehicle object

The first line declares **minivan** as a **reference** to an object of type **Vehicle**. Thus, **minivan** is a variable that can refer to an object, but it is not an object, itself. At this point, **minivan** contains the value **null**, which means that it does not refer to an object. The next line creates a new **Vehicle** object and assigns a reference to it to **minivan**. Now, **minivan is linked with an object**.

After this statement executes, **minivan** will be an instance of **Vehicle**. Each time you create an instance of a class; you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Vehicle** object will contain its own copies of the instance variables **passengers**, **fuelcap**, and **mpg**.

To **access these variables**, you will use the **dot (.)** operator. **The dot operator links the name of an object with the name of a member**. The general form of the dot operator is shown here:

object.member

Thus, the object is specified on the left, and the member is put on the right. For example, to assign the **fuelcap** variable of **minivan** the value 16, use the following statement:

minivan.fuelcap = 16;

In general, you can use the dot operator to access both instance variables and methods.

Here is a **complete program** that uses the **Vehicle** class:

```
/* A program that uses the Vehicle class.
Call this file VehicleDemo.java
*/
class Vehicle {
int passengers; // number of passengers
int fuelcap; // fuel capacity in gallons
int mpg; // fuel consumption in miles per gallon
}

// This class declares an object of type Vehicle.
class VehicleDemo {
public static void main(String args[]) {
Vehicle minivan = new Vehicle();
int range;
// assign values to fields in minivan
minivan.passengers = 7;
minivan.fuelcap = 16;
minivan.mpg = 21;
// compute the range assuming a full tank of gas

range = minivan.fuelcap * minivan.mpg;

System.out.println("Minivan can carry " + minivan.passengers +" with a range of "
+ range);
}
}
```

The following output is displayed:

```
Minivan can carry 7 with a range of 336
```

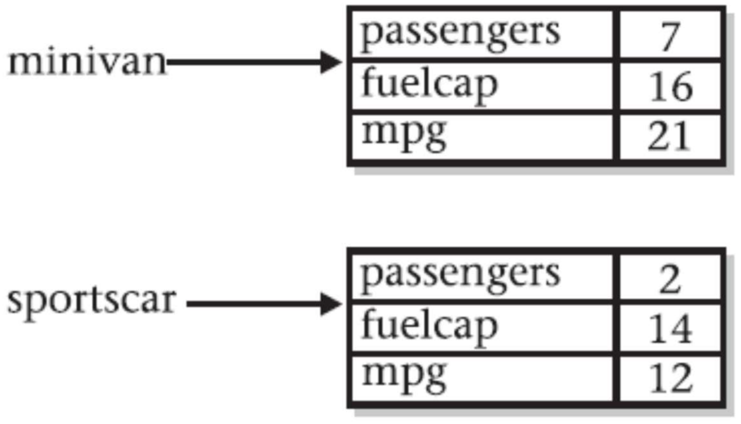
Before moving on, let’s review a fundamental principle: each object has its own copies of the instance variables defined by its class. Thus, the contents of the variables in one object can differ from the contents of the variables in another. There is no connection between the two objects except for the fact that they are both objects of the same type. For example, if you have **two Vehicle objects**, each has its own copy of **passengers**, **fuelcap**, and **mpg**, and the contents of these can differ between the two objects. The following program demonstrates this fact. (Notice that the class with **main( )** is now called **TwoVehicles**.)

```
// This program creates two Vehicle objects.
class Vehicle {
int passengers; // number of passengers
int fuelcap; // fuel capacity in gallons
int mpg; // fuel consumption in miles per gallon
}

// This class declares an object of type Vehicle.
class TwoVehicles {
public static void main(String args[]) {

Vehicle minivan = new Vehicle();
Vehicle sportscar = new Vehicle();
int range1, range2;
// assign values to fields in minivan
minivan.passengers = 7;
minivan.fuelcap = 16;
minivan.mpg = 21;

// assign values to fields in sportscar
sportscar.passengers = 2;
sportscar.fuelcap = 14;
sportscar.mpg = 12;


// compute the ranges assuming a full tank of gas
range1 = minivan.fuelcap * minivan.mpg;
range2 = sportscar.fuelcap * sportscar.mpg;
System.out.println("Minivan can carry " + minivan.passengers +" with a range of "
+ range1);
System.out.println("Sportscar can carry " + sportscar.passengers +" with a range of
" + range2);
}
}
```

The output produced by this program is shown here:

```
Minivan can carry 7 with a range of 336
Sportscar can carry 2 with a range of 168
```

As you can see, **minivan**'s data is completely separate from the data contained in **sportscar**. The following illustration depicts this situation.



Reference Variables and Assignment

In an assignment operation, object reference variables act differently than do variables of a primitive type, such as int. When you assign one primitive-type variable to another, the situation is straightforward. The variable on the left receives a **copy of the value** of the variable on the right. When you assign an object reference variable to another, the situation is a bit more complicated because you are changing the object that the reference variable refers to.

The effect of this difference can cause some counterintuitive results. For example, consider the following fragment:

```
Vehicle car1 = new Vehicle();
Vehicle car2 = car1;
```

At first glance, it is easy to think that car1 and car2 refer to different objects, but this is not the case. Instead, car1 and car2 will both refer to the same object. The assignment of car1 to car2 simply makes car2 refer to the same object as does car1. Thus, the object can be acted upon by either car1 or car2. For example, after the assignment

```
car1.mpg = 26;
```

executes, both of these println( ) statements

System.out.println(car1.mpg);
System.out.println(car2.mpg);

display the same value: 26.

Although car1 and car2 both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to car2 simply changes the object to which car2 refers. For example:

```
Vehicle car1 = new Vehicle();
Vehicle car2 = car1;
Vehicle car3 = new Vehicle();
car2 = car3; // now car2 and car3 refer to the same object.
```

After this sequence executes, car2 refers to the same object as car3. The object referred to by car1 is unchanged.

Code Example: Defining a Class

```
public class Car {
    // Fields
    String make;
    String model;
    int year;

    // Method
    void displayInfo() {
        System.out.println("Car Make: " + make + ", Model: " + model + ",
Year: " + year);
    }
}
```

Code Example: Creating an Object

```
public class Main {
    public static void main(String[] args) {
        // Creating an object of Car
        Car myCar = new Car();

        // Assigning values to fields
        myCar.make = "Toyota";
        myCar.model = "Corolla";
        myCar.year = 2021;

        // Calling method
        myCar.displayInfo();}
}
```