Object Oriented Programming through JAVA™

P Radha Krishna

Universities Press

1 Object Oriented Programming

Computer programming languages have undergone a dramatic change from being procedure oriented to object oriented in the last decade. One reason for such a shift in approaches is the increased usage of software in dealing with problems having a high level of complexity. This and the need to reuse parts of software in different applications have led to the popularity of object oriented programming (OOP). This chapter covers the basics of OOP.

1.1 Introduction to OOP

In early days, programs were collections of procedures acting on data. A procedure is defined as a collection of instructions executed in sequential order. Data were independent of the procedures and programmers have to keep track of functions and the way they modify data. Structured programming is a simpler way to tackle this situation.

In *structured programming*, a complex program is broken into sets of smaller, understandable tasks; however, the drawbacks of structured programming soon became clear. These were, first, that data and task independency were too hard to maintain. Second, programmers were reinventing new solutions to old problems. To address these problems, *object oriented programming* came into existence. This provides a technique for managing huge complexity, achieving reuse of software components and coupling data with the tasks that manipulate data.

Object oriented programming, as a concept, was introduced by Xerox Corporation in the early 1970s. The first object oriented language was Smalltalk. The concept of object orientation in this language did not prove successful for about 10 years because of the limitations of early computers. Some languages that successfully implemented object oriented programming are C++, Java and Eiffel. Object oriented programming is thought to be a relatively new concept in computer science. Contrary to this, all the major concepts such as objects, classes, inheritance herarchies were developed by researchers of Norwegian Computing Center as part of development of programming language Simula in the 1960s. Yet the importance of these concepts was slowly recognized by the developers of Simula.

Object oriented programming deals with things called 'objects'. Objects are just extremely functional ways of organizing information. Objects in an OOP language are combinations of *code* and *data* that are treated as a single unit. Each object has a unique name and all references to that object are made using that name.

Allowing the programmer to write applications that interact with what appears to be physical objects makes the task of writing computer programs easier.

When an application is being created using OOP, the programmer needs only to remember a small set of flow-control commands and how to interact with the objects that his/her application uses.

Object oriented programming is built on the concept of reusing code through the development and maintenance of *object libraries*. These objects are available for building and maintaining other applications. Usually, a library contains the code for various functions that are commonly required during the program development. These libraries can be language-specified or user-defined. Unlike traditional libraries, OOP languages provide object libraries which contain code along with the data in the form of objects, in addition to the names and entry points of the code located within. Objects of object library are available for building and maintaining other applications.

1.2 Objects and Classes

Objects and classes are the building blocks of OOP. To understand OOP, first we have to know what objects and classes are.

An *object* is a programming entity in OOP. It has data components representing the present state of the object as well as functions (also called methods). Data components are termed attributes in the context of OOP languages. An area of memory is kept aside to store values for the attributes when an object is created. Functions describe the behaviour of the object and allow the other programming entities to interact with the object.

> An object contains attributes (variables) and methods (member functions). Attributes are the data associated with the object and the methods are the functions and code which operate on the data.

A *class* is a model to define objects. In other words, an object is said to be an instance of a class.

An object may be real or abstract. That is, objects can model concrete things such as people, and data entry forms as well as abstractions such as numbers, equations or geometrical concepts.

Consider an example of a class—car. The class car describes all car objects by specifying the data components and the methods that each can have. Various objects defined in the class car could be Maruti 800, Zen, Santro, Accent and so on. The class car has various attributes such as the seating capacity, maximum speed, engine capacity, parking light, current speed and current position. Typical functions of the car class are to find acceleration and set parking lights on. See Figures 1.1(a) and (b) for an illustration.

Real-world objects have two characteristics: *state* and *behaviour*. For example, consider an object *ball*. A ball has particular characteristics such as its diameter, colour and elasticity. Formally, we say that these characteristics contribute to the ball's *state* of being. We also describe a ball by what it does, such as the fact that it can be thrown, bounced or rolled. These activities define the ball's *behaviour*. Similarly the object *car* has state (colour, wheels, current gear, number of gears and so on) and behaviour (braking, changing gears, accelerating, decelerating and so on).

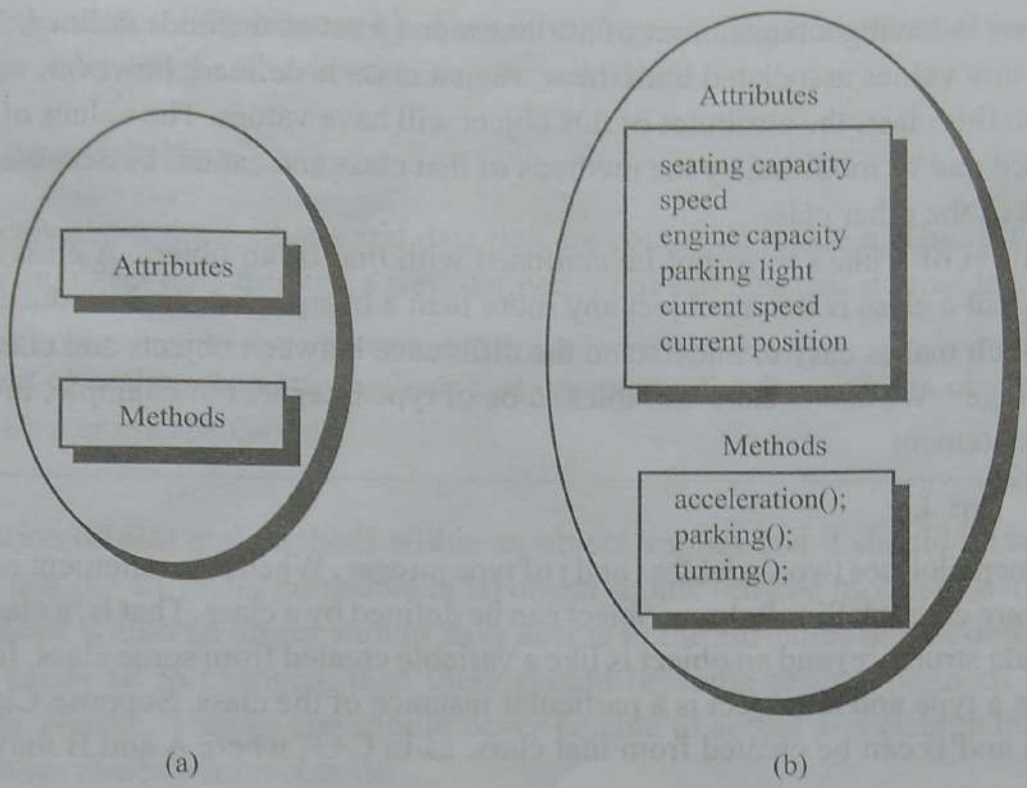

Fig. 1.1 (a) An object; (b) example of object car.

Software objects are modelled after real-world objects in that they too have state and behaviour. Attributes define the state of the object and methods define the behaviour of the object. That is, a software object maintains its state in one or more *attributes*. Attributes can have different values at different instances of time; however, values of a particular attribute must have the same data type.

An attribute/variable is an item of data named by an identifier, and a software object implements its behaviour with *methods*. A method is a function/sub-routine associated with an object.

The state of an object and its behaviours work together. For example, how high a ball bounces depends on its elasticity. An object's behaviour often modifies its state. For example, when a ball is rolled, its position changes.

An object must also have a *unique identity* that distinguishes it from all other objects. Two objects can have the same data types of attributes with the same names, and those attributes can have the same values. Therefore two objects could have the same state. There must be a way of distinguishing the two objects and that is done by using an object's name. In a program, to define an object we need a class.

A class is a model or pattern from which an object is created. In other words, a class is a special module that defines an object.

A class defines the data types of attribute that will be held in an object, and defines code for the methods. After a class was defined, an object can be created from it. The process of creating an object from a class is called *instantiation*. Every object is an instance of a particular class.

Defining a class is having a name, a set of attributes and a set of methods defined. The attributes will not have any values associated with them when a class is defined; however, when an object is created from this class, the attributes of this object will have values. The values of the attributes can be accessed and/or modified by the methods of that class and cannot be accessed or modified by the objects of the other class.

The definition of a class must not be confused with that of an object. A class is the outline for an object. But a class is not an object any more than a blueprintis. We introduce the notion of data types, which makes easy to understand the difference between objects and classes. Consider a data type integer. We can declare variables to be of type integer. For example, in C, we can do this with the statement

```
int i, j;
```

This statement defines two variables i and j of type integer. When this statement is executed, the two variables are created. Similarly, an object can be defined by a class. That is, a class is similar to a data type (data structure) and an object is like a variable created from some class. In other words, a class defines a type and an object is a particular instance of the class. Suppose Circle is a class, the objects A and B can be created from that class, as in C++, where A and B may be circles of different diameters.

```
Circle A, B;
```

An object can be viewed from two perspectives. The first is useful during design and implementation of the object. We have to define the attributes that will be held in the object and write the program code for the methods that make the object useful. The second comes in when making use of the services of an object that was already created. When designing a solution to a large problem, we should know only the services that an object provides rather than the details of how the services are accomplished. That is, from the second point of view, an object is a black box. It is a part of a system whose inner workings and structure are currently hidden. The rest of the system only interacts with the object through a well-defined set of services (called messages) that it provides.

In OOP, objects communicate with one another by sending and receiving information using messages. For an object, a message is a request for execution of a method. When a request is made to an object, it invokes the specified method in the receiving object, then the receiving object responds by generating the desired result. Message passing involves specifying the name of the object, the name of the method and any information that needs to be sent.

1.3 Characteristics of OOP

At the heart of OOP are three main characteristics. These are the features from which the advantages of using OOP are born.

- Encapsulation
- Inheritance
- Polymorphism

These characteristics differentiate object oriented programming from the traditional procedural programming model. They are now examined in more detail.

1.3.1 Encapsulation

An object encapsulates the methods and data that are contained inside it. The rest of the system interacts with an object only through a well-defined set of services that it provides.

> The concept of sealing data and methods that operate on the data into an object (a single unit) is known as encapsulation.

Encapsulation of data and methods within an object implies that it should be self-governing, which means that the attributes contained in an object should only be modified within the object. Only the methods within an object should have access to the attributes in that object. If the state of an object needs be to changed, then there should be some service methods available that accomplish the change. That is, one should never be able to access and change the values of an object's attributes directly from outside.

For example, embedding the attributes and methods defined for car in a single class car is referred to as encapsulation (see Figure 1.1(b)). The methods described in the car class can only modify/access the attributes defined in its class. Suppose, Zen is an object of car class, to set the parking light of Zen, the attribute parking_light can be set using the method set_parking_light_on; however, other objects of the same class such as Maruti 800 or objects of any other class cannot access/modify the value of parking_light for Zen object.

Encapsulation helps in building better, structured and more readable programs.

1.3.1.1 Abstraction via encapsulation

Encapsulation facilitates abstraction. A software object is an abstract entity, in that we view it from outside without concerning ourselves with the details of how it works on the inside. For example, we do not need to understand the internal workings of the gears of a car to use it, we only need to know how to make it work. Because of this the source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system.

Since, an object is a self-contained entity in a program, it can be referenced at different parts of the program wherever it is required. This feature enhances the *modularity* of the program. That is, the program consists of several modules, and if there are problems with any module, they can be addressed independently. Also, these modules can interact with each other without having to know the internal implementation of the other. Dividing a program into modules is called modularity; not having to know the internal implementation of a module is called abstraction. Hence, the feature of 'abstraction' facilitates a programmer to do 'modular' programming.

1.3.1.2 Data hiding

Encapsulation provides the hiding of data/information. It prevents users from seeing the internal workings of an object. The main reason for doing this is to protect data that should not be manipulated by the user. This makes the code more reliable and reusable.

An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting other objects that depend on it.

Every class can hide some of its parts. For example, if a part of code is called only by methods of that class, it can be made a private method, that is, invisible from outside that class.

Every element (field or method) of one class can have one of the three levels of visibility: public, private and protected.

- Public elements are completely visible from outside the class. They form the interface with the outside world.
- Private elements are visible only to methods of the class itself.
- Protected elements are something between public and private. To the outside world, they act as private elements, but they are completely visible to inherited child classes (these are discussed later)—they have public behaviour for these classes.

Some benefits of encapsulation are the following:

- Improves program reliability
- Reduces maintenance
- Lends reusability
- Facilitates information hiding
- Improves modularity
- Makes coding easier (This allows for inefficient codes to be optimized later.)

1.3.2 Inheritance

Inheritance is the feature of OOP by which the functionality of a class, called the parent or base class, can be extended by creating another class, called the child or derived class, which inherits some or all of the features of the parent class. The derived class has its own methods apart from the methods of base class and thus extending the functionality. This is the key for code-reusability.

Inheritance is the process of creating a new class from previously defined classes.

When a derived class inherits from the base class, it will contain all the attributes and methods of the base class but adds new characteristics of its own. Any changes that are made to base classes are propagated to all derived classes that inherit from them, unless explicitly overridden. Changes to a base could fix or break an application.

Consider the example of the car class. Suppose we want to create a new class called Maruti. We know that a Maruti is a car so we can derive this new class from our base car class. Some of the data for this new class may be size and mileage and a function would be engine. But since this is a derived class, we can also have access to the base class functions, accelerations and deceleration, plus the data of the base class type, colour and wheels.

Thus, inheritance can reduce the amount of coding by letting any derived class use the base class's functionality when it needs to, thus simplifying implementation of similar classes and structures. In addition, the derived class is compatible with the base class, and can be used in place of the base class. This allows a system to override any of the base class functionality it wants and use the new objects where the base class would be used, thus adapting it to its own needs.

Derived classes may be formed in many ways. They can be derived from a single base class, or they can be formed from different parent classes.

When a class is derived from only one base class the process is called *single inheritance*. When a class is derived from more than one base class the process is called *multiple inheritance*. Multiple inheritance is used to combine two or more existing classes.

Inheritance is often shown diagrammatically in terms of an *inheritance hierarchy* or *inheritance tree* as shown in Figure. 1.2. Note that the arrowhead points towards the base class to indicate that the child class is derived from it. In single inheritance the inheritance graph is a tree, whereas in multiple inheritance the inheritance graph is a directed acyclic graph (DAG).

Figure 1.3 is an example of a multiple-inheritance graph.

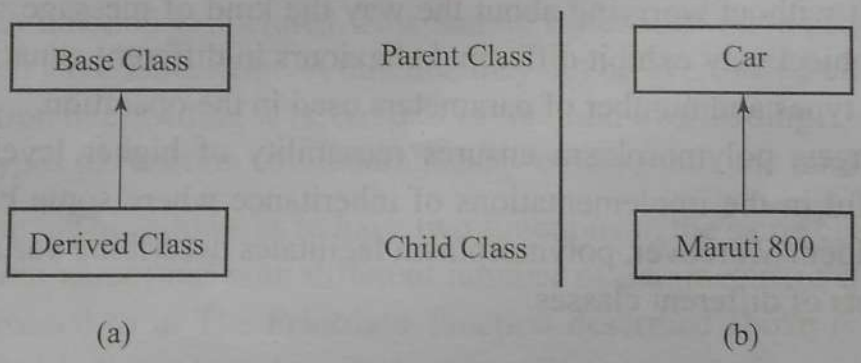

Fig. 1.2 (a) Inheritance tree; (b) an example of single inheritance.

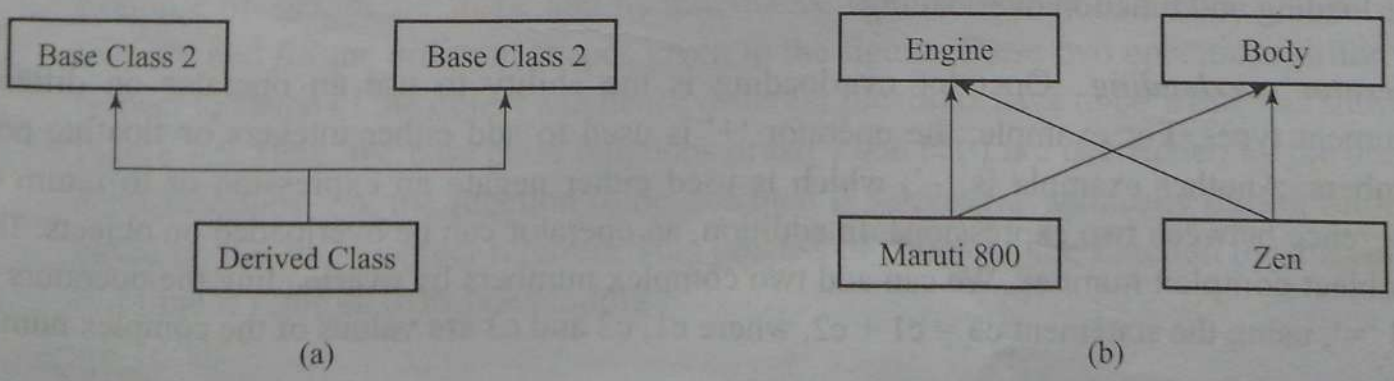

Fig. 1.3 (a) Multiple inheritance graph; (b) an example of multiple inheritance.

The advantages of using inheritance are the following:

- Increased productivity
- Reduced maintenance
- Standardization

The major drawback of inheritance is added overhead. This is explained below:

- An user must know changes in the methods of base class before using the derived class objects.
- Though the derived class objects may not required all the functionalities of a base class, the derived class implements all the methods of base class. This increases the amount of memory required for execution of the program.
- Since common functionalities exist in the base class, there is an execution overhead for derived class due to the implementation of base class methods when compared to having the functionalities in a single class itself. In the case of a chain of inheritances, the execution overhead may increase further.

1.3.3 Polymorphism

Polymorphism allows an entity (for example, an attribute, function or object) to take a variety of representations. It refers to the ability of an object to respond differently to the same message.

The advantage of this property is that the behaviour of an object varies depending on the message passed to it without worrying about the way the kind of message was passed. That is, an operation on an object may exhibit different behaviours in different situations. The behaviour depends on the data types and number of parameters used in the operation.

For OOP languages, polymorphism ensures reusability of higher level abstractions. This feature is also helpful in the implementations of inheritance where some base class behaviour needed to be overridden. Moreover, polymorphism facilitates describing variables that may refer, at run-time, to objects of different classes.

1.3.3.1 Overloading

Overloading is one kind of polymorphism. In OOP, there are two types of overloading: operator overloading and function overloading.

Operator overloading Operator overloading is the ability to use an operator on different argument types. For example, the operator '+' is used to add either integers or floating point numbers. Another example is '–', which is used either negate an expression or to return the difference between two expressions. In addition, an operator can be overloaded on objects. Take an object complex number. We can add two complex numbers by overloading the operators '+' and '=', using the statement c3 = c1 + c2, where c1, c2 and c3 are values of the complex number type.

Another example of operator overloading is that an addition sign (+) can be used to specify addition of two numbers (say, 200 + 6 = 206) or concatenation of two strings (say '200' + '6' = '2006').

Java does not support user-defined operator overloading.

Function overloading Function overloading is the ability to access different implementations of functions using the same name.

Consider an example that displays the current date. Usually, the date is represented in two forms:

- As a string: December 20, 2005
- As a triplet: (day, month, year): 20, 12, 2005

It is easier for a programmer if same function name can perform one of the above tasks depending on the user requirements. This can be achieved using function overloading by writing two different functions (methods) bearing same name as given below:

```
Printdate (Sting Str) {.....}
Printdate(int dd, int mm, int yyyy) {...}
```

Here, the first Printdate function takes one string parameter as argument and the second function takes three integer parameters as arguments. Based on the data types of the arguments passed in a message, the corresponding function is invoked. Thus, two or more functions may have the same name, so long as the parameter types or number of parameters are sufficiently different enough to distinguish which function is intended. This feature is not allowed in procedural languages, as two functions cannot have same name. A function may not be overloaded on the basis of its return type. In Java, function overloading is referred to as 'method overloading'.

There are two types of function overloading: early binding and late binding.

- *Early Binding* This allows us to have two functions in the same class that have the same name (function name) but with different number of parameters or different data types of parameters passed to it. The Printdate function described above is an example of early binding. Early binding takes place during compile time.
- *Late Binding* This allows us to have functions with same name in base and derived classes. But the derived class function will override the base class function. Consider the example of geometric shape and its inheritance tree, as shown in Figure 1.4. We want to *draw* and *fill* the different shapes given in the figure. These two operations differ with respect to shape of the object and hence different functions have to be written as shown in Figure 1.5. Here, the base class functions draw( ) and fill( ) are overridden by the derived class functions. So, the function to be executed is selected dynamically during run-time, based on the shape object referred. This process of choosing the function dynamically at run-time is referred to as late binding.

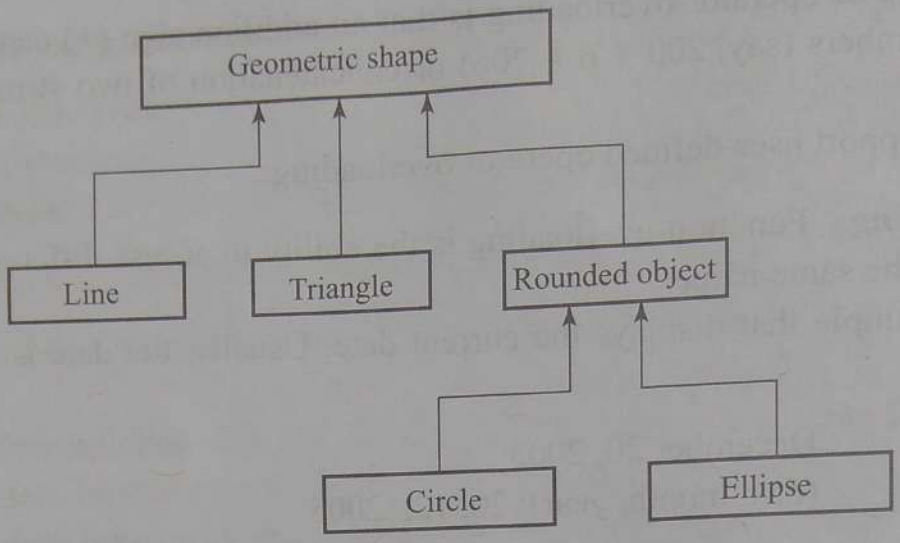

Fig. 1.4 An inheritance tree of geometric shape.

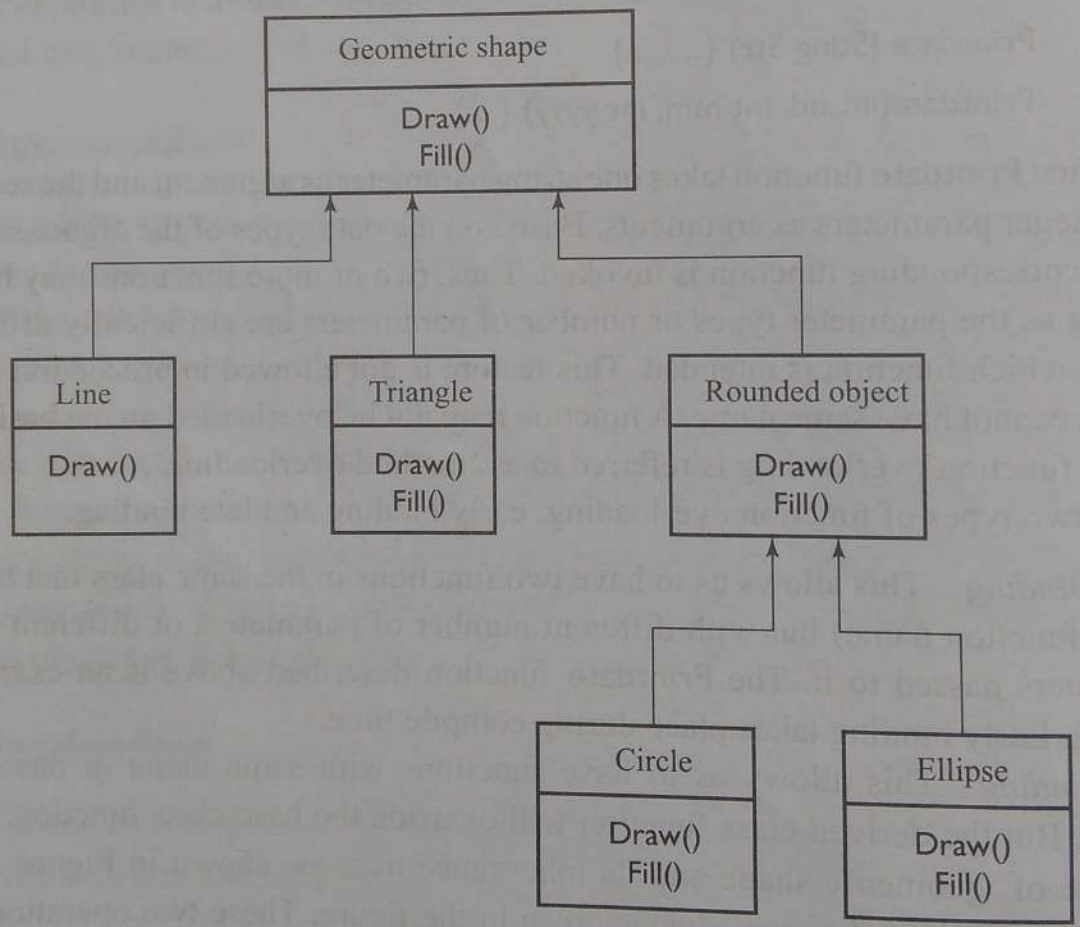

Fig. 1.5 An inheritance tree of geometric shape with functions.

The main benefit of polymorphism is it improves flexibility.

Overloading is not unique to OOP, it exists in procedure oriented programming languages also, such as C, to a limited extent. In C, we can use the operator '+' for two integers as well as for floating point numbers. Similarly, the functions printf and scanf can have different number of arguments with different types.

1.4 Difference between OOP and Procedure Oriented Programming

Now that we know the basic concepts in OOP, we are in a position to compare it with classical procedure oriented programming.

In classical procedural (or structural) programming, there are techniques to write a code that handles some general task The most important technique of that kind are functions (procedures or sub-routines). The large program is partitioned into functions that each perform a specific task. The main program calls these functions, which in turn may call other functions. The major concern of this programming is with respect to the data.

Data are passed around from one non-member function to another and are available everywhere throughout the large program. This allows the user to access the data directly, alter it from several points in the program and inadvertently introduce mistakes. If one programmer changes the representation of the data in the computer's memory by rearranging fields within a structure (record), other functions in the program may need to be rewritten and must then be retested to account for this change. That means the data are not well protected in procedure oriented programming. Though local variables in a function can be accessed within the function, they are not useful if they must be accessed by different functions.

In contrast, OOP safely encapsulates collections of functions (called methods) with the data they manipulate. In other words, the data can only be manipulated by its own functions. This feature protects the data from rest of the program.

Though the concepts of object oriented programming are different from procedural oriented programming, structured programming constructs are still used in OOP especially in the coding of methods.

1.5 Summary

As stated earlier, the principles of encapsulation, inheritance and polymorphism form strong structures and define the major benefits and drawbacks of OOP. Let us take a look at these aspects.

1.5.1 Benefits of OOP

- *Code reusability* New objects can be derived from old objects, allowing for improvement and refinement of the code at each stage and also preserving parts of the code for in other programs. This is used to develop many class libraries using class codes that have already been written, for example, Microsoft Foundation Classes (MFC).
- *Code Modularity* Everything in OOP is an object; these objects can be interchanged or removed to meet the users' needs.
- *Easier maintenance* Inheritance usually reduces maintenance because of the domino effect it has on derived classes when a change is made in a base class.

- *Design stability* Once a stable base class has been developed, the new classes that are derived may have fewer less errors and bugs.
- *Improved communication between developers and users* Objects can be broken down into real life entities, hence it is easier it communicate ideas.
- *Seamless transition from design to implementation* This is mainly because communications are improved.

1.5.2 Drawbacks of OOP

- *Execution overhead* Once a derived class is initiated all the data and functions from the base classes are carried along with it. Some of these, or even most of it, may not be used. Derived classes can be very complex because of inheritnce and polymorphism.
- Abstraction may lead to performance degradation.

1.5.3 Challenges of OOP

- *High learning curve* OOP is different from traditional programming. We must develop strong base classes and understand the functionality of class libraries before we can take advantages from it.
- *Difficulty in establishing base classes* A good foundation must be created before you derive other classes. Base classes have to be generic enough to meet the needs of your application and any future upgrades to that application.

Objective type questions

1. _____ are combination of *code* and *data* that are treated as a single unit.
2. In objects, the behaviour of the real-world entities is represented by _____.
3. In objects, _____ define the state of an object.
4. _____ is a model or blue-print from which an object is created.
5. Binding the data to methods of an object is called _____..
6. _____ provides the mechanism of 'One name but different implementations'.
7. A _____ method is used to assign values to private instance variables of a class.
8. Inheritance enables _____ which saves time in development and encourages using previously proven and high quality software components.

2

Introduction to Java Programming

Java is an object oriented programming language that also provides run-time environment. This chapter gives a brief introduction to Java and what makes it more powerful than other object oriented programming languages. It also gives an overview of arrays, methods and structures in Java and a summary on Java applications and applets, so as to familiarize the reader with Java programming. It is an introduction to Java programming concepts meant for beginners, while experienced programmers would find it helpful in revising their skills.

2.1 Introduction

A computer program is usually written using a programming language (such as C, C++ or Java). It can be defined as follows:

> A program is a series of instructions (usually long and complex) that are carried out (or executed, or run) by a computer.

These instructions define what the program does, whether it is a word processor, spreadsheet, game or any other kind of program.

Programming can fall into the category of application programming or systems programming. *Application programming* is a programming process that is aimed at developing an application (such as banking applications, railway reservation application and so on) to be used by people (end users). *Systems programming*, on the other hand, is the activity of writing, amending or extending the *operating system*, which is a program that runs on the computer hardware in order to allow other applications to be run.

2.1.1 Overview of Java technology

Java technology is simultaneously a programming language and a platform.

2.1.1.1 Java as a programming language

Java is a general-purpose high-level object oriented programming language that provides the following characteristics: (or) Key Concepts

- Simple
- Interpreted
- Platform independent
- Dynamic
- Distributed
- Secure
- Object oriented
- Architecture-neutral
- Multithread
- High-performance
- Robust

Usually, a programming language requires either a compiler or an interpreter to translate a code written in high-level programming language (source code) into its machine language (object code) equivalent. But, to run a Java program, Java uses a combination of compiler and interpreter. The Java compiler translates the Java program into an intermediate code called Java byte code. A Java interpreter is used to run the compiled Java byte code. This allows Java programs to run on different platforms. A compiled Java byte code can be executed on any computer platform on which the Java interpreter is installed. For each type of computer, a different Java interpreter is required to interpret the Java byte code. This is illustrated in Figure. 2.1.

Java is a strongly typed language. That means, that every variable (or expression) has a type associated with it and the type is known at the time of compilation. Thus, in Java, data type conversions do not take place implicitly.

There are three kinds of Java program:

- *Applications* These are stand-alone programs that run on a computer.
- *Applets* These are programs that run on a web browser. Appletviewer is provided to ensure that these programs run without a web browser.
- *Servlets* These are programs that run at the server side

2.1.1.2 Java as a platform

A software and/or hardware environment in which a program runs is referred to as a 'platform'. The Java platform is a software platform that is compatible with and/or executes on platforms based on different hardwares. The Java run-time environment is shown in Figure 2.2. Figure 2.3 shows the Java programming platform.

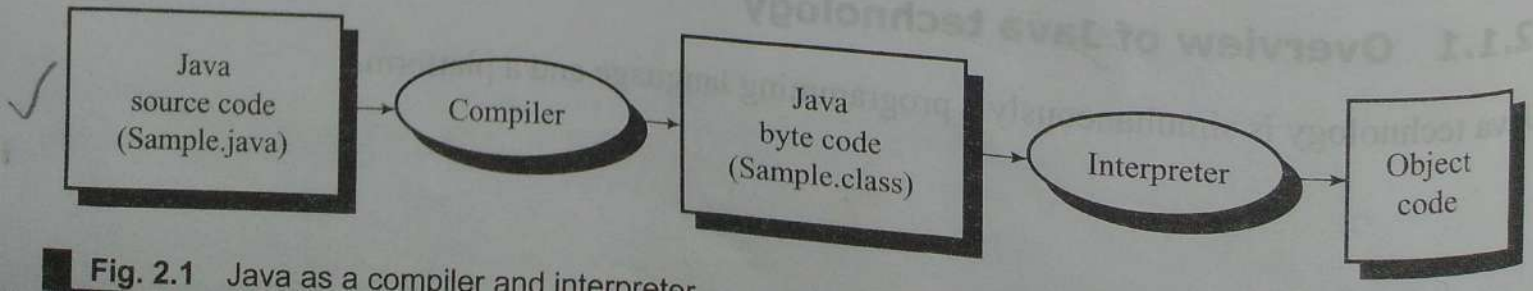

Fig. 2.1 Java as a compiler and interpreter

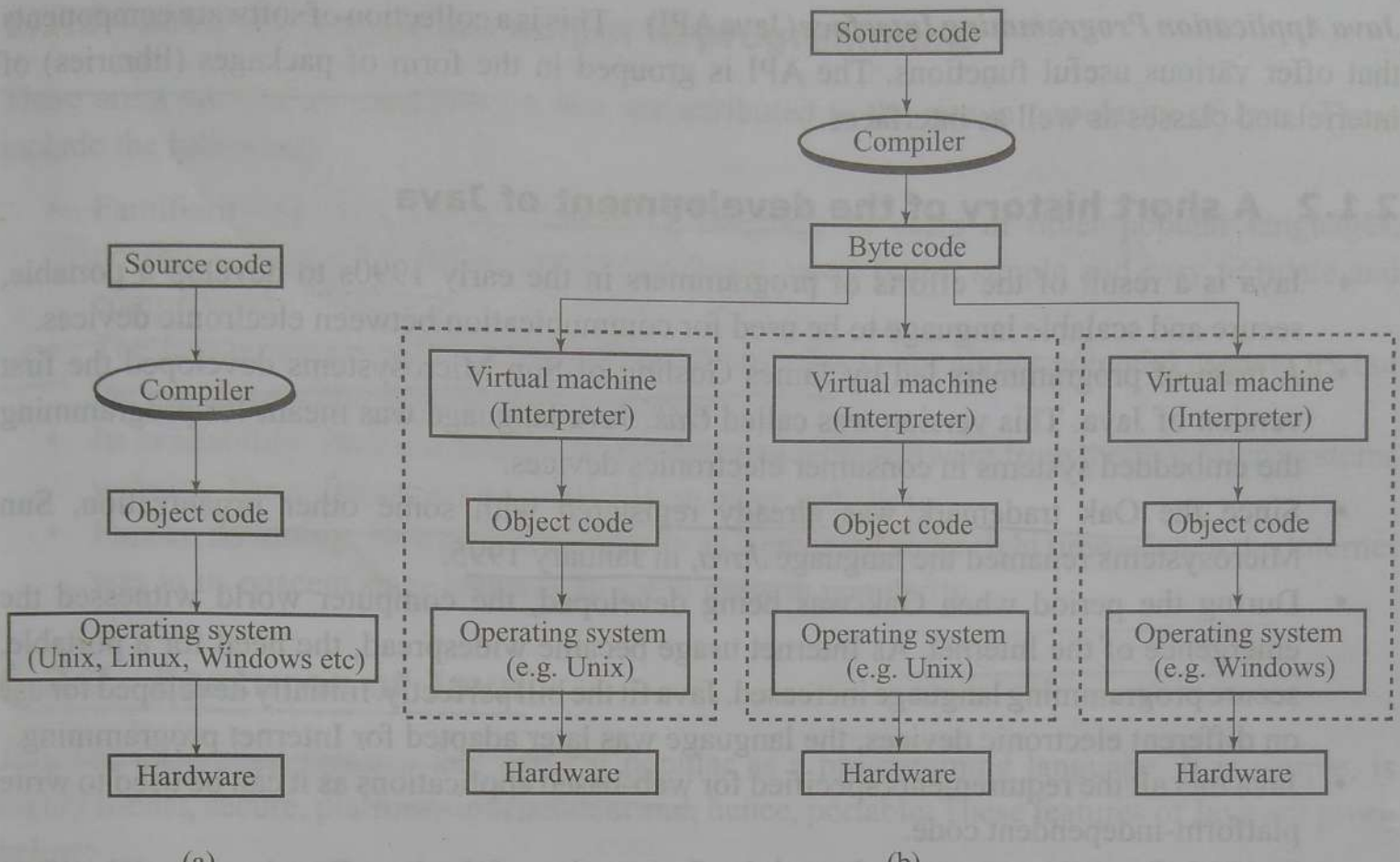

Fig. 2.2 Run-time environment for (a) a typical program; (b) Java.

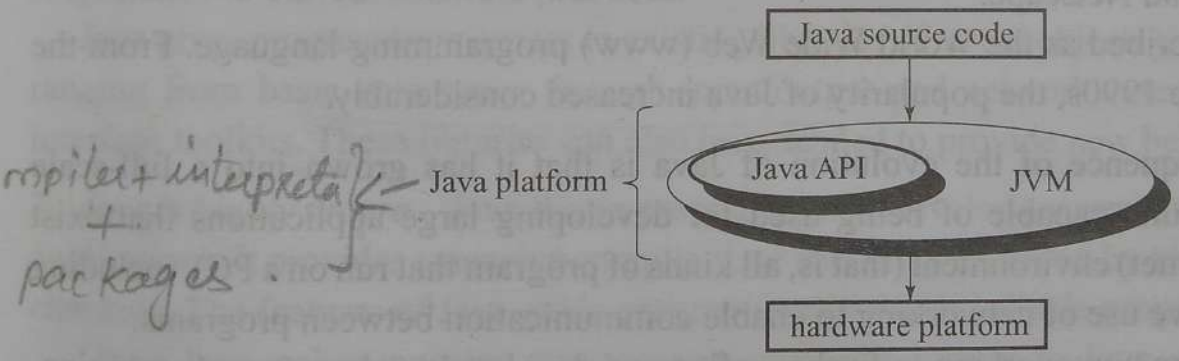

Fig. 2.3 Java programming platform.

The Java platform consists of the Java Virtual Machine and the Java Application Programming Interface. These two important components of the Java platform can be analysed and described as follows:

Java Virtual Machine (JVM) It is an interpreter, which receives bytecode files and translates them into machine language (object code) instructions. The JVM is specific to each operating system (for example, Unix, Linux, Windows).

Java Application Programming Interface (Java API) This is a collection of software components that offer various useful functions. The API is grouped in the form of packages (libraries) of interrelated classes as well as interfaces.

2.1.2 A short history of the development of Java

- Java is a result of the efforts of programmers in the early 1990s to develop a portable, secure and scalable language to be used for communication between electronic devices.
- A team of programmers led by James Gosling of Sun Microsystems developed the first version of Java. This version was called *Oak*. Java language was meant for programming the embedded systems in consumer electronics devices.
- Since the Oak trademark was already registered with some other organization, Sun Microsystems renamed the language *Java*, in January 1995.
- During the period when Oak was being developed, the computer world witnessed the emergence of the Internet. As Internet usage became widespread, the need for a portable, secure programming language increased. Java fit the bill perfectly. Initially developed for use on different electronic devices, the language was later adapted for Internet programming.
- Java met all the requirements specified for web-based applications as it can be used to write platform-independent code.
- In 1995, Sun Microsystems released the first version of the Java Development Kit (JDK) and *HotJava*, a Java-enabled browser. HotJava was capable of running Java applications in the form of Java Applets embedded in web pages, a feature since taken up by browsers from Microsoft and Netscape.
- Java is often described as the World Wide Web (www) programming language. From the second half of the 1990s, the popularity of Java increased considerably.

An important consequence of the evolution of Java is that it has grown into a full-scale development system, quite capable of being used for developing large applications that exist outside of the web (Internet) environment (that is, all kinds of program that run on a PC), including those that make extensive use of networking to enable communication between programs.

Sun Microsystems are now working to further refine and develop Java language. In addition, they are also working on other Java tools and applications. Prominent among these are the following:

- *Java Beans*—the Java object component technology.
- *JavaServer*—a complete web server application written in Java, supporting servlets.
- *JDBC*—Java Database Connectivity, providing a Java-based interface to SQL databases.
- *Java Workshop*—a Java programming environment for developing Java programs which is itself written in Java.

To get the latest updates on other Java tools and applications, readers can refer to the website of Sun Microsystems—http://www.sun.com.

2.1.3 Java as a new paradigm in programming

There are a number of other factors, that are attributed to the rise in popularity of Java. These include the following:

- Familiarity of Java as a programming language to users of other popular languages, notably C++ and Smalltalk: due to this factor, users find it simple and easy to relate and comprehend.
- The Java development kit is offered free of cost: this factor allows potential users to try out the language with minimal start-up cost.
- Its availability: Java is offered as freely downloadable software from the Sun Microsystems website. The software is also available on other websites.
- Finally, its timing: Java gained popularity as it emerged at the right time—when the Internet was in its nascent stage and was all set to grow in popularity.

2.2 Features of Java

Java has unrivalled features that make it popular as a programming language. It is simple, is highly robust, secure, platform-independent and, hence, portable. These features of Java are given below:

1. *Java is a simple and object oriented language* Java is based on object oriented concepts. This makes it familiar to programmers using C++ and other object oriented programming languages. It is attuned to current software practices.

Java gives programmers access to existing libraries of tested objects that provide functionality ranging from basic data types through input/output and network interfaces to graphical user interface toolkits. These libraries can also be extended to provide new behaviour.

2. *Java is highly secure* Java allows programmers to develop (or create) highly reliable software applications. It provides extensive compile-time checking, followed by a second level of run-time checking. The features of Java guide programmers towards reliable programming practices.

Using Java makes memory management extremely simple. In Java, objects are created with the operator `new`. There are no explicit programmer-defined pointer data types and no pointer arithmetic. At the same time, garbage collection is automatic. This simple memory management model eliminates entire classes of programming errors that haunt C and C++ programmers.

When a pointer goes out of scope, a garbage collector usually relies on the compiler to deallocate the object the pointer points to. Most of C and C++ compilers do not provide garbage-collection mechanisms and the programmer has to explicitly specify the `free()` in C or `delete` in C++. In large and complex programs, improper use of references to memory may lead to errors during execution. In Java, when an object is no longer required, the Java garbage collector automatically reclaims the memory occupied by the object. The memory allocated to an object is not deallocated explicitly when it is used.

Java technology is designed to operate in distributed environments, which means that security is of paramount importance. The security features of Java allow programmers to design applications that are extremely difficult to tamper with. In network environments, applications written in Java are safe from intrusion by unauthorized codes attempting to get behind the scene and create viruses or invade file systems.

3. *Java is architecture neutral and portable* In heterogeneous network environments, applications must be capable of being executed on a variety of hardware architectures and operating systems. In addition, the applications should be able to interoperate with multiple programming language interfaces. Java supports applications that are deployed in heterogeneous network environments. To accommodate diversity of operating environments, the Java compiler generates a *byte code*.

Unlike a C compiler, the Java compiler does not produce a native executable code for any particular machine. Instead it produces a special format called the *byte code* which is an architecture-neutral intermediate format designed to transport the code efficiently to multiple hardware and software platforms. It does not contain any machine- or hardware-related information.

Using the byte code, Java solves the problem of platform-independence. The Java byte code written in hexadecimal characters, byte by byte, looks like this:

CA FE BA BE 00 03 00 2D 00 3E 08 00 3B 08 00 01 08 00 20 08

Byte codes are precisely defined and remain the same for all platforms. Java programs that have been compiled into byte codes still need an interpreter to execute them on any given platform. The interpreter reads the byte code and translates it into the native language of the host machine. See Figure 2.4 for an illustration of the process of compiling and running a Java program.

Understanding the byte code Since the byte code is completely platform-independent, only the interpreter and a few native libraries need to be ported to get Java run on a new computer or operating system. The rest of the run-time environment, including the compiler and most of the class libraries, are written in Java.

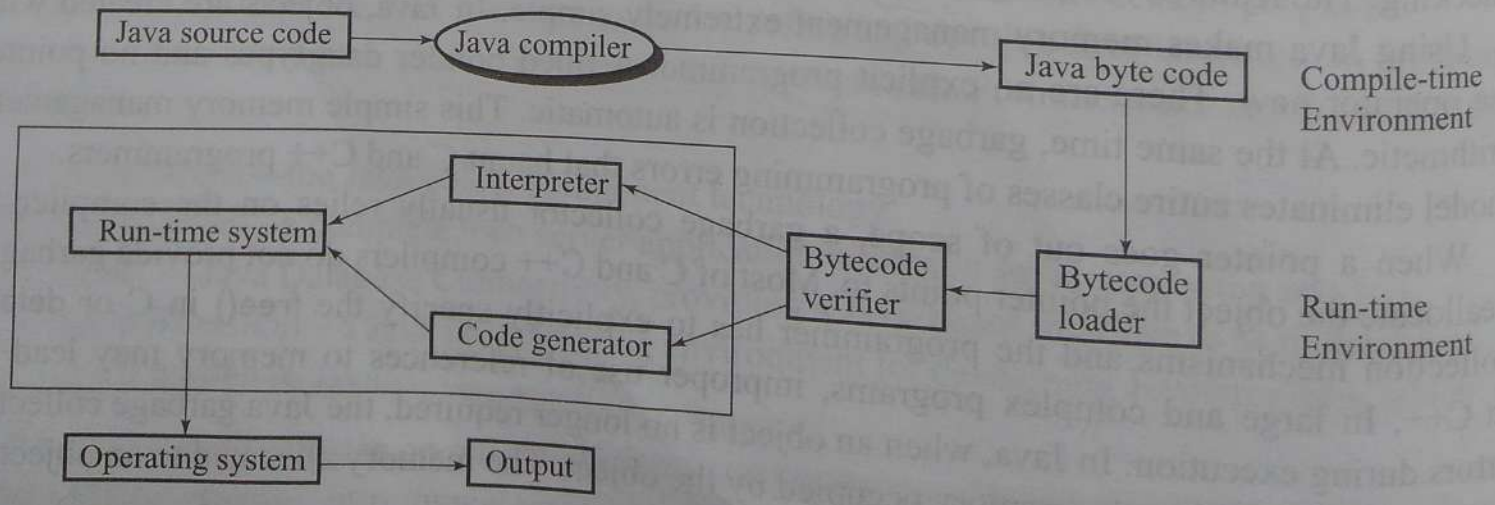

Fig. 2.4 Complete Java environment.

Use of the interpreter in Java technology solves both the binary distribution problem and the version problem. This ensures that the same Java programming language byte codes will run on any platform.

Java technology takes portability a stage further by being strict in its definition of the basic language. Java technology specifies the sizes of its basic data types and the behaviour of its arithmetic operators. All programs are the same on every platform—there are no data type incompatibilities across hardware and software architectures. To get an idea of the variety of machines and platforms where a Java byte code may be executed, see Figure 2.5.

The architecture-neutral and portable language platform of Java is known as the Java Virtual Machine (JVM). The JVM is the specification of an abstract machine for which the Java programming language compilers can generate codes. Implementing the JVM on new architectures is a relatively straightforward task as long as the target platform meets basic requirements such as support for multithreading and other such requirements.

4. *Java shows a high performance* Higher performance is ensured by Java by adopting a scheme in which the interpreter runs/executes at full speed without needing to check the run-time environment. The automatic garbage collector runs as a low-priority background thread, ensuring a high probability that memory is available when required, leading to better performance.

5. *Java is interpreted, threaded, and dynamic* The Java interpreter executes the Java byte code directly on any machine to which the interpreter and run-time system have been ported. The multithreading capability of Java provides the means to build applications with many concurrent threads of activity. Multithreading thus results in a high degree of interactivity for the end user.

Java supports multithreading at the language level which results in asynchronous behaviour of the program. This is discussed in detail in chapter 6.

The language library provides the *thread* class, and the run-time system provides monitor and condition lock primitives. At the library level, moreover, Java technology's high-level system libraries have been written to be thread-safe and the functionality provided by the libraries is available without conflict to multiple concurrent threads of execution.

While the Java compiler is strict in its compile-time static checking, the language and run-time system are dynamic in their linking stages. Classes are linked only as needed. New code modules can be linked in on demand from a variety of sources, even from sources across a network.

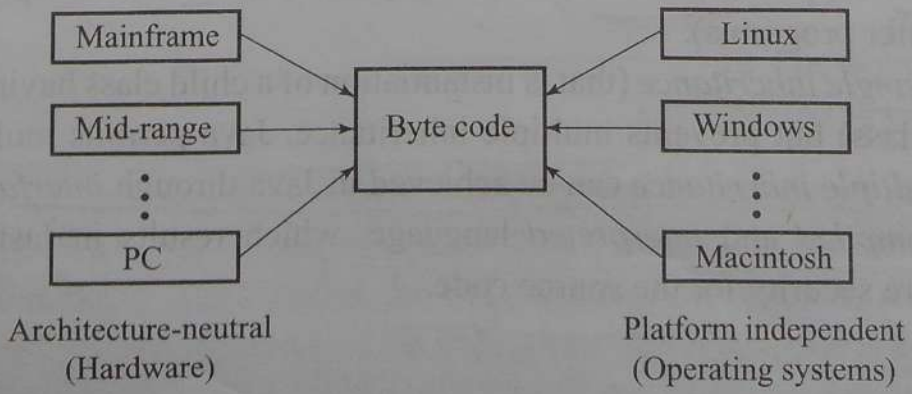

Fig. 2.5 Applicability of the Java byte code.

6. ***Java is dynamically linked*** Java does not have an explicit link phase. The Java source code is divided into .java files, roughly one per each class. The compiler compiles these into .class files containing byte code. Each .java file generally produces exactly one .class file.

The compiler searches the current directory and directories specified in the CLASSPATH environment variable to find other classes explicitly referenced by name in each source code file. If the file that is being compiled depends on other, non-compiled files, the compiler will try to find them and compile them as well. The compiler handles circular dependencies as well as methods that are used before they are declared. It also determines whether a source code file has changed since the last time it was compiled.

More importantly, classes that were unknown to a program when it was compiled can still be loaded at run-time. For example, a web browser can load applets of differing classes that it has never seen before without recompilation.

Furthermore, Java .class files tend to be quite small, a few kilobytes at most. It is not necessary to link in large run-time libraries to produce a (non-native) executable. Instead the necessary classes are loaded from the user's CLASSPATH.

7. ***Other features of Java*** Java has no pointers. Java programs cannot access arbitrary addresses in memory. All memory access is handled behind the scenes by the (presumably) trusted run-time environment. Java provides automatic memory allocation and de-allocation.

2.3 Comparing Java and Other Languages

The best way to describe a computer language is to compare its characteristics with those of other languages. The major features in which Java differs from other languages are listed below:

- Java is similar to C and C++ but is *strongly typed* (that is, every identifier must have a type) like Pascal or Ada.
- Java is *dynamically linked* which means that it does not look for a sub-program until it is called. This results in smaller programs than are possible with statically linked programs and permits more efficient transfers of code over the Internet.
- Java is *case-sensitive*, therefore care should be taken in naming fields, identifiers and other attributes.
- Java does not support *pointers*. It does not permit the direct manipulation of *addresses* (results in safer programs).
- Java allows *single inheritance* (that is instantiation of a child class having properties of only one parent class) but prevents multiple inheritance. Java permits multi-level inheritance, however. *Multiple inheritance* can be achieved in Java through *interfaces*.
- Java is a *compiled* and *interpreted* language, which results in faster applications and provides more security for the source code.

2.4 Applications and Applets

Typically, Java programs are compiled into *applets* or *applications*. Applications are stand-alone programs executed by a virtual machine, while applets are intended to be loaded into and interpreted by a browser such as Netscape or Internet Explorer. Applets are more complex than applications because they interact with the current environment of the browser, while applications are more like standard programs. Applets are discussed in detail in chapter 10.

Java has no pointer types and presents a computation model that guarantees safety. Thus, when an user executes a Java application or applet from an untrusted source, he or she can be assured that the Java program will not use the memory in unknown or perhaps malignant manner.

Moreover, the popularity of Java code distribution has come about because it is portable. Unlike source code, byte code can be distributed. When an application has to be given to multiple users working on different platforms, the Java byte code can be distributed instead of the source code. A byte code cannot be modified by a programmer this limits software piracy.

2.4.1 Simple applications using Java

A clear understanding of the above-mentioned features can be gained by writing a simple program using Java. Program 2.1 illustrates how to print a sentence. The labels indicate parts of the program.

Program 2.1 A simple Java program.

```
class StartProgram                                  ---> Class declaration

{                                                   ---> Start of program block
    // Comments
    public static void main(String args[ ])        ---> Method
    {
        System.out.println ("This is a simple java program.");
    }
}                                                   ---> End of program block
```

The above program declares a class named StartProgram. In this class a method called main() is defined. In this method a single method is invocated that displays the string 'This is a simple java program.' This string is displayed on the console when the program is executed. Output is achieved by invoking the println method of the object out. The object out is a class variable in the class System (various classes in Java are described later) that performs output operations on files.

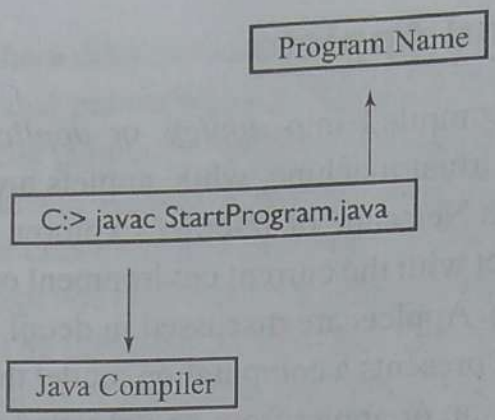

Fig. 2.6 Compiling a Java program.

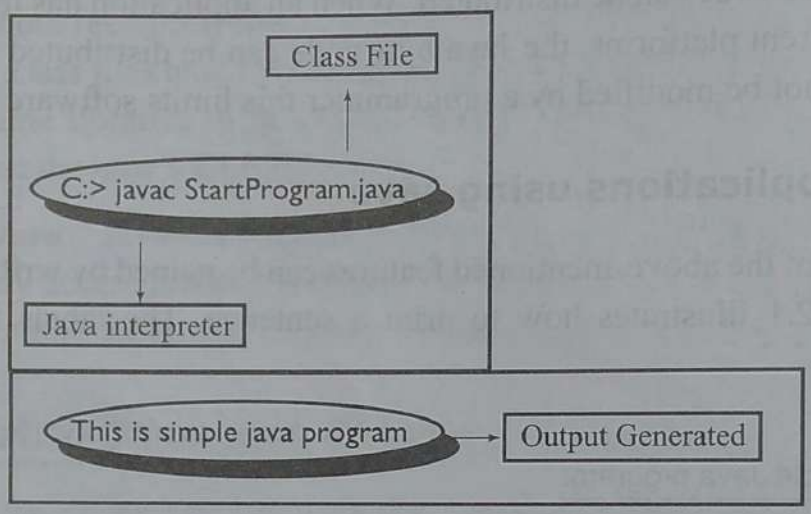

Fig. 2.7 Invoking the Java interpreter.

2.4.1.1 Compiling and running a Java program

A simple program such as Program 2.1 is usually written using notepad (or any other compatible editor) and saved with the extension *java*. The name of the program should match the name of the class. In the case of Program 2.1, the program name would be StartProgram.java (as the class name declared within the program is StartProgram). The Java compiler is used to compile the program. The Java compiler is offered with the Java development kit, as shown in Figure 2.6.

On compiling the program a class file is created, that is, StartProgram.class. For the purpose of executing (running) the class file, Java provides an interpreter (java). This can be invoked as shown in Figure 2.7.

2.4.2 Other applications and applets

Look at Program 2.2.

Program 2.2 Calculating the sum of two integers.

```
import java.io.*;                                          Package
class FirstJavaProgram                                     Class name
{
                                  Method name
    public static void main(String[ ] args) throws IOException
    {
                                  Object      Class       Class
        InputStreamReader isr = new InputStreamReader(System.in);
                                  Object
        BufferedReader br = new BufferedReader(isr);
                                                          Variable
        System.out.print("Enter First Integer:");
                                                          Method
        int a = Integer.parseInt(br.readLine());
                                                          Class
        System.out.print("Enter Second Integer:");
        int b = Integer.parseInt(br.readLine());
        System.out.print("Sum:"+(a+b));
    }
}                                 Class       Class       Class
```

Output:

```
Enter First Integer: 23
Enter Second Integer: 45
Sum: 68
```

Program 2.2 reads two integers numbers and gives their sum. Java provides the I/O (input/output) package to conduct input and output operations. This package has to be imported in any Java program that undertakes I/O operations.

In Program 2.2, I/O streams (this topic is given indepth coverage in chapter 7) are used to perform console operations. System.in returns the bytes from console, which is wrapped by the InputStreamReader, which converts bytes to characters. BufferedReader is used to buffer the characters.

The readLine() method on the BufferedReader will return the data String upto the carriage return. This method is used to read data from the console. By using the method Integer.parseInt()

the string data are converted into the integer values. The data entered are added and the output is directed to console using the System.out.print() method.

Look at Program 2.3, which is a simple applet program:

Program 2.3 A simple applet program.

```
import java.applet.*;
public class StartApplet extends Applet
{
    String message = " Applet program.";
    String message1 ="";

    public void Start()
    {
        System.out.println(message);
        System.out.println(message1);
    }
    public void init()
    {
        StartApplet Text;
        Text = new StartApplet();
        Text. message1 ="Writing the message.";
        Text.Start();
    }
}
```

In Program 2.3, the line

```
import java.applet.*;
```

implies that the standard class Applet should be imported before writing an applet program. This would ensure that all the features of the class Applet can be accessed by the program (child class). Another thing to note is that any applet should extend the class Applet. This ensures that StartApplet would be a child class of the class Applet (in other words, StartApplet inherits the methods and objects in Applet).

In Program 2.3, two String types message and message1 are defined. These two types would be utilized by the methods Start() and init().

The method Start() outputs two lines of text to the default system output that is the DOS window.

The method init() is the main executable part of a Java program when StartApplet is run as an applet. The init() method instantiates the class Text and replaces the empty string in message1 with 'Writing the message,' and then calls the method Start to print the two lines of text.

Stand-alone applications written in Java have the main method. Not being stand-alone applications, applets and servlets have no main method.

Stand-alone applications are programs that are written and run on a stand-alone computer. When a stand-alone Java program is executed, the Java compiler searches for a class which has the same name as the name given to the program file, and then finds the main method in this class. One the other hand, applets and servlets are useful to work in a web environment. These programs are placed inside a web page that is usually written in HTML (Hyper Text Markup Language). The applet/servlet is executed through a browser and the browser handles the starting point of the execution. Applets can also be executed through appletviewer, which is a software provided by the Java environment to run applets.

After the program StartApplet.java is compiled, a HTML document calls and runs the applet StartApplet.class (note that when the StartApplet.java is compiled the output is StartApplet.class.)

Writing HTML code for the program StartApplet Once the program has been written and compiled, it has to be executed. The HTML document that calls and executes the applet is written as follows:

```
<html>
<head>
<title>StartApplet</title>
</head>
<body bgcolor ="aaaaaa">
<h1> An Applet Program </h1>
<applet code=StartApplet.class  width=40 height=40></applet>
</body>
</html>
```

The above HTML file is saved as AppletHtml.html. When this file is opened through a web browser (either Internet Explorer or Netscape Navigator), the program StartApplet.class is executed.

Invoking the applet without a HTML code Another method to invoke the applet without writing the HTML code is by including the following code (code given in bold) in the StartApplet program itself.

Program 2.4 Executing an applet program without a HTML code.

```
import java.applet.*;
/*<applet code = StartApplet height = 400 width = 400></applet>*/
public class StartApplet extends Applet
{
    String message = "Applet program.";
    String message1 ="";
```

```
    public void Start()
    {
        System.out.println(message);
        System.out.println(message1);
    }
    public void init()
    {
        StartApplet Text;
        Text = new StartApplet();
        Text. message1 ="Writing the message.";
        Text.Start();
    }
}
```

To run this program use appletviewer:

```
C:\> appletviewer StartApplet.java
```

Running the applet gives the following output:

```
Applet program.
Writing the message.
```

2.5 Java Development Kit

Sun Microsystems offer the JDK, which is required to write, compile and run Java programs. This is freely available on the web site of Sun Microsystems. In addition, JDK is offered as part of commercial development environments by third-party vendors.

The JDK consists of the following:

- Java development tools, including the compiler, debugger and the Java interpreter.
- Java class libraries, organized into a collection of packages.
- A number of sample programs.
- Various supporting tools and components, including the source code of the classes in the libraries.
- Documentation describing all the packages, classes and methods in Java.

As already mentioned in the previous section, to transform a program into a format that a computer can understand, a compiler is needed. The default Java compiler provided by the JDK is called **javac**. In order to execute or run a program a *run-time environment* containing a Java *interpreter* is needed. The Java interpreter provided by JDK is called **java**. Other run-time environments include Java Run-time Environment (JRE), browsers with inbuilt runtime environments and AppletViewer to execute/run applets even without a browser. The JDK also

provides a debugger tool for *debugging* programs. Debugging is the process of locating and fixing errors in programs.

2.6 More Complex Programs

Look at Program 2.5. It introduces the concepts of access specifier, constructor and the keyword static.

Program 2.5 A Java program with two classes.

```
class StartProgram                                   <-- Class declaration
{
    public void Message()                            <-- Method
    {
        System.out.println ("This is a simple java program.");
    }
}                                                    <-- Access specifier
public class SecondProgram
{
    public static void main(String args[])
    {                                                <-- Used to reserve memory
        StartProgram s= new StartProgram();
        s.Message();
        System.out.println("Two classes in a single java file");
    }
}
```

In Java, the class that contains the main method is the first one to be invoked by the JVM. On compiling the file, the following two class files are created.

```
StartProgram.class
SecondProgram.class
```

On executing these class files, the output generated is

```
This is a simple java program.
Two classes in a single java file
```

2.6.1 Special features of this program

Access Specifier Access specifiers define the scope of a variable, method or class. Java provides four access specifiers for attributes and methods, namely private, protected, public and default.

'package private' is a default specifier, which means there is no explicit specification given for an attribute or a method. Access specifiers will be covered in chapter 4.

Constructors A constructor is a function that is automatically executed whenever an object is created. It is used for initializing any class member. Constructors are dealt with in chapter 4.

Static The static keyword allocates memory. Until a method is static it will not be loaded into memory.

The method main() Every Java file must contain a main() method (except applets, and servlets). Each Java file may have more than one class and atleast one of these classes must have main. Java executes the file specified at the command line in which the class containing the main method exists. The signature of the main method is as follows:

```
public static void main(String args[])
```

The main() method always has public accessibility so that the interpreter can call it. It is a static method belonging to the class. It does not return a value, that is, it is declared void. It always has an array of String objects as its only formal parameter. This array contains any arguments passed to the program on the command line.

2.7 Java Source File Structure

The source code of a Java program is written in a plain text file, which is commonly referred to as Java source file. The extension of this file name is .java. The source file contains the code required to perform a particular task.

A Java source file consists of the following elements:

- An optional package definition to specify a package name: classes and interfaces defined in the file will belong to this package. If the package name is not given, the definitions will belong to the *default package.*
- A number of class and interface definitions: technically a source file need not have any such definitions, but that is hardly useful. The classes and interfaces can be defined in any order. Note that *JDK imposes the restriction that only one* public *class may be defined per source file*, and it requires that the file name match this public class. If the name of the public class is NewApp then the file name must be NewApp.java.

The above structure is depicted by a skeletal source file as shown below.

Java Source File Structure

```
// Filename: NewApp.java
// PART 1: (OPTIONAL)
// Package name
package com.company.project.Package;
// PART 2: (ZERO OR MORE)
```

```
// Packages used
import java.util.*;
import java.io.*;
// PART 3: (ZERO OR MORE)
// Definitions of classes and interfaces (in any order)
public class NewApp {}
class C1 {}
interface I1 {}
// ...
class Cn {}
interface Im {}
// end of file
```

2.8 Prerequisites for Compiling and Running Java Programs

To write, compile and execute a java program, the first requirement is a Java development kit. The Java 2 Standard Edition (J2SE) is a premier solution for rapidly developing and deploying mission-critical, enterprise applications. It provides the essential compiler, tools, run-times and APIs for writing, deploying and running applets and applications in the Java programming language.

The J2SE is at the core of Java technology, and version 1.4 raises the Java platform to a higher standard. From client to server, from desktop to supercomputer, improvements have been made to J2SE across the board. With version 1.4, enterprises can now use Java technology to develop more demanding business applications with less effort and in less time.

J2SE is available on the website of Sun Microsystems. This software development kit is freely downloadable from the following website.

http://java.sun.com/j2se/1.4.1/download.html

The following link would give access to documentation.

http://java.sun.com/j2se/1.4/docs/index.html

In order to compile Java programs, the Standard Java Development Kit (J2SDK) has to be downloaded and should be installed on the computer. After the software is installed, the class path has to be set. The class path specifies where the Java library is available. In the Java library, two Java files dt.jar and tools.jar are available.

Java became a popular object-oriented programming language because of its various features and the complete Java software is freely available to work. In this chapter, we have covered the history and characteristics of Java language. This chapter forms an introduction towards compiling and executing Java program. From here we move on to the fundamentals of Java in chapter 3.