PROGRAMMING IN JAVA LANGUAGE

Feon Jainson
Dr. Preethi D

BOOKS ARCADE

Programming in Java Language

Programming in Java Language

Feon Jainson
Dr. Preethi D

BOOKS ARCADE
KRISHNA NAGAR, DELHI

Programming in Java Language

Feon Jainson
Dr. Preethi D

BOOKS ARCADE

Regd. Office:
F-10/24, East Krishna Nagar, Near Vijay Chowk, Delhi-110051
Ph. No: +91-11-79669196, +91-9899073222
E-mail: info@booksarcade.co.in, booksarcade.pub@gmail.com
Website: www.booksarcade.co.in

Year of Publication 2023

International Standard Book Number-13: 978-81-19199-83-9

CONTENTS

CHAPTER 1
JAVA LANGUAGE FUNDAMENTALS

Feon Jainson, Assistant Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- feon.jaison@jainuniversity.ac.in

To carry out their computational jobs, programmers create instructions in a variety of programming languages, such as: Machine level language, assembly level language, high level language, and other languages

Machine level Language: A collection of instructions that are immediately carried out by a computer's central processing unit is known as machine code or machine language (CPU). Each instruction does a highly particular action on a data unit in a CPU register or memory, such as a load, a jump, or an ALU operation. Any programme that a CPU directly executes consists of a set of these instructions.

Assembly level language: An assembly language (or assembler language) is a low-level programming language for a computer or other programmable device in which the language and the architecture's machine code instructions have a very close (often one-to-one) connection. An application known as an assembler transforms assembly language into executable machine code; the conversion process is known as assembly, or assembling the code.

High level Language: A high-level language is any programming language that allows for the creation of programmes in significantly less complicated programming environments and is often independent of the hardware architecture of the machine. High-level languages are more abstracted from the computer and concentrate on the programming logic rather than the supporting hardware, such as register use and memory access.

In the 1950s, the first high-level programming languages were developed. Ada, Algol, BASIC, COBOL, C, C++, JAVA, FORTRAN, LISP, Pascal, and Prolog are just a few of the many languages that are now available. Since they are farther away from machine languages and more similar to human languages, these languages are regarded as high-level. In contrast, due to their strong resemblance to machine languages, assembly languages are regarded as low-level.

The two major groups of high-level programming languages are as follows:

1. Procedure-oriented programming (POP) language.
2. A language for object-oriented programming.
3. Procedure-Oriented Language for Programming

The issue is seen as a series of tasks to be completed, such as reading, calculating, and printing, in the procedure-oriented approach. Writing a set of instructions or actions for the computer to execute and grouping those instructions into functional units is procedure-oriented programming (Figure 1.1).

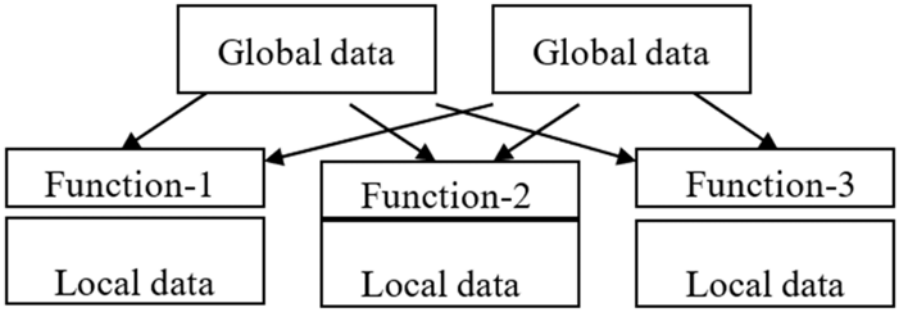


Figure 1.1: Procedure-Oriented Language for Programming

Global data access is one drawback of procedure-oriented programming languages. It does not adequately mimic real-world word problems. Data is not hidden

Procedure-oriented programming characteristics:

1. The focus is on acting (algorithm)
2. Big programmes are broken up into smaller ones called functions.
3. The vast majority of functions exchange world data.
4. Data flow between functions of the system is open.
5. The function changes the format of the data.
6. Uses top-down methodology while designing programmes

Programming with objects

By providing partitioned memory areas for both data and functions that may be used as templates for constructing copies of such modules on demand, object-oriented programming is a method for modularizing applications Figure 1.2.

Object-oriented programming features

1. Action is prioritised above protocol.
2. Programs are split up into units referred to as objects.
3. The data structures are created in a way that they describe the objects.
4. The data structure links together functions that work on an object's data.
5. Data is buried and inaccessible to outside functions.
6. Functions allow objects to interact with one another.
7. It's simple to add new information and features.
8. Uses a bottom-up design process for programmes.

Basic Object-Oriented Programming Concepts

1. Objects
2. Classes
3. Data encapsulation and abstraction
4. Inheritance
5. Polymorphism
6. Dynamic binding
7. Passing of messages

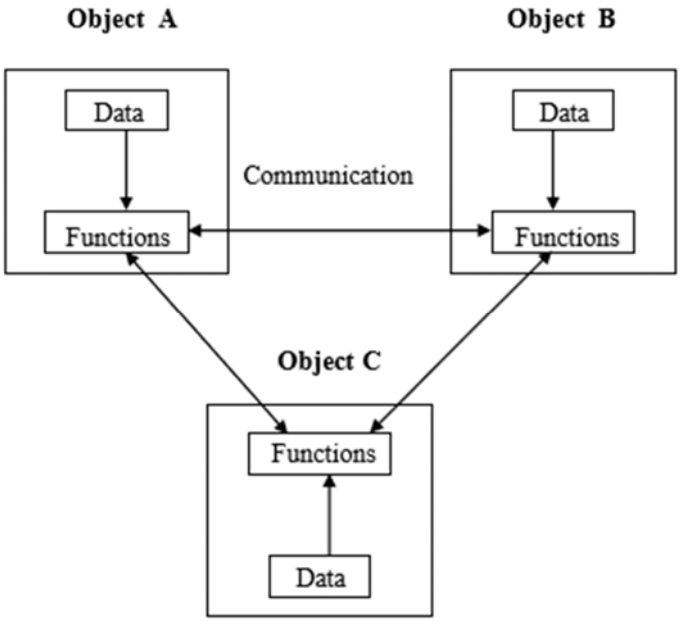


Figure 1.2: Programming with objects

Objects: In an object-oriented system, objects serve as the fundamental run-time entities. They might stand in for any object that the software has to manage, such a person, a location, a bank account, a table of data, etc. The core concept underpinning object-oriented thinking is the creation of objects, which are singular entities that mix data and function. Objects are a mix of data and software that stand in for some real-world thing. Consider the case of Amit, who is 25 years old and earns $2500 each year. In a computer application, the Amit may be shown as an object. The object's data component would be (name: Amit, age: 25, salary: 2500)

The object's programme component might consist of a collection of programmes (retrive of data, change age, change of salary). Any user-defined type, like employee, may generally be utilised. Name, age, and income are referred to as characteristics of the item in the Amit object (Figure 1.3).

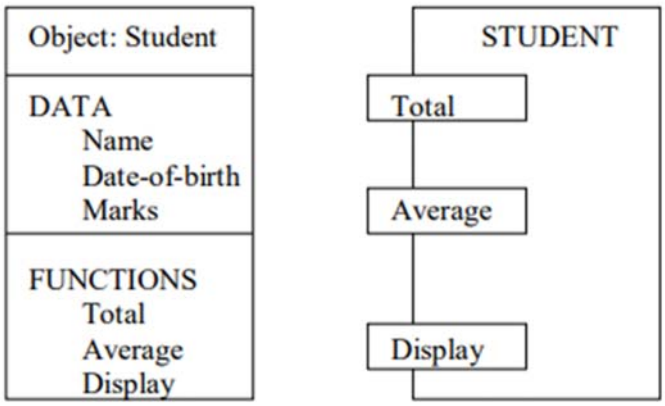


Figure 1.3: Objects are a mix of data and software

CLASS: A class is a collection of objects that have similar characteristics for certain programme parts and some data parts. A class is a brand-new data type in C++ that includes

member variables and member functions that work with the variables. One of the most well-liked and commonly used platforms and programming languages is Java. An environment that supports the creation and execution of writing programs in any computer language is known as a platform. Java is quick, dependable, and safe. Java is utilized everywhere, from desktop to online apps, game consoles to scientific supercomputers, mobile phones to the Internet. Java is a language for object-oriented programming (OOP). It has nearly all OOP features. In Java, which is somewhat similar to C++ but more straightforward than aristocrat C++, basic OOP features include object creation through object templates, or classes, data abstraction and encapsulation, data and code sharing through inheritance, the overloading concept through polymorphism, and data/process hiding. The confusing, unpredictable, and superfluous elements of C++ have really been removed from this attractive language. In Java, pointers are not used; instead, heterogeneous collections of data/objects are minimally bound. Pointers manipulation is one of the richest sources of flaws in virtually all programs, hence the absence of any form of pointer equates to the absence of significant bugs. Java is smart enough to aid programmers in communicating intricate concepts. Again, unlike C/C++, global data and stand-alone functions are not feasible here. The fundamental building blocks of Java programming are objects, which are either direct or indirect descendants of classes. Java's class definition allows for both static and dynamic binding, allowing for complete code reuse and data exchange. Java is genuinely object-oriented programming since it allows for dynamic inheritance of various class definitions. Java does not, however, permit multiple inheritance, whose usage and semantics have generated considerable debate. In Java, type casting and operator overloading are also prohibited .

Varieties of Java:

The capabilities of Java vary depending on the edition. Java is available in three editions:

Java Standard Editions (JSE): JSE, or Java Standard Editions, is a programming language for desktop computers.
Java Enterprise Edition (JEE): It's used to build sizable server-based apps that handle tremendous traffic and intricate transactions.
Java Micro Edition (JME): It is used to create programs for compact gadgets like set-top boxes, phones, and appliances.

Java Application Types

Java programming allows for the creation of four different types of Java applications:

Apps that run independently: Java standalone applications make advantage of GUI elements like AWT, Swing, and JavaFX. They include buttons, a list, a menu, a scroll panel, etc. Aliens on the desktop are another name for it.

Enterprise Applications: A distributed application is referred to as an enterprise application.

Web Applications: Web Applications are programs that operate on a server. For building web applications, we leverage JSP, Servlet, spring, and Hibernate technologies.

Mobile Apps: Java ME is a cross-platform development environment for creating mobile apps that work on smartphones. The platform for Android app development is Java.

Properties of JavaSimple: Java is an easy-to-understand language due of its clear, concise grammar. Java either replaces or eliminates C++'s voluminous and unclear notions. For instance, Java does not employ operator or pointer overloading.

Object-oriented: Everything in Java takes the shape of an object. It indicates the device has some data and behavior. There must be at least one class and object in a program.

Robust: Java tries to check errors both during compilation and during runtime. It makes advantage of the powerful garbage collector memory management mechanism. It is powerful because of its excellent handling and trash collecting characteristics.

Secure: Java contains no explicit pointers and runs programs in a virtual machine, making it a secure programming language.

Platform-Independent: Java offers the assurance that code may be written once and executed on any platform. Any machine may execute this platform-neutral byte code.

Portable: Java Byte code is portable, meaning it can run on any platform. No features that depend on the implementation. The size of primitive data types, for instance, is fixed, as is everything else related to storage .

High Performance: Java is an interpreted language with high performance. Java uses the Just-In-Time compiler to achieve excellent speed.

Distributed: Java offers networking features as well. Because it supports the TCP/IP protocol, it is made for the distributed internet environment. It is capable of operating online. To build a distributed system, EJB and RMI are utilized. The first real-world example that transitions to components and distributed computing is Java. In this new kind of programming environment, a program is really task-oriented and disposable. The advantages of running pre-compiled code as a single transparent statement that may be dynamically downloaded from anywhere (perhaps too far away) in the network of heterogeneous computers are available. The majority of software developers have had this ambition for years.

Multi-threading: Multi-threading is supported by Java as well. It refers to managing many tasks at once. The capacity to conduct many processes concurrently within the framework of a single big program is what multithreading essentially entails. In its simplest form, a program is a list of instructions, and the thread of execution is the path through the list of instructions. Often, it makes more logical to run a program over numerous threads .

Introduction to OOPS: A computer programming paradigm known as object-oriented programming (OOP) arranges the architecture of software around data or objects rather than functions and logic. An object is a data field with particular characteristics and behavior. OOP places more emphasis on the objects that programmers desire to handle than on the logic necessary to do so. Large, sophisticated, and actively updated or maintained applications are a good fit for this kind of development. This encompasses mobile apps as well as design and production software. OOP, for instance, may be applied to manufacturing system simulation software. In collaborative development, when projects are separated into groups, the technique is advantageous due to the structuring of an object-oriented program. OOP also offers the advantages of efficiency, scalability, and reused code .

Class: An individual data type is a class. When a class instance is created, its data members and member methods may be accessed and utilized. It stands for the collection of traits or operations that all objects of a particular type share. Take the class of cars as an example. There may be a lot of automobiles with various names and brands, but they all have some features in common, such as four wheels, a speed limit, a range in miles, etc. Therefore, wheels, speed restrictions, and mileage are their attributes, and Car is the class in this case.

Object: It is a fundamental building block of Object-Oriented Programming and represents actual physical things. A Class is an instance of an Object. No memory is allocated when a class is specified, but it is allocated when it is instantiated (i.e., when an object is formed). An object's identity, state, and behavior are all defined. Data and code to alter the data are both contained in each object. It suffices to know the form of message that the objects accept and

the type of response that they give in order for them to communicate with one another. For instance, the "Dog" object in real life contains traits like color, breed, bark, sleep, and eating habits.

Data Abstraction: One of the most fundamental and significant aspects of object-oriented programming is data abstraction. Data abstraction is the process of exposing to the outside world just the information that is absolutely necessary while obscuring implementation or background information. Take a look at a man operating a vehicle in the real world. The guy only understands that pressing the accelerators will make the car go faster and that using the brakes would make the car stop, but he is unaware of the inner workings of the car or how the accelerator, brakes, and other driving controls are implemented. Abstraction is just this .

Encapsulation: Data wrapped up into a single unit is referred to as encapsulation. It serves as the link between the code and the data that it works with. The elements or data of a class are concealed from other classes when it uses encapsulation, and the only way to access them is through any member function of the class in which they were declared. It is sometimes referred to as data-hiding because, similar to encapsulation, a class's data is concealed from other classes.

Inheritance: Inheritance is a crucial OOP tenet (Object-Oriented Programming). Inheritance is the capacity of a class to derive traits and attributes from another class. We inherit properties from existing classes when we create a new class. As a result, we may inherit properties and methods from another class that already has them, saving us from having to write them all by hand when we build a class. The usage of inheritance enables the user to minimize code repetition by reusing it whenever possible.

Polymorphism: Polymorphism refers to the existence of several forms. Polymorphism may be simply defined as a message's capacity to be presented in several forms. One person may, for instance, exhibit several distinct traits at once. A father, a spouse, and an employee are all dual roles that a man plays. Therefore, the same person exhibits diverse behavior depending on the circumstance. It is known as polymorphism.

Binding dynamically: In dynamic binding, a decision is made at runtime on the code that will be performed in response to a function call. With dynamic binding, the code connected to a specific procedure call is unknown until the call is made at run time.

Binding of Dynamic Methods: The fact that a derived class D contains every member of its base class B is one of the fundamental benefits of inheritance. An item of D can represent B in every situation where a B might be utilised, as long as D is not concealing any of the public members of B. Subtype polymorphism is the term for this characteristic.

Passing of Messages: It is a method of communication used in both parallel and object-oriented programming. When objects exchange information with one another, they are said to be communicating. A communication for an object is a request to run a procedure, and as a consequence, it will cause the receiving object to call a function that produces the required outcomes. The name of the entity, the name of the procedure, and the content of the message are all specified during message passing.

Benefits of OOPS

1. Programs run more quickly.
2. More reuse of the code.
3. Coding structure that is simple to grasp.
4. The division of work within a project based on items is quite simple.
5. Fully reusable applications may be created quickly and with little coding.

Drawbacks of OOPS

1. Since OOPS is not a universal language, it cannot be used everywhere.
2. To utilize OOPS, one must have a solid grasp of classes and objects.
3. It requires a lot of work to produce.
4. Compared to other applications, it is slower.
5. OOPS programs are larger than typical programs in size .

Classes and Objects

Classes: A class is a user-defined blueprint or layout of an item that specifies how a certain type of object will appear. In a class description, there are two components:

1. Member variables or attributes
2. Behavior or member function implementations.

Consequently, in object-oriented language: A class is a template that outlines the variables and methods that are shared by all objects of a particular sort. In contrast to procedural language, it aids in the binding of data and methods, making the code reusable. For instance, a mobile phone offers features like a brand name, RAM, and messaging and calling capabilities. Consequently, a class of varied phones is the mobile phone (the objects).

An object is produced from a user-defined class, which functions as a blueprint or prototype. It stands for the collection of attributes or operations that are shared by all objects of a particular type. The following elements can often be found in class declarations, in that order:

Modifiers: A class may be public or have default access as a modifier.

Class keyword: a class is created using a class keyword.

Class name: Name should start with an initial letter for the class name (capitalized by convention).

If there is a superclass: if applicable, the name of the class's parent (superclass), followed by the term extends. Only one parent can be extended (sub classed) by a class.

If any, interfaces: a list of the interfaces that the class, if any, implements, separated by commas, should be included. A class may support several interfaces.

Body: Braces, or, enclose the class body.

New objects are initialized using constructors. While methods are used to implement the action of the class and its objects, fields are variables that give the state of the class and its objects. Real-time applications employ a variety of class types, including nested classes, anonymous classes, and lambda expressions.

Components of a Class:

Access Modifier: A class can be made public by prepending the word public before its name. In contrast to default classes, which are exclusively available inside the same code, public classes can be accessed by any class in any module. Following the class keyword comes the class name (Mobile Phone), which has to be a legal identifier. Instance variables, or properties, and class methods are declared in the class's body (behaviors). Each method or attribute has an access specifier that is either private, protected, or public; private is the default. In our discussion of objects, we'll learn more about specifiers. The characteristics may be either user-defined (a class is also a user-defined data type) or primitive, and may be of any legal data type. The generic functions are the methods.

Class Declaration: A class declaration outlines the appearance and functional capabilities of an object. This provides an interface (what the user sees) without requiring the user to be aware of the implementation's specifics. By generating objects and using the accessible methods for those objects, the user of a class makes use of the class.

Class Definition: These are the implementation specifics for the class, which include member definitions. For the interface user, it is invisible.

Object:

An object is an entity with state and activity, such as a chair, bike, marker, pen, table, or automobile. It could be intellectual or physical (tangible and intangible). The banking system is an illustration of an intangible entity.

An item has the following three qualities:

State: A representation of an object's data (value).

Behavior: depicts how an object behaves (functions), such as when you deposit or withdraw money.

Identity: Usually, an object's identity is represented by a special ID. The external user cannot see the value of the ID. However, the JVM uses it internally to uniquely identify each object.

Pen, for instance, is an object. Reynolds is its name; its state is noted as being white. Writing is its behavior since it is utilized for writing. A class's instances are objects. A class serves as a model or blueprint from which new objects may be made. Therefore, a class's instance (or result) is an object .

Difference between Class and Objects:

The key distinction between class and object is as follows (Table 1.1):

Table 1.1: Difference between Class and Objects

S. No.	Class	Object
1	In programming, a class serves as a template for constructing objects.	The object is a class instance.
2	When a class is formed, no memory is allocated.	Every time an object is created, memory is allocated.
3	A class is a type of logical object.	An object is a real thing.
4	Consider a car.	Examples include Tesla, Jaguar, and BMW.
5	Class can only be declared once.	A class can be used to produce several objects.
6	Classes cannot be changed since they are not stored in memory.	They are controllable.
7	A class produces objects	The class has life because of its objects.

8	Using the "class" keyword, classes may be created.	In Java, you may create objects by using the "new" keyword.
9	It does not include any values related to the fields.	Every object has unique values that are connected to the fields .

Method in Java

A stack is used to implement method calls. When a method is called, a stack frame is formed in the stack area. The parameters supplied to the called method, any local variables, and the result that will be returned are then placed in this stack frame, which is then removed when the called function has completed its execution. The top of the stack is tracked by a stack pointer register, which is updated accordingly. A method is a section of code, series of statements, or body of code used to carry out a certain job or action. It helps to make code more reusable. A technique is created once and used repeatedly. Don't demand that write the same code again and over. Additionally, it offers simple code change and readability, merely by adding or deleting a section of code. The method is only ever called or invoked once.

A Java method, also known as a collection of statements that carry out a certain operation and provide the outcome to the caller, is called a method. Java methods provide the option of carrying out certain tasks without producing any output. Java's methods let us reuse code without having to retype it. In contrast to languages like C, C++, and Python, every method in Java must be a component of a class. A method exposes an object's behavior in the same way as a function does. It is a collection of programs that carry out a certain function.

Method Statement

Generally speaking, method declarations consist of six parts:

Modifier: It specifies the method's access type, or the point in your application where it may be accessed. The four different forms of access specifiers in Java.

Public: It is available in all classes of your application and is public.

Private: It is only available inside the class in which it is declared. Protected: It is accessible both within the class in which it is defined and in any subclasses.

Default: By default, no modifier is used when declaring or defining it. It may be accessed from the class and package where its class is specified.

Return type: The data type of the value that the method returned, or void if it returned nothing.

Method Name: While the protocol for method names is slightly different from that for field names, the requirements for field names still apply.

Input parameter list: Within the enclosed parentheses, an input parameter list separated by commas is defined, along with the data type for each parameter. Use empty parenthesis if there are no parameters ().

Exception list: You can provide the exceptions you anticipate the procedure to throw (s).

Method body: The method body is encased in brackets. The program that must be run in order for you to achieve your goals .

Java's Types of Methods

Java methods may be roughly divided into two categories:

Pre-defined Techniques: Predefined methods are those that are previously defined in Java class libraries. You may use these methods directly by calling them and include the required package imports. Keep in mind that anytime a specified method is invoked, a set of code is executed in the background to carry out that method. Predefined methods in Java are those that the Java class libraries have previously defined, as the name implies. This indicates that they don't need to be defined and may be called and used anywhere in our software. There are various predefined methods, each of which is specified inside its corresponding class, including sqrt(),print(),length() and max().

User-defined Methods: User-defined methods are special methods that the user has specified. These techniques can be adjusted and changed depending on the circumstance. Here is an illustration of a user-defined technique.

Statics Methods: A static method, such as DemoClass.sampleMethod, that allows us to call a method on the class itself (). Additionally, a method is referred to be an instance method when it is not marked as static and must be called from a class instance. The methods that are part of a class and not a specific instance are known as static methods. The main benefit of static methods is that calling them does not require the creation of an object. The "static" keyword can be used to construct a static method. Additionally static is the main method, which is where the Java program's execution starts.

Instance Methods: The class and its instance are both members of the non-static instance method. The instance method cannot be called without first creating an object. Additionally, instance techniques are categorized into two groups:

Accessory Approach: Accessor methods in Java are only allowed to read instance variables, hence they are used to retrieve the value of a private field. The word "get" is usually used before them.

Mutator Technique: In Java, mutator methods have the ability to read and write to instance variables as well as obtain and set the value of a private field. The word "set" is in front of them at all times.

Factory Method: The methods that return an object to the class to which it belongs are called factory methods. This category typically includes all static methods .

Encapsulation and Abstraction in Java

Encapsulation

The definition of encapsulation is the grouping of data into a single unit. It is the technique that connects the data the code manipulates with the code itself. Encapsulation may also be viewed as a barrier that stops programs from the outside of the barrier from accessing the data. Technically speaking, encapsulation means that a class's variables or data are concealed from all other classes and are only accessible through member functions of the class in which they are stated. The data hiding idea, similar to encapsulation, protects a class's data by keeping its members or methods private while still making the class visible to the end user or it is sometimes referred to as a mix of data-hiding and abstractions since it hides implementation details from the user or the outside world using the abstraction idea.Encapsulation may be accomplished by declaring all class variables as private and creating public methods for setting and obtaining variable values.

The setting and getter methods give it greater definition .

Java's Encapsulation Advantage

1. The class may be made read-only or write-only by merely including a setter or getter method. Alternatively put, you may omit the getter or setter method.
2. It gives you access to and control over the data. Might write the logic inside of the setter method, for example, if you wanted to set an ID value that could only be bigger than 100. In the setter methods, you may add logic to prevent the storage of negative integers.
3. Due to the fact that other classes won't be able to obtain the information through the private data members, it is a method for concealing data in Java.
4. Testing the encapsulate class is simple. It is hence preferable for unit testing.
5. The ability to generate getters and setters is offered by the common IDEs. Therefore, creating an enclosed class in Java is simple and quick.
6. A class's fields can be set to either read-only or write-only status.
7. Whatever is stored in a class' fields is entirely within the authority of the class.

Java's need for encapsulation

Encapsulation takes coding process improvisation to a new level. Encapsulation is necessary in Java for a number of reasons. The list of them is below.

Better Control: Absolute control over the associated data and data techniques inside the class is provided via better control encapsulation.

Obtain and Establish: The built-in support for "Getter and Setter" methods in the common IDEs speeds up development.

Security: Any external classes cannot access data members or data methods thanks to security encapsulation. The encapsulation procedure increases the security of the data that has been encased.

Flexibility: It is possible to effectively apply changes to one section of the code without having an impact on any other sections.

Abstraction in Java:

The object-oriented programming notion of abstraction "shows" only relevant properties and "hides" extraneous data. Abstraction's fundamental goal is to shield people from pointless information. Abstraction is the process of choosing information from a bigger pool such that the user only sees pertinent features of the item. It aids in lowering programming effort and complexity. It is one of the most crucial OOPs ideas.

1. An abstract keyword must be used when declaring an abstract class.
2. Both abstract and non-abstract approaches are possible.
3. You can't instantiate it.
4. Both static methods and constructors are possible.
5. It may have final methods that compel the subclass not to alter the method body.

Features of Abstraction in Java:

Template: By giving programmers the opportunity to conceal the code implementation, the abstract class in Java offers the best approach to carry out the process of data abstraction. It also provides a template that outlines the procedures to the user.

Unstable Coupling: Java's data abstraction facilitates loose coupling by exponentially decreasing dependencies.

Reusability of Code: It saves time to use an abstract class in the code. Anywhere that the abstract method is required, we may call it. Abstract classes eliminate the need to repeatedly write the same code.

Abstraction: By condensing the project's entire features to just those that are required, data abstraction in Java enables developers to conceal the code complexity from the end user.

Adaptive Resolution: Developers can use one abstract method to handle a number of problems with the assistance of dynamic method resolution.

Advantages of Abstraction:

The fundamental advantage of employing an abstraction in programming is that it enables to assemble as siblings a number of related classes.
In Object Oriented Programming, abstraction serves to simplify the program design and implementation process .

Constructors

To create something in our applications, we employ Java constructors or constructors in Java. A Java function Object () is a unique function used to initialize objects. When a class object is formed, the function Object () is invoked. It can be applied to give object properties their initial values. A function Object () is a piece of Java code that is comparable to a method. When a class instance is created, it is called. RAM for the object is allocated in the memory at the moment the function Object () is called. The method is a unique kind that is used to initialize the object. The new () keyword calls at least one function Object () each time an object is created. Because it builds the values when an object is created, it is termed a function Object (). A class does not require a function Object () to be written. It's because if your class doesn't have a function Object (), the Java compiler produces one .

Developing Java function Object () guidelines

The function Object () is subject to two rules.

1. Constructor names need to match class names exactly.
2. There can be no explicit return type in a function Object ().

Java constructors cannot be synchronized, static, final, or abstract.

Java constructors have the following characteristics:

The function Object () name is the same as the class name.

Explanation: Constructors are distinguished from other member functions of the class by the compiler using this character.
A function Object () cannot have any sort of return. It automatically calls and is frequently employed for initializing values, as explained. When an object is created for the class, the function Object () is automatically called. Contrary to C++ constructors, Java prohibits the declaration of a function Object () outside of a class. Typically, constructors are used to initialize all of the class's data members (variables). Additionally, a class may define many constructors with various arguments. 'Constructor overloading' is the name of it .

Types of Constructor in Java

There are three types of constructor in java.

Default Constructor: If no function Object () is explicitly stated in the class, a default function Object () is automatically built when an object is created. This kind of function Object () automatically creates without being defined and declared in the class; it is not visible in the application.

No-argument Constructor: Contrary to default constructors, no-argument constructors are accessible in the class. Additionally, a function Object () definition with no arguments is present (or parameters). When a class object is created, they are called. These constructors typically have variables defined for other member functions.

Parameterized Constructor: It has parameters (or arguments) in the function Object () definition and declaration, in contrast to No-argument function Object (). Java's parameterized constructors support passing multiple arguments.

Copy Constructor: A copy function Object () is a specific form of function Object () that makes use of another object from the same class to build a new object. It provides a duplicate of an already-created object in exchange .

Difference between method and constructor in Java:

The distinctions between constructors and methods are many. The list of them is below Table 1.2.

Table 1.2: distinctions between constructors and methods

S. No.	Java Constructor	Java Method
1	An object's state is initialized using a function Object ().	An object's behavior is exposed through a method.
2	Implicitly, the function Object () is used.	The approach is specifically called
3	A return type is not permitted in a function Object ().	A return type is necessary for a method.
4	The class name and the function Object () name must match.	The class name and the method name could or might not be the same.
5	If a class doesn't have a function Object (), the Java compiler offers a default function Object ().	The compiler in any situation does not supply the method.

Java Constructor Chaining

Calling one function Object () from another with regard to the current object is known as function Object () chaining. By using function Object () overloading, function Object () chaining is primarily used to reduce code duplication and improve readability when there are several constructors. Java constructors are required. Two approaches of function Object () chaining are possible:

Similar classes: For constructors in the same class, you may accomplish it by using the () keyword.**From base class:** Using the super () keyword, you may invoke the base class's function Object () from a subclass.

Through inheritance, function Object () chaining takes place. The function of a subclass function Object () is to first invoke the function Object () of the superclass. This makes sure that the superclass's data members are initialized before creating an object for a subclass. . Any number of classes might be included in the inheritance chain. Up the chain each function Object () calls until the top class is reached.

Need of constructor chaining: This method is used to do numerous jobs in a single function Object () rather than writing separate pieces of code for each duty in a single function Object () which improves readability of the program.

CHAPTER 2
CLASSIFICATION OF OBJECT ORIENTATED PROGRAMMING

Raghavendra R., Assistant Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- r.raghavendra@jainuniversity.ac.in

Our world and surroundings are continually changing due to computing. Large computers known as mainframes were developed in the 1960s to handle massive amounts of data (numbers) effectively. Programs for managing payroll and bank accounts altered how businesses operated and greatly increased the efficiency of some departments. Personal computers became widespread in the 1980s and transformed the way many people worked. People began to have personal computers, and many of them utilised word processing and spreadsheet software (to write letters and to manage home accounts). In the 1990s, email spread widely, and the internet was established. Technological innovations in technology have revolutionised communication by enabling people to publish information that is readily accessible on a global scale. While society adjusts to the advantages of internet commerce, new social networking tools (such as Twitter, Facebook, MySpace, and online gaming), and the difficulties of internet-related crime, the implications of these new technologies are still not completely understood.

The way we create computer systems is evolving in tandem with how new computing technologies are altering our reality. Machine code, which uses simple, complicated, and machine-specific languages, was widely used in the 1950s. High level languages, which simplified programming, were popular in the 1960s. Yet as a result, huge, complicated programmes that were challenging to administer and maintain were created.

The structured programming paradigm became the de facto industry norm for creating big, complicated computer programmes in the 1970s. The structured programming paradigm offered strategies for logically dividing up the generated programmes into distinct, smaller, and easier to manage components. Also, techniques for data analysis were put forward that enabled the creation of effective huge databases, avoided the wasteful duplication of data, and shielded us from the dangers of data being out of sync. Nonetheless, there were still considerable issues with a) comprehending the systems we needed to build and b) updating current software when consumers' needs changed.

The development of "modular" languages like Modula-2 and ADA in the 1980s served as the forerunner of contemporary Object Oriented languages. Component-based software development and the Object Oriented paradigm were created in the 1990s, and starting in 2000, Object Oriented languages became the standard. The object-oriented paradigm is based on many of the abstraction, encapsulation, generalisation, and polymorphism ideas that have been developed over the past 30 years. This has resulted in the creation of software components where the operation of the software and the data it operates on are modelled together. The argument put forth by proponents of the object-oriented software development paradigm is that it results in the creation of software components that can be reused in various applications, saving both time and money while, more importantly, enabling the creation of better software models that make systems easier to maintain and comprehend. Maybe it should be highlighted that new agile working techniques are being developed and tried, and software development

concepts are continuously developing. It remains to be seen where these will take us in 2020 and beyond.

Several Programming Methodologies

1. According to the structured programming paradigm, programmes might be created in logical pieces that would make them simpler to comprehend and maintain.
2. According to the object-oriented paradigms, we should characterise computer programmes' instructions and the data they manage as separate components and store them together. The creation of reusable software components is one benefit of doing this.
3. The concepts underlying the structured programming paradigm of the 1970s are expanded upon and built upon by the object-oriented paradigm.

An object-oriented approach:

Regardless of the development process chosen, we may concentrate our attention on the actual computer code that we are producing, but most issues don't originate with the code itself. Most issues result from:

Inadequate analysis and design: The computer system we develop doesn't act appropriately.

Poor maintainability: When, as is inevitable, demands for change emerge, the system is difficult to comprehend and modify.

According to statistics, 70% of the cost of software is spent after the original creation period when the programme is modified to satisfy the constantly changing demands of the organisation for which it was created. Software developers must take all necessary steps to make sure that their work is simple to maintain in the years after its original development for this reason. By assisting with the analysis and design duties during the early software development phase, and by ensuring software is reliable and maintainable, the object-oriented programming paradigm seeks to assist in resolving these issues.

Principles of Object-Orientation

The foundational ideas that guide the Object Oriented approach to software development are abstraction and encapsulation. By removing unnecessary details that might be confusing, abstraction enables us to think through complicated concepts. Encapsulation enables us to concentrate on what something accomplishes rather than worrying about the intricate details of how it operates.

Encapsulation: When each object retains a private state inside of a class, encapsulation is achieved. Other objects may only call one of a number of public functions; they cannot directly access this state. Using these functions, the object controls its own state; no other class may change it unless specifically permitted. You must use the offered ways in order to communicate with the object. I prefer to think about encapsulation in terms of people and their pets as an example. Encapsulation is a programming technique wherein all "dog" logic is contained inside a class. The private variables playful, hungry, and energy contain the dog's "state," and each of these variables has a corresponding field.

There is a secret technique as well: bark (). The other classes cannot dictate to the dog when to bark; the dog class is free to call this anytime it pleases. Moreover, other classes have access to public methods like sleep(), play(), and eat(). Each of these methods alters the Dog class's internal state and has the potential to call bark(); when this occurs, the link between the methods' private and public states is formed.

Abstraction: Encapsulation is a continuation of abstraction. It is the process of choosing information from a bigger pool such that the object only sees the relevant elements. Imagine you're requested to gather all the user data for a dating programme you want to develop. Your user may provide you with the following information: Name, address, phone number, favourite movie, cuisine, pastimes, tax information, social security number, and credit score are all included. While there is a lot of information here, not all of it is necessary to build a dating profile. From that pool, you simply need to choose the data that is relevant to your dating application. On a dating app, information like full name, favourite meal, favourite movie, and interests makes logical. Abstraction is the method of obtaining, deleting, or choosing user data from a bigger pool. Being able to utilise the same data you used for the dating application to other apps with little to no change is one of the benefits of Abstraction.

Inheritance: The capacity of one thing to acquire part or all of the attributes of another object is known as inheritance. For instance, a kid acquires the characteristics of his or her parents. Reusability is a significant benefit of inheritance. The current class's fields and methods may be used again. Java supports single, multiple, multilevel, hierarchical, and hybrid inheritances among other variations. Assume, for instance, that the class Fruit has a subclass called Apple since Apple is a fruit. The characteristics of the Fruit class are added to our Apple. Additional categories can include grape, pear, mango, etc. Fruit refers to a group of foods that are fleshy, ripe plant ovaries that may or may not contain a big seed. The subclass of Apple, which differs from other subclasses of Fruit by being red, spherical, and having a depression at the top, derives these characteristics from Fruit.

Polymorphism: We may utilise a class precisely like its parent thanks to polymorphism, eliminating any type mixing complications. Having said that, each child sub-class maintains its own functions and methods. If there was a method named mammalSound in the superclass Mammal, for example (). Mammals may be divided into subclasses that include dogs, whales, elephants, and horses. Each of them would produce a distinct animal sound. We may describe the general features and functions of an object via generalisation and then develop more specialised versions of that item. All of the traits of the more generalised object will be immediately inherited by the more specialised variants of this object.

The capacity to interact with an object as a member of its generic category rather than its more specialised category is known as polymorphism, and it is the last guiding principle of object orientation. Following the same principle, polymorphism enables the expansion of computer systems by enabling the creation of new, specialised objects. At the same time, it enables existing system components to interact with the new objects without having to take into account their unique qualities.

Programming with objects

You may even consider other intangible entities, like a bank account, to be objects. Despite the fact that a bank account cannot be physically touched, we may nevertheless think of it as an item. Also, it has a present state and behaviour connected to it. A programming approach known as "object oriented programming" entails the construction of intellectual objects that simulate a business issue we are attempting to solve (e.g. a bank account, a bank customer and a bank manager could all be objects in a computerised banking system). We represent the information (such as an item's state at any given moment) and behaviour associated with each object (what our computer programme should allow that object to do). In order to develop an object-oriented software, we first describe the attributes of a class of objects, from which we then generate individual objects. A ‘class’ is a software design which describes the general properties of something which the software is modelling. Individual ‘objects’ are created from the class design for each actual thing

However deciding just what classes we should create in our system is not a trivial task as the real world is complex and messy. In essence we need to create an abstract model of the real world that focuses on the essential aspects of a problem and ignores irrelevant complexities. For example in the real world bank account holders sometimes need to borrow money and occasionally their money may get stolen by a pick pocket. Using object-oriented analysis and design techniques our job would be to look at the real world and come up with a simplified abstract model that we could turn into a computer system. How good our final system is will depend upon how good our software model is shown in Figure 2.1.

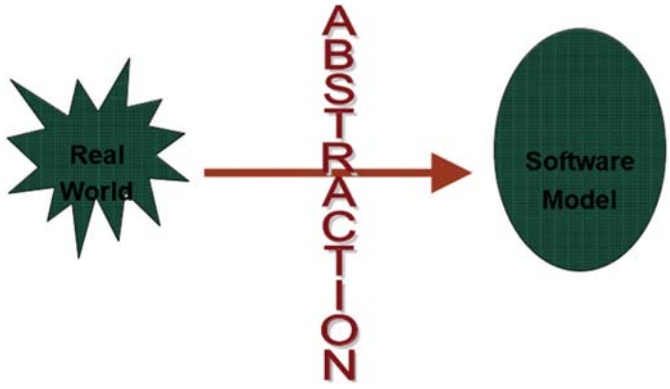


Figure 2.1: Represent the Abstractions

The Benefits of the Object Oriented Programming Approach

Whether or not you develop programs in an object oriented way, before you write the software you must first develop a model of what that software must be able to do and how it should work. Object oriented modelling is based on the ideas of abstraction, encapsulation, inheritance and polymorphism. The general proponents of the object oriented approach claims that this model provides:

1. Better abstractions (modelling information and behaviour together)
2. Better maintainability (more comprehensible, less fragile)
3. Better reusability (classes as encapsulated components)

Object oriented programming involves the creation of classes by modelling the real world. This allows more specialised classes to be created that inherit the behaviour of the generalised classes. Polymorphic behaviour means that systems can be changed, as business needs change, by adding new specialised classes and these classes can be accessed by the rest of the system without any regard to their specialised behaviour and without changing other parts of the current system.

Operator overloading or function overloading are both examples of overloading.

It is capable of expressing the addition operation with a single operator, such as "+." When this is feasible, you may represent the sum of x and y using the equation x + y, which works for a variety of x and y types, including integers, floats, and complex numbers. Even better, you may describe the + operation for two strings as simply joining the strings together.

Physical binding:

The term "binding" describes the connection between a procedure call and the code that is run in response to the call. With dynamic binding, the code connected to a specific procedure call

is unknown until the call is made at run-time. Depending on the dynamic type of the reference, it is connected to a polymorphic reference.

Passing of message:

A collection of interconnected items make up an object-oriented application. A message for an object requests the execution of a procedure, which causes the receiving object to call a function (procedure) that produces the desired outcome. The name of the object, the name of the function (message), and the content to be transmitted must all be specified when sending a message.

Advantages of OOP:

Oop has a number of advantages for both users and programme designers. Several issues relating to the creation and calibre of software products are solved thanks to object-oriented programming. The following are the main benefits:

1. We may expand the usage of existing classes and remove superfluous code using inheritance.

2. Rather of needing to start from zero while developing the code, we may construct programmes from the common working modules that interact with one another. This results in reduced development time and increased production.

3. By using the data hiding concept, programmers may create safe programmes that cannot be compromised by code in other sections of the programme.

4. It is conceivable for several instances of the same thing to coexist peacefully.

5. Dividing a project's workload based on items is simple.

6. It is simple to update object-oriented systems from small to big systems.

7. Using message passing mechanisms for object communication makes describing the interface to other systems much easier.

8. It is simple to control software complexity.

OOP Application:

Up until now, the most widely used use of oops has been in the field of window-style user interfaces. Many windowing systems have been created utilising oop principles. The complexity and number of objects with sophisticated properties and functions in real-world business systems are often significantly higher. Oop is helpful in these kinds of applications since it may make a difficult issue simpler. The potential applications for oop include.

1. Real-time computing.
2. Modeling and simulation
3. Databases that are object-oriented.
4. Expertext, hypertext, and hypermedia
5. Expert systems and Al.
6. Parallel computing and neural networks.
7. Office automation and decision support systems.
8. A CAD, CAM, and CIM system.

CHAPTER 3

THE UNIFIED MODELLING LANGUAGE (UML)

D Janet Ramya, Assistant Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- d.janet@jainuniversity.ac.in

A methodology is a set of procedures used to accomplish a certain goal, such as a planned series of tasks to collect user needs. On the other side, UML is a precise diagramming notation that enables the representation and discussion of software designs. The information offered in the diagram is simple to see, comprehend, and debate due to its pictorial character. There is a certain syntax that must be followed for the diagrams to function since they must be clear and exact because they reflect technical information. As UML is not a technique, it is up to the user to utilise the procedures they believe necessary to produce the designs shown by the diagrams. This is not restricted by UML; rather, it only enables the expression of such designs in a simple but accurate graphical language.

Diagrams of UML classes

Each object-oriented software system's fundamental building blocks are classes, and UML class diagrams provide them a simple method to be represented. Class diagrams depict many classes and their relationships in depth in addition to displaying individual classes. Thus, a class diagram depicts a system's architecture.

Constitutes a class

1. a special name (conventionally starting with an uppercase letter)
2. a list of qualities (int, double, boolean, String etc)
3. a list of techniques

This is shown using a basic box structure (Figure 3.1).

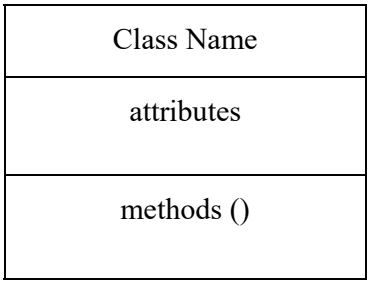


Figure 3.1: Basic Box Structure

Visibility modifiers (+ for public access, - for private access) are shown for attributes and methods. Methods are often made public, but attributes are typically kept private. Of course, almost all Java projects will consist of a variety of classes, and classes will interact with one another and use other classes. Arrows are used to illustrate these connections. Various arrow types signify various connections (including inheritance and aggregation relationships). Class diagrams may also include keywords, notes, and comments.

The examples that follow will demonstrate how a class diagram may display the information below:-

a. Classes: attributes, operations, visibility
b. Relationships: navigability, multiplicity, dependency, aggregation, composition
c. Inheritance, generalisation, and specialisation, as well as interfaces
d. Notes and Comments
e. Keywords

UML Syntax

The syntax for UML diagrams must be used precisely since they offer exact information. Attributes need to be shown as:

Visibility name: type multiplicity

Where visibility is one of:-

1. '+' public
2. '-' private
3. '#' protected
4. '~' package

Multiplicity is one of:-

1. 'n' exactly n
2. '*' zero or more
3. 'm..'n' between m and n

The following are examples of attributes correctly specified using UML: -

custRef: int

a private attribute custRef is a single int value

This would often be shown as – custRef: int However with no multiplicity shown we cannot safely assume a multiplicity of one was intended by the author.

itemCodes: String

a protected attribute itemCodes is one or more String values

validCard: boolean

ValidCard is an attribute with undetermined visibility and unspecified multiplicity. Moreover, operations have a precise syntax that is shown as:

visibility name (par1: type1, par2: type2): returntype

When the return type is provided after each argument is shown (in parentheses). As an example, consider + addName (newName: String): Boolean. This designates a public method called "addName" that accepts a single String-type argument named "newName" and returns a boolean result.

Making Relationships Known

Class diagrams show links between classes as well as individual classes. An "Association" is one of these connections.

Attributes will be specified in a class. Attributes might be complicated objects as described by other classes or they can be simple data types (int, boolean, etc.). Therefore, a class called "OneClass" with the property "value" is shown in the above image. This value is an object of

the 'OtherClass' type, not a basic data type (Figure 3.2 and Figure 3.3).

The figure below might be used to represent the exact same data.

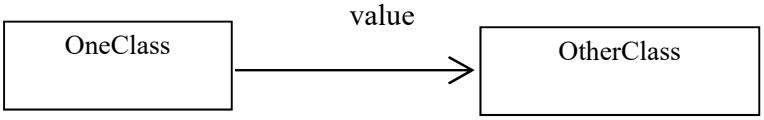


Figure 3.2: Illustrate the connection between the one class and other class with the help of value.

When we wish to emphasize the link between two related classes on a class diagram, we utilise an association.

The "target" class is referenced by the "source" class. The String class is a component of the Java platform and is "taken for granted" like an attribute of a basic type, so technically we might utilise an association when a class we design contains a String instance variable. Nevertheless, we would not do this. Unless we are particularly creating the diagram to explain some feature of the library class for the benefit of someone unfamiliar with its purpose and operation, this would typically be the case for all library classes. Moreover, we may demonstrate many connections at both ends of a chain:

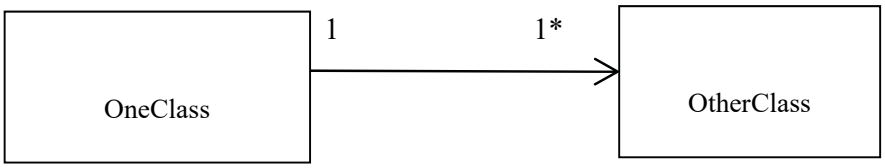


Figure 3.3: Illustrate the connection between the one class and other class with the help of value 1.

This suggests that "OneClass" keeps a set of objects of the "OtherClass" type. Collections play a key role in the Java architecture Figure 3.4.

Figure 3.4: Illustrate the connection between the one class and other class with the help of value 0 and 1.

Notice that although attribute and method names start in lowercase, class names always do. Keep in mind that the class ItemForSale only refers to one object (not multiple items). But, 'listOfItems' keeps a list of 0 or more distinct objects.

Several Associations

Several distinct sorts of relationship are shown by various arrows, including the following:

1. Dependence
2. Simple association
3. Bidirectional association
4. Composition
5. Aggregation
6. Dependency

Dependency: A relationship between two classes is referred to as a dependency when one class relies on another class but another class may or may not also rely on the first class. Since the operation of one class relies on the others, every modification to one may have an impact on the others.

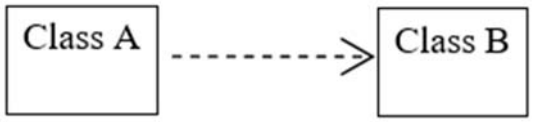


Figure 3.5: Represent the Dependency

Class A utilises facilities that are specified by Class B in some manner. Class B changes may have an impact on Class A. Dependence is the least defined connection between classes (not precisely a "association"). Dependency lines are often used when a method in Class A receives an object from Class B as an argument, utilises a local variable from that class, or invokes static methods in Class B (Figure 3.6).

Simple Association

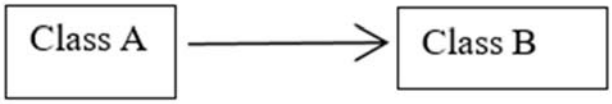
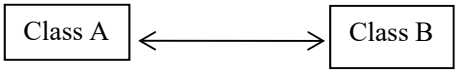


Figure 3.6: Simple Association

a. In an association, Class A "uses" Class B objects.
b. Usually, Class A possesses a property that belongs to Class B.
c. Navigation from point A to point B:

For example, a Class A object may interact with any Class B objects that it is linked to. Contrary to popular belief, the Class B item is unaware of the Class A object. An instance variable in Class A of the target class B type often equates to a straightforward relationship. For instance, in order for products to be added to or deleted from the catalogue above, access to 0 or more ItemsForSale is required. A Catalogue is not required in order to establish an ItemForSale's pricing or carry out any other actions related to the item itself .

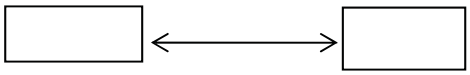
Bidirectional Association



Classes A and B may refer to each other in a bidirectional connection, which occurs when there is a two-way relationship between them.

1. A to B and B to A navigation:
2. A Class A object has access to any connected Class B object(s).
3. Class B object(s) "belong to" Class A, implying a reference from A to B
4. Moreover, a Class B object has access to any connected Class A object(s).

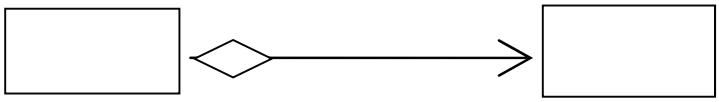
Due to the need that each item have a reference to the other object or objects, bidirectional relationships are more difficult to create than unidirectional ones.



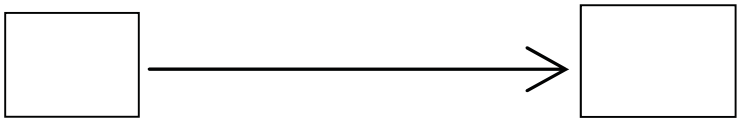
A degree and a student might be two examples of a bidirectional relationship. Specifically, we could be interested in learning which Students are enrolled in a specific degree. Instead, if we were to start with a student, we may ask what degree they are pursuing.

There is still a one to many connection since many students are pursuing the same degree at the same time, but most students only enrol in one degree.

Aggregation



Aggregation refers to a circumstance in which one or more objects of Class B "belong to" Class A. Infers a link from A to B. Aggregation indicates that items of Class B are related to those of Class A, but it also implies that Class B objects continue to exist separately from Class A. According to some designers, there is little difference between basic association and aggregation. A Class Vehicle with a Class Tyre would be an illustration of aggregation.

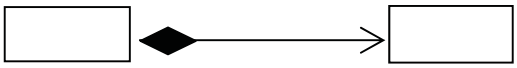


We often think of tyres as being attached to the vehicle they are on, but at the garage, they could be taken off and arranged on a rack for repair. Their existence is independent of the presence of the automobile they are linked to.

Composition:

A kind of connection that is more strongly connected in Java is called composition. Strong association is another name for composition in Java. This relationship is sometimes referred to as a "belongs-to" association since it exists because one class effectively belongs to another

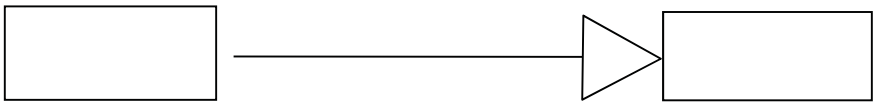
class. The classes in a Composition relationship cannot exist apart from one another. It follows logically that the smaller class cannot exist if the bigger class that contains the objects of the smaller class is eliminated.



In this case, composition is much "stronger" than aggregation because Class B objects are an integral part of Class A, and in general, Class B objects never exist other than as part of Class A, i.e., they have the same "lifetime." Composition is similar to aggregation but implies a much stronger belonging relationship, i.e., Object(s) of Class B are "part of" a Class A object. Points, Lines, and Forms as Image components are an example of composition. These items can only be found in photographs, and if the photograph is removed, so are they. Class diagrams are used to represent associations as well as the following:

Interfaces, Keywords, Inheritance, and Notes

Inheritance



In addition to connections, inheritance is another key modelling relationship:

Class: A "inheres" Class B's implementation and interface, while it is free to replace or enhance either

Interfaces: Interfaces are comparable to inheritance in that just the interface is passed down in interfaces. Any class that implements the interface must implement the methods specified by the interface. Interfaces can be represented using the <<interface>> keyword:

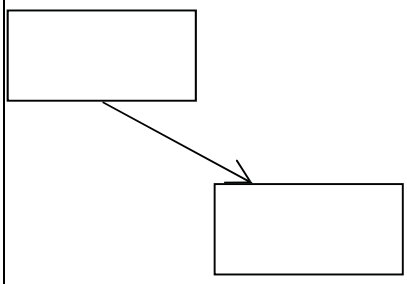
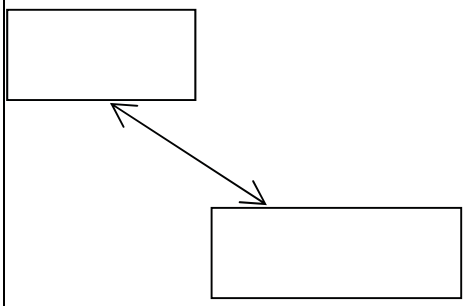
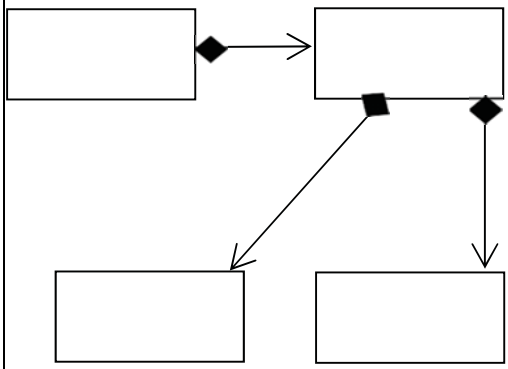
These examples show that the SaleableItem interface is necessary for CashTill in both situations and is implemented by Publication. Observe that the inheritance line/arrow is shown with a dotted line to indicate that Publication "implements" or "realises" the SaleableItem interface. The "ball and socket" notation, which is new in UML 2, is a useful visual representation of how interfaces link classes. In order to clarify the meaning of the graphical symbols, UML introduces keywords. There are other more, but we have already seen interface and will utilise abstract. Italicizing the name of an abstract class is another way to indicate it, however a casual reader may find this less evident. We may now add comments to a diagram element to provide commentary (Table 3.1).

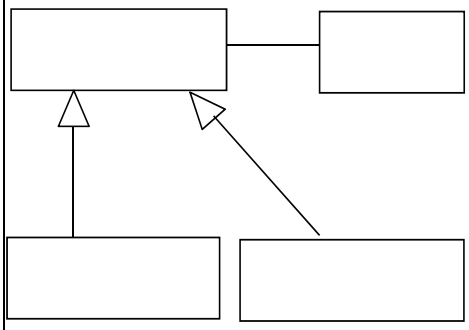
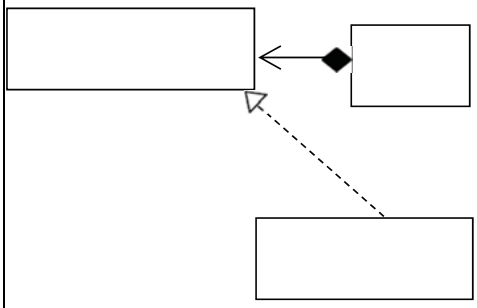
UML package diagrams:

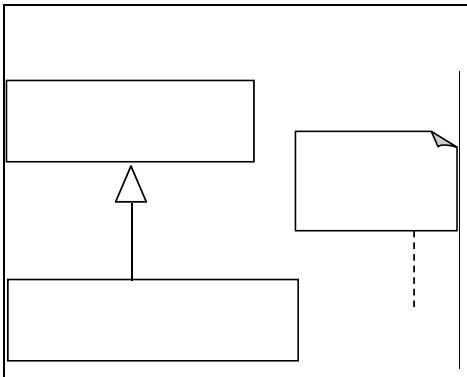
Although class diagrams are the UML notation diagram that is most often used, and we will use them extensively in this book, there are other diagrams that represent other sorts of information. Here, discuss three of them:

1. Packaging Schematics
2. Sequence Diagrams

Table 3.1: This provides us with a "catch all" tool for adding information that the pictorial notation is unable to communicate.

Catch all tools	**Descriptions**
	In a University administration system we might produce atranscript of results for each year the student has studied (including a possible placement year). This association relationship is naturally unidirectional – given a student we might want to find their transcript(s), but it seems unlikely that we would have a transcript and need to find the student to whom it belonged.
	In a library a reader can borrow up to eight books. A particular book can be borrowed by at most one reader. We might want a bidirectional relationship as shown here because, in addition to being able to identify all the bookswhich a particular reader has borrowed, we might want tofind the reader who has borrowed a particular book (for example to recall it in the event of a reservation).
	This might be part of the model for some kind of educational virtual anatomy program. Composition – the “strong” relationship which shows that one object is (and has to be) part of another seems appropriate here. The multiplicities would not always work for real people though – they might have lost a finger due to accident or disease, or have an extra one because of a genetic anomaly. And what if we were modelling the “materials” in a medical school anatomy lab? A hand might not always be part of

	a body! Perhaps the "weaker" aggregation relationship would reflect this better.
	A customer can have any number of bank accounts, anda bank account can be held by one person or two people (a "joint account"). We have suppressed the navigability of this relationship, perhaps because we have not yet decidedthis issue. A bank account **must** either be a current account or a savings account – hence BankAccount itself is abstract. (We could have shown this using italics rather than the <<abstract>> keyword) *Bank Account*
	Part of a clock is a display to show the time. This might be an analogue display or a digital display. We could use a superclass and two subclasses, but since the Implementation of the two displays will be entirely different it may be more appropriate to use an interface to define the operations which AnalogDisplay and DigitalDisplay must provide.

	A note is used here to add some information which cannot be conveyed with standard UML notation and may not be obvious simply from the names and relationships of the classes depicted.

Object Diagrams

Maps of the world, countries, and cities all display geographical information, but at varying sizes and degrees of detail. If the class diagram were the only method to depict the architecture of a huge system, it would become excessively vast and complicated since large OO systems may have hundreds or even thousands of classes. So, to illustrate the overall architecture of a big system, package diagrams are required, just as we need globe maps. Even small systems may be divided into a few fundamental parts, or packages. A "package" is a directory that contains a set of linked classes rather than merely being a visual depiction of a group of classes (and interfaces). All of the standard Java platform classes are organised in a single, massive package hierarchy, giving packages the ability to offer a degree of organisation and encapsulation above that of individual classes. Similar to how we may organise our own classes using Java packages. Packages are identified by a list of names separated by commas, such as java.awt.event. The names are associated with a number of file system subdirectories, including as (Figure 3.7).

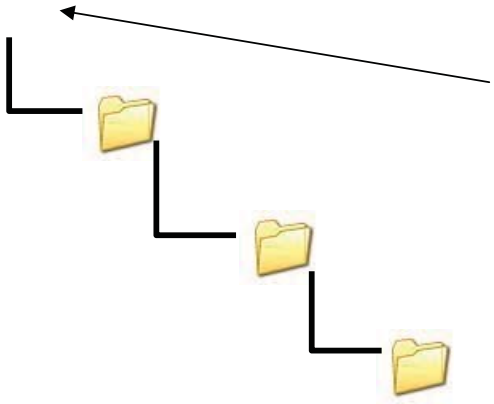


Figure 3.7: Package Diagram

During the design stage, a big Java development should be divided into appropriate packages. To depict the links between classes and packages, UML offers a "Package Diagram." Classes within packages; Package nesting; Package interdependencies Java and Javax are represented as two packages in the figure below (Figure 3.8).

When we take a closer look at this, we can see that the "java" package contains the "awt" package, and the "javax" package contains the "swing" package. Container is a class in the package "awt," while JFame, JComponent, and JButton are three classes in the package "javax." At the conclusion, we demonstrate how the java.awt package is dependent on the javax.swing package.

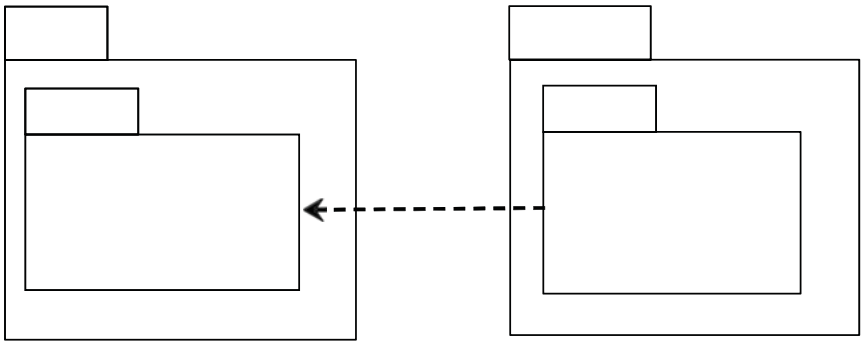
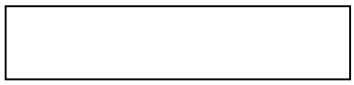


Figure 3.8: Java and Javax

Notice that both java.awt and javax are affected by the usual UML concept of suppression in this situation.

The reason we decided not to display them is that swing has many more classes and that 'java' contains other sub-packages.

An alternate method of showing that a JButton is in the javax. swing package is shown in the below.



Again, here is a form that presents all three classes in a more compact manner than it does at the top. Depending on the message a package diagram is trying to communicate, these various representations will be beneficial in various situations.

Naming a package

1. Package names are often lowercase as a matter of convention
2. Individual packages for local projects might be given names that suit the user, e.g.
3. mysystem
4. mysystem.interface
5. mysystem.engine
6. mysystem.database mysystem.engine.util

Nevertheless, since packages must have globally unique names in order to be delivered, a naming system based on URLs has been implemented. Keep in mind that each element on a package diagram is separated by a '::' rather than a '.' Object Diagrams in UML: Class diagrams and package diagrams help us visualise and discuss a system's design, but there are instances when we also want to talk about the data that a system handles. With object diagrams, we can see a certain instant in time and the data that a system could be holding at that time.

Object diagrams superficially resemble class diagrams, but the boxes in an object diagram represent particular instances of things.

Box titles include:

className: objectName

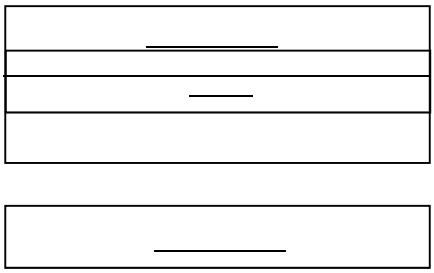
Each box comprises properties and their values as it depicts a certain item at a given time (at

that moment in time).

value = attribute

For depicting certain "snapshot" circumstances during design, these diagrams are helpful. The object diagram that follows illustrates many items that could exist right now for a library catalogue system. The system has two classes: Book and Library, which keep track of a book collection and store information about books, respectively. with books being added, sought out, or eliminated as necessary.

Looking at this picture, we can observe that at a certain point in time, only two books have been added to the library while three have been produced. So, we wouldn't anticipate finding the book "Cooking for Beginners" if we searched the library. Elements on object diagrams may be arbitrarily suppressed, just as in class diagrams. For instance, either of these is acceptable:



UML Sequence Diagrams:

Class diagrams and object diagrams are completely unrelated to sequence diagrams. Class diagrams illustrate a system's architecture, whereas object diagrams show the system's current state. Sequence diagrams, on the other hand, show the system's behaviour across time. Sequence diagrams show the system in a "dynamic" manner as opposed to a "static" one. They display the order of method calls made inside and between objects over time. They are helpful for comprehending how several items work together in a certain situation. Three items are present in this situation. The vertical dashed lines (lifelines) represent the items' continuity over time as time moves from top to bottom. The following actions are shown in this figure as they happen:

1. To begin with, object1 receives a method call (commonly referred to as a message in OO language) to method0() from elsewhere; this might be a different class outside of the diagram.
2. The vertical bar, known as an activation bar, starts at this point, indicating that object1 has started performing method0().
3. The activation bar shows that object1.method0() calls object2.method1(), which operates for a while before returning control to method0 ()
4. After that, object1.method0() calls object2.method2() with two arguments.
5. method2() then calls object3.method3 (). After method3() is finished, it returns a value to method2 ()
6. Once method2() has finished, object1.method0 is given control ()
7. Lastly, method0() invokes method4 on the same object ()

Choice and Repetition

1. Selection ('if') and iteration are often used to determine a scenario's logic (loops).
2. Ifs and loops may be described in sequence diagrams using a notation called "interaction frames," although this can make the diagrams seem crowded.
3. Sequence diagrams are often best utilised to illustrate specific scenarios, with the actual code being the only place where further refinement is possible.
4. Fowler provides a short analysis of these notions in "UML Distilled," 3rd Edn.

UML is a precise diagramming notation, not a technique

Class diagrams and package diagrams are useful for outlining a system's architecture. Sequence diagrams show how a system operates across time, while object diagrams show the data in an application at any one time.

As various arrows in UML have distinct meanings, it is important to utilise the notation exactly as it is intended. Suppression is encouraged in all UML diagrams, therefore the creator of a diagram is free to exclude whatever information they want in order to communicate the most important information to the viewer.

CHAPTER 4
INHERITANCE AND METHOD OVERRIDING

Bhuvana J, Associate Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- j.bhuvana@jainuniversity.ac.in

A class can acquire the attributes (methods and fields) of another class through a process known as inheritance. Information may be managed in a hierarchical manner with the help of inheritance. Subclass (derived class, child class) and Superclass (class from which properties are inherited) are terms used to describe classes that share properties (base class, parent class). Use extends keyword to inherit from a class. Inheritance One class can gain a property from another class using the Java Inheritance mechanism. In Java, we utilise inheritance when there is a "Is-A" link between two classes. The inherited class is referred to as a subclass, while the parent class is referred to as a super class. The subclass uses the term extends to inherit the properties from the superclass. The ability to reuse code is made possible through inheritance.

Need of Inheritance:

There are two primary reasons why we require inheritance:

Run-Time Polymorphism: A method call in the execution process that overrides another method in the run time is referred to as "runtime," also known as "dynamic polymorphism."

Reusability of Code: The methods and data elements declared in the parent class are reused during the inheritance process. The necessity to write the identical code in the child class is removed through inheritance, saving time.

Important terminologies used in Inheritance:

Class: Objects that belong to the same class will all have similar traits, behaviors, and attributes. There is no such thing as a class in the real world. In order to produce items, it only serves as a model, blueprint, or prototype.

Superclass/Parent Class: A superclass is a class whose characteristics are inherited.

Sub Class/Child Class: Class that inherits from another class is referred to as a subclass, or "child class". Along with the attributes and methods provided by the superclass, the subclass can also add additional fields and methods.

Reusability: The idea of "reusability" is supported by inheritance; for example, if we want to construct a new class but an existing class already has part of the code we need, we may derive our new class from the old class. Utilizing the methods and fields of the pre-existing class by doing this .

Types of Inheritance:

Single Inheritance: A class inherits the attributes of another class through single inheritance. It makes it possible for derived classes to take behavior and attributes from a single parent class. This will therefore make it possible to reuse current code and give it new functionalities.

Here, Class A serves as the parent class while Class B, the child class, inherits the traits and characteristics of the parent class. The code following illustrates a comparable idea:

Multi-level Inheritance: Multilevel inheritance is the process of deriving a class from another class that itself was descended from another class, i.e., a class has more than one parent class but descended from them at various levels. Take an example, class B inherits the traits and conduct of class A, whereas class C inherits those of class B. Here, class A is the parent class of class B, which in turn is the parent class of class C. In this instance, class C automatically inherits class A's methods and attributes together with class B's. Inheritance at several levels is what it is.

Hierarchical Inheritance: When several subclasses derive from one main class, this type of inheritance is known as hierarchical inheritance in Java. Hierarchical inheritance refers to a blending of several inheritance types. It is distinct from multilevel inheritance since several classes are descended from a single superclass. These more recent classes follow this superclass, inheriting its attributes, methods, etc. Dynamic polymorphism and code reuse are made feasible by this process (method overriding). Take the parent class Car, for instance. Audi, BMW, and Mercedes are now considered to be kid classes. Classes Audi, BMW, and Mercedes all extend class Car under Java's hierarchical inheritance model.

Hybrid Inheritance: Inheritance is combined with hybrid inheritance in Java. Multiple inheritance types are seen in this form of inheritance. This sort of inheritance is referred to as hybrid inheritance, for instance, if class A and class B extend class C and another class D extends class A. Multiple inheritance and single inheritance are both included in a hybrid inheritance. Java does not support multiple inheritance, hence interfaces are the only way to do it. It blends simple, many, and hierarchical inheritances in its essence .

Access Modifiers: An accessible parent class is specified by access modifiers. We can't let kid classes access all of the other classes when we're doing real-time coding. There are four different ways to express whether a data member, method, or function Object () is available.

Default: When the user does not give a particular access modifier for a class, the default access modifier is an option that automatically makes a class accessible. As with the public access modifier, the default access specifier is similar. The distinction is that the parent class may only be accessed from within the package that is now active for the Java project, not from outside of it.

Public: When a consumer wants all child classes to have accessibility to the parent class from any location in the Java project as well as any other packages, they utilize the public access modifier.

Protected: Different from the other modifiers is the protected access modifier. Only its child class has access to this class. Accessing the protected class's offspring class is the sole option if a user needs to utilize the protected data members and methods from a different class.

Private: For accessing members of and methods on private data, the private access modifier has stringent guidelines. Access is granted within the class but not outside of it thanks to the private access specifier .

Polymorphism in Java

Java's polymorphism idea allows us to carry out a single operation in several ways. Greek terms poly and morphs are the roots of the word polymorphism. Poly means numerous, and morphs implies forms. Polymorphism entails diversity of forms. Compile-time polymorphism and runtime polymorphism are the two forms of polymorphism used in Java. By using method overloading and method overriding, may implement polymorphism in Java. The capacity of an

item to assume several forms is known as polymorphism. Polymorphism in Java refers to a class's capacity to offer several implementations of a method based on the kind of object it receives as a parameter. Simply told, polymorphism in Java enables us to carry out the same activity in a variety of ways. A Java object is considered polymorphic if it can pass several IS-A tests. Given that the IS-A test for both their own type and the class Object was passed, all Java objects are polymorphic.

Compile-Time Polymorphism: Compile-Time Polymorphism is a phenomenon that occurs during the compilation phase of a typical Java application. In this case, the compilation step is where the overloaded method resolution happens. Two instances of compile-time polymorphism exist, and they are

Java Method Overriding: If both the superclass and the subclass include the same function during inheritance in Java. The method in the subclass then replaces the corresponding method in the superclass. Method overriding is the term for this.

Operator Overloading: When a class contains two or more methods with the same name, this is known as overloading. However, the number of arguments in the method specification determine how the given method is implemented. To prevent ambiguity, Java does not enable Operator Overloading.

Run-Time Polymorphism: Run-Time When using polymorphism, the program is executed while it is running. An override is resolved in this instance at the execution phase. The term "run-time polymorphism" is used twice.

Overriding Method: with Method Through a process called overriding, the compiler can let a child class to implement a certain method already present in the parent class.

Operator Overriding: An operator can be defined in both parent and child classes with the same signature but a distinct operational capability using the operator overriding process. Operator overriding, which eliminates ambiguity, is not supported by Java.

Static Polymorphism: Multiple methods can be incorporated into a class thanks to Java polymorphism. Despite having the same name, the techniques' parameters differ. This is an example of static polymorphism. Through the use of method overloading, this polymorphism is resolved at build time. Build-time polymorphism Java's static polymorphism determines which method to call at compile time. For the same trigger with static polymorphism, the object might respond differently. For this reason, different methods are included in the same class.

The number of parameters should change.

Different parameter types ought to be used. Various parameter orders. For instance, suppose one method only accepts a long, but the other function also accepts a string and a long. However, the API finds it challenging to comprehend this kind of arrangement. Every method has a distinct signature since the arguments vary. The method that is called is known to the Java compiler.

Dynamic Polymorphism: The method that will be called doesn't depend on the compiler when using this type of polymorphism in Java. The Java Virtual Machine (JVM) is responsible for carrying out the operation during runtime. In Java, the process where a call to an overridden process is resolved at runtime is referred to as dynamic polymorphism. A superclass's reference variable invokes the override method. Dynamic polymorphism, as the name implies, occurs between various classes as opposed to static polymorphism. Run-time polymorphism, which is at the foundation of dynamic polymorphism, makes it easier to override methods in Java. Before comprehending the idea of run-time polymorphism, it is essential to grasp the Up-

casting process. Up casting is the procedure when a reference variable from the superclass is used to refer to an object of the child class .

Cohesion: Cohesion is a measure of how well a module fits together. It is the intra-module idea. There are several degrees of cohesion, but typically, software benefits from strong cohesiveness.

Coupling: Coupling also serves as a symbol for the connections between modules. It is the inter-module idea. There are several sorts of coupling, but generally speaking, software benefits from minimal coupling (Table 4.1).

Table 4.1: Difference between the Coupling and Cohesion:

S. No.	Coupling	Cohesion
1.	Inter-Module Binding is yet another name for coupling.	Additionally known as intra-module binding, cohesion.
2.	Modules are connected to one another during coupling.	The module focuses on just one element in cohesiveness.
3.	The relationships between modules are displayed through coupling.	The link between the modules is shown by cohesion.
4.	When constructing, you should strive for low coupling, which means that there should be less reliance between modules.	When developing, you should strive for high cohesion, which is defined as a component or module that concentrates on a single task (i.e., is single-minded) and interacts with other system modules sparingly.
5.	The relative independence of the modules is demonstrated through coupling.	The module's cohesion reveals its relative functional strength .

Several different types of items may be classified as belonging to one or more 'families'. The concept of "inheritance," which refers to the notion of "passing down" traits from parent to kid, is crucial to object-oriented design and programming. While access control (public/private) and constructors are undoubtedly concepts you are already aware with, there are specific problems when connecting them to inheritance. Method overriding and the usage of abstract classes must also be taken into account since they are crucial ideas in the context of inheritance. We will examine the "Object" class, which plays a unique function in regard to all other Java classes, last.

Overgeneralization and Underspecialization

A generic form from which objects with various features may be constructed is a class. Hence, objects are "instances" of their class. Student, for instance, is an instance of the class Student. Simply said, 051234567 is a student.

These types of relationships may often be used to organise the classes themselves.

The hierarchy that depicts the animal world is one that most of us are acquainted with:-

1. Kingdom -(e.g. animals)

2. Phylum -(e.g. vertebrates)
3. Class -(e.g. mammal)
4. Order -(e.g. carnivore)
5. Family -(e.g. cat)
6. Genus -(e.g. felix)
7. Species -(e.g. felix leo)

We can represent this hierarchy graphically (Figure 4.1):

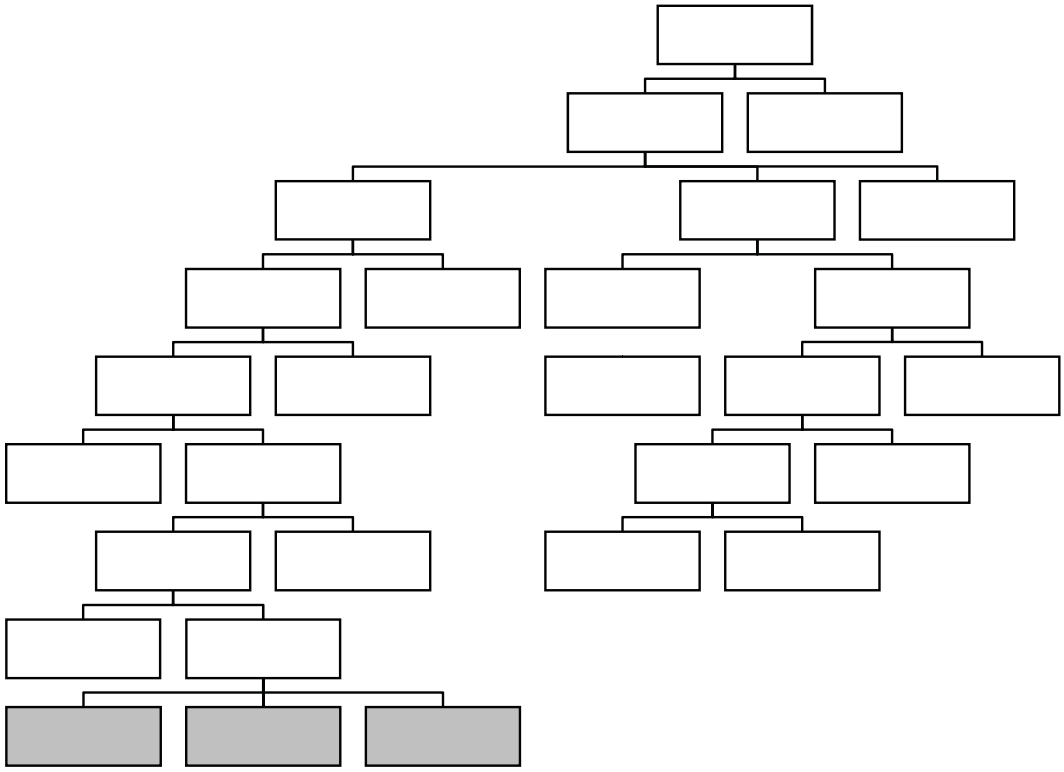
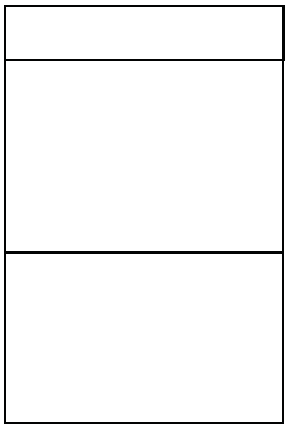


Figure 4.1: hierarchy graphically

Of course, it would take more time and room than we have available to create the whole diagram. One particular animal is shown here, and his name is "Fred." Fred is a genuine mammal, not a subspecies. Felix is Fred. Leo is a carnivorous cat named Felix. Fred has the trait "eats meat" because carnivores consume flesh. Felix is Fred. Leo is a felix, a cat, a mammal, a vertebrate, and a carnivore. As vertebrates have backbones, Fred possesses the quality of having a backbone. The "is a" connection connects a person to a hierarchy of qualities. There are several real-world entities that have this kind of interaction, including as Both a bank account and a savings account are BonusSuperSaver.

Inheritance: In the hierarchy, we define more broad features at the top and more particular traits at the bottom. Generalization and specialisation are key concepts in object-oriented programming (OO). We refer to this automatic inclusion of all the traits from classes above a class or object in the hierarchy as inheritance. Take into account books and periodicals, two distinct categories of media. Using a UML class diagram, we may demonstrate classes that represent these. As a result, some of the instance variables and methods that these classes may have are visible.



Title, author, and price are apparent attributes. Less evident is "copies," which indicates the quantity that is presently on hand. The function orderCopies() for books requires an argument that specifies the quantity to be added to stock. OrderQty is the quantity of copies of each new issue of a magazine received, and currIssue is the month or season of the most recent issue. OrderQty copies are added to stock when a new issue is received, and the old ones are deleted. RecvNewIssue() thereby restores copies to orderQty and sets currIssue to the date of the new issue. To change how many copies of upcoming issues will be stored, adjustQty() alters orderQty. These classes' common components may be separated out (or "factored out") and placed in a superclass called Publishing. For the "subclasses" Book and Magazine, the variations must be listed as extra members.

Java Implementing Inheritance

A superclass may be created without any unique characteristics. This means that unless explicitly prohibited, any class may be a superclass. By employing the term extends, a subclass indicates that it is deriving features from a superclass. For instance:

```
class MySubclass extends MySuperclass
{
      // additional instance variables and
      // additional methods
```

Constructors: Every class, whether it is a subclass or a superclass, should have its own initialization, which often involves setting the initial value of its instance variables. A superclass's function Object() { } should handle generic initialization. Each subclass may have a separate function Object() { } for customised initialization, however these constructors often follow the actions of their superclass counterparts. This is achieved by utilising the term super.

```
class MySubClass extends MySuperClass
{
      public MySubClass (sub-parameters)
      {
            super(super-parameters);
            // other initialization
```

This must be the first sentence in the function Object() { } if super, the superclass function Object() { }, is invoked.

The superclass function Object() { } will often get some of the inputs supplied to MySubClass as initializer values for superclass instance variables, and this will happen frequently. In other words, some (or all) of the sub-parameters will be super-parameters.

Two constructors, one for the Publication class and one for the Book class, are shown below. The superclass function Object() { } receives three of the four arguments the book function Object() { } needs to initialise its instance variables right away.

```
public Publication (String pTitle, double pPrice, int pCopies)
{
      title = pTitle;
      // etc.
}
public Book (String pTitle, String pAuthor, double pPrice,
                                                    int pCopies)
{
      super(pTitle, pPrice, pCopies);
      author = pAuthor;
```

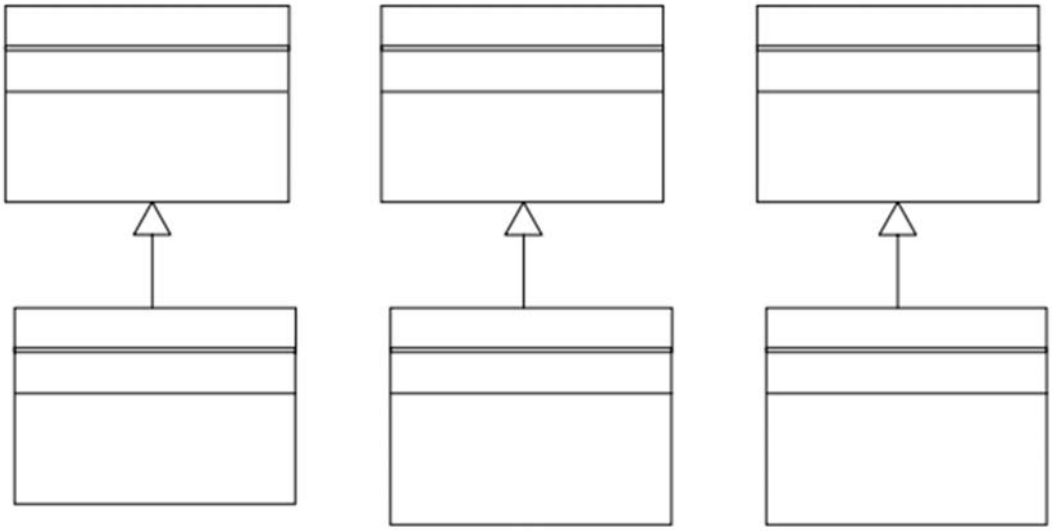
Constructor Rules: There are rules that control how a superconstructor is invoked.

While an explicit call to super is still preferable for clarity reasons, if the superclass has a parameterless (or default) function Object() { }, this will be called automatically if no call to super is made in the subclass function Object() { }.

Nevertheless, if the superclass contains a parameterized function Object() { } but no parameterless ones, then super must be used to explicitly invoke the parameterized function Object() { }. To clarify this:

The upper left: While it is bad practise, having a subclass without a parameter is permissible since the superclass provides a method Object() without any arguments in native code. The parameterless superclass function Object() { } will be invoked if the subclass function Object() { } doesn't call super.

On the right: Subclasses are required to have constructors that call super since the superclass lacks a parameterless function Object() { }. This is due to the fact that a (super) class with a single parameterized function Object() { } may only be instantiated by supplying the necessary argument (s).



Access Management

We often keep instance variables private and provide accessor/mutator methods when required to guarantee encapsulation. If the value of the variable "copies" has to be changed, the sellCopy() function of Publishing may do it even if "copies" is a private variable. Yet both books and magazines call for altered "copies."

There are two options for doing this.

To make copies accessible to subclasses, either 1) make "copies" "protected" rather than "private," or 2) provide accessor and mutator methods. While though protected may be beneficial to enable subclasses to utilise methods (such as accessors and mutators) that we would not want widely exposed to other classes, we normally prefer to build accessors/mutators for variables rather than sacrifice encapsulation. As a result, we specify the variable "copies" as private in the superclass "Publication," but we also provide two methods that may access and set its value. When a subclass needs access to "copies," these accessor methods may be utilised since they are either public or protected.

As a result, in the superclass publication, we would have:

```
private int copies;
public int getCopies ()

{
 return copies;
}


public void setCopies(int pCopies)
{
```

Superclasses may regulate access to private instance variables using these techniques. They don't really impose any constraints as they are now designed, but let's say, for example, we wanted to make sure that the value of "copies" wasn't set to zero. The validation (i.e., an if statement) may be

placed inside the setCopies method here, and if 'copies' is private, we can be certain that the rule cannot be compromised. b) If the protected portion of "copies" is partly exposed, we would need to check each instance where a subclass method altered the instance variable and do the validation at each unique location.

To ensure that only subclasses can use these methods and that other classes cannot affect the value of "copies," we may even think about making them protected rather than public.

Using these techniques in our sublease books and magazine:

```
// in Book
public void orderCopies(int pCopies)
{
 setCopies(getCopies() + pCopies);
}

// and in Magazine
public void recvNewIssue(String pNewIssue)
{
      setCopies(orderQty);
```

These statements are equivalent to mCopies = mCopies + pCopies and

mCopies = mOrderQty

Abstract Classes: The concept of a publication that is neither a book nor a magazine, as well as the concept of a person who is neither a male person nor a female person, are useless. As a result, even if we are glad to generate Book or Magazine objects, we may desire to stop creating objects of type Publication. When dealing with a new sort of publication that is really neither a book nor a magazine, such as a calendar, it makes sense that it would be considered a new subclass of publication. The sole reason the class exists is to compile all of the generic characteristics of its subclasses into one location for them to inherit, since It will never be instantiated, i.e., we will never construct objects of this type. Declaring Publication to be "abstract" will require us to enforce the fact that it cannot be instantiated:-

```
abstract class Publication
{
// etc.
```

Method Overrides

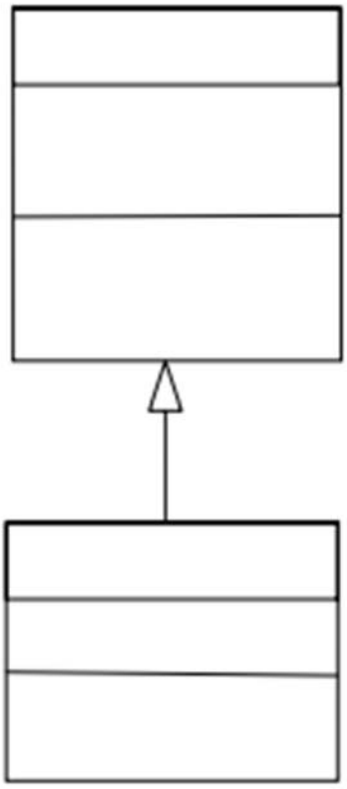
A subclass must always offer at least that number of methods, and often more, since it inherits those methods from its superclass. A method's implementation, however, may be altered in a subclass. This replaces the technique. To do this, a new version in the subclass is created and it replaces the inherited version.

The replacement method should fundamentally perform the same functions as the one it is

replacing, but by altering the functionality, we may enhance the method and tailor its use to a particular subclass.

Consider a particular kind of magazine—call let's it a DiscMag—that comes with a disc attached to each copy. To handle DiscMags, we would construct a subclass of the Magazine class. In addition to updating the existing stock when a new issue of a DiscMag is released, we also want to make sure that the discs are securely fastened. In order to remind us to do this, we need some more functionality in the recvNewIssue() method. Redefining recvNewIssue() in the DiscMag subclass allows us to do this.

Note: Since Magazine issues don't come with a disc, we want to use the original recNewIssue() method specified in the Magazine class whenever a new issue of a Magazine becomes available.



Java automatically chooses the new overriding version when the recvNewIssue() method on a DiscMag object is called; the caller need not express this or even be aware that it is an overriden function. The function in the superclass is called when the recvNewIssue() method on a Magazine is called.

Installing DiscMag

We must use extends to construct a subclass of Magazine in order to implement DiscMag. While they may be added if necessary, no more instance variables or methods are needed. DiscMag's function Object() { } simply provides ALL of its input arguments to the superclass, and a newIssue() function is written in discMag to replace the one inherited from Magazine (see code below).

```
public     DiscMag extends
  {     //
       the Magazineconstructor

     {

           super(pTitle, pPrice, pOrderQty,
           pCurrIssue, pCopies);

     }

     // the overridden method

     public void recvNewIssue(String pNewIssue)
}
```

Like with constructors, the super keyword is used to invoke a method from the superclass, reusing the previous functionality as part of the replacement. The needed message is subsequently shown in addition for the user.

Operations: RecvNewIssue() is formally a kind of operation. One method in Magazine and the overriding one in DiscMag are used to accomplish this single action.

The "Object" Class: All objects in Java are (direct or indirect) subclasses of the 'Object' class. In Java, the inheritance hierarchy has an object as its "root." Thus, this class is present in every Java application ever written. A class inherently extends Object if it does not explicitly declare to do so. Object specifies a number of methods but no instance variables. In order to make these methods usable, new classes often override them. The function function toString() { }() is one example. So, our own classes are direct subclasses of Object by default when we create them.

The top superclass in the hierarchy, if our classes are organised in a hierarchy, is a direct subclass of object, while all subsequent superclasses are indirect subclasses. As a result, all Java classes, whether directly or indirectly, inherit function toString() { } ().

Overriding the 'Object' function function toString() { }()

One of numerous practical methods provided by the Object class is the function toString() { }() function. The signature for function toString() { }() is public String function toString() { } (). Its goal is to provide a string value that symbolises the currently selected item. The function toString() { }() function provided by Object returns results." This is both the object's "hash code" and class name. We must, however, modify this to provide a more relevant string if we want to be generally helpful.

In Publication

```
public String toString()
{
 return mTitle;
}
```

In Book

```
public String toString()
{
 return super.toString() + " by " + mAuthor;
}
```

In Magazine

```
public String toString()
{
 return super.toString() + " (" + mCurrIssue + ")";
}
```

To String(), which was initially defined in Object, has been totally modified in the code above so that Publication.toString() just returns the title. The function toString() { }() function has been overridden once again in Book, resulting in the return of the title and author through the function toString() { }() method of the superclass. The version specified in Publication is used in this overridden version, for example. Book.toString() would thus automatically produce the title, ISBN number, and author if Publication.toString() were modified to return the title and ISBN number. Title and issue are returned by Magazine.toString() through the superclass function toString() { }() function. Because the version of the technique that DiscMag inherits from Magazine is OK, we won't further override it. According to the design assessment, these will be the most universally usable printed representations of these classes' objects. In this situation, the title and author of a book or the title and current issue of a magazine work effectively to distinguish one publication from another.

Subclasses are used to simulate certain behaviours and properties. All subclasses inherit the code from a superclass. All subclasses are affected when we change or enhance the code for a superclass. As a result, we need to write less code for our applications. Subclass constructors are subject to certain guidelines. It is possible to declare a superclass abstract to prohibit its instantiation (i.e. objects created). Inherited methods may be "overridden" such that a subclass implements a function differently from its superclass. All classes in Java are descended from the Object class. Several universal operations defined by "Object" may be effectively overridden in our own classes.

CHAPTER 5
OBJECT ROLES AND THE IMPORTANCE OF POLYMORPHISM

Kamalraj R, Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- r.kamalraj@jainuniversity.ac.in

Pinky or any other pig might be given to someone if they requested you to do so. You could offer Pinky, any other pig, any other mammal (such any lion, any mouse, or any human being!) if someone requested you to give them a mammal. You could offer Pinky, any other pig, any other mammal, or any other animal to someone if they requested you to give them an animal (bird, fish, insect etc).

Each item in a classification hierarchy is seen to have a "is a" connection with each class from which it is derived, and each classification is thought to represent a particular kind of animal.

Animal class types within a hierarchical structure

1. Pig Pinky is (species sus scrofa)
2. A mammal is (also, more generally) Pinky.
3. Animals are (also, more broadly) what Pinky is.
4. At several degrees of specificity, we may describe what kind of object an organism is:
5. less specific at increasing levels less generalised = more precise

The same holds true for object-oriented software. Every time we declare a class, a new "type" is created. The compatibility of variables, parameters, etc. is determined by types. A subtype may be used in place of a supertype wherever one is anticipated since a subclass type is a subtype of the superclass type. After that, if objects of a supertype are needed, we may replace objects of a subtype (as in the example above). The class diagram below displays a hierarchy of object kinds, or classes (Figure 5.1).

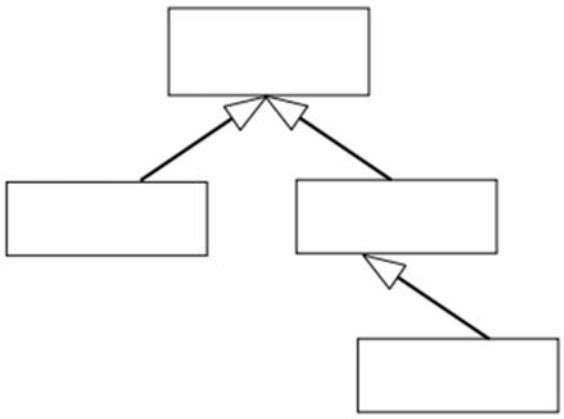


Figure 5.1: Polymorphism

In other words we can ‘substitute’ an object of any subclass where an object of a superclass is required.This is NOT true in reverse!

Substitutability

The type mechanism enables us to insert a subclass object where a superclass is designated when creating class/type hierarchies. This, however, has ramifications for the design of subclasses; we must ensure that they are indeed interchangeable with the superclass. As all of the methods described in the superclass are inherited by the subclass, it is obvious that if an object is substitutable, it must implement every single method of the superclass. As a subclass must at least have all of the methods described in the superclass, it should be interchangeable even though it may have more methods.

We must make sure that methods we override may still be used in lieu of the original method. Because of this, methods should not be overridden with code that performs an intrinsically distinct action, even if it is completely permissible to customise the method to the requirements of the subclass.

For instance, recvNewIssue() in DiscMag overrides recvNewIssue() in Magazine, but both versions do the same essential task (i.e., "fulfils the contract") when it comes to changing the quantity of copies and the current issue. But, by presenting a reminder to check the cover discs, it expands that capability in a manner that is particularly pertinent to DiscMags.

"Publication": It is an object that at least supports the following operations: sellCopy() double get void SetCopies() void Price() int getCopies() (int pCopies) Text to String () Objects of any Publications subclass are guaranteed to supply at least these through inheritance. The promise cannot be violated by a subclass removing an operation that was inherited from its superclass(es). Subclass functionality may be taken for granted since superclasses' capabilities are expanded by subclasses.

The function toString() { }() function (originally implemented inside 'Object') is very likely to be overridden within Publication and again within Magazine so that the String produced offers a richer description of Publications and Magazines. To return the price, we should use the function toString() { }() method, but we shouldn't override it since it would change the method's purpose and cause it to accomplish something else entirely. This would violate the notion of substitutability.

Polymorphism

We may treat subclass objects in the same way as superclass objects since an instance of a subclass is also an instance of its superclass. Additionally, we may call those actions without worrying about which subclass the actual object is an instance of since a superclass ensures certain operations in its subclasses. The word "polymorphism," which originally meant "having numerous forms," is used to describe this trait.

Hence, a publication might take on several forms.

the item can be a book, magazine, or disc magazine. Regardless of their unique specifics, we may call the sellCopy() function on any of these Publications. A frequent concept goes by the fancy term of polymorphism. Since most automobiles have a same set of essential features, such as a steering wheel, gear shift, clutch, brake, and accelerator pedals, which the driver is familiar with using, they can all be gotten into and driven by someone who understands how to drive. Any two automobiles will have many differences, but you can conceive of them as subclasses of a superclass that specifies these essential common "operations."

If "p" "is a publication," it might be a book, magazine, or disc magazine. It has a sellCopy() function, either. Hence, we don't need to worry about what precisely 'p' is when we use p.sellCopy().

When handling items inside an inheritance hierarchy, this may greatly simplify life. Without having to write any code inside the new class, we may construct new kinds of Publication, such as a Newspaper, and use p,sellCopy() on it. The necessary functionality is already provided in Publication.

Polymorphism makes it extremely simple to increase the functionality of our applications.

Extensibility: Every year, enormous amounts of money are spent on developing new computer systems, but over time, far more has been spent on modifying and adapting existing programmes to fit the changing demands of an organisation. So, it is our responsibility as qualified software engineers to make these systems simpler to maintain and modify.

The use of good programming practises, such as commenting, layout, and so on, is obviously important in this situation, but polymorphism may also be useful since it enables the creation of readily extensible systems.

CashTill category

Consider creating a class called CashTill that manages a series of sold products. Without polymorphism, each kind of item would need a different procedure. sellBook (Book pBook) (Book pBook) sellMagazine (Magazine pMagazine) (Magazine pMagazine) sellDiscMag (DiscMag pDiscMag) (DiscMag pDiscMag). We just require sellItem (Publication pPub) with polymorphism since each subclass is "type-compatible" with its superclass. As a result, any object of a subclass may be supplied as a Publication argument. Also, this has significant effects on how extensible systems may be. Even though these subtypes weren't known when the CashTill was created, we may add more Publication subclasses in the future and the sellItem() function of a CashTill object will accept them.

Publications are self-selling

Without polymorphism, we would have to make sure that we were calling the appropriate method for each item "p" in order to sell a copy of that subtype: if "p" is a book, call the sellCopy() method for a book; otherwise, call the sellCopy() method for a magazine; if "p" is a disc mag, call the sellCopy() method for a disc mag. Instead, we put our confidence in the Java virtual machine, which will examine object 'p' at runtime, ascertain its 'type,' and use that information to decide how to sell itself. As a result, we may call p.sellCopy () and it will call the sellCopy() function for a Book if the object is one. Again, Java will detect this at runtime and call the sellCopy() function for a Magazine if 'p' is a magazine. Because of polymorphism, we may often avoid using conditional "if" statements since the "choice" is usually determined implicitly based on the type of subclass object that is really there.

Using CashTill

The code below demonstrates how Polymorphism may be used in CashTill.

```
class CashTill
{
      private double runningTotal;
CashTill ()
      {
            runningTotal = 0;
      }
      public void sellItem (Publication pPub)
      {
      runningTotal = runningTotal + pPub.getPrice();
            pPub.sellCopy();
            System.out.println("Sold " + pPub + " @ " +
                              pPub.getPrice() + "\nSubtotal = " +
                              runningTotal);
      }
      public void showTotal()
      {
      System.out.println ("GRAND TOTAL: " + runningTotal);
      }
}
```

The ongoing transaction total is stored in a double instance variable for the CashTill. This is initialised to 0 by the function Object() { }. The centrepiece of CashTill is the sellItem() function. It requires a parameter called Publication, which may be either a Book, Magazine, or DiscMag. With the getPrice() accessor, the publication's cost is first added to the ongoing total. The publication is then given the sellCopy() command. Eventually, a message is created and shown to the user, for example.

Window-cleaning Weekly Sold at 2.75 (Sept 2005) the total is 2.75. Keep in mind when the string concatenation operator "+" is used with pPub. Keeping in mind that function toString() { }() differs for books and magazines, this indirectly runs the function toString() { }() method for the subclass of this object. Here is an example of polymorphism in action, since the relevant function toString() { }() method for the relevant class is called using the function toString() { }() operation!. In a class diagram, we may display CashTill as seen below (Figure 5.2):-

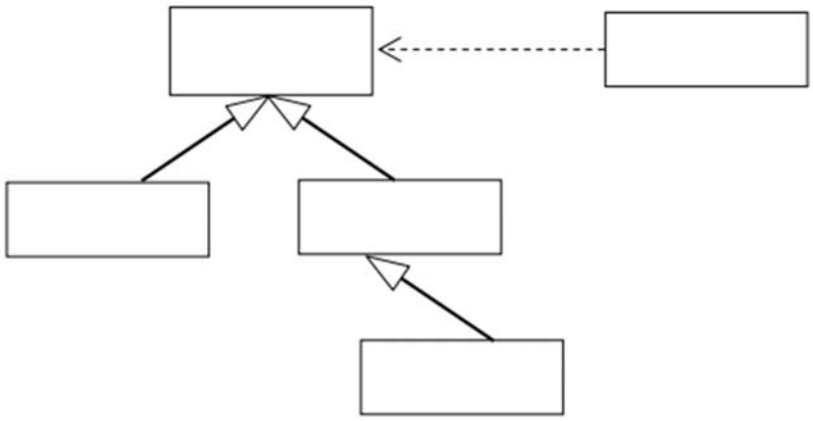


Figure 5.2: represent the CashTill

When a parameter of type Publication is supplied to the sell() function, it should be noted that CashTill is dependent on Publication. Of course, what really transpires will be a physical item descended from Publication? Interfaces: In the Java programming language, an interface is referred to as an abstract type that specifies a class's behaviour. In Java, an interface is a behavior's blueprint. Static constants and abstract methods are components of a Java interface. Java uses the interface as a tool to accomplish abstraction. The Java interface can only include abstract methods; method bodies are not allowed. In Java, it is used to accomplish multiple inheritance and abstraction. In other words, interfaces are capable of including abstract methods and variables. There can be no method body. The IS-A connection is also represented by the Java interface. When an entity's type is determined by its behaviour rather than an attribute, the entity should be defined as an interface. An interface, like a class, may include variables and methods, but by default, the methods stated in an interface are abstract (only method signature, no body).

Inheritance has two components:

1. Polymorphism is feasible because the subclass inherits the interface (access to public members) of its superclass.
2. Instead of duplicating the contents of the superclass declaration into the subclass definition, the subclass inherits the implementation of its superclass (i.e., instance variables and method implementations).

The extends keyword in Java automatically applies both of these features.

A subtype is a subclass. Its interface must include all of the interface of its superclass, albeit the implementation may change (through overrides), and the subclass interface may be longer and have more features. Yet sometimes we would prefer it if two classes could share an interface without being in an inheritance tree. This might be due to:

We want a class to share interfaces with more than one would-be superclass, but Java does not support such "many inheritances," and we want to establish a "plug and socket" connection between software components, some of which may not even be developed at the moment. This is analogous to ensuring that the controls on two vehicles operate precisely the same but allowing separate engineers to create engines that "implement" the operation of the car, maybe in very different ways. Keep in mind that the word "interface" has at least three different meanings in Java programming. A class's public members - as in the definition above. A program's "user interface," sometimes a "Graphical User Interface" - a somewhat unrelated meaning

A particular Java concept we're soon to encounter

Because of this, the CashTill class may handle a "Publication" without concern for the precise subclass of which it is an instance. Consider the potential that we could wish to offer tickets in addition to books and periodicals, such as for entertainment events, public transportation, etc. They are different from publications in that, Tickets consist just of a description, price, and the customer (to whom they are being sold); we don't have a limited "stock"; they are printed on demand at the register; these transactions are really services rather than products. Tickets seem to have nothing in common with publications in terms of their selling interface, but even in this case, they will vary in terms of the implementation since we won't be subtracting them from a stock that is already in existence.

Because to these factors, we do not wish to include Ticket and Publication in an inheritance hierarchy since they do not seem to be tightly connected. Nevertheless, we do want to make both of them sellable to CashTill, thus we need a method for doing so. What we want is a more generic method of expressing "items of this class may be sold" that can be applied to any (current and future) classes we desire, making the system easily expandable to Tickets and anything else. This is done without placing them in an inheritance structure. While we don't want to include the Ticket class in an inheritance structure since it is sufficiently distinct from a Publication, it does have certain characteristics, such as having the getPrice() and sellCopy() methods that CashTill requires. The sellCopy() function, however, differs significantly from the sellCopy() technique described in Publishing. With a ticket, we just need to print it once instead of having to lower the stock of a magazine by one to sell it.

```
public void sellCopy()
 {
     System.out.println("************************************");
     System.out.println(" TICKET VOUCHER ");
     System.out.println(toString());
     System.out.println("************************************");
     System.out.println();

 }
```

We do not believe that Ticket should be in an inheritance tree with Publications because of how different the sellCopy() function is, and we do not want to inherit its implementation details. But as with publications, we do want to be able to verify tickets via the till.

Tickets supply the operations that CashTill requires, much as newspapers do:

1. sellCopy()
2. getPrice()

So that a Ticket may be sold by the CashTill. In actuality, not only publications may be sold via CashTill using these ways. This collection of operations will be defined as a "Interface" named SaleableItem in order to make this possible.

```
interface SaleableItem
{
     public void sellCopy ();
     public double getPrice ();
}
```

Keep in mind that the interface only specifies the operation signatures and not their implementations. Even if it isn't explicitly mentioned, all methods are inherently public, and constructors and instance variables are not allowed. In other words, an interface just specifies the existence of a set of actions without describing how they are implemented. Classes that implement the interface are left to handle it. A contract of sorts would be an interface. "I vow to offer, at least, methods with these signatures," the SaleableItem interface states.

1. public double getPrice ();
2. public void sellCopy ();
3. even if I may also add other stuff.

There is a guaranteed region of commonality when many classes implement an interface, which polymorphism may take advantage. Imagine playing a driving game in an arcade with a vehicle. They are undoubtedly not connected in any way by the "is a" connection since they are completely distinct types of objects—one is a car and the other is an arcade game. Even while the implementations mechanical linkage in the automobile vs visual electronics in the game are extremely different, they both have what we may term a "SteeringWheel interface" that we can use in precisely the same manner. We must now assert that the operations specified by this interface are provided by both Tickets and Publication (and all of its subclasses).:

```
class Publication implements SaleableItem
{
 [...class details...]
}
```

```
class Ticket implements SaleableItem
{
 [...class details...]
}
```

Compare extends to implements

extends results in the inheritance of the superclass's interface and implementation.

Implementing a particular interface ensures that the actions described by it will be available, which is sufficient to permit polymorphic handling of any classes that do so

The Multifaceted CashTill

The CashTill class already uses polymorphism since the selling function receives a Publication-type argument that may be any of its subclasses:

public void sellItem (Publication pPub)

In order to further extend this, we will now accept anything that uses the SaleableItem interface: void public sellItem (SaleableItem pSelb). When a variable or parameter's type is specified as an interface, it behaves exactly like a superclass type. Given that the interface is a type and all classes that implement it are subtypes of that type, any class that implements the interface is suitable for assignment to the variable or parameter. Now, as indicated below Figure 5.3,

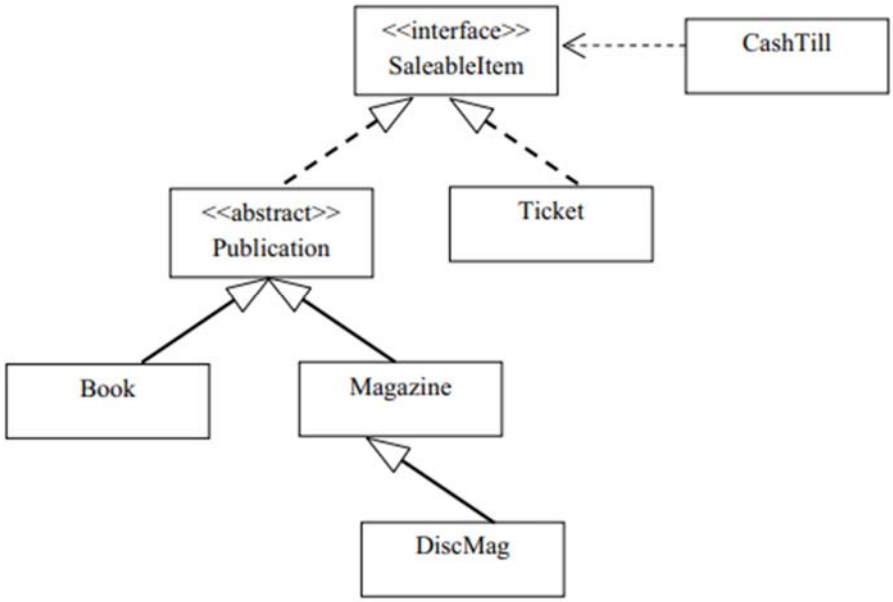


Figure 5.3: Represent the public void sellItem

CashTill now depends on the interface SaleableItem rather than the class Publication, which was formerly its direct dependency. The lines connecting Publication and Ticket to SaleableItem are dotted, rather than being connected by inheritance arrows, to indicate that each class has implemented the interface.

Extensibility Again

Because of polymorphism, objects may be handled regardless of their exact class. This may help systems become expandable while maintaining the present design's encapsulation. For instance, if we make new classes for more goods or services that implement the SaleableItem interface, the CashTill will be able to process them without needing to make any changes to its code. Sweets is a possible illustration. In order to represent sweets in a jar, we may create the class Sweets. In order to sell the sweets, we may determine their price based on their weight and deduct that amount from our total stock. This is not the same as selling publications, where we constantly deduct 1 from the stock, or selling tickets, where we just print them. But, our upgraded polymorphic cash till can sell them since it can sell any SaleableItem if we construct a class called "Sweets" that implements the SaleableItem interface.

In this instance, a new "sell" method would need to be added to CashTill in order to handle Tickets, Sweets, and other new methods for every new kind of goods to be sold if polymorphism didn't exist. With the definition of the SaleableItem interface, more goods may be added without having any impact on CashTill. Our applications may be easily extended thanks to polymorphism, which is crucial. Similar to the several different types of plugs and sockets that enable audio, video, power, and data connections in the real world, interfaces allow software components to connect more freely and extensibly. Consider how many different electrical appliances may fit into a normal power outlet and how unpleasant it would be if you had to hire an electrician to set up each new one you purchased instead.Differentiating the Subclasses. What if we already handle an object polymorphically but still need to determine which subtype it truly belongs to? Here is what the instanceof operator may do:

class object instance

If the object belongs to the provided class (or a subclass), the test is true; otherwise, it is false. Since a DiscMag is a Magazine, it should be noted that (myDiscMag instanceof Magazine) would be TRUE. Moreover, instanceof may be used in conjunction with an interface name on the right to determine if a class implements the interface. By using strictly instanceof, you are determining if the object on the left belongs to the type provided on the right or is a subtype of that type. By doing so, we might modify the CashTill class so that it displays a certain message based on the item sold.

```
public void saleType (SaleableItem pSelb)
{
     if (pSelb instanceof Publication)
     {
         System.out.println("This is a Publication");
     }
     else if (pSelb instanceof Ticket)
     {
         System.out.println("This is a Ticket");
     }
     else
     {
         System.out.println("This is a an unknown sale type");
     }
}
```

If pSelb is any subclass of Publication, then pSelb instanceof Publication will be true (i.e. a Book, Magazine or DiscMag). We could similarly test for a more precise subtype, such as pSelb instanceof Book, if we so desired.

The system's extensibility is also compromised once the polymorphism is compromised by checking for subtypes; new classes (such as Sweets) that implement the SaleableItem interface

may need to add new clauses to this if statement as a result. As a result, maintaining the system becomes more expensive and error-prone. Instead of doing this, we should strive to encapsulate certain behaviours into the subclasses themselves. For instance, we might create a describeSelf() method in the SaleableItem interface; any class that implements the SaleableItem interface would then need to implement this method. Consequently, a message indicating the sort of object being sold would be shown for each subtype. Then, in CashTill, pSelb.describeSelf may be used in lieu of the if line above (). As a result, we wouldn't need to modify the CashTill class when adding new classes to the system.

Because of polymorphism, we may refer to objects by a superclass rather than by their actual class. By introducing new classes without altering existing ones, polymorphism makes it simple to expand our applications. We don't need to worry about which subclass is involved in any given situation since we may modify objects by using operations that are specified for the superclass. Java makes sure that the proper method is called at run-time for the actual class of the object.

There are situations when we wish to use polymorphism without requiring that all of the involved classes be in an inheritance structure. In this case, we may offer common interfaces (i.e., collections of actions) thanks to the "interface" concept. There is no inherited implementation when doing this; instead, each class is required to implement EVERY action specified by the interface.

CHAPTER 6

BASICS OF OVERLOADING

Mr. Soumya A K, Assistant Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- soumya.k@jainuniversity.ac.in

Overloading: In the past, method names in computer programmes had to be distinct. As a result, the compiler could tell which method was being called just by reading its name. Yet, it was sometimes necessary to use many ways to do very similar tasks. For instance, adding two integer values would call for one technique, while adding two floating point numbers might call for a different one. Which of these two methods would you name "add()" if you had to give them both unique names?

Each technique would need a different name, therefore the names would need to be longer and more detailed. Hence, we could give one method the name addInt() and the other addFloat(), but this may result in a profusion of names for methods that are effectively doing the same operation—adding two numbers—under different names. To solve this issue, you are not needed to give each method in Java a distinct name; hence, both of the aforementioned methods might be named add (). The Java Runtime Environment (JRE) must, however, provide another mechanism of determining which method to call at run time if method names are not unique. In other words, the computer must choose between the two techniques when a request to add(number1, number2) is made.

Although sharing the same name, the two methods may still be separated by glancing at the argument list. add(int number1, int number2) (int number1, int number2) add(float number1, float number2) (float number1, float number2) The JRE handles this at run time. i.e., the JRE examines the method call and the actual arguments supplied at runtime. The first function is called if two integers are given. Nevertheless, the second approach is employed if two floating-point values are given. When numerous methods have the same name, this is referred to as overloading. The term "overloaded" refers to a method name that no longer uniquely identifies the method.

Using Overload to Promote Flexibility

It may improve a system's flexibility and resilience to have many methods that effectively execute the same function but accept various parameter lists. Consider a method for managing students at a university. It would likely take a process to enrol or register a new student. A method like that may have the following signature.

enrollStudent(String pName, String pAddress, String pCoursecode) (String pName, String pAddress, String pCoursecode). This might be accomplished by overloading the enrollStudent() function and providing an alternate method in its place. enrollStudent(String pName, String pCoursecode) (String pName, String pCoursecode). According to the specified argument list, the JRE might choose which method to call at runtime. As a result, the JRE would use the first technique in response to a call to enrollStudent("Fred", "123 Abbey Gardens," "G700").

As a result, overloading methods may make the system more resilient to changing needs and future

proof. Overloading methods not only gives users more freedom, but also gives programmers who may need to expand the system greater flexibility. Both conventional procedures and constructors are susceptible to overload.

Overloading constructors and other methods will enable us to increase the adaptability of our programmes. Even if we don't initially utilise all of the various constructors or techniques, by including them we increase the adaptability and flexibility of our programmes to satisfy shifting needs.

The idea that many methods may exist that effectively execute the same action and so have the same name is known as method overloading. The parameter list is used by the JRE to differentiate between them. If more than one method has the same name, then each method's argument list must be unique. The JRE resolves each confusing method call at runtime by examining the arguments given and comparing the data types with the method signatures specified in the class. We provide the programmers who could use our classes more freedom by overloading constructors and common methods. Even if not all of them are utilised right once, having them available may help the application be more adaptable to changing user needs.

CHAPTER 7

OBJECT ORIENTED SOFTWAREANALYSIS AND DESIGN

Dr. Smitha Rajagopal, Assistant Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- smitha.rajagopal@jainuniversity.ac.in

An engineer who specializes in designing bridges may need some software to create three dimensional models of the designs so people can visualise the finished bridge long before it is actually built. A manager may need a piece of software to keep track of personal, what projects they are assigned to, what skills they have and what skills needs to be developed. Requirement analysis, or the effort of determining precisely what is expected of the product, is a specialty activity for certain software developers. This is often accomplished by repeatedly carrying out the following actions:-

1) Speaking with customers and future users of the system to get their opinions on it

2) recording the outcomes of these discussions

2) recording the outcomes of these discussions

4) Creating prototype designs (and possibly prototypes of the system)

5) Assessing these first ideas with the client and possible users

6) continuing in this manner until a final design has been created.

While doing requirements analysis is a specialised skill that is beyond the purview of this article, we will concentrate on steps three and four above, that is, how can we translate a system description into a viable object-oriented design. Even if we may aspire to obtain some basic design abilities, experience is crucial for this endeavour. If we want the software to function properly and be easy to build, we must create simple and beautiful designs. Nevertheless, it is difficult to distinguish between strong designs and weaker designs, and expertise is a crucial aspect. Even though a beginner may be familiar with all the rules, it takes practise to understand how to distinguish between good and poor movements, and practise is crucial to developing into a great player. Similar to doing user requirements research, experience is crucial to developing effective design skills. Here, we'll try to build some fundamental abilities in the hopes that you'll later have the chance to use them and get experience.

Beginning with a problem definition, we will proceed in the order shown below:-

Listing nouns and verbs, identifying items outside the system's purview, locating synonyms, locating potential classes, locating potential attributes, locating potential methods, locating common characteristics, refining our design using CRC cards, and elaborating classes are all examples of things to do. This will allow us to start with a broad description of a problem and produce a workable, and preferably elegant, OO design for a system to address these requirements.

The Issue: The issue that we will try to solve is how to create a simple management system for a running club that is planning a marathon. For the sake of this exercise, we'll assume that preliminary needs analysis has been completed and that the following textual description has been produced after interviewing club management and system users.

List of Nouns and Verbs

Finding the nouns and verbs in the description above is the first stage in analysis:

The nouns denote entities, or objects; some of them will show up as classes and others as characteristics in the final system. The verbs denote the performance of activities; some of these actions will be represented in the final system as methods. When a noun or verb alone is not sufficiently descriptive, such as when the word "print" is not as obvious as "print receipt," noun and verb phrases are used instead. Plural nouns and verbs are stated in their singular forms (e.g., "books" becomes "book").

Recognizing Items

That Are Not within the System's Purview: Finding the components of the issue that are irrelevant or beyond the system's purview is a crucial step in system design. Elements of the description may not directly explain parts of the system we are developing because they are entirely contextual, or for general information reasons. Furthermore, while some of the description may refer to tasks that users of the system perform while using the system, thereby describing functions that need to be implemented within the system, other portions of the description may refer to tasks performed by users while they are not using the system, thereby omitting any mention of system functions. We keep the challenge as straightforward as feasible by pointing out details in the description that are unrelated to the system we are designing.

Recognizing Synonyms: Two words that have the same meaning are said to be synonyms. It is crucial to include them in the system description. Failing to do so will result in the modelling of the same object twice, which will cause confusion and duplication.

Finding Possible Classes: We can now begin to identify prospective classes in the system that will be implemented once we have simplified the issue by identifying components that are beyond the system's purview and by identifying various portions of the description that are really describing the same entities and processes.

Some nouns will denote implemented classes, while others will denote class properties. Classes are complicated conceptual entities for which we can identify related data and activities, in accordance with the good OO design principle that says that data and operations should be packaged together (or methods). Basic entities, like addresses, may be kept as simple attributes inside a linked class since they contain connected data but no activities.

Finding Possible Qualities: After prospective classes have been determined, further nouns may be used to specify characteristics of those classes.

Finding Possible Approaches: We can now utilise the verbs to determine the classes' methods after identifying probable classes.

Recognizing Shared Features: We may begin organising our classes into suitable inheritance hierarchies by finding those classes with similar traits after we have found the candidate classes with connected properties and methods.

Using CRC Cards to Improve Our Design: We might now continue to develop these designs by defining the data types and other minute details, documenting this information on a UML diagram, and programming the system after having determined the primary classes in our system, as well as the characteristics and methods of these classes. The analysis and design refinement would be a lengthier, more involved process that we can realistically mimic in a real-world system since the issue would be greater and less well defined than the one we are working on here. As real-world problems are more complicated than our original concepts, it becomes important to examine our designs and address any possible issues before implementing them into a final system. One way to

verify our ideas is to use CRC cards to record them, then test their functionality by simulating various situations.

The UML standard does not include CRC cards, which are also not the sole method of achieving this. Courses, Responsibilities, and Collaborations, or CRC. The duties for the class are shown on the left and the collaborations are displayed on the right of the three panes that make up a CRC card, which is seen below Figure 7.1.

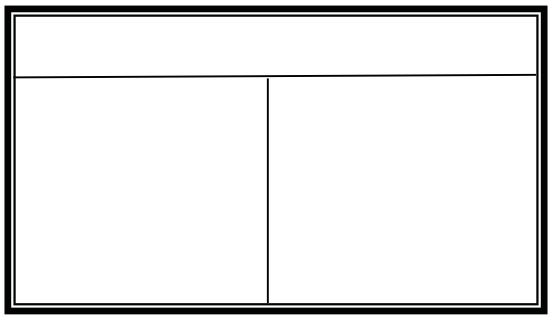


Figure 7.1: CRC card

While they are not as clearly stated on a CRC card as they are on a UML diagram, responsibilities are the things the class needs to know about (i.e., the attributes) and the things it should do (i.e., the methods). The partnerships are other courses that this class must cooperate with in order to carry out its duties. The Figure 7.2 below displays CRC cards created for two system classes.

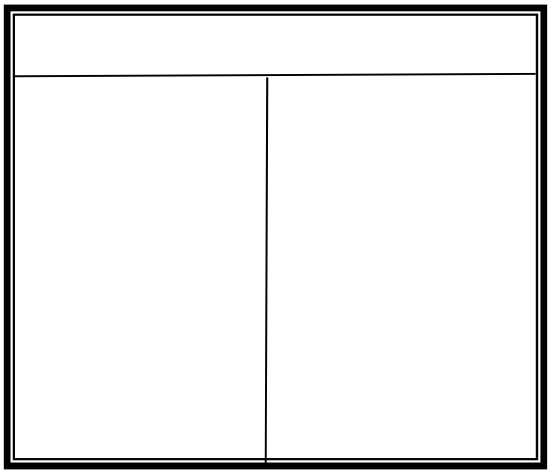


Figure 7.2: Displays CRC cards created for two system classes.

We may simulate a variety of situations to make that the system works "on paper" after developing CRC cards. If not, we may make changes instead of spending time developing a poor strategy. One example would be if a runner received a second sponsor. In this instance, a Runner working with the SponsorshipForm class is able to record sponsorship information by glancing at the CRC cards above. A list of sponsors is kept on file by the SponsorshipForm class, and more sponsors may be added. A variety of situations may be tested to identify design problems in our systems, which can then be fixed before any time is lost in creating shoddy designs.

classes that elaborate: We can now extend on our CRC cards and use a UML class diagram to describe our classes after identifying the classes in our system design, documenting and testing them using CRC cards. To do this, we must first clarify any ambiguities, such as the precise data types, in our general specification, which is written using CRC cards. We can now create a class diagram for the suggested design after further developing our CRC cards (see below Figure 7.3):-

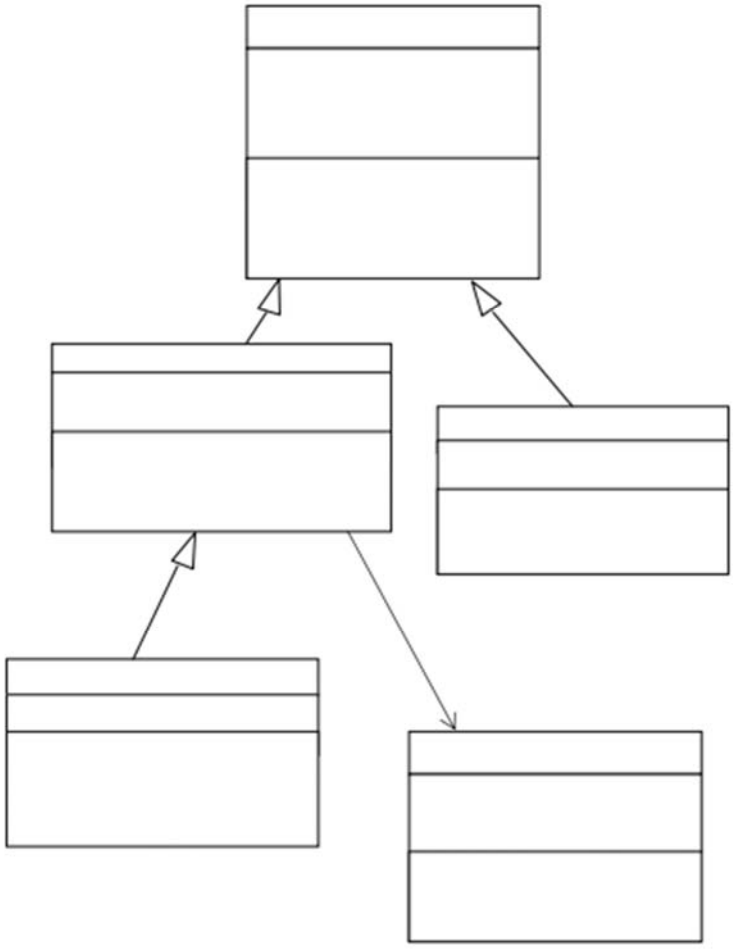


Figure 7.3: Design after further developing our CRC cards

A key step in the software engineering process is gathering user requirements (and outside the scope of this text). It takes talent and experience to convert a complicated needs specification into a beautiful, straightforward object-oriented architectural design. Nonetheless, following the straightforward procedure. We learned how to analyse a written description of an issue via a series of exercises. We possess:

1. Determined where the description refers to identical objects using different terms; • Sought out portions of the description that are beyond the purview of the system.
2. Recognized probable classes, properties, and methods using the nouns and verbs
3. Examined the classes to find any possible inheritance hierarchies and other connections between the classes (e.g. associations).
4. Record the resultant classes using CRC cards, evaluate the validity of our design through role-playing, and adjust our designs as necessary.
5. Lastly, a class diagram may be used to illustrate these specifics and record the outcomes.

Software Engineering Principles

The following are the guidelines and strategies need to follow in order to remain rational and make technical decisions that are appropriate given the needs, budget, timeframe, and expectations. Following these guidelines will aid in the smooth progression of your project .

KISS (Keep It Simple, Stupid)

According to the idea of simplicity, complex structures should be avoided in programs in order to make debugging and maintenance easier. In addition, it will take more effort and time for another programmer to comprehend the logic of the code. Make careful to produce clear, understandable code while developing your next major project.

1. It's great if your approaches are brief—no more than 40–50 lines—in length.
2. For easier comprehension by other developers, all crucial/critical methods should have commented documentation.
3. Only one issue at a time should be addressed by techniques.
4. There are many requirements for your project, right. As you continue, separate the codes into smaller sections.
5. Use straightforward solutions that address the issue without a lot of branching, deep nesting, or intricate class structures, if at all feasible.
6. Always Keep It Simple, Stupid helps you and your coworkers find bugs more quickly (KISS). The idea also makes it simpler to change and enhance code. Recall what Edsger W. Dijkstra said: "Reliability requires simplicity ."

DRY (Don’t Repeat Yourself)

The DRY principle, in a nutshell, says that we shouldn't repeat the same thing too frequently or in too many places. It seeks to lessen repetitious code and work in software systems. Unknowingly, developers frequently rewrite code. Avoid continually copying and pasting the same code when creating your program. If you don't, then you'll need to maintain them in sync since any changes you make to the code anywhere else will also need to be made there. It will require additional time, energy, and attention (which isn't always simple).

Make sure code is clear of duplicate lines in addition to being error-free. A piece of code should be relocated to a different function if it appears more than twice in the codebase. Even if discover that it is repeated a second time, should still develop a different way. Automate any manual procedures you can to keep the code lean as a bonus. These actions will make it easier for software code to be reused, reducing the need for repetition. The code becomes less buggy, more extendable, and reusable as a consequence .

YAGNI (You Aren't Gonna Need It)

Programmers should follow this rule and avoid adding functionality unless it is absolutely required. It states that you shouldn't introduce stuff to address hypothetical, future issues. Most of the time, programmers attempt to include all the functionality at once, from the very beginning. Over time, the majority of these features become worthless. Additionally, the absence of YAGNI could result in a lot of unorganized code and refactoring. A class should always start out with a small number of methods. Dead code shouldn't be added to the project. May include more functionality as project takes shape and as new demands emerge. As a result, lean software development will be produced. As a result, trying to comprehend or debug the code doesn't need as much time, effort, or money. It is advised to implement only the most fundamental functionality at initially, and then extend them as needed. YAGNI also stays away from complexity, especially that which results from incorporating features that could be required in the future.

BDUF (Big Design Upfront)

This rule states that a developer should first design the project, then draw the diagram's flow, and last implement it. Prior to creating functionality, we should carefully consider the architecture and design of the entire system. After that, we should carry out the procedures indicated in our implementation plan. This makes it easier to find and swiftly fix problems at the requirements stage. However, it is feasible for software needs to change during the course of a project, and these changes to software requirements may cause issues or even make the design outdated. The optimal approach to this would be to first design the architecture, then segment the needs into phases based on their importance. During the development process, move from the stage with the highest importance to the one with the lowest. Apply the BDUF concept at each step before coding.

SOLID: An abbreviation representing a group of object-oriented design concepts is SOLID. The following principles are represented by one of the letters in the word "SOLID": A class, function, module, or service must only have one cause to change, or only one duty, in order to comply with the Single Responsibility Principle (SRP). It is simpler to comprehend, maintain, and adjust your code when you create classes or functions that are focused on a specific functionality. Be aware of the precise DRY area where the code has to be modified if you wanted to change the system's functionality. It improves the organization and readability of the code. It also makes it simpler to reuse the code.

O - OCP (Open Closed Principle): When developing software, we go through stages. We build several capabilities as a team, test them, and finally provide them to the users. The following set of capabilities are then implemented. The last thing we want to do when creating new functionality is to alter already-existing functionality that has been tried and tested and is functioning as intended. So we attempt to layer additional functionality on top of already existing features. The Open-Closed principle aids in the realization of this concept. It states that our modules, classes, and functions should be created so that they are open to expansion but closed to alteration.

Open to Extension: Classes and modules can have new features added to them without affecting the existing code. To do this, composition and inheritance can be employed.

Declared Final for Modification: It's best to avoid making changes that disrupt existing functionality because doing so would need reworking a significant amount of existing code and the creation of numerous tests to verify the changes are effective.

Liskov Substitution Principle (L-LSP): The Liskov Substitution Principle states that all child/derived classes should be interchangeable with their parent/base classes without compromising the correctness of the program. Consequently, objects in your subclass (derived/child class) need to exhibit behavior that is comparable to that of objects in your superclass (parent/base class). As a result, you should be cautious while using inheritance in your applications. Although inheritance might be advantageous, it is best to use it sparingly and in the right circumstances. Must take into account the class's post conditions and preconditions before you can conduct inheritance.

I - ISP: Clients shouldn't be made to rely on or utilize methods that they don't employ, in accordance with the Interface Segregation Principle. How is this accomplished? Make your interfaces brief, compact, and focused. A complex interface is challenging to maintain and develop. Therefore, divide huge interfaces into smaller ones that are each focused on a single set of tasks, allowing users to just rely on the functionality they actually need.

Dependency Inversion (DIP): The Dependency Inversion Principle (DIP), sometimes known as the DIP, aims to reduce tight dependency between software units. This rule states that high-

level modules should be independent of lower-level modules and instead rely on their abstractions. It may be divided into two sections:

The independence of a high-level module from a low-level module is required. Abstractions should be used by both. The specifics should depend on the abstractions, while the abstractions should be independent of the details.

CHAPTER 8

FRAMEWORK FOR COLLECTIONS

Dr. Solomon Jebaraj, Assistant Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- solomon.j@jainuniversity.ac.in

Different groups of programmers need to agree on a "contract" that details how their software interacts in a number of circumstances in software engineering. Without any previous knowledge about how the other group's code is created, each company should be able to write their own code. Interfaces are, in general, such contracts. Consider a civilization of the future where autonomous robotic automobiles are used to carry people across urban streets without the need for a human driver. Automakers provide software (of course in Java) that controls the vehicle's stop, start, acceleration, left turn, and other functions. Manufacturers of electronic guidance instruments, another industry sector, create computerized vehicles that employ wireless traffic information and GPS (Global Positioning System) location data to navigate. The automakers must release a manufacturing interface that describes in full the techniques that can be used to move the vehicle (any car, from any manufacturer). The guidance makers may then create software that instructs the interface's ways for controlling the vehicle. Neither industrial group has to be aware of the software implementation methods used by the other group. Each organization actually views its software as highly proprietary and reserves the right to make changes to it at any moment as long as they maintain the disclosed interface.

Default method:

The primary purpose of the default methods' introduction was to fix Java's compatibility problems. Interfaces could only have abstract methods prior to Java 8's default methods. The implementations of these functions have to be provided by developers in a different class. Therefore, the implementation code must be included in the class that is implementing the interface if a developer wished to add a new function to an interface. This problem is addressed with default methods, which let interfaces contain implementation-ready methods without impacting the classes that implement the interface. The default methods may be distinguished from conventional methods by their body and default modifier. If you wish to include a default method in an interface, the default modifier is required.

Although default methods in Java 8 provide interfaces the ability to have concrete methods, this is not an upgrade and may perhaps be a long overdue Java corrective. The creators of Java's core APIs utilized interfaces to specify the API interfaces for Java Collections API, for example. As a result, Java Interfaces make up the majority of the interfaces in the collections API. Later, when Java 8 Streams were released, the collections API needed to be changed in order to handle streams. It must be implemented, though, if you wish to add new methods to a Java interface. Additionally, they attempted to maintain backward compatibility by supporting older versions of List or Map while updating them. Java developers introduced the idea of default methods in Interfaces to fulfil both of these goals. Now, the implementations have the option of using the default implementation or implementing a method.

Static Method:

The methods specified in an interface with the keyword static are known as static methods. These static methods unlike other methods in Interface contain the whole definition of the function, and because they are static and the specification is full, they cannot be modified or altered in the implementation class. The static method in an interface can be declared in the interface but cannot be overridden in Implementation Classes, much like the Default Method in an Interface. Because a static method is exclusive to an Interface, it must be instantiated with it in order to be used.

Static method conventions of the Java interface:

1. The static modifier is a requirement for all static methods in their definition.
2. They must be physical beings.
3. The interface name is the only thing we need to use to call the static methods.
4. Static methods cannot be overridden or inherited by any subclass or sub interface.
5. They are not allowed to use any other predefined or abstract methods.Make them private even if they are by default public.

Private interface methods: A Java interface method is by default public. Any class or interface extending this interface is now able to invoke this function. The following may be utilized in interfaces when using the Java programming language:

1. Constant variables
2. Abstract methods
3. Default methods
4. Static methods
5. Private methods
6. Private Static methods

Advantages of Private Interface Techniques:

Some advantages of employing private interface techniques are listed below:

Code reuse: Developers can utilize private interface methods to reuse code inside the declaring interface, but you should conceal across interface implementations.

Encapsulation: Programmers can use private interface methods to encapsulate functionality that shouldn't be shared among interface implementations.

Requirements for Java's Private Methods in Interfaces

Developers should adhere to the guidelines and best practices listed below when utilizing private methods in Java programs.

1. Private interfaces are not permitted to use abstract methods.
2. The use of private interface methods is restricted to interfaces.
3. The use of both private and abstract modifiers at once is not permitted.
4. It is possible to employ a static method inside of a non-static or static method.
5. Using a private non-static method inside of a private static method is not feasible.

An Overview of Collections: Instead, than only storing individual things, most software systems also need to store multiple groupings of entities. The Java Collections enable significantly more diverse and flexible ways of grouping than the Arrays provide. Collections (sometimes known as "containers") are classes used in Java to organize groupings of other objects. A standardized collection of interfaces and implementations known as the "Collections Framework" is made available by the Java platform. Interfaces provide the functionality that is

accessible, and implementations have an impact on other factors, such as performance. Interfaces for Collections: An interface with the name "Collection" serves as the foundation of the collection's framework. This outlines the framework's fundamental techniques. The interfaces List and Set are variations on Collection; they take on all of its activities and add some more.

The fact that they are interfaces—not classes—means that they only describe operation signatures, not any of the methods or other elements of their implementation. 'Map' is a significant extra interface. Maps do not quite belong in the Collection hierarchy for reasons we shall discuss later, despite the fact that they are still a component of the "Collections Framework," therefore this is not an extension of Collection. Thus, we have the following interfaces.

Collection: the broadest category

A list is a collection of items stored in a certain order, some of which may be duplicates.

SortedSet: a set with items ordered in ascending order. Set: a collection of distinct objects in no particular order.

SortedMap: a map with items ordered in ascending order of keys; Map: a collection of distinctive "keys" and related things SortedSet and SortedMap are extensions of Set and Map, respectively, much as List and Set are extensions of the Collection interface.

The New and Old Collections: As all objects are subclasses of Object, collections stored items of type Object up until Java SDK 1.4. This enabled a collection to include a variety of items. Unfortunately, when items are removed from a collection, the compiler is not aware of what the item is, which might lead to some extra issues. In reality, we practically never want anything other than items of a certain type in a collection. As a result, "generics" were introduced in Java JDK 5.0. A class may be created without identifying the kinds of data objects it handles thanks to "generics." Instead, the types are fixed for certain objects since they are given when the class is created.

Lists: The most popular collection type is a list. A list will often provide a more practical way of processing the data than an array would have. Lists are very versatile data structures where every item has a specified place (Figure 8.1). They are similar to arrays in many respects, but since they are automatically resized when new data is added, they are more versatile. They are also considerably simpler to deal with than arrays since there are so many helpful methods available that take care of the majority of the work for you. Duplicate items are allowed in lists, which store things in a certain order (but not necessary in any meaningful order).

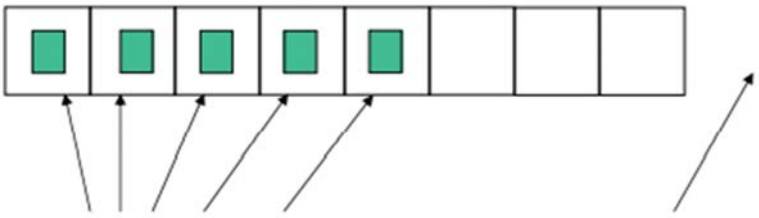


Figure 8.1: Lists.

Sets: Instead, than being a list, a set is more like a "bag" of items. They are based on the mathematical concept of a "set," which is a collection of "members" that may include 0–1 or n–1 different object. In contrast to a list, duplicate items are not allowed, and a set does not

place things in chronological order (but a SortedSet does). Like lists, sets automatically adjust their size as new items are added. A Set may do many of the operations that are accessible for Lists, however As the elements are not in any particular order, we CANNOT add an object at a specified point. We CANNOT "replace" an item for the same reason (though we can add one and remove another). We can get all the elements, but there is no way to tell what order they are in.

Map: Since they hold pairs of items instead of single ones, Maps vary from Lists and Sets in a significant way. Each pair consists of a "key" and a "value." Anything that identifies the duo is the key. An item or piece of data connected to the key is the value. An address book, for instance, may include names of individuals as keys and addresses, phone numbers, and emails as values. Each key has a single value, but since values are objects, they may hold several bits of data.

Duplicate values are allowed, but duplicate keys are not. In the preceding example, if you searched up two persons at the same address in the address book, you may discover that they both live there, but only one person would have two residences. A SortedMap does so, in the order of keys, while a Map does not. Like lists and sets, maps are automatically resized when things are added or removed (Figure 8.2).

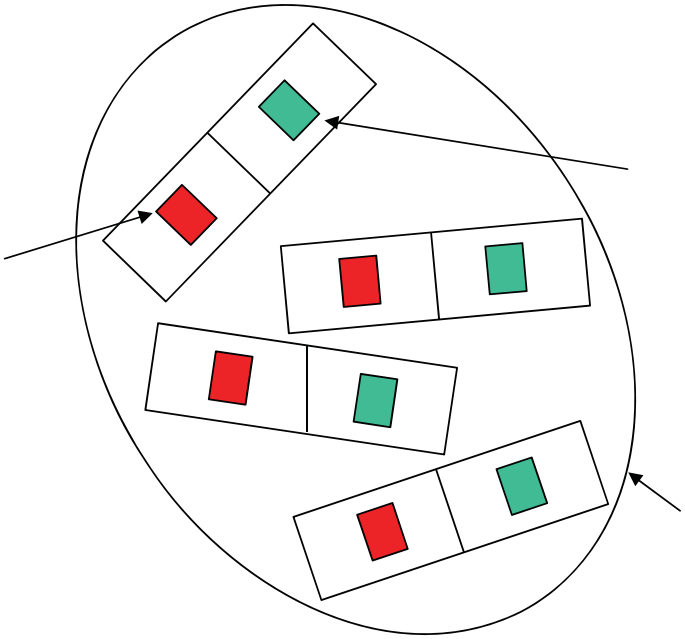


Figure 8.2: Represent the SortedMap

Implementations of Collection: We have been looking at the collections interfaces up to this point, which are functional specifications from which we can choose the features we wish to employ. We need classes that implement these interfaces in order to utilise them effectively. While the implementations shown here are all included in the Java platform library packages, programmers may also create their own implementations to suit particular needs. LinkedList, which also implements the List interface, may be quicker than ArrayList in certain less common situations. ArrayList implements the List interface and often provides rapid access for an ordered list. TreeSet also implements the Set interface; it is slower than HashSet but maintains elements in order, giving us a SortedSet. HashSet implements the Set interface; it allows rapid access but the items are unordered. TreeMap, which also implements the Map interface, is slower than a HashMap but maintains elements in order of the Key, giving us a SortedMap. HashMap implements the Map interface and allows quick access, but the items are

unordered. As List is an interface, you cannot construct a "List object," but you may create either an ArrayList or a LinkedList object, and either of these will provide the functionality specified by the List interface. Collections Framework Overview: We can see a little condensed overview of the collections structure in the figure below. It displays the interfaces as well as the classes that use them. Recall that interfaces just include public method signatures that any class that implements them must satisfy. They do not really contain any implementation.

Two different kinds of arrows are shown. Inheritance is shown by arrows with solid lines; for instance, the Set interface extends the Collection interface. Implementation is indicated by arrows with dotted lines; for instance, the HashSet class implements the Set interface. Keep in mind that Map is not a Collection subinterface. This is due to the fact that Maps hold pairs of key objects and value objects rather than individual items as Lists and Sets do. We continue to see Map as a component of the collections framework (Figure 8.3).

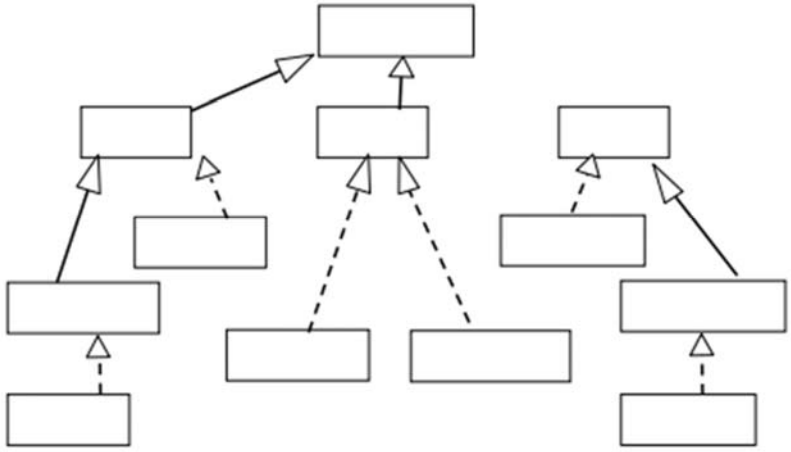


Figure 8.3: Component of the collection's framework.

The code above is mostly self-explanatory, but maybe a little explanation of the show method is necessary. The iterator () function is first implemented by the List, Set, and Map classes. This method returns an object of type Iterator, which may iterate across Lists, Sets, or Maps. Notice that this is not an easy operation since each collection might include a variety of things, but happily the hard work has already been done. As a result, a variable named "it" of the generic type Iterator is created in the show() function above. Then, an object that can iterate over our List of Strings is returned by using the function iterator() on our particular collection. Two very helpful methods are provided by the Iterator class: hasNext(), which returns true if there is another item in the collection, and next(), which provides the object that comes next in the collection. As a result, each item in the list is run again and shown in turn by the loop above.

The JRE does not know what sort of object is being received from the list since we are using an untyped collection, which means each item in the list may be of a different type. But, because we are just using this list to hold Strings, we can cast each object obtained onto a String. If we add non-String objects to the list, this will fail, resulting in a compiler warning. While this software manipulates and saves a list of Strings, it could just as well store a list of publications or any other object built using a class we define.

A Typed Collections Example: This is easier to do with typed collections for two reasons: 1) we don't need to use the cast operator because we know what type of object will be returned from the

list; and 2) Java 5.0 introduced an improved for loop that can iterate through every item in a typed collection without using a loop counter (or iterator). for example, see below:

```
import java.util.List;
import java.util.ArrayList;
public class ListDemo


{
	private List<String> mAList;


	/**
	* Constructor
	*/
	public ListDemo ()
	{
		mAList = new ArrayList<String>();
	}


	/**
	* Display list of strings
	*/
	public void display()
	{
		for (String nextItem: mAList)
		{
			System.out.print(nextItem + " ");
		}
		System.out.println();
	}
}
```

The process of generating a typed collection is fairly similar to that of constructing an untyped collection, with the exception that we must provide the contents of the collection when we create the instance variable and the actual collection (in this case it is simply a collection of Strings). The > syntax may be interpreted as meaning "of," or in this example, a list of strings. While adding, adding, and removing items in untyped collections is same, presenting the collection is considerably easier since we are aware of what each member in the list contains. There are just two import lines

above since we don't need to cast the returned items and we don't even require an iterator.

The next item in the list receives the automatic assignment of nextItem. Since we are utilising a list of Strings in this case, take note that nextItem is specified as a String variable. Because the compiler is aware of this, he or she will make sure the appropriate type of variable is utilised.

A Remark on Sets: While a detailed study of Sets is beyond the purview of this article, there is one Sets-related problem that must be mentioned. Duplicate objects cannot be saved in sets. Consequently, we must consider what qualifies as a copy of an item. Think about a group of bank accounts:

Now consider two bank accounts one for Mr Smith and one for Mrs Jones. A second account with the identical account number should not be able to be opened and added to a group of bank accounts. Despite the fact that Sets do not permit duplicate objects within them, Java will regard the two items below as separate objects until instructed differently since they each have distinct names (Account1 and Account2). To fix this issue, we must modify the equals() function in the Object class to assert that two objects are identical if their accountNumber matches. The steps are as follows:

```
public boolean equals (Object pObj)
{
      Account account = (Account) pObj;
      return (accountNumber.equals(account.accountNumber);
}
```

This replaces the Account class's Object.equals() method. While an Account object will always be supplied as a parameter, we must convert the argument to an Object type in order to override the equals() function that Object inherits, which has the signature

Boolean equals public (Object obj)

This function would not replace Object.equals if we signed it with an Account type argument (). To compare them with those of the present object, we must convert the argument to an Account before extracting its accountNumber. The storage of items in sets presents another challenge.

Java utilises a hashcode based on the object's name to do this. Yet, even if the object name is different, two accounts with the same accountNumber ought to produce the same hashcode. Just like we had to modify the equals() function, we must override the hashCode() method to ensure that a hashcode is created using the accountNumber rather than the object name. By altering this function as shown below, we can make sure that the hashcode produced is based on the account number:

```
/**
 * Override hashCode() inherited from Object
 * @return hashcode based on accountNumber
 */
public int hashCode()
{
      return accountNumber.hashCode();
}
```

The easiest approach to redefine an object's hashCode is to create a single string by joining all of the instance variables that establish equality, then calculate the hashcode of that string. In this instance, the accountNumber, which must be unique, serves as the only criterion for equality. Despite the fact that we are overriding the hashCode() function for objects of type Account, it seems a bit unusual, but we can still use it on this String. The same hash code will always be generated by hashCode() for the same String. Sometimes, various key values may result in the same hash code, but that is not an issue. Java will stop objects with duplicate data (in this example, duplicate account numbers) from being added to sets by overriding the equals() and hashCode() methods. For storing collections of objects, the Java "Collections Framework" offers ready-made interfaces and implementations. This almost eliminates the need for arrays.

There are "untyped" collections as well as "typed" collections as of JDK 5.0. List, Set, and Map are a few collection interfaces that each define the necessary operations. ArrayList, HashSet, and HashMap, which are implementations of the List, Set, and Map interfaces, respectively, are examples of collection implementations. When creating objects to be kept in Sets (or as keys in Maps), special care must be taken to clarify the meaning of "duplicate." We must override the equals() and hashCode() methods that are inherited from Object for these.

CHAPTER 9

JAVA DEVELOPMENT TOOLS

Dr Preethi D, Assistant Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- preethi.D@jainuniversity.ac.in

Software Development: Software must be created and installed on the system it will operate on before a computer may do beneficial activities for us, such as checking the spelling in our writings. The process of implementing software include developing the program's source code and getting it ready to run on a specific hardware. Naturally, the programme must be planned before it is created, and testing must be done at some point. To support the design, implementation, and testing processes, which require several implementation stages, there are numerous iterative lifecycles. The three well-known methods for getting source code to run on a certain computer are of special interest here.

Direct execution of source code by a "interpreter" software; compilation into machine-language object code, compilation into intermediate object code that the run-time system interprets. Java applications are implemented by compiling the source code (Java) into intermediate object code, which is then translated into human-readable language by a run-time system known as the JRE. To fully understand the consequences for the Java developer, it is important to weigh the pros and cons of these three approaches.

Compilation: For the appropriate hardware/OS combination, the compiler converts the source code into machine code. There are technically two stages: the compilation of program units (often files), then the process of "linking," in which the various program units, required library code, etc. are combined to create the final executable program. The built software is subsequently executed on the target platform as a "native" application. The first model, which is still used by many contemporary languages like C++ and historical ones like Fortran and Cobol. While it permits quick execution, every time the code is updated, the application must be recompiled.

Interpretation: Here, machine code is not converted from the original code. Instead, a program called an interpreter examines the source code and executes the commands it contains. The interpreter may be compared to a "virtual machine" whose machine language is the language of the source code. Even if the code may be changed without requiring a new compilation, the pace at which it is executed is significantly slowed down by the interpretation process. This is how scripting languages often operate.

Code Intermediate: This design is a cross between the first two.

Compilation transforms source code into a faster-running intermediate representation that may be run by a "run-time system" (again, a kind of "virtual machine") than when the source code is directly interpreted. The compilation process may be independent of the OS/hardware platform thanks to the usage of intermediate code, which run-time system software subsequently executes. As long as the right run-time system is available for each platform, the same intermediate code should run on

multiple systems. This method is well-known (for example, in Pascal from the early 1970s) and is how Java functions.

JRE: To execute Java applications we must first build intermediate code (called bytecode) using a compiler supplied as part of the Java Development Kit (JDK).

To run the Java library packages and bytecode, you need a version of the Java Runtime Environment (JRE), which includes a Java Virtual Machine (VM). Hence, a JRE is required on any system that executes Java applications. Java applications are very portable since the Java bytecode is standardised and platform neutral, and JREs have been developed for the majority of computer platforms (including PCs, laptops, mobile devices, mobile phones, internet devices, etc.).

Java Applications: Programmers may use a number of tools to develop source code regardless of the execution mode used. There are two options: either utilise an Integrated Development Environment (IDE), which integrates various implementation tools under a unified interface, or use simple discrete tools (such an editor, compiler, or interpreter) that can be manually launched as needed. The implementation process can be integrated with other stages of the development cycle using even more advanced CASE (Computer Aided Software Engineering) tools. For example, such software could take UML class diagrams and automatically generate classes and method stubs, reducing the time and effort needed to write Java code.

Each class (or interface) in a Java application gets its own name.java file with the source code when written in Java. The compiler transforms them into name. Class files that contain the matching bytecode. One of the classes must have a main () method with the above-described signature in order for the program to function as an application (Figure 9.1).

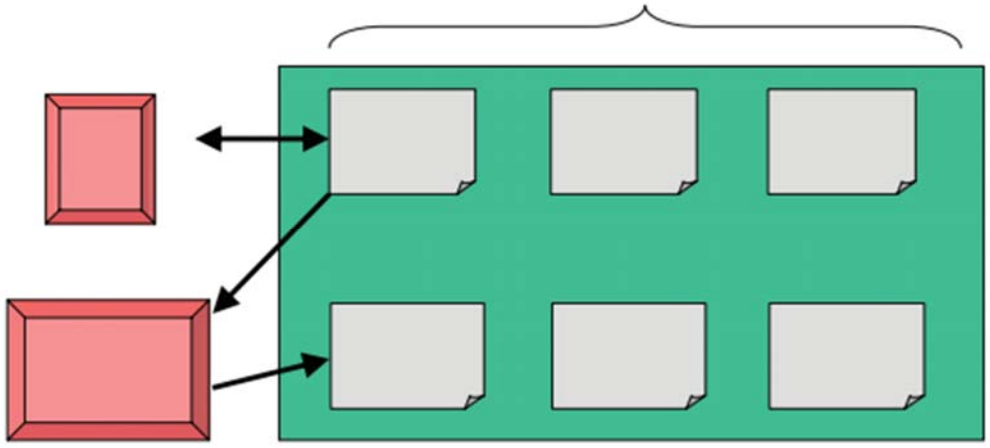


Figure 9.1: Integrated Development Environment

The Java Development Kit (JDK) must initially be installed in order to create Java applications (JDK). In order to add IDE capabilities, such as interface development tools, code debugging tools, testing tools, and refactoring tools, a Java IDE, such as Eclipse or NetBeans, sits on top of the JDK (more on these later). While using an IDE, it's easy to forget that a lot of the functionality is really a component of the JDK, and that the IDE is only relaying requests to the JDK running beneath when requested to build a programme. While we can create and execute Java applications straight from the command line using the JDK, most of the time it is

simpler to utilise the extra features provided by an IDE. It may be a little perplexing since Java's most recent release goes by the names 6.0, 1.6, and even 1.6.0. Don't worry; they are designed to have somewhat different meanings. The following are the top two fundamental tools: the Java compiler, or javac. Java is the launcher for Java programmes (that runs the VM). You type javac to build MyProg.java. MyProg.java. If successful, MyProg.class will be created. If MyProg contains a main() function, we can execute it by typing java MyProg. Another fantastic tool is javadoc, which automatically creates programme documentation from comments in Java source code.

Eclipse: switching to "industrial strength" IDE is a crucial step in your development as a software developer, similar to learning to ride a bike without stabilisers for the first time. After you get the hang of it, you'll realise that it has a tonne of time- and labor-saving features that you won't want to work without again. The IDE platform Eclipse is adaptable and expandable.

As Eclipse was created in Java, it requires a JRE to function like all other Java applications, but thanks to this, it can operate on a variety of operating systems. It comes pre-installed with the Java Development Tools (JDT), but it also offers add-ons and other tools to assist the creation of applications in other languages. Because of its adaptability, the Eclipse "runtime platform" may also be used as a foundation for the creation of applications unrelated to IDEs. Eclipse Architecture: Because to its reliance on the JRE, Eclipse is inherently cross-platform (Windows, Linux, Mac etc.) The basic Eclipse download includes the items shown in blue below (Figure 9.2).

Figure 9.2: Basic Eclipse download includes the items

There are many other plug-ins that support additional tools (such as style-checking, testing, GUI design, etc.) and languages (such as C/C++). Like any other IDE, Eclipse needs a current JDK installed in order to build applications and provide Java documentation.

Characteristics of Eclipse: A programmer looking to create extensive Java applications may take use of a number of capabilities offered by Eclipse. They consist of: sophisticated programme development tools, such as refactoring and automated testing facilities; code completion and help facilities; a code formatter that can automatically alter a wide range of code format features.

NetBeans: Another very potent IDE is called NetBeans. Originally created and sold by Sun Microsystems, NetBeans was turned open source in 2000 and has since received substantial support from the open source community in the form of plug-ins and instructional videos.

You may get NetBbeans for free at www.netbeans.org. Many features of NetBeans have been created to help new programmers and save time. They consist of online videos, tutorials, and a fast start guide (covering among other topics an introduction to the IDE, the Javascript debugger, and PHP support). Help functions with automated pop-up windows indicating pertinent Java API elements, Formatting and bug-fixing tools code completion tools that

automatically insert constructors, try-catch blocks, and for loops (among others). These code completion tools accomplish much more than simply cutting and pasting sample text blocks; for instance, when adding a subclass function Object() { }, the code to call a super-constructor is automatically added, and any pertinent parameters are defined for the sub-constructor and passed to the super-constructor.

Project management tools that keep track of changes to a project and enable it to be returned to a previous state. Moreover, Netbeans offers templates for a variety of Java applications, such as windows, corporate, web, and mobile ones. Even database programmes in C++, PHP, Ruby, and other languages use templates. As a result, Netbeans can quickly develop a simple Java application with a main function that launches a minimalist GUI, as shown below (Figure 9.3).

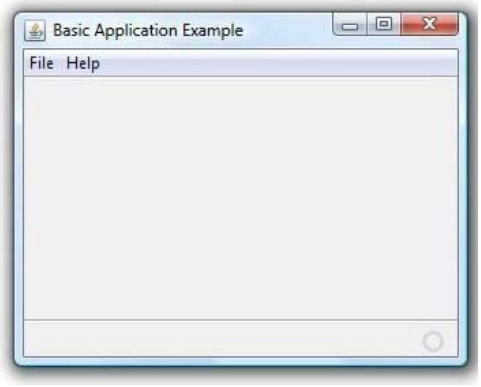


Figure 9.3: Create UML

The ability to create UML diagrams is another feature offered by one of the numerous plug-ins available (Figure 9.4). Even though it has several drawbacks, this tool offers features that you may anticipate from a CASE tool, such automated code development (from a diagram) and reverse engineering (i.e. analyzing current code to generate a UML diagram from the code).

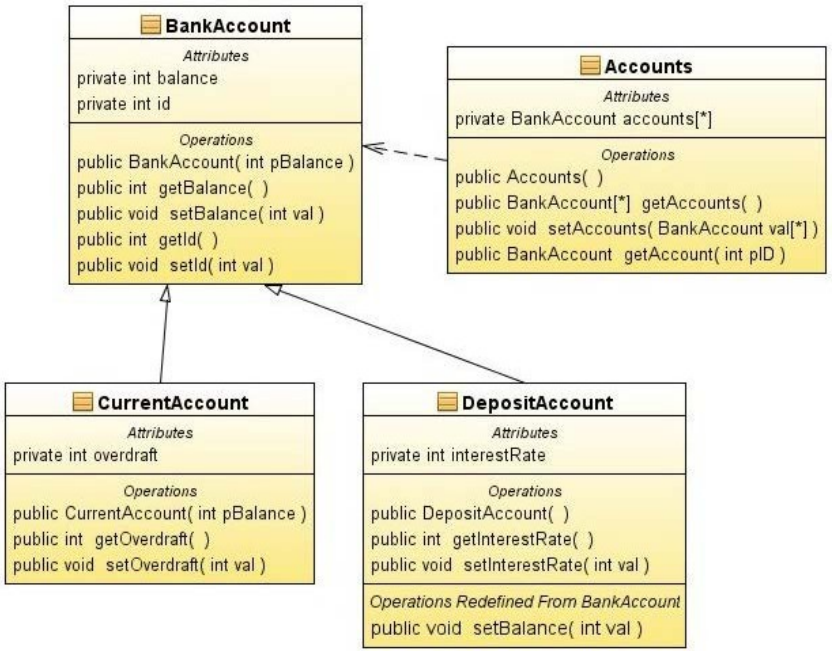


Figure 9.4: analyzing current code to generate a UML diagram from the code

In light of the aforementioned model, NetBeans would thus produce four classes, three of which would be inadvertently added to an inheritance hierarchy. The function Object() { } for CurrentAccount would automatically launch the function Object() { } for Bank Account sending the argument pBalance upwards, even though the actual code for each of the stated methods would still need to be written. Using NetBeans to Create Graphical User Interfaces NetBeans includes a visual tool to assist in the creation of graphical user interfaces. With the use of this tool, a window may be created and a variety of items, such as panels, tabbed panes, scrolling panes, buttons, radio buttons, check boxes, combo boxes, password fields, progress bars, and trees, can be put into it.

Using NetBeans to Implement Layout Managers: It is "usual" for elements to be positioned precisely on a window when building a graphical user interface. Hence, interfaces are shown precisely as they were intended. An interface designer typically has some concept of the size of the display they are building an interface for in order for this to operate successfully. The majority of PC games are designed to operate on windows 14" to 21". The majority of mobile games are designed to function on significantly smaller displays. The strength of Java is that Java code will run on a PC, laptop, or small mobile device regardless of the hardware or operating systems these devices employ. Java applications are, however, supposed to be platform agnostic (as long as they all have a JRE).

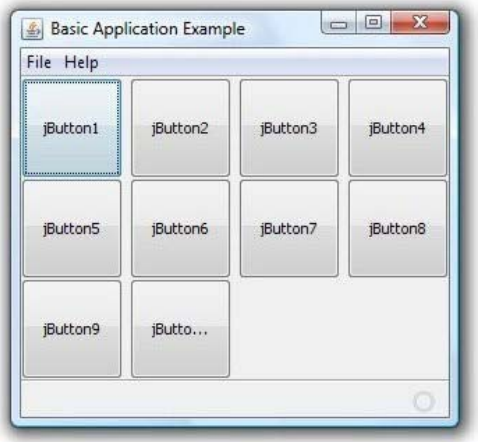
Hence, whether using a tiny mobile screen or a huge visual display unit, the same interface should function. For this reason when building interfaces in Java it is typical to give Java some discretion over determining precisely how / where to display objects. Java may thus modify a window at runtime to suit whichever device is being used to execute the application. The display will automatically adjust to match whatever device is being used to execute the software, which has advantages, but this flexibility has a price.

The interface designer can provide Java certain "instructions" on how an interface should be shown, but because Java is granted some control, they cannot be completely confident of how the interface will appear.

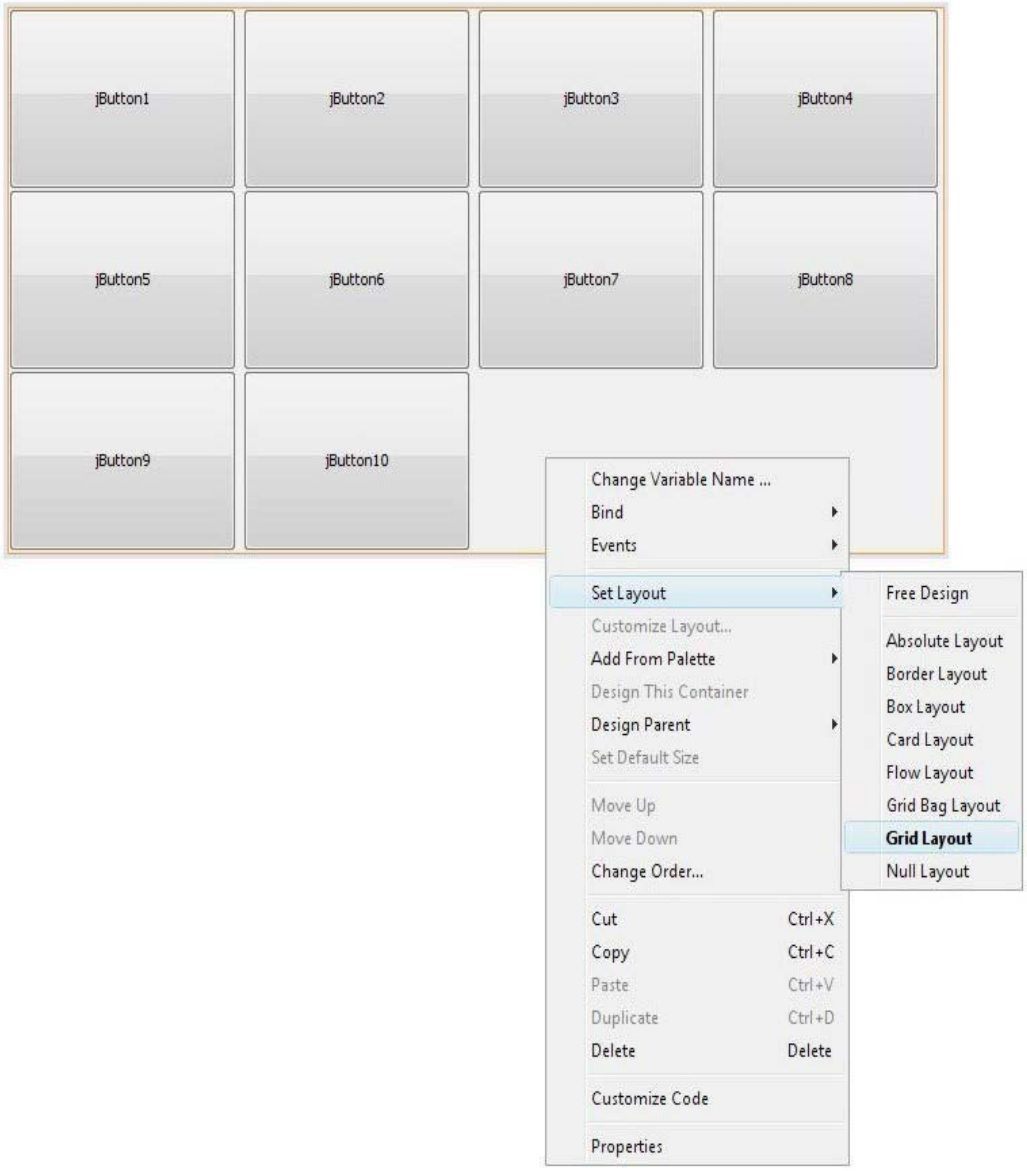
It is typical to build a layout manager and attach it to a "container" object like a window frame since layout managers are "objects" that specify how an interface should be shown. There are many popular layout managers; the two most popular ones are flow layout and grid layout. If the window is too small to fit all the items on one row, Java will automatically move the objects to the next row using flow layout.

A Flow Layout in Use: Here are two windows with 10 buttons each. In the second example, Java moves some of the buttons to a second row since the window was enlarged and the object no longer fits on one line.

In order to specify the object's justification (center, left, or right), as well as the horizontal and vertical distance between items, a flow manager may be established. Effect of Grid Layout: Gird layout is another widely used layout manager. A number of rows may be given when applying items to a grid. In order to fit the necessary number of objects into that many rows, Java will determine how many columns it needs to employ. The items will be scaled down to suit the window. Java will adjust the objects' size to suit the window, even if this means that some of the text cannot be shown in its entirety. The same 10 buttons are shown on a grid with three rows in the next two figures.

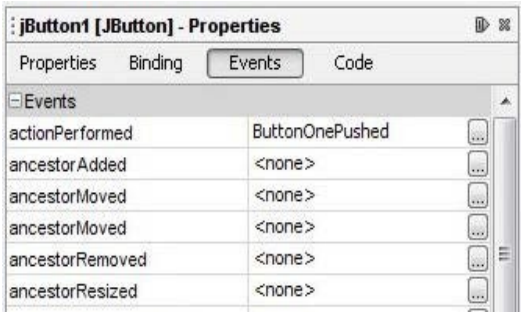


There is not enough room to show all of the text for button 10 since the buttons in the second of these have been resized to suit a separate window. We may design a grid where one element includes several items ordered in a flow by using layout managers, which can be applied to any container object and utilized in layers. Layout managers allow us to construct flexible displays that can run on a variety of display devices, giving platform independence for our applications. But we do give up some fine design control in the process. With NetBeans, creating a layout manager and configuring its attributes is fairly simple. Applying a layout manager will be one of the options in the context-sensitive menu that appears when you right-click on a container object. The list of available layout managers will be shown if you choose this option. After a layout manager has been chosen, a window will emerge to enable its attributes to be defined.



Implementing Action Listeners: Action Listeners must be created and applied to each item on the interface in order for it to function properly. When a user clicks the "calculate" button, for example, a software will compute and show some results. Action listeners are objects that listen for user inputs and reply appropriately. For every item on an interface, as well as for a variety of actions for each object, action listeners may be built. However many items, including labels, won't necessarily need to react to human interactions. It is quite simple to create an action listener in NetBeans.

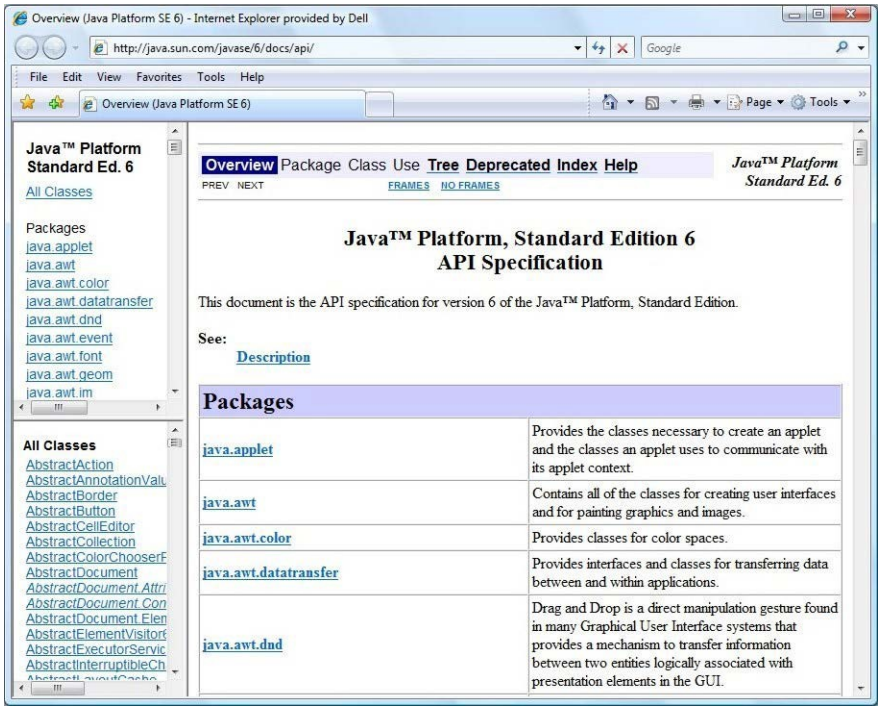
In design mode, a properties window appears when any item is chosen. This object may have a variety of actions generated for it by choosing the events tab. After chosen, an "Action Listener" stub will be generated. An object of this type will be automatically created and assigned to the design object that was chosen. The programmer still has to develop the code to specify how to react to the user's activity. In the example below, a listener event has been set up to watch for the pressing of button 1.



Javadoc Tool

Javadoc is one of the JDK's most helpful tools.

The arduous and time-consuming duty of producing documentation has been placed on programmers for many years. An explanation of what a programme does and how to use it may be found in some of this documentation that is aimed at users. Additional documentation is meant for programmers in the future who will have to modify and adapt the software so that it will perform differently when demands of an organisation change. These programmers need an understanding of the functionality and organisation of the application. They must be aware of What packages it includes; What classes are there in each of these packages and what these classes perform; What methods are present in each class; and for each of these methods, what the method does; what arguments are needed; and what result, if any, is return. While it won't create a user manual, the javadoc tool will provide a technical overview of the application. The programme analyses *.java source files and generates documentation as a collection of web pages (HTML files) in the same style as API documentation on the Sun website.



The Java language standard, including all packages, classes, methods, all method arguments and return values, is provided by this website as a series of indexed web pages. This really helpful information was produced using the javadoc tool rather than by hand.

The same tool may generate documentation in the same format as any Java application. Nevertheless, it depends on well structured (and educational!) javadoc-style comments in source files, using tags like @author, @param, etc., to do this. Programmers are spared from a difficult, time-consuming, and error-prone operation since this documentation is created automatically at the touch of a button. Also, if a programme is altered, the documentation may be updated. When the tool is being built, a programmer must include significant comments in the source code; otherwise, the javadoc output will be subpar. On the other side, the reference documentation is created "for free" if the commenting is done appropriately. Using the usual tags @author and @version, Javadoc comments should be added to each class at the beginning to provide an overview of that class in the following way.

```
/**********************************
* A description of the class
*
* @author name of author
* @version details of version (and date).
**********************************/
```

A similar comment should be provided for every method using the @param and @return to describe each parameter and to describe the value returned. The details of each parameter, starting with the name of the parameter, should be provided on separate lines as shown below.

```
/**********************************
* A description of the method*
* @param nameOf1stParam description of first parameter
* @param nameOf2ndParam description of second parameter
* @return description of value returned
**********************************
```

The description of a method cannot be examined by the javadoc tool to see whether it accurately summarises that method. The tool, however, is capable of far more than just pasting a programmer's comments into a web page. To check for logical problems, this tool analyses method signatures and compares them to the tags in the comments. If a method needs three arguments but the comment only mentions two then this problem will be recognised and reported to the programmer.

Warning: "pNumber3" is not a valid parameter name in the @param argument." The tool also offers further details about the class, the inherited methods, and the methods that have been overridden by analysing the code. The reference documentation created is intended for programmers using the relevant class(es); it does not include comments inside of the methods intended for programmers updating the source code of the class during maintenance work.

Open the file index.html in a web browser to see the created documentation, which may be stored in a subdirectory. Since only those classes and members are accessible to other programmers, "javadoc" will only document those that are public or protected. Things that are off-limits or for which no access restrictions are specified are excluded

There are several methods in which source code in a computer language may be executed. A main function is necessary because Java employs an intermediary language known as "bytecode." The Java Development Kit's tools and a text editor allow us to create Java

programmes in a "back to basics" manner, but employing a professional IDE that sits on top of the JDK may provide extra programming assistance features. There are specialised tools for development tasks like GUI design, diagramming, and documentation, but some IDEs go above and above by including some of these features right within the IDE.

Java programme development is supported by two strong open source IDEs, Eclipse and NetBeans, as standard. Both of these are accessible as free downloads. There is online assistance available, and both tools offer some level of automatic code generation, such as the main method, to make the job of the programmer easier. Despite the fact that these tools offer extensive support for professional development, including code formatting and refactoring, they can initially seem intimidating.

The Javadoc tool is an excellent valuable and timesaving technology but it does need the programmer to adding relevant Javadoc style comments into the code (for all classes and all public methods) (for all classes and all public methods).

CHAPTER 10

CREATING AND USING EXCEPTIONS

Ms. Sonali Gowardhan Karale, Assistant Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- gk.sonali@jainuniversity.ac.in

An exception is an unintended or unexpected occurrence that takes place while the program is running and disturbs its normal course. For instance, there would be an exception if a user attempted to divide an integer by 0 because it is technically impossible to do so. While running any program, there are many different interruptions, including bugs, failures, and exceptions. Programming errors or system problems may be to blame for these hiccups. They may fall within the categories of mistakes or exceptions depending on the circumstances. In Java, there are classes for handling built-in exceptions and features for adding user-defined exceptions. Let's start by defining the terms "exception" and "term mistake."

Exception

As was already discussed, exceptions are unwelcome circumstances that interfere with the program's flow. Exceptions often happen as a result of the code and can be fixed. There are two types of exceptions: checked (exceptions that the compiler checks) and unchecked (exceptions that the compiler cannot check). They can happen at build time as well as run time. Exceptions in Java are a member of the java.lang.Exception class.

Error

1. Even while errors are undesirable conditions, they are also caused by a lack of resources and are a sign of more significant issues.
2. Errors are unrecoverable, and programmers are unable to address them.
3. Errors only come in the unchecked variety.
4. They can only happen during run time.
5. Errors in Java are members of the java.lang.error class.
6. For example, out of Memory Error

Types of exception in Java

Checked Exceptions

1. Checked exceptions are those that are evaluated during compilation.
2. With the exception of Runtime Exception, they are all children of Exception.
3. If they are not handled, the software won't build.

Example: Class Not Found Exception, IO Exception, etc.

Unchecked Exceptions

1. Unchecked exceptions are those that are not checked at runtime.
2. They are Runtime Exception's offspring classes.
3. They provide runtime faults if not expressly handled.
4. Examples include Null Pointer Exception and Arithmetic Exception.

Java Exception Handling:

Java's exception handling contributes to exception minimization and exception recovery. It is one of the effective methods for handling runtime errors and ensures that the program is bug-

free. Exception handling aids in keeping the program's flow intact. An abnormal circumstance that could arise during runtime and impede the program's regular flow is referred to as an exception.

Keywords for Java Exception Handling

Java has particular terms for managing exceptions.

Throw: We are aware that when an error happens, an exception object is produced, and the Java runtime begins handling the error. We may occasionally wish to intentionally create exceptions in our code. For instance, if the password is null in a user authentication application, we should raise errors to the clients. To send exceptions to the runtime for handling, use the throw keyword.

Throws: To inform the calling program of potential exceptions that the method may throw, we must use the throws keyword in the method signature when we throw an exception without handling it. Using the throws keyword, the caller method may handle these exceptions or pass them along to its caller method. The throws clause allows us to specify a number of exceptions, and it may also be used with the main () function.

Try-catch: Our code handles exceptions using the try-catch block. The catch statement comes after the try statement at the conclusion of the block to deal with exceptions. With a try block, we may have numerous catch blocks. You may nest the try-catch block as well. The argument for the catch block must be of type Exception.

Finally: only a try-catch block may be used with the optional finally block. . Since an exception stops the execution process, the finally block can be used if we have any open resources that won't be closed. Whether or not there was an exception, the finally block is always carried out.

Java Exception Hierarchy

As previously mentioned, an exception object is produced when an exception is triggered. Java exceptions are categorized using inheritance since they are hierarchical in nature. The parent class of the Java Exceptions Hierarchy is called Throwable, and its two children are called Error and Exception. Checked exceptions and runtime exceptions are additional categories for exceptions.

Errors: Errors are extraordinary events that fall outside the purview of application, and it is impossible to foresee and recover from them. For instance, a hardware malfunction, JVM (Java Virtual Machine) crash, or memory problem Because of this, we have a different hierarchy of errors, and we shouldn't attempt to manage these circumstances. Out Of Memory Error and Stack Over flow Error are two examples of frequent errors.

Checked Exceptions: In a program, checked exceptions are extraordinary events that we can foresee and attempt to recover from. A File Not Found Exception is an example. We should catch this issue, provide the user a helpful message, and appropriately report it for troubleshooting. All Checked Exceptions belong to the parent class known as The Exception. If a Checked Exception is being thrown, it must be caught within the same procedure or propagated to the caller using the throws keyword.

Runtime exception: Poor programming is the root cause of runtime exceptions. Attempting to obtain an element from an array, as an example. Before attempting to obtain an element, we should first determine the length of the array to prevent an Array Index out Of Bound Exception from being thrown at runtime. All Runtime Exceptions belong to the parent class known as Runtime Exception. It is not necessary to include a throws clause in the method signature if

any Runtime Exceptions are being thrown within the method. Better programming can prevent runtime exceptions .

Exceptions Try-with-Resources and User-defined Exceptions

Exceptions Try-with-Resources: In Java, a try statement that declares one or more resources is known as a try-with-resources statement. When your program is finished utilizing an object, it must be closed, which is known as a resource. A File resource or a Socket connection resource, for instance. At the conclusion of the statement execution, the try-with-resources statement makes sure that each resource is closed. If we don't seal the resources, there may be a resource leak and the program might use up all of the resources at its disposal.

Exceptions: There are differences between a try-catch-finally block and a try-with-resources block when it comes to exceptions. When an error occurs in both the try and finally blocks, the function returns the error that occurred in the finally block. If an exception is thrown both in a try block and a try with resources statement, try-with-resources, the method returns an exception thrown in the try block. Since the exception thrown by the try-with-resources block are suppressed, we may argue that they are suppressed exceptions .

Advantages of using try-with-resources:

1. The resource can be closed without using the finally block.
2. Use numerous resources while trying with them

Try-with-resources Exception Handling

A Java try-with-resources block's exception handling semantics differ slightly from a try-catch-finally block's exception handling semantics. Even if you don't fully grasp the differences, the new semantics will often benefit you more than the semantics of the original try-catch-finally block. Even so, it may be a good idea to properly grasp how the try-with-resources construct handles exceptions. I will thus go through the try-with-resources construct's exception handling semantics in this section. Any resource opened inside the parenthesis of a Java try-with-resources block will still shut immediately even if an exception is triggered from within it. Throwing the exception compels the execution to exit the try block, which compels the resource to automatically close. When the resources are closed, the exception that was thrown from within the try block will go up the call stack.

When you attempt to close some resources, exceptions could also be thrown. Any resources opened within the same try-with-resources block will still be closed even if a resource throws an exception when you attempt to close it. The exception from the unsuccessful close-attempt will propagate up the call stack after all resources have been closed. If repeated resource shutdown attempts result in exceptions, the first exception that is encountered will be the one that propagates up the call stack. The remaining exceptions will be disabled. The exception raised inside the try block will move up the call stack if it occurs both inside the try-with-resources block and when a source is closed (when the call to close() is made). The resource closure attempt exception that was thrown will be silenced. In a typical try-catch-finally block, the exception that was last detected is the exception that is transmitted up the call stack .

User-defined Exceptions: The Java user-defined exception is a special exception that is constructed and is thrown using the keyword "throw." By extending the class "Exception," it is possible. An issue that occurs while the program is running is referred to as an exception. Java's Object-Oriented Programming language offers a potent technique to deal with these exceptions. Java enables the creation of custom exception classes, each of which offers a unique exception class implementation. User-defined exceptions or custom exceptions are the names given to such exceptions.

Utilize Custom Exceptions: Custom exceptions will shine a light on exception handling even though Java has the Exception class, which covers practically all instances of exceptions that might be produced during program execution. The flexibility of adding messages and methods that are not a part of the Exception class is made possible by custom exceptions. They can be used to override an existing method to deliver the exception according to the use case or to contain case-specific messages like status codes and error codes. When developing exceptions for business logic, custom exceptions are useful. It aids in the better understanding of the business-specific exception by the application developers. To build custom exceptions in java now that we are aware of what they are, how to use them, and when to use them. Nearly all of the common types of programming exceptions are covered by Java exceptions. But occasionally we have to make our own exceptions. A few justifications for using custom exceptions are as follows:

1. To detect a subset of current Java exceptions and provide them particular handling.
2. Exceptions to business logic these are the business logic and process exceptions. Understanding the precise issue is helpful for both app users and developers .

The creation of Java User-Defined Exceptions:

Java requires that we build a class and extend it with the Exception class from the java.lang package in order to generate a custom exception. To make our exception class's benefits and identification easier to understand, it's usually a good idea to include comments and adhere to naming conventions. The Java template for writing a user-defined exception is provided below.

Pass the Exception Message to Super Class Constructor and Retrieve It Using the getMesssage () Method Override the toString() Method and Customize It with Our Exception Message.

Requirement of Custom Exceptions:

Nearly all common exceptions that are certain to occur when programming are covered by Java exceptions. However, there are situations when we need to add our own exceptions to these common ones. The basic justifications for implementing custom exceptions are as follows:

Business logic exceptions are exceptions that are unique to the workflow and business logic. These aid in clarifying the precise nature of the issue for both application users and engineers. To detect a subset of current Java errors and treat them specificallyIt is possible to check and uncheck Java exceptions.

Utilizing these user-defined exceptions has benefits since it enables users to throw an exception that means whatever they want it to. Users may also utilize any already-existing program; any code that catches exceptions deals with the possibility that the real exception was actually produced by some other third-party code rather than by the code itself. Users can classify and differentiate error kinds by separating Error handling code from Regular code. There are guidelines for developing Exception classes as well.

Constructor: Adding a function Object () to the custom exception class is not required. It's a good idea to provide parameterized constructors in the custom exception class.

Naming Standard: Since the JDK end provides all exception classes, a custom exception must adhere to a naming convention.

Extends Exception class: If a user is developing a unique exception class, they must extend Exception class.

Recognizing the Value of Exceptions: Writing Java applications requires a crucial understanding of exception management. The file handling classes in the Java language were aware of this and developed routines that used the Java exception handling capabilities. The creator of the getClient()

function opted to return a NULL result in this situation since they were well aware that a client may not be discovered when they wrote the method. Yet for this to work, any programmer who ever used this technique must be aware of and prepared for this possibility. The software may crash, perhaps costing the bank a lot of money, if any programmer using this approach neglected to safeguard against a NULL return. Of course, a software crash might have fatal consequences in other applications, like an aviation control system. To make sure that a possible crash scenario is always handled, a more secure programming approach is needed.

There is such a system; it is known as a "exceptions" mechanism. With the help of this method, we can make sure that other programmers who use our code will be informed of any possible crash conditions, and the compiler will make sure that these programmers address the "problem". In this way, we can guarantee that no such circumstance is "forgotten." A programmer using our methods may decide how to handle them, but the compiler will make sure they at least recognise a crash circumstance. The getClient() function should throw an exception in the aforementioned scenario rather than returning a NULL result. The Java compiler will make sure that this circumstance is handled by raising an exception. Several Exceptions: There are two Exception classes built into the Java language that we must enhance in order to produce meaningful exceptions (normal exceptions and run time exceptions).

Subclasses of java.lang.: Exceptions are used for situations that need to be controlled and are expected. They must be declared in the throws clause of the original method, and a call to the method must be made in the try/catch block.

Subclasses of java.lang: RuntimeExceptions are utilised in circumstances when a runtime failure occurs and no practical corrective action may be available. They are not required to be stated in the throws clause or the try/catch block (but can be). So, we have the option of deciding whether the Java compiler should require us to explicitly handle a certain kind of exception. For situations when we can take appropriate corrective action and if we are aware that anything can go wrong, such as IO errors, exception subclasses are suitable.

Runtime Exception: subclasses are suited for situations when nothing should go wrong and there is likely nothing we can do to fix it, such as when memory runs out or the system is found to be in an inconsistent state that shouldn't be possible to occur.

Expanding the Exception Class: We should be on the lookout for probable failure scenarios while developing our own solutions (e.g. value that cannot be returned, errors that may occur in calculation etc). We should create a "Exception" object—an object of the Exception class—when a probable problem occurs. Nevertheless, it is preferable to construct a specialised class first, declare a subclass of the generic Exception, and then throw an object of this kind. A new exception is the same as a new class; in this situation, java.lang is a subclass of the new exception.

Exception: In the scenario above, if a client matching the requested ID cannot be located, an error may result. A new exception class named "Unknown Client Exception" might be created as a result. The function Object() { } for Unknown Client Exception likewise needs a String since the argument to the function Object() { } for the Exception requires one. This string is used to describe the issue that might lead to an exception. This is the code to create this new class:

```
import java.lang.Exception;


/*******************************************************

 * Exception thrown when attempting to get an non-existent client ID
 *
 * @author Simon Kendal
 * @version 1.0 (11th July 2009)
 *******************************************************/
class UnknownClientException extends Exception

{
      /**

      * Constructor
      *

      * @param pMessage description of exception
      */

      UnknownClientException (String pMessage)

      {

            super(pMessage);

      }

}
```

This seems to be a little strange in several ways. While we are here building a subclass of Exception, our subclass neither adds any new methods nor replaces any already existing ones. As a result, its functionality is the same as that of the superclass, despite the fact that it is a subtype with an interesting and evocative name. We would only be able to catch the most generic sort of exception, which is an Exception object, if subclasses of Exception did not exist. Hence, the only catch block we could create would have to capture every single exception.

Having specified a subclass, we now have a decision to make. A catch block may be defined to capture objects of the generic type "Exception," i.e., it would catch ALL exceptions; alternatively, a catch block could be defined to handle UnknownClientExceptions but ignore other exception kinds. We can observe that there are several specified subclasses of exceptions by looking at the web API. There are plenty of these, such as:

1. IOException
2. CharConversionException
3. EOFException
4. FileNotFoundException
5. ObjectStreamException
6. NullPointerException
7. PrinterException
8. SQLexception

Hence, instead of writing a catch block that would respond to any exception, we might restrict it to input and output errors or, to be even more particular, FileNotFound exceptions. Throwing Exceptions: After creating our own exception, we must tell the getClient() function to raise it (assuming a client has not been found with the specified ID). In order to do this, we must first inform the compiler that this class could produce an exception; the compiler will then make sure that any future programmers who use this method catch this exception. We add the phrase "throws UnknownClientException" to the method's signature to inform the compiler that it throws an exception.

```
public Client getClient(String pClientID)
                                        throws UnknownClientException
```

The throw keyword must be used on the newly formed object and a new instance of the UnknownClientException class must be produced. To throw an exception at the proper spot in the method's body, we utilise the keyword "throw."

```
if (foundClient != null)
{
 return foundClient;
}
else
{
 throw new UnknownClientException("BookOfClients.getClient():
                                        unknown client ID:" + pClientID);
}
```

If a client is discovered in the example above, the method will return the client object. The return value will no longer be NULL, nevertheless. Instead, the function Object() { } for UnknownClientException is used when a client cannot be located. This function Object() { } takes a String argument - and the text we are supplying here is an error message that is aiming to be instructive and helpful. As stated in the message:

BookOfClients, the class that produced the exception, its getClient() function, some text describing the problem's root cause, and the value of the argument for which no clients could be located. We are alerting any methods invoking the getClient() function that an error can be raised by declaring an UnkownClientException and utilising the throw clause in the method's header. This gives the Java compiler the ability to ensure that a try/catch block is present when necessary.

By doing this, we avoid having potentially serious mistakes go unreported!

Catching Exceptions: We are requiring other programmers to defend against catastrophic mistakes by requiring calls to this method to be made within a try/catch block by telling the compiler that this method may raise an exception. If an exception is raised, the code in the try

block will be stopped and the code in the catch block will start in its place. The awardLoan() function may thus choose what to do in the example above if a customer matching the supplied ID cannot be located.

```
try
{
      Client c = listOfClients.getClient(clientID) ;
      c.determineCreditRating();

      // add code to award or reject a loan application based on this
         credit rating

}

catch (UnknownClientException uce)
{
      System.out.println("INTERNAL ERROR IN BankManager.awardLoan()\n"
                                    + "Exception details: " + uce);

}
```

Now, if the client ID is not found, the programme will handle the UnknownClientException we intentionally threw rather than crashing with a NullPointerException. The Java Virtual Machine will then stop the code in the try clause and call the code in the catch clause, which in this case will display a message alerting the user to the issue. Java exceptions provide a way to handle unusual circumstances that arise during programme execution. We are defending against possibly fatal software failure by using the Java exception mechanism. By using our approaches, other programmers will be able to recognise and respond to fault scenarios thanks to the exception system. The code in a catch block will be started when an exception is thrown; this code may perform a corrective action or end the programme, producing the appropriate error message. In either scenario, the software at least doesn't just "stop."

CHAPTER 11
AGILE PROGRAMMING

Dr. Ramkumar Krishnamoorthy, Assistant Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- ramkumar.k@jainuniversity.ac.in

Classical software development methods prioritised thorough preparation in advance and a logical progression across the programme lifecycle. Do it properly the first time, not late. Modern "agile" development methodologies place a focus on flexible cycle development with the system developing towards a solution. Write your code early, then repair and enhance it as you go. Now, this is a very hot issue in the world of software engineering, and like every new development, it has its share of zealots and idealists. Agile methodologies, according to its proponents, better reflect the realities of software development. Agile development, however, needs tools that will allow software to adapt and change. Refactoring and testing tools are two particular tools offered by contemporary IDEs that help agile programming.

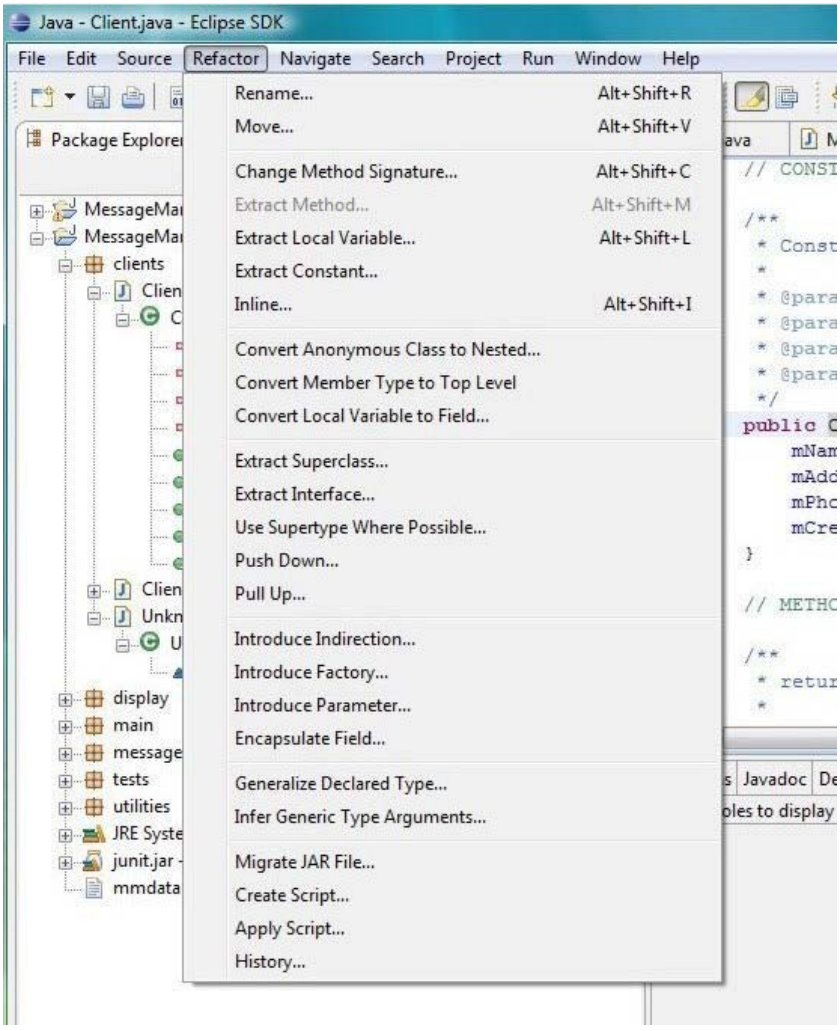
Refactoring: Refactoring is a crucial component of "agile" methodologies. This approach recognises that some initial design and execution choices will be subpar, if not worse. Refactoring is the process of making changes to a system to enhance its design and implementation quality without affecting its functioning (preventive maintenance was the conventional word for this activity in traditional development). While the notion of fundamentally upgrading old software is not new, the distinction is as follows. It was seen in conventional development as a corrective measure used when the quality of the programme design has declined, often as a consequence of stages of functional change and expansion. Refactoring is seen as a natural, healthy component of the development process in agile techniques.

Refactoring Examples: A programmer may discover that a variable inside a programme has been poorly named throughout the development process. Yet altering this is not a simple procedure. If a variable with the same name already exists in another method, updating a local variable will only need modifications in that function. Alternately, altering a public class variable can need system-wide modifications (one reason why the use of public variables are not encouraged). So, it's important to understand the effects of a change before adopting it, even if it seems little. Further, more complicated adjustments could be necessary. They consist of. Renaming an identifier wherever it appears, moving a method from one class to another, separating code from one method into a different method, altering a method's parameter list, and moving class members around in an inheritance tree are all examples of renaming identifiers.

Refactoring Support: To ensure that all the essential changes are done consistently, even the smallest refactoring action, such as renaming a class, function, or variable, needs thorough examination. For this action, Eclipse offers advanced automated assistance. Eclipse comprehends the Java grammar and operates intelligently, so don't mistake this with a straightforward find/replace

in a text editor. For instance, if you rename one of two local variables with the same name in two distinct methods, Eclipse is aware that the other variable has a different name and does not update it. While methods from these classes may access and utilise this variable, renaming a public instance variable may need modifications in other classes and even in other packages.

Another refactoring task is to rearrange existing classes inside a package hierarchy Eclipse makes doing this extremely simple. Similar to how we may sometimes determine a class has become too big and complicated and decide to break it into two or more different classes as a system design matures, we may also decide to split a package. Less often, we may relocate classes across existing packages or combine packages. We can move a class from one package to another using Eclipse. When this occurs, the class's package statement is modified at the beginning, and import statements inside other classes are immediately altered to reflect the fact that the class in question is now located in a new package. The following screenshot displays the refactoring choices offered by the Eclipse IDE.



Automated testing tools are another another crucial resource offered by contemporary IDEs.

Unit Testing: Unit testing is the process of testing each method of a class separately from its intended context inside the developing system. The implementation process is often "wrapped" around this testing:

1. Create a class
2. Create exams
3. Execute tests, fixing issues as appropriate

Automatic unit testing

The unit testing process may be automated using various tools and frameworks. Generally speaking, using such a programme entails a bit more work than doing tests once manually. The flexibility to just press a button and repeat the tests as much as needed is the biggest advantage, however. This is a huge help for the "regression testing" that has to be done anytime code that has already been tested is updated. Automated unit testing demonstrates that regression testing was created before agile approaches were suggested. Yet, Test Driven Development techniques, which are crucial to the agile software development methodology, are also supported by automated unit testing.

Regression Analysis

Regression testing is necessary when software is modified to accommodate changing business requirements. We want to make sure that code changes do not "break" already-existing functionality, therefore we perform regression tests. To do this, we must simultaneously create test cases that show how the classes function. So, we go through a process of...

1. Create a class
2. Create exams
3. Execute tests, fixing issues as appropriate
4. Create additional lessons

We must re-run all prior tests to ensure they still pass if we feel it's essential to alter any of the earlier classes (or classes on which they rely) due to defects, changing user requirements, or refactoring. Any update of the old code without regression testing is highly risky. It is beneficial and very time-saving to have automated testing capabilities, such as those that exist inside Java, since we often need to rerun groups of test cases.

JUnit

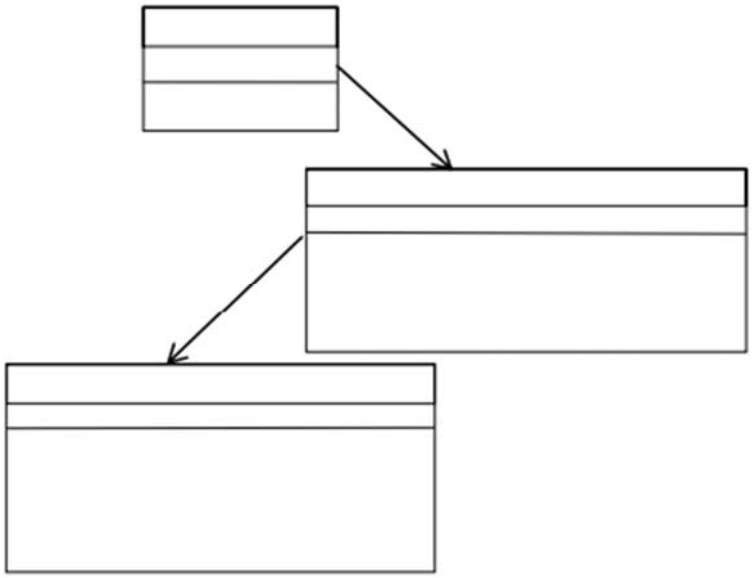
JUnit is a Java unit testing framework that is quite popular. It was created as a collection of open-source classes and is included in well-known Java IDEs like Eclipse and NetBeans. While JUnit offers an automated testing framework, the test cases still need to be put up. Once these have been done, however, a thousand test cases may be performed at the touch of a button, and the same set of test cases can be repeated each time a programme is changed. JUnit test classes are created as junit.framework.TestCase subclasses.

Using assert () methods, the right behaviour of the code under test is verified, and it must be true for the test to succeed. JUnit versions 3.8 and 4 are now in widespread usage, however they vary significantly from one another. Here, we're use the more well-known Junit 3.8. Illustrations of Assertion: Assertions are made while creating test cases. An assertion is a claim that, if true, indicates that the code has worked as intended. An example of an assertion is:

1. assertTrue(…)
2. assertFalse(…)
3. assertEquals(…)

4. assertNotEquals(…)
5. assertNull(…)
6. assertNotNull(…)
7. fail()

As we shall see, assertEquals() and fail() are suitable for various testing needs. Reaching this line implies the test failed, according to the function fail(). Many Test Cases: We will develop three test cases to check the functioning of a BankManagment system, as shown below, in order to demonstrate the JUnit testing framework:-



A BankManager class in this system keeps track of a "map" of customers. The addClient() function, which needs an ID for both the client and the client object to be added, may add client instances. The getClient() function, which takes a ClientID as an argument and returns a client object (if one exists) or throws an error, may be used to get clients (if a client with the specified ID does not exist).

The function Object() { } of the Client class asks for the client's name and address in addition to providing accessor methods. We'll precisely check your capacity to. Add a client to the BookofClients, search up an invalid client, or use the string output by the function toString() { }() function of the Client class. Of fact, this is only a tiny portion of the test cases that would be required to fully illustrate how all classes and all methods inside the system should operate.

Initiating a client test: We must construct a JUnit test case that allows us to verify that a client may be added.

1) Generates a fresh, blank Book of Customers.

2) Adds a newly created customer to the Book of Clients.

Finally, we need to verify that the client we just added has the same properties as the client we just obtained to make sure it wasn't corrupted. 3) Attempts to get a client with the same ID as the client just added (this, of course, should work).

The code is provided below:

```
public void testAddClient() {

      BookofClients bofc = new BookofClients();
      Client c = new Client("Simon",

                              "No 5, Main St., Sunderland, SR1 0DD");

      Client c2 = null;
      bofc.addClient("SK001", c);
      try {

            c2 = bofc.getClient("SK001");

      } catch (UnknownClientException uce) {

            fail();

      }

      assertEquals("Simon", c2.getName());
      assertEquals("No 5, Main St., Sunderland, SR1 0DD",

                                                  c2.getAddress());

}
```

The name of a test method will always start with test ()

The test fails if an UnknownClientException is raised since we were supposed to find the newly added "SK001" client. If we are able to successfully locate the client, we assert that the values returned by getName() and getAddress() should both be "Simon" and "No 5, Main St., Sunderland, SR1 0DD," respectively. The test will not pass if one of these conditions is not met. The anticipated value and the actual value are the two arguments of the assertEquals() function, it should be noted. Even though the assertion's result will be the same if they are switched around, JUnit will show the test's failure inaccurately. The test is successful if the test procedure is completed without any assertions being false. Keep in mind that the Client class's accessor methods getName() and getAddress() are required for this test to function. Even though they are not needed in the real system, it is standard practise when building classes to include accessor methods that were created only to enable unit testing.

Tests for an unidentified customer: We need to verify, for example, that if we attempt to get a client that doesn't exist, an exception is raised. To test this, we create a brand-new BookofClients that is empty and attempt to get any client from it.

The code is provided below:

Because we haven't added the client, we anticipate an UnknownClientException to be raised in this situation. and the test fails if it reaches the line following the call to getClient().

The catch block has nothing to do and the test passes if the exception is caught, skipping the fail() step.

```
public void testGetUnknownClient() {
      BookofClients bofc = new BookofClients();
      try {
            bofc.getClient("SK001");
            fail();
      } catch (UnknownClientException uce) {
      }
}
```

The function toString() { }() method test

Testing if the function toString() { }() function delivers the desired result is one approach of indirectly verifying that ALL the characteristics of a client have been saved properly. This is a bit challenging since the string's format must match perfectly, down to the last space, punctuation mark, and newline. Testing the value that each accessor method returns is an option, however. Here, we make the claim that the string produced by c.toString() satisfies our expectations. This is challenging since the string's format must match perfectly, down to the last punctuation mark and newline. This test is helpful since it indirectly verifies that ALL of the characteristics have been accurately recorded.

Conducting tests

After creating a batch of test cases, we must set them up so that they may be executed whenever necessary with the click of a button. To do this with Eclipse, we must first build a new package called "tests," then a new JUnit test case called "ClientTests," and finally the three test methods.

Run A JUnit test: Edit the code to make one of the tests fail, then review the outcome

Development that is test-driven (TDD)

The practise of Test Driven Development (TDD), which is mostly related to "agile" development techniques, is supported by automated unit testing. In software engineering, this has recently become a hot issue.

The goal of the Test Driven Development method is to

1) Create the exams (before writing the methods being tested).

2) Set up automated unit testing, which fails as a result of the incomplete classes!

3) Construct the classes and methods to pass the tests.

This reversal first looks unusual, but many distinguished participants in software engineering discussions think it represents a significant "paradigm shift." One may compare the endeavour to teaching. Which of the following is easier?

1. Educate someone on all you know about a topic before deciding how to assess their understanding;
2. Specify precisely what they must learn (i.e., pick what to test), and then impart to them the information they need to pass the exam.

Cycles TDD

The method being tested doesn't exist, hence when using test-driven development, the test will initially result in a compilation error!. While a stub method permits the test to build, the test

will still fail since the method does not include the real functionality being tested. The approach is then put into practise correctly so that the test is successful. To build up a sophisticated method, we could go through numerous cycles of: writing test, failing, implementing functionality, passing, extending test, failing, extending functionality, passing.

Requests for TDD

A few of the benefits of TDD include:

Testing stops being a last-minute afterthought and instead becomes an integral component of development. A thorough set of unit tests is created concurrently with the code development, encouraging programmers to produce straightforward code that directly addresses requirements. A quick cycle of "write test, write code, run test," each for a tiny developmental increment, boosts programmers' productivity. If a software project goes behind schedule according to traditional software lifecycles, financial demands often lead to the programme being pushed to market without having been thoroughly tested and debugged. This is not achievable with test-driven development since the tests are developed before the system is put into use.

Agile development methodologies place an emphasis on flexible, iterative cycles where the system develops towards a resolution. Regardless of the kind of development method used, unit testing is a crucial component of software engineering practise. As system development advances, unit tests may be repeatedly run using an automated framework (like JUnit). Test-Driven Development, which is a key component of "agile" techniques, flips the usual order of creating code and tests.

CHAPTER 12

FILES AND CONNECTING TO DATABASE

Dr. Ananta Charan Ojha, Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- oc.ananta@jainuniversity.ac.in

About all of the classes you might possibly need to accomplish input and output (I/O) in Java are included in the java.io package. An input source and an output target are represented by each of these streams. Several types of data, including primitives, objects, localised characters, etc., are supported by the stream in the java.io package. A succession of data is what is meant by a stream. The OutputStream is used to write data to a destination whereas the InputStream is used to receive data from a source. While this lesson only covers extremely fundamental features regarding streams and I/O, Java offers robust yet customizable support for I/O linked to files and networks. One by one, we would observe the most typical examples:

Streams of Bytes

8-bit byte input and output are handled by Java byte streams. While there are several classes related to byte streams, FileInputStream and FileOutputStream are the most often used classes. The example that uses these two classes to transfer an input file into an output file is shown below:

```
import java.io.*; public class CopyFile {

public static void main(String args) throws IOException

{

FileInputStream in = null; FileOutputStream out = null;

try {

in = new FileInputStream("input.txt");

out = new FileOutputStream("output.txt");

int c;

while ((c = in.read()) != -1) { out.write(c);

}

}finally {

if (in != null) {

in.close();

}

if (out != null) {

out.close();

}
```

```
}
}
}
```

Let's add the following text to the input.txt file now:

This is a test copy file.

The programme specified before should then be created and executed. Using the same content as our input.txt file, this will produce an output.txt file. Implement the following instructions in CopyFile.java using the aforementioned code:

```
$javac CopyFile.java
$java CopyFile
```

Character Stream

Java Byte streams are used for input and output of 8-bit bytes, while Java Character streams are used for input and output of 16-bit unicode. FileReader and FileWriter are the two most often used classes related to character streams, while there are other classes as well. Internally, both FileReader and FileWriter use FileInputStreams; however, the fundamental difference between the two is that FileReader reads two bytes at a time, whilst FileWriter writes two bytes at a time. With these two classes, we can adapt the aforementioned example to create an output file that replicates an input file containing Unicode characters:

```
import  java.io.*; public class CopyFile {
public static void main(String args) throws IOException
{
FileReader in = null; FileWriter out = null;

try {
in = new FileReader("input.txt"); out = new FileWriter("output.txt");

int c;
while ((c = in.read()) != -1) {out.write(c);
}
}finally {
if (in != null) {
in.close();
}
if (out != null) {out.close();
}
}
}
```

```
}
":
import java.io.*;
public class ReadConsole {
public static void main(String args) throws IOException
{
```

Let's now have the following text in the file input.txt:

This is a copy file test.

Next, construct the aforementioned software and run it. This will create an output.txt file with the same content as our input.txt file. Put the aforementioned code in CopyFile.java and carry out the following actions:

$javac CopyFile.java

$java CopyFile

Standard Streams

All programming languages provide standard I/O, which enables user programmes to accept keyboard input and display results on the computer screen. Three common devices, STDIN, STDOUT, and STDERR, must be known to programmers who use the C or C++ languages. The following three standard streams are offered by Java in a similar fashion.

1. Standard Input: This is used to provide data to the user's application and is typically represented by System.in and the keyboard.
2. Standard Output: This is used to output data generated by the user's application, and is typically a computer screen. It is represented as System.out.
3. Standard Error: This is used to output error data generated by the user's software, and it is often shown on a computer screen as System.err.

This straightforward programme generates an InputStreamReader to read the standard input stream up until the user inputs the letter "q."

```
try {
cin = new InputStreamReader(System.in); System.out.println("Enter characters, 'q' to quit.");
char c;
do {
c = (char) cin.read();System.out.print(c);
} while(c != 'q');
}finally {
if (cin != null) {cin.close();
}
}
}
```

}

Let's keep above code in ReadConsole.java file and try to compile and execute it as below. This program continues reading and outputting same character until we press 'q':

$javac ReadConsole.java

$java ReadConsole

Enter characters, 'q' to quit.1

1

eeqq

Reading and Writing Files:

As previously mentioned, a stream is a collection of data. The InputStream is used to receive data from a source, while the OutputStream is used to write data to a destination. Input and Output streams are handled by the following hierarchy of classes. The two crucial streams are FileInputStream and FileOutputStream, which will be covered in this tutorial:

FileInputStream:

Data is read from the files using this stream. The keyword new may be used to create objects, and there are many function Object() { } types that are accessible.

The following function Object() { } accepts a file name as a string to generate an input stream object to read the file:

InputStream f = new FileInputStream("C:/java/hello"); Using the File() function, we first build a file object as follows:

InputStream f = new FileInputStream(f); File f = new File("C:/java/hello");

There is a set of assistance methods that may be used after you have an InputStream object in your possession to read from the stream or do other operations on the stream. There are other important input streams available, for more detail you can refer to the followinglinks:

3 ByteArrayInputStream

4 DataInputStream

FileOutputStream: This stream is used to create files and write data into them. If a file doesn't already exist, the stream will create it before opening it for output.

These two constructors may be used to build an object that is a FileOutputStream. The function Object() { } that creates an input stream object to write the file accepts a file name as a string:

OutputStream f = new FileOutputStream("C:/java/hello")

The function Object() { } that comes after accepts a file object to build an output stream object and write the file. First, we create a file object using File() method as follows:

File f = new File("C:/java/hello"); OutputStream f = new FileOutputStream(f);

Once you have an OutputStream object in your possession, you can use a number of helper methods to write to the stream or perform other actions on it.

There are more significant output streams available; for more information, click on the following websites:

1) ByteArrayOutputStream

2) DataOutputStream

Example:

Following is the example to demonstrate InputStream and OutputStream:import java.io.*;

```
public class fileStreamTest{
public static void main(String args){try{
byte bWrite  = {11,21,3,40,5};
OutputStream os = new FileOutputStream("test.txt");for(int x=0; x < bWrite.length ; x++){
os.write( bWrite ); // writes the bytes
}
os.close();
InputStream is = new FileInputStream("test.txt");int size = is.available();
for(int i=0; i< size; i++){ System.out.print((char)is.read() + " ");
}
is.close();
}catch(IOException e){ System.out.print("Exception");
}
}
}
```

The above code would create file test.txt and would write given numbers in binary format. Samewould be output on the stdout screen.

File Navigation and I/O:

We would take many more seminars to learn the fundamentals of file navigation and I/O.

FileClass, FileReaderClass, and FileWriterClass

Java directories: A directory is a File that may include a list of additional files and directories. You may list down the files that are present in a directory and construct directories using the File object. Check out a list of every method you may call on a File object that is connected to directories for more information.

Making Directories: There are two practical File utility techniques that may be used to the creation of directories. The mkdir() function creates a directory, returning true when it succeeds and false when it fails. The mkdirs() function generates a directory as well as all of the directory's parents.

The following example generates the "/tmp/user/java/bin" directory: ipublic class CreateDir {

```
public static void main(String args) { String dirname = "/tmp/user/java/bin"; File d = new
File(dirname);
// Create directory now. d.mkdirs();
}
}
```

Create "/tmp/user/java/bin" by compiling and running the aforementioned code.

Note that, in accordance with traditions, Java automatically handles path separators on UNIX and Windows. With a Windows version of Java, the route will still resolve successfully if you use a forward slash (/).

Listing directories

To list all the files and directories present in a directory, utilise the list() function made accessible by the File object:

```
import java.io.File;

public class ReadDir {

public static void main(String args) {

File file = null;String paths;

try{

// create new file objectfile = new File("/tmp");

// array of files and directorypaths = file.list();

// for each name in the path arrayfor(String path:paths)

{

// prints filename and directory nameSystem.out.println(path);

}

}catch(Exception e){

// if any error occurse.printStackTrace();

}

}

}
```

This would produce following result based on the directories and files available in your Tmp directory:

```
test1.txt test2.txt ReadDir.java ReadDir.class
```

CHAPTER 13

JAVA DATABASE CONNECTIVITY

Kannagi Anbazhagan, Associate Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- a.kannagi@jainuniversity.ac.in

JDBC, which stands for Java Database Connectivity, is a widely used Java API for connecting a variety of databases to the Java programming language. The JDBC library has APIs for all of the typical database-related tasks:

Establishing a connection to a database, Writing SQL or MySQL commands, Running those statements in the database, Seeing and editing the records that result, JDBC is, at its core, a standard that offers a full set of interfaces that enable portable access to an underlying database. Java may be used to create several kinds of executables, including Enterprise JavaBeans; Java Applications; Java Applets; Java Servlets; Java ServerPages (JSPs); (EJBs)

All of these various executables have the ability to access databases using a JDBC driver and benefit from the data that is kept there. JDBC has the same characteristics as ODBC, enabling Java projects to incorporate code that isn't reliant on any one database.

Architecture for JDBC:

While the JDBC API supports both two-tier and three-tier database access processing architectures, the JDBC Architecture primarily consists of two layers:

1) JDBC API: This enables the connection between the application and the JDBC Manager.
2) The JDBC Manager-to-Driver Connection is supported by the JDBC Driver API.
3) The JDBC API provides seamless connection to heterogeneous databases via the use of a driver manager and database-specific drivers.
4) To access each data source, the JDBC driver manager makes sure the right driver is being utilised. The driver manager may handle many drivers running simultaneously and linked to several heterogeneous databases.

The architecture diagram that follows indicates where the driver management is located in relation to the JDBC drivers and the Java application (Figure 13.1):

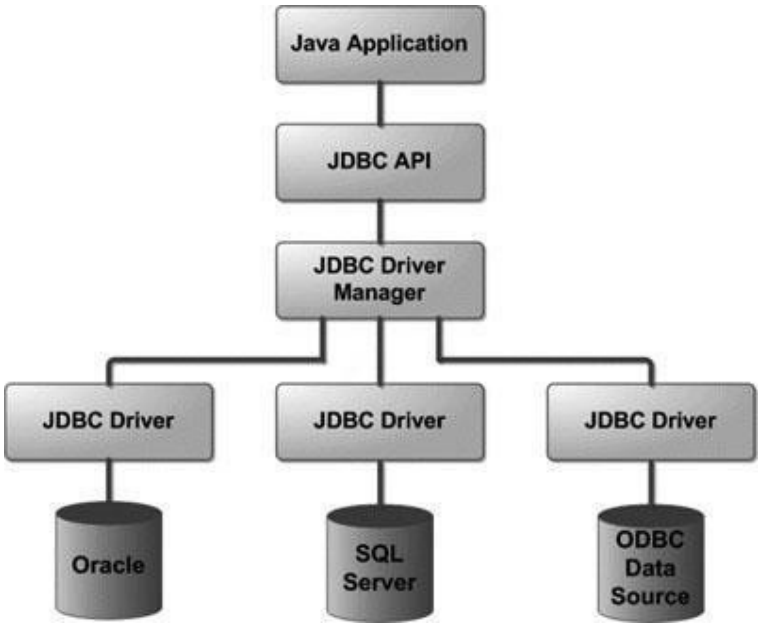


Figure 13.1: Typical JDBC Components

The aforementioned classes and interfaces are offered by the JDBC API:

DriverManager: A list of database drivers is managed by this class use communication subprotocol to match connection requests from the Java application with the appropriate database driver. To create a database Connection, the first driver to identify a certain subprotocol under JDBC will be utilised.

Driver: The database server's communications are handled by this interface. Seldom will you interact directly with a driver object. Instead, you use DriverManager objects, which control this kind of object. Moreover, it abstracts the specifics of using Driver objects.

Connection: This is the interface for reaching a database via any available method. All contact with the database is done via the connection object, which represents the communication context.

Claim: To submit SQL statements to the database, utilise objects made from this interface. Some derived interfaces don't only run stored procedures—they also take arguments.

ResultSet: After utilising Statement objects to conduct a SQL query, these objects store data that was fetched from a database. It serves as an iterator so you may cycle over the data.

SQLException: All faults that arise in a database application are handled by this class. JDBC Application Creation: Building a JDBC application involves the following six steps:

Bring the packages in. specifies that you must include the packages containing the JDBC classes required for database programming. The most frequent solution is to use import java.sql.*. Install the JDBC driver requires you to setup a driver before you can establish a channel of communication with the database. Establish a relationship uses the DriverManager and is necessary. To build a Connection object, which symbolises a real-world connection to the database, use the getConnection() function.

Carry out an inquiry: requires creating and submitting a SQL statement to the database using an object of type Statement.

Take data out of the result set. demands that you use the suitable ResultSet. To access the data from the result set, use the getXXX() function. Make the environment cleaner. requires terminating every database resource manually as opposed to depending on trash collection from the JVM.

JDBC Application Creation:

Creating a JDBC application involves six stages, which I'll outline in this tutorial:

Bring the packages in:

The packages containing the JDBC classes required for database programming must be included for this to work. The following example shows how to use import java.sql.* most often:

```
//STEP 1. Import required packages
import java.sql.*;
```

Register the JDBC driver:

This requires that you initialize a driver so you can open a communications channel withthe database. Following is the code snippet to achieve this:

```
//STEP 2: Register JDBC driver
Class.forName("com.mysql.jdbc.Driver");
```

Open a connection:

In order to do this, a Connection object must be created using the DriverManager.getConnection() function. This object represents a physical connection to the database and is created as follows:

```
//STEP 3: Open a connection
// Database credentials
static final String USER = "username";
static final String PASS = "password";
System.out.println("Connecting to database...");
conn = DriverManager.getConnection(DB_URL,USER,PASS);
```

Execute a query:

This requires using an object of type Statement or PreparedStatement for building and submitting an SQL statement to the database as follows:

```
//STEP 4: Execute a query
System.out.println("Creating statement...");
stmt = conn.createStatement();
String sql;
    sql = "SELECT id, first, last, age FROM Employees";
    ResultSet rs = stmt.executeQuery(sql);
```

If there is an SQL UPDATE,INSERT or DELETE statement required, then followingcode snippet would be required:

```
//STEP 4: Execute a query
System.out.println("Creating statement...");
stmt = conn.createStatement();
String sql;
sql = "DELETE FROM Employees"; ResultSet rs = stmt.executeUpdate(sql);
```

Extract data from result set:

This step is required in case you are fetching data from the database. You can use the appropriate ResultSet.getXXX() method to retrieve the data from the result set as follows:

```
//STEP 5: Extract data from result set
while(rs.next()){
   //Retrieve by column name
   int id = rs.getInt("id");
   int age = rs.getInt("age");
   String first = rs.getString("first");
   String last = rs.getString("last");

   //Display values
   System.out.print("ID: " + id);
   System.out.print(", Age: " + age);
   System.out.print(", First: " + first);
   System.out.println(", Last: " + last);
}
```

Clean up the environment:

You should explicitly close all database resources versus relying on the JVM's garbage collection as follows:

```
//STEP 6: Clean-up
environment
rs.close();
stmt.close();
conn.close();
```

First JDBC Program:

Based on the above steps, we can have following consolidated sample code which we canuse as a template while writing our JDBC code:

This sample code has been written based on the environment and database setup done in Environment chapter.

```
//Handle errors for JDBC
se.printStackTrace();
}catch(Exception e){
//Handle errors for Class.forName
e.printStackTrace();
   }finally{
//finally block used to close resources
try{
if(stmt!=null)
stmt.close();
}catch(SQLException se2){
}// nothing we can do
try{
if(conn!=null)
conn.close();
}catch(SQLException se){
se.printStackTrace();
}//end finally try
}//end try
System.out.println("Goodbye!");
}//end main
}//end FirstExample
```

Now let us compile above example as follows:

```
C:\>javac FirstExample.java
C:\>
```

When you run **FirstExample**, it produces following result:

```
C:\>java FirstExample
Connecting to database...
Creating statement...
ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
C:\>
```

```
//Handle errors for JDBC
se.printStackTrace();
}catch(Exception e){
//Handle errors for Class.forName
e.printStackTrace();
   }finally{
//finally block used to close resources
try{
if(stmt!=null)
stmt.close();
}catch(SQLException se2){
}// nothing we can do
try{
if(conn!=null)
conn.close();
}catch(SQLException se){
se.printStackTrace();
}//end finally try
}//end try
System.out.println("Goodbye!");
}//end main
}//end FirstExample
```

Now let us compile above example as follows:

```
C:\>javac FirstExample.java
C:\>
```

When you run

FirstExample, it produces following result:

```
C:\>java
FirstExample
Connecting to database...
Creating statement...
ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
C:\>
```

SQLException Methods:

A SQLException can occur both in the driver and the database. When such an exception occurs, an object of type SQLException will be passed to the catch clause (Table 13.1).

Table 13.1: The passed SQLException object has the following methods available for retrievingadditional information about the exception:

Method	Description
getErrorCode( )	Gets the error number associated with the exception.
getMessage( )	Gets the JDBC driver's error message for an error handled by the driver or gets the Oracle error number and message for a database error.
getSQLState( )	Gets the XOPEN SQLstate string. For a JDBC driver error, no useful information is returned rom this method. For a database error, the five-digit XOPEN SQLstate code is returned, This method can return null.
getNextException( )	Gets the next Exception object in the exception chain.
printStackTrace( )	Prints the current exception, or throwable, and itsbacktrace to a standard error stream.
printStackTrace(PrintStream s)	Prints this throwable and its backtrace to the printstream you specify.
printStackTrace(PrintWriter w)	Prints this throwable and its backtrace to the printwriter you specify.

By utilizing the information available from the Exception object, you can catch an exception and continue your program appropriately. Here is the general form of a try block:

{
ır risky code goes between these curly braces!!!

:h(Exception ex) {
ır exception handling code goes between these
y braces, similar to the exception clause
PL/SQL block.

.lly {
ır must-always-be-executed code goes between these
y braces. Like closing database connection.

JDBC - Data Types:

The following table 13.2 summarizes the default JDBC data type that the Java data type is converted to when you call the setXXX() method of the PreparedStatement or CallableStatement object or the ResultSet.updateXXX() method.

Table 13.2: default JDBC data type

SQL	**JDBC/Java**	**setXXX**	**updateXXX**
VARCHAR	java.lang.String	setString	updateString
CHAR	java.lang.String	setString	updateString
LONGVARCHAR	java.lang.String	setString	updateString
BIT	Boolean	setBoolean	updateBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	updateBigDecimal
TINYINT	Byte	setByte	updateByte
SMALLINT	Short	setShort	updateShort
INTEGER	Int	setInt	updateInt
BIGINT	Long	setLong	updateLong
REAL	Float	setFloat	updateFloat
FLOAT	Float	setFloat	updateFloat
DOUBLE	Double	setDouble	updateDouble
VARBINARY	byte	setBytes	updateBytes

BINARY	byte	setBytes	updateBytes
DATE	java.sql.Date	setDate	updateDate
TIME	java.sql.Time	setTime	updateTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	updateTimestamp
CLOB	java.sql.Clob	setClob	updateClob
BLOB	java.sql.Blob	setBlob	updateBlob
ARRAY	java.sql.Array	setARRAY	updateARRAY
REF	java.sql.Ref	SetRef	updateRef
STRUCT	java.sql.Struct	SetStruct	updateStruct

JDBC 3.0 has enhanced support for BLOB, CLOB, ARRAY, and REF data types. The ResultSet object now has updateBLOB(), updateCLOB(), updateArray(), and updateRef() methods that enable you to directly manipulate the respective data on the server. The setXXX() and updateXXX() methods enable you to convert specific Java types to specific JDBC data types. The methods, setObject() and updateObject(), enable you to map almost any Java type to a JDBC data type. ResultSet object provides corresponding getXXX() method for each data type to retrieve column value. Each method can be used with column name or by its ordinal position (Table 13.3).

Table 13.3: Illustrate the JDBC data type

SQL	**JDBC/Java**	**setXXX**	**getXXX**
VARCHAR	java.lang.String	setString	getString
CHAR	java.lang.String	setString	getString
LONGVARCHAR	java.lang.String	setString	getString
BIT	Boolean	setBoolean	getBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	getBigDecimal
TINYINT	Byte	setByte	getByte
SMALLINT	Short	setShort	getShort
INTEGER	Int	setInt	getInt
BIGINT	Long	setLong	getLong
REAL	Float	setFloat	getFloat

FLOAT	Float	setFloat	getFloat
DOUBLE	Double	setDouble	getDouble
VARBINARY	byte	setBytes	getBytes
BINARY	byte	setBytes	getBytes
DATE	java.sql.Date	setDate	getDate
TIME	java.sql.Time	setTime	getTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	getTimestamp
CLOB	java.sql.Clob	setClob	getClob
BLOB	java.sql.Blob	setBlob	getBlob
ARRAY	java.sql.Array	setARRAY	getARRAY
REF	java.sql.Ref	SetRef	getRef
STRUCT	java.sql.Struct	SetStruct	getStruct

Sample Code:

You may use this sample example as a model to construct your own JDBC application in the future.

Based on the environment and database configuration completed in the preceding chapter, this example code was created.

In FirstExample.java, paste the following example, build it, and then run it as follows:

```
//STEP 1. Import required packages
import java.sql.*;

public class FirstExample {
  // JDBC driver name and database URL
  static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
  static final String DB_URL = "jdbc:mysql://localhost/EMP";

  // Database credentials
  static final String USER = "username";
  static final String PASS = "password";

  public static void main(String[] args) {
  Connection conn = null;
  Statement stmt = null;
  try{
    //STEP 2: Register JDBC driver
    Class.forName("com.mysql.jdbc.Driver");
    //STEP 3: Open a connection
    System.out.println("Connecting to database...");
    conn = DriverManager.getConnection(DB_URL,USER,PASS);
```

```
conn = DriverManager.getConnection(DB_URL,USER,PASS);
//STEP 4: Execute a query
System.out.println("Creating statement...");
stmt = conn.createStatement();
String sql;
sql = "SELECT id, first, last, age FROM Employees";
ResultSet rs = stmt.executeQuery(sql);



//STEP 5: Extract data from result set
while(rs.next()){
  //Retrieve by column name
  int id = rs.getInt("id");
  int age = rs.getInt("age");
  String first = rs.getString("first");
  String last = rs.getString("last");

  //Display values
  System.out.print("ID: " + id);
  System.out.print(", Age: " + age);
  System.out.print(", First: " + first);
  System.out.println(", Last: " + last);
}
//STEP 6: Clean-up environment
rs.close();
stmt.close();
conn.close();
}catch(SQLException se){
  //Handle errors for JDBC
  se.printStackTrace();
}catch(Exception e){
  //Handle errors for Class.forName
  e.printStackTrace();
}finally{
  //finally block used to close resources
  try{
    if(stmt!=null)
      stmt.close();
  }catch(SQLException se2){
  }// nothing we can do
  try{
    if(conn!=null)
      conn.close();
  }catch(SQLException se){
    se.printStackTrace();
  }//end finally try
}//end try
System.out.println("Goodbye!");
}//end main
}//end FirstExample
```

the preceding example together now as follows:

```
C:\>javac FirstExample.java
C:\>
```

When you run **FirstExample**, it produces following result:

```
C:\>java FirstExample
Connecting to database...
Creating statement...
ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
C:\>
```

Types of JDBC Drivers:

Since Java runs on such a broad range of hardware platforms and operating systems, JDBC driver implementations differ. Types 1, 2, 3, and 4 are the four categories into which Sun has split the implementation types.

Type 1: JDBC-ODBC Bridge Driver:

An ODBC driver installed on each client system is accessed via a JDBC bridge in a Type 1 driver. The destination database must be configured on your system as a Data Source Name (DSN) before using ODBC. This sort of driver was helpful when Java initially launched since the majority of databases only allowed ODBC access, but it is currently only advised for experimental usage or in cases when there are no other options.

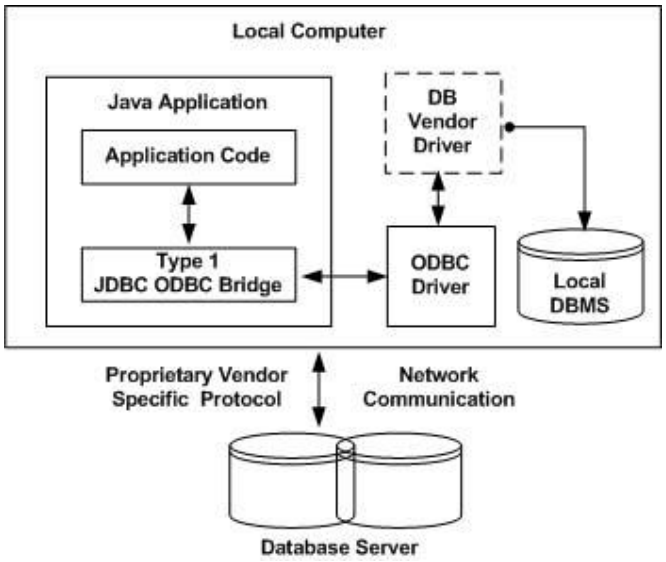


Figure 13.1: JDBC-ODBC Bridge Driver:

A excellent example of this kind of driver is the JDBC-ODBC bridge that is included with JDK 1.2.

Type 2: JDBC-Native API:

JDBC API requests are transformed into native C/C++ API requests specific to the database in a Type 2 driver. The vendor-specific driver must be installed on each client system when using these drivers, which are normally given by the database manufacturers and utilised similarly to how the JDBC-ODBC Bridge is used.

Since the native API is database-specific and must be changed whenever the database is changed, even if Type 2 drivers may speed up your computer because they do away with ODBC's overhead (Figure 13.2).

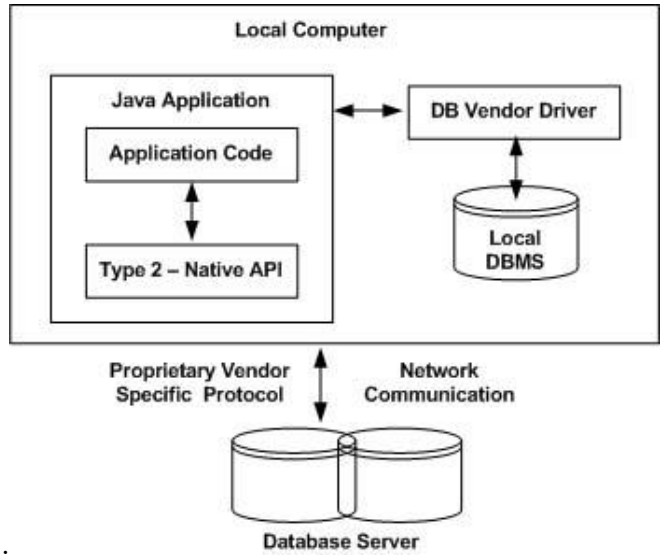


Figure 13.2: JDBC-Native API:

An example of a Type 2 driver is the Oracle Call Interface (OCI) driver.

Type 3: JDBC-Net pure Java:

A three-tier technique is utilised in a Type 3 driver to access databases. Standard network sockets are used by JDBC clients to connect to a middleware application server. The middleware application server then converts the socket information into the call format needed by the DBMS and forwards it to the database server.

This kind of driver is quite versatile since it doesn't need clients to install any code and because a single driver may really provide access to several databases.

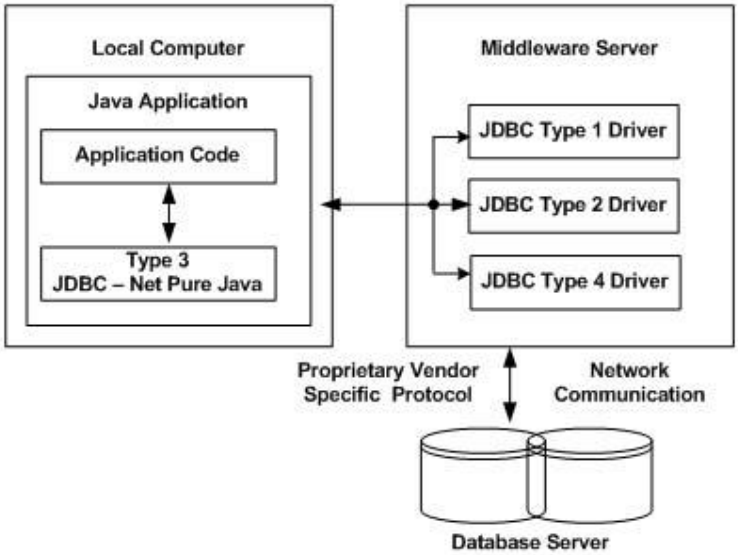


Figure 13.3JDBC-Net pure Java:

The application server functions as a JDBC "proxy," making calls on behalf of the client application. As a consequence, to utilise this driver type efficiently, you need to have some familiarity with the setup of the application server (Figure 13.3).

Understanding the subtleties will be useful since your application server may employ a Type 1, 2, or 4 driver to interface with the database.

Type 4: Completely pure Java

A Type 4 driver is a wholly Java-based driver that uses a socket connection to interact directly with the vendor's database. The vendor often offers this driver, which is the fastest database driver currently available.,This kind of driver is quite adaptable, and neither the client nor the server need be equipped with any additional software. Moreover, these drivers may be dynamically downloaded (Figure 13.4).

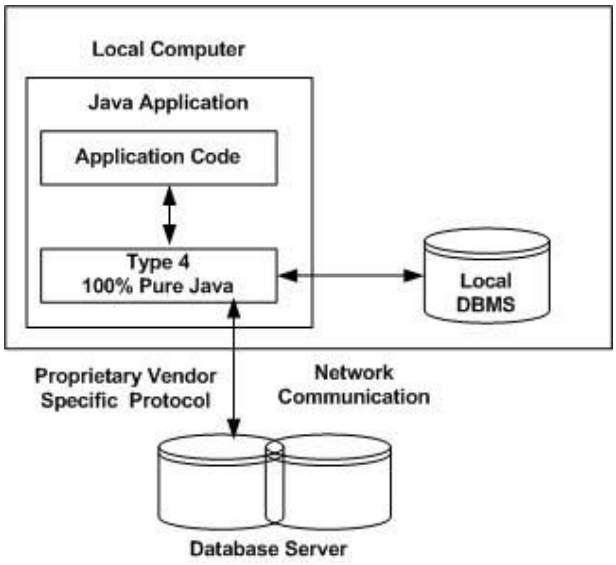


Figure 13.4: Completely pure Java

MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

Driver should be used:

1. If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred drivertype is 4.
2. If your Java application is accessing multiple types of databases at the same time, type 3 is thepreferred driver.
3. Type 2 drivers are useful in situations where a type 3 or type 4 driver is not available yet for yourdatabase.
4. The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

CHAPTER 14

INTRODUCTION OF SWING

Adlin Jebakumari, Assistant Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- j.adlin@jainuniversity.ac.in

javax.swing and its associated subpackages, such javax.swing.tree, contain the Swing-related classes. There are several other Swing-related classes and interfaces.The next sections of this analyse different Swing components and provide examples using applets.

JApplet

The JApplet class, which extends Applet, is essential to Swing. Swing-using applets must be JApplet subclasses. JApplet offers a wealth of features not present in Applet. JApplet, for instance, provides a variety of "panes," including the content pane, the glass pane, and the root pane. The majority of JApplet's expanded capabilities won't be used in the example. Nevertheless, since it is used by the example applets, one distinction between Applet and JApplet is crucial to this study. Do not use the applet's add() function when adding a component to an instance of JApplet. Instead, make a call to add() for the JApplet object's content pane. The following procedure may be used to get the content pane:

getContentPane in a container ( )

A component may be added to a content pane using the add() function of the Container class. This is how it looks: void add (comp)

1. The component to be added to the content pane in this case is called comp. Labeling and Symbols
2. The ImageIcon class in Swing is responsible for painting icons from images. Here, two of its builders are shown.
3. ImageIcon(String filename) (String filename) ImageIcon(URL url) (URL url)
4. The first method makes use of the picture found in the file filename. The second version makes use of the picture from the url-referenced page.

The ImageIcon class carries out the functions listed here by implementing the Icon interface:

Method Description

Method Description
int getIconHeight( ) Returns the height of the icon in pixels.
int getIconWidth( ) Returns the width of the icon in pixels.
void paintIcon(Component *comp*, Graphics *g*, int *x*, int *y*)

Paints the icon at position *x*, *y* on the graphics context *g*. Additional information about the paint operation can be provided in *comp*.

Swing labels are instances of the JLabel class, which extends JComponent. It can display text

and/or an icon. Some of its constructors are shown here:

JLabel(Icon *i*)
Label(String *s*)
JLabel(String *s*, Icon *i*, int *align*)

The label's text and symbol in this instance are s and i. Either LEFT, RIGHT, CENTER, LEADING, or TRAILING are options for the align argument. Together with numerous other constants required by the Swing classes, these constants are specified in the SwingConstants interface.

The following techniques may be used to read and write the label's icon and accompanying text:

Icon getIcon( )
String getText( )
void setIcon(Icon *i*)
void setText(String *s*)

The text and symbol in this case are I and s, respectively.

The next example shows how to create and display a label that has both an icon and a string on it. The applet gets its content pane first. The France.gif file's ImageIcon object is then produced. This serves as the JLabel constructor's second parameter. The label text and the alignment are the first and final parameters for the JLabel function Object() { }. The label is then included in the content pane.

Text Areas

The JTextComponent class, which extends JComponent, contains the Swing text field. It offers features that are typical of Swing text components. JTextField, one of its subclasses, enables one-line text editing. These are a few of its constructors:

JTextField( )
JTextField(int *cols*)
JTextField(String *s*, int *cols*)
JTextField(String *s*)

The string to be shown in this case is s, and the number of columns in the text field is cols. The creation of a text field is shown in the example below. A flow layout is then allocated as the applet's layout manager when the content pane is obtained. Then the content pane receives a JTextField object that has been constructed.

Buttons

Swing buttons provide functionalities that are absent from the AWT's Button class definition. For instance, you may link an icon to a Swing button. Subclasses of the AbstractButton class, which extends JComponent, include swing buttons. There are several methods in AbstractButton that let you modify the behaviour of buttons, check boxes, and radio buttons. For instance, you may choose which icons are shown for the component when it is chosen, pushed, or deactivated. When the mouse is placed over a component, a rollover icon made from another icon is shown.

The techniques used to stop this behaviour are as follows:

void setDisabledIcon(Icon *di*)
void setPressedIcon(Icon *pi*)
void setSelectedIcon(Icon *si*)
void setRolloverIcon(Icon *ri*)

Here, the appropriate icons for these various situations are di, pi, si, and ri.

The following techniques may be used to read and write the text that is connected with a button:

```
String getText( )
void setText(String s)
```

In this case, s is the text that will go with the button.

When pushed, concrete subclasses of AbstractButton provide action events. For these events, listeners register and unregister using the following procedures:

void addActionListener(ActionListener al) void removeActionListener(ActionListener al) Here, al is the action listener.

Push buttons, check boxes, and radio buttons all belong to the superclass known as AbstractButton. Next, each is investigated.

The class of JButtons

Push button functionality is provided by the JButton class. Push buttons may be linked to an icon, a string, or both using JButton. These are a few of its constructors:

```
JButton(Icon i)
JButton(String s)
JButton(String s, Icon i)
```

The string and icon for the button in this case are s and i.

A Checkbox

A concrete implementation of AbstractButton is the JCheckBox class, which offers check box functionality. JToggleButton, which supports buttons with two states, is its immediate superclass. These are a few of its constructors:

```
JCheckBox(Icon i)
JCheckBox(Icon i, boolean state)
JCheckBox(String s)
JCheckBox(String s, boolean state)
JCheckBox(String s, Icon i)
JCheckBox(String s, Icon i, boolean state)
```

The icon for the button in this case is i. S. specifies the text. The check box is originally checked if state is true. If not, it is not. The following procedure may be used to modify the check box's current state: setSelected: void (boolean state). In this case, state is true if the checkbox has to be selected. A text field and four checkboxes are shown in an applet created in the manner shown in the example below. The text of a check box appears in the text field when it is selected. A flow layout is chosen to serve as the JApplet object's content pane's layout manager. Next, icons are allocated for the normal, rollover, and selected states and four check boxes are placed to the content pane. After that, the applet is set up to accept item events. The content pane is then given a text field.

An item event is produced when a check box is chosen or deselected. ItemStateChanged handles this ( ). The JCheckBox object that caused the event is obtained using the getItem() function inside itemStateChanged(). The text for that check box is obtained via the getText() function, which is then used to put the text in the text field.

Radio Buttons

The JRadioButton class, a real-world implementation of AbstractButton, supports radio

buttons. JToggleButton, which supports buttons with two states, is its immediate superclass. These are a few of its constructors:

JRadioButton(Icon *i*)
JRadioButton(Icon *i*, boolean *state*)
JRadioButton(String *s*)
JRadioButton(String *s*, boolean *state*)
JRadioButton(String *s*, Icon *i*)
JRadioButton(String *s*, Icon *i*, boolean *state*)

The icon for the button in this case is i. S. specifies the text. The button is first chosen if state is true. If not, it is not. A group of radio buttons must be set up. Each time, just one button from that group may be chosen. Each button in a group that was previously selected is immediately deselected. A button group is created by instantiating the ButtonGroup class. For this, its default function Object() { } is used. The following approach is then used to add elements to the button group:

Bundle Boxes

The JComboBox class, which extends JComponent, is how Swing offers a combo box—a drop-down list and text field combined. Typically, a combination box just shows one selection. It may, however, also provide a drop-down menu from which a user can choose an alternative item. Instead, you may input your choice in the text box. Here are two builders for:

JComboBox( )
JComboBox(Vector *v*)

The vector v in this case initialises the combo box. The addItem() function, whose signature is seen below, adds items to the list of options:

empty addItem (Object obj). The item that will be added to the combo box in this case is indicated by the placeholder obj.

Tabbed windows

A component known as a tabbed pane resembles a row of file folders in a filing cabinet. Each folder is given a name. The contents of a folder become visible when the user chooses it. One of the folders at a time may only be chosen. Tabbed windows are often used to configure settings. The JTabbedPane class, which extends JComponent, is responsible for encapsulating tabbed panes. We'll make advantage of its built-in function Object() { }. The following procedure is used to define tabs:

void addTab(String *str*, Component *comp*)

The tab's title in this case is str, and the component that has to be put to the tab is comp. A JPanel or a subclass of it is often added. Here is a basic explanation of how to utilise a tabbed pane in an applet:

1. Make a JTabbedPane object first.
2. To add a tab to the window, use addTab(). The title of the tab and the component it contains are defined by the inputs to this function.
3. For each tab, repeat step 2 above.
4. Include the tabbed pane in the applet's content window.

Scroll windows

A component called a scroll pane displays a rectangular area where a component may be seen. If required, horizontal and/or vertical scroll bars may be offered. Swing uses the JScrollPane

class, which extends JComponent, to create scroll panes. These are a few of its constructors:

JScrollPane(Component *comp*)
JScrollPane(int *vsb*, int *hsb*)
JScrollPane(Component *comp*, int *vsb*, int *hsb*)

The component to be added to the scroll pane in this case is called comp. When the vertical and horizontal scroll bars for this scroll pane are shown, they are determined by the intconstants vsb and hsb. The ScrollPaneConstants interface specifies these constants. The following list of examples of these constants is provided:

Repeated Description

HORIZONTAL_SCROLLBAR_ALWAYS Always provide horizontal scroll bar
HORIZONTAL_SCROLLBAR_AS_NEEDED Provide horizontal scroll bar, if needed
VERTICAL_SCROLLBAR_ALWAYS Always provide vertical scroll bar
VERTICAL_SCROLLBAR_AS_NEEDED Provide vertical scroll bar, if needed

The procedures to utilise a scroll pane in an applet are as follows:

1.Create a JComponent object.

2. Construct an object named JScrollPane. The component and the guidelines for the vertical and horizontal scroll bars are specified in the constructor's parameters.

3. Include the scroll pane in the applet's content pane.

Trees

A tree is a component that displays a data hierarchy. With this presentation, each subtree may be expanded or collapsed by the user. Swing uses the JTree class, which extends JComponent, to implement trees. These are a few of its constructors:

JTree(Hashtable *ht*)
JTree(Object *obj*[ ])
JTree(TreeNode *tn*)
JTree(Vector *v*)

The first form builds a tree with a child node for each entry of the hash table ht.

With the second version, each member of the array obj is a child node. The third version of the tree's root is represented by the tree node tn. The final version employs the components of vector v as child nodes.

Events are produced by a JTree object whenever a node is enlarged or collapsed. Listeners may sign up for and unregister from receiving these messages using the addTreeExpansionListener() and removeTreeExpansionListener() methods. These methods' signatures are shown here:

void addTreeExpansionListener(TreeExpansionListener *tel*)
void removeTreeExpansionListener(TreeExpansionListener *tel)*

The listener object in this case is tel.

To convert a mouse click on a particular spot in the tree to a tree path, use the getPathForLocation() function. Below is a sample of its signature:

TreePath getPathForLocation(int *x*, int *y*)

The mouse is clicked at these coordinates, which are x and y. The returned result is a TreePath object that contains details about the user-selected tree node.

Tables

A table is an element that shows data in rows and columns. To resize columns, move the pointer along the edges of the columns. A column may also be moved by dragging it. The JTable class, which extends JComponent, implements tables.

JTable(Object *data*[ ][ ], Object *colHeads*[ ])

The information to be shown is represented by a two-dimensional array called data, and the column headings are represented by a one-dimensional array called colHeads. The procedures for utilising a table in an applet are as follows:

1. Make a JTable object first.

2. Construct an object named JScrollPane. The table and the guidelines for the vertical and horizontal scroll bars are specified in the constructor's parameters.

3. Provide a scroll pane and a table.

4. Include the scroll pane in the applet's content pane.

Events

An event is an object that specifies a state change in a source according to the delegation model. It may be produced as a result of a user interacting with graphical user interface components. Pressing a button, typing a character using the keyboard, choosing something from a list, and using the mouse are some actions that result in the generation of events. There are a lot of additional user activities that might be used as examples. Events may also take place that are not directly related to user interface interactions. An event could be produced, for instance, when a timer expires, a counter reaches a certain value, a hardware or software failure takes place, or an action is finished. It is up to you to define events that fit your application.

Event Sources

A source is an item that causes an event to occur. When the object's internal state changes in whatever manner, something happens. More than one sort of event may be produced by sources. For listeners to get alerts about a certain kind of event, a source must register them. Each kind of event has a unique registration process.

The basic format is as follows:

public void add*Type*Listener(*Type*Listener *el*)

Type is the event's name in this case, and el stands for event listener. For instance, addKeyListener is the method used to register a keyboard event listener ( ). AddMouseMotionListener is the name of the method that registers a mouse motion listener ( ). Each registered listener is informed of an event's occurrence and given a copy of the event object. The event is being disseminated in this situation. Only listeners who sign up to get alerts are ever given them. Some providers may only let one listener to sign up. Such a procedure often takes the following form:

public void add*Type*Listener(*Type*Listener *el*)
throws java.util.TooManyListenersException

Type is the event's name in this case, and el stands for event listener. An alert is sent to the registered listener when such an occurrence takes place. This is referred to as a unicasting occurrence. A source must also provide a way for a listener to withdraw interest in a particular kind of event. The general form of such a method is this:

public void remove*Type*Listener(*Type*Listener *el*)

Type is the event's name in this case, and el stands for event listener. For instance, you might use removeKeyListener to remove a keyboard listener ( ). The source that produces events offers the tools to add or delete listeners. The Component class, for instance, has methods for adding and removing keyboard and mouse event listeners

Event Classes

The foundation of Java's event handling system is made up of classes that represent events. As a result, we take a tour of the event classes to kick off our study of event handling. You'll find that they provide a reliable, simple way of encapsulating events. EventObject, which is in the java.util package, is at the top of the hierarchy of Java event classes. The superclass for all events is this one.

EventObject(Object *src*)

The object that creates this event in this case is src.

GetSource() and function toString() { } are two of the methods found in EventObject ( ). The source of the event is returned by the getSource() function. Here is a general example of it:

Object getSource( )

The desired result of function toString() { }() is the event's string equivalent.

A subclass of EventObject is the class AWTEvent, which is specified in the java.awt package. It is the superclass of all AWT-based events utilised by the delegation event model (directly or indirectly). The kind of the event may be ascertained using its getID() function. This method's signature is shown here:

int getID( )

provides further information regarding AWTEvent. At this point, the only thing that need to know is that AWTEvent is the superclass of all the other classes covered in this section.

1. The superclass of all events is called EventObject.
2. All AWT events that are handled by the delegation event model belong to the superclass AWTEvent.

Many different sorts of events that are produced by various user interface components are defined in the java.awt.event package. The most significant of these event types are included in Table 14.1, along with a short explanation of when they are created. The following sections include descriptions of each class's most popular constructors and functions.

Event Attendants

An object that receives notification of an event is called a listener. There are two key prerequisites. It must must have registered with one or more sources in order to get alerts about a certain category of occurrences. Second, it must put in place procedures for obtaining and handling these notifications.

A group of interfaces included in java.awt.event describe the methods used to receive and handle events. For instance, the MouseMotionListener interface specifies two ways to get alerts when the mouse is moved or dragged. If an object implements this interface, it may receive and handle any or both of these events.

The Model for Delegation Events

The delegation event model, which outlines standardised and consistent processes to produce and process events, is the foundation of the contemporary approach to managing events. A source creates an event and delivers it to one or more listeners; this is the principle behind it.

The listener in this design just waits till it gets an event. The listener processes the event once it is received before returning. The benefit of this approach is that the user interface logic that creates events and the application logic that handles events are clearly separated. The processing of an event may be "delegated" by a user interface element to another piece of code.

Listeners must sign up with a source in order to get event notifications under the delegation event model. One significant advantage of this is that only listeners who desire to receive alerts are notified. Compared to the traditional Java 1.0 approach's architecture, this is a more effective method of handling events. Prior to this change, an event was sent through the containment hierarchy until it reached a component. This wasted time by requiring components to receive events that they couldn't handle. This overhead is removed by the delegation event mechanism.

The class MouseEvent

Mouse events come in eight different varieties. The following integer constants are defined by the MouseEvent class and may be used to reference them.:

MOUSE_CLICKED The user clicked the mouse.
MOUSE_DRAGGED The user dragged the mouse.
MOUSE_ENTERED The mouse entered a component.
MOUSE_EXITED The mouse exited from a component.
MOUSE_MOVED The mouse moved.
MOUSE_PRESSED The mouse was pressed.
MOUSE_RELEASED The mouse was released.
MOUSE_WHEEL The mouse wheel was moved (Java 2, v1.4).

MouseEvent is a subclass of **InputEvent**. Here is one of its constructors. MouseEvent(Component src, int type, long when, int modifiers,

int *x*, int *y*, int *clicks*, boolean *triggersPopup*)

Src is a reference to the event's originating component in this case. The event's kind is determined by type. When is passed in the system time at which the mouse event occurs. Whatever modifiers were pushed when a mouse event happened are shown via the modifiers argument. The mouse's coordinates are sent in x and y. Clicks are used to pass the click count. The catalysts If a pop-up menu appears as a result of this occurrence on this platform, the popup flag lets you know. With the addition of a second function Object() { } in Java 2, version 1.4, it is now possible to specify the button that triggered the event. The getX() and getY methods of this class are the most frequently utilised ones ( ). They provide back the mouse's X and Y coordinates at the time the event happened. These are their forms:

int getX( )
int getY( )

Alternatively, you can use the getPoint( ) method to obtain the coordinates of the mouse.It is shown here:

Point getPoint( )

It returns a Point object that contains the X, Y coordinates in its integer members: x and y.The translatePoint( ) method changes the location of the event. Its form is shown here:

void translatePoint(int *x*, int *y*)

Here, the arguments *x* and *y* are added to the coordinates of the event.

The **getClickCount( )** method obtains the number of mouse clicks for this event.Its signature is shown here:

int getClickCount( )

The isPopupTrigger( ) method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:

boolean isPopupTrigger( )

Java 2, version 1.4 added the getButton( ) method, shown here.

int getButton( )

It returns a value that represents the button that caused the event. The return value will be one ofthese constants defined by MouseEvent.

NOBUTTON BUTTON1 BUTTON2 BUTTON3

The NOBUTTON value indicates that no button was pressed or released

Events Handled by the Keyboard

The basic architecture used to handle mouse events in the section above may also be used to handle keyboard events. Naturally, there will be a change as you will be using the KeyListener interface.

It is helpful to examine how important events are produced before looking at an example. A KEY PRESSED event is produced when a key is pushed. This causes the keyPressed() event handler to be called. A KEY RELEASED event is produced and the keyReleased() handler is called when the key is relinquished. A KEY TYPED event is received and the keyTyped() handler is called if the keystroke produces a character. As a result, at least two and often three events are produced whenever the user pushes a key. You may disregard the information shared at the major publicity and release events if all that matters to you are the actual characters. The keyPressed() handler must be used by your software to keep an eye out for special keys, such as the arrow or function keys, if it needs to handle them.

Before processing keyboard events, your software must also fulfil another prerequisite: it must ask for input attention. Use requestFocus(), a function defined by Component, to do this. Your software won't get any keyboard events if you don't.

```
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet
implements KeyListener {
String msg = "";
int X = 10, Y = 20; // output coordinates
public void init() {
addKeyListener(this);
requestFocus(); // request input focus
}
```

```
public void keyPressed(KeyEvent ke) {
showStatus("Key Down");
}
public void keyReleased(KeyEvent ke) {
showStatus("Key Up");
}
public void keyTyped(KeyEvent ke) {
msg += ke.getKeyChar();
```

Here is an example of output:

You must react to the keyPressed() handler in order to handle the special keys, such as the arrow or function keys. These are not accessible through keyTyped ( ). You utilise the keys' virtual key codes to recognise them. As an example, the following applet displays the names of a handful of the special keys.

Adapters Class

An adapter class, a unique feature offered by Java, may sometimes make it easier to create event handlers. An event listener interface's methods are all provided by an adapter class as an empty implementation. When you wish to receive and process just some of the events that are handled by a certain event listener interface, adapter classes might be helpful. By extending one of the adapter classes and implementing only the events you are interested in, you can create a new class that will serve as an event listener.

For instance, the mouseDragged() and mouseMoved functions of the MouseMotionAdapter class ( ). These empty methods' signatures are identical to those found in the MouseMotionListener interface. You could easily expand MouseMotionAdapter and implement mouseDragged if you were just interested in mouse drag events ( ). You may take care of the mouse motion events by using the empty implementation of mouseMoved(). In Table 20-4, the most popular adapter classes in the java.awt.event package are listed along with the interfaces they implement.

An adapter is shown in the example below. When the mouse is clicked or moved, a message appears in the status bar of an applet viewer or browser. All other mouse activities, however, go unnoticed. Three courses make up the curriculum. Extending Applet is AdapterDemo. MyMouseAdapter is created and registered to receive notifications of mouse events via the init() function of the object. Moreover, it creates a MyMouseMotionAdapter instance and registers that object to get alerts when mouse motion events occur. Both constructors accept the applet's reference as an argument. The mouseClicked() function is implemented by MyMouseAdapter. Code inherited from the MouseAdapter class discreetly ignores all other mouse events.

The mouseDragged() function is implemented by MyMouseMotionAdapter. Code borrowed from the MouseMotionAdapter class discreetly disregards the other mouse motion event.

Adapter Class	**Listener Interface**
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	
MouseMotionListener	
WindowAdapter	WindowListener

```
Demonstrate an adapter.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet {
public void init() {
addMouseListener(new MyMouseAdapter(this));
addMouseMotionListener(new MyMouseMotionAdapter(this));
}
}
class MyMouseAdapter extends MouseAdapter {
AdapterDemo adapterDemo;
public MyMouseAdapter(AdapterDemo adapterDemo) {

this.adapterDemo = adapterDemo;


}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
adapterDemo.showStatus("Mouse clicked");
}
}
class MyMouseMotionAdapter extends MouseMotionAdapter {
AdapterDemo adapterDemo;
public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
this.adapterDemo = adapterDemo;
}

// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
adapterDemo.showStatus("Mouse dragged");
}
}
```

Inner Classes

Consider the applet in the below listing to grasp the advantage offered by inner classes. There is no inner class employed. When the mouse is pushed, it aims to show the string "Mouse Pressed" in the status bar of the applet viewer or browser. In this curriculum, there are two upper-level courses. Applet is extended by MousePressedDemo, and MouseAdapter is extended by MyMouseAdapter. MyMouseAdapter is created by MousePressedDemo's init() function, which also passes this object to the addMouseListener() method as an argument.

You'll see that the MyMouseAdapter function Object() { } accepts a reference to the applet as an input. In order for the mousePressed() function to utilise this reference later, it is saved in an instance variable. The saved applet reference is used to call the showStatus() function of the applet when the mouse button is pressed. To put it another way, MyMouseAdapter stores the applet reference when showStatus() is called..

```
// This applet does NOT use an inner class.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="MousePressedDemo" width=200 height=100>
</applet>
*/
public class MousePressedDemo extends Applet {
public void init() {
addMouseListener(new MyMouseAdapter(this));
}
}
class MyMouseAdapter extends MouseAdapter {
MousePressedDemo mousePressedDemo;
public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
this.mousePressedDemo = mousePressedDemo;
}
public void mousePressed(MouseEvent me) {
mousePressedDemo.showStatus("Mouse Pressed.");
}
}
```

Private Inner Classes

An inner class without a name is said to be nameless. This section demonstrates how creating event handlers may be made easier by using an anonymous inner class. Have a look at the applet in the listing that follows. Like previously, its objective is to have the string "Mouse Pressed" appear in the browser's or applet viewer's status bar whenever the mouse button is pushed.

```
// Anonymous inner class demo.

import java.applet.*;
import java.awt.event.*;
/*
<applet code="AnonymousInnerClassDemo" width=200 height=100>
</applet>
*/
public class AnonymousInnerClassDemo extends Applet {
public void init() {
addMouseListener(new MouseAdapter() {
public void mousePressed(MouseEvent me) {
showStatus("Mouse Pressed");
}
});
}
}
```

This programme has one top-level class, AnonymousInnerClassDemo. The addMouseListener() function is called by the init() method. An expression that defines and creates an anonymous inner class serves as its argument. Let's carefully examine this term. The compiler may be informed by the syntax new MouseAdapter() that the code within the brackets constructs an anonymous inner class.

The class also extends MouseAdapter. This expression instantiates a new class without giving it a name when it is evaluated. As AnonymousInnerClassDemo is the class in which this anonymous inner class is declared, it has access to all of the variables and methods that are included in that class's definition. As a result, it may immediately call the showStatus()

function. As was just shown, both named and anonymous inner classes provide a straightforward but efficient solution to several irksome issues. They also enable you to write code that is more effective.

CHAPTER 15
CLASSIFICATION OF APPLETS

V Haripriya, Assistant Professor,
Department of Computer Science and Information Technology, Jain (Deemed to be University) Bangalore, India
Email Id- v.haripriya@jainuniversity.ac.in

Application programmes are ones that are typically written, compiled, and run in a manner comparable to that of other programming languages. Programs are written locally on a computer, then they are built using a javac compiler and run in a java interpreter. Each application programme has one main() function.

Applet program

An applet programme is a unique application that may be included in a web page so that it has control over a specific area of the visible page. Applets are constructed from classes and vary from application programmes in that they do not have a main entry. Instead, use a variety of controls to manage certain elements of how an applet is executed.

Applet Class Hierarchy

1. Each applet we produce has to be a subclass of Applet.
2. This Applet is an expansion of Panel.
3. This Panel extends beyond the Container.
4. The Container class is an extension of Component.
5. All Java API classes descend from the Object class, which is the parent of the Component class (Figure 15.1).

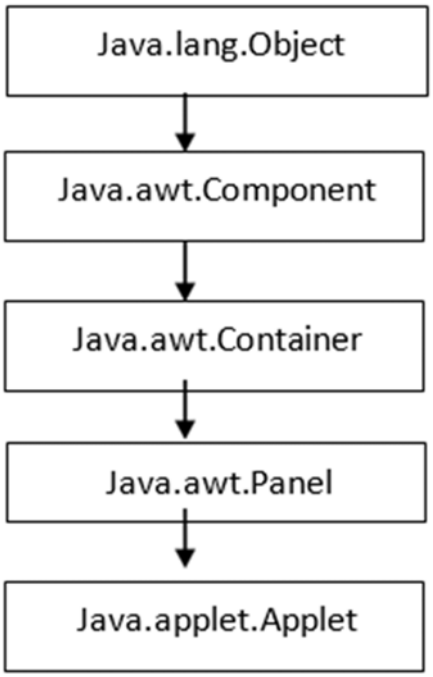


Figure 15.1: About Java.awt.Component

Concerning Java.awt.Component

1. java.lang.Object

   java.awt.Component

2. open class abstract Component grows ImageObserver, MenuContainer, and Serializable are implemented by Object.
3. A component is a physical thing with a visual representation that can be shown on a screen and can communicate with the user. The buttons, checkboxes, and scrollbars of a conventional graphical user interface are examples of components.

About Java.awt.Container

- java.lang.Object
  - java.awt.Component
    - **java.awt.Container**

1. Container is a public class that extends Component.
2. An object that may include other AWT components is known as a generic Abstract Window Toolkit (AWT) container object.
3. A list keeps track of the components that are introduced to a container. The front-to-back stacking order of the components within the container will be determined by the list's order. When a component is added to a container, the component will be inserted at the end of the list if no index is supplied (and hence to the bottom of the stacking order).
4. Example: Add()

About Java.awt.Panel

- java.lang.Object
  - java.awt.Component
    - java.awt.Container
      - **java.awt.Panel**

1. public class **Panel** extends Container implements Accessible
2. Panel is the simplest container class. A panel provides space in which an application canattach any other component, including other panels.
3. The default layout manager for a panel is the FlowLayout layout manager.

Java.applet.Applet

- java.lang.Object
  - java.awt.Component
    - java.awt.Container
      - java.awt.Panel
        - **java.applet.Applet**

public class Applet extends Panel

1. An applet is a short software that is designed to be embedded into another application rather than operate alone.
2. Every applet that is going to be viewed by the Java Applet Viewer or embedded in a Web page must have the Applet class as its superclass. A standardised interface between applets and their environment is provided by the Applet class.

The design of an applet:

An applet is a software that runs in a window. Compared to console-based applications, it has a distinct architecture. In a window-based software, we need to comprehend the following ideas:

1. Applets are event driven, to start.
2. See how the design of an applet is affected by the event-driven architecture.
3. An applet looks like a collection of interrupt service procedures.

This is how the procedure goes: An applet watches for an event to happen. By invoking an applet-provided event handler, the AWT tells the applet about an event. When this occurs, the applet must immediately perform the necessary action before handing over control to the AWT. This idea is really important. Generally speaking, your applet shouldn't operate in a mode where it retains control for a lengthy period of time. Instead, it must respond to events by taking particular actions before handing over control to the AWT run-time system. You must begin a second thread of operation whenever your applet has to carry out a repeating job on its own (for example, when it needs to scroll a message across its window).

Second, rather than the other way around, the user initiates interaction with an applet.

1. As you are aware, a programme without windows will ask the user for input before using an input method like readLine ( ). It operates differently in an applet. Instead, the user may engage with the applet whenever and whenever they choose. The applet receives these interactions as events to which it must react.
2. A mouse-clicked event is produced, for instance, when the user clicks a mouse within the applet's window. A keypress event is produced if the user presses a key when the input focus is on the applet's window. additional controls, like push buttons and check

boxes, may be included in applets. An event is produced when the user interacts with one of these controls.

Applet Initialization and Termination

When an applet begins, the AWT calls the following methods, inthis sequence:

1. **init( )**
2. **start( )**
3. **paint( )**

When an applet is terminated, the following sequence of method calls takes place:

1. **stop( )**
2. **destroy( )**

init( ):

The first method that is invoked is the init() function. You should initialise variables here. For the duration of your applet's runtime, this function is only invoked once.

start( )

After init, the start() function is invoked ( ). Restarting an applet after it has been halted is another name for it. Start() is called each time an applet's HTML content is shown onscreen, as opposed to init(), which is only called once when an applet is loaded for the first time. As a result, the applet begins execution from the beginning if a user leaves and returns to a web page ( ).

paint( ):

Each time the output of your applet has to be repainted, the paint() function is invoked. There are various possible causes for this circumstance. As an example, the window in which the applet is operating can be covered up by another window before being revealed. Instead, you may minimise and then restore the applet window. Also, when the applet starts running, paint() is called. Regardless of the reason, the applet calls paint() everytime it has to redo its output. There is one Graphics-type argument in the paint() function.

The graphics context, which specifies the graphical environment the applet is executing in, will be included in this argument. When output to the applet is necessary, this context is employed.

stop( ):

When a web browser exits the HTML document containing the applet—for example, when it navigates to another page—the stop() function is invoked. The applet is likely active when stop() is invoked. While the applet is not visible, you should use stop() to suspend any threads that are not necessary to execute. If the user visits the website again, you may restart them when start() is invoked.

destroy( )

When the environment considers that your applet has to be totally deleted from memory, the destroy() function is used. You should now release any resources that the applet may be using. Before calling destroy, the stop() function is always used ( ).

replacing update ( )

In certain circumstances, your applet may need to replace the update method, which is one of the AWT's declared methods ( ). When your applet asks to have a piece of its window repainted, this method is invoked. The default update() method calls paint after first filling an applet with

the predetermined background colour ( ). When update() is run, or anytime the window is repainted, the user will see a flash of the default background if the background is filled with a different colour in paint().

Overriding update(): function to have it do all essential display tasks is one approach to get around this issue. then just call update with paint() ( ). Hence, the applet skeleton will override paint() and update() in certain apps, as shown here:

```
public void update(Graphics g) {
// redisplay your window, here.
}
public void paint(Graphics g) {
update(g);
}
```

Use the Graphics class member drawString() to output a string to an applet. Normally, either update() or paint is where it is called ( ). It has the broad form shown below:

void drawString(String *message*, int *x*, int *y*)

Here, message is the output string that starts at x and y. The upper-left corner of a Java window is at position 0,0. Newline characters are not recognised by the drawString() function. You must explicitly indicate the exact X, Y position where you want the line to begin if you wish to begin a text line on another line.

Use setBackground() to alter the background colour of an applet's window.

Use setForeground() to change the foreground colour.

These methods have the generic forms shown below and are specified by Component.:

void setBackground(Color *newColor*)
void setForeground(Color *newColor*)

Here, *newColor* specifies the new color. The class **Color** defines theconstants shown here that can be used to specify colors:

Color.black
Color.magenta
Color.blue
Color.orange
Color.cyan
Color.pink
Color.darkGra
Color.red
Color.gray
Color.white
Color.green
Color.yellow
Color.lightGray

For example, this sets the background color to green and the text color to red:

```
setBackground(Color.green);
setForeground(Color.red);
```

1. We has to set the foreground and background colors is in the **init( )** method.
2. we can change these colors during execution of program.
3. The default foreground color is black.
4. The default background color is light gray.
5. We can obtain the current settings for the background and foreground colors by calling

getBackground( ) and **getForeground( )**, respectively. They are also defined by **Component** and are shown here:

Color getBackground( )
Color getForeground( )

Using Repaint method

1. Typically, an applet will only write to its window when the AWT calls its update() or paint() function.
2. How can the applet itself make sure that when its data changes, its window is updated? What technique, for instance, does an applet utilise to refresh the window each time the banner scrolls while it is showing a moving banner?
3. A loop in paint() that continually scrolls the banner is not possible.
4. The AWT defines the repaint() function. It triggers a call to your applet's update() function, which, in its typical implementation, calls paint, on the AWT run-time system ( )
5. There are four forms for the repaint() function.
6. This is repaint() in its most basic form:

void repaint( )

This version requires repainting the whole window.

The area that will be repainted is specified in the following version:

void repaint(int *left*, int *top*, int *width*, int *height*)

Left and top are used to provide the region's upper-left corner's coordinates, while width and height are used to convey the region's width and height. These measurements are given in pixels. Requesting that your applet be repainted shortly by using the repaint() function. Update() may not be called right away if your system is sluggish or busy. The AWT has the ability to compress many requests for repainting that happen quickly such that update() is only invoked seldom.

This may be an issue in many circumstances when a constant update time is required, including animation. Using the repaint() forms shown below is one way to deal with this issue:

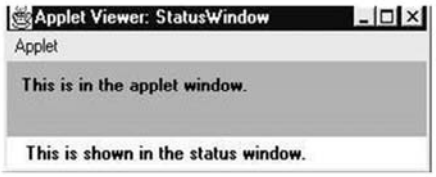
void repaint(long *maxDelay*)
void repaint(long *maxDelay*, int *x*, int *y*, int *width*, int *height*)

Status window use

An applet may emit a message to the status window of the browser or applet viewer it is executing on in addition to showing information in its own window. Use showStatus() with the string you wish to appear in order to achieve this. Giving the user feedback on what is happening in the applet, making recommendations, or even reporting any issues can all be done

via the status pane. The status window is a great tool for debugging since it makes it simple to report data about your applet.

```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/
public class StatusWindow extends Applet{
public void init() {
setBackground(Color.cyan);
}
// Display msg in applet window.
public void paint(Graphics g) {
        rawString("This is in the applet
    window.", 10, 20); showStatus("This is
    shown in the status window.");
}
}
```



Syntax of Applet tag in HTML

The syntax for the standard APPLET tag is shown here. Bracketed items are optional.

```
< APPLET
[CODEBASE = codebaseURL]
CODE = appletFile
[ALT = alternateText]
[NAME = appletInstanceName]
WIDTH = pixels HEIGHT = pixels
[ALIGN = alignment]
[VSPACE = pixels] [HSPACE = pixels]
>
[< PARAM NAME = AttributeName VALUE = AttributeValue>]
[< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
. . .
[HTML Displayed in the absence of Java]
</APPLET>
```

CODEBASE:

The CODEBASE optional property indicates the directory to look for the applet's executable class file, which is the base URL of the applet code (specified by the CODE tag). If this property is omitted, the URL directory of the HTML page is utilised as the CODEBASE. The CODEBASE need not be on the same host as the HTML file that was read.

CODE

CODE is a necessary property that specifies the name of the file containing the compiled.class file for your applet. This file is related to the applet's code base URL, which is the directory in which the HTML file was located or, if set, the directory specified by CODEBASE.

ALT

If the browser recognises the APPLET tag but is unable to execute Java applets at the moment, the ALT tag is an optional property that may be used to provide a brief text message that should

be shown. The alternative HTML you provide for browsers that don't support applets is different from this.

An optional property called NAME is used to give the applet instance a name. Applets must be given names so that other applets on the same page can locate and contact them. Use the getApplet() method, which is part of the AppletContext interface, to get an applet by name.

HEIGHT AND WIDTH

The necessary characteristics WIDTH and HEIGHT provide the size (in pixels) of the applet display area. The alignment of the applet is specified by the optional parameter ALIGN. With the following potential values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM, this property is used in the same way as the HTML IMG element.

Both VSPACE and HSPACE

These qualities are not required. VSPACE describes the area above and below the applet in terms of pixels. HSPACE describes the space on either side of the applet in terms of pixels. They get the same treatment as the VSPACE and HSPACE characteristics of the IMG tag.

NAME AND VALUE OF A PARAM

You may provide applet-specific parameters in an HTML page by using the PARAM element. Applets use the getParameter() function to retrieve their attributes.

MANAGING ANCIENT BROWSERS

Some very outdated web browsers are unable to run applets and do not comprehend the APPLET tag. You may sometimes need to work with these browsers, despite the fact that they are now all but obsolete (having been superseded by Java-compatible ones). Using HTML content and markup inside your applet>/applet> tags is the ideal method to build your HTML page to work with such browsers.

You will see the alternative markup if your browser does not recognise the applet tags. If Java is accessible, it will ignore the alternative markup and use all of the markup in between the applet> and /applet> tags.

This is the HTML to launch the Java applet SampleApplet and show a message on earlier browsers:

```
<applet code="SampleApplet" width=200 height=40>
If you were driving a Java powered browser,
you'd see &quote;A Sample Applet&quote; here.<p>
</applet>
```

Passing Parameters to Applets:

1. the APPLET tag in HTML allows you to pass parameters to your applet. To retrieve a parameter, use the **getParameter( )** method.
2. It returns the value of the specified parameter in the form of a **String** object. Thus, for numeric and **boolean** values, you will need to convert their string representations into their internal formats.
