

INTRODUCTION TO PROGRAM DESIGN WITH JAVA

A Comprehensive Guide to Object-Oriented Development

Specially Designed for CSCI 1302

University of North Georgia

AUTHORS

Mohammad Mohebbi

Paul Murray

Phillip Williams

Mingyuan Yan

Jianjun Yang

First Edition

2026

Table of Contents

Chapter 1: Using Classes and Objects	1
1.1 What Java Is and Why It Matters	1
1.2 Java Is Object-Oriented.....	1
1.3 Begin from a Simple Java Program.....	2
1.4 Creating and Using an Object.....	3
1.5 Defining a Method.....	4
1.6 Using a Constructor	5
1.7 Encapsulation (Getters/Setters)	6
1.8 Objects Working Together (Composition)	7
1.9 Understanding References (Aliasing)	8
1.10 Arrays of Objects.....	9
1.11 Full Programs on using classes and objects	9
Self-Review Questions	12
Chapter 2: Conditionals and Loops.....	14
2.1 Boolean expressions and operator precedence	14
2.1.1 Introduction to Boolean Expressions.....	14
2.1.2 Relational Operators.....	14
2.1.3 Logical Operators	14
2.1.4 Objects vs. Primitives in Equality	14
2.1.5 Operator Precedence.....	14
2.1.6 Example of Precedence in Action.....	15
2.2 If, If-Else, and If-Else-If.....	15
2.2.1 If statement	15
2.2.2 If-Else statement.....	17
Example 2.3: A Simple If-Else statement.....	17
2.2.3 If-Else-If statement	18
2.3 switch and switch expressions	21
2.4 Ternary operator and guard clauses.....	22
2.4.1 Using the Ternary Operator and Guard Clauses to Improve Code Clarity.....	22
2.4.2 The Ternary Operator: A Compact Conditional Expression.....	22
2.5 For loops (classic and enhanced).....	23
2.5.1 Understanding Classic for Loops and Enhanced for-each Loops in Java	23
2.5.2 The Classic for Loop: Full Control Over the Iteration Process.....	23
2.5.3 The Enhanced for Loop: Cleaner, Safer Iteration for Collections and Arrays	24
2.6 while and do-while loops	25
2.6.1 The while Loop: Repeating as Long as a Condition Is True	25
2.6.2 The do-while Loop: Run First, Then Check.....	25
2.7 Nested loops and computational complexity	26

2.8 Loop control: break, continue, and labeled breaks	26
Self-Review Questions.....	27
Chapter 3: Writing Classes	28
3.1 Review of Classes and Objects	28
3.2 Structure and Contents of a Class	29
3.3 What is Instance Data?	30
3.4 Structure of Class Methods	31
Chapter 4: Graphical User Interface	38
The JavaFX Framework.....	38
Graphical User Interface (GUI) controls, events, and event handlers	39
Chapter 5: Arrays.....	60
Introduction to Arrays	60
Array Basics: Declaration, Creation, and Initialization	60
Declaration of Arrays	60
Creation of Arrays	61
Array Indexing and Access.....	61
Operation on Arrays	62
Enhanced for loop (for-each).....	63
Arrays Storing Reference Data	64
Arrays as Parameters.....	65
Passing arrays to methods	66
Modifying array elements inside a method	68
Returning arrays	70
Variable-Length Parameters (varargs)	71
Arrays and Command-Line Arguments.....	73
Multidimensional Arrays	74
Two Dimensional Arrays	74
Multi-dimensional Arrays.....	78
The java.util.Arrays Utility Class	79
The Hidden World Inside Arrays.....	79
Chapter 6: Inheritance.....	82
What is Inheritance?.....	82
Basics.....	82
UML Class Diagram (for Inheritance)	83
How Subclasses Inherit Variables and Methods.....	84
Understanding Access Modifier for variables and methods	84
Inheritance of Fields (Variables).....	85
Inheritance of Methods.....	85
Method Overriding	86
Super Reference	87

Calling the Superclass Constructor with super(...)	88
Using super to Access Superclass Methods and Variables	88
Shadow variables	88
Class Hierarchy	90
The Java Platform Class Hierarchy	90
Frequently Overridden Methods from the Object Class.....	91
Inheritance and Subclass Constructors	92
Class Visibility and Inheritance	93
Abstract Classes.....	93
<i>Chapter 7: Polymorphism</i>	<i>97</i>
7.1 Types of polymorphism	97
7.2 When to use polymorphism	98
7.3 Casting- Upcasting and Downcasting	103
7.4 Avoid hiding variables	111
<i>Chapter 8: Exception Handling and Working with Files.....</i>	<i>113</i>
8.1 Errors vs Exceptions	114
8.2 try..catch statements.....	114
8.3 Common exceptions	115
8.4 What code goes inside a try block	116
8.5 Checked or unchecked exceptions	116
8.6 Exception propagation	116
8.7 Using multiple exceptions in a single catch	119
8.8 Using multiple catches.....	119
8.9 The finally block and try-with-resources	120
8.10 Creating your own exception.....	121
8.11 Working with files.....	123
8.12 Types of files that can be used	123
8.13 Invisible markers.....	124
8.14 Delimiters, tokens, and iterator objects.....	124
8.15 Using more than one delimiter	127
8.16 Working with longer tokens	129
<i>Chapter 9: Time Complexity Analysis of Algorithms</i>	<i>130</i>
Examples to practice calculating time complexity of functions	132
Some nonpolynomial functions	134
Finding time complexity, or Big-O, of a method via programming.....	135
Time Complexity Lab.....	137
How to draw a time complexity graph in Excel?	137

Chapter 10: Introduction to Collections	139
Linear vs. Non-Linear collections	140
Abstract Data Types (ADT): Lists, Stacks, and Queues	140
Generic Classes and Generic Methods	141
Generic methods.....	141
Creating a generic class	142
Chapter 11: LinkedList and Stack	143
The PersonNode Class.....	143
Activity:	145
Traversing through a LinkedList:	145
Java Container: Stack.....	146
How to use Stack class of Java?	146
Activity 1:	146
Some practices with stacks:	146
Activity 2:	147
Activity 3:	147
Activity 4:	148
Chapter 12: Introduction to Recursion	149
Definition of Recursion	149
Essential parts of a recursive solution	149
Solving problems with recursion.....	149
Hanoi Tower problem.....	149
Recursive solution for Hanoi Tower problem:	150
Time Complexity of Hanoi Tower solution	151
Computational overhead of recursive solutions	152
Possibility of repetition in solving a subproblem	152
More recursive examples:	152

Chapter 1: Using Classes and Objects

1.1 What Java Is and Why It Matters

Java is a high-level, object-oriented programming language designed to help developers build reliable and portable software systems. At its core, Java provides a structured way to give instructions to a computer through a clear and readable syntax. Unlike languages that run directly on hardware, Java programs are executed by the Java Virtual Machine (JVM), which allows the same program to run on different operating systems without modification. This “write once, run anywhere” capability has made Java one of the most influential and widely adopted languages in the history of computing.

A defining characteristic of Java is its object-oriented programming model. Java organizes data and functionality into “objects”, which are self-contained units representing things, behaviors, or concepts in the software system. This structure helps beginners understand modular design and enables developers to build complex applications in a more maintainable and scalable way. Java is also strongly typed, which means that programmers must clearly specify the type of data they are working with. This reduces common errors and encourages disciplined programming practices, making Java an excellent language for students who are learning foundational concepts.

Java matters because it is a truly versatile language that powers a broad range of modern technologies. It is the backbone of many enterprise-level applications used in banking, healthcare, transportation, and e-commerce. Java is also a leading language for Android development, enabling developers to create mobile apps used by billions of people. In addition, Java supports robust web development frameworks such as Spring, which are widely used by companies needing secure, scalable, and high-performance systems. Its stability, portability, and extensive library ecosystem make Java a dependable choice for long-term software development.

For beginners, learning Java is an investment that opens doors to many career paths. The language teaches strong fundamentals, such as algorithms, data structures, and object-oriented design—that transfer to almost every other modern programming language. Java’s large community, extensive documentation, and abundance of learning resources further support new learners as they develop confidence and problem-solving skills.

Java is both a foundational and forward-looking programming language. It provides a powerful yet accessible way for beginners to understand how software works while enabling professionals to build systems that drive industry, innovation, and everyday digital life.

1.2 Java Is Object-Oriented

Java is fundamentally an object-oriented programming (OOP) language, and this design philosophy shapes every aspect of how programs are written, organized, and executed. To understand Java, one must first understand what it means for a language to be object-oriented. In its simplest form, object orientation is a way of organizing software around “objects”—entities that combine data and behavior into a single, coherent unit. Java’s commitment to this paradigm allows developers to model complex systems using structures that closely resemble real-world concepts, making programs easier to understand, extend, and maintain.

The core of Java's object-oriented structure includes classes and objects. A class is a blueprint that defines the properties and behaviors of something, while an object is a concrete instance created from that blueprint. For example, a Car class may define attributes such as color, speed, and model, along with methods that describe actions like accelerating or braking. By creating multiple objects from this class, Java allows programmers to represent many individual cars, each with its own data yet sharing the same set of behaviors. This abstraction is a powerful way to reduce complexity when building large software systems.

Java's object-oriented nature is also reflected in its support for encapsulation, one of the fundamental OOP principles. Encapsulation means grouping data and the methods that operate on that data into a single unit and restricting direct access to the internal state. Java enforces this through access modifiers such as `private`, `protected`, and `public`. This not only improves program reliability by preventing accidental modification of critical data but also promotes clean, modular code that is easier to debug and reuse.

Another key principle supported by Java is inheritance, which allows one class to acquire the attributes and behaviors of another. Inheritance facilitates code reuse and reduces redundancy. For instance, a Truck class can inherit from the Car class, automatically gaining properties like speed and methods like `startEngine()` without rewriting them. This hierarchical relationship enables developers to create more specialized classes while keeping the overall design consistent and efficient.

Java also embraces polymorphism, the ability for different classes to respond differently to the same method call. Through method overriding and interfaces, Java allows developers to write flexible code that adapts to different contexts. For example, both Car and Bicycle may implement a `move()` method, but each will behave differently when the method is called. This flexibility makes Java well-suited for designing systems that must interact with various types of objects through a common interface.

In addition to these classical OOP features, Java encourages object-oriented thinking through its standard library and application frameworks. Nearly every component of Java, such as strings, collections, graphical interfaces, file handlers, is represented as an object. This consistency strengthens the developer's ability to reason about software in terms of interacting objects rather than isolated commands or procedures.

Java is object-oriented because it structures software around classes and objects, enforces key principles such as encapsulation, inheritance, and polymorphism, and promotes modular, scalable design. For beginners and experts alike, Java's object-oriented model provides a powerful and disciplined way to build robust and maintainable applications. Understanding this model is essential for mastering the language and appreciating its strengths in both education and industry.

1.3 Begin from a Simple Java Program

Here we use a simple Java class Dog to illustrate a concise example of how object-oriented programming organizes data and behavior into a single, coherent unit. This small program illustrates several foundational Java concepts, including classes, instance variables, methods, and the relationship between objects and the data they store.

Example 1.1: A Simple Class Dog

```
public class Dog {
    String name;
    int age;
```

```

public void bark() {
    System.out.println(name + " says: Woof!");
}
}

```

In Java, a class is a blueprint that describes what an object should contain and what it should be able to do. The Dog class serves this purpose by defining the essential characteristics and behavior of a dog in a program. Inside the class, two instance variables, name and age, are declared. These variables represent the attributes of a dog, such as what the dog is called and how old it is. The term “instance variable” means that each object created from the Dog class will have its own independent copy of these variables. If one Dog object is named “Buddy” and another is named “Coco,” their names are stored separately, reflecting the idea that each object maintains its own internal state.

The class also defines a method called bark(), which represents a behavior of the dog. Methods in Java correspond to actions or operations that an object can perform. The bark() method prints a message to the console using the dog’s current name, followed by the phrase “says: Woof!” When this method executes, Java looks at the name variable belonging to the specific object that called the method. This connection between data and behavior is a central idea in object-oriented programming: methods operate on the data of the object they belong to.

An interesting detail is that if the name variable has not been assigned a value, Java automatically sets it to null, which means “no value.” As a result, calling bark() before initializing the dog’s name would produce the output null says: Woof! This behavior demonstrates two important lessons for beginners: Java assigns default values to instance variables, and programmers must ensure that object fields are properly initialized before use.

Overall, the Dog class illustrates the fundamental OOP structure used throughout Java. It shows how classes define attributes and behaviors, how objects created from those classes carry their own data, and how methods use that data to perform actions. This class is a blueprint for Dog objects. It defines two instance variables (name, age) and one behavior (bark). Instance variables belong to each object created from the class; different Dog objects can hold different names/ages. Method bark() reads the current object’s name and prints a message. If name is null, Java prints 'null says: Woof!', a reminder to initialize fields. Even in this simple example, the essential principles of abstraction, encapsulation of data within objects, and behavior-driven methods are already present, making it an excellent starting point for anyone beginning to learn Java and object-oriented programming.

1.4 Creating and Using an Object

Let’s continue the Dog class to explore the creation and features of object.

Example 1.2: Continued Dog Class

```

Dog myDog = new Dog();
myDog.name = "Lucky";
myDog.age = 3;
myDog.bark();

```

The short Java program 1.2 that creates and uses a Dog object is a straightforward demonstration of how object-oriented programming works in practice. Although the code

consists of only a few lines, it illustrates foundational concepts such as object instantiation, field access, method invocation, and how objects maintain their own internal state during program execution.

The statement `Dog myDog = new Dog();` is the first critical step. `Dog()` is called constructor, which is to create a new object. Here, the `new` keyword instructs the Java runtime to allocate memory for a new `Dog` object on the heap, which is the region of memory reserved for dynamically created objects. This process is called instantiation, and it transforms the abstract blueprint defined by the `Dog` class into an actual, usable object. The memory address (or reference) of this newly created object is then stored in the variable `myDog`. Rather than holding the object itself, `myDog` holds a reference that points to the object in memory, allowing the program to interact with it.

Once the object has been created, the program uses the dot operator (`.`) to access and modify the object's fields:

```
myDog.name = "Lucky"; and myDog.age = 3;
```

The dot operator is the mechanism Java provides to reach inside an object and assign values to its instance variables. Each `Dog` object maintains its own independent name and age, and the assignments above modify only the object referenced by `myDog`. If several `Dog` objects existed simultaneously, each one could hold different values, representing different real-world dogs. This demonstrates Java's principle that an object carries its own internal state.

The call `myDog.bark();` activates the behavior defined in the `Dog` class. When this method is invoked, Java automatically uses the data belonging to the object referenced by `myDog`. Because the program previously set the name to "Lucky," the method prints:

```
Lucky says: Woof!
```

This illustrates how methods and instance variables work together in object-oriented programming. The behavior `bark()` depends on the current state (name) of the specific object that calls it.

This simple sequence—creating an object, setting its attributes, and invoking one of its behaviors—captures the essence of object-oriented programming. Objects are created from classes, they hold their own data, and they expose methods that operate on that data. The program also highlights the importance of initialization, as behavior often depends on properly assigning values before methods are called.

This small piece of code provides an excellent example of how Java objects are created, configured, and used. It demonstrates the relationship between classes and objects, the significance of object references, the role of the dot operator, and the tight coupling between an object's state and its behaviors. These principles form the foundation for understanding more complex Java programs and learning how to design software using object-oriented techniques.

1.5 Defining a Method

The `bark()` method in the `Dog` class provides a simple yet informative example of how Java uses methods to represent the behaviors of an object. Although the method contains only one line of code, it demonstrates several important principles of object-oriented programming, including encapsulated behavior, access to instance data, and the distinction between methods that return values and those that simply perform an action.

```
public void bark() {
    System.out.println(name + " is barking.");
}
```

The method header, `public void bark()`, specifies both the method's visibility and its purpose. The keyword `public` means that this behavior can be invoked from outside the class by any code that has access to a `Dog` object. This is consistent with modeling real-world interactions: if an object represents a dog, any part of the program should be able to request that the dog bark. The keyword `void` indicates that the method does not return a value. Instead of computing and sending back information, the method performs an action—printing a message to the console. The empty parentheses signify that the method requires no external input or parameters; all the information it needs is already stored within the object.

Inside the method, the statement `System.out.println(name + " is barking.");` accesses the object's internal state. The expression uses the instance variable `name`, which belongs to the specific `Dog` object that called the method. This detail highlights one of the defining characteristics of object-oriented programming: methods operate on the data contained within the object they belong to. If multiple `Dog` objects exist, each one carries its own name, and calling `bark()` on each will produce a personalized message, such as "Lucky is barking." or "Coco is barking." The method does not modify the object's state; it simply reads the current value of `name` to generate output. This makes `bark()` a behavioral method rather than a state-changing one.

The simplicity of this method belies its instructional value. It shows how Java links data and behavior within the same object, how methods encapsulate actions, and how an object's state influences its behavior at runtime. Understanding methods like `bark()` is essential for learning how objects interact, respond to requests, and reflect their internal data through visible actions.

1.6 Using a Constructor

The updated `Dog` class introduces one of the most important concepts in object-oriented programming: the constructor. A constructor is a special method that runs automatically whenever a new object is created, allowing programmers to initialize an object's state at the moment of instantiation. This mechanism ensures that every object begins its life with meaningful, predictable values rather than relying on default settings.

Example 1.3: Defining the Constructor `Dog`

```
public class Dog {
    String name;
    int age;

    public Dog(String n, int a) {
        name = n;
        age = a;
    }
}

// Usage
Dog d = new Dog("Bella", 5);
```

In the example `Dog` class, it defines two instance variables, `name` and `age` that represent the characteristics of a dog. Unlike local variables, instance variables belong to the object itself. Every `Dog` object contains its own copy of these fields, allowing different objects to store different names and ages. Without initialization, Java would assign default values (null for `String` and 0 for `int`), which may not represent realistic or useful information. The constructor solves this by enforcing explicit initialization.

In example 1.3, the constructor `public Dog(String n, int a)` illustrates how Java binds external input to the internal state of an object. The parameters `n` and `a` serve as placeholders for the values provided at the moment the object is created. Within the constructor body, the assignments `name = n;` and `age = a;` store these values in the object's own instance variables. This is the defining purpose of a constructor: to prepare the object so that it is fully configured and ready for use immediately after creation.

The usage example, `Dog d = new Dog("Bella", 5);`, demonstrates this process in action. The new `Dog("Bella", 5)` expression instructs Java to allocate memory for a new `Dog` object and immediately run the constructor with "Bella" and 5 as arguments. As a result, the newly created object, referenced by `d`, has its `name` field set to "Bella" and its `age` field set to 5. This ensures that the object begins in a valid and meaningful state without requiring additional assignment statements.

This pattern reflects a fundamental principle of object-oriented design: objects should be born initialized. Constructors provide a reliable, consistent mechanism for establishing an object's state, reducing errors and making programs easier to reason about. Instead of creating an object and then manually assigning its fields, the programmer expresses all necessary information at creation time, improving code clarity and reducing the opportunity for uninitialized or inconsistent states.

This version of the `Dog` class demonstrates how constructors provide structure, reliability, and convenience in object-oriented programming. By allowing values to be supplied during object creation, constructors help ensure that every object is fully initialized, self-contained, and ready to perform its behaviors from the moment it comes into existence. This makes constructors an essential tool for writing clean, robust Java programs.

1.7 Encapsulation (Getters/Setters)

We use `Student` class as a clear example to show encapsulation, a foundational principle of object-oriented programming. Encapsulation refers to the practice of bundling data (fields) with the methods that operate on that data, while also controlling how the data is accessed or modified from outside the class. In Java, encapsulation is implemented primarily through access modifiers and getter/setter methods, as demonstrated in this code.

Example 1.4: Student Class

```
public class Student {
    private String name;
    private int id;

    public Student(String n, int i) {
        name = n;
        id = i;
    }

    public String getName() { return name; }
    public void setName(String n) { name = n; }
    public int getId() { return id; }
    public void setId(int i) { id = i; }
}
```

The instance variables `name` and `id` are declared as `private`, which means that they cannot be accessed directly by code outside the `Student` class. This restriction is intentional. By hiding the internal data, the class protects its fields from unintended or unsafe modifications. Without

encapsulation, any external code could freely change a student's name or ID, potentially causing inconsistencies or violating program rules. The `private` keyword ensures that all interactions with these fields must occur through well-defined methods, giving the class full control over how its data is managed.

To allow controlled access to the fields, the class provides public methods known as getters and setters. The getter method `getName()` returns the current value of the name field, enabling other parts of the program to read the student's name without having direct access to the underlying variable. The setter method `setName(String n)` allows the name to be updated, but only through a designated method that the class author can customize. For example, validation logic or formatting rules could be added later without changing any code outside the class. The `id` field is exposed only through a getter method, `getId()`, meaning the ID can be read but not changed after the object is created. This design reflects a common real-world constraint: IDs are often fixed and should not be reassigned once a student is created.

The constructor `public Student(String n, int i)` initializes the student's name and ID when a new object is created. By assigning values through the constructor, the class ensures that every `Student` object begins in a valid state, with both fields properly initialized. This complements the goals of encapsulation by preventing partially formed or inconsistent objects from existing in the system.

Overall, this example illustrates how encapsulation promotes reliability, maintainability, and data integrity in Java programs. By hiding fields behind a controlled interface of getters and setters, the class protects its internal state from accidental misuse while still allowing other parts of the program to interact with it in safe, predictable ways. Encapsulation is one of the key reasons object-oriented programming leads to cleaner and more robust software designs, and the `Student` class provides a simple yet powerful demonstration of how this principle works in practice.

1.8 Objects Working Together (Composition)

The `Engine` and `Car` classes provide an excellent introduction to composition, a fundamental technique in object-oriented programming where one object is built from or "composed of" other objects. Composition allows software designers to model real-world relationships in a modular and intuitive manner, and it is one of the clearest demonstrations of how objects collaborate to perform meaningful tasks.

Example 1.5: Engine and Car classes for composition

```
public class Engine {
    public void run() {
        System.out.println("Engine is now running.");
    }
}

public class Car {
    private Engine engine = new Engine();
    public void start() {
        engine.run();
    }
}
```

In this example, the Engine class represents a simple component with one behavior: the ability to run. Its method `run()` prints a message indicating that the engine is operating. Although the class is small, it follows the object-oriented principle that each class should represent a single, coherent concept. The Engine object encapsulates all behaviors related to running a vehicle's engine, keeping the functionality isolated and easy to maintain.

The Car class uses composition to incorporate an Engine object as one of its internal parts. The line `private Engine engine = new Engine();` creates an engine and stores it inside the car. This mirrors the real world: every car has an engine as a component. By marking the field private, the class ensures that only the Car class itself can directly access or manipulate the engine. External code cannot tamper with the engine, reinforcing strong encapsulation and making the Car class responsible for coordinating its internal components.

The `start()` method demonstrates how objects work together through composition. When a program calls `car.start()`, the Car object does not do all the work itself. Instead, it delegates part of the task to its internal engine by calling `engine.run()`. This delegation shows one of the biggest advantages of composition: complex behavior emerges from the collaboration of simpler, well-defined objects. The car provides the higher-level operation “start the car,” while the engine performs the lower-level action “run the engine.” Each object has a clear role, and together they produce the complete behavior expected of a car.

This design pattern makes programs more modular, flexible, and maintainable. If the program later needs a different type of engine, perhaps an electric engine instead of a combustion engine, the Car class can easily be modified to use a different engine object without rewriting the rest of its logic. Composition therefore supports extensibility and promotes clean software architecture.

The Engine and Car classes illustrate how composition allows objects to collaborate to achieve more complex behaviors than they could on their own. By embedding an Engine object inside a Car object and delegating work through method calls, Java models real-world structures in a clear and intuitive way. Composition is one of the core strengths of object-oriented programming, and this example shows how even simple classes can combine to form powerful and expressive designs.

1.9 Understanding References (Aliasing)

In Java, aliasing occurs when two or more reference variables point to the same object in memory. Because Java variables for objects do not store the objects themselves—only references to them—assigning one variable to another (e.g., `b = a`) does not create a copy. Instead, both variables become aliases that share the same underlying data. Any modification made through one reference is immediately visible through the others, since they all access the same object. Aliasing is an important concept because it can lead to confusing bugs if a programmer assumes a new object was created when only the reference was copied. Understanding aliasing helps developers reason correctly about object behavior, avoid unintended side effects, and write safer, more predictable code.

The example 1.6 demonstrates an important concept in Java known as object references or aliasing. When the statement `Dog a = new Dog();` is executed, a new Dog object is created in memory, and the variable `a` stores a reference to that object. After setting `a.name = "Max";`, the `name` field inside the same object is updated. When we write `Dog b = a;`, no new object is created—both `a` and `b` now point to the same Dog object in memory. Therefore, when `b.name = "Changed";` is executed, it modifies the shared object. As a result, printing `a.name` also outputs

"Changed", because a and b are simply two references to one underlying object. This illustrates that assigning one object reference to another does not create a copy; it creates an alias.

Example 1.6 : Aliasing

```
Dog a = new Dog();
a.name = "Max";
Dog b = a; // b refers to the same object as a
b.name = "Changed";
System.out.println(a.name); // prints "Changed"
```

1.10 Arrays of Objects

In Java, an array of objects is actually an array of references, not an array containing the objects themselves. When you create an array such as `Dog[] dogs = new Dog[3];`, Java allocates space for three object references, all initially set to null. No Dog objects exist yet. To use the array, you must create each object individually and assign it to an array position, such as `dogs[0] = new Dog();`. Each element then holds a reference to a distinct Dog instance, allowing you to access fields and methods through expressions like `dogs[0].name = "A";`. This design highlights an important concept: Java arrays store pointers to objects, and every object in the array must be explicitly instantiated before it can be used.

The example 1.7 illustrates how arrays of objects work in Java. When the statement `Dog[] dogs = new Dog[3];` is executed, Java creates an array that can hold three references to Dog objects, but it does not create any actual Dog objects yet. At this point, all three elements of the array contain null. The next three lines, `dogs[0] = new Dog();`, `dogs[1] = new Dog();`, and `dogs[2] = new Dog();`, create three separate Dog objects and assign their references into the array positions. Now each array element holds a different Dog instance. Finally, the code sets the name field for each Dog object: "A", "B", and "C". Because each element refers to a distinct object, changing one Dog's name does not affect the others. This example demonstrates that arrays of objects store references, not the objects themselves, and that each element must be individually instantiated before use.

Example 1.7: Arrays of Objects

```
Dog[] dogs = new Dog[3]; // array of references, not objects yet
dogs[0] = new Dog();
dogs[1] = new Dog();
dogs[2] = new Dog();
dogs[0].name = "A";
dogs[1].name = "B";
dogs[2].name = "C";
```

1.11 Full Programs on using classes and objects

The BankAccount program is an excellent illustration of how Java uses classes and objects to organize software around real-world concepts. In Java, a class defines the structure and capabilities of something, while an object is an actual, active instance created from that class during program execution. By examining this program, we can clearly see how classes describe data and behavior, and how objects maintain their own state and perform actions through method calls.

Example 1.8: Full Program – BankAccount

```

    public class BankAccount {
    private String owner;
    private double balance;

    public BankAccount(String owner, double initial) {
        this.owner = owner;
        this.balance = Math.max(0, initial); // prevent negative start
    }

    public void deposit(double amount) {
        if (amount <= 0) throw new IllegalArgumentException("amount must be positive");
        balance += amount;
    }

    public boolean withdraw(double amount) {
        if (amount <= 0 || amount > balance) return false;
        balance -= amount;
        return true;
    }

    public double getBalance() { return balance; }

    public void show() {
        System.out.println(owner + " has $" + String.format("%.2f", balance));
    }
}

public class TestBank {
    public static void main(String[] args) {
        BankAccount a = new BankAccount("Alice", 200);
        a.deposit(50);
        boolean ok = a.withdraw(30);
        a.show(); // Alice has $220.00
        System.out.println("Withdraw success? " + ok);
    }
}

```

The `BankAccount` class serves as a blueprint that specifies what every bank account object will contain and what operations it can perform. It defines two private instance variables, `owner` and `balance`, which together represent the *state* of any bank account object created from this class. Because these fields are private, only methods inside the class may change or inspect them—an important aspect of encapsulation. Every time the program creates a `BankAccount` object, that object receives its own independent copy of these fields. Thus, multiple objects created from the same class can behave similarly but maintain different internal values.

The constructor, `public BankAccount(String owner, double initial)`, is another crucial component of the class. A constructor runs automatically whenever a new object is created. When the statement `new BankAccount("Alice", 200)` executes, Java allocates memory for a brand-new `BankAccount` object, initializes its `owner` field with "Alice", and sets its `balance` field to the provided initial amount (never below zero). This shows how a class defines *how objects come into existence* and ensures that they start with valid, predictable state.

The behaviors of a bank account are represented through the class's instance methods. When an object calls `deposit`, it updates its own `balance`. When an object calls `withdraw`, it

checks its own balance before subtracting money. When an object calls `show`, it prints its owner and balance. All of these actions are tied to the specific object that receives the message. Even though the methods belong to the class, each method operates on the unique state stored inside the object that invoked it. This distinction—shared behavior but individual state—is at the heart of object-oriented programming.

The `TestBank` class demonstrates how objects created from a class participate in a program. The statement

```
BankAccount a = new BankAccount("Alice", 200);
```

creates a `BankAccount` object and assigns its reference to the variable `a`. Everything that follows

```
a.deposit(50);
a.withdraw(30);
a.show();
```

is simply a sequence of messages sent to that object. The object responds by updating its internal state, validating operations, and printing information. Importantly, these method calls affect only that single object; another bank account object would maintain a completely separate balance and owner, proving that each object behaves independently even when created from the same class.

Overall, this program highlights the essential relationship between class and object in Java. The class defines what data and behaviors a bank account should have, while objects represent actual, functioning accounts that store their own data and carry out their own operations. Through constructors, encapsulated fields, and instance methods, the `BankAccount` class provides a complete specification for creating reliable bank account objects. The `TestBank` program then shows how such objects interact with the rest of the system in a realistic way. The example makes clear that object-oriented programming in Java is fundamentally about defining classes and then using those classes to create objects that model meaningful entities in a software system.

Example 1.9: Calculator (Stateless Utility)

```
public class Calculator {
    public int add(int a, int b) { return a + b; }
    public int sub(int a, int b) { return a - b; }
    public int mul(int a, int b) { return a * b; }
    public double div(int a, int b) {
        if (b == 0) throw new ArithmeticException("division by zero");
        return (double) a / b;
    }
}
```

The `Calculator` program is a simple but effective example of how Java uses classes and objects to organize behavior in an object-oriented manner. In this program, the `Calculator` class defines a set of arithmetic operations—addition, subtraction, multiplication, and division—that any calculator object created from this class can perform. Even though the class does not store internal data, it still demonstrates the core idea that a class bundles related operations together and provides them as behaviors that objects can execute.

The class is named `Calculator`, and it contains four instance methods: `add`, `sub`, `mul`, and `div`. Each method receives input parameters and returns the result of the corresponding arithmetic operation. For example, the method

```
public int add(int a, int b) { return a + b; }
```

tells us that any Calculator object can take two integers and compute their sum. The implementation is simple, but the structure demonstrates an important point: the *class* defines the operation, and the *object* performs the operation when the method is called.

The subtraction and multiplication methods work in the same way. Each one accepts two parameters, performs the requested calculation, and returns the result. Because the methods belong to the Calculator class, they represent the capabilities that every calculator object will have. When a program creates a Calculator object using

```
Calculator c = new Calculator();
```

the variable *c* holds a reference to an object that can now execute all of the arithmetic methods defined in the class.

The division method, *div*, adds an extra layer of robustness. It performs the same role as the other methods but includes a safety check:

```
if (b == 0) throw new ArithmeticException("division by zero");
```

This prevents the object from performing an invalid operation. If a program calls *c.div(10, 0)*, the calculator object informs the programmer that the operation is mathematically impossible. After validating the input, the method returns the result as a double, ensuring that the division is performed with decimal precision.

Although the class does not have instance variables, it still demonstrates object-oriented principles. The Calculator class groups related behaviors into a single unit, and the methods become the actions that a Calculator object can perform. When a program needs to perform arithmetic, it creates a calculator object and sends it messages in the form of method calls. Each call triggers the corresponding behavior defined in the class.

In summary, the Calculator program illustrates how Java uses classes to define a set of operations and how objects created from those classes perform the actual work. The program reinforces the idea that a class serves as a blueprint and that methods represent the behaviors that objects can carry out. Even a simple class like Calculator demonstrates the power of organizing related operations into an object-oriented structure, making the program clear, modular, and easy to extend.

Self-Review Questions

1. What does “Java is object-oriented” mean?
2. What is the difference between a class and an object?
3. In the Dog class, what happens if you call *bark()* before assigning a name?
4. What does the *new* keyword do in Java?
5. Why do we use constructors?
6. Why are fields in the Student class private?
7. What is composition? Give an example from the chapter.
8. What is aliasing in Java references?
9. Why does *new Dog[3]* not create 3 Dog objects?
10. In the BankAccount class, why does *withdraw()* return a boolean?

Programming projects:

1. Create a Cat class with fields *name* and *color*, a constructor, and a *meow()* method.
2. Write a program that creates two Dog objects with different names and calls *bark()* on both.
3. Add a setter *setAge(int newAge)* to the Dog class and update a dog’s age.
4. Create a class Book with private fields *title* and *price*, getters, and a *show()* method.

5. Demonstrate aliasing using a Student object.
6. Create an array of 5 BankAccount objects with different owners.
7. Modify the Calculator class to add a method square(int x) that returns x^2 .
8. Create a Car that uses an ElectricEngine instead of a normal Engine (composition).
9. Add a deposit() and withdraw() test program: deposit 100, withdraw 30, print balance.
10. Create a method in Dog called info() that prints "Name: X, Age: Y".

Chapter 2: Conditionals and Loops

2.1 Boolean expressions and operator precedence

2.1.1 Introduction to Boolean Expressions

Boolean expressions are fundamental to decision-making in Java. They determine program flow by evaluating to true or false. Java uses the primitive type `boolean` to represent these values. Conditional logic in if-statements, loops, and switch expressions all depend on Boolean expressions.

2.1.2 Relational Operators

Relational operators compare two values and produce a Boolean result:

Operator	Meaning
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code><</code>	Less than
<code><=</code>	Less than or equal to
<code>></code>	Greater than
<code>>=</code>	Greater than or equal to

2.1.3 Logical Operators

Logical operators allow combining or negating multiple Boolean expressions:

- `&&` (AND)
- `||` (OR)
- `!` (NOT)

Java uses short-circuit evaluation for `&&` and `||`. This means Java stops evaluating as soon as the outcome is determined. For example, in `(a != null && a.length() > 0)`, the second part is evaluated only if `a != null` is true.

2.1.4 Objects vs. Primitives in Equality

In Java, `==` behaves differently depending on the data type:

- For primitives: compares actual values.
- For object references: checks if two references point to the same object.

To compare object content, use `equals()` or `Objects.equals(a, b)`.

2.1.5 Operator Precedence

Java follows strict rules about which operators are evaluated first. The precedence order from highest to lowest is shown below.

Highest
Arithmetic Operators

Comparison Operators
(`<`, `<=`, `>`, `>=`)

Equality Operators
(`==`, `!=`)

Logical AND (`&&`)
Logical OR (`||`)

2.1.6 Example of Precedence in Action

Consider the following expression:

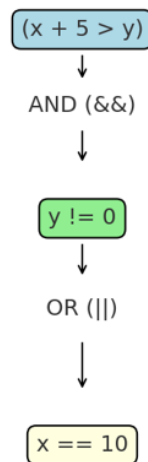
```
boolean result = x + 5 > y && y != 0 || x == 10;
```

According to precedence rules, this is equivalent to:

```
((x + 5) > y) && (y != 0) || (x == 10)
```

Using parentheses improves readability and avoids logical mistakes.

The diagram below illustrates how a complex Boolean expression is evaluated:



Understanding Boolean expressions is essential for mastering Java. They control program flow, influence conditions in loops and branches, and determine logical outcomes in complex expressions. Mastery of operator behavior, precedence, and short-circuit logic leads to clearer, safer, and more reliable code.

2.2 If, If-Else, and If-Else-If

2.2.1 If statement

The `if` statement executes a block when a condition is true. Use braces `{ }` even for single statements for clarity and to avoid bugs when modifying code.

The example 2.1 is a simple but meaningful example of how decision-making works in Java programming. Although only one conditional statement is used, it demonstrates the fundamental idea that programs can choose different actions depending on the values of variables.

Example 2.1: A Simple if Statement

```
if (score >= 90) {
    grade = 'A';
}
```

This statement represents a basic if condition, one of the most important control structures in any programming language. Its purpose is to evaluate whether a certain requirement is met, in this case, whether a student's score is high enough to receive an A. The condition inside the parentheses, `score >= 90`, is a boolean expression: it compares the value of `score` to the number 90 and produces either true or false. If the student's score is 90 or higher, the expression becomes true; if the score is less than 90, the expression becomes false.

When the condition evaluates to true, the Java program executes the statement inside the curly braces. Here, that action is a simple assignment: the variable `grade` is set to the character 'A'. This means that students who meet the criterion, earning a score of 90 or above, are awarded the top grade. If the condition evaluates to false, the program does nothing and simply moves on to the next line of code. The if statement, therefore, acts as a gatekeeper, allowing only certain inputs to produce an effect.

This kind of logic mirrors everyday decision-making. For example, we might say, "If the temperature is below freezing, wear a coat." In the same way, the program says, "If the score meets the standard, give an A." Just as people use conditions to guide actions, programs rely on conditional statements to determine what should happen under different circumstances. Without such logic, a program would always behave the same way, no matter what data it received.

Although the example is short, it plays an important role in many real-world applications. Grading systems, online quizzes, automated scoring tools, and data-processing programs all use similar conditionals to classify information. This simple pattern also forms the foundation for more complex structures, such as if-else chains and nested conditions, which allow programs to select from many possible actions instead of just one.

In Java, the if statement is one of the most essential tools for controlling the flow of a program. It allows developers to make decisions based on conditions and to execute certain actions only when those conditions are true. Although the syntax of if statements appears simple, the way it is written can dramatically affect the reliability, safety, and long-term maintainability of the code. The examples 2.2 below demonstrate this concept clearly:

Example 2.2: Comparison with and without braces in if Statement

```
// Bad: missing braces can cause errors when adding lines later
if (x < 0)
    System.out.println("negative");

// Good: always use braces
if (x < 0) {
    System.out.println("negative");
}
```

At first glance, both versions appear to do exactly the same thing. In each case, the program checks whether the variable `x` is less than zero, and if so, prints the word "negative." Java permits the shorter form without braces when there is only a single statement inside the if. Beginners often prefer this style because it saves a few characters and looks clean. However, while the compact version seems harmless, it introduces subtle risks that can lead to errors and misunderstandings when the code is modified later.

The main problem with omitting braces is that it makes the code fragile. Suppose another programmer (or even the same programmer months later) decides to add another line inside the if block, such as logging, updating a counter, or performing additional checks. Without braces, the new line may unintentionally fall outside the intended block:

```

if (x < 0)
    System.out.println("negative");
    logError(x); // This line is NOT part of the if-block!

```

Because only the first statement belongs to the if, the new line executes regardless of whether x is negative. This kind of bug is notoriously difficult to spot because the indentation may look correct, even though the logic is not. Entire categories of software errors, and even real-world system failures, have been traced back to missing braces around single-line if statements.

In contrast, the version that uses braces is safer, clearer, and more professional:

```

if (x < 0) {
    System.out.println("negative");
}

```

Even though it takes up a few more characters, the braces remove all ambiguity. Anyone reading the code, whether a student learning Java or an experienced engineer reviewing the code, can immediately see which statements belong to the conditional block. Adding new statements becomes trivial and risk-free. This clarity significantly improves maintainability, especially in large projects where code undergoes frequent revisions.

Moreover, always using braces aligns with widely accepted industry standards and style guides, including those from Google, Oracle, and numerous open-source communities. Consistency in formatting is not just a matter of personal preference; it prevents subtle errors, improves team collaboration, and makes code easier to read. Professional developers understand that clarity and correctness are far more important than shaving off a few characters.

2.2.2 If-Else statement

If-Else Flow

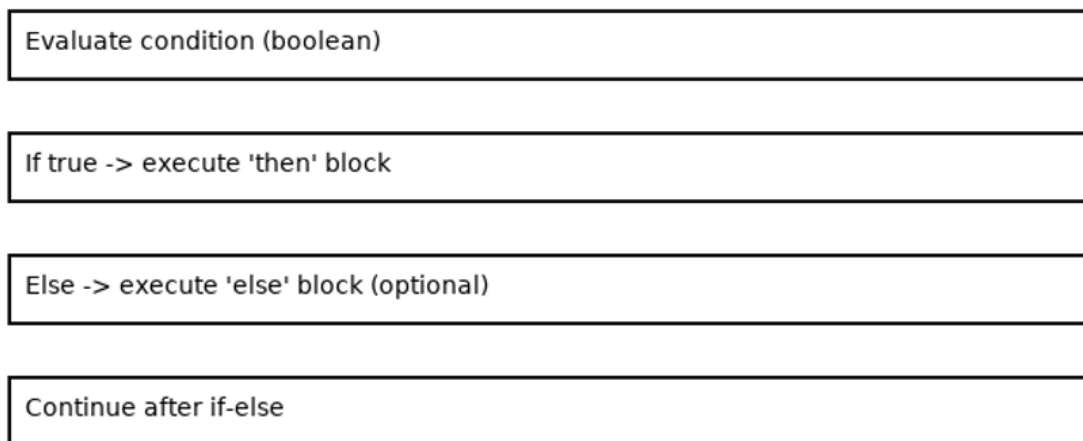


Figure 2.1: If-Else flow (evaluates condition and branches accordingly).

The following code segment in example 2.3 demonstrates how Java handles decisions that depend on more than one condition:

Example 2.3: A Simple If-Else statement

```

if (age >= 18 && hasID) {
    System.out.println("Entry allowed.");
} else {
    System.out.println("Entry denied.");
}

```

At first glance, this appears to be a simple decision-making structure. However, it illustrates an important programming concept: combining multiple logical conditions using the logical AND operator (&&) to control the flow of a program. This example also shows how programmers can express real-world rules precisely and clearly in code.

The goal of this snippet is to determine whether a person is allowed to enter a restricted area, perhaps a secure building. In everyday life, such places often require two things: the person must be at least a certain age, and they must present valid identification. Java captures this rule directly in the condition `age >= 18 && hasID`.

The expression `age >= 18` is the first condition. It checks whether the individual meets the required age threshold. The second condition, `hasID`, is a boolean variable that represents whether the person possesses a valid identification card. Both conditions must be true for entry to be permitted, which is why they are combined with the logical AND operator. In Java, `A && B` evaluates to true only if both A and B are true.

This structure mirrors real-world reasoning:

- A person who is old enough but does not have ID cannot be admitted, and
- A person who has ID but is not old enough cannot be admitted either.

The program enforces these rules consistently and without ambiguity.

The if statement uses this combined condition to determine which block of code to execute. If both requirements are satisfied, the program prints “Entry allowed.” If either requirement is not met, the combined condition becomes false, and the program executes the else block instead, printing “Entry denied.” This ensures that the program always produces a clear and appropriate result based on the inputs.

Another important detail is Java’s use of short-circuit evaluation. With the && operator, Java evaluates the left condition first. If the left condition is false, for example, if age is only 15, Java does not bother evaluating `hasID` because it already knows the entire expression must be false. This behavior improves efficiency and prevents unnecessary checks.

The simplicity of this example should not overshadow its importance. Many real-world decisions depend on multiple criteria: a bank might approve a loan only if a customer has sufficient credit and a stable income; a program might unlock advanced features if a user is both logged in and has premium status. The pattern shown here, combining conditions within an if statement, lies at the heart of much of the logic found in modern software systems.

2.2.3 If-Else-If statement

The code in example 2.4 is a classic example of how Java uses conditional logic to make multi-way decisions:

Example 2.4: Multi-way Decisions

```
if (score >= 90) {
    grade = 'A';
} else if (score >= 80) {
    grade = 'B';
} else if (score >= 70) {
    grade = 'C';
} else if (score >= 60) {
    grade = 'D';
} else {
    grade = 'F';
}
```

At first glance, this code simply maps a numerical score to a corresponding letter grade, a system familiar to almost every student. Yet beneath its simplicity lies an important programming concept: the if-

else-if ladder, a fundamental structure that allows a program to choose exactly one action from several possible alternatives based on a chain of conditions.

This structure works by evaluating each condition in order from top to bottom. The program first checks whether the score is at least 90. If this condition is true, the program assigns the grade 'A' and skips all remaining conditions. This behavior ensures that only one grade is assigned, even if multiple conditions could theoretically be true. If the score is less than 90, the program moves to the next condition and checks whether the score is at least 80. This process continues until one of the conditions is satisfied or until it reaches the final else block, which serves as a catch-all for any score that did not meet any of the earlier thresholds.

This design reflects how humans make multi-step decisions. For example, a school might say:

- If a student earns 90 or above, award an A.
- Else if they earn 80 or above, award a B.
- Else if they earn 70 or above, award a C.
- And so on.

Each condition narrows the decision further, just as in the code. This ordered structure is essential. If the conditions were rearranged or written incorrectly, the program could assign the wrong grade. For instance, checking `score >= 60` before `score >= 90` would cause all scores above 60, including 95 or 100, to incorrectly receive a D. The code works correctly because the highest thresholds appear first and the lower ones follow in descending order.

The final else clause is also important. It handles all remaining cases in this context, any score below 60. Including such a default branch ensures that every possible input receives a valid output. This guards against uninitialized variables and makes the logic complete.

This grading example is more than a simple classroom exercise. Systems like these appear everywhere in software:

- determining tax brackets
- selecting membership levels
- calculating shipping costs based on weight
- categorizing performance scores in business or sports
- validating user input and classifying results

In each scenario, the program must choose exactly one outcome based on a set of ordered criteria. The if-else-if structure is ideal for such tasks because it is readable, intuitive, and guarantees that only one branch executes. Example 2.5 is another If-Else-If chain.

Example 2.5: An If-Else-If chain

```
If-Else-If chain:
if (points >= 1000) {
    level = "Platinum";
} else if (points >= 500) {
    level = "Gold";
} else {
    level = "Standard";
}
```

The following code illustrates a common pattern that new programmers sometimes write when checking multiple conditions:

Example 2.6: An Example of Deep Nesting

```
if (order != null) {
    if (order.isPaid()) {
        if (!order.isCancelled()) {
            ship(order);
        }
    }
}
```

At a surface level, the logic is straightforward:

- If the order exists,
- and if it has been paid for,
- and if it has not been canceled,
- then ship the order.

Although this sequence of checks is logically correct, the structure of the code reveals an important programming problem: deep nesting. Each condition is placed inside another, forming a staircase of indented blocks. While the logic remains functional, this style quickly becomes difficult to read, understand, and maintain, especially as programs grow in complexity.

Deep nesting harms code quality in several ways. First, it makes the essential action, the call to `ship(order)` harder to see. Developers must visually “dig” through layers of indentation to find the actual purpose of the code. When reading software, clarity is crucial. Code is read far more often than it is written, and unnecessary nesting obscures the intent.

Second, deeply nested conditionals are more error-prone. If a programmer later adds another condition or modifies an existing one, it becomes easy to misplace braces or misunderstand which block a piece of logic belongs to. Even small indentation mistakes can lead to logic errors that are difficult to detect. As nested structures deepen, the cognitive load increases, and bugs become harder to avoid.

Finally, nested blocks are a sign that the structure could be simplified. A common best practice in modern programming is to use guard clauses or early returns. Instead of placing code inside progressively deeper layers, we handle invalid or undesirable cases first and return (or exit the block) immediately. This flattens the structure and highlights the core logic. For example:

```
if (order == null) return;
if (!order.isPaid()) return;
if (order.isCancelled()) return;
ship(order);
```

Now the core operation shipping the order appears clearly at the bottom, and each condition is treated independently. This version is both easier to read and easier to maintain. It also reduces the risk of logical mistakes because each case is handled explicitly and cleanly.

Deep nesting also makes scaling the logic more difficult. Suppose new business rules require additional checks: whether the payment has cleared, whether the shipping address is valid, or whether the customer is verified. Adding these conditions to the nested version would cause the code to balloon sideways with even more indentation. In contrast, the flattened approach remains simple and clear regardless of how many conditions are added.

2.3 switch and switch expressions

`switch` is useful for branching on discrete values. Since Java 12+, switch expressions were introduced to make switch more concise and less error-prone. Always prefer switch when branching on enums, integers, or strings with many alternatives.

Traditional switch with fall-through:

In Java programming, making decisions based on different inputs is a key part of building interactive and responsive software. One common way to achieve this is through the switch statement, which allows a program to choose between multiple possible actions depending on the value of a variable. The following code in example 2.7 demonstrates how a switch statement can be used to execute different commands:

Example 2.7: An Example of Switch Statement

```
switch (command) {
    case "start":
        startService();
        break;
    case "stop":
        stopService();
        break;
    default:
        throw new IllegalArgumentException("Unknown command");
}
```

This short program captures a powerful control-flow structure. It checks the value of a variable called `command`, and depending on its value, the program performs one of several actions. Understanding this example provides insight into how Java handles multi-branch decisions.

The switch begins by comparing the value of `command` to the available cases. If the value is "start", the program executes the `startService()` method. This method name suggests that it activates or initiates a service or process. After calling the method, the `break` statement ensures the program exits the switch block immediately, preventing it from accidentally running the next case.

If the value of `command` is "stop", the program instead calls the `stopService()` method. This likely shuts down or deactivates the service. Again, a `break` statement ends the case to avoid fall-through behavior.

When the value of `command` does not match any of the defined cases, Java executes the default block. In this example, the default behavior is to throw an `IllegalArgumentException` with the message "Unknown command". Including a default case is considered good programming practice because it ensures that invalid or unexpected input does not go unnoticed. Rather than silently ignoring an error, the program clearly reports the problem, preventing unpredictable behavior.

This switch structure is both clear and practical. It organizes actions in a readable way, making the code easier to understand and maintain. It is also easy to extend because additional commands can be added simply by defining more cases. Meanwhile, the default section guarantees that improper input is handled safely and explicitly, improving the program's robustness.

As Java continues to evolve, new language features are introduced to make programs more expressive, readable, and concise. One such improvement is the modern switch expression, introduced as a preview feature in earlier versions and standardized in Java 14. Unlike the traditional switch statement, the new switch expression allows developers to produce a value directly from a switch block, reducing boilerplate and eliminating common errors. The following code in example 2.8 demonstrates this modern syntax:

Example 2.8: The New Switch Expression

```
String result = switch (code) {
    case 1 -> "one";
    case 2 -> "two";
```

```
    default -> "other";
};
```

This short snippet highlights several important enhancements in the newer switch expression.

First, the switch is used as an expression, meaning it evaluates to a value that can be assigned directly to a variable—in this case, the variable `result`. This is different from older switch statements, which could only perform actions but could not themselves produce a value without additional variables and repeated break statements.

Second, the use of the arrow syntax (`->`) makes the code more compact and avoids accidental “fall-through,” a common issue in traditional switch statements where forgetting a break could cause multiple cases to execute unintentionally. Here, each case cleanly maps a specific input to a specific output:

- If code is 1, the switch expression yields the string "one".
- If code is 2, it yields "two".
- For any other value of code, the default branch produces "other".

The arrow syntax clearly indicates that each case leads directly to a result, improving readability and reducing the chance of logic errors.

Another benefit of the modern switch expression is that it encourages developers to think in terms of mapping values to outputs rather than writing long conditional blocks. This approach aligns well with functional programming concepts, where expressions return results rather than relying on side effects.

Additionally, because the switch is now an expression, the compiler enforces exhaustiveness, meaning all possible paths must produce a value. This leads to safer, more predictable code. In this example, the presence of the default branch ensures that no matter what value code holds, the switch expression will return a valid string. Also in the example, when using enums for domain-specific choices, switch on enums is compile-time safe and clearer.

2.4 Ternary operator and guard clauses

2.4.1 Using the Ternary Operator and Guard Clauses to Improve Code Clarity

Writing clear and maintainable code is one of the most important skills a programmer can develop. Two common techniques that contribute to clean code are the ternary operator and the use of guard clauses—also known as early returns. When used appropriately, these tools help reduce unnecessary complexity and improve the readability of a program.

2.4.2 The Ternary Operator: A Compact Conditional Expression

The ternary operator `?:` is a shorthand way to choose between two values based on a boolean condition. It has the following general form:

```
condition ? valueIfTrue : valueIfFalse
```

This operator evaluates the condition and returns one of the two values accordingly. For example:

```
int abs = (x >= 0) ? x : -x;
```

In this line of code, the program computes the absolute value of `x`. If `x` is greater than or equal to zero, the expression returns `x`; otherwise, it returns `-x`. The ternary operator allows this logic to be expressed in a single, compact line instead of writing a multi-line if-else statement.

Although the ternary operator can make code shorter, it should be used sparingly. It works best when the condition and both return values are simple and easy to understand at a glance. Overusing it, or using it with complex logic, can make code more confusing than helpful. A good rule of thumb is that if someone must pause to decipher the expression, an if-else block would likely be clearer.

2.5 For loops (classic and enhanced)

2.5.1 Understanding Classic for Loops and Enhanced for-each Loops in Java

Looping is a fundamental part of programming, allowing developers to process data repeatedly and efficiently. Java provides multiple looping constructs, each designed to address different iteration needs. Two of the most common forms are the classic for loop and the enhanced for loop, also known as the for-each loop. Although both allow you to iterate over data, they differ significantly in structure, capabilities, readability, and intended use. Understanding how each loop works, and when to use it, will help you write clear, efficient, and professional-quality code.

2.5.2 The Classic for Loop: Full Control Over the Iteration Process

The classic for loop follows this general form:

```
for (init; condition; update) {
    body
}
```

This structure gives the programmer complete control over the flow of the loop:

- Initialization (init) runs once before the loop starts.
- Condition (condition) is checked before each iteration; when it becomes false, the loop stops.
- Update (update) executes after each iteration of the body, usually modifying the loop counter.

For Loop Flow

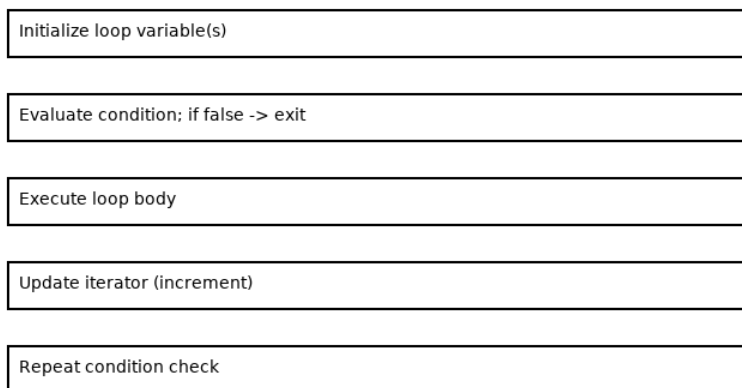


Figure 2.2: For-loop flow - initialization, condition, body, update.

Because of its flexibility, the classic for loop is highly versatile. It is ideal when you need:

- Index-based access, such as retrieving elements from an array by their position.
- Complex iteration patterns, like skipping elements, counting backward, or iterating with a custom step size.
- Simultaneous tracking of multiple variables, for example incrementing one counter while decrementing another.

A typical example showing index-based access is:

```
for (int i = 0; i < arr.length; i++) {
```

```

    System.out.println(i + ": " + arr[i]);
}

```

This loop prints both the index and the corresponding array element. The ability to access the index directly is something the enhanced for loop does not provide. Thus, whenever indexing is important, the classic for loop is the natural choice.

Additionally, the classic loop is helpful when you need more advanced logic, such as:

- Incrementing by more than 1
- Iterating from the end of the array to the beginning
- Using multiple loop variables, e.g., for (int i = 0, j = n-1; i < j; i++, j--)

Its flexibility is the reason it continues to be widely used even as newer loop constructs have been introduced.

2.5.3 The Enhanced for Loop: Cleaner, Safer Iteration for Collections and Arrays

Introduced in Java 5, the enhanced for loop was designed to simplify iteration when only the values matter. It follows a much cleaner syntax:

```

for (Type item : collection) {
    body
}

```

This loop automatically retrieves each element from the collection or array, allowing the programmer to focus solely on the logic applied to each element.

Example 2.9: Enhanced For Loop

```

List<String> names = List.of("Alice", "Bob", "Carol");
for (String name : names) {
    System.out.println(name);
}

```

In example 2.9, the approach eliminates the need for manual index management, making the code more readable and less prone to errors such as off-by-one mistakes.

Since the enhanced for loop directly exposes each element, it is particularly useful for:

- Iterating through lists, sets, arrays, or any class that implements Iterable
- Writing clean, simple loops where the index is irrelevant
- Avoiding clutter from initialization and update statements

The enhanced for loop is often preferred because it reduces boilerplate code and emphasizes the programmer's intent.

Although the enhanced for loop offers convenience, it has limitations that developers must understand.

Because the loop retrieves elements directly, it does not expose their position in the collection. If you need to know which iteration you are on or if you must access elements by index, the classic for loop is more appropriate.

The enhanced for loop does not allow structural modifications such as adding or removing elements while looping. Attempting to do so will result in a `ConcurrentModificationException`. This is because the underlying iterator used by the enhanced for loop is not designed to track changes in the collection's structure.

If you need to remove elements safely, you must use an explicit Iterator:

Example 2.10: Element Removal in Enhanced For Loop

```

Iterator<String> it = names.iterator();

```

```

while (it.hasNext()) {
    String name = it.next();
    if (name.startsWith("A")) {
        it.remove(); // Safe removal
    }
}

```

The iterator provides controlled, safe modification, whereas the enhanced for loop enforces a read-only structure during iteration.

2.6 while and do-while loops

In programming, loops allow us to repeat actions until certain conditions are met. While the classic for loop is useful when the number of iterations is known, Java's while and do-while loops are designed for situations where repetition depends on changing conditions rather than fixed counts. These loops are essential tools for building interactive programs, processing input, and managing tasks that continue until a particular event occurs.

2.6.1 The while Loop: Repeating as Long as a Condition Is True

The while loop checks its condition before running the body of the loop. Its structure is:

```

while (condition) {
    body
}

```

This loop is ideal when the number of iterations is unknown and depends on some ongoing condition, such as reading input until no more is available, waiting for a sensor, or continuing until a user chooses to exit.

Example 2.11: Reading Input Until No More Lines Exist

```

while (scanner.hasNextLine()) {
    String line = scanner.nextLine();
    // process line
}

```

In the example 2.11, the loop continues as long as the scanner detects another available line. This pattern is common when reading from files, network streams, or user input. Since the condition is evaluated before each iteration, the loop might not run at all if no input is available from the start.

Use while when:

- You do not know how many times the loop will run.
- The loop depends entirely on a changing condition.
- You may want the body to run zero times if the condition is false initially.

2.6.2 The do-while Loop: Run First, Then Check

The do-while loop guarantees that the body of the loop runs at least once. Its structure is:

```

do {
    body
} while (condition);

```

Because the condition is checked *after* running the body, this loop is ideal for interactive programs where the user must see a menu or prompt before a condition is evaluated.

Example 2.12: Showing a Menu Until the User Chooses to Exit

```

int choice;
do {

```

```

showMenu();
choice = scanner.nextInt();
} while (choice != 0);

```

In example 2.12, the menu is displayed once, and then repeatedly as long as the user does not enter 0. Even if the user decides to exit immediately, the menu still appears at least once.

Use do-while when:

- You need the loop to run **at least once**.
- You are implementing menu-driven programs.
- You want to validate user input by looping until it satisfies a condition.

While / Do-While Flow

While: Check condition first; if true run body

Do-While: Run body, then check condition

Repeat until condition false

Figure 2.3: While vs Do-While behavior.

2.7 Nested loops and computational complexity

Nested loops multiply iteration counts. In example 2.13, two nested loops include outer loop and inner loop, making the complexity to $O(N^2)$ time. When using nested loops, please always analyze worst-case complexity and consider algorithmic improvements.

Example 2.13: Nested Loops

```

// naive pairwise comparison ( $O(n^2)$ )
for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        // compare items i and j
    }
}

```

2.8 Loop control: break, continue, and labeled breaks

`break` exits the innermost loop. `continue` skips to next iteration. As show in example 2.14, you can use labeled breaks to exit outer loops sparingly.

Example 2.14: Loop Control

```

// find first null in 2D array
int rowFound = -1;
outer:
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {

```

```

    if (matrix[i][j] == null) {
        rowFound = i;
        break outer; // labeled break exits both loops
    }
}
}

```

Self-Review Questions

1. What is the purpose of a boolean expression in Java?
2. Why is it recommended to always use braces {} with if statements?
3. What is an if-else-if chain used for?
4. What problem does deep nesting cause in code?
5. How does Java's short-circuit evaluation work with &&?
6. What is the benefit of modern switch expressions (-> syntax)?
7. When should the ternary operator ?: be used?
8. Why can't the enhanced for-each loop modify the structure of a collection?
9. When should a while loop be used instead of a for loop?
10. What is the purpose of a labeled break?

Programming projects:

1. Write an if-else-if chain to assign grades: A (90+), B (80+), C (70+), D (60+), F.
2. Rewrite the following deeply nested code using guard clauses:

Original:

```

if (order != null) {
    if (order.isPaid()) {
        if (!order.isCancelled()) {
            ship(order);
        }
    }
}

```

3. Write a switch expression that converts an integer day code to a string. Codes: 1=Mon, 2=Tue, otherwise "Other".
4. Write a ternary expression to compute the absolute value of x.
5. Write a classic for loop that prints all values of an array with indices.
6. Write an enhanced for loop to print all names in a list.
7. Write a while loop that reads lines until no more input exists.
8. Write a do-while loop showing a menu until the user enters 0.
9. Write nested loops to print all pairs (i, j) where both range from 0 to 2.
10. Use an iterator to safely remove names that start with "A" from a list.

Chapter 3: Writing Classes

Chapter Objectives:

- Identifying classes and objects.
- Structure and content of classes.
- What is Instance data?
- Access (Visibility) Modifier
- Accessor (Getter) and Mutator (Setter) methods
- Using UML Diagram to Demonstrate the Relationships among classes
- Importance of Encapsulation
- Static methods and data.
- Structure of class methods.
- What is a Constructor?
- The Purpose of a constructor in a class.
- The this reference

The foundation of programming using an object-oriented program like Java is classes and objects. Therefore, any individual who aspires to creating programs using Java needs to grasp the idea of creating classes. When a user creates a unique class in Java this is referred to as creating a user-defined class. This chapter will explore how to define a unique class, which includes its structure and content. It also examines the importance of visibility modifiers in the creation of a user-defined class.

3.1 Review of Classes and Objects

Before we dive into user-defined classes, we need to completely grasp the idea of an object and a class. An object has a “state” with established “behaviors”. State of an object refers to the value stored in the instance variables, attributes, or data fields at a certain time. This might be primitive data or other objects. For example, if you have a class named `LightBulb` that represents a light bulb, and actions associated with it. The state of a light bulb may be represented by a boolean instance variable that stores the state of the light bulb, whether it is ON (true) or OFF (false):

```
private boolean lightbulb_state;
```

The behavior of an object refers to the methods or operations it uses to carry out activities. For example, the behavior of a light bulb object is to turn ON or turn OFF:

```
public void turnON(){
    lightbulb_state = true;
}
public void turnOFF(){
    lightbulb_state = false;
}
```

A class is the blueprint used to create an object. Multiple objects can be created using the same class. Where each object can contain its own state or data. For example, you can define multiple light bulb objects from the `LightBulb` class. Each object will keep track of its state: whether it is ON or OFF. Note: a class is not an object.

There are several pre-defined classes available in the Java Application Programming Interface (API). Previous chapters have highlighted and used a few of these classes in the introductory java programs. An example, the String class. This class has several predefined methods such as `charAt()`, `compareTo()`, etc. There is also the Scanner class that is used to read in input from the user's keyboard or from a file. In this chapter you will learn how to create your own user-defined class.

Identifying Classes and Objects

A software engineer will be required to write a program that is a solution to a problem statement. Based on this statement the programmer is required to decide which classes to create for the program. One way to identify potential classes is to highlight the nouns in the problem statement.

Create a **class** that represents a **light bulb** object. The **state** of the **light bulb** is ON or OFF. The **user** should be able to turn ON, turn OFF, and check the status of the **light bulb**.

Identifying Nouns in a Problem Statement

If a class becomes too complex, then consider breaking it into multiple smaller classes. More specifically the user needs to decide what instance variables to create to represent the attributes of the class and methods to carry out activities. If the user decides to use multiple classes, then the user needs to decide what instance variables and methods to define in each class. It is recommended to write this information down before you start to code the program.

3.2 Structure and Contents of a Class

In all your previous java projects you created programs that contain the main method. The user-defined classes covered in this chapter will not contain the main method. But it will contain instance variable(s) and method(s) declaration(s). This is shown in the LightBulb user-defined class below:

```

/*****
// LightBulb.java      Java Foundations
//
// Demonstrates the anatomy of a user-defined class.
*****/

public class LightBulb
{
    private boolean lightbulb_state;    Data field (or instance
                                       variable) declaration
    //Constructor
    public LightBulb(){
        lightbulb_state = false;
    }

    //Method used to check the light bulb's state
    // "true" means it's ON and "false" it's OFF
    public boolean isON(){
        return lightbulb_state;
    }
}

```

```

    }

    //Method used to turn ON the lightbulb
    public void turnON()           Method
    {                             Declarations
        if(!isON()){
            lightbulb_state = true;
        }
    }
    //Method used to turn OFF the lightbulb
    public void turnOFF()
    {
        if(isON()){
            lightbulb_state = false;
        }
    }
}

```

The difference between the classes you have used up to this point and the `LightBulb` class is that the `LightBulb` class is not part of the Java API or predefined classes. The `LightBulb` class was created by the user and is considered a user-defined class. A general programming convention in Java is to declare the instance variables at the top of the class followed by the methods declarations.

3.3 What is Instance Data?

The `lightbulb_state` and `name` are called instance data, because each object (or instance) of the class that is created has its own version of the data. The objects that belong to the same class will share the method declarations, but have their own data space to differentiate each object state.

```

//*****
// HouseLight.java      Java Foundations
//
// Demonstrates the implementation of a user-defined class in a // program. Using the LightBulb
// class.
//*****
import java.util.Scanner;
public class HouseLight{
    public static void main(String[] args){
        Scanner scan = new Scanner(System.in);
        LightBulb flashLight = new LightBulb();
        LightBulb refrigeratorLight = new LightBulb();
    }
}

```

```

System.out.println("Enter 0 to turn on the flashlight);
System.out.print("Enter 1 to turn on the refrigerator
                light: ");
int value = scan.nextInt();
if(value == 0){
    flashlight.turnON();
} else if(value == 1){
    refrigeratorLight.turnON();
}
}
}
}

```

3.4 Structure of Class Methods

The method is associated with the class not an object or instance of the class. The structure of the class method contains several keywords. Namely, visibility (access) modifier, static keyword, return type, the name of the method, parameters in parentheses, and in curly braces the body of the method and a return statement.

Syntax:

```

visibility modifier static keyword return type method name( parameters ){
    //method body
    //return statement
}

```

Access (Visibility) Modifier

These keywords determine access to the method. The access modifiers found in Java are public, private, and protected.

- public modifier – methods or instance variables with this modifier can be accessed from anywhere.
- private modifier – methods or instance variables with this modifier can only be accessed from inside the class.
- protected modifier – methods or instance variables with this modifier can be accessed by the class it is defined in and subclasses that belong to the class. This modifier is oftentimes implemented in inheritance relationships.

Accessor (Getter) and Mutator (Setter) methods

For an instance variable with the *private* access modifier, we need to create methods to access and modify them. The accessor (getter) and the mutator(setter) methods are used to access and modify the values stored in the variables with the *private* modifier.

The *accessor* method returns the value stored in the instance variable, while the *mutator* method changes or updates the value stored in the instance variable.

The naming convention for mutator and accessor methods is *set* followed by the variable name and *get* followed by the variable name respectively. For example, create the getter and setter

methods for a class named Square with an instance variable that represents the length of a square.

```
public class Square{
    private int length; //data field with private modifier

    //Accessor (getter) method definition
    public int getLength(){
        return length;
    }

    //Mutator (setter) method definition
    public void setLength(int len){
        length = len;
    }
}
```

The Importance of Encapsulation

For a user-defined class encapsulation refers to the grouping of data fields (or attributes) and methods into a single unit. A data field or method with encapsulation protects it from direct access or modification from outside the class. Encapsulation is implemented in a user-defined class with attributes (or data fields) and methods using one of the three (3) access (or visibility) modifiers discussed above.

Using Unified Modeling Language (UML) Diagram to Demonstrate the Relationships among classes

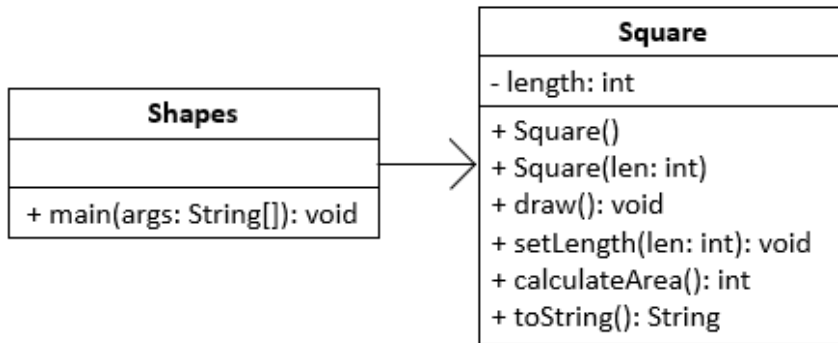
This diagram shows the relationships among classes and objects. An UML diagram consists of one or more classes. The box representing a class contains three rows:

1. Top row contains the name of the class or interface.
2. Middle row contains data fields descriptions.
3. Bottom row contains methods and constructors' descriptions.

Symbols are used in the UML diagram to represent the access modifier for the data fields, constructors, and methods. These are:

1. + represents the public modifier.
2. - represents the private modifier.
3. # represents the protected modifier.

The diagram also consists of lines that are used to describe the association between the classes. For example, a solid line means that there are statements in one class that call the method of another class. An example of an UML diagram is shown below.



Static Methods and Data

Static members belong to the class, not an instance (or object) of the class. A static method can only reference a static or a variable declared in the method (or local variable). The main method is a static method. You do not need to create an object of the class to invoke the main method as it is invoked by the Java interpreter. For all user-defined or predefined classes with static methods can be invoked using the name of the class and the dot (.) operator.

```

public class Square{
    private static int length;
    public static int calculateArea(){
        return length*length;
    }
    //Draws the shape of the square using *
    public static void draw() {
        for (int i = 0; i < length; i++) {
            if (i == 0 || i == (length - 1)) {
                for (int j = 0; j < length; j++)
                    System.out.print("* ");
                System.out.println();
            } else {
                for (int k = 0; k < length; k++) {
                    if (k == 0 || k == (length - 1)) {
                        System.out.print("* ");
                    }
                    else {
                        System.out.print(" ");
                    }
                }
                System.out.println();
            }
        }
    }
}

```

You invoke the calculateArea method with the following statement:
`int area = Square.calculateArea();`

Return Type

Determines the data type or object to be returned after the execution of the method. A return statement with the keyword return is added to the method body. The void keyword is used if the method does not return a value.

```
public class Square{
    private int length;
    public int calculateArea(){
        return length*length;
    }
    public void setLength(int len){
        length = len;
    }
}
```

Name of the Method

The common naming convention for a method is to use a verb. The first letter in the method name is lowercase. For example, the draw() method declared in the Square class above. The name of the method can also contain multiple words, where the first letter of the internal word is an uppercase letter. For example, the calculateArea() method defined in the Square class above.

Parameters in Parentheses

The parameter in the parentheses “()” represents the input passed to the method when it is invoked or called. These parameters are oftentimes referred to as formal parameters. There can be no parameter, one parameter, or multiple parameters in the method header definition. Each parameter consists of the data type and the name of the parameter.

Example#1: the calculateArea() method that does not have a parameter:

```
public int calculateArea(){
    return length*length;
}
```

Example#2: the setLength() method has a parameter of type integer (or int) and the name is len.

```
public void setLength(int len){
    length = len;
}
```

The Method Body and Return Statement in Curly Braces

This section within the curly braces contains statements that define what the method does. The statement might be variable declarations. Note: any variable declared within the method body is local to the method. The statement could be if-statement, for-loop, while-loop, etc.

One of the statements within the method body is a return statement. The return statement passes a value to the method caller. When the return statement is executed, this signals the end of the method execution. The return statement can be placed anywhere within the method body.

Syntax:

return; (used for void methods)

return value; (where the data type of the variable value matches the return type listed in the method's header)

Example#1: the calculateArea method's return type is int. Therefore, the value return is an integer.

```
public int calculateArea(){
    int product = length*length;
    return product;
}
```

The toString() method

The toString method is inherited from the predefined Object class. Inheritance will be covered in later chapters. Every user defined or predefined class is a descendant of the Object class. The toString method returns a string description of an object. Example, toString method definition in the Square class:

```
public String toString(){
    return "The Length is" +length;
}
```

There are two ways to invoke the toString method:

1. Explicit invocation: use the reference variable for the object along with the dot operator. For example, square1 is a reference variable for a Square object. The toString method can be called using the statement: square1.toString();
2. Implicit invocation: place the reference variable for the object in a java print statement. For example, square1 is a reference variable for a Square object. The toString method can be called using the following statement: System.out.println(square1);

What is a Constructor?

This is called when you are creating an object. Its main function is to initialize the data fields for objects of the class.

Main characteristics of a constructor:

1. It has the same name as the class in which it is defined.
2. It does not have a return statement.

There are three (3) types of constructors:

1. Default constructor – this is provided by Java as a default, if the user did not explicitly define a constructor.
2. Constructor without parameter(s) – the user can define a constructor without parameters. Inside the constructor the user will write statements that initialize the data fields.
3. Parameterized constructors – the user defines the constructor with parameters that are used to initialize the data fields.

```
public class Square{
    private int length; //data field variable
```

```

//Constructor without Parameter(s)
public Square(){
    length = 10;
}
//Parameterized Constructor
public Square(int len){
    length = len;
}
//Draw the shape of the square using * symbol
public void draw() {
    for (int i = 0; i < length; i++) {
        if (i == 0 || i == (length - 1)) {
            for (int j = 0; j < length; j++)
                System.out.print("* ");
            System.out.println();
        } else {
            for (int k = 0; k < length; k++) {
                if (k == 0 || k == (length - 1)) {
                    System.out.print("* ");
                }
                else {
                    System.out.print(" ");
                }
            }
            System.out.println();
        }
    }
}

//setter method
public void setLength(int len){
    length = len;
}

//calculate the area of the square
public int calculateArea(){
    int product = length*length;
    return product;
}

//the toString method
public String toString(){
    return "The Length is" +length;
}
}

```

The Purpose of a constructor in a class:

In java the purpose of a constructor is to initialize an instance or object of a class. An object is created by using the new keyword to instantiate the object and invoking the constructor to initialize the object.

The this reference:

The this reference in Java allows an object to refer to itself. The this reference is normally used inside a method and represents the specific object for the method using it. One of the most common way the this reference is used in Java programming is to differentiate the data field variable with the same name or identifier as a parameter. The constructor for the Square constructor with a parameter could be written as:

```
public class Square{
    private int length; //data field variable
    //Constructor without Parameter(s)
    public Square(){
        length = 10;
    }
    //Parameterized Constructor
    public Square(int length){
        this.length = length; //use the this reference
    }
}
```

Chapter 4: Graphical User Interface

Chapter Objectives:

- The JavaFX framework
- GUI controls, events, and event handlers
- Buttons, text fields, check boxes
- Layout panes
- Color pickers and date pickers
- Mouse and keyboard events
- Dialog boxes
- Sliders

The JavaFX Framework

The programs that you have seen so far in this book are command line applications. In this chapter we will look at creating programs that use graphics and the Graphical User Interface (GUI). We will use predefined classes in the Java API to create these programs. JavaFX class replaced older approaches such as AWT and Swing to create graphical programs in Java.

Characteristics of the JavaFX class:

1. It extends the Application class. Thus, inheriting several graphical functionalities. You will cover inheritance in detail in later chapters.
2. The JavaFX program has a start method defined. This start method accepts the window used by the program as a parameter (add more information).

First JavaFX program:

```

/*****
// FirstJavaFXProgram.java      Java Foundations
//
// Demonstrates a basic JavaFX application.
*****/
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.text.Text;
import javafx.stage.Stage;
public class HelloJavaFX extends Application
{
    //-----
    // Creates and displays two Text objects in a JavaFX window.
    //-----
    public void start(Stage primaryStage)
    {
        Text hello = new Text(5, 20, "My First JavaFX Program!");
        Text greeting = new Text(5, 40, "Welcome to CSCI 1302!");
        Group root = new Group(hello, greeting);
        Scene scene = new Scene(root, 300, 120, Color.GREEN);
        primaryStage.setTitle("A JavaFX Program");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

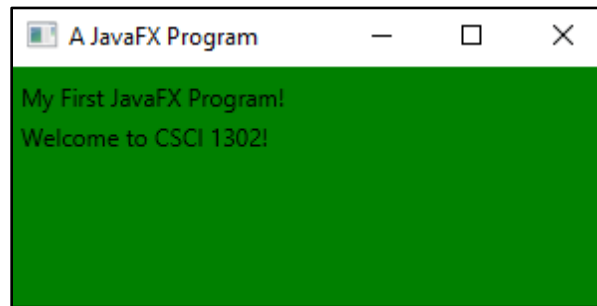
```

```

}

//-----
// Launches the JavaFX application. This method is not required
// in IDEs that launch JavaFX applications automatically.
//-----
public static void main(String[] args)
{
    launch(args);
}
}

```



Screenshot: HelloJavaFX Program

Description of the first JavaFX program:

1. Create two Text objects that are used to define nodes used to represent texts. The Text constructor allows you to set the position. This includes the distance from the left margin (i.e. X) and the distance from the top margin (i.e. Y). The actual text is the third argument that is placed within double quotes. Constructor: Text(X, Y, actual text);
2. The Group object that is created groups or acts as a container for the number of nodes to be included in the display. Therefore, the two Text objects created in part one (1) above are the arguments or nodes passed to the Group's constructor when the object is created. The Group object is the root node of a Scene graph.
3. A Scene object is created to contain all the contents to be displayed in a scene graph. Therefore, the arguments passed to the constructor are the Group object, size based on length and width, and the color of the background is set to green.
4. Set the Title of the dialog box by using the Stage parameter to call the setTitle() method.
5. Add the Scene object to the Stage object and then call the show() method to view the results of running this program.

Graphical User Interface (GUI) controls, events, and event handlers

The creation of any GUI in Java requires at least three (3) types of objects:

- a. **Controls:** control is an element that displays information or allows the user to interact with the JavaFX program. Common controls available in JavaFX can be grouped into label, input, selection controls, etc. An example of labeled control is Button. An example of input control is a TextField. An example of selection control is ChoiceBox.

- b. **Events:** event represents some activity or change in the state of an object or a GUI component. Control objects are used to generate an event. Examples of events are pressing a graphical button, pressing the keyboard, and clicking the mouse button.
- c. **Event Handlers:** this is the response of your program to a user event such as moving the mouse or typing on the keyboard. The event handler is activated when an event occurs.

Buttons, Text Fields, Sliders, Choice Boxes:

A button in the JavaFX class is implemented using the Button class. By pressing the button, the user generates an action event.

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.text.Text;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;
/*****
// NameAgeDisplay.java      Java Foundations
//
// Demonstrates JavaFX buttons and event handlers.
*****/
public class NameAgeDisplay extends Application {

    private Text nameText, ageText;

    @Override
    //-----
    // Presents a GUI containing Three buttons. One button display
    // the name, the 2nd button displays the age, and the third button
    // hides the name and age when pushed.
    //-----

    public void start(Stage primaryStage) {

        nameText = new Text("Name: -----");
        Button showName = new Button("Show Name!");
        showName.setOnAction(this::showNameButtonPress);

        ageText = new Text("Age: ----");
        Button showAge = new Button("Show Age!");
        showAge.setOnAction(this::showAgeButtonPress);

        Button hide = new Button("Hide!");
        hide.setOnAction(this::hideButtonPress);

        FlowPane pane = new FlowPane(showName, showAge, hide,
            nameText, ageText);
        pane.setAlignment(Pos.CENTER);
        pane.setHgap(20);
```

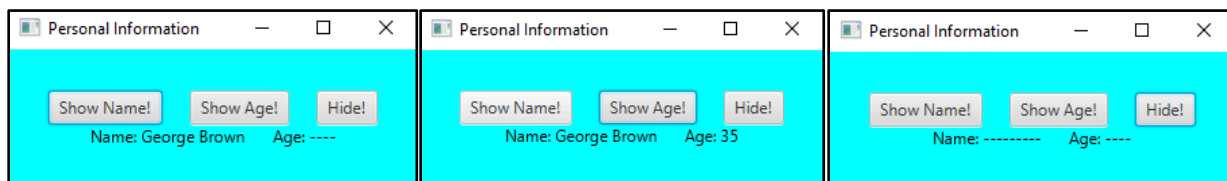
```

pane.setStyle("-fx-background-color: cyan");

Scene scene = new Scene(pane, 300, 100);
primaryStage.setTitle("Personal Information");
primaryStage.setScene(scene);
primaryStage.show();

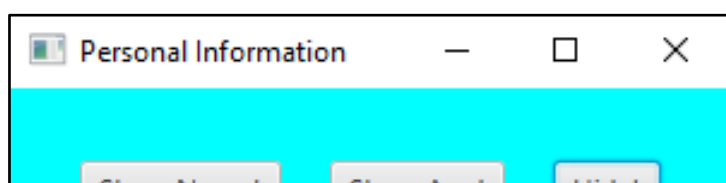
}
//-----
// Displays the name text when the button is pushed.
//-----
public void showNameButtonPress(ActionEvent event) {
    nameText.setText("Name: George Brown");
}
//-----
// Displays the age text when the button is pushed.
//-----
public void showAgeButtonPress(ActionEvent event) {
    ageText.setText("Age: 35");
}
//-----
// Hides the name and age text when the button is pushed.
//-----
public void hideButtonPress(ActionEvent event) {
    nameText.setText("Name: -----");
    ageText.setText("Age: ----");
}
}

```



Overview of the NameAgeDisplay JavaFX program:

- Relationship between the button that generates the event and event handling:
To set up this relationship a call to the `setOnAction` method is made. Example, `push.setOnAction(this::hideButtonPress);` statement from the example above.
- The `::` operator and this reference:
The statement `push.setOnAction(this::hideButtonPress);` that includes `::` is a method reference that is used to specify the event handler method. The `this` reference indicates the event handler method is in the same class.
- The `NameAgeDisplay` class represents the event handler for this application. The `hideButtonPress()` method carries out the event handling after the button is pressed. This method must contain an `ActionEvent` object as a parameter.
- A `FlowPane` object is used as the root node for the scene. A `FlowPane` is a layout pane, that displays contents horizontally in rows and vertically in columns.



Alternate Event Handler:

Instead of using a method (i.e. `hideButtonPress`) the event handling could be carried out by a separate class that implements the `EventHandler` interface. This can be implemented in the same class with a private modifier or public in its own file. For example,

```
public class ButtonHandler implements EventHandler<ActionEvent>
{
    public void hideButtonPress(ActionEvent event)
    {
        nameText.setText("Name: -----");
        ageText.setText("Age: ----");
    }
}
```

Determining Event Sources:

One event handler object could be used to listen to multiple controls. An example is to control the value displayed using two buttons. The first button increments the counter, while the second button decrements the counter.

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.text.Text;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;
/*****
// AddSubCounter.java      Java Foundations
//
// Demonstrates JavaFX buttons and event handlers.
*****/
public class AddSubCounter extends Application
{
    private int count;
    private Text countText;

    // -----
    // Presents a GUI containing a button and text that displays
    // increments or decrements the value based on the button that is
    // pushed.
    // -----
    public void start(Stage primaryStage) {
        count = 0;
        countText = new Text("Count: 0");
        Button incrementButton = new Button("Add");
        incrementButton.setOnAction(this::incrementButtonPress);

        Button decrementButton = new Button("Sub");
```

```

        decrementButton.setOnAction(this::decrementButtonPress);

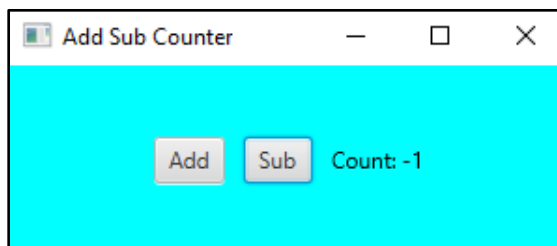
        FlowPane pane = new FlowPane(incrementButton,
            decrementButton, countText);
        pane.setAlignment(Pos.CENTER);
        pane.setHgap(10);
        pane.setStyle("-fx-background-color: cyan");
        Scene scene = new Scene(pane, 300, 100);

        primaryStage.setTitle("Add Sub Counter");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    // -----
    // Increments the counter when the Add button is pushed.
    // -----
    public void incrementButtonPress(ActionEvent event) {
        count++;
        countText.setText("Count: " + count);
    }

    // -----
    // Decrements the counter when the Sub button is pushed.
    // -----
    public void decrementButtonPress(ActionEvent event) {
        count--;
        countText.setText("Count: " + count);
    }
}

```



Result: after clicking the Add button twice then clicking the Sub button three times.

Text Fields:

A text field using JavaFX allows the user to enter one line as the input. When reading in an input from the keyboard, the user signals the end of the input by pressing the Enter key on the keyboard. In the same way the TextField objects generate an action event when the user presses the Enter button on the keyboard.

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

```

```

//*****
// NameDisplay.java    Java Foundations

```

```
//
// Demonstrates the use of a TextField and a GridPane.
//*****
public class NameDisplay extends Application
{
    //-----
    // Launches the name display application.
    //-----
    public void start(Stage primaryStage)
    {
        Scene scene = new Scene(new Name(), 300, 150);

        primaryStage.setTitle("Display Greeting");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

import javafx.event.ActionEvent;
import javafx.geometry.HPos;
import javafx.geometry.Pos;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Font;

//*****
// Name.java      Java Foundations
//
// Demonstrates the use of a TextField and a GridPane.
//*****

public class Name extends GridPane
{
    private Label result;
    private TextField name;
    //-----
    // Sets up a GUI containing a labeled text field for Displaying the
    // greeting.
    //-----
    public Name()
    {
        Font font = new Font(12);

        Label inputLabel = new Label("First Name:");
        inputLabel.setFont(font);
        GridPane.setHalignment(inputLabel, HPos.RIGHT);

        result = new Label("");
        result.setFont(font);
        GridPane.setHalignment(result, HPos.CENTER);
    }
}
```

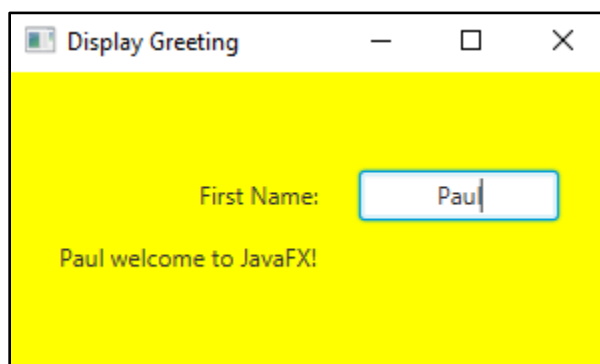
```

name = new TextField();
name.setFont(font);
name.setPrefWidth(100);
name.setAlignment(Pos.CENTER);
name.setOnAction(this::processReturn);

setAlignment(Pos.CENTER);
setHgap(20);
setVgap(10);
setStyle("-fx-background-color: yellow");

add(inputLabel, 0, 0);
add(name, 1, 0);
add(result, 0, 1);
}
//-----
// Displays the greetings
//-----
public void processReturn(ActionEvent event)
{
    String getName = name.getText();
    result.setText(getName + " welcome to JavaFX!");
}
}

```



TextField Program: display a greeting after the user enters first name and press the enter key

The details of the user interface are set up in a separate class that extends `GridPane`. `GridPane` is a JavaFX layout pane that displays nodes in a rectangular grid. The GUI elements are set up using the `Name` constructor. The event handler `processReturn` is defined in the `Name` class.

Through inheritance, the `Name` class extends the `GridPane`. As a result, it inherits the `add` method. The parameters to `add` specify the grid cell to which to add the node. The row and column numbering in a grid pane starts at 0. When the user enters their first name and press the Enter button on the keyboard, the event handler method is called, which displays the greeting with the name the user entered.

Check Boxes:

This represents a binary (i.e. on or off) choice. This is implemented using the JavaFX CheckBox class. An action event is generated by checking or unchecking the check box.

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.stage.Stage;
/*****
// Name.java      Java Foundations
//
// Demonstrates the use of check boxes.
*****/
public class Name extends Application
{
    //-----
    // Creates and presents the program window.
    //-----
    public void start(Stage primaryStage)
    {
        NameStructurePane pane = new NameStructurePane();
        pane.setAlignment(Pos.CENTER);
        pane.setStyle("-fx-background-color: skyblue");
        Scene scene = new Scene(pane, 400, 150);

        primaryStage.setTitle("Name Structure");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Layout Panes:

The NameStructurePane class uses two layout panes: HBox and VBox. The HBox pane arranges its nodes into a single row horizontally, while the VBox pane arranges its nodes into a single column vertically. NameStructurePane extends VBox and is used to put the text above the check boxes.

```
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.control.CheckBox;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.scene.text.FontWeight;
/*****
// NameStructure.java      Java Foundations
//
// Demonstrates the use of check boxes.
*****/
```

```

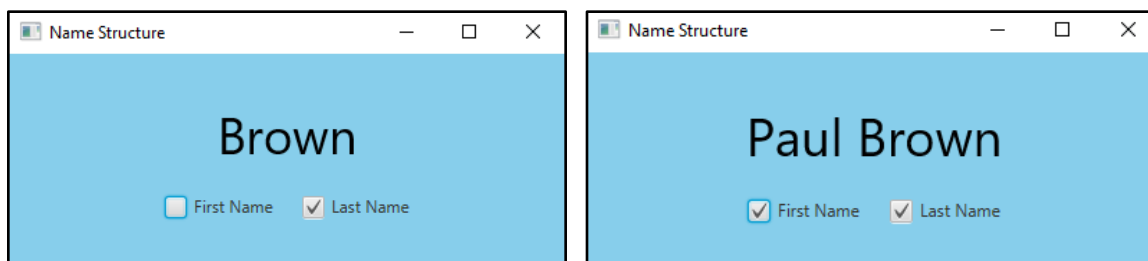
public class NameStructurePane extends VBox
{
    private Text name;
    private CheckBox firstNameCheckBox, lastNameCheckBox;
    private String fname = "Paul", lname = "Brown";
    //-----
    // Sets up this pane with a Text object and check boxes that
    // determine the style of the text font.
    //-----
    public NameStructurePane()
    {
        name = new Text(fname + " " + lname);
        name.setFont(new Font("Helvetica", 36));
        firstNameCheckBox = new CheckBox("First Name");
        lastNameCheckBox = new CheckBox("Last Name");

        firstNameCheckBox.setOnAction(this::processCheckBoxAction);
        lastNameCheckBox.setOnAction(this::processCheckBoxAction);

        HBox options = new HBox(firstNameCheckBox, lastNameCheckBox);
        options.setAlignment(Pos.CENTER);
        options.setSpacing(20); // between the check boxes

        setSpacing(20); // between the text and the check boxes
        getChildren().addAll(name, options);
    }
    //-----
    // Displays the section of the name based on the box that is checked
    //-----
    public void processCheckBoxAction(ActionEvent event)
    {
        if (firstNameCheckBox.isSelected())
            name.setText(fname);
        if (lastNameCheckBox.isSelected())
            name.setText(lname);
        if (lastNameCheckBox.isSelected() &&
            firstNameCheckBox.isSelected())
            name.setText(fname + " " + lname);
    }
}

```



Text Box with Layout Panes Program

Radio Buttons:

A radio button is a specialized `ToggleButton` where only one item or button can be selected from the list of items unlike the `CheckBox` objects above.

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.stage.Stage;
/*****
// TrueFalseQuestion.java    Java Foundations
//
// Demonstrates the use of radio buttons.
*****/
public class TrueFalseQuestion extends Application
{
    //-----
    // Creates and presents the program window.
    //-----
    public void start(Stage primaryStage)
    {
        NameStructurePane pane = new NameStructurePane();
        pane.setAlignment(Pos.CENTER);
        pane.setStyle("-fx-background-color: lightgreen");
        Scene scene = new Scene(pane, 500, 150);

        primaryStage.setTitle("Name Structure");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.HBox;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.scene.text.Font;
/*****
```

```

// NameStructurePane.java      Java Foundations
//
// Demonstrates the use of radio buttons.
//*****
public class NameStructurePane extends HBox
{
    private Text name;
    private String firstName, lastName, fullName;
    private RadioButton firstNameButton, lastNameButton,
    fullNameButton;

    //-----
    // Sets up this pane with a Text object and radio buttons that
    // determine which phrase is displayed.
    //-----
    public NameStructurePane() {
        firstName = "John";
        lastName = "Brown";
        fullName = firstName + " "+lastName;

        name = new Text(fullName);
        name.setFont(new Font("Helvetica", 24));

        StackPane quotePane = new StackPane(name);
        quotePane.setPrefSize(300, 100);

        ToggleGroup group = new ToggleGroup();

        firstNameButton = new RadioButton("First Name");
        firstNameButton.setToggleGroup(group);
        firstNameButton.setOnAction(this::processRadioButtonAction);

        lastNameButton = new RadioButton("Last Name");
        lastNameButton.setToggleGroup(group);
        lastNameButton.setOnAction(this::processRadioButtonAction);

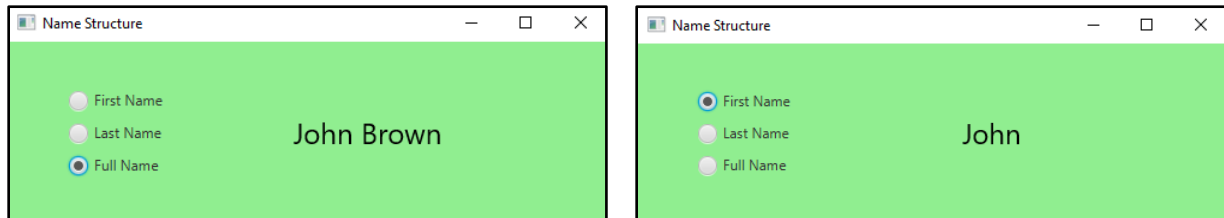
        fullNameButton = new RadioButton("Full Name");
        fullNameButton.setSelected(true);
        fullNameButton.setToggleGroup(group);
        fullNameButton.setOnAction(this::processRadioButtonAction);

        VBox options = new VBox(firstNameButton, lastNameButton,
                                fullNameButton);
        options.setAlignment(Pos.CENTER_LEFT);
        options.setSpacing(10);
        setSpacing(20);
        getChildren().addAll(options, quotePane);
    }

    //-----
    // Updates the content of the displayed text.

```

```
//-----
public void processRadioButtonAction(ActionEvent event) {
    if (firstNameButton.isSelected())
        name.setText(firstName);
    else if (lastNameButton.isSelected())
        name.setText(lastName);
    else
        name.setText(fullName);
}
}
```



Radio Buttons program display

To establish where only a single button can be selected at one time, the radio buttons work together as a group are added to a ToggleGroup object. The setToggleGroup() method is used to specify which toggle group to which a button belongs. The isSelected() method of a radio button returns true if that button is currently "on" and false otherwise.

Color and Date Pickers:

A color picker allows the user to select a color from a palette and the date picker allows the user to select a calendar date. They both use a drop-down pane from which to select.

```
import java.time.LocalDate;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.ColorPicker;
import javafx.scene.control.DatePicker;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Text;
import javafx.stage.Stage;
//*****
// DateColorPicker.java      Java Foundations
//
// Demonstrates the use of the color picker and date picker.
//*****
public class DateColorPicker extends Application
{
    private Text message;
```

```

private DatePicker datePicker;
private ColorPicker colorPicker;
//-----
// Allows the user to select a date and a color. A Text object
// displays the day of the week, month, day, and year in the color
// chosen.
//-----
public void start(Stage primaryStage)
{
    datePicker = new DatePicker(LocalDate.now());
    datePicker.setOnAction(this::processDateChoice);

    colorPicker = new ColorPicker(Color.PURPLE);
    colorPicker.setOnAction(this::processColorChoice);
    message = new Text("'" + LocalDate.now().getDayOfWeek() + " " + LocalDate.now().getMonth() + "
" + LocalDate.now().getDayOfMonth() + ", " + LocalDate.now().getYear());

    HBox pickers = new HBox(datePicker, colorPicker);
    pickers.setSpacing(10);
    pickers.setAlignment(Pos.CENTER);

    VBox root = new VBox();
    root.setStyle("-fx-background-color: white");
    root.setSpacing(15);
    root.setAlignment(Pos.CENTER);
    root.getChildren().addAll(pickers, message);
    Scene scene = new Scene(root, 400, 150);

    primaryStage.setTitle("Picker Demo");
    primaryStage.setScene(scene);
    primaryStage.show();
}

//-----
// Gets the value of the date from the date picker and updates the
// message with the corresponding day of the week.
//-----
public void processDateChoice(ActionEvent event)
{
    LocalDate date = datePicker.getValue();
    message.setText(date.getDayOfWeek() + " " + date.getMonth() + " " + date.getDayOfMonth() + ",
" + date.getYear());
}

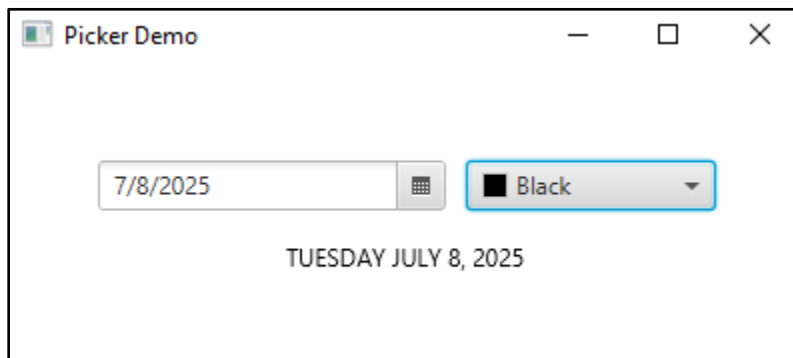
//-----
// Gets the color specified in the color picker and sets the
// color of the displayed message.
//-----
public void processColorChoice(ActionEvent event)
{

```

```

        //Color selectColor = colorPicker.getValue();
        message.setFill(colorPicker.getValue());
    }
}

```



Date and Color Picker display

Mouse Events:

There are several mouse-based events represented in JavaFX:

1. Mouse pressed: means the mouse button is pressed. Event registration method is `setOnMousePressed(EventHandler<MouseEvent>)`.
2. Mouse released: means the mouse button is released. Event registration method is `setOnMouseReleased(EventHandler<MouseEvent>)`.
3. Mouse clicked: means the mouse button is pressed and released. Event registration method is `setOnMouseClicked(EventHandler<MouseEvent>)`.
4. Mouse entered: mouse pointer is moved onto a node. Event registration method is `setOnMouseEntered(EventHandler<MouseEvent>)`.
5. Mouse exited: mouse pointer is moved off a node. Event registration method is `setOnMouseExited(EventHandler<MouseEvent>)`.
6. Mouse moved: mouse is moved. Event registration method is `setOnMoved(EventHandler<MouseEvent>)`.
7. Mouse dragged: mouse is moved while the mouse button is pressed and held. Event registration method is `setOnMouseDragged(EventHandler<MouseEvent>)`.

A `MouseEvent` object is an event that can be used to obtain the mouse position. The event handler can be set for each type of mouse event. A stream of mouse moved, or mouse dragged events occurs while the mouse is in motion.

Some methods from the `MouseEvent` class are listed below:

Method	Description
<code>MouseButton getButton()</code>	Represents the mouse button is pressed
<code>int getClickCount()</code>	Returns the number of times the mouse is clicked.
<code>double getX()</code>	Returns the X-coordinate of where the mouse points
<code>double getY()</code>	Returns the Y-coordinate of where the mouse points
<code>double getSceneX()</code>	Returns the X-coordinate of the where the mouse points in the scene.
<code>double getSceneY()</code>	Returns the Y-coordinate of the where the mouse points in the scene.

double getScreenX()	Returns the X-coordinate of the where the mouse points in the screen.
double getScreenY()	Returns the Y-coordinate of the where the mouse points in the screen.

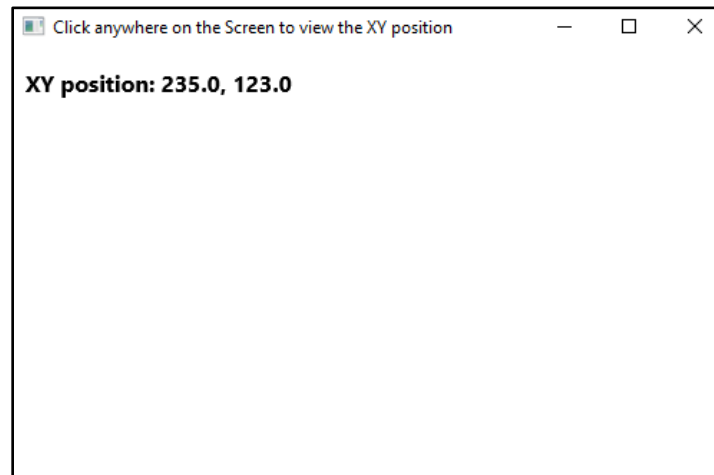
The program below displays the X and Y position of where the mouse is clicked on the screen.
Note: this is the position where the mouse button is pressed and released.

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;

/*****
// ClickPosition.java      Java Foundations
//
// Demonstrates the handling of a mouse click event.
*****/
public class ClickPosition extends Application
{
    private Text positionText;
    //-----
    // Shows the X & Y position where the mouse is clicked.
    //-----
    public void start(Stage primaryStage){
        positionText = new Text(10, 30, "");
        positionText.setFont(Font.font("Helvetica", FontWeight.BOLD,
            FontPosture.REGULAR, 16));
        Group root = new Group(positionText);

        Scene scene = new Scene(root, 500, 300);
        scene.setOnMouseClicked(this::processMouseClicked);
        primaryStage.setTitle("Click anywhere on the Screen to view
            the XY position");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    //-----
    // Displays the X & Y position after a mouse click event
    //-----
    public void processMouseClicked(MouseEvent event) {
        double clickX = event.getX();
        double clickY = event.getY();
        positionText.setText("XY position: " + clickX + ", " +
            clickY);
    }
}
```

```
}
}
```



Displays the X & Y position of a mouse click event

Keyboard Events:

There are three events represented in the JavaFX programs. They are:

1. Key Pressed: this is an event where any keyboard key is pressed down. Event registration method is `setOnKeyPressed(EventHandler<MouseEvent>)`.
2. Key Released: this is an event where any keyboard key is released. Event registration method is `setOnKeyReleased(EventHandler<MouseEvent>)`.
3. Key Typed: this is an event where the keyboard key generates a character (pressed and released). Event registration method is `setOnKeyTyped(EventHandler<MouseEvent>)`.

Some methods from the `KeyEvent` class are listed below:

Method	Description
<code>String getCharacter()</code>	Returns the character represented by the key that was pressed
<code>String getText()</code>	The method returns the string that describes the key code.
<code>KeyCode getCode()</code>	The method returns the key code that is associated with key that was pressed.

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.input.KeyEvent;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
//*****
// BallMovement.java      Java Foundations
//
// Demonstrates the handling of keyboard events.
//*****
```

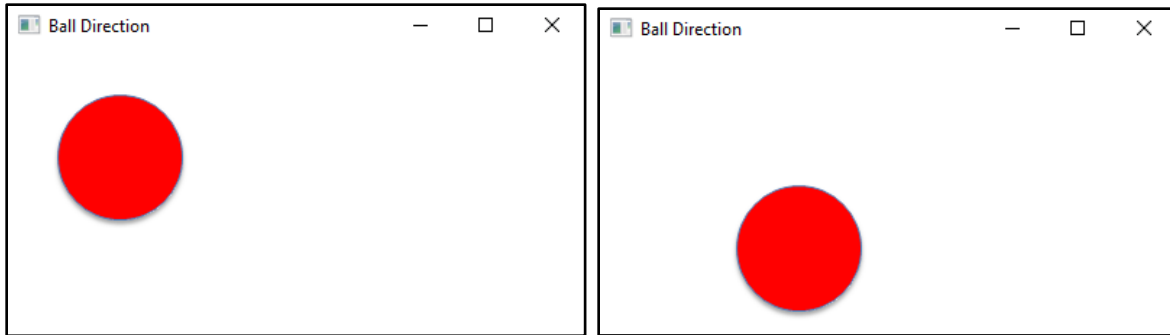
```

public class BallMovement extends Application{
    private int ball = 30;
    private ImageView imageView;

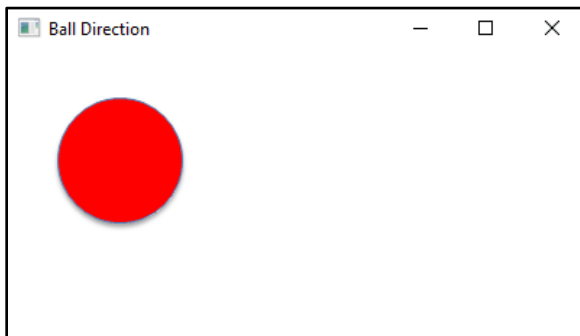
    public void start(Stage primaryStage) {
        Image redBall = new Image("ball.png");
        imageView = new ImageView(redBall);
        imageView.setX(30);
        imageView.setY(30);
        Group root = new Group(imageView);
        Scene scene = new Scene(root, 400, 200, Color.WHITE);
        scene.setOnKeyPressed(this::processKeyPress);
        primaryStage.setTitle("Ball Direction");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public void processKeyPress(KeyEvent event) {
        switch (event.getCode()) {
            case UP:
                if((imageView.getY() - ball) >=0)
                    imageView.setY(imageView.getY() - ball);
                break;
            case DOWN:
                if((imageView.getY() + ball) <=100)
                    imageView.setY(imageView.getY() + ball);
                break;
            case RIGHT:
                if((imageView.getX() + ball) <=300)
                    imageView.setX(imageView.getX() + ball);
                break;
            case LEFT:
                if((imageView.getX() - ball) >=0)
                    imageView.setX(imageView.getX() - ball);
                break;
            case HOME:
                imageView.setX(30);
                imageView.setY(30);
                break;
            default:
                break;
        }
    }
}

```



Left: starting position of the ball. Right: press the right arrow key twice and the down arrow key twice



Press the Home key to move the ball back to the starting position

Note: place the image file in the same location as the “src” folder for the JavaFX project. Then right click on the project in your IDE and select “Refresh”.

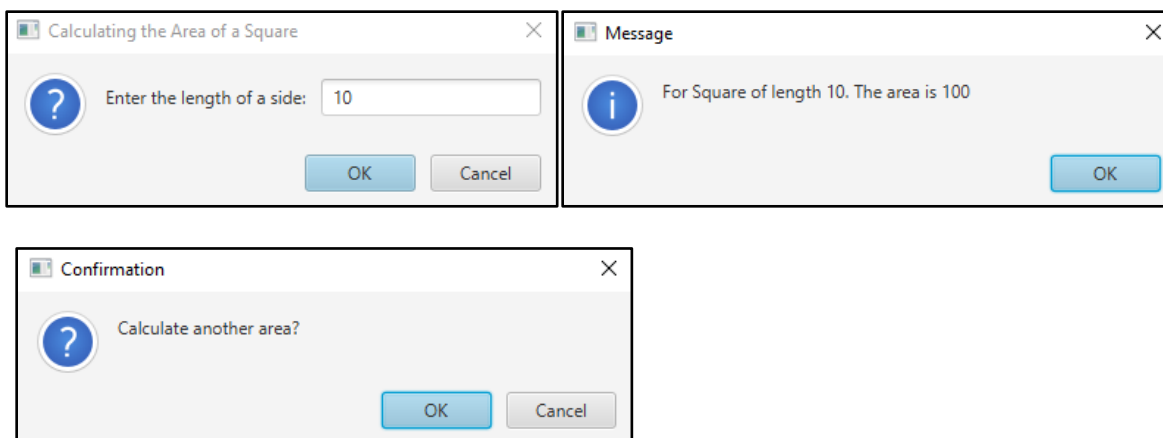
Dialog Box:

This is a window that pops up that shows information in the application window. It can be used to gather input or display a message to the user. There are several classes that provide support for dialog boxes. These include:

1. **Alert:** this is used when the user is prompted for a response. For example, `AlertType.CONFIRMATION` displays a dialog box that is seeking confirmation from the user, `AlertType.WARNING` displays a dialog box with the aim of warning the user about some action or fact. When creating an `Alert` object, the `Alert.AlertType` enumeration value is passed as an argument to its constructor. For example, `Alert rebootAlert = new Alert(Alert.AlertType.CONFIRMATION, “Do you want to do another?”);`.
2. **TextInputDialog and ChoiceDialog:** these are used when the application prompts the user to enter a text input or to make a choice from a drop-down choice box.

```
import java.util.Optional;
import javafx.application.Application;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.ButtonType;
import javafx.scene.control.TextInputDialog;
import javafx.stage.Stage;
//*****
// SquareArea.java    Java Foundations
```

```
//
// Demonstrates the use of information and confirmation alerts, as well
// as text input dialog boxes.
//*****
public class SquareArea extends Application
{
    //-----
    // Prompts the user for an integer, calculate the area of the
    // square using the integer, then asks if the user would like to
    // calculate the area of another square. All interactions are
    // performed using dialog boxes.
    //-----
    public void start(Stage primaryStage) throws Exception
    {
        boolean doAnother = true;
        while (doAnother)
        {
            TextInputDialog inputDialog = new TextInputDialog();
            inputDialog.setHeaderText(null);
            inputDialog.setTitle("Calculating the Area of a Square");
            inputDialog.setContentText("Enter the length of a side:");
            Optional<String> numString = inputDialog.showAndWait();
            if (numString.isPresent())
            {
                int num = Integer.parseInt(numString.get());
                String result = "For Square of length "+ num +". The
                area is " + (num * num);
                Alert answerDialog = new Alert(AlertType.INFORMATION);
                answerDialog.setHeaderText(null);
                answerDialog.setContentText(result);
                answerDialog.showAndWait();
                Alert confirmDialog = new
                    Alert(AlertType.CONFIRMATION);
                confirmDialog.setHeaderText(null);
                confirmDialog.setContentText("Calculate another area?");
                Optional<ButtonType> another =
                    confirmDialog.showAndWait();
                if (another.get() != ButtonType.OK)
                    doAnother = false;
            }
            else
                doAnother = false;
        }
    }
}
```



Dialog Box Program Calculating and Displaying the Area of a Square

Slider:

A slider is a GUI control that provides the user with a numeric range of choices from which to choose a value. To set the numeric range of the slider the minimum and maximum values are passed as arguments in the Slider constructor. A slider can have a vertical or horizontal orientation. To obtain the current value of the slider, the user can invoke the `getValue()` method.

To react dynamically to a slider's motion property binding or a change listener can be implemented in the program. Below is a JavaFX program that contains a slider that changes the shape of a rectangle.

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Orientation;
import javafx.scene.Scene;
import javafx.scene.control.Slider;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
//*****
// RectangleSliders.java      Java Foundations
//
// Demonstrates the use of slider controls and property binding.
//*****
public class RectangleSliders extends Application
{
    private Rectangle rectangle;
    private Slider xSlider, ySlider;
    //-----
    // Displays a rectangle with sliders that control the width and
    // height of the rectangle.
    //-----
    public void start(Stage primaryStage) {
        rectangle = new Rectangle(250, 150, 50, 100);
        rectangle.setFill(Color.SALMON);
```

```

xSlider = new Slider(0, 300, 150);
xSlider.setShowTickMarks(true);
xSlider.setPadding(new Insets(0, 20, 20, 80));

rectangle.widthProperty().bind(xSlider.valueProperty());

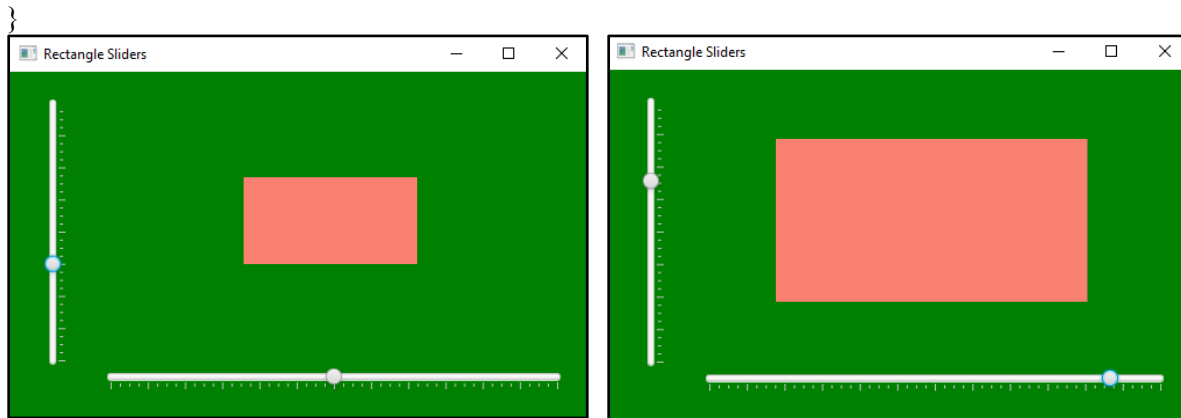
ySlider = new Slider(0, 200, 75);
ySlider.setOrientation(Orientation.VERTICAL);
ySlider.setShowTickMarks(true);
ySlider.setPadding(new Insets(20, 0, 0, 30));

rectangle.heightProperty().bind(ySlider.valueProperty());

BorderPane pane = new BorderPane();
pane.setLeft(ySlider);
pane.setBottom(xSlider);
pane.setCenter(rectangle);
pane.setStyle("-fx-background-color: green");
Scene scene = new Scene(pane, 500, 300);

primaryStage.setTitle("Rectangle Sliders");
primaryStage.setScene(scene);
primaryStage.show();
}

```



Slider for a rectangle object

Chapter 5: Arrays

Introduction to Arrays

Imagine you are building a Student Registration System. As part of the project, you must store information for 100 students—their names, IDs, grades, or contact numbers. If you try to store each piece of data using separate variables such as:

```
int student1ID;  
int student2ID;  
int student3ID;  
//...  
int student100ID;
```

Your program will quickly become disorganized, repetitive, and almost impossible to manage. Changing the number of students from 100 to 200 would require adding even more variables. Searching for a specific student, printing all student names, or updating records would be extremely difficult.

This is a common problem in programming: as programs grow more complex, you need to handle large collections of related data. Storing each student's data in a separate variable is inefficient, error-prone, and does not scale.

To solve this, Java provides a powerful structure called an **array**, which allows you to store multiple values of the same type under a single variable name and access each value using an index. Arrays help simplify your code, improve readability, and make data processing easier and faster.

Array Basics: Declaration, Creation, and Initialization

An array is a **fixed-length data structure** that stores multiple values of the **same type** using a single variable name. Each value stored in an array is called an **element**, and each element can be accessed using an **index**.

We will start our exploration of arrays with simple **one-dimensional arrays** and mainly focus on arrays that store primitive type elements (byte, short, int, long, char, boolean, float, and double).

Declaration of Arrays

Declaration Syntax: *elementType* [] *arrayName*;

Example 1: Array Declaration

```
int [ ] studentID;  
Student [ ] studentList;
```

The [] symbol here is used to indicate this is a one-dimensional array.

An array is considered as **reference type** in Java. This means that the declaration of an array **does not create an array instance**; instead, it simply defines a **reference variable** that can be used to refer to an array instance that will be created later.

In the previous example:

- studentID can be used to refer to an array instance that stores a collection of *int* values.
- studentList can be used to refer to an array instance that stores a collection of Student variables.

Note: Java also allows you to use the following syntax to declare an array variable: *elementType arrayName[]*. For example, `int arr[ ]`; However, this format is less commonly used.

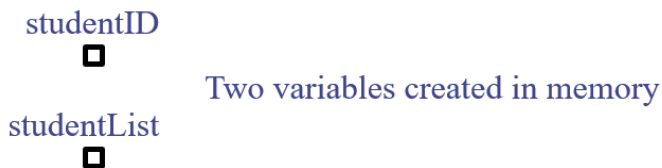


Fig. 5.1: Declaration of two array reference variables

Creation of Arrays

Creation Syntax: *new elementType[size of array]*;

Example 2: Array Creation

```
new int[100];
new Student[100];
```

As with all other reference types in Java, an array instance is created using the **new** operator. However, since an array is a container used to store a collection of elements that will be added later, there is no constructor to call during array creation. Instead, we must specify the **length** (size) of the array so that the appropriate amount of memory can be allocated to meet its storage requirements.

Once an array is created, its size is **fixed** and **cannot** be modified. The reason for this will be explained in a later section.

Example 3: Array Initialization

```
int[] studentID = new int[100];
```

This statement creates an `int` array with space for 100 elements. The assignment operator (`=`) causes the reference variable `studentID` to refer to the block of memory in which these 100 consecutive elements have been allocated. The physical memory allocated due to the right-hand side operation cannot be resized. However, the reference variable `studentID` itself may be reassigned to refer to a different `int` array. This behavior will be examined in detail later.

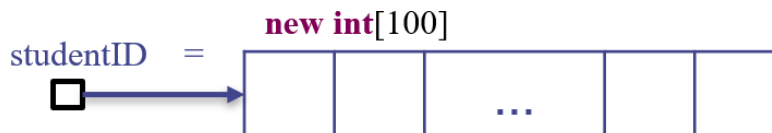


Fig. 5.2: Array Initialization

Array Indexing and Access

After the creation of an array, we can start to operate on the array, including adding elements, removing elements, searching for elements, modifying elements, and so on.

Since an array contains multiple positions (each storing one element), any operation on the array involves the concept of indexing. **Indexing** defines how we can identify each position in the array and how we can access the elements stored there.

The positions in an array are **indexed by integers starting from 0 to length-1**. For example, if an array contains 100 elements, the positions are indexed from 0, 1, ..., 99. To access an element stored at a specific position, the syntax is: *arrayName[index]*

Example 4: Access Element Using Index

- `studentID[0]` refers to the element stored at the **first** position of the `studentID` array.

- `studentID[99]` refers to the element stored at the **last** position of the `studentID` array.

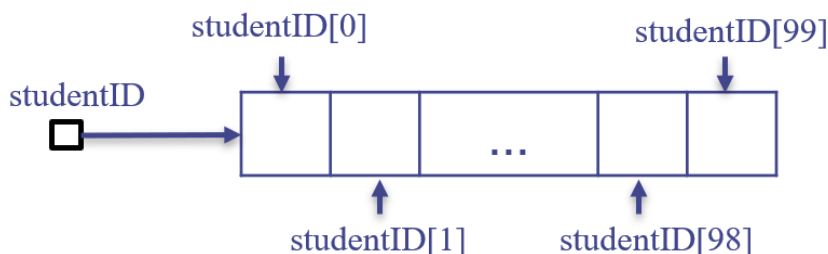


Fig. 5.3: Access Element Using Index

Operation on Arrays

Now that we understand how to identify and access each position in an array using indices, the next step is to learn how to assign values to these positions. In Java, any operation that changes the content of a variable is performed using the **assignment** operation.

Example 5: Assigning initial values to array elements

```
studentID[0] = 0;
studentID[1] = 1;
studentID[2] = 2;
```

When we have an array of 100 `int` elements, we definitely don't want to initialize each element manually. In the example above, notice that the value assigned to each element matches its index, and the index increases by 1 each time. This pattern makes iteration—using a `for` loop or `while` loop—a natural choice.

Here is how we can use a `for` loop to initialize `studentID` so that each element's value is equal to its index:

```
for (int i=0;i<studentID.length;i++)
    studentID[i]=i;
```

Note: In Java you can find the size of an array by using the variable `arrayName.length`. If a program attempts to access an index that is outside of its valid range (from 0 to `arrayName.length-1`), Java will throw an exception (an error) called **ArrayIndexOutOfBoundsException**. In particular, because array indices start at 0, a very common beginner mistake is to try to access the last element using the index `length` instead of `length - 1`, which is known as an **off-by-one** error.

```
1 public class Array {
2     public static void main(String[] args)
3     {
4         int[] studentID= new int[100];
5         for (int i=0;i<=studentID.length;i++)
6             studentID[i]=i;
7     }
8 }
9
```

Problems Javadoc Declaration Console x

terminated> Array [Java Application] C:\Program Files\Eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.3.v20240426-1530\jre\bin\javaw.exe (Nov 2

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 100 out of bounds
at Array.main(Array.java:7)

Fig. 5.4: An example of `ArrayIndexOutOfBoundsException`

Enhanced for loop (for-each)

Besides the regular for loop, Java also provides enhanced for loop (also known as a *for-each* loop) that makes it easier to access elements in an array. The enhanced loop has the following format:

```
for (dataType variableName : arrayName) {  
    // use variableName directly  
}
```

In this loop, variableName takes on the value of each element in the array, one at a time, from the first element to the last. This allows you to read or process each element without needing to manage an index manually.

Example 6: Print All Elements In Studentid Using an Enhanced for Loop

```
for (int id : studentID)  
    System.out.println(id);
```

Example 7: Print All Elements In Studentid Using Regular for Loop

```
for(int i=0;i< studentID.length;i++)  
    System.out.println(studentID[i]);
```

The enhanced loop is simpler and reduces the chance of common mistakes, such as off-by-one errors.

Note: The enhanced for loop is meant for reading elements, not changing them. This is because the loop variable is just a **copy** of the element (for primitive types) or a reference that shouldn't be reassigned (for objects). If you try to change the loop variable, it **won't affect the original array**.

```
1 public class Array {  
2     public static void main(String[] args)  
3     {  
4         int[] studentID= new int[100];  
5         for (int i=0;i<studentID.length;i++)  
6             studentID[i]=i;  
7  
8         for (int id : studentID) {  
9             id=0;  
10            System.out.print(id+"\t");  
11        }  
12        System.out.println();  
13        for (int id : studentID) {  
14            System.out.print(id+"\t");  
15        }  
16    }  
17 }  
18
```

Problems Javadoc Declaration Console x

<terminated> Array [Java Application] C:\Program Files\Eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

Fig. 5.5: An example showing that the loop variable is just a copy of the element

Arrays Storing Reference Data

After a thorough discussion of arrays that store primitive data types, let's now look at arrays that store reference data. In Java, any data type other than the eight primitive types is a reference type. This includes:

- Classes from the Java library, like `String`
- All classes that you define yourself

Example 8: Array of Strings

```
String[] names = {"Alice", "Bob", "Charlie"};
for (String name : names)
    System.out.println(name);
```

Example 9: Array of User-Defined Objects

```
public class Student { //user defined class
    private int id;
    private String name;
    public Student(int id) {
        this.id = id;
    }
    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getId() {
        return id;
    }
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Student other = (Student) obj;
        return this.id == other.id;
    }
    public String toString() {
        return "Student [" + id + ", " + name + "];"
    }
}

Student[] studentList = new Student[100]; //created in a separate class
studentList[0] = new Student(0, "Anna");
studentList[1] = new Student(1, "Bob");
studentList[98] = new Student(98, "Charlie");
for (Student s : studentList)
```

```
System.out.print(s+"--");
```

In Java, arrays of reference types store **references** (or “pointers”) to objects instead of storing the objects themselves. This happens because objects are created while the program is running and are stored in a special part of memory called the **heap**, which is separate from the memory used to hold the array structure.

Examples 7 and 8 demonstrate two arrays: one that contains String elements and one that contains Student elements. The studentList array from Example 8 is an array that holds 100 Student **reference variables**, not 100 Student objects. Each reference can later point to a Student object. This allows studentList to act as a container that can store up to 100 Student objects, even though the array itself only stores references.

The example also shows how to initialize elements in an array of reference types.

studentList contains 100 reference variables—studentList[0], studentList[1], ..., studentList[99]. Each one can refer to a Student object. In the example, we created two Student objects [0, “Anna”] and [1, “bob”] using the new operator followed by a constructor call following the definition of the Student class, and then assigned them to studentList[0] and studentList[1]. Since only the first two elements were assigned, all the other reference variables still contain null, which is the default value for reference variables in Java.

The following diagram visualizes the state of the array.

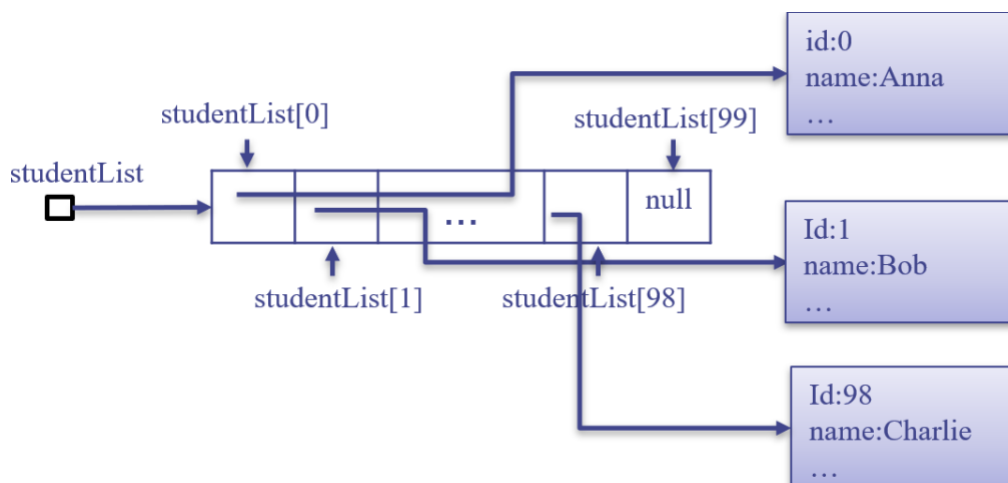


Fig. 5.6: Example illustrating arrays storing reference data

This design has several advantages:

- **Efficiency:** Copying large objects into arrays would be slow and use a lot of memory. Storing references instead is faster and more memory efficient.
- **Shared access:** Multiple arrays or variables can refer to the same object without creating duplicates.

However, we must keep in mind that we are *not* working with a copy of the object. This becomes especially important when arrays are used as parameters in methods, which we will discuss in detail in the next section.

Arrays as Parameters

Now that we understand how arrays store either primitive values or references to objects, we can take a closer look at how arrays behave when they are used as parameters in methods.

As you may have learned in previous chapters, the Java programming language conceptually describes two ways parameters can be passed to methods. One is **pass-by-value**, where a copy of the actual parameter's value is passed into the formal parameter. In this case, operations inside the method do not affect the original argument. The other is **pass-by-reference**, where the formal parameter refers to the same object as the actual parameter. As a result, changes made inside the method also affect the original object.

In Java, all parameters are technically passed by value. However, when the value being passed is a reference (such as an array), the method receives a copy of the reference that still points to the same underlying array. In practice, this makes using arrays as parameters behave like **pass-by-reference**. This means that when an array is passed to a method, any modifications made to the array inside the method will also affect the original array outside the method. If you want to avoid this behavior, you must manually create and work with a copy of the array.

Passing arrays to methods

In this subsection, we will explore the basics of passing arrays to methods for display or search operation.

Example 10: Find the Minimum Element in an int Array

```
public static int findMin(int[] array) {
    int min=array[0];
    for(int i=0;i<array.length;i++) {
        if (array[i]<min)
            min=array[i];
    }
    return min;
}
//testing code in main method
int[] testArray= {9, 3, 4, 7, 9};
System.out.println(findMin(testArray));
//output: 3
```

This method accepts an int array, finds, and returns the minimum element. The method is defined as static because we want to test it in the main method. Using the provided testing code in the main method, the output is 3, which represents the method's return value.

Example 11: Find a target element in an int array

```
public static boolean findTarget(int[] array,int target) {
    for(int value: array){
        if (value==target)
            return true;
    }
    return false;
}
//test code in main method
int[] testArray= {9, 3, 4, 7, 9};
System.out.println(findTarget(testArray, 10));
//output: false
```

This method accepts an int array and a target value. It performs a linear search and returns true if it finds the target value in the array, or false otherwise. In this implementation, an enhanced for loop is used. Using the provided testing code in the main method, the output is false because the array does not contain the value 10.

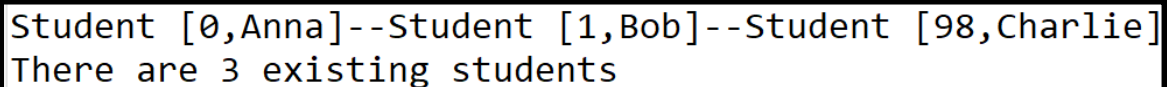
Example 12: Print a Student Array

This method accepts a Student array, prints the existing elements, and returns the number of existing students in the array.

```
public static int printStudents(Student[] array)
{
    int counter=0;
    for(int i=0;i<array.length;i++) {
        if (array[i]!=null) {
            counter++;
            if(i!=array.length-1)
                System.out.print(array[i]+"--");
            else
                System.out.print(array[i]);
        }
    }
    System.out.println();
    return counter;
}
```

Below is the testing code in the main method, along with a screenshot of the output:

```
Student[] studentList = new Student[100];
studentList[0]=new Student(0, "Anna");
studentList[1]=new Student(1, "Bob");
studentList[98]=new Student(98, "Charlie");
System.out.println("There are "+printStudents(studentList)+ " existing students");
```



```
Student [0,Anna]--Student [1,Bob]--Student [98,Charlie]
There are 3 existing students
```

Example 13: Find a target element in a Student array

This method accepts a Student array and a target value; it performs a linear search and returns true if it finds the target value, or false otherwise.

```
public static boolean findTarget(Student[] array, Student target) {
    for(int i=0;i<array.length;i++) {
        if (array[i]!=null && array[i].equals(target))
            return true;
    }
    return false;
}
```

Below is the testing code in the main method, along with a screenshot of the output:

```
Student target1 = new Student(0, "Anna");
Student target2 = new Student(4, "David");
if (findTarget(studentList, target1)) {
    System.out.println("Found target student");
} else {
    System.out.println("Target student not found");
}
```

```

if (findTarget(studentList, target2)) {
    System.out.println("Found target student");
} else {
    System.out.println("Target student not found");
}

```

Below is the testing code in the main method, along with a screenshot of the output:

```

Found target student
Target student not found

```

In this example, we created two Student objects and checked whether they were in the array. As you can see, one student is found in the list and the other is not. The search method works by using the equals method from the Student class to compare students. In our current implementation, two Student objects are considered the same if they have the same student ID. We will learn more about how the equals method works and why we override it in a future chapter.

One important difference between searching an int array and searching a Student array is that we must check whether array[i] is null before calling equals. In other words, we write:

```
array[i] != null && array[i].equals(target)
```

This is necessary because calling a method on something that is null will cause an error in Java. In a Student[] array, some elements may not yet point to a real Student object or has been reset to null after a removal operation. If we try to do array[i].equals(target) when array[i] is null, Java will throw a *NullPointerException*. By checking array[i] != null first, we make sure the comparison is performed only on existing Student objects.

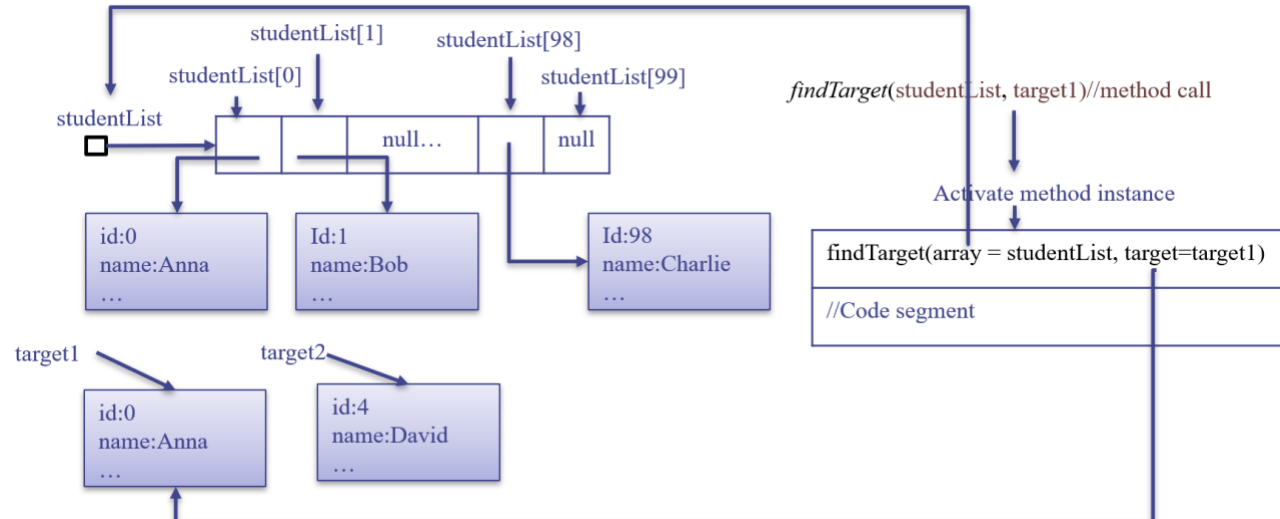


Fig. 5.7: Example illustrating Method operations that change the original array contents

Modifying array elements inside a method

In this subsection, we will explore how arrays are passed to methods and how modifications made inside a method demonstrate the effect of pass-by-reference behavior. Similar to before, we will use examples with both int arrays and Student arrays.

Example 14: Reset All Elements to 0 in an int Array

```
public static void resetArray(int[] array) {
```

```

    for(int i=0;i<array.length;i++) {
        array[i]=0;
    }
}

```

This method accepts an int array and sets all its elements to 0. The following method is used to print the array and show the effect.

Example 15: Print an int Array

```

public static void printIntArray(int[] array) {
    for(int value: array){
        System.out.print(value+" ");
    }
    System.out.println();
}

```

Below is the testing code in the main method, along with a screenshot of the output:

```

int[] testArray= {9, 3, 4, 7, 9};
printIntArray(testArray); // Print the original array
resetArray(testArray);
printIntArray(testArray); // Print the updated array

```

```

9 3 4 7 9
0 0 0 0 0

```

As you can see, the changes made inside the resetArray method also affect testArray in the main method. This happens because when we pass testArray to the method, both the method's array variable and testArray point to the same array object in memory. So, any changes we make to the array inside the method are reflected outside the method as well. This is how pass-by-reference works for arrays in Java.

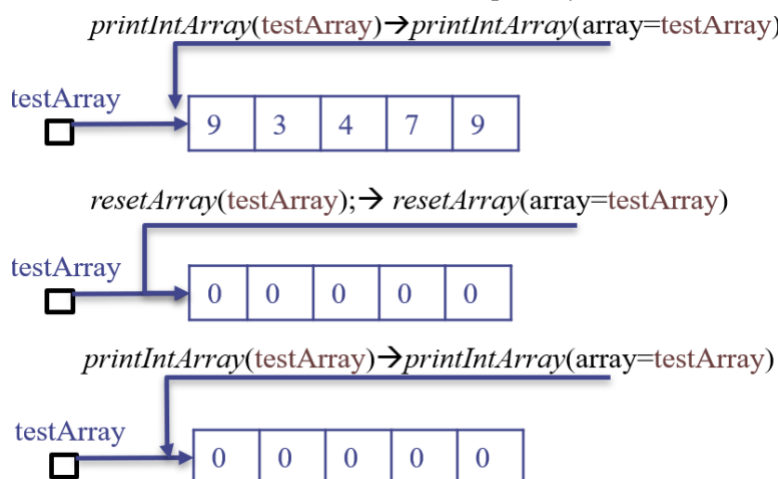


Fig. 5.8: Example illustrating arrays passed as parameters

Example 16: Method operations that change the original array contents

```

public static boolean updateName(Student[] array, int targetId, String newName) {
    boolean flag=false;
    for(int i=0;i<array.length;i++) {
        if (array[i]!=null && array[i].getId()==targetId) {

```

```

        array[i].setName(newName);
        flag=true;
    }
    return flag;
}

```

In this example, the `updateName` method accepts a `Student` array, searches for a student with the given `targetId`, and updates its name to the specified `newName`. When we pass `studentList` to the method, the method's array variable points to the same array object in memory as `studentList` in the main method.

This means that when we update a student's name inside the method, the change is reflected in the original array. After calling `updateName`, printing `studentList` again shows the updated name. This example further demonstrates how arrays in Java behave like pass-by-reference: modifications made inside a method affect the original array outside the method.

Below is the testing code in the main method, along with a screenshot of the output showing the expected results.

```

Student[] studentList = new Student[100];
studentList[0]=new Student(0, "Anna");
studentList[1]=new Student(1, "Bob");
studentList[98]=new Student(98, "Charlie");
printStudents(studentList); // Print the original array
// Update the name of the student with ID 1
if (updateName(studentList, 1, "Bob Junior")) {
    System.out.print("Name updated: ");
    printStudents(studentList);
}

```

```

Student [0,Anna]--Student [1,Bob]--Student [98,Charlie]--
Name updated: Student [0,Anna]--Student [1,Bob Junior]--Student [98,Charlie]

```

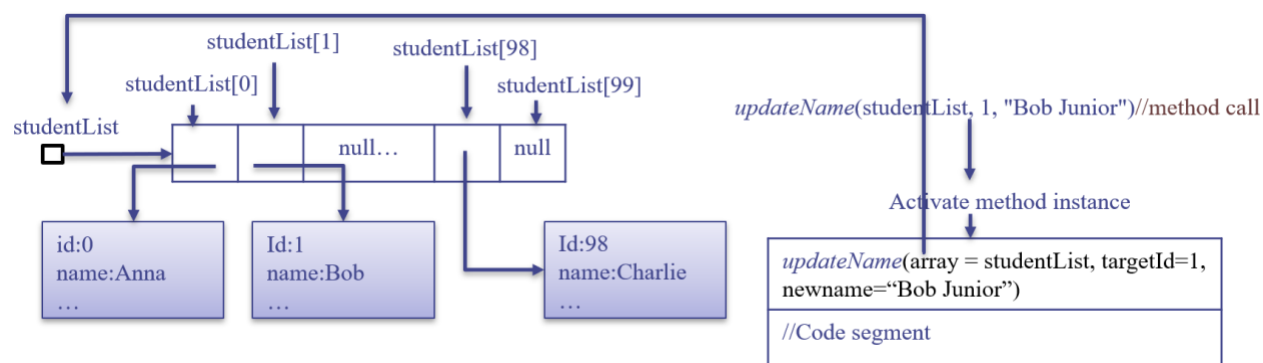


Fig. 5.9: Example illustrating arrays passed as parameters

Returning arrays

Based on the previous discussion, you should have a good idea of how arrays work as parameters, particularly how operations inside a method can affect the original array due to pass-by-reference behavior. Now you may wonder: what if you don't want to change the original array?

In this case, one solution is to **create a copy of the array**, operate on it inside the method, and return the copy for further use. The following method demonstrates a commonly used example - the **clone method**, which makes a copy of an array and returns it:

```

public static int[] clone(int[] array) {
    int[] cloneArray=new int[array.length];
    for(int i=0;i<array.length;i++)
        cloneArray[i]=array[i];
    return cloneArray;
}

```

Below is the testing code in the main method, along with a screenshot of the output showing the expected results.

```

int[] testArray= {9, 3, 4, 7, 9};
int[] cloneArray=clone(testArray);
resetArray(cloneArray); // all elements in the cloneArray are set to 0
printIntArray(testArray); //Printing both arrays shows the difference
printIntArray(cloneArray);

```

In this example, a copy of testArray is created inside the clone method and returned. The cloneArray variable in the main method then refers to this new array. This means that cloneArray and testArray now point to **two separate arrays** (two separate objects in memory), so modifying cloneArray does not affect testArray.

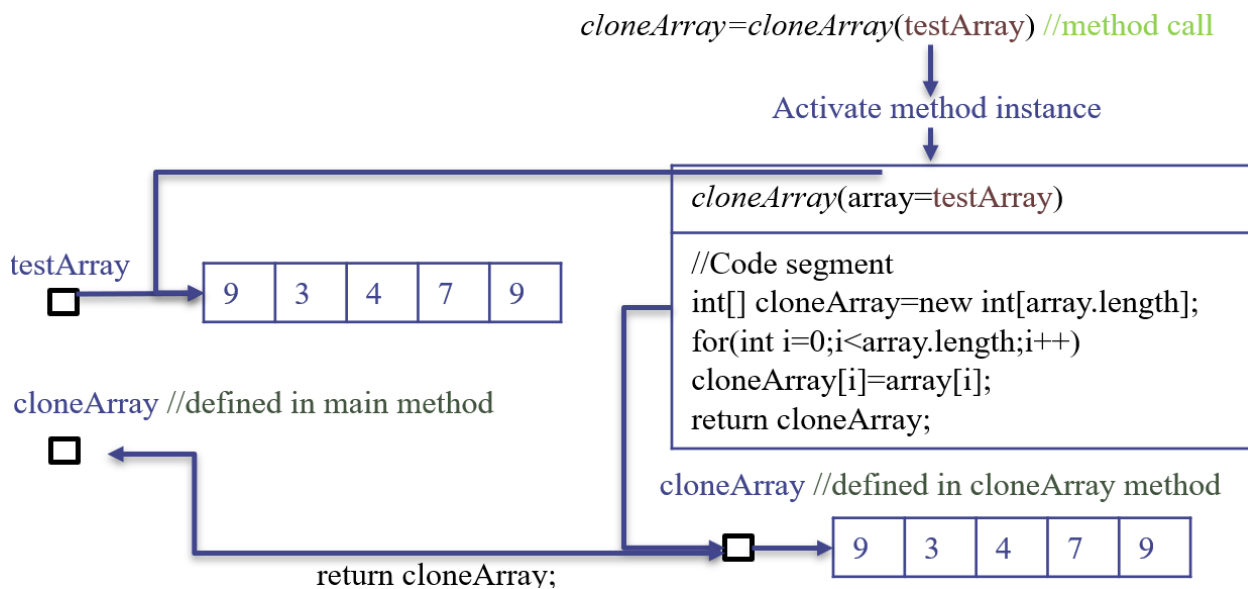


Fig. 5.10: Example illustrating a method that returns an array

Variable-Length Parameters (varargs)

After our thorough discussion of arrays as parameters, we will now move on to a related concept that is supported by arrays—essentially a behind-the-scenes use case of arrays. This concept is **varargs**, short for *variable-length arguments*. Varargs were introduced in Java 5 and allow methods to accept an arbitrary number of parameters of the same type.

Although varargs are useful, they are still less commonly used by developers—probably because, in practice, there are not many situations that require them.

Varargs in Java are declared using an ellipsis (...) after the data type of the parameter. A method can have **at most one** varargs parameter, and it **must be the last parameter** in the method's parameter list.

The following two examples illustrate how varargs work:

Example 17: Passing A Variable Length of Arguments to a Method

This method accepts a variable length int list, calculates and returns the summation.

```
public static int sum(int... varList) {
    int sum=0;
    for(int i: varList) {
        sum += i;
    }
    return sum;
}
//testing code in main method
int[] testArray= {9, 3, 4, 7, 9};
System.out.println(sum(testArray));
System.out.println(sum(1, 2, 3, 4));
System.out.println(sum(1, 2));
```

Example 18: Combining Regular Parameters with a Varargs Parameter

```
public static void printGrades(String studentName, int... grades) {
    System.out.print(studentName + ": ");
    for (int g : grades) {
        System.out.print(g + " ");
    }
    System.out.println();
}
```

This method takes a regular parameter (studentName) followed by a varargs parameter (grades). A method can only have one varargs parameter and it must be the last parameter, so the compiler can correctly identify the number of parameters during execution. You can call the method in several ways:

Example 19: Call a method with Varargs Parameter

```
printGrades("Alice", 90, 85, 88);
printGrades("Bob", 72);
```

Now that you've seen how varargs work, you may wonder how Java makes this feature possible. Although varargs allow you to pass any number of arguments to a method, Java does not treat them as a special new data structure. Instead, the Java compiler automatically converts all varargs arguments into an array of the specified type. For example, the method:

```
public static int sum(int... varList)
```

is internally converted by the compiler into something equivalent to:

```
public static int sum(int[] varList)
```

This means: When you call `sum(1, 2, 3, 4)`, Java automatically creates an array containing `{1, 2, 3, 4}` and passes it to the method. Inside the method, `varList` behaves exactly like a normal array parameter—because it is an array behind the scenes. This is also the reason why varargs can only accept arguments of the same data type.

Because varargs automatically package the provided arguments into an array behind the scenes, they work best when the method only needs to process, iterate, or inspect the values—such as summing

numbers, combining strings, or logging messages. In these cases, using varargs avoids the extra step of creating an array manually, resulting in cleaner and more natural method calls.

Arrays provide much clearer intent when the method is meant to *change* the data or when the structure of the data matters. They also avoid unexpected side effects—varargs make it less obvious to callers that an array might be altered, whereas an explicit array parameter signals that the array itself is involved directly.

Arrays and Command-Line Arguments

In all the previous examples of arrays as parameters, we explored situations where array parameters are passed to a method during the execution of the main method. But what if there are some values you want to provide when the program starts running—for example, parameters to the main method?

In Java, command-line arguments allow you to pass information into a program at startup, rather than typing input during execution. These arguments are supplied after the program name in the terminal and are received inside the main method.

You may have already used the main method in the following format many times:

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

As you can see, main method can accept a String array, denoted as args, which contains command-line arguments.

The following example demonstrates a use case where the main method verifies a username and password before allowing the rest of the program to execute:

Example 20: Authentication Verification via Command-Line Arguments

```
public static void main(String[] args) {
    if(args.length!=2)
        return; // Stop if there are not exactly two arguments
    String username=args[0];
    String password=args[1];
    if(!username.equals("admin") ||!password.equals("mySecret!"))
        return; // Stop if credentials are incorrect
    // Continue with the rest of the program if authentication passes
}
```

In this example, the main method is expected to receive two String parameters from the command line: args[0] stores the username, and args[1] stores the password. The program will only continue if both the number of arguments and the login credentials are correct. Otherwise, it stops immediately.

To run this program, there are two ways to pass the parameters:

1. **From the command line:** Run *java className username password*

For example, if your class is Array, first compile it using:

```
javac Array.java
```

Then run:

```
java Array admin Secret2025!
```

2. **Using a Java IDE:** Many Java development IDEs provide a GUI to enter command-line arguments. Below is an example from Eclipse.

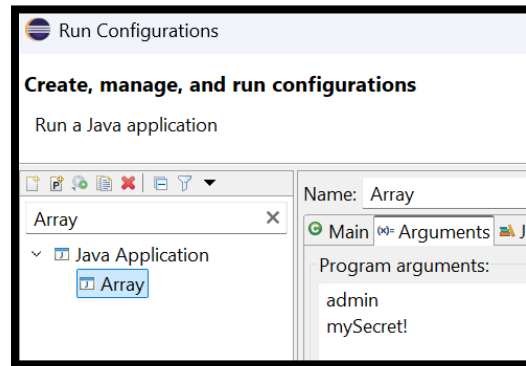


Fig. 5.11: Example illustrating command line arguments configuration in Eclipse

Note: As you may notice, command-line arguments are received as **Strings** in the args array, which means all parameters passed to this method are handled as Strings. If your program needs other types, such as int or double, you must convert the Strings to the required type.

The following example demonstrates a network application need to verify an IP address (String type) and a port number (int type):

Example 21: Verifying an IP address and a port number

```
public static void main(String[] args) {
    if(args.length!=2)
        return; // Stop if there are not exactly two arguments
    String ip=args[0];
    int portnumber=Integer.parseInt(args[1]); //Convert the second argument from String to int
    if(!ip.equals("192.168.1.10") ||!(portnumber>10000 && portnumber<65535))
        return; // Stop if credentials are incorrect
    // Continue with the rest of the program if authentication passes
}
```

In this example, we convert the port number from a String to an int so that we can compare it correctly.

Multidimensional Arrays

After a thorough discussion of one-dimensional arrays, we now turn our attention to multidimensional arrays. Among these, the two-dimensional array is the most commonly used, so we will begin by exploring its structure, declaration, and applications. Once the fundamentals of two-dimensional arrays are established, we will extend our discussion to more general multidimensional arrays and examine how they can be used to represent higher-dimensional data.

Two Dimensional Arrays

Declaration and Creation

In Java, a two-dimensional array can be declared and created using the following syntax:

DataType[][] twoDArray = new DataType[numberOfRows][numberOfColumns];

Based on our previous discussion, this syntax actually performs three distinct operations, each of which contributes to the creation and use of a two-dimensional array:

1. Declaration of the Reference Variable: `DataType[][] twoDArray`

This declares a variable named `twoDArray`. The notation `[][]` indicates that the variable refers to a two-dimensional array. The array can store only values of the specified `DataType`, ensuring that all stored elements share the same data type.

2. Creation of the Array Object: `new DataType[numberOfRows][numberOfColumns]`

This expression creates a two-dimensional array object in memory. The total size of the array is determined by the product: $\text{numberOfRows} \times \text{numberOfColumns}$. Java dynamically allocates the required memory space for the array elements.

3. Assignment of the Reference: “=”

The assignment operator links the reference variable `twoDArray` to the newly created array object. After this assignment, any access through `twoDArray` will operate on that specific two-dimensional array in memory.

For example,

`int[][] twoDArray = new int[3][4];`

This statement creates a two-dimensional array consisting of three rows and four columns, which can be visualized as a table or matrix.

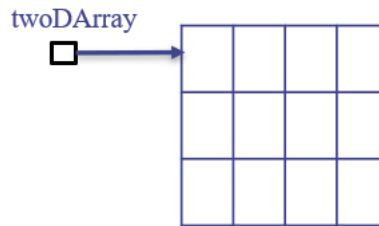


Fig. 5.12: Visualization of 2D array

Elements in a two-dimensional array are accessed using indexing. Each element is identified by two indices: the row index and the column index. Thus, `twoDArray[i][j]` refers to the element in `twoDArray` located at row i and column j . Since indexing starts at 0, the valid range for i is from 0 to `numberOfRows` - 1, and the valid range for j is from 0 to `numberOfColumns` - 1. As you may recall, in Java the `.length` property helps us obtain size information. The following is a summary of the variables, their use, and their meaning in a two-dimensional array.

Table 1: Common Variables and Expressions for Two-Dimensional Arrays

Expression / Variable	Meaning	Notes
<code>twoDArray</code>	The reference variable referring to the two-dimensional array	Declared as <code>DataType[][] twoDArray</code>
<code>twoDArray[i][j]</code>	The element at row i and column j	Valid only when indices are within bounds
<code>twoDArray.length</code>	Number of rows in the two-dimensional array	Always gives the size of the first dimension
<code>twoDArray[i].length</code>	Number of columns in row i	Useful because different rows may have different lengths (jagged arrays)
<code>i</code>	Row index	Valid range: 0 to <code>twoDArray.length</code> - 1

J	Column index	Valid range: 0 to twoDArray[i].length - 1
---	--------------	--

Indexing and Operation

Since elements in a two-dimensional array are indexed by two indices, a nested for loop is a natural choice for iterating over them. The first index corresponds to the row, and the second index corresponds to the column. The code segment below fills the previously generated 3-by-4 twoDArray with numbers produced by a random number generator.

Example 22: Randomly initializing a 2D array using nested for loops

```
public static void main(String[] args){
    int[][] twoDArray = new int[4][3];
    Random rand = new Random();
    // Fill the 4-by-3 array with random numbers
    for (int row = 0; row < twoDArray.length; row++) {
        for (int col = 0; col < twoDArray[row].length; col++) {
            twoDArray[row][col] = rand.nextInt(100); // random 0–99
        }
    }
}
```

To test the code yourself, be sure to import `java.util.Random`, as it is required for generating the random numbers.

Two-dimension arrays passed as parameters

In Java, two-dimensional arrays behave the same way as one-dimensional arrays when passed as method parameters: **they are passed by reference**. This means that the method receives a *copy of the reference* to the same array object. As a result, any modifications made inside the method directly affect the original array.

When a two-dimensional array is passed into a method:

- The **formal parameter** inside the method refers to the **same array object** as the actual parameter in the calling method.
- Any assignment to elements of the array inside the method will also be visible outside the method.
- Both variables act as different “names” pointing to the same underlying memory location.

The following example shows how to fill a 2D array with random values, print it, reset all values to zero, and print it again.

Example 23: Filling, Resetting, and Printing a 2D Array

```
public class TwoDArrayDemo {

    public static void main(String[] args){
        int[][] twoDArray = new int[4][3];
        fillTwoDArray(twoDArray);
        printTwoDArray(twoDArray);
        resetTwoDArray(twoDArray);
        System.out.println("*****");
        printTwoDArray(twoDArray);
    }
}
```

```

// Fills the array with random integers (0–99)
public static void fillTwoDArray(int[][] twoDArray) {
    Random rand = new Random();
    for (int row = 0; row < twoDArray.length; row++)
        for (int col = 0; col < twoDArray[row].length; col++)
            twoDArray[row][col] = rand.nextInt(100);
}
// Resets all elements to zero
public static void resetTwoDArray(int[][] twoDArray) {
    for (int row = 0; row < twoDArray.length; row++)
        for (int col = 0; col < twoDArray[row].length; col++)
            twoDArray[row][col] = 0;
}
// Prints the 2D array in matrix format
public static void printTwoDArray(int[][] twoDArray) {
    for (int row = 0; row < twoDArray.length; row++) {
        for (int col = 0; col < twoDArray[row].length; col++)
            System.out.print(twoDArray[row][col] + "\t");
        System.out.println();
    }
}
}

```

After calling `fillTwoDArray(twoDArray)`, the array is filled with random integers and printed. After calling `resetTwoDArray(twoDArray)`, all elements are set to 0. The final printout (as shown below) shows the updated array with zeros, confirming that the modifications made inside the methods changed to the original array.

98	10	38
52	91	41
60	70	95
82	14	29

0	0	0
0	0	0
0	0	0
0	0	0

Fig. 5.13: Output of Example 23

This occurs because both the `twoDArray` variable defined in main method and the `twoDArray` parameter defined in each of the other methods reference the same array object in memory. In other words, operations performed through any of these reference variables modify the shared underlying array.

To further illustrate the concept, we present an example that computes the sum of the elements in a single row of a two-dimensional array, leaving the column-wise operations as an exercise for the reader. The method receives a two-dimensional array and an integer representing the row index and returns the sum of the elements in that row. A single for loop is sufficient: the row index remains constant, while the column index iterates over each element in the specified row.

Example 24: Sum a row in 2D array

```

public static int sumByRow(int[][] twoDArray, int row) {
    int sum = 0; // Initialize sum accumulator
    // Iterate through each column in the specified row

```

```

    for (int col = 0; col < twoDArray[row].length; col++)
        sum += twoDArray[row][col]; // Add the element to the sum
    return sum; // Return the final sum
}

```

Note on Variable Scope: It is common and perfectly valid to use the same variable name (such as "twoDArray") in both the main method and in the parameter lists of other methods. This works because each variable exists in a different scope:

- The "twoDArray" declared in main is a local variable of the main method.
- The "twoDArray" used as a parameter in another method is a separate local variable within that method's scope.

Even though they share the same name, they do not conflict. Java resolves variable names based on scope: each method sees only its own local variables and parameters.

When the array is passed to a method, a copy of the reference is passed. Therefore, both variables (the one in main and the one in the method) refer to the same underlying array objects in memory. Any changes made through either reference will modify that object.

If the data type is a reference type rather than a primitive type, operations behave similarly to those on a one-dimensional array. Additionally, it is important to ensure that the instance exists before attempting to access its fields or methods.

Multi-dimensional Arrays

Having explored two-dimensional arrays, let's now move on to general multidimensional arrays. In our previous examples, we primarily used two-dimensional arrays in a matrix format, where each row has the same number of elements, determined by the number of columns. In Java, multidimensional arrays are implemented as **arrays of arrays**. For example, a 3-by-4 two-dimensional array can be implemented as a one-dimensional array with 3 elements, where each element in the first dimension is itself an array of 4 elements. With this approach, the number of elements in each row can be customized, although this is less commonly used in practice.

Example 25: 2D array implemented as an array of arrays (jagged array)

```

int[][] twoDArray = new int[3][]; // 3 rows, column sizes unspecified
twoDArray[0] = new int[2]; // Row 0 has 2 columns
twoDArray[1] = new int[3]; // Row 1 has 3 columns
twoDArray[2] = new int[4]; // Row 2 has 4 columns

```

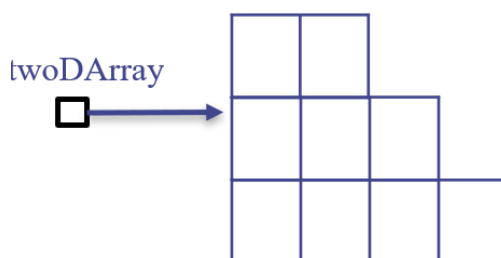


Fig. 5.14: Jagged 2D Array

Using the concept of arrays of arrays, we can create arrays with more than two dimensions. Below are two examples showing how to create both rectangular and jagged 3×3×3 three-dimensional arrays.

Example 26: Rectangular 3D array

```
int[][][] threeDArray=new int[2][2][3]
```

Example 27: Jagged 3D array

```
int[][][] threeDArray = new int[2][][]; //2 layers, columns unspecified
// Initialize jagged layers
threeDArray[0] = new int[2][3];
threeDArray[1] = new int[2][];
threeDArray[1][0] = new int[3];
threeDArray[1][1] = new int[3];
```

The java.util.Arrays Utility Class

Having explored the fundamentals of arrays, it is important to recognize that Java provides a set of built-in tools to simplify common array operations. The `java.util.Arrays` utility class, which is a member of the Java Collections Framework, offers a collection of static methods for tasks such as sorting, searching, comparing, and converting arrays to strings. By leveraging the `Arrays` class, we can write cleaner and more efficient code without manually implementing common array operations. The example below demonstrates how to use `Arrays` to sort an array. Be sure to include `import java.util.Arrays;` when testing the code. For more details, you can refer to the official documentation: [Java Arrays](#)

Example 28: Jagged 3D array

```
int[] numbers = {5, 2, 9, 1, 7};
// Sort the array in ascending order
Arrays.sort(numbers);
System.out.println("Sorted array: " + Arrays.toString(numbers));
```

The Hidden World Inside Arrays

Now that you understand what arrays are and how to use them, let's take a closer look at how they work behind the scenes.

When an array variable is created, Java allocates space in main memory (RAM) to store all of the elements. One important property of arrays is that all elements are stored in consecutive (contiguous) memory locations. This means the entire array occupies one continuous block of memory. The amount of memory required for an array depends on two factors:

1. **The data type of the elements**

Each element has a fixed size based on its data type. For example, an *int* uses 32 bits (4 bytes), a *double* uses 64 bits (8 bytes), and a *char* uses 16 bits (2 bytes)

2. **The number of elements in the array**

The number of elements defines the maximum number of values that can be stored in the array. This is commonly referred to as the size or length of the array, and it cannot be changed once the array is created.

The total memory needed for an array can be calculated as:

$$\text{array length} \times \text{size of Element}$$

For example, for the previous studentID array that we have created. Each element is an *int* occupies **32 bits (4 bytes)**. The total memory needed is: $100 \times 4 \text{ bytes} = 400 \text{ bytes}$

The JVM (Java Virtual Machine) will search for a 400-byte consecutive block of free memory to store this array. Because arrays require contiguous space, the allocation will only succeed if such a block is available. If no sufficiently large contiguous block can be found, memory allocation fails, and the program may throw an **OutOfMemoryError**. This is also the reason why an array is a fixed-length structure and cannot be resized after creation—expanding it would require additional consecutive memory, which may already be occupied by other data or instructions.

Once memory is allocated to the array, the physical memory address of the first position is stored in the array variable. For any other position in the array, we can obtain its physical memory address by using:

$$\text{address of first position} + (\text{element size} \times \text{index})$$

Because of this, accessing an element in an array is very fast, which is referred to as **constant time (O(1))**. This is one of the advantages of using an array as a storage structure. It also explains why array indices start at 0: starting from 0 makes it easy for the computer to use the index directly to calculate the memory address of each element.

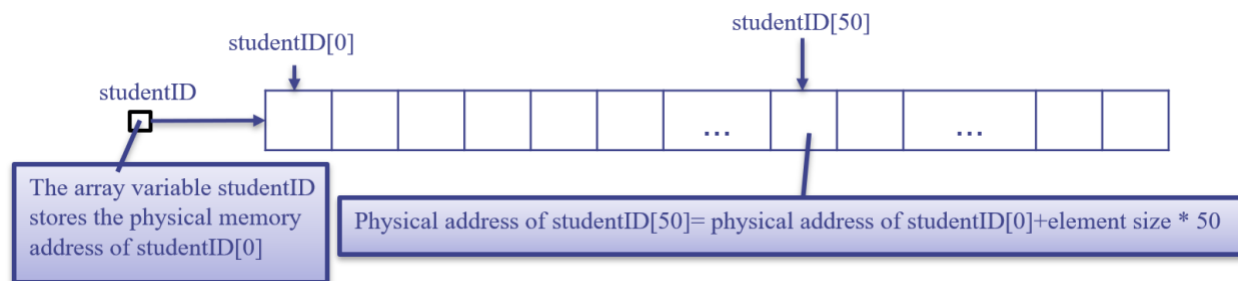


Fig. 5.15: Accessing an array element in constant time by directly locating it in memory

Now that we have seen how memory is calculated for arrays of primitive types, things become more interesting when working with arrays of objects. Remember that an object is an instance of a class created at runtime, and its memory is allocated differently from primitive types.

For example, consider:

```
Student[] studentList = new Student[100];
studentList[0] = new Student(0, "Anna");
```

As discussed before, this statement creates an array of 100 **reference variables** (`studentList[0]`, `studentList[1]`, ...), each of which can refer to a `Student` object. Consecutive memory space is allocated to store these 100 reference variables. If a `Student` object is created later and `studentList[0]` is assigned to refer to it, the reference stores the physical memory location of the object.

Accessing an element in an array containing reference types works as follows

`studentList` -> `studentList[0]` -> `Student` object "Anna"

This access is still constant time (O(1)). However, different objects created at different times are not stored in consecutive memory locations, unlike primitive type arrays.

Consecutive memory spaces are also allocated for multidimensional arrays. For example, for a two-dimensional 2×4 array, $2 \times 4 \times \text{element size}$ units of memory will be allocated. The array variable stores

the physical address of the first element. For an element with row index i and column index j , its physical address can be calculated as:

$$\text{address of element}[i][j] = \text{address of first element} + ((i * \text{number_of_columns}) + j) * \text{size_of_element}$$

This formula allows the computer to quickly compute the memory location of any element in the array, which occurs in constant time ($O(1)$).

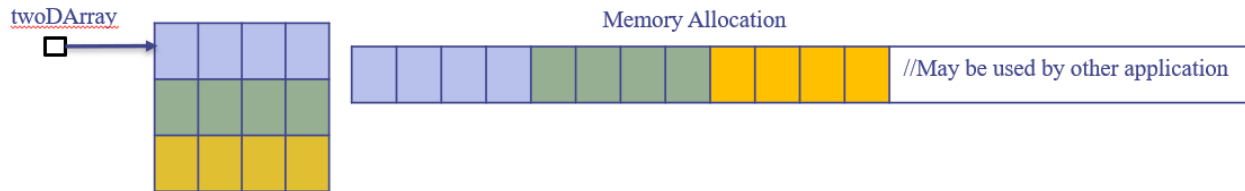


Fig. 5.16: Visualization of two-dimensional memory allocation

Although the memory for a multidimensional array is allocated as a contiguous linear block, the array is logically organized in multiple dimensions. This means that, physically, all elements are stored consecutively in memory, but we access them using multiple indices to represent rows, columns, and higher dimensions.

Chapter 6: Inheritance

What is Inheritance?

Imagine you are building a system for an institution, and the system has different types of users with different behaviors, such as undergraduate student, graduate student, staff, and faculty. Without inheritance, each class would need to repeat the same basic fields—such as name, id, and age—and many of the same methods, like `getName()` or `printInfo()`. For example, both `UndergraduateStudent` and `GraduateStudent` would contain identical code for storing student information, and `Staff` and `Faculty` would duplicate the same employee-related code. This leads to duplicated work, makes the code harder to maintain, and increases the likelihood of errors. Inheritance solves this problem by allowing common code to be written once in a superclass and shared automatically with all related subclasses.

Basics

Inheritance is one of the most effective mechanisms provided by Java to help us reuse existing code. It allows us to create a new class based on an existing class, so the new class automatically gets the fields and methods of the original one. A class that is derived from another class is called a **subclass** (aka a derived class, extended class, or child class). The class from which the subclass is derived is called a **superclass** (aka a base class or parent class). This creates an “is-a” relationship between the derived class and the existing class. Because of this, an object created from a subclass is also considered an object of its parent class.

For example, in a school system, you might have `Faculty` and `Staff` classes that both inherit from an `Employee` superclass. A faculty member is an `Employee`, and a staff member is also an `Employee`. The `Employee` class acts as a blueprint, defining common characteristics and behaviors that all employees share. The subclasses reuse this information through inheritance, and more specific customizations can be added within each subclass.

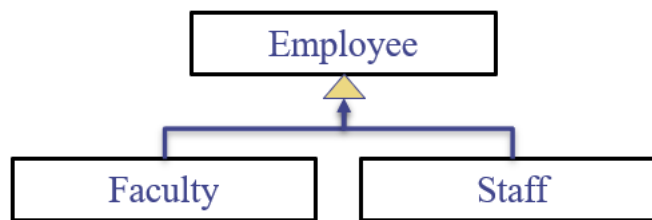


Fig. 6.1: Employee Inheritance Diagram

In Java, the keyword **extends** is used to create a subclass from a superclass.

Syntax : `class SubclassName extends SuperclassName {`
 // additional fields and methods
`}`

Example 1: Simple Employee, Faculty and Staff Implementation

```

public class Employee{
    public String name;
    protected String address;
    private String id;
    String phone;

    public Employee(String name, String id) { //defined in Employee class
        this.name = name;
    }
}
  
```

```

        this.id = id;
    }
    public String getId() {
        return id;
    }
    public void setId(String id) { //defined in Employee class
        this.id = id;
    }
    public String toString() {
        return id+ ", " +name + ", " + address + ", " + phone;
    }
    public class Faculty extends Employee {
        protected double annualSalary;
        public Faculty(String name, String id, double annualSalary){
            super(name, id);
            this.annualSalary= annualSalary;
        }
    }
    public class Staff extends Employee{
        protected double monthlySalary;
        public Staff(String name, String id, double monthlySalary) {
            super(name, id);
            this.monthlySalary=monthlySalary;
        }
    }
}

```

In this example, the Employee class defines information that **all employees** share, such as their name, address, and phone number. It also includes a toString() method that returns the employee's details in a simple text format. Two subclasses—Faculty and Staff—are created using the extends keyword. This means they **inherit all fields and methods** from the Employee class. Faculty inherits name, address, phone, and toString() from Employee. Staff also inherits the same fields and method. Each subclass adds its own specific field: Faculty adds annualSalary, and Staff adds monthlySalary. This avoids repeating the common employee information in each class.

UML Class Diagram (for Inheritance)

A **UML (Unified Modeling Language) class diagram** is a type of diagram used to visually represent the structure of a program in object-oriented programming. Since inheritance usually involves multiple classes, UML class diagrams are commonly used to show the classes and relationships between them. A UML class diagram represents a class using a rectangle divided into three sections: class name, attributes and methods. A hollow arrow is used to indicate inheritance. The UML class diagram for the Employee example is shown below.

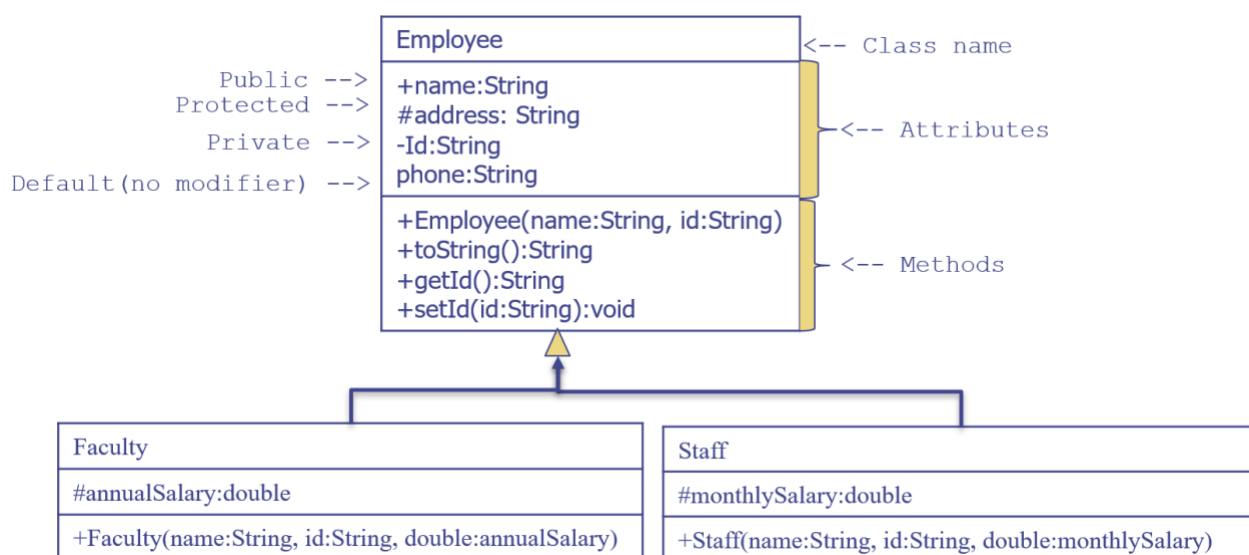


Fig. 6.2: Employee, Faculty, and Staff Inheritance UML Class Diagram

How Subclasses Inherit Variables and Methods

Note: When a subclass extends a superclass, it inherits all fields (variables) and methods from the superclass. Access modifiers determine which members the subclass can see and use directly.

Understanding Access Modifier for variables and methods

In the example shown in Fig. 6.2, you may notice the symbols that represent Java's access modifiers. Java provides four access modifiers- public, protected, private, and default (no modifier). As the name suggests, those are reserved keywords that define how accessible a class, method, or variable is within a program.

You will encounter access modifiers in class definitions, method definitions, and variable declarations. They play an important role in controlling how different parts of a program interact with one another and help prevent unintended access or modification of data.

Although the same keywords are used, the effect of each modifier differs slightly depending on whether it is applied to a class, method, or variable. The following tables summarize the access levels and behaviors of each modifier when used with variables and methods. You will explore each modifier in greater detail, with examples and illustrations, throughout the remainder of this chapter.

- Table 6.1: Access Modifiers for Variables (Fields)**

Modifier	Allowed for Variables?	Accessible From
public	✓	Anywhere in the program
protected	✓	Same package + subclasses (even in different packages)
default (<i>no modifier</i>)	✓	Only within the same package
private	✓	Only within the same class

- Table 6.2 Access Modifiers for Methods**

Modifier	Allowed for Methods?	Accessible From
Public	✓	Anywhere in the program

Protected	✓	Same package + subclasses (even in different packages)
default (<i>no modifier</i>)	✓	Only within the same package
Private	✓	Only within the same class

Inheritance of Fields (Variables)

When a subclass extends a superclass, it automatically inherits all of the superclass's fields. However, the access modifier of each field determines whether the subclass can use that field directly. As shown in Table 6.1, a subclass can access any inherited field directly by name if it is not declared as private.

Private fields are still inherited, but the subclass cannot access them directly because they are visible only inside the class where they were declared. If the superclass provides a public method—such as a getter or setter—that works with a private field, then the subclass can access the field indirectly by calling that method. This idea will be shown in the next subsection when you look at the inheritance of methods.

Figure 6.3 shows an example where the subclass can directly access all non-private fields from the Employee superclass but attempting to access the private field `phone` results in a compile-time error. Figure 6.4 continues the example by creating a Faculty object in a separate class. When using the dot operator to access the object's fields, only non-private fields are visible and accessible.

```

4 public void testAvaivableAccess() { //defined in Faculty class
5     System.out.println(name);
6     System.out.println(address);
7     System.out.println(id);
8     System.out.println(phone);
9 }

```

Fig. 6.3: Private variables are not directly accessible in the subclass

```

public static void main(String[] args){
    Faculty f= new Faculty();
    f.

```

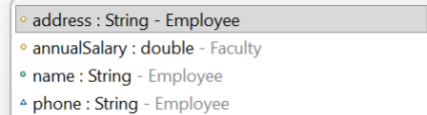


Fig. 6.4: Non-private variables are visible and can be accessed directly by name

Inheritance of Methods

After seeing how a subclass inherits variables from its superclass, it is important to understand that methods are also inherited in a similar way. When a subclass extends a superclass, it automatically inherits all the superclass's methods. Whether a subclass can use an inherited method directly depends on the method's access modifier. As shown in Table 6.2, a subclass can call any inherited method by name if it is not declared private.

Figure 6.5 illustrates how a private `phone` field defined in the Employee superclass can be accessed and printed using a public getter method. Because the getter is public, the Faculty class inherits it and can invoke it directly. Moreover, since the getter method is defined within the Employee class, it can access the private `phone` field directly, even though the field itself is not accessible to the subclass.

```

public void testMethodAccess() { //defined in Faculty class
    System.out.println(getId());
}
public String getId() { //defined in Employee class
    return id;
}

```

Fig. 6.5: Accessing a private variable indirectly through an inherited public method

Method Overriding

In the previous subsections, you learned how a subclass inherits fields and methods from its superclass. As a regular class, a subclass can also define new fields and methods to customize its behavior as needed. In addition, Java supports methods overriding, which allows a subclass to modify or provide a specific implementation for a method defined in the superclass.

Method overriding enables a subclass to redefine a method originally defined in the superclass, subject to the following rules:

- The method signature must remain the same, including method name, parameters, and return type.
- The access modifier of the overriding method cannot be more restrictive than that of the method in the superclass.
- The return type can be a subclass type (covariant return type), because a subclass object is also considered an instance of the superclass.

As an example, you will use the `toString` method implemented in the `Employee` class. The code below shows its implementation. This method returns the employee's information as a `String` and defines the format used when calling `System.out.print()` or `System.out.println()`. Since `toString` is a public method, the `Faculty` class inherits it from the `Employee` class and can call it directly by name, as illustrated in Fig. 6.6.

Example 2: `toString()` defined in `Employee` class

```

public String toString() { //defined in Employee class
    return id + ", " + name + ", " + address + ", " + phone;
}

public static void main(String[] args) {
    Faculty f = new Faculty("Anna", "123");
    f.t

```

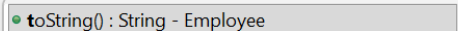


Fig. 6.6: `Faculty` object inheriting the public `toString` method from the `Employee` class

Example 3: Test Case and Output

```

//separate test class with main method
Faculty f = new Faculty("Anna", "123", 100000.00);
System.out.println(f);
//output: 123, Anna, null, null

```

When you add a new field, such as `annualSalary`, to the `Faculty` class, the inherited `toString` method from the `Employee` class does not include this information. To display the `annualSalary` as part of the output, you can override the `toString` method in the `Faculty` class. The new method must have the same method signature as the one in the `Employee` superclass.

Example 4: `toString()` defined in `Faculty` class

```
public String toString() { //defined in Faculty class
return id+ ", " +name + ", " + address + ", " + phone + ", " + annualSalary;
}
```

The Faculty class now provides its own implementation of the toString method. When toString is called on a Faculty object, Java executes the overridden method in the subclass rather than the version inherited from Employee (see Fig. 6.5). This allows us to customize the implementation of the inherited method to include subclass-specific details.

```
public static void main(String[] args){
    Faculty f= new Faculty("Anna", "123");
    f.t
    toString() : String - Faculty
```

Fig. 6.7 Non-private variables are visible and can be accessed directly by name

If you repeat the previous example test case, the output will include annual Salary.

Example 5: Test Case and Updated Output

```
//separate test class with main method
Faculty f= new Faculty("Anna", "123", 100000.00);
System.out.println(f);
//output: 123, Anna, null, null, 100000.0
```

Note: Java also allows **method overloading**, which means a class can have multiple methods with the same name but different parameters. This lets a class perform similar tasks in different ways. One common example is that a class can have several constructors, each designed to initialize the object in a different way.

In a subclass, you can both:

- **Override** an inherited method to give it a new behavior.
- **Overload** a method to create additional versions with different parameters.

Super Reference

Now that you understand how a subclass inherits variables and methods from a superclass, and how you can customize inherited methods through method overriding. There is one more important feature to look at: constructors.

A constructor is a special method that runs when an object is created. Its job is to set up (initialize) the object's fields. Every constructor must have the same name as its class. For example, the Employee class has a constructor named Employee(...), and the Faculty class must have a constructor named Faculty(...).

Because a constructor must match the class name, a subclass cannot inherit its superclass's constructor the same way it inherits other methods. This means the Faculty class cannot directly reuse the Employee constructor, even though it still needs to initialize the fields defined in Employee.

So what should you do?

Java provides a solution using the reserved keyword **super**. In Java, super is a special keyword that refers to the superclass of the current object. You can think of it as a way for a subclass to talk to or use features from its parent class. Inside a subclass constructor, you can write super(...) to call the superclass constructor. The super reference is used for two main purposes:

- Calling the Superclass Constructor
- Accessing Superclass Methods or Variables

Calling the Superclass Constructor with `super(...)`

In Java, a subclass can use **`super(...)`** inside its constructor to call the constructor of its superclass. This lets the subclass reuse the code that initializes the fields defined in the superclass.

The example below shows the constructor in the `Employee` class and how the `Faculty` class can reuse it by calling `super(...)`. You pass the needed arguments just like a normal method call. The only difference is that instead of writing the superclass's constructor name, you simply use the keyword `super`.

Example 6: Use `Super` to call Constructors

```
public Employee(String name, String id) { //defined in Employee class
    this.name = name;
    this.id = id;
}
public Faculty(String name, String id, double annualSalary) { //defined in Faculty class
    super(name, id);
    this.annualSalary = annualSalary;
}
```

Using `super` to Access Superclass Methods and Variables

Usually, a subclass can use non-private methods from its superclass directly by name. But if the subclass overrides a method, calling it by name will use the subclass version instead of the one in the superclass (as shown in Examples 2 to 5). In this case, you can use **`super`** to call the method or access a variable from the superclass.

For example, in Example 5, the `Faculty` class overrides the `toString` method. This new method repeats what the `Employee` class's `toString` does and then adds the `annualSalary`. You can make it simpler by using **`super`** to reuse the superclass method, like this:

Example 6: Using `super.toString()` to call the superclass method

```
public String toString() { //defined in Faculty class
    return super.toString() + ", " + annualSalary;
}
```

Here, `super.toString()` calls the `toString` method from `Employee`, and then you add the `annualSalary` for the `Faculty` object. This avoids repeating code from the superclass and makes it easier to implement the overriding method when it builds on the behavior defined in the superclass.

Note: If a method is declared with the **`final`** modifier, it cannot be overridden

Shadow variables

Overriding methods in a subclass is common in Java. However, if a subclass defines a variable with the same name as one in its superclass, it can create confusing behavior. This situation is called a **shadow variable**.

A shadow variable in the subclass “hides” the variable with the same name in the superclass. Here's what this means:

- When you access the variable by name inside the subclass, Java uses the subclass's **version** (the shadow variable).
- But methods inherited from the superclass still use the superclass's variable, because those methods were written in the superclass and work only with the variables defined there.

Because of this, it's important to understand which variable your code is actually using.

To see how this works, in Example 7, the Faculty class below has been modified to introduce a shadow variable named `id`, and the `getId` method has been overridden.

Example 7: Modified Faculty class with shadow variable

```
public class Faculty extends Employee {
    protected double annualSalary;
    private String id; //shadow variable
    public Faculty(String name, String id, double annualSalary)
        super(name, id); // initializes the Employee version of id
        this.annualSalary = annualSalary;
    }
    public String toString() {
        return super.toString() + ", " + annualSalary;
    }
    public String getId() {
        return id; //access shadow variable
    }
}
```

Below is a test case showing how shadowing affects output.

Example 8: Test Case and Output

```
//separate test class with main method
Faculty f = new Faculty("Anna", "123", 100000.00);
System.out.println(f);
System.out.println(f.getId());
//output: 123, Anna, null, null, 100000.0
//output:null
```

In this example, When the Faculty class introduces a shadow variable named `id`, the subclass ends up holding two different `id` variables—one in `Employee` and one in `Faculty`. This leads to confusing behavior.

- **Constructor Execution:** The Faculty constructor calls `super(name, id)`, which runs the `Employee` constructor. The `Employee` constructor initializes the `Employee` version of `id` with the value "123". The `Faculty` version of `id` (the shadow variable) is never initialized and therefore keeps its default value, `null`.
- **Calling `System.out.println(f)`:** This activates the overridden `toString()` method in the `Faculty` class. Inside that method, the call to `super.toString()` uses the `Employee` version of the variables, including the correctly initialized `id`. As a result, the output includes "123" along with the other `Employee` fields: **123, Anna, null, null, 100000.0**
- **Calling `System.out.println(f.getId())`:** This activates the **overridden `getId()` method** in `Faculty`. The method returns the **Faculty version** of `id`—the shadow variable. Since this variable was never initialized, the output is: **null**

This example shows why shadow variables can cause unexpected behavior—they split the data into two places, depending on whether the code runs in the superclass or the subclass

Note: Shadow variables are strongly **discouraged** in Java because they make code confusing and error-prone. In almost all cases, you should avoid using shadow variables.

Class Hierarchy

The Java Platform Class Hierarchy

In Java, every class is part of a **class hierarchy**, which is a way of organizing classes based on inheritance. At the top of this hierarchy is the `Object` class, defined in the `java.lang` package, defines and implements behavior common to all classes—including the ones that you write. This means that every class in Java, whether it is a standard library class or a class you define, ultimately inherits from `Object`. In Java, every class can only have one superclass.

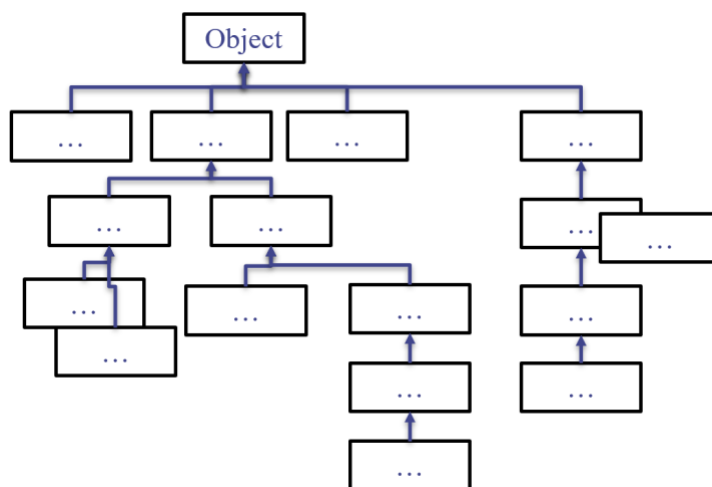


Fig. 6.8: Java Platform Class Hierarchy

As you saw earlier, the **extends** keyword is used in a class declaration to show that a class is a subclass of another class (its superclass). If a class does not explicitly extend any class, Java automatically makes it a subclass of the `Object` class. This means **every class has a parent class** — either one you write using `extends` or the `Object` class that Java adds for you. Figure 6.9 shows the updated inheritance structure for Example 1.

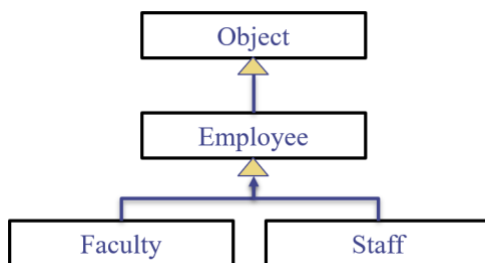


Fig. 6.9: Inheritance hierarchy for the Employee Application

In this hierarchy, the `Employee` class inherits all the variables and methods from `Object`. Then the `Faculty` class inherits everything from `Employee`. Because of this, code inside the `Faculty` class can:

- use the variables and methods defined in `Faculty` itself
- directly access all **non-private** variables and methods defined in `Employee`
- directly use all **non-private** methods inherited from `Object`

Figure 6.10 shows examples of what can be accessed by name inside the `Faculty` class.

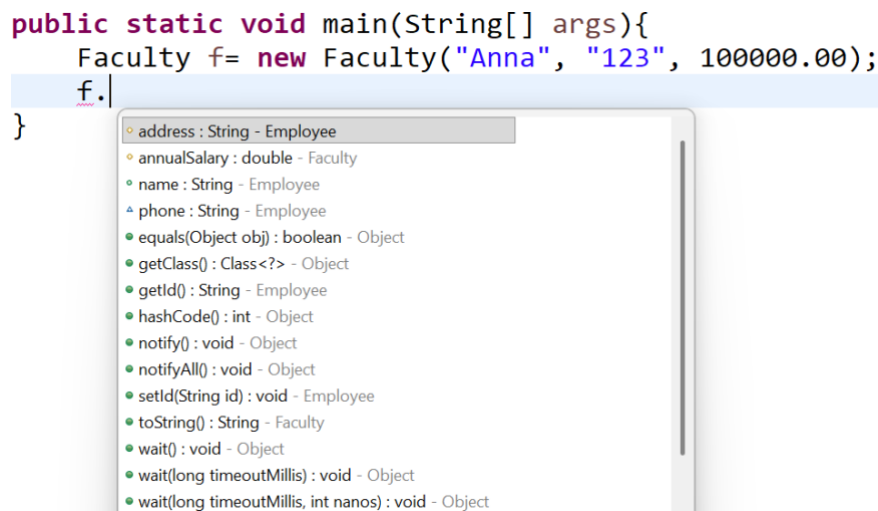


Fig. 6.10: Variables and methods accessible by name in the Faculty class

Frequently Overridden Methods from the Object Class

Since every Java class ultimately inherits from the `Object` class, all classes automatically receive several basic methods. Many of these default implementations are simple and generic, so subclasses often override them to provide more useful or meaningful behavior. Below are the most common `Object` methods that programmers override.

- **toString()**

The `toString()` method in the `Object` class returns a memory-based representation of an object, which is usually not very helpful. For example, if you remove the `toString()` methods overridden in the `Employee` and `Faculty` classes, calling: `System.out.println(f)` would print something like `Faculty@24d46ca6`—just the class name followed by a random number.

To make the output readable and meaningful, you override the `toString()` method in subclasses. This allows the method to return a string that clearly represents the object, such as its name, id, and other important information as shown in Examples 2, 4, and 6.

- **equals(Object obj)** The `equals(...)` method in the `Object` class checks whether two references point to the same object in memory. It returns `true` if they do, and `false` otherwise. In most cases, you want to compare objects based on their data instead of memory location. For example, you may want to check whether two `Employee` objects have the same id and name, even if they are different objects in memory.

By overriding `equals(...)` in a subclass, you can compare two objects for logical equality, based on the fields you care about. Below is a recommended template that can be used to override `equals(...)` in `Employee` class.

Example 9: Override equals() in Employee class

```
public boolean equals(Object obj) {
    if (this == obj) {
        return true; // Check if comparing the object to itself
    }
    if (obj == null || getClass() != obj.getClass()) { //Check if obj is null or a different class
        return false;
    }
    Employee other = (Employee) obj; //cast obj to Employee
```

```
return this.id==other.id; // Compare important fields
}
```

- *hashCode()*

Java uses the `hashCode()` method when storing objects in hash-based collections such as `HashMap`, `HashSet`, and `Hashtable`. If a class overrides `equals(...)` but does not override `hashCode()`, then two objects that are logically equal may not behave correctly in these collections. To avoid this problem, any time `equals(...)` is overridden, `hashCode()` should also be overridden so that objects considered equal also produce the same hash code.

Paired with the overridden `equals(...)` method in Example 9, the following is a recommended implementation of `hashCode()`:

Example 10: Override `hashCode()` in `Employee` class

```
public int hashCode() {
    return Objects.hash(id); // the same fields used in equals()
}
```

Inheritance and Subclass Constructors

With all the code reuse benefits provided by inheritance, it's also important to understand how inheritance affects subclass constructors.

When a subclass is created, it may need to call a constructor from its superclass:

- If the superclass has a no-argument constructor, the subclass does not need to define its own constructor, as long as it has no additional fields to initialize.

Example 11: Class with no Constructor

```
public class Animal {
    //variables and methods
}
```

You can create an instance of `Animal` like this: `Animal a = new Animal();` This works without any errors. That's because Java automatically adds a **default constructor** during execution:

Example 12: Automatically added constructor

```
public Animal() {
    super();
}
```

This default constructor ensures that the class can be instantiated even if you didn't explicitly write a constructor.

- If the superclass does not have a no-argument constructor, the subclass must define a constructor. In this case, the first statement in the subclass constructor must be a `super(...)` call to one of the superclass's constructors. Otherwise, a compile-time error will occur. This ensures that all fields in the superclass are properly initialized when a subclass object is created.

An example is shown in **Example 13**. Since all constructors defined in the `Employee` class require arguments, the constructors for `Faculty` and `Staff` cannot be omitted. At the same time, the first statement in the `Faculty` or `Staff` constructor must be the appropriate `super(...)` call. Changing its position or skipping it will cause a compile-time error.

Class Visibility and Inheritance

In Java, access modifiers determine which classes can access other classes. This is also known as class visibility. For top-level classes (classes that are not nested inside another class), only some modifiers are allowed. Table 6.3 summarizes the rules:

Table 6.3: Access Modifiers for Classes

Modifier	Allowed for Classes?	Accessible From
public	✓	Any class in any package
default (<i>no modifier</i>)	✓	Only within the same package
protected	✗	Not allowed for top-level classes
private	✗	Not allowed for top-level classes

As shown in the table,

- a public class can be used by any other class, regardless of package.
- A default class (no modifier) is only visible to other classes in the same package.
- Protected and private cannot be applied to top-level classes; they are only used for members inside a class.

Choosing the right class visibility is important for **inheritance**, because a subclass must be able to **see its superclass** to extend it.

Abstract Classes

In Java, an **abstract class** is a special type of class that is meant to be inherited (extended) but not instantiated (created) directly. You declare an abstract class using the ***abstract*** keyword in the class header. Abstract classes are useful when you want to define common features for a group of subclasses, while leaving some details for the subclasses to implement. In particular, you do not want instances of the superclass itself to exist.

The key features of abstract class can be summarized as:

1. An abstract class can have constructors

Even though you cannot create an object of an abstract class (meaning its constructor cannot be called using the new operator), its constructors can be called from a subclass constructor using a `super(...)` call. This allows fields defined in the superclass to be initialized through the call of the abstract class's constructor.

2. An abstract class can have fields and regular methods

Just like a normal class, it may contain instance variables, non-abstract (fully implemented) methods. These are automatically inherited by its subclasses.

3. An abstract class can contain abstract methods

An **abstract method** is a method that is declared using the abstract keyword and does not have a body. For example: `public abstract void draw();` This means the method defines what a subclass should do, but does not say how to do it. Subclasses that inherit this method must provide their own implementation. Notice that an abstract method ends with a semicolon instead of a method body.

4. If a class has even one abstract method, the class must be declared abstract

This rule makes sure that you cannot create objects from a class that has unfinished methods.

5. Subclasses must implement inherited abstract methods, or be abstract

If a subclass inherits an abstract method, it must provide its own implementation by removing the abstract keyword, replacing the semicolon “;” with a method body using curly braces “{ }”, and adding the appropriate code inside the method.

If the subclass does not implement the abstract method, then the subclass itself must also be declared abstract. This rule also prevents you from creating objects from a subclass that still have unfinished methods (inherited from superclass).

You can think of an abstract class as a partial blueprint: it provides shared fields and methods but may leave some methods “unfinished” for its subclasses to complete.

The Employee application is a perfect example. In your school system, you have faculty and staff members. They share many common features, so it makes sense to put all these common features in the Employee superclass. In practice, your application allows Faculty and Staff instances to be created and used, but you do not want an Employee instance to be created directly, as this could cause confusion. In this situation, the Employee class can be defined as an abstract class.

Furthermore, every employee needs to be paid, so you want to add a common method `pay()` in the Employee class. However, since faculty contracts are based on annual salary and staff contracts are based on monthly salary, these two types of employees require different implementations. This makes `pay()` a perfect candidate for an abstract method.

Based on these considerations, the Employee, Faculty, and Staff classes are modified as follows.

Example 13: Abstract Employee Class and its subclasses

```
import java.util.Objects;

public abstract class Employee{
    public String name;
    protected String address;
    private String id;
    String phone;

    public Employee(String name, String id) { //defined in Employee class
        this.name = name;
        this.id = id;
    }

    public Employee(String name, String address, String id, String phone) {
        this.name = name;
        this.address = address;
        this.id = id;
        this.phone = phone;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
}
```

```

public abstract double pay();

public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }
    Employee other = (Employee) obj;
    return this.id==other.id;
}

public int hashCode() {
    return Objects.hash(id);
}

public String toString() {
    return id+ ", " +name + ", " + address + ", " + phone;
}
}

```

```

public class Faculty extends Employee {
    protected double annualSalary;

    public Faculty(String name, String id, double annualSalary) {
        super(name, id);
        this.annualSalary= annualSalary;
    }

    public Faculty(String name, String address, String id, String phone, double annualSalary) {
        super(name, address, id, phone);
        this.annualSalary= annualSalary;
    }

    public double pay() {
        return annualSalary/12;
    }

    public String toString() {
        return super.toString()+ ", " + phone;
    }
}

```

```

public class Staff extends Employee{
    protected double monthlySalary;
}

```

```
public Staff(String name, String id, double monthlySalary) {  
    super(name, id);  
    this.monthlySalary=monthlySalary;  
}  
  
public Staff(String name, String address, String id, String phone, double monthlySalary) {  
    super(name, address, id, phone);  
    this.monthlySalary=monthlySalary;  
}  
  
public double pay() {  
    return monthlySalary;  
}  
}
```

In a UML class diagram, abstract classes and abstract methods are shown in *italics* to indicate that they are abstract.

Chapter 7: Polymorphism

Scenario: You are creating a program that reads in a university's student/faculty/staff information from a file. There is a need for a Student class, a Faculty class, and a Staff class as the type of data stored for a student will be different from a faculty or staff member. Each line of the input file contains data for a single person. However, you have no idea in advance what the next type of personnel data will be on the next line. Will it be a student, faculty, or staff member?

This can cause a problem with program design as the program can't simply read all the data in and assign it to a generic object. What type of object needs to be stored can't be determined until run-time when the program can determine what type of person is on the next line of input.

Situations such as this are where polymorphism is very useful. The simple definition for **polymorphism** is the ability of an object to be assigned one of many forms. If you break the word down, "Poly" means multi and "Morph" means to change. Combine the two words, it means that an object can change or take on one of many forms. This is most commonly achieved through inheritance, but it can also be achieved through interfaces. This gives a program more flexibility and reusability. Given the problem introduced, you could create a simple Person object and then decide later at run-time what type of person object needed to be created (student, faculty, or staff member).

7.1 Types of polymorphism

1. Compile-time polymorphism: Also known as static polymorphism, this is achieved with **method overloading**. The compiler must decide at compile time which version of a method to use based on the parameters supplied to the method. The process of determining which version of a method to use at compile time is known as **early binding** or static binding.

An example of this is the `valueOf()` method of the `String` class.

`String.valueOf(3.49)`, `String.valueOf(true)`, `String.valueOf('a')` all use different parameters and the compiler must determine which version of the method to use based on the data provided the method.

- a. `String.valueOf(3.49)` would require the `valueOf` version of the method that takes a single double parameter.
 - b. `String.valueOf(true)` would require the boolean parameter version of the method.
 - c. `String.valueOf('a')` would require the character parameter version of the method.
2. Run-time polymorphism: Also known as dynamic polymorphism, this is achieved with **method overriding**. At runtime, the method to be used must be determined based on the current form of the object. **Late binding** or dynamic binding describes the process of determining which specific method to use at runtime.

For example: Assume there is an `Employee`, `Manager`, and `Supervisor` class. Also assume that `Manager` and `Supervisor` are subclasses of `Employee`. All 3 classes have a `calculateOvertime()` method. All company personnel are employees and

therefore can be created as Employee objects, but they can also be cast to a different class. The Java virtual machine must determine which calculateOvertime() method should be used when the Employee object uses the method.

```
Employee employee123= new Employee("Scott", "Spetka");
Employee employee456= new Manager("Jorge", "Novillo", "Accounting");
Employee employee789= new Supervisor("Sam", "Sengupta", "IT", 34);
Employee123.calculateOvertime(); //this uses calculateOvertime from Employee class
Employee456.calculateOvertime(); //this uses calculateOvertime from Manager class
Employee789.calculateOvertime(); //this uses calculateOvertime from Supervisor class
```

For the duration of this chapter, we will be exploring dynamic polymorphism. A polymorphic reference variable can also have different forms throughout the lifecycle of a program.

Consider the following code snippet in which each time through the for loop the object variable can be assigned/reassigned as a Bird or Parrot. Every time through the loop a call to the method birdNoise() is called. Assuming there is a birdNoise() method in both the Bird and Parrot classes, the Java virtual machine must determine which noise to make based on the current state of the object.

```
Bird current_bird;
String bird_name, bird_type;
for(int counter=0; counter<15; counter++)
{
    System.out.print("What is the name of the bird? ");
    bird_name=scan.nextLine();
    System.out.print("Is the bird a parrot?(Y/N) ");
    bird_type=scan.nextLine();
    if (bird_type.equalsIgnoreCase("Y"))
        current_bird=new Parrot(bird_name);
    else
        current_bird=new Bird(bird_name);
    current_bird.birdNoise(); //what noise does the current bird make?
}
```

7.2 When to use polymorphism

Polymorphism is best used when the final form of an object cannot be determined until run-time. The most common situation for this is when data is unknown in advance. Polymorphism allows flexibility for a program to handle data in a generic way and then at run time determine which subcategory the data belongs to.

Here are some examples of when to use polymorphism:

- A car dealership that needs to create a list of automobiles in inventory. There are common features that make all of them automobiles like number of doors and automobile length, but then there are characteristics that make something uniquely a car, truck, or SUV. For instance, a truck will have a cargo bed, but a car will not. Every car object can be created as Car, but later each vehicle could be instantiated to the proper class type.
- Creating a group of computer objects. There are common features that all computers have such as processors, RAM memory, and weight. However, there are also subcategories such as desktop, laptop, and tablet that would have unique characteristics.

They are all computers, but assigning a computer to a subcategory at run-time is more code efficient.

- Reading multiple patient stats from a file at a medical office. The basics like age, weight, height, gender would all be common but then based on medical conditions additional information may need to be collected. For example, a diabetic patient would have information pertaining to the type of diabetes, blood sugar levels, and diabetic medicine the patient is on. This information would not be collected for a patient that does not have diabetes. This is something that the program can do based on the data it is receiving at run time.

Example 7.1

Consider the following example: A company needs to assign an employee to the program. However, there are two types of employees: base employees and managers. There is a lot of common data collected for both employee types. However, base employees are paid an hourly rate and Managers are paid a yearly salary. The Manager class is a subclass(child) class of the Employee class. Both the Employee class and Manager class each have their own `monthlyPay()` and `toString()` methods. At the end of the program there are 2 polymorphic references. The first is: `monthly_salary=employee1.monthlyPay();` The program must determine at this point if the employee is a base employee or manager and then it must use the correct version of `monthlyPay()`. The next programming statement prints the object. The program must determine which of the `toString()` methods to use to print the correct output to the screen.

Example7_1.java

```
/*
 * Filename:Example7_1.java
 * Description: This driver file creates an Employee object and then the user decides
 * if the employee should stay a base employee or be assigned as a Manager.
 * Created by: Paul Murray
 * Date Created: 6/7/2025
 */
```

```
import java.util.Scanner;
import java.text.NumberFormat;

public class Example7_1 {
    public static void main(String[] args)
    {
        Scanner scan= new Scanner(System.in);
        BaseEmployee employee1; //create and object based on parent class
        String employee_name, employee_rank;
        int year_started;
        double monthly_salary,num_work_hours;
        NumberFormat fmt = NumberFormat.getCurrencyInstance();

        System.out.print("What is the employee's name? ");
        employee_name=scan.nextLine();

        System.out.print("What is the employee's start year? ");
```

```

year_started=scan.nextInt();
scan.nextLine(); //read the EOL character after the integer

System.out.print("How many hours did they work this month? ");
num_work_hours=scan.nextDouble();
scan.nextLine(); //read the EOL character after the double

System.out.print("Is this employee a base employee or manager? ");
employee_rank=scan.nextLine();

if(employee_rank.equalsIgnoreCase("Manager"))
{
    String dept_managed;
    double manager_salary;

    System.out.print("What department do they manage? ");
    dept_managed=scan.nextLine();

    System.out.print("What is their yearly salary? ");
    manager_salary=scan.nextDouble();

    //upcast employee1 as a Manager
    employee1= new Manager(employee_name, year_started, num_work_hours, dept_managed,
manager_salary);
}
else //anything other than manager leaves them as a "Base Employee"
{
    employee1= new BaseEmployee(employee_name,year_started,num_work_hours);
}

monthly_salary=employee1.monthlyPay(); //it must decide which of the 2 monthlyPay methods to
use
System.out.println(employee1+" and they are paid "+fmt.format(monthly_salary)+" this month"); //1
of 2 possible toString methods will be used
scan.close();

} //end main method
} //end Example7_1 class

```

BaseEmployee.java

```

/*
 *Filename:BaseEmployee.java
 *Description: This parent class defines common characteristics of all employees.
 *Created by: Paul Murray
 *Date Created: 6/7/2025
 */
public class BaseEmployee {

    protected String employee_name;
    protected int start_year;
    protected double hours_worked; //only used for base employee's pay
    final double BASE_HOURLY_PAY=19.50, OVERTIME_EXTRA=9.75;

    /*
     * Method Name: BaseEmployee

```

```

* Description: This is the constructor for assigning the common data for all employees
* Formal Parameters: _name(String)- The employee's name
*           _year_started(int)- The year the employee started working
*           _hours_worked(int)- The number of hours worked this month.
* Returns: nothing
*/
public BaseEmployee(String _name, int _year_started, double _hours_worked)
{
    employee_name=_name;
    start_year=_year_started;
    hours_worked=_hours_worked;

} //end BaseEmployee constructor method

/*
* Method Name: monthlyPay
* Description: This method determines how much this employee gets paid per month.
*           Base employees are hourly workers and get paid an hourly rate
* Returns: (double)- Monthly pay for this employee
*/
public double monthlyPay()
{
    if (hours_worked<=160)
        return BASE_HOURLY_PAY*hours_worked;
    else
    {
        double overtime_hours=hours_worked-160;
        return (BASE_HOURLY_PAY*hours_worked)+(overtime_hours*OVERTIME_EXTRA);
    }

} //end monthlyPay

/*
* Method Name: toString
* Description: This toString method defines what will get printed from a Employee object.
*           Note: Child classes can invoke this method through a super.toString() call.
* Formal Parameters: none
* Returns: (String)- the text to be printed from an object of this class
*/
public String toString()
{
    return employee_name+" started working for the company in "+start_year;
} //end toString method

} //end class BaseEmployee

```

Manager.java

```

/*
*Filename:Manager.java
*Description: This child class of the BaseEmployee class defines characteristics specific
* to a Manager.
*Created by: Paul Murray
*Date Created: 6/7/2025
*/
public class Manager extends BaseEmployee{

```

```

private String department_name;
private double salary;

/*
 * Method Name: Manager
 * Description: This is the constructor for assigning the data needed for a Manager
 * Formal Parameters: _name(String)- The employee's name
 *                   _year_started(int)- The year the employee started working
 *                   _num_work_hours(double)- The number of hours worked this month
 *                   _dept_name(String)- The department they are a manager of
 *                   salary(double)- The yearly salary for the manager
 * Returns: nothing
 */
public Manager(String _name, int _year_started, double _num_work_hours, String _dept_name, double
_salary)
{
    super(_name, _year_started, _num_work_hours);
    department_name=_dept_name;
    salary=_salary;
} //end Manager constructor method

/*
 * Method Name: monthlyPay
 * Description: This method determines how much this employee gets paid per month
 *             Managers are salaried workers and get paid 1/12 of salary every month
 *             plus a possible bonus
 * Returns: (double)- Monthly pay for this employee
 */
public double monthlyPay()
{
    if (hours_worked>=170)
        return (salary/12.0)+150; //manager's working over 170 hours a month will get a $150 bonus
    else
        return (salary/12.0);
} //end monthlyPay

/*
 * Method Name: toString
 * Description: This toString method defines what will get printed from a Manager object.
 *             Note: This method does not use the toString() method of the BaseEmployee parent
 *             class, but does refer to the employee_name instance variable
 *             assigned to the Employee object.
 * Formal Parameters: none
 * Returns: (String)- the text to be printed from an object of this class
 */
public String toString()
{
    return employee_name+" is a manager of the "+department_name+" team";
} //end toString method
} //end Manager class

```

Example 7.1 sample input/output:

Attempt #1- User picks “base employee”. This means Marcie is an hourly base employee. The BaseEmployee monthlyPayment() is called to calculate her pay and the correct toString is used to output the results.

Input

What is the employee's name? Marcie Jo Potter

What is the employee's start year? 2023

How many hours did they work this month? 165

Is this employee a base employee or manager? employee

Output

Marcie Jo Potter started working for the company in 2023, and they are paid \$3,266.25 this month

Attempt #2- User picks “manager”. This means Richard is a salaried manager. The Manager monthlyPayment() is called to calculate his pay and the correct toString is used to output the results.

Input:

What is the employee's name? Richard David Partner

What is the employee's start year? 2019

How many hours did they work this month? 179

Is this employee a base employee or manager? manager

What department do they manage? Sales

What is their yearly salary? 97900

Output:

Richard David Partner is a manager of the Sales team, and they are paid \$8,308.33 this month

In this example, you might question what the point of creating employee1 as a BaseEmployee at the top of the program and then later on using the constructor to instantiate as a Manager or BaseEmployee. Why not just create a Manager object or BaseEmployee object once the program knows what it needs to create? Well, first it is okay to upcast a BaseEmployee to a Manager as Manager is a child of the BaseEmployee class. But is there an advantage to doing this? The advantage of doing it this way is demonstrated in the last three lines of the *Example7_1.java* file where the common methods shared in both classes are used(monthlyPay() and toString()). Rather than complicating the code with code specific to one particular class, the program can simply refer to a method and thus allow the program to pick the correct method.

7.3 Casting- Upcasting and Downcasting

Upcasting occurs when an object of a child class(subclass) is treated as if it was an object of its parent class(superclass). Like a standard widening conversion, this conversion happens automatically.

In the previous example there was the following code:

```
BaseEmployee employee1;
...
employee1= new Manager(employee_name, year_started, num_work_hours, dept_managed,
manager_salary);
```

This is an example of upcasting where employee1 was converted to a Manager when it was instantiated.

Downcasting occurs when an object is converted from a superclass to a subclass. This conversion does not happen automatically and requires an explicit casting. Like narrowing conversions, this is a technique that you need to be careful with as going from a generic type to a specific type can cause errors with existing code.

Using the same example again, consider the following code modifications:

```
BaseEmployee employee1;
employee1= new BaseEmployee(employee_name,year_started,num_work_hours);
Manager new_employee=(Manager)employee1; //this is the attempted cast
```

The final line of code will throw an exception. This happens because the program is trying to cast a child class of BaseEmployee to the parent class of Manager. The difference is that there are things about the Manager that are not supplied in the BaseEmployee class like what department they manage and what their salary is, and therefore a simple conversion is not possible.

Example 7.2

As stated before, the most common technique for creating a polymorphic object is utilizing inheritance. Consider a program that rents boats to customers. The marina knows what kind of watercraft they have for rent, but they may not know in advance what boats will be rented on any given day.

In this example, general watercraft will always have a manufacturer, the length of the boat, and the rental price. These values can be assigned to a parent class called Watercraft. Three child classes of the Watercraft class called RowBoat, MotorBoat and JetSki further refine specifics to each one of these watercraft. For example, a rowboat will have no engine and oars, while the other two watercraft will have an engine and no oars.

Note: This example uses a dialog box from the Java swing package which includes radio buttons and a command button.

Example7 2.java

```
/*
 *Filename:Example7_2.java
 *Description: This driver file creates a reservation system for multiple types of watercraft.
 *Created by: Paul Murray
 *Date Created: 6/7/2025
 */

import javax.swing.*;
import java.awt.*;
import java.util.Scanner;

public class Example7_2 extends JFrame{

    String selected_button="unselected";

    /*
     * Method Name: main
     * Description: This is the driver method of the program. This launches the program.
     * Formal Parameters: args(String[])- An array that allows command line arguments to be
     *                    passed to the program.
     * Returns: void
     */
}
```

```

public static void main(String[] args)
{
    new Example7_2(); //call the embedded constructor method
} //end main method

/*
 * Method Name: Example7_2
 * Description: This is the constructor for setting up the default start of the program including
 * the dialogbox for a boat rental
 * Formal Parameters: none
 * Returns: nothing
 */
public Example7_2() {

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JDialog dialogbox = new JDialog(this, "Murray's Boat Rental- Pick your watercraft");
    dialogbox.setLayout(new FlowLayout()); //setup the layout of the dialog box inside the JFrame

    //Create 3 radio buttons
    JRadioButton option1 = new JRadioButton("Rowboat");
    JRadioButton option2 = new JRadioButton("Jet Ski");
    JRadioButton option3 = new JRadioButton("Motorboat");

    //Add the buttons to a group so that only one can be picked from the group
    ButtonGroup group = new ButtonGroup();
    group.add(option1);
    group.add(option2);
    group.add(option3);

    dialogbox.add(option1); //add all the buttons to the dialog box
    dialogbox.add(option2);
    dialogbox.add(option3);

    JButton submit_button = new JButton("Submit Choice"); //a form submit button

    dialogbox.add(submit_button);
    dialogbox.pack();
    dialogbox.setLocationRelativeTo(this); //move the dialogbox out of the upper left corner
    if(selected_button.equals("unselected")) //dialog box appears until submit button is pressed
        dialogbox.setVisible(true);

    submit_button.addActionListener(buttonlistener -> { //a listener that notes which radio button
// has been selected when the submit button is pressed
        if (option1.isSelected()) {
            selected_button = "Rowboat";
        } else if (option2.isSelected()) {
            selected_button = "Jet Ski";
        } else if (option3.isSelected()) {
            selected_button = "Motorboat";
        } else {
            selected_button = "unselected"; //user clicks "submit choice" without picking a radio button
        }

        dialogbox.setVisible(false); //once selection is made- hide the dialog box
        completeRental(selected_button); //finish the rental process
    }); //end listener

```

```

} //end constructor method

/*
 * Method Name: completeRental
 * Description: This method completes the rental process after the user has already selected their
 * watercraft of choice in another method
 * Formal Parameters: the _watercraft(String)- The value selected as their watercraft rental choice
 * Returns: void
 */
public void completeRental(String the _watercraft)
{
    Watercraft boat_choice; //parent class
    Scanner scan= new Scanner(System.in);
    String craft_manufacturer;
    int craft_length;
    double the_rental_price=0;

    System.out.print("What is the watercraft's manufacturer? ");
    craft_manufacturer=scan.nextLine();
    System.out.print("What is the watercraft's length? ");
    craft_length=scan.nextInt();
    System.out.print("What is the watercraft's rental price? ");
    the_rental_price=scan.nextDouble();
    scan.nextLine(); //eliminates the end of line (EOL) marker left on the input line

    if (the _watercraft.equals("Rowboat"))
    {
        int num_seats;
        System.out.print("How many seats does the rowboat have? ");
        num_seats=scan.nextInt();
        boat_choice= new RowBoat(craft_manufacturer,craft_length,the_rental_price,num_seats);
    }
    else if (the _watercraft.equals("Jet Ski"))
    {
        int hp;
        System.out.print("How much horsepower does the jet ski have? ");
        hp=scan.nextInt();
        boat_choice= new JetSki(craft_manufacturer,craft_length,the_rental_price,hp);
    }
    else if (the _watercraft.equals("Motorboat"))
    {
        boolean inboard=false;
        String inboard_reply;
        int num_hulls;

        System.out.print("Does the motorboat use an inboard engine? (Y/N) ");
        inboard_reply=scan.nextLine().toUpperCase();
        if (inboard_reply.equals("YES")||inboard_reply.equals("Y") )
            inboard=true;
        System.out.print("How many hulls does the boat have? ");
        num_hulls=scan.nextInt();
        boat_choice= new MotorBoat(craft_manufacturer,craft_length,the_rental_price,inboard,num_hulls);
    }
    else

```

```

        boat_choice= new Watercraft(craft_manufacturer, craft_length,the_rental_price);

        System.out.println(boat_choice); //output the data from the appropriate toString method.
        System.exit(0); //close the program when finished
    } //end method completeRental
} //end Example7_2 class

```

Watercraft.java

```

/*
 *Filename:Watercraft.java
 *Description: This parent class defines common characteristics of all watercraft rentals.
 *Created by: Paul Murray
 *Date Created: 3/17/2025
 */

import java.text.NumberFormat;

public class Watercraft {

    protected String manufacturer;
    protected int length;
    protected double rental_price;

    /*
     * Method Name: Watercraft
     * Description: This is the constructor for assigning the common data for all boat rentals
     * Formal Parameters: _manufacturer(String)- The boat's manufacturer
     *                     _length(int)- The length of the boat (from bow to stern)
     *                     _rental_price(double)- How much the boat rents for
     * Returns: nothing
     */
    public Watercraft(String _manufacturer, int _length, double _rental_price)
    {
        manufacturer=_manufacturer;
        length=_length;
        rental_price=_rental_price;
    } //end Watercraft constructor method

    /*
     * Method Name: toString
     * Description: This toString method defines what will get printed from a Watercraft object.
     *             Note: Child classes can invoke this method through a super.toString() call.
     * Formal Parameters: none
     * Returns: (String)- the text to be printed from an object of this class
     */
    public String toString()
    {
        NumberFormat theformat=NumberFormat.getCurrencyInstance();
        return "This "+manufacturer+" watercraft rents for "+theformat.format(rental_price)+" and is "+length+"
feet long";
    } //end toString method
} //end class Watercraft

```

MotorBoat.java

```

/*
 *Filename:MotorBoat.java
 *Description: This child class defines specific characteristics MotorBoats.
 *Created by: Paul Murray
 *Date Created: 3/17/2025
 */
public class MotorBoat extends Watercraft{

    private boolean inboard;
    private int hulls;

    /*
     * Method Name: MotorBoat
     * Description: This is the constructor for assigning the common data for all boat rentals (through a
     *               super call to the parent's constructor) and the specific data for a motorboat
     * Formal Parameters: _manufacturer(String)- The boat's manufacturer
     *               _length(int)- The length of the boat (from bow to stern)
     *               _rental_price(double)- How much the boat rents for
     *               _inboard (boolean)- Is true if the boat uses an inboard engine OR is false if
     *                               the boat uses an outboard engine
     *               _hulls(int)- The number of hulls the boat has. The most common are 1 or 2, though there
     *                               are manufacturers that use 3. There are certain occasional instances greater
     *                               than 3.
     * Returns: nothing
     */
    public MotorBoat(String _manufacturer, int _length, double _rentalprice, boolean _inboard, int _hulls)
    {
        super(_manufacturer, _length, _rentalprice);
        inboard=_inboard;
        hulls=_hulls;
    } //end MotorBoat constructor method

    /*
     * Method Name: determineHullType
     * Description: Method simply exists to clean up the code in toString. The purpose is to determine
     *               what string of text to print for the number of hulls the motorboat has.
     *               Note: Unlike many methods being public, this method can remain private as it is not meant
     *               to be accessed from outside this class.
     * Formal Parameters: none
     * Returns: (String)- the text to be sent to the toString method for the "hull type"
     */
    private String determineHullType()
    {
        String hullstring;
        if (hulls==1)
            hullstring="mono";
        else if (hulls==2)
            hullstring="double";
        else if (hulls==3)
            hullstring="tri";
        else
            hullstring="unknown";

        return hullstring;
    }
}

```

```

    } //end determineHullType method

/*
 * Method Name: toString
 * Description: This toString method defines what will get printed from a MotorBoat object
 *             Note: This class invokes toString from the parent class for part of the output.
 *                   through a super.toString() call.
 *             Note: Calls determineHullType() to determine what type of hull the motorboat has
 * Formal Parameters: none
 * Returns: (String)- the text to be printed from an object of this class
 */
    public String toString()
    {
        if (inboard)
            return super.toString()+" and has a "+determineHullType()+"hull and uses an inboard motor";
        else
            return super.toString()+" and has a "+determineHullType()+"hull and uses an outboard motor";
    } //end toString method
} //end MotorBoat class

```

JetSki.java

```

/*
 * Filename:JetSki.java
 * Description: This child class defines specific characteristics Jetskis.
 * Created by: Paul Murray
 * Date Created: 3/17/2025
 */
public class JetSki extends Watercraft{

    private int hp;

/*
 * Method Name: JetSki
 * Description: This is the constructor for assigning the common data for all boat rentals (through a
 *             super call to the parent's constructor) and the specific data for a jetski
 * Formal Parameters: _manufacturer(String)- The boat's manufacturer
 *                   _length(int)- The length of the boat (from bow to stern)
 *                   _rental_price(double)- How much the boat rents for
 *                   hp(int)- The hp of the jetski engine
 * Returns: nothing
 */
    public JetSki(String _manufacturer, int _length, double _rentalprice, int _hp)
    {
        super(_manufacturer, _length, _rentalprice);
        hp=_hp;
    } //end JetSki constructor method

/*
 * Method Name: toString
 * Description: This toString method defines what will get printed from a JetSki object
 *             Note: UNLIKE the MotorBoat and RowBoat classes, this method DOES NOT invoke
 *                   the toString from the parent class.
 * Formal Parameters: none
 * Returns: (String)- the text to be printed from an object of this class
 */
    public String toString()

```

```

    {
        return "The Jet Ski is "+length+" ft long and has a "+hp+" horsepower engine";
    } //end toString method
} //end class JetSki

```

RowBoat.java

```

/*
 *Filename:RowBoat.java
 *Description: This child class defines specific characteristics RowBoats.
 *Created by: Paul Murray
 *Date Created: 3/17/2025
 */
public class RowBoat extends Watercraft{

    private int seats, oars;

    /*
    * Method Name: RowBoat
    * Description: This is the constructor for assigning the common data for all boat rentals (through a
    *               super call to the parent's constructor) and the specific data for a rowboat
    * Formal Parameters: _manufacturer(String)- The boat's manufacturer
    *               _length(int)- The length of the boat (from bow to stern)
    *               _rental_price(double)- How much the boat rents for
    *               _seats(int)- The number of seats in the rowboat
    * Returns: nothing
    */
    public RowBoat(String _manufacturer, int _length, double _rentalprice, int _seats)
    {
        super(_manufacturer, _length, _rentalprice);
        seats=_seats;
        calculateOars(); //call a method to calculate the number of oars
    } //end constructor method

    /*
    * Method Name: calculateOars
    * Description: This method simply exists to calculate the number of oars for a rowboat. It simply takes
    *               the provided number of seats and determines there need to be 2 oars per seat. No, this
    *               method doesn't need to really exist as the calculation could have been done in the
    *               constructor method.
    * Formal Parameters: none
    * Returns: void
    */
    private void calculateOars()
    {
        oars=seats*2;
    } //end calculateOars method

    /*
    * Method Name: toString
    * Description: This toString method defines what will get printed from a RowBoat object
    *               Note: This class invokes toString from the parent class for part of the output.
    *               through a super.toString() call.
    * Formal Parameters: none
    */

```

```

    * Returns: (String)- the text to be printed from an object of this class
    */
    public String toString()
    {
        return super.toString()+" and has room for "+seats+" rowers and it has "+oars+" oars";
    } //end toString method
} //end class RowBoat

```

Input: Attempt #1- User picked Motorboat from dialog box

What is the watercraft's manufacturer? BayLiner

What is the watercraft's length? 21

What is the watercraft's rental price? 487.25

Does the motorboat use an inboard engine? (Y/N) Y

How many hulls does the boat have? 1

Output:

This BayLiner watercraft rents for \$487.25 and is 21 feet long and has a monohull and uses an inboard motor

Input: Attempt #2- User picked Rowboat from dialog box

What is the watercraft's manufacturer? Empacher

What is the watercraft's length? 55

What is the watercraft's rental price? 572.83

How many seats does the rowboat have? 8

Output:

This Empacher watercraft rents for \$572.83 and is 55 feet long and has room for 8 rowers and it has 16 oars

7.4 Avoid hiding variables

A common mistake when using inheritance is **hiding variables**. This happens when a programmer creates the same named field in both a superclass (parent) and subclass (child). When you create this duplicate in the child class it hides the parent variable inside the child's scope. This is something you usually want to avoid doing as it can lead to confusion especially when debugging the program. It should be noted that both variables exist and both values can be accessed. You will need to use the super keyword to reference the parent variable's value. This is not exclusive to polymorphic inheritance as this can be done with any inheritance scenarios.

Example 7.3

Consider the following example that defines a value for a *teacher_name* in both classes.

Example7 3.java

```

/*
 * Filename:Example7_3.java
 * Description: This file demonstrates how the same variable can be assigned in both the parent and
 * child class. Though it is not technically wrong to do so, you have one employee with 2 different
 * names. This can lead to potential mistakes or logical errors down the line.
 * Created by: Paul Murray
 * Date Created: 6/21/2025
 */

```

```

class Personnel
{
    public String teacher_name="Bill Jenkins";
    public int age=34;

    public void printName()
    {
        //just an empty stub method
    } //end printName method
} //end Personnel class

class Teacher extends Personnel
{
    public String teacher_name="April Briggerton";

    public void printName()
    {
        System.out.println("Teacher: "+teacher_name+" - "+age+" years old"); //print the local teacher_name
        System.out.println("Teacher: "+super.teacher_name+" - "+age+" years old"); //print the parent
        teacher_name
    } //end printName method
} //end Teacher class

public class Example7_3
{
    public static void main(String[] args)
    {
        Personnel our_teacher= new Teacher(); //upcast
        our_teacher.printName();
    } //end main method
} //end Example7_3 class

```

Output:

Teacher: April Briggerton - 34 years old

Teacher: Bill Jenkins - 34 years old

As you can see from the output, even though there is only one object, there are 2 *teacher_name* values and 1 *age* value stored in that object. So, is the name of the teacher April or Bill? Well, it is both at the same time.

Because of the upcast casting of the *our_teacher* object, the *printName()* method of *Teacher* class will be used instead of the empty stub method found in the *Personnel* class.

Chapter 8: Exception Handling and Working with Files

Scenario: You've developed this wonderfully robust program for a client that collects and processes data from user input. After fully testing the program, the software is released. Two hours after going live, you get a phone call. The program has crashed. After talking with the client, you find out that when the user was prompted for the number of children they had they typed in the word *none*. Since your program used the `nextInt()` method of the `Scanner` class, it expected an integer and received a `String` instead. The `nextInt()` method failed and threw an exception message and the program closed prematurely.

Whose fault is this? The programmer or the user? Without a doubt, this is the fault of the programmer. A simple rule for programmers to follow is to assume users will not read instructions and also that the program should be ready to handle any input it receives. As a programmer, you must develop your code to handle just about any unexpected input from a user or unexpected data coming from any other input source.

Why does this happen? What is an exception? Does an exception always crash a program? Basically, a **catchable exception** is generated during runtime when an unexpected condition happens that disrupts the normal flow of the program. An exception is thrown from a piece of a programming code that can't handle the unexpected condition. In this example, the `nextInt()` method cannot handle the unexpected `String` input. The `nextInt()` method has no exception handling, so the exception is propagated (passed) to the calling program. If the catchable exception is not handled in your program, the program will crash when it receives invalid data. The good news is that there is a way to handle exceptions. By handling an exception, you can prevent a program from crashing. Most programming languages have some method of exception handling. In Java this is performed with a `try..catch` statement.

Do you have to handle an exception? What are the options?

1. You can decide the program is not going to handle the exception. This will usually cause the program to terminate prematurely. This is also known as having a program crash.
2. You can handle the exception where it occurs.
3. You can decide to handle the exception at a future time in the program through exception propagation.

If you decide to handle the exception, how do you handle it? As a programmer you have a few options:

1. Simply report the exception to the user and allow the program to terminate.
2. Correct the problem so that the program can continue to operate without notifying the user. For example, if a program couldn't open a file to write data, the program could simply not output the data. This technique is problematic as it can be deceiving to the user of the program. It is often recommended to let the user know if something doesn't work as designed.
3. Correct the problem so the program does not prematurely terminate and notify the user that something went wrong. For example, if a divide-by-zero exception happened, you could catch the exception and assign a default result value and notify the user that the calculation couldn't be properly ascertained.

4. Catch the exception and notify the user and give them the ability to fix the problem. This can often be handled with a loop that will continue until an exception does not occur. For example, what if the program prompted the user for an integer and the user provided a String value? This will cause an exception if the program is using the `nextInt()` method of the `Scanner` class. The program can catch the exception, notify the user of the issue and re-prompt them for the data again. The program could continue catching the exception until the user provided a valid integer.

8.1 Errors vs Exceptions

Many programmers assume that errors and exceptions are the same thing. In fact, it is common for experienced programmers to incorrectly refer to exceptions as errors. There are a lot of similarities between the two. However, there is one significant difference between errors and exceptions. Errors are severe issues and a program often can't recover from an error. Exceptions can usually be caught and dealt with without crashing the program.

Examples of errors include `OutOfMemoryError` and `StackOverflowError`. Unlike exceptions, it is not advisable to try catching and handling an error. These are usually fatal issues for a program to experience and will cause the program to crash without a chance for recovery. Exceptions on the other hand are usually not as severe and can be recovered from. It is advisable to try to catch any exceptions and handle them so that the program does not prematurely terminate.

There are two types of exceptions, catchable and non-catchable.

- **Non-catchable exceptions** happen at *compile time* and must be fixed before the program will compile successfully.
- **Catchable exceptions** happen at *run-time* and can be caught and handled.

8.2 try..catch statements

The `try..catch` statement allows a program to catch exceptions and handle the exception without having the exception terminate a program and leaving a cryptic exception message.

The code that could possibly generate an exception must go inside the body of the `try` block. If the code executes correctly, the code in the `catch` statement is bypassed. However, if an exception is generated, the code in the `catch` statement block is used as long as the exception is listed in the `catch` statement header.

```
try
{
    //code that could generate an exception
}
catch (exception to be caught )
{
    //code to handle the exception and a possible message to the user of the program
}
```

Unlike other programming languages, in Java the header of the `catch` statement cannot be left blank. Each `catch` block must explicitly define the exception or exceptions that it is trying to

catch. Thus, the programmer must be aware of what type of exceptions that may be generated from a piece of code.

8.3 Common exceptions

Here is a list of common exceptions:

- **InputMismatchException**- This occurs when input does not match what is expected. For example, if a program uses a `nextInt()` method from the `Scanner` class but it receives a floating point number or a `String` from user input it will generate this exception.
- **NullPointerException**- This occurs when referencing a null object as if it was a valid object reference.
- **ArithmeticException**- This occurs when performing an illegal math statement such as dividing by zero or integer overflow/underflow.
- **NumberFormatException**- This occurs when trying to convert a string value into a numeric equivalent that cannot be accomplished. For instance, you cannot convert the string "123xyz" to an integer.
- **ArrayIndexOutOfBoundsException**- This occurs when a program tries to reference an index that is out of the bounds of the array. For example, if the program tries to reference index location 7 in an array of size 7 will generate an exception as the valid index ranges from 0-6.
- **IllegalArgumentException**- This occurs when a method is provided an inappropriate argument for a parameter. For example the `charAt()` method of the `String` class requires a single `int` value as a parameter. If a program called the method and passed it a `String` value, an exception would be generated.
- **IOException**- This occurs when something prevents input or output to be properly be read from or written to a file.
- **FileNotFoundException**- This occurs when the program tries to access an external file and cannot find it. This sometimes occurs when the program simply cannot find a file by that name, or the file is in a path location that the program has not been given. For some IDEs this requires that an external file be included in a project or program/project folder.

A full list of exceptions can be found in the online JavaDocs document under the `java.lang.Exceptions` class. Note: Not all exceptions are direct exceptions from the `Exceptions` class. For instance, the `ArrayIndexOutOfBoundsException` class is a child of the `IndexOutOfBoundsException` class, which is a child of the `java.lang.RuntimeException` class, which is a child of the `java.lang.Exception` class.

It should be noted that most exceptions are part of the `java.lang` package and are therefore automatically included in any program. However, there are some exceptions that are defined in other locations that will need to be imported. For example, `java.io.IOException`, `java.sql.SQLException` are located outside of the `java.lang` package.

8.4 What code goes inside a try block

A programmer should be careful how much code is placed inside a try block. It should be understood that once an exception is thrown the program immediately jumps to the catch block. Any remaining code in the try block after the exception was thrown will be skipped over. Avoid putting a lot of code inside of a try block to avoid program flow issues. It is quite common for programmers to place only one or two programming statements inside of a try block.

What kind of programming statements can generate exceptions? The most common exceptions come from input streams. Whether this comes from user input or file input, the program cannot anticipate potential unexpected data coming from input. For example, what if your program prompts the user for their age and you decide to read that value in as an integer using the `nextInt()` method of the `Scanner` class. If the user types in a value such as 20 would be perfectly fine and the program would continue just fine. However, if the user types twenty, `nextInt()` will generate an `InputMismatchException` exception. Therefore you would want the `nextInt()` programming statement inside the try statement, but you may also want to include the user prompt inside the try statement so if you use a loop to repeat the code, both the prompt and scan is included in the attempt.

Should you use a loop around a try..catch statement? In some situations, a program may want to continue attempting a set of programming statements until an exception is no longer generated. For example, if a program is attempting to read in a numeric value from the user, it may continue to try to until the user types in a numeric value. However, there are scenarios where this is not always recommended. For example, if a program attempts to open a file and it can't find it, the program will generate an exception. If this code is placed in a loop, it is unlikely that the program is going to find the file at any point. This type of exception would generate an infinite loop as the exception would continue to happen.

8.5 Checked or unchecked exceptions

Java separates exceptions into two categories: checked and unchecked.

Checked exceptions are exceptions that the compiler forces the program to handle. These are considered compile-time exceptions as the compiler will not let you continue without handling the exceptions. Handling the exception can be accomplished with either a **try..catch** statement for that exception or utilizing a **throws** clause on the method that is invoking the problematic programming statement. Examples of checked exceptions include `FileNotFoundException` and `ClassNotFoundException`.

Unchecked exceptions are run-time exceptions that can be handled or not handled. If the program does not handle an exception, it will prematurely terminate the program. Using the **throws** clause on an unchecked exception is optional as it is thrown automatically for run-time exceptions.

It should also be noted that there is a *throws* keyword and *throw* keyword. The *throw* keyword will be introduced later in the section of creating your own exception.

8.6 Exception propagation

An alternative to handling an exception is to use **exception propagation**. Exception propagation occurs when a method does not handle an exception and passes it back to the calling program. It is optional to use the keyword **throws** in the header of a method allowing the exception to

passed back to the method that called it. The exception should be handled in the calling program.

Example 8.1

Consider the following example that attempts to add students to a course at a university.

Example8_1.java

```

/*
 *Filename:Example8_1.java
 *Description: This example file demonstrates exception propogation. The exception
 *             is generated in the course() method and is not handled in that
 *             method. The exception is then passed to the campus() method.
 *             The campus method doesn't handle the exception either and passes
 *             the exception back to the main method. The exception was handled
 *             here. If it hadn't been handled, the exception would have
 *             prematurely terminated the program.
 *Created by: Paul Murray
 *Date Created: 7/8/2025
 */
public class Example8_1
{
    /*
     * Method Name: main
     * Description: This is the driver method of the program. This launches the program.
     * Formal Parameters: args(String[])- An array that allows command line arguments to be passed to the
     program.
     * Returns: void
     */
    public static void main(String[] args)
    {
        System.out.println("Welcome to the Registration app..");
        try
        {
            campus();
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Course registration failed. Please contact IT at (717)-555-1212");
        }

        System.out.println("..Registration app closing");
    } //end main method

    /*
     * Method Name: campus
     * Description: This method produces output particular to the college campus.
     *             Note: if an exception is generated in the call to the course()
     *             method, the last println() statement won't be reached.
     *             Note: This method uses a throws statement in the header. This
     *             basically is a way of not dealing with an exception and
     *             passing it back to the program that called this method.
     * Formal Parameters: none
     * Returns: void
     */
    public static void campus() throws ArrayIndexOutOfBoundsException
    {
        System.out.println("Southern Polytechnic University student registration..");
    }
}

```

```

        course(); // Calls the 2nd method
        System.out.println("..Finished registering students"); //might not be reached
    } //end campus method

/*
 * Method Name: course
 * Description: This method produces output particular to the college course.
 *             It attempts to assign student's to a course array.
 *             Note: if an exception is generated if too many students are
 *                   added to the array, the last println() statement won't
 *                   be reached.
 *             Note: This method uses a throws statement in the header. This
 *                   basically is a way of not dealing with an exception and
 *                   passing it back to the method that called this method.
 * Formal Parameters: none
 * Returns: void
 */
public static void course() throws ArrayIndexOutOfBoundsException
{
    String[] students= new String[2];

    System.out.println("Adding students to course..");

    students[0]= "Tom";
    students[1]= "Riya";
    //Comment out the following line of code if you don't want an exception happening
    students[2]= "Akiko"; //will cause exception as index 2 doesn't exist in the array

    System.out.println("..Finished adding students to course"); //might not be reached
} //end course method
} //end Example8_1 class

```

Output if exception left in the code:

```

Welcome to the Registration app..
Southern Polytechnic University student registration..
Adding students to course..
Course registration failed. Please contact IT at (717)-555-1212
..Registration app closing

```

The exception happens in the course() method and immediately leaves the method, this not printing the “..Finished adding student to course” output and control goes to the campus() method that called this. Since there is no exception handling in the campus() method either, it’s final output statement is missed as control then passes immediately to the main() method that called the campus() method. Since the try..catch statement in the main() method correctly catches an ArrayIndexOutOfBoundsException, the exception is handled and an appropriate message is printed to the user.

Output if the students[2]= "Akiko"; exception was removed from program

```

Welcome to the Registration app..
Southern Polytechnic University student registration..
Adding students to course..
..Finished adding students to course
..Finished registering students
..Registration app closing

```

8.7 Using multiple exceptions in a single catch

Though not necessarily advised, a single catch statement can handle more than one catch to handle multiple exceptions. This allows the programmer to handle multiple types of exceptions in a more generic way. For example, let's assume that we are collecting a number from the user and immediately using it in a math statement. There is a possibility that two types of exceptions could happen here. We may have the user type in a non-integer value or type in an integer that could cause a division by zero in the math statement. The program could catch either exception and simply report back to the user that the value isn't valid. However, this generic message doesn't really clue the user into why their value wasn't valid.

8.8 Using multiple catches

An alternative to using multiple exceptions in a single catch is to have multiple catches for each try block. This allows a program to handle each unique exception separately. Using the example from the previous section, the program could still have a single try block, but use 2 catches instead. The first catch could handle non-integer values, and the second catch could handle division by zero.

Example 8.2

Consider the following example that uses two catches with a single try block. This example catches 1 of 2 exceptions and will report the type of exception that was caught. The program will then continue to prompt the user of the program for a valid value until the user submits a non-zero integer value.

Example8_2.java

```
import java.util.Scanner;
import java.util.InputMismatchException;

/*
 *Filename:Example8_2.java
 *Description: This example file demonstrates a single try block that is connected to 2
 *             catch blocks. One catch block ensures the value submitted to the
 *             Scanner method is a valid integer. The other catch block ensures
 *             that the integer value doesn't cause a divide by zero condition. The
 *             program will continue to loop until the user types in a valid, non-zero
 *             integer value.
 *Created by: Paul Murray
 *Date Created: 7/9/2025
 */
public class Example8_2
{
    public static void main(String[] args)
    {
        Scanner scan= new Scanner(System.in);
        boolean no_exception;
        int todays_income=56013;
        double average_per_order=0;
        do
        {
            no_exception=true; //reset to valid every time through loop
            try
            {
```

```

        System.out.print("How many orders have you processed today? ");
        average_per_order=todays_income/scan.nextInt(); //2 exceptions can happen in single statement
    }
    catch (InputMismatchException e)
    {
        no_exception=false; //force to loop again
        System.out.println("\t**Invalid integer value**");
        System.out.println("\tThe value you supply needs to be a numeric integer");
        System.out.println("\tPlease try again\n");
        scan.nextLine(); //read the invalid token to avoid infinite loop
    }
    catch (ArithmeticException e)
    {
        no_exception=false; //force to loop again
        System.out.println("\n\t**Divide by zero**");
        System.out.println("\tThe value you typed needs to be a non-zero integer value");
        System.out.println("\tPlease try again\n");
    }
}

} while (!no_exception);

scan.close(); //close Scanner object

System.out.println("\n\nThe average $ for each order was: $"+(int)average_per_order);

} //end main method
} //end Example8_2 class

```

Input/Output if the user typed in a non-numeric value:

```

How many orders have you processed today? three
**Invalid integer value**
The value you supply needs to be a numeric integer
Please try again

```

How many orders have you processed today? *(program is looping back to the prompt)*

Input/Output if the user typed in a zero:

```

How many orders have you processed today? 0

**Divide by zero**
The value you typed needs to be a non-zero integer value
Please try again

```

How many orders have you processed today? *(program is looping back to the prompt)*

Input/Output if the user typed in a valid integer:

```

How many orders have you processed today? 13

The average $ for each order was: $4308

```

8.9 The finally block and try-with-resources

A single **finally** block can be added to every try..catch statement.

What goes inside the body of the finally block is guaranteed to be performed in these scenarios:

- whether an exception occurs and is caught
- whether an exception occurs but is not caught
- whether there is no exception.

Usually, the finally statement is used for general memory management housekeeping such as closing file streams or database connections to avoid memory leaks.

```
try
{
    (FileReader input_file = new FileReader("ProgramData.txt"))
}
catch(IOException e)
{
    System.out.println("The input file could not be found");
}
finally
{
    input_file.close();
}
```

With Java 7, a **try-with-resources** feature was added to the try statement where a resource can be defined if needed. If an exception occurs, the resource is auto-closed without the need for a finally block.

```
try (FileReader input_file = new FileReader("ProgramData.txt"))
{ //body of the try that uses the file opened in the header
}
catch(IOException e)
{
    System.out.println("The input file could not be found");
}
```

Note that the FileReader object is declared in the parentheses of the try statement. Therefore, it is considered a resource and is managed automatically. Since FileReader implements the AutoCloseable interface, when the try block finishes, the close() method is automatically called.

8.10 Creating your own exception

A programmer is also allowed to create their own exception by extending either the Exception class or the RuntimeException class. It is important that an exception has a constructor method to handle what happens when an exception is generated. Once your own exception has been created, it is an exception that can be thrown using the keyword **throw**. It is up to your discretion if you want to handle the exception the program just threw with a try..catch statement.

Example 8.3

Consider the following example that creates an user created InvalidGPA exception when a GPA does not fall in the range between 0 and 4. The program then catches and handles the exception.

Example8 3.java

```
import java.util.Scanner;
/*
 *Filename:Example8_3.java
 *Description: This example file demonstrates how to create your own custom exception.
               Though not required, the program both throws the exception and catches
               and handles it.
 *Created by: Paul Murray
 *Date Created: 7/10/2025
 */
```

```

public class Example8_3
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        double current_gpa;

        try
        {
            System.out.print("Please input your current GPA ");
            current_gpa=scan.nextDouble();
            if (current_gpa>4.0 || current_gpa<0.0)
                throw new InvalidGPA("The GPA must be between 0.0 and 4.0 "); //generate custom exception
            else
                System.out.println("Thank you! Your GPA has been collected.");
        } //end try block
        catch(InvalidGPA e)
        {
            System.out.print(e.getMessage());
        } //end catch block
    } //end main method
} //end class CollectGPA

/* InvalidGPA is a custom exception class based on the RuntimeException class*/
class InvalidGPA extends RuntimeException
{
    /*
    * Method Name: InvalidGPA
    * Description: This is the constructor that simply takes the exception message passed to
    *               it and sends it to the constructor method of the RuntimeException class.
    *               This message will then be added to the thrown exception. If not handled,
    *               this exception will cause the program to prematurely terminate.
    * Formal Parameters: exception_message(String)- The message that will be added to the
    *                   thrown exception
    * Returns: nothing
    */
    public InvalidGPA(String exception_message)
    {
        super(exception_message); //Uses standard exception message format defined in
        RuntimeException
    } //end constructor method
} //end class Example8_3

```

In this example when the GPA was outside of the valid range, the keyword *throw* was used to create a *new* exception using the InvalidGPA class. In this case, the InvalidGPA class was defined inside the same file, but it could easily be placed in a separate file so that the InvalidGPA class could be a shared exception with other programs.

It should also be noted that the CollectGPA class both threw and handled the exception. It is always an option to throw an exception and not handle it.

You may wonder what the purpose of defining your own exception might be. In the example above, this situation could have been handled by not throwing an exception and handling the scenario with a simple if statement. That is a true statement in the scope of this example.

However, if this InvalidGPA exception was shared with multiple programs, it becomes more useful to define boundaries for GPA checking.

8.11 Working with files.

There are 3 basic things you can do with an external file:

1. Reading a file- This allows a program to extract/import data from a file into a program. For example, loading a list of Employees from a file.
2. Writing to a file- This allows a program to overwrite the contents of an existing file or create a brand-new file and add data to it. For example, a program could create a list of ingredients for a recipe and then output them to a file instead of standard user output.
3. Appending to a file- Unlike writing to a file, appending a file does not overwrite what currently exists in the file, it simply adds to whatever is already there. For example, assume a file already had a list of clients stored, a program could update the list with new clients as needed.

8.12 Types of files that can be used

Probably the most commonly used file is a plain text file that contains data that needs to be parsed and analyzed. However, not all files will be text files.

Other commonly used files are:

- Image files- These graphical files (.jpg, .gif, .png, .bmp) are often picture or image files that can be manipulated by a program. Examples of simple changes would be resizing the image, cropping the image, rotating the image, and changing color saturation. This can be performed with methods found in the javax.imageio.ImageIO class.
- Audio/Video files- These A/V files (.mp3, .mp4, .avi, .wav) contain audio, video or both audio and video data. For simple audio playback or manipulation, a Java program can use methods found in the javax.sound.sampled.AudioSystem or javax.sound.sampled.AudioInputStream classes. For video files, a Java file can either use classes such as javafx.scene.media.Media, javafx.scene.media.MediaPlayer, or a number of downloadable 3rd party classes.
- Database files (.db, .accdb, .sql, .dat) contain data stored in database applications. This is the most common tool used to store large volumes of data. Often times the Java program will need to incorporate the database language SQL to extract, manipulate, and save data to the database file. To connect to a database file is not as simple as the other file types as typically it requires the use of a JDBC (Java Database Connectivity) to establish a connection with the file. The type of JDBC varies depending on the database application being utilized.
- Spreadsheet files (.xlsx, .ods, .gsheet, .csv) contain data stored in spreadsheet applications like Microsoft Excel or Google Sheets. While with the correct 3rd party classes, a Java program can directly work with most of these file extensions, it is more common to create a .CSV file (Comma Separated Value) which takes the data stored in the spreadsheet file

and stores it as comma delimited plain text file. This makes simple tasks such as reading in the data or searching for a value easier to accomplish since the program does not need to be concerned with the format of the file. The `FileReader` and `BufferedReader` classes allow for data to be collected from a CSV file and the `FileWriter` and `BufferedWriter` classes allow data to be saved to a CSV file.

- Executable files (.jar). JAR files are executable files that Java can work with. The strong advantage of JAR files is that they are not platform specific like the .EXE (Windows) file.

The remainder of this chapter will focus on plain text files. It should be noted that plain text files are different than text enhanced files typically produced by word processing programs such as MS-Word, Google Docs, or Adobe Acrobat. These files not only contain the text that you see when viewing a document, but also a lot of embedded codes for things like font size, font type, margin settings, whether or not the text is bolded or italicized, line spacing and many other things found in word processing documents that format the text. To make things simpler to process, avoid using files with the following extensions: .doc, .docx, .gdoc, .pdf, .rtf, .pages, and .odt. Instead, use .txt and .csv files instead. Both .txt and .csv files contain just plain unformatted text along with end of line markers (EOL) and an end of file marker (EOF).

8.13 Invisible markers

Invisible markers, also known as invisible characters, appear in plain text files.

An end of line (EOL) marker, also known as a newline character, is not usually visible when viewing a document and is added to a document every time an author presses the Enter key to move the text down to the next line of the document. It quite literally represents where one-line ends. This importance of these EOL markers will be explored later. A document may contain 0 to many EOL markers per file.

The end of file (EOF) marker is also not usually visible inside a file. Unlike other markers, there is only one EOF marker per file. This marker is significant when reading data from a file as the file should stop trying to extract data once it reaches the EOF marker.

Other invisible markers include Tab markers and blank spaces.

8.14 Delimiters, tokens, and iterator objects

A **delimiter** is a separating value used to separate tokens. The default and most common delimiter used is a blank space. We use it in the written language to separate words. However, a blank space is not always the ideal delimiter when working with data in a computer file.

Take for example a person's name: Mary Beth Faylynn Jones. Is Mary her first name or is Mary Beth her first name? In situations like this it is often suggested to come up with another delimiter to avoid confusion inside the program. The most common delimiter used that isn't a blank space is the comma. You may have heard of CSV files before. The CSV stands for Comma Separated Values and they are commonly used to separate values with commas. Using the previous example: Mary Beth,Faylynn,Jones now makes it obvious to the program that Mary Beth is the person's first name, Faylynn is the middle name, and Jones is the last name.

A **token** is a single value that exists between two delimiters. Here are a few examples of valid tokens:

- A single word which is surrounded by two spaces.
- A line of text that is surrounded by two end-of-line(EOL) markers.
- A number which is surrounded by two commas. Note: be careful when using commas as delimiters with numbers. It is common for commas to be used in formatting numbers. For example: 1,234,567 may not be the three numbers 1 234 and 567 but rather a single number over 1 million.
- A single character. In this case there would be no delimiters, each character would be its own token.

A delimiter is not considered a token in most cases.

An **iterator object** is an object that has methods that allow a program to move through the contents of the object. These objects are created from classes that utilize the Iterator interface. One of the classes that uses the Iterator interface is the Scanner class. As you will see in Example 8.4, the Scanner class has 2 important methods that most iterator objects have: *hasNext()* and *next()*.

The *hasNext()* method is used to determine if there is another valid token. This method returns a boolean *true* until it the objects pointer reaches the end of file (EOF) marker. Once the EOF marker is reached, *hasNext()* will return a boolean *false*. Typically this method is used as a loop control value where the loop will continue to collect data until it reaches the end of the file.

The *next()* method collects the next available token inside the object and moves the pointer to the next token. The value is returned from the *next()* method as a String. The returned value can be stored in a String variable if needed. It is important to notice that not only does the *next()* method collect a token, it moves the pointer to the next token. A common “rookie” mistake is to overuse the *next()* method and advance further through the object than you expected to.

Example 8.4

This example reads text from a file one character at a time and then shifts each character to create a shift cipher, and then saves the results to a different file.

Some important things to note:

- The main method uses throws *IOException* in the method header. This is necessary as *IOException* is a checked exception and therefore it must be addressed with either a throws clause or a try..catch.
- The *useDelimiter* method is set to an empty string “”. This means that there is no delimiter. This allows each character to be read in one at a time.
- The file opens a file called *quote.txt*. This file needs to be visible to the program. In some IDEs this means the file needs to be included in the project or the project folder. For example, in Visual Studio Code, you may make this file visible by simply using the Open Folder menu option to make the folder and all its files visible to the running program.
- The output file must be closed using the *close()* method in order for the updated text to be saved to the file.
- The output file *scramble.txt* does not need to exist. If it doesn’t exist, it will be created. If it does exist, it will be overwritten.

- There are many different classes to read and/or write to a file with. This example uses both the Scanner and PrintWriter class. Other classes that can be used: FileReader/FileWriter and BufferedReader/BufferedWriter. It should be noted that the Scanner class is best for parsing a file as it allows a program to use delimiters.

quote.txt (this is a simple text file)

"Only those who dare to fail greatly can ever achieve greatly"- from Robert F. Kennedy

Example8_4.java

```
import java.util.Scanner;
import java.io.*;

/*
 *Filename:Example8_4.java
 *Description: This program shows how to read from a file and write to a file
 *Created by: Paul Murray
 *Date Created: 3/31/2025
 */
public class Example8_4
{
    public static void main(String[] args) throws IOException
    {
        Scanner userInput= new Scanner(System.in); //user input
        Scanner input = new Scanner(new File("quote.txt")); //opening this origin file

        input.useDelimiter(""); //delimiter set to empty string so that each character is its own token
        int shift_amount;

        System.out.print("How much do you want to shift by? ");
        shift_amount=userInput.nextInt();

        PrintWriter output = new PrintWriter(new File("scramble.txt")); //saving changes to this file
        String new_string="";
        char new_char;
        int new_int;
        System.out.print("\nThe original quote was: ");

        while(input.hasNext()) //continue until object reaches EOF mark
        {
            new_char=input.next().charAt(0); //extract character from String
            System.out.print(new_char); //output the current character to "standard output"
            new_int=new_char+shift_amount; //take unicode value of character and add shift number to it
            new_string=String.valueOf((char)new_int); //convert the integer back to a character and then..
                                                    //convert character back to a String
            output.print(new_string); //output the shifted character to the file
        } //end loop

        userInput.close();
        input.close();
        output.close(); //VERY IMPORTANT..you must close the file object to write to file
    }
}
```

```
} //end main method
} //end Example8_4 class
```

Output to standard I/O window:

The original quote was: "Only those who dare to fail greatly can ever achieve greatly"- from Robert F. Kennedy

Output to the file scramble.txt (assuming user chose to shift by 1 character):

```
*Pomz!uipft!xip!ebsf!up!gbjm!hsfbumz!dbo!fwfs!bdijfwf!hsfbumz#!gspn!Spcfsu!G!/Lfoofez
```

Note that even blank spaces are shifted. In this case that means that blank spaces become exclamation marks.

8.15 Using more than one delimiter

Typically, Java file classes only allow one delimiter at a time. This works in some cases, but in other cases, you may want to define multiple delimiters at one time. For example, what if you want to extract all the words from a file, would you use just a blank space to separate one word from another? What happens when there is only an EOL marker between two words. Would you include punctuation as a delimiter so that it wasn't considered part of the word?

Example 8.5

This example uses 4 delimiters when parsing the text from the input file: a blank space, an end of line (EOL) marker, a comma, and a dollar sign.

sales.txt (this is a simple text file)

```
89.99,9.99,129.99 17.99
24.74$4.99 18.84
123.99
```

Example8_5.java

```
import java.util.Scanner;
import java.io.*;

/*
 *Filename:Example8_5.java
 *Description: This program shows how to extract data from a file
 *             using multiple delimiters.
 *Created by: Paul Murray
 *Date Created: 7/8/2025
 */
public class Example8_5
{
    public static void main(String[] args)
    {
        int number_of_sales=0;
        double total_of_sales=0;
        String temp_token;

        try (FileReader fileReader = new FileReader("sales.txt");
            Scanner scan = new Scanner(fileReader))
        {
            // Set 4 delimiters. In this case a blank space, EOL marker,
            // comma and dollar sign are used as delimiters
            scan.useDelimiter("[ \\n,$]");
```

```

        while (scan.hasNext())
        {
            number_of_sales++; //count how many times through loop
            temp_token=scan.next(); //grab next token as a String
            System.out.println(temp_token); //Output String
            total_of_sales+=Double.valueOf(temp_token); //convert String to double to add to total
        } //end while loop
        System.out.println("\nThere were "+number_of_sales+" sales today with a total of
        $" +total_of_sales);
    }
    catch (IOException e)
    {
        System.out.println("An exception occurred when trying to open the file");
    }
} //end main method
} //end Example8_5 class

```

Input from file:

89.99,9.99,129.99 17.99
 24.74\$4.99 18.84
 123.99

Output:

89.99
 9.99
 129.99
 17.99
 24.74
 4.99
 18.84
 123.99

There were 8 sales today with a total of \$420.52

A few things to note:

- This program did not use the *throws IOException* clause in the class header like in previous examples. Instead, if an *IOException* was generated, the program would catch it and report a “friendly message”.
- If the input file contained something non-numeric, a *NumberFormatException* would be thrown and in this example, not caught.
- If more than one delimiter is used back-to-back, unexpected results will be given as the 2nd delimiters will be considered a valid token. For example, a blank space followed by an end-of-line (EOL) marker in this example would be two delimiters in a row.
- It is not necessary to close the scan object as the try-with-resources automatically does that.
- `useDelimiter()` uses square braces to list all the delimiters. Some things to note about the delimiters used in the example:
 - The space after the `[` is actually one of the delimiters. If that space was removed, a space wouldn’t be one of the delimiters

- Similar to the space, the comma is also one of delimiters. Often times commas are used to separate tokens inside a set of []. However, in this case it is just another delimiter in the list.
- The `\n` is the code needed to check for EOL markers. A `\t` could also be used to make a tab marker a delimiter.

8.16 Working with longer tokens

In the examples shown, the programs are typically parsing individual words or individual characters from a file. This is not always necessary. Sometimes you will want a token to be a complete line of text or an entire file.

To read an entire line until it reaches the EOL marker, a program can use a `readLine()` method found in the `BufferedReader` class or `Console` class. Also, you can use the `nextLine()` method found in the `Scanner` class.

To read an entire file until it reaches the EOF marker without using a loop, a program can use the `readString()` method of the `Files` class. This method allows the entire file to be placed inside of single `String` without using a loop. Another non-loop option is to set the delimiter of `Scanner` class to `\Z`, which is the delimiter for the end-of-file (EOF) marker and then use a single `next()` method to read the data in.

Chapter 9: Time Complexity Analysis of Algorithms

Big-O Notation:

It is a notation to measure, compare, and express time efficiency of different algorithms (or programs). We measure the time a program takes to do a task, as a function of the input size.

- For example if you pass an array of size n to a method, then the input size is n .
- For this topic terms program, algorithm or method are equivalent.
- The time needed to finish the task by an algorithm, is called time complexity of the algorithm.
- We want to measure the maximum time needed by an algorithm, because we want to calculate an upper bound and give a guarantee that the algorithm never run more than this maximum.

Let's go over a clear example to see how to measure time complexity of a method:

```
public int returnIndex(int value, int[] arr)
{
    int index = -1;
    for (int i=0; i<arr.length; i++)
        if (value == arr[i])
            index = i;
    return index;
}
```

What the method does?

Assumptions:

- A method might have several parameters, i.e. inputs, but for time complexity, the one matters is the one that its length could change a lot.
 - For example for returnIndex() method, which input matters? Int[] arr.
 - What is the input size? n .
- single java statements take a fixed time, let's assume that time a constant, like 1.
- We show time complexity of a method as a function of input length, like $T(n)$ or $f(n)$ or $g(n)$.

Time complexity of returnIndex() is shown by $T(n)$:

Time needed for "int index = -1;" is 1.

Time for the loop block:

```
for (int i=0; i<arr.length; i++)
    if (value == arr[i])
        index = i;
```

How many statements are run in each iteration (maximum)? 3.

How many iterations the loop go over? n .

Time needed for "return index;" is 1.

Total time is: $T(n) = 3n + 1 + 1$

We represent $T(n) = 3n+2$ with Big-O notation as $O(n)$.

To convert a function to Big-O notation:

1. We remove constant terms and constant coefficients.
For example if $f(n) = 3n+2$, then it is $O(n)$.
2. if after step 1, you left with a polynomial expression like $n^3 + n^2 + n$, keep the term with highest power.
For example if $f(n) = n^3 + n^2 + n$, then it is $O(n^3)$.

The reason we do this, is because Big-O notation is going to show an upper bound for time complexity; for example for $f(n) = n^3 + n^2 + n$, you can find a constant and coefficient like K , such that $f(n) = n^3 + n^2 + n$ is $< K \cdot n^3$;

Then we can say $K \cdot n^3$ is an upper bound for $f(n)$.

We remove K or any constant in Big-O notation, then we say $f(n)$ is of order of $O(n^3)$.

Compute the big-O of the following functions:

1. $g(n) = 7n^2 - 5n^3 + 10n^4$ _____
2. $g(n) = 29 + 9\log_2 n$ _____
3. $g(n) = 17n + 4n\log_3 n + 2^n$ _____

What is the time complexity of the following method in big-O notation?

```
public int method0 (int n)
{
    int steps = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            steps++;
        }
    }
    return steps;
}
```

Big-O of a function like $f(n)$ is going to give an UPPER bound for the function.

For a function $f(n)$ there might be many upper bounds, but we are interested in the closest one (the **tightest upper bound**);

- $f(n) = n^2 + n \log n$
- Upper bounds of $f(n)$ are n^2 , n^3 , $n^2 \log n$, n^3 , 2^n etc.
- But its tightest upper bound is ... n^2 ... and we say $f(n) = n^2 + n \log n$ is of order of $O(n^2)$.

If an algorithm always takes a fixed amount of time (or algorithm always run a fixed number of statements) then its running time is **constant** and we show it with $O(1)$.

- for any constant number c , $f(n) = c$ it is of order $O(1)$.
- Can you name a function with $O(1)$?

Computing time complexity of some functions could be difficult, but the good news is there are not many different types of expressions like X for $O(X)$.

(A) Some of most important ones are:

$O(1) < O(\log n) < \text{square_root}(n) < O(n) < O(n \log n) < O(n^k)$ “for fixed k and $k > 1$ ” $< O(2^n) < \dots$
 $\dots O(p^n)$ “for fixed p and $p > 2$ ” $< O(n!)$.

When you know this list, computing time complexity of an algorithm in big-O notation is easy! Because all you need is to guess which one of these cases match the best to the running time.

- For example, what is running time of a method which finds the maximum number in a 3D array with dimensions n by n by n ?
 - o Which of functions in section (A) is more likely to be the answer of this question?

Examples to practice calculating time complexity of functions

- The best way to learn almost any subject, is to first try to solve a problem **individually!** then check your answer with a resource or person.

So, work on the following examples individually and then check your answer with your classmates or me.

```
public int method1 (int n)
{
    int steps = 0;
    for (int i = 0; i < n; i++)
    {
        steps++;
    }
    return steps;
}

public int method2 (int n)
{
    int steps = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j =0; j < n*n; j++)
        {
            steps++;
        }
    }
    return steps;
}

public int method3 (int n)
{
    int steps = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j =0; j < i; j++)
        {
            steps++;
        }
    }
    return steps;
}

public int method4 (int n)
{
    int steps = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j =0; j < i*i; j++)
        {
            for (int k = 0; k < j; k++)
            {
                steps++;
            }
        }
    }
    return steps;
}
```

Some nonpolynomial functions

To see some examples of functions with big-O other than polynomial functions, i.e. other than $O(n^k)$ when k is an integer, consider the following examples:

1.

```
public static int method1(int n){
    while (n > 0){
        steps++;
        n = n/2;
    }

    return steps;
}
```

2.

```
public static int method2(int x){
    int steps = 0;
    int tmp = 1;
    int i = 0;
    while (tmp <= x) {
        steps++;
        i++;
        tmp = i * i;
    }
    return steps;
}
```

For these cases consider different values of n , for example 1, 5, 10, 20, 40, 100, 500, 1000 and see what the value of steps variable is when the function finishes.

Let's do it for method1:

n	Steps
1	1
5	3
10	4
20	5
40	6
100	7

$f(n)$ = Steps, can you guess what is $f()$ function? (you can look at the list of important functions for big-O in this note).

Hint for number 2: You can use the same approach, but the function is not in that list.

Finding time complexity, or Big-O, of a method via programming

To see this approach, let's try it on one of the examples from the previous note to see if we could reach to the same answer:

```
public static int method01 (int n)
{
    int steps = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < i; j++)
        {
            steps++;
        }
    }
    return steps;
}
```

- Create a Java project, you may call it big_O_proj.
- Create a class with main method, call it alg3_test.
- Type the above method inside the class.
- In main method write the following code:

```
for (int n=1; n<100; n++) {
    System.out.println(n + "\t" + alg3(n) + "\t" + n*n + "\t" + n*n*n + "\t" + n*n*n*n);
}
```

- Run your program, you will see columns of numbers. Check to see alg3 column is $\leq$ to which column, and that column is the closest to alg3 one.
 - o The you conclude Big-O of alg3 is

Let's try another example:

```
public static int method3(int x){
    int steps = 0;
    int tmp = 1;
    int i = 0;
    while (tmp <= x) {
        steps++;
        i++;
        tmp = i * i;
    }
    return steps;
}
```

- Create another class, call it method3_test.
- Type the method3 inside the class.
- In main method write the following code:

```
for (int n=1; n<100; n++) {
    System.out.println(n + "\t" + method3(n) + "\t" + Math.sqrt(n));
}
```

- Run your program. You can see the column for sqrt is very close to method3 one, but not bigger than method3 one (It is not an upper bound yet).

- In such cases we can add a constant coefficient or constant term to the upper bound column to make it bigger. So, change the for block to the following:

```
for (int n=1; n<100; n++) {
    System.out.println(n + "\t" + method3(n) + "\t" + 2*Math.sqrt(n));
}
```

- It is fine to do this, because based on the method in previous notes, you can get ride of constants when you write the time complexity in terms of Big-O notation.
- Run your program again.

Try this experimental approach for the following methods:

- For each of the following methods, repeat the above steps and find Big-O of each.

```
public static int method1(int n){
    while (n > 0){
        steps++;
        n = n/2;
    }

    return steps;
}

public static int method2 (int n){
    int steps = 0;
    while (n > 0){
        for (int k=0; k<n; k++)
            steps++;
        n = n/2;
    }

    return steps;
}
```

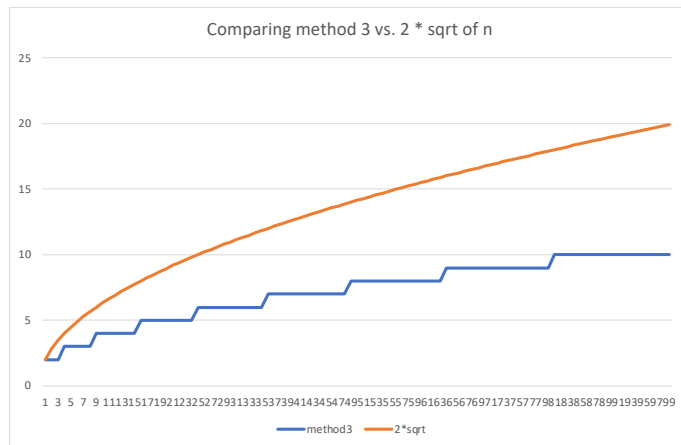
Time Complexity Lab

Determine time complexity of the following methods via experiment or coding. Please draw the method graph and the function you compare with via a software like Excel or any online tool you want to use.

Example:

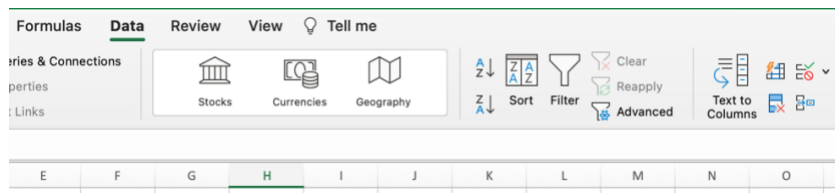
For method3 in the previous activity you need to draw the following graph:

- Method3's curve is blue.
- $2*\sqrt{n}$ line is orange.
- Since orange line is above blue line, you can conclude $2*\sqrt{n}$ function is an upper bound for method3.
- You get rid of coefficient 2 and write the time complexity of method3 as $O(\sqrt{n})$.



How to draw a time complexity graph in Excel?

- Copy and paste the output of your java program into an Excel sheet.
- When you paste the output into excel, you need to transform it into different columns, to draw their curves.
 - o To do so, first paste your output in a sheet, then from the menu go to **Data** and click **Text to Columns**. (See the next picture).



- After you see the output in separate columns, **select** those columns, go to **Insert**, **Recommended Charts**, then choose second option.

For the following methods, draw the graphs and find their time complexity in big O notation.

Method_A:

```
public static int method_A(int n) {
    int steps = 0;
```

```

        for (int i=0; i*i < n; i++) {
            steps ++;
        }

        return steps;
    }

```

Method_B:

```

public static int method_B(int n) {
    int steps = 0;

    for (int i=0; Math.sqrt(i) < n; i++) {
        steps ++;
    }

    return steps;
}

```

Method_C:

```

public static int method_C(int n){
    int steps = 0;
    int tmp_max = n;
    while (n > 0){
        for (int k=0; k<tmp_max; k++)
            steps ++;
        n = n/2;
    }

    return steps;
}

```

Method_D:

```

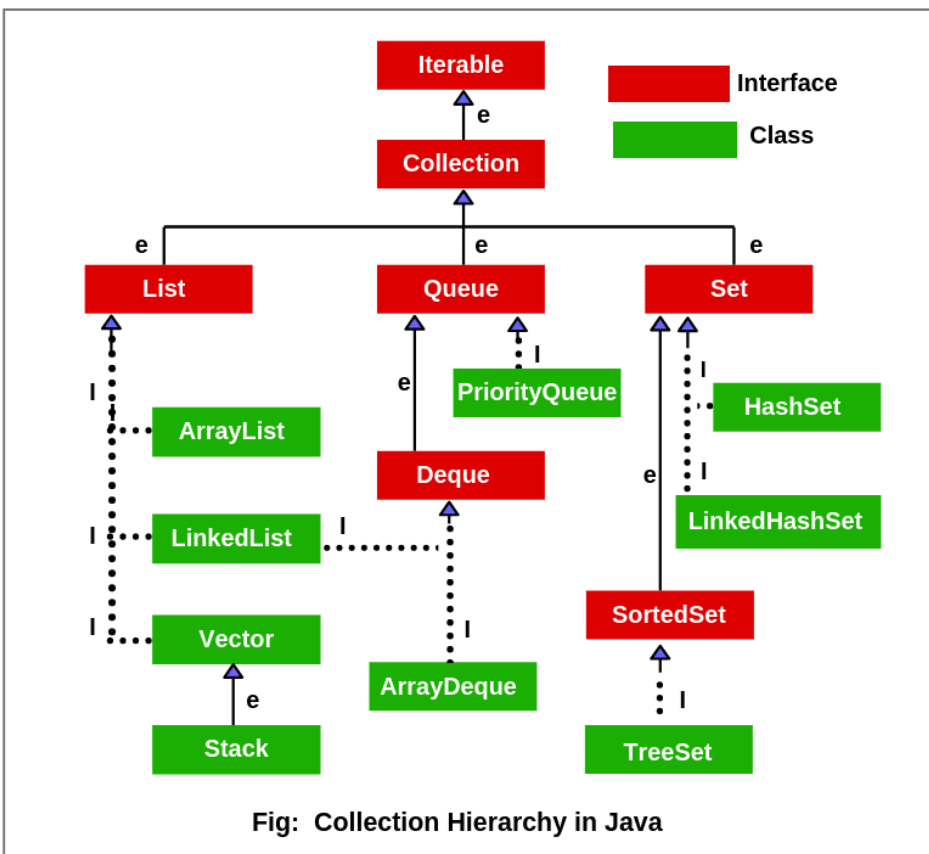
public static int method_D(int x){
    int steps = 0;
    int tmp = 1;
    int i = 0;
    while (tmp <= x) {
        steps++;
        i++;
        tmp = i / 2;
    }
    return steps;
}

```

Chapter 10: Introduction to Collections

In Java programming language a collection is a class that serves as a container of other objects. An object of a collection class through its methods makes it easier for a programmer to store, traverse and manipulate objects in the memory of a computer. An example of a collection in Java is ArrayList. By creating an object of ArrayList you can store unlimited objects, add or remove them, without dealing with the technical details of how those operations are done by Java language.

There are several collection classes in Java, each provides different pros and cons. These classes called Java Collection framework, and they form the following hierarchy of classes and interfaces¹. In this diagram 'e' stands for extends, for example List interface extends Collection interface, and 'I' stands for implements, for instance ArrayList implements List interface, and implements all methods declared in the List interface.



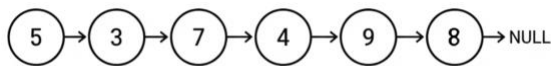
In Java an interface only provides a list of method declarations that classes implementing that interface, must include the actual code for the methods. Perhaps you noticed collections ArrayList, LinkedList and Vector have exact the same set of methods, such as add(), remove(), size(), insert(). The reason for this consistency is that these methods are declared in the interface List, and all the collections implementing List must have these methods.

¹ <https://www.scientecheasy.com/2020/09/collection-hierarchy-in-java.html/>

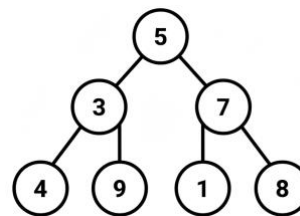
The other two interfaces in Collection hierarchy are Queue and Set; In a list, i.e. classes implementing List interface you can add and remove in the middle of the collection, but in a queue, we only can add to the end and remove from the front. In a List and a queue, you can have repeated values, but in a set, we have unique values. The collections (classes) implementing Queue and Set interfaces provide these distinct behaviors associated with queue and set containers.

Linear vs. Non-Linear collections

Collection classes under List interface are used to store values that there is a sequential (or linear) order between all element values. Examples of Linear collections are ArrayList, LinkedList and Vector. In the Collection hierarchy, there is another type of Collection class such as TreeSet which values are stored in a structure like a tree and there is not a linear ordering between all values.



Linear collection



Non-linear collection

Abstract Data Types (ADT): Lists, Stacks, and Queues

Collections have another name in Java, and they are called Abstract Data Types (ADT). They are called **Abstract** because to simply use them in your program you don't need to know how they implemented in Java. As an example, ArrayList is a collection that holds other objects and provides methods to access and manipulate objects. That is an ADT because you don't need to be concerned if the array inside the ArrayList get filled, what would happen. Theoretically you can add to an ArrayList without any limit.

Generic Classes and Generic Methods

You can define an ArrayList to hold any object type, because ArrayList was implemented as a generic class. Generic class is a class that could be used for any object data type.

Generic methods

Similar to the definition of a generic class, a generic method could work on any object type input data.

Let's see how to write a generic method:

- Let's go by an example. Write a method which gets an array of double values as an input and reverse its elements.
- Now replace *double* with a letter such as T (generic type T).
- To tell Java that T is a generic type, we need to put <T> before return type of the method. (it is called generic type parameter).
 - o Your method signature should look like this:


```
public static <T> void reverse (T[] data)
{
    // your code.
}
```
- Test your first generic method:
 - o Create a simple Java program and add your generic method to it.
 - o Inside your main method create an array of Integer objects (with integer values), and an array of Double objects.
 - o Call your generic method for each of arrays and print the array after each function call.
 - o You can see that your generic method works fine independent of input object types.

Creating a generic class

We add generic type parameter T as <T> after the class name, to tell Java compiler that T is a type that will be passed when an object of the generic class is initiated.

Normal class: Node.java	Generic class: NodeGeneric.java
<pre> public class Node{ private String data; private Node link; public Node(String data, Node link){ this.data = data; this.link = link; } // the rest of methods. } </pre>	<pre> public class NodeGeneric<T>{ private T data; private NodeGeneric<T> link; public NodeGeneric (T data, NodeGeneric<T> link){ this.data = data; this.link = link; } // the rest of methods. } </pre>

How to create objects from a generic class?

Example:

```

NodeGeneric<String> head1 = new NodeGeneric<String>("One", null);
NodeGeneric<Double> head2 = new NodeGeneric<Double>(new Double(1), null);

```

There are some limitations with generic types in Java:

- A. You can not create a generic type object!
 - a. `T obj = new T();` // Error!
- B. You can not directly create an array of generic type with the following syntax:
 - o `T[] arr1 = new T[100];` // Error!

To solve the issue for the case A you can pass object of type T into the generic class instead of creating the object.

For the case B, the work around is to create an array of Object type, and that array can hold any type of objects, because Object is the most generic type in Java object hierarchy and all Java objects inherit from Object class directly or indirectly.

Solution for the case B:

```
T[] arr1 = (T[]) (new Object[100]);
```

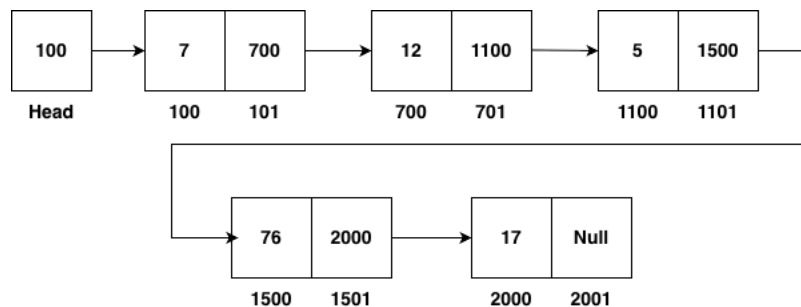
Chapter 11: LinkedList and Stack

First, we need to understand the differences between arrays and linked lists at a conceptual level before we look at the implementation details of linked lists.

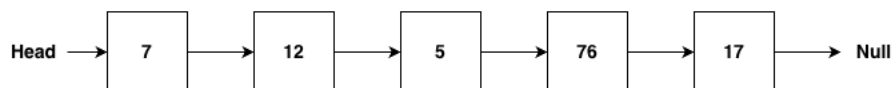
Arrays are stored in consecutive cells in RAM. When you create an array you have to specify a type and a count of the number of elements. Java knows how much space one element of that type takes, so it multiplies that size times the number of elements you want, and allocates a block of memory of that total size for the array. **Random access** is possible because array elements are stored consecutively. A **program can go directly to any element in an array by using the element's index**. Java knows where the array begins in RAM. When you want to access element i of a single-dimension array, Java multiplies i by the size of one element and adds that amount to the address of element 0.

Random access is the primary benefit of arrays. One drawback is that the size of an array is fixed. Another drawback is that it is expensive to insert a new element into an array or to delete an element – all elements from that point forward have to be shifted.

Linked lists are composed of structures that we call **nodes**. In a singly-linked list, each node contains both **data** and the address in RAM of the next node in the structure. These addresses are what we call **links**. The last node in the list has a null link to mark it as the end of the list.



Typically, we simplify the picture of a linked list by removing the addresses and leaving arrows to indicate the links. We use a "ground" symbol to indicate null.



The PersonNode Class

We will first examine a PersonNode, a node that stores a String for its data. Our node has two fields. One is for the data portion, and the other is a link (reference) to the PersonNode that is next in the linked list. The class has various constructors, getters, and setters as shown here:

```
public class PersonNode {
    private String data;
    private PersonNode link;
```

```
// Default Constructor
```

```
public PersonNode() {
    data = null;
    link = null;
}

// Constructor with data
public PersonNode(String data) {
    this.data = data;
    link = null;
}

// Constructor with data and a link
public PersonNode(String data, PersonNode link) {
    this.data = data;
    this.link = link;
}

// Setter for the link
public void setLink(PersonNode newLink) {
    link = newLink;
}

// Setter for the data
public void setData(String newData) {
    data = newData;
}

// Getter for the link
public PersonNode getLink() {
    return link;
}

// Getter for the data
public String getData() {
    return data;
}
}
```

Now consider the following code:

```

public static void main(String[] args)
{
    PersonNode person = new PersonNode("Alex");
    PersonNode person2 = new PersonNode("Bethany", person);
    PersonNode person3 = new PersonNode("George", person2);
    PersonNode person4 = person3;
    while (person4 != null)
    {
        System.out.println(person4.getData());
        person4 = person4.getLink();
    }
}

```

- Try to figure out the output of code by drawing the picture of LinkedList.

Activity:

- Create a java project called linked_list.
- Create PersonNode class and fill it with the code given above.
- Create another class called test_of_PersonNode and type the above code in it.
- Run it and check the input.
- Then add statements to the code to make a new PersonNode with "Sam", and insert it between "George" and "Bethany".
- Run it again and see the changes.

Traversing through a LinkedList:

There are two common ways to traverse through a LinkedList:

- 'head' in the following loops is a reference of type Node, pointing to the first node in the list, like 'person' in the previous example.

- **while** loop:

```

Node traverse = head;
while (traverse != null)
{
    System.out.println (traverse.getData()); // print the actual data portion
    traverse = traverse.getLink();
}

```

- **for** loop:

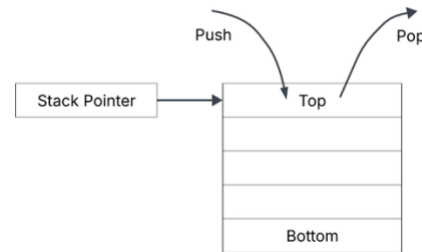
```

for (Node traverse = head; traverse != null; traverse = traverse.getLink())
{
    System.out.println (traverse.getData());
}

```

Java Container: Stack

A stack is a container of ordered items such that items can be inserted and removed only at one end (called the **top**). Stacks are often given the synonym LIFO (last in, first out).



How to use Stack class of Java?

- It is inside java.util package, so we import java.util.Stack;
- to put an object into a Stack object, we call push() method.
- to remove and get the object on top of stack, we call pop() method.
- to check to see if there is any object in the stack, we use isEmpty() method.

Activity 1:

Let's have a hands-on experience with Java Stack class and its methods:

- Write a simple java program and add these statements in the main method:

```
Stack<Integer> numbers_stack = new Stack<Integer>();
    for (int i=0; i<10; i++){
        numbers_stack.push(Integer.valueOf(i));
    }
    while (!numbers_stack.isEmpty()){
        System.out.println(numbers_stack.pop());
    }
```

- Run it and see how the methods work.

Some practices with stacks:

List the elements in the order in which they are popped from the stack, given this series of stack operations: push(5), push(3), pop(), push(2), push(8), pop(), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop(), pop().

Consider a series of pushes and pops in which you pushed the integers 1, 2, 3 (in that order). Could you have popped them off in the order 1,2,3? What about 3,2,1?

Suppose a stack has the values {3, 4, 8, 12, 15, 12, 3, 4, 5, 4}, in that order, where 4 is at the top of the stack. Trace the following method on that stack, and tell what the stack would look like when the method ends. Hint: statement **list.contains(temp)** checks if temp object is inside list or not. It returns true if yes, otherwise false.

```
public static void what_it_does(Stack<Integer> stack)
{
    ArrayList<Integer> list = new ArrayList<Integer>();
    Integer temp;
    while (!stack.isEmpty())
    {
        temp = stack.pop();
        if (!(list.contains(temp)))
        {
            list.add(temp);
        }
    }
    for (int i = list.size() - 1; i >= 0; i--)
    {
        stack.push(list.get(i));
    }
}
```

Activity 2:

- Create a java program and add the above method to the block of the java class.
- In main method create a stack and push these numbers in order: 3, 4, 8, 12, 15, 12, 3, 4, 5, 4.
- Print your stack, you can check Activity 1 for this part.
- In main method pass your stack to **what_it_does()** method.
- Print your stack again and see the difference.

Suppose a stack, myStack, contained the values {5, 3, 1, 2, 4}, with 4 at the top. What would be the result of the statement, myStack = guess_me(myStack), given the following method?

```
public static Stack<Integer> guess_me (Stack<Integer> stack)
{
    Stack<Integer> newStack = new Stack<Integer>();
    while (!stack.isEmpty())
    {
        newStack.push(stack.pop());
    }
    return newStack;
}
```

Activity 3:

- Create a java program and add the above method to the block of the java class.

- In main method create a stack call it myStack and push these numbers in order: 5, 3, 1, 2, 4.
- Print your stack, you can check Activity 1 for this part.
- In main method add this statement `myStack = guess_me (myStack)`.
- Print your stack again and see the difference.

Activity 4:

- Write a program that gets a string from input and print if the input is a palindrome string or not.

A palindrome string is like ABBCBBA that if you read from left to right or right to left, it will be the same.

- o To practice working with stacks, use a stack to write this activity.

Chapter 12: Introduction to Recursion

In this chapter, we will cover the following topics in the order listed below:

- Definition of Recursion
- Essential parts of a recursive solution
- Solving problems with recursion
- Calculating time complexity of recursive methods
- Computational overhead of recursive solutions

Definition of Recursion

An algorithm is recursive if it calls itself for the portion of its input. This simple approach could provide an easier solution to many classical computer science problems.

Example:

$$4! = 4 * 3 * 2 * 1 = 24$$

$$5! = 5 * 4 * 3 * 2 * 1 = 5 * 4! = 5 * 24$$

$$\text{In general } n! = n * (n-1)! \quad \text{for } n > 0, \text{ and } 0! = 1$$

We can implement a Java function to calculate factorial by using the recursive formula as below:

```
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

Essential parts of a recursive solution

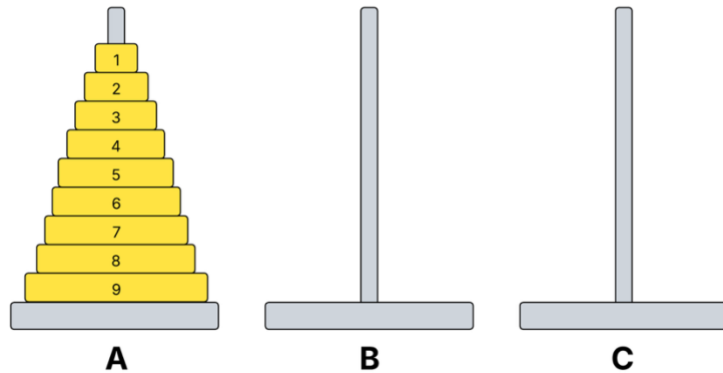
A recursive method needs to have two main parts:

1. A recursive function call
2. A base case

Solving problems with recursion

Hanoi Tower problem

The tower of Hanoi is a famous puzzle with a recursive solution. The puzzle consists of three poles and a collection of rings that stack on the poles. The rings are different sizes and are stacked in decreasing order of their size on the first pole. The second and third poles are empty. The goal is to move the rings from the first pole to the second pole, one at a time, under the restriction that you cannot put a larger ring on top of a smaller one. You can use the third pole to hold rings temporarily.



Let's find the solution for some simple cases first:

- When the number of rings in pole A is 1, i.e. $n=1$:
 - move the ring from A to C (one move).
- When $n=2$:
 - Move the ring #1 from A to B
 - Move the ring #2 from A to C
 - Move the ring #1 from B to C.

Total moves: 3

- When $n=3$: Since solving this case could be more complex than $n=1$, or $n=2$, let's think in a recursive way:
 - I. Move rings 1 and 2 from A to B with the process we showed for $n=2$.
 - II. Move ring 3 from A to C.
 - III. Move rings 1 and 2 from B to C with the process we showed for $n=2$.

Total moves = Number of moves for cases I., II., and III. = $3 + 1 + 3 = 7$.

Recursive solution for Hanoi Tower problem:

Based on the recursive approach we show for $n=3$, we can design the following function that solves Hanoi tower problem for any value of n :

```
public void hanoi(int numberOfRings, String start, String end, String temp)
{
    if (numberOfRings == 1)
        System.out.println ("From " + start + " to " + end);
    else
    {
        hanoi(numberOfRings-1, start, temp, end);
        System.out.println ("From " + start + " to " + end);
        hanoi(numberOfRings-1, temp, end, start);
    }
}
```

Practice: Trace the function behavior on the calls `hanoi(2, "A", "B", "C")` and `hanoi(3, "A", "B", "C")`. Students are encouraged to implement a Java program that utilize the above function, to test their answers for this practice case.

Time Complexity of Hanoi Tower solution

An interesting question for Hanoi's problem is this that how many moves we should take to transfer n rings from A to C. To solve this problem, in the following table, we list the number of moves for a range of n values, and check if there is any pattern for the number of moves.

Number of Rings	Number of Moves
1	1
2	3
3	$2T(2) + 1 = 7$
4	$2T(3) + 1 = 15$
n	$2T(n-1) + 1$

We show the number of moves for n by the function $T(n)$. $T(n)$ per the last row of the table is equal 2 function calls for $n-1$, plus 1. $T(n) = 2T(n-1) + 1$.

Solving $T(n)$:

$$T(n) = 2T(n-1) + 1$$

$$T(n) = 2(2T(n-2) + 1) + 1 = 2^2 T(n-2) + 2 + 1$$

$$T(n) = 2^2(2T(n-3) + 1) + 2 + 1 = 2^3 T(n-3) + 2^2 + 2 + 1$$

$$T(n) = 2^k T(n-k) + 2^{(k-1)} + 2^{(k-2)} + \dots + 2 + 1$$

$$\Rightarrow K = n-1$$

$$T(n) = 2^{(n-1)} + 2^{(n-2)} + \dots + 2 + 1 = 2^{(n)} - 1$$

$$T(n) = O(2^n)$$

Computational overhead of recursive solutions

To understand the computational cost of running a recursive method, let's try to trace the following method and see what it prints for $k = 10$:

```
public static void methodx (int k)
{
    //if (k == 0)
    //    return;

    methodz(k-1);    // recursion line
    System.out.println(k);
}
```

When a CPU runs a method instructions line by line, if it reaches to a function call, the CPU puts the state of the current method (local variables and the current line number) on the top of Call stack. The CPU then switches to run the function. The same happens when CPU runs a recursive method; in each recursive call it puts current state of the method on the top of Call stack. This could cause stack overflow if the number of recursion calls goes beyond the Call stack limit.

Possibility of repetition in solving a subproblem

Consider a recursive solution for Fibonacci:

```
public static int Fib (int n)
{
    if (n == 1 || n==2)
        return 1;

    return Fib(n-1) + Fib(n-2);
}
```

More recursive examples:

Develop an algorithm that gets an array of positive integers and a sum value, it returns true if there is a subset of numbers in the array that their summation is equal to the sum value.

For example, for inputs array = [3, 10, 5, 4, 7], and sum=12, your algorithm returns true, but for the same array value and sum=6, your algorithm returns false!

Let's think loudly, to see how we could build the recursion formula.

What are base cases, i.e., the situations that you could quickly return true or false?

Assume array = [7] and sum=7, you returnTrue.....

How about array = [7] and sum = 5, you return ...False.....

What if sum = 0, you return ...True.....

What if array = [3, 4] and sum=7? How can you convert the problem to a sub problem that the sub problem is a base case?

Choose 3 => array=[4], sum = 4

What if for array = [10, 3, 4] and sum=7? How do you convert the problem to sub problem(s)?

Now you got the idea of how to develop the recursive algorithm. Write your complete algorithm, implement it either with Java or Python, and test it with some arbitrary inputs.